

AuD - Zusammenfassung

Moritz Gerhardt

1 Inhaltsverzeichnis

1	Inhaltsverzeichnis	
2	Was ist ein Algorithmus?	
3	Laufzeitanalyse	
3.1	O Notation	1
3.2	Ω Notation	2
3.3	Θ Notation	3
3.4	Rekursionsgleichungen	3
4	Sortieren	
4.1	Sortierproblem	5
4.2	Insertion Sort	6
4.3	Bubble Sort	7
4.4	Merge Sort	8
4.5	Quicksort	9
4.6	Radix Sort	9
5	Grundlegende Datenstrukturen	
5.1	Stacks	10
5.2	Queues	11
5.3	Linked List	12
5.4	Binary Search Tree	13
6	Fortgeschrittene Datenstrukturen	
6.1	Red-Black Tree	14
6.2	AVL Trees	15
6.3	Splay Trees	16
6.4	Binary Heap Trees	17
6.5	B-Tree	18
7	Probabilistische Datenstrukturen	
7.1	Deterministisch und Probabilistisch	19
7.2	Skip-Lists	20
7.3	Hash Tables	21
7.4	Bloom Filter	22
8	Graphen Algorithmen	
8.1	Graphen	23
8.2	Breadth-First Search (BFS)	24
9	Advanced Design	
9.1	Divide & Conquer	25
9.2	Backtracking	26
9.3	Dynamic Programming	27
9.4	Greedy Algorithms	28
9.5	Metaheuristiken	29
10	NP	
10.1	Leichte Probleme	30
10.2	Berechnungs- & Entscheidungsprobleme	31
10.3	Komplexitätsklasse P	32
10.4	Komplexitätsklasse NP	33
10.5	NP-Vollständigkeit	34

2 Was ist ein Algorithmus?

Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe. Dabei sollte ein Algorithmus im allgemeinen folgende Voraussetzungen erfüllen:

1. Bestimmt:

- Determiniert: Bei gleicher Eingabe liefert der Algorithmus gleiche Ausgabe.
 \Rightarrow Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
- Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
 \Rightarrow Gleiche Schritte, Gleiche Zwischenstände.

2. Berechenbar:

- Finit: Der Algorithmus ist als endlich definiert. (Theoretisch)
- Terminierbar: Der Algorithmus stoppt in endlicher Zeit. (Praktisch)
- Effektiv: Der Algorithmus ist auf Maschine ausführbar.

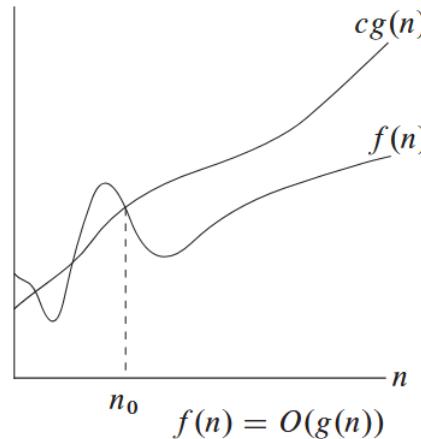
3. Anwendbar:

- Allgemein: Der Algorithmus ist für alle Eingaben einer Klasse anwendbar, nicht nur für speziellen Fall.
- Korrekt: Wenn der Algorithmus ohne Fehler terminiert, ist die Ausgabe korrekt.

3 Laufzeitanalyse

3.1 O Notation

Die O-Notation wird grundsätzlich für *Worst-Case* Laufzeiten verwendet. Sie gibt also eine obere Schranke an, die der Algorithmus im schlechtesten Fall erreicht. Dabei wird oft zwischen Big O-Notation und Little o-Notation unterschieden. Ein Graph zur Repräsentation der O-Notation ist hier zu sehen:



3.1 (a) Big-O Notation

Mathematische Definition:

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

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $f(n) \geq 0$ und $f(n) \leq c \cdot g(n)$. Das bedeutet, dass die Funktion $f(n)$ für $n \rightarrow \infty$ den gleichen Wachstumsfaktor hat wie die Funktion $g(n)$. Einfache Berechnung findet wie folgt statt (anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = O(n^2)$

Dies kann man dann im Rückschluss so anwenden: Um die Konstanten c und n_0 zu finden, wird die obige Gleichung benutzt:

1. Simplifizierte die Ungleichung $5n^2 + 2n \leq c \cdot n^2$ zu $5 + \frac{2}{n} \leq c$
2. Da $n \geq n_0$ kann man die Gleichung für $n \geq 1$ auflösen um die Konstanten c und n_0 zu finden.
 $\Rightarrow 5 + \frac{2}{1} = 7 \leq c \Rightarrow c \geq 7$
3. Dementsprechend kann man dann die Konstanten $c = 7$ und $n_0 = 1$ auswählen.

3.1 (b) Little-o Notation

Mathematische Notation:

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

Es existieren die positive Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $f(n) \geq 0$ und $f(n) < c \cdot g(n)$. Little-o Notation unterscheidet sich also von Big-O Notation nur über der Schranke. Während bei Big-O der Wachstumsfaktor beider Funktion gleich sein kann ($f(n) = c \cdot g(n)$), gilt bei Little-o, dass der Wachstumsfaktor der Funktion $f(n)$ kleiner ist als der Wachstumsfaktor der Funktion $g(n)$.

Einfache Berechnung findet analog zu Big-O wie folgt statt (anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = o(n^2)$

Hier muss allerdings noch geprüft werden, ob der Wachstumsfaktor der Funktion $f(n)$ kleiner ist als der Wachstumsfaktor der Funktion $g(n)$. Wenn ja, ist die Little-o Notation korrekt für $g(n)$.

Um zu zeigen, dass $f(n) = o(g(n))$:

1. Finde den Limes des simplifizierten Ausdrucks $\frac{f(n)}{g(n)}$, der die Wachstumsrate der Funktion $f(n)$ zur Wachstumsrate der Funktion $g(n)$ vergleicht.
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{5n^2 + 2n}{n^2} = \lim_{n \rightarrow \infty} 5 + \frac{2}{n} = 5$
 $\Leftarrow \frac{2}{n} \text{ für } n \rightarrow \infty = 0$
2. Da der Limes $\neq 0$ ist, bedeutet das, dass der Wachstum von $f(n)$ nicht geringer ist als der von $g(n)$. Deshalb müssen wir ein Polynomgrad hochgehen, weswegen $f(n) = o(n^3)$ sein muss.
3. Um nun die Konstanten c und n_0 zu finden müssen wir einfach $\frac{f(n)}{g(n)}$ auflösen
 - $\frac{5n^2 + 2n}{n^3} < c$
 - $\frac{5}{n} + \frac{2}{n^2} < c$
 - $\frac{5}{n} < c$, da $\frac{2}{n^2}$ für $n \rightarrow \infty$ schneller abfällt als $\frac{5}{n}$
 - Für $c = 1$ muss dann $n_0 > 5$ sein und kann somit als $n_0 = 6$ gewählt werden.

3.1 (c) Rechenregeln

Sind sowohl für *Big – O* als auch *Little – o* gültig

- **Konstanten:**

$$f(n) = a \text{ mit } a \in \mathbb{R}_{>0} \implies f(n) = O(1)$$

Ist die Funktion konstant, so ist die Komplexität $O(1)$.

- **Skalare Multiplikation:**

$$f(n) = O(g(n)) \implies a \cdot f(n) = O(g(n))$$

Multiplikation der Funktion ändert die Komplexität nicht.

- **Addition:**

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \implies f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$$

Die Komplexität der Summe zweier Funktionen ist der Maximalwert der Komplexität der beiden Funktionen.

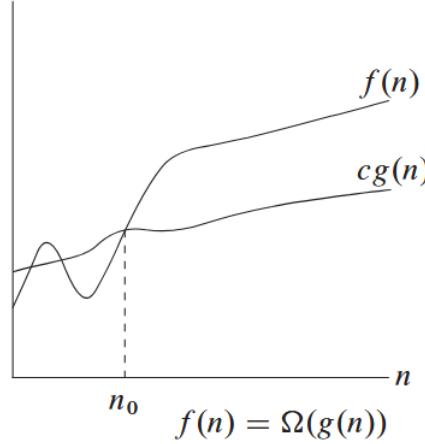
- **Multiplikation:**

$$f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)) \implies f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

Die Komplexität des Produkts zweier Funktionen ist das Produkt der Komplexität der beiden Funktionen.

3.2 Ω Notation

Ähnlich zur O Notation, allerdings geht es hier um den *Best-Case* also minimale Anzahl der Schritte, die ein Algorithmus ausführt.



Wird auch wieder in Ω und ω aufgeteilt, die sich nur darin unterscheiden, wie strikt die Grenze ist.

3.2 (a) Ω Notation

Mathematische Definition:

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

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $0 \leq c \cdot g(n) \leq f(n)$. Das bedeutet, dass der Wachstumsfaktor von $f(n) \geq c \cdot g(n)$ ist.

Die Berechnung von Ω ist leider nicht immer so simpel wie die Berechnung von O Notation. Nehme zum Beispiel einen Linearen Suchalgorithmus, der eine Liste so lange durchläuft, bis er die gesuchte Zahl gefunden hat. Die Komplexität ist $O(n)$, da, wenn das Element an letzter Stelle steht alle Eingaben durchlaufen werden müssen. Gleichermassen kann es aber sein, dass das Element an erster Stelle steht, was dann die Komplexität $\Omega(1)$ besitzt. Dies muss allerdings durch Analyse des Algorithmus selbst erkannt werden und kann nicht aus der Funktionsrepräsentation ermittelt werden.

Gilt allerdings nicht sowas, wie vorzeitiger Abbruch bei Suche, so kann Ω ähnlich zu O verwendet werden (Anhand vom Beispiel $f(n) = 5n^2 + 2n$):

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = \Omega(n^2)$
Da $5n^2 + 2n$ für $n \rightarrow \infty$ mindestens so schnell wächst wie n^2 .

Um Werte für c und n_0 zu finden, kann das Prinzip wie in O Notation verwendet werden, jedoch auf der Definition von Ω angepasst (Umgekehrtes Gleichheitssymbol).

3.2 (b) ω Notation

Mathematische Definition:

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

Es existieren die positiven Konstanten c und n_0 , sodass für alle $n \geq n_0$ gilt, dass $0 \leq c \cdot g(n) < f(n)$. Das bedeutet, dass der Wachstumsfaktor von $f(n) > c \cdot g(n)$ ist.

Für die Bestimmung von ω gilt das selbe wie für Ω , nur das zusätzlich noch folgendes beachtet werden muss:

- Hat der Algorithmus einen konstanten Best-Case, so ist ω nicht anwendbar, da $\omega < 1$ sinnlos ist, da per Definition die Komplexität nicht kleiner als 1 sein kann und so der Best-Case schon durch Ω definiert ist.
- Falls nicht konstant, dann muss bei ω ähnlich zu Little-o herausgefunden werden, ob der Wachstumsfaktor von $f(n)$ strikt größer ist als der Wachstumsfaktor der Funktion $g(n)$.
 - $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$
 - Wenn $\lim = \infty$, so gilt $\omega(g(n))$
 - Andernfalls muss der Polynomgrad von $g(n)$ verringert werden:
 $\rightarrow n^x = n^{x-1} \implies n = 1$

3.2 (c) Rechenregeln

Sind sowohl für $\text{Big} - \Omega$ als auch $\text{Little} - \omega$ gültig

- **Konstanten:**

$$f(n) = a \text{ mit } a \in \mathbb{R}_{>0} \implies f(n) = \Omega(1)$$

Ist die Funktion konstant und positiv, so ist die Komplexität $\Omega(1)$.

- **Skalare Multiplikation:**

$$f(n) = \Omega(g(n)) \implies a \cdot f(n) = \Omega(g(n)) \text{ für } a > 0$$

Eine positive skalare Multiplikation der Funktion ändert die Komplexität nicht.

- **Addition:**

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \implies f_1(n) + f_2(n) = \Omega(\min\{g_1(n), g_2(n)\})$$

Die Komplexität der Summe zweier Funktionen ist der Minimalwert der Komplexität der beiden Funktionen.

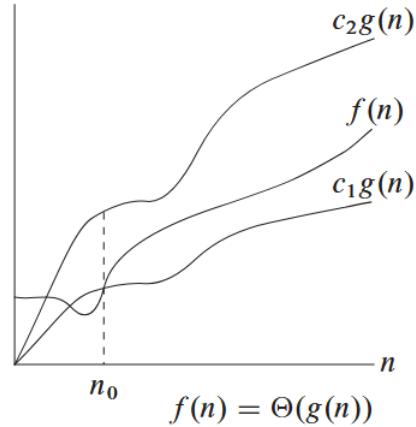
- **Multiplikation:**

$$f_1(n) = \Omega(g_1(n)), f_2(n) = \Omega(g_2(n)) \implies f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

Die Komplexität des Produkts zweier Funktionen ist das Produkt der Komplexität der beiden Funktionen.

3.3 Θ Notation

Θ Notation kombiniert O und Ω Notation. Das heißt sie stellt Durchschnittswachstum (Average-Case) einer Funktion dar und liegt somit zwischen O und Ω .



Mathematische Notation:

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

Es existieren die positiven Konstanten c_1, c_2 und n_0 , sodass für alle $n \geq n_0$ gilt, dass $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.
 $f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

Die Berechnung von Θ läuft dementsprechend auch ähnlich zu O und Ω ab (Anhand vom Beispiel $f(n) = 5n^2 + 2n$).

1. Finde den Term mit dem höchsten Wachstumsfaktor ($5n^2$)
2. Konstanten werden weggelassen (n^2)
3. Demnach ist $f(n) = \Theta(n^2)$
 Da $5n^2 + 2n$ für $n \rightarrow \infty$ mindestens so schnell wächst wie n^2 .

Die Berechnung der Konstanten ist allerdings ein klein wenig komplizierter, da es eine mehr gibt. Prinzipiell bleibt es aber gleich:

- Simplifizierte die Gleichung: $c_1 \cdot n^2 \leq 5n^2 + 2n \leq c_2 \cdot n^2 = c_1 \leq 5 + \frac{2}{n} \leq c_2$
- Da hier für alle $n > 0$ der mittlere Term positiv ist, kann man $n_0 = 1$ wählen.
- Dadurch erhalten wir $c_1 \leq 5 + \frac{2}{1} = 7 \leq c_2$, wodurch man hier die Konstanten dann z.B. $c_1 = 7$ und $c_2 = 7$ für $n_0 = 1$ auswählen kann.

3.4 Rekursionsgleichungen

Im Allgemeinen wird $T(n)$ als die maximale Anzahl von Schritten für Eingaben der Größe n angenommen.
So gilt als allgemeine Form für Rekursionsgleichungen:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

mit $a \geq 1$ und $b > 1$ und $f(n)$ asymptotisch positiv.

Dies ist so anzunehmen, dass das Problem in a Teilprobleme der Größe $\frac{n}{b}$ aufgeteilt wird.

Demnach benötigt das Lösen jedes Teilproblems immer $T\left(\frac{n}{b}\right)$ Zeit

Die Funktion $f(n)$ umfasst hierbei dann die Kosten der anderen Operation wie Aufteilen und Zusammenfügen.

3.4 (a) Mastertheorem

Das Mastertheorem bietet eine einfache Weise an die Laufzeit von verschiedene Rekursionsgleichungen abzuschätzen.
Dabei wird die oben genannte Form vorrausgesetzt.

So gibt es im Mastertheorem grundsätzlich drei verschiedene Fälle:

1. $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$

Wenn $f(n)$ polynomiell kleiner ist als $n^{\log_b a}$

- Hier ist die Rekursion wichtiger als die sonstigen Operationen.
- Demnach: $T(n) = \Theta(n^{\log_b a})$

2. $f(n) = \Theta(n^{\log_b a})$

Wenn $f(n)$ und $n^{\log_b a}$ gleiche Größenordnung haben

- Hier tragen Rekursion und sonstige Operationen die selbe Signifikanz
- Demnach: $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

3. $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$

Wenn $f(n)$ polynomiell größer ist als $n^{\log_b a}$

- Hier sind die anderen Operationen dominanter.
- (Unter der Bedingung: $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ für ein $c < 1$)
(Anders: Der rekursive Teil der Funktion $a \cdot f\left(\frac{n}{b}\right)$ ist proportional, aber kleiner als der nicht-rekursive Teil $f(n)$. Skalar von $c < 1$ impliziert ähnlichen, aber kleineren Wachstum für $n \rightarrow \infty$)
- Demnach: $T(n) = \Theta(f(n))$

4 Sortieren

4.1 Sortierproblem

Sortieralgorithmen sind die wohl am häufigsten verwendeten Algorithmen. Hierbei wird als Eingabe eine Folge von Objekten gegeben, die nach einer bestimmten Eigenschaft sortiert werden. Der Algorithmus soll die Eingabe in der richtigen Reihenfolge (nach einer bestimmten Eigenschaft) zur Ausgabe umwandeln. Es wird hierbei meist von einer total geordneten Menge ausgegangen. (Alle Elemente sind miteinander vergleichbar).

Eine Totale Ordnung wird wie folgt definiert:

Eine Relation \leq auf M ist eine totale Ordnung, wenn:

- Reflexiv: $\forall x \in M : x \leq x$
(x steht in Relation zu x)
- Transitiv: $\forall x, y, z \in M : x \leq y \wedge y \leq z \implies x \leq z$
(Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \wedge y \leq x \implies x = y$
(Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass $x = y$)
- Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
(Alle Elemente müssen in einer Relation zueinander stehen)

4.2 Insertion Sort

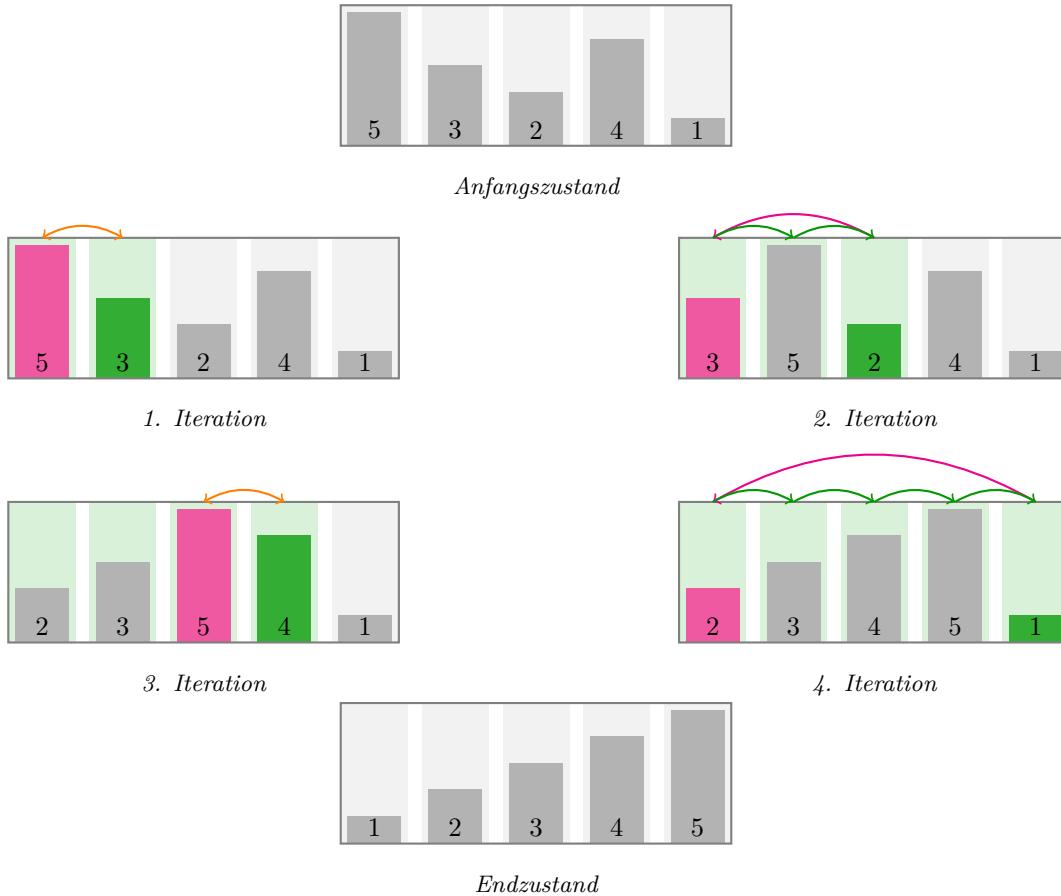
```

1 class InsertionSort {
2     void insertionSort(int[] arr) {
3         for (int i = 1; i < arr.length; i++) {
4             // 1 to n - 1
5             int key = arr[i];
6             int j = i - 1;
7             while (j >= 0 && arr[j] > key) {
8                 // Loops backwards through the array starting at i - 1
9                 // until it finds an element that is greater than the key or the beginning of the array
10                arr[j + 1] = arr[j];
11                // Shifts the element to the right
12                j--;
13            }
14            arr[j + 1] = key;
15            // Assigns the key to the correct position
16        }
17    }
18 }
```

4.2 (a) Vorgehensweise

Die Eingabe wird von links nach rechts durchlaufen startend bei $i = 1$. Das Element i wird dann mit allen Elementen verglichen, die links von i stehen, bis es 0 erreicht oder das die Einfügestelle gefunden wurde (Vor einem Element, das kleiner als das Element i ist). Die Elemente, die im betrachteten Bereich liegen und größer sind werden während dem Durchlauf eins nach rechts verschoben.

4.2 (b) Visuelle Darstellung



Grün ist das momentan betrachtete Element/Bereich. Magenta der Einfügepunkt des Elements.

4.2 (c) Komplexität

- **Worst-Case:**

- Der Worst-Case ist ein array, der in reverse order sortiert ist.
- Demnach muss jedes Element den kompletten array durchlaufen.
- Dies ergibt eine Worst-Case Laufzeit von $\Theta(n^2)$

- **Best-Case:**

- Der Best-Case ist ein array, der schon sortiert ist.
- Demnach muss kein Element verschoben werden, aber trotzdem muss bei jedem Element einmal geprüft werden, ob es größer als sein Vorgänger ist.
- Dies ergibt eine Best-Case Laufzeit von $\Theta(n)$

- **Average-Case:**

- Der Average-Case ist ein array, der in random order sortiert ist.
- Demnach muss für jedes Element der array durchschnittlich bis zur Hälfte durchlaufen werden.
- Nach der quadratischen Steigerung für große Zahlen ist die Hälfte aber irrelevant, weswegen $\Theta(n^2)$ ist.

Algorithmus: Insertion Sort

Durch $A[j] \leq \text{key}$
wohldefiniert

```
insertionSort(A)
1 FOR i=1 TO A.length-1 DO
    // insert A[i] in pre-sorted sequence A[0..i-1]
2   key=A[i];
3   j=i-1; // search for insertion point backwards
4   WHILE j>=0 AND A[j]>key DO
5     A[j+1]=A[j]; // move elements to right
6     j=j-1;
7   A[j+1]=key;
```

Wir beginnen mit **i=1**, aber erstes Element ist **A[0]**

Short Circuit Evaluation (wie in Java):

Wenn erste AND-Bedingung **false**, wird zweite Bedingung nicht mehr ausgewertet

4.3 Bubble Sort

```
1 class BubbleSort {
2
3     void bubbleSort(int[] arr) {
4         for(int i = arr.length - 1; i > 0; i--) {
5             // Runs from arr.length - 1 to 0 (non exclusive)
6             // (i = 0 would immediately terminate)
7             boolean sorted = true;
8             for(int j = 0; j < i; j++) {
9                 // Runs from 0 to i - 1
10                if(arr[j] > arr[j + 1]) {
11                    // If the current element is greater than the next
12                    // Swap them
13                    int temp = arr[j];
14                    arr[j] = arr[j + 1];
15                    arr[j + 1] = temp;
16                    sorted = false;
17                }
18            }
19            if(sorted)
20                break;
21        }
22    }
23 }
```

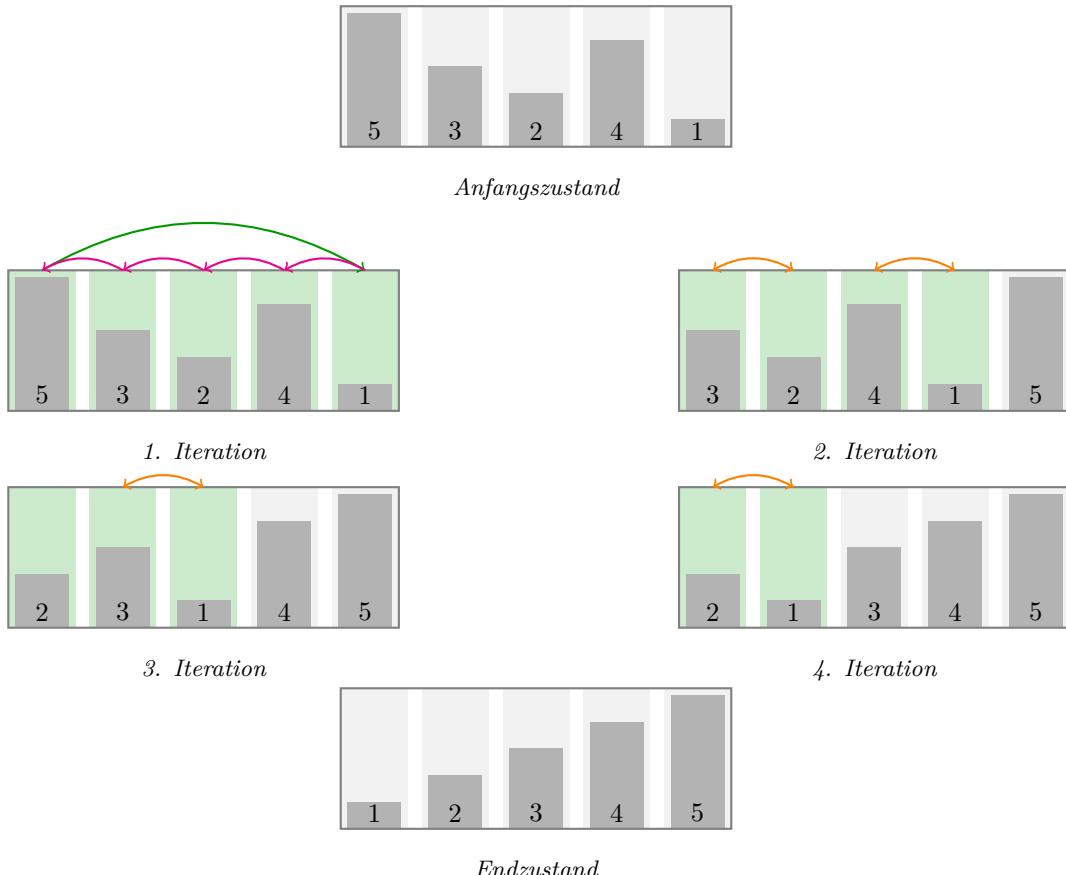
4.3 (a) Vorgehensweise

BubbleSort durchläuft die Eingabe umgekehrt zu InsertionSort: Während bei InsertionSort erst ein Element in einem Teil der Eingabe sortiert wird und der Bereich pro Iteration größer wird, wird bei BubbleSort zuerst der komplette array1 durchlaufen und beieinander liegende Elemente getauscht, wenn sie größer/kleiner sind und der Bereich mit Iteration weiter eingeschränkt. D.h., dass nach der ersten Iteration bereits das größte Element an richtiger Stelle steht, nach der zweiten das zweitgrößte etc.

Hier in dem Beispiel handelt es sich schon um einen optimierten BubbleSort. Dafür wird zusätzlich der Boolean sorted erstellt, der angibt, ob die Eingabe nach dem ersten durchlauf schon sortiert ist, was der Fall ist, wenn kein Element vertauscht wurde. Ist dies der Fall müssen keine weiteren Iteration mehr durchgeführt werden und der Algorithmus kann vorzeitig abgebrochen werden. Dies führt zu einem besseren Best-Case.

Im Vergleich zu InsertionSort ist BubbleSort meist ineffektiver als InsertionSort, obwohl sie die gleichen Komplexitäten haben. Das liegt daran, dass InsertionSort weniger Operationen ausführen muss.

4.3 (b) Visuelle Darstellung



4.3 (c) Komplexität

- **Worst-Case:**

- Die Eingabe liegt in reverse order vor.
- Das heißt, das jedes Element immer vom Anfang bis zum Ende des Bereichs durchgewechselt werden muss.
- Die Komplexität beträgt also $\Theta(n^2)$

- **Best-Case:**

- Die Eingabe ist bereits sortiert.
- Das heißt der Algorithmus muss die Eingabe nur einmal durchlaufen um zu schauen, ob Elemente getauscht werden.
- Die Komplexität beträgt also $\Theta(n)$
- (Bei nicht optimierten BubbleSort, läuft der Algorithmus immer komplett durch $\implies \Theta(n^2)$)

- **Average-Case:**

- Die Eingabe ist zufällig sortiert.
- Im Durchschnitt müssen die Elemente demnach in den meisten Fällen getauscht werden.
- Die Komplexität beträgt also $\Theta(n^2)$



Was macht der folgende Sortier-Algorithmus Bubble-Sort?

```
bubbleSort(A)
1 FOR i=A.length-1 DOWNT0 0 DO
2     FOR j=0 TO i-1 DO
3         IF A[j]>A[j+1] THEN SWAP(A[j],A[j+1]);
            //temp=A[j+1]; A[j+1]=A[j]; A[j]=temp;
```



Welche Laufzeit hat der Algorithmus?



Wie verhält er sich im Vergleich zu Insertion Sort?

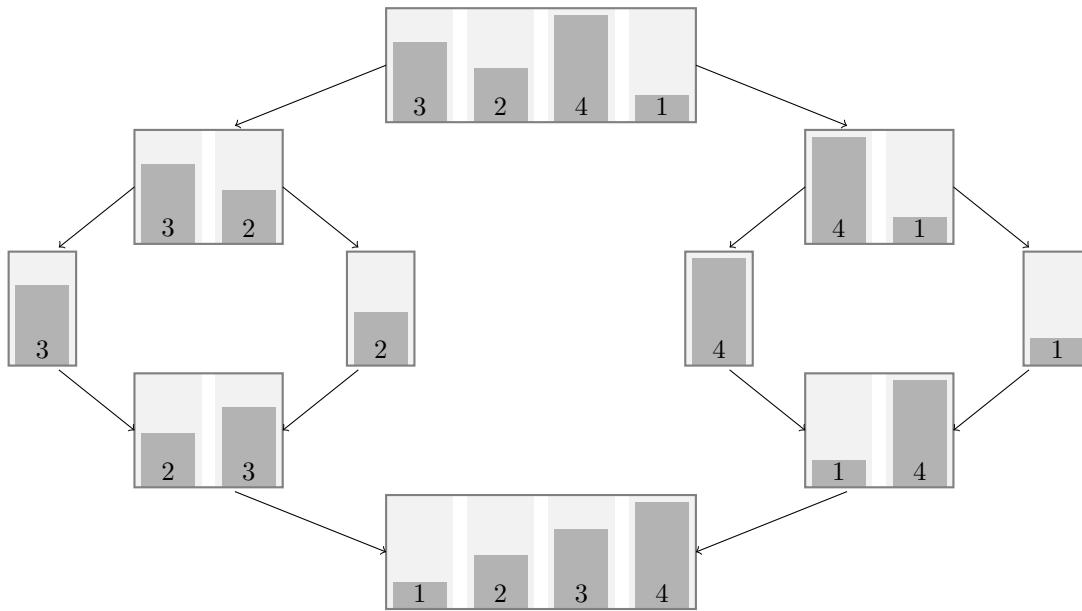
4.4 Merge Sort

```
1 class MergeSort {
2     void mergeSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // left < right, otherwise the region has no elements
5             int mid = (left + right) / 2; // Integer division -> round down
6             // Split the region into two halves and do the recursive calls
7             mergeSort(arr, left, mid);
8             mergeSort(arr, mid + 1, right);
9             // Merge the two (now sorted) halves
10            merge(arr, left, mid, right);
11        }
12    }
13
14    private void merge(int[] arr, int left, int mid, int right) {
15        int[] temp = new int[right - left + 1];
16        // Create a temporary array to store the merged elements
17
18        int p = left;
19        int q = mid + 1;
20        for (int i = 0; i < right - left + 1; i++) {
21            // Loops for each element in the region
22            if (q > right || (p <= mid && arr[p] <= arr[q])) {
23                // If p > mid the left half is finished, therefore the element needs to be in right half
24                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
25                temp[i] = arr[p];
26                p++;
27                // Adds the element at p to the temporary array and increases p
28            }
29            else {
30                temp[i] = arr[q];
31                q++;
32                // Adds the element at q to the temporary array and increases q
33            }
34        }
35        // Copy the merged elements from the temporary array back to the original array
36        for (int i = 0; i < right - left + 1; i++)
37            arr[left + i] = temp[i];
38            // left + 0 is the start of the region
39    }
40 }
```

4.4 (a) Vorgehensweise

Die Eingabe wird jeweils immer in der Mitte in zwei Teile aufgeteilt, die jeweils wieder aufgeteilt werden. Dies passiert so lange, bis alle Elemente einzeln vorhanden sind. Danach werden immer zwei dieser entstandenen Teillisten so zusammengeführt, dass sie geordnet sind. Dies wird dann wieder durchgeführt, bis alle Elemente in der Eingabe vorhanden sind und nun auch sortiert. Dieses Prinzip wird auch *Divide-and-Conquer* genannt. Bei *Divide* wird die Eingabe in zwei Teile aufgeteilt. Bei *Conquer* werden diese Teile sortiert. Dies geschieht durch die Zusammenführung von den einelementigen Teillisten, die trivial sortiert sind.

4.4 (b) Visuelle Darstellung



4.4 (c) Komplexität

- **Worst-Case:**

- Der Algorithmus funktioniert unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Worst-Case Eingabe.
- Die Eingabe kann $\log n$ ($\log_2 n$) mal in zwei aufgeteilt werden kann. Zusätzlich benötigt der Algorithmus zum Kombinieren von den Teillisten n
- Es ergibt sich also die Komplexität von $\Theta(n \log n)$

- **Best-Case:**

- Wie zuvor angesprochen, läuft der Algorithmus unabhängig von der Sortiertheit der Eingabe, demnach gibt es keine Best-Case Eingabe und der Best-Case ist gleich dem Worst-Case.
- Es ergibt sich also $\Theta(n \log n)$

- **Average-Case:**

- Wie oben, für alle Fälle gleich, also $\Theta(n \log n)$

Algorithmus: Merge Sort

Wir sortieren im Array **A** zwischen Position **left** (links) und **right** (rechts)

```
mergeSort(A, left, right) //initial left=0, right=A.length-1

1 IF left<right THEN //more than one element
2   mid=floor((left+right)/2); // middle (rounded down)
3   mergeSort(A, left, mid); // sort left part
4   mergeSort(A, mid+1, right); // sort right part
5   merge(A, left, mid, right); // merge into one
```

genauer: letzter Index
des linken Teils

$$mid = \left\lceil \frac{right - left + 1}{2} \right\rceil + \frac{2left}{2} - 1 = \left\lceil \frac{right + left - 1}{2} \right\rceil = \left\lfloor \frac{right + left}{2} \right\rfloor$$

Anzahl Elemente /2
(gerundet)

Offset
(beginnend mit 0)

Beispiele:
left=3, right=4, mid=3
left=3, right=5, mid=4

Algorithmus: Merge (für sortierte Teillisten)

rechte Liste noch aktiv und
[linke Liste bereits abgearbeitet oder
nächstes Element rechts] rechte Liste bereits abgearbeitet oder
[linke Liste noch aktiv und nächstes Element links]

```
merge(A, left, mid, right) // requires left<=mid<=right
                           // temporary array B, right-left+1 elements

1 p=left; q=mid+1;           // position left, right
2 FOR i=0 TO right-left DO // merge all elements
3   IF q>right OR (p<=mid AND A[p]<=A[q]) THEN
4     B[i]=A[p];
5     p=p+1;
6   ELSE //next element at q
7     B[i]=A[q];
8     q=q+1;
9 FOR i=0 TO right-left DO A[i+left]=B[i]; //copy back
```

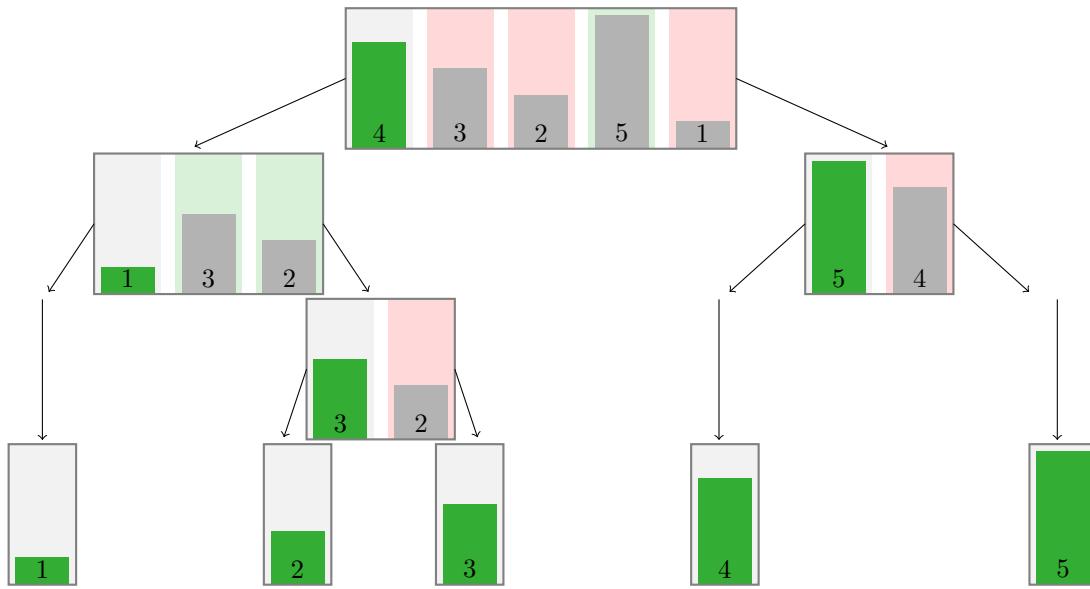
4.5 Quicksort

```
1 class Quicksort {
2     void quickSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // Region contains more than one element
5             int part = partition(arr, left, right);
6             quickSort(arr, left, part);
7             quickSort(arr, part + 1, right);
8         }
9     }
10
11     private int partition(int[] arr, int left, int right) {
12         int pivot = arr[left];
13         // Pivot is the first element in the region
14
15         int p = left - 1;
16         int q = right + 1;
17         while (p < q) {
18             do p++; while (arr[p] < pivot);
19             do q--; while (arr[q] > pivot);
20             // Increase / decrease p and q until the elements are bigger/smaller-equal pivot
21             if (p < q) {
22                 /* p < q here means that theres a number bigger equal pivot on the left side
23                  and a number smaller equal than the pivot on the right side
24                  Therefore, we swap them to sort them into their halves*/
25                 int temp = arr[p];
26                 arr[p] = arr[q];
27                 arr[q] = temp;
28                 // Swap arr[p] and arr[q]
29             }
30         } /* This loop runs until p and q cross each other
31            which means that */
32         return q;
33         // q is the index at which:
34         // all indices greater than q contain elements bigger equal pivot
35         // all indices smaller equal q contain elements smaller equal pivot
36     }
37 }
```

Quicksort funktioniert vom Prinzip ähnlich zu Mergesort. Auch hier wird die Eingabe in zwei Teillisten aufgeteilt und der rekursiv wiederholt. Hier findet die Sortierung allerdings anders statt. Anstatt die Sortierung durch die Zusammenführung zweier Listen zu realisieren, werden hier die einzelnen Elemente anhand des Vergleiches an einem anderen Elementes links oder rechts von diesem eingeordnet. Dies führt durch das *Divide-and-Conquer* Prinzip dazu, dass die Eingabe die Element in die zwei Teile, größer und kleiner des Pivots einordnet. Diese beiden Teile werden dann wiederum genauso behandelt, bis schließlich der gesamte array1 geordnet ist.

Bei der Implementation wird häufig anstatt den Pivot als erstes Element des Bereiches zu definieren, dieser zufällig gewählt, was zu einem besseren average-case führt, wenn die Eingabe bereits einigermaßen sortiert ist. Quicksort ist zwar in der Theorie in den meisten Situationen nicht unbedingt besser als Merge sort auf die Komplexität bezogen, in der Praxis aber oft schneller, durch die Ineffizienz von Kopieroperationen, die für Quicksort wegfallen.

4.5 (a) Visuelle Darstellung



4.5 (b) Komplexität

- **Worst-Case:**

- Im Worst-Case wird für pivot immer das größte oder kleinste Element verwendet, was sehr unausgeglichenen Partitionen erzeugt.
- Dies würde eine Rekursionstiefe von n bedeuten
- Pro Rekursion muss dann der Bereich immernoch mit n durchlaufen werden
- Dies bedeutet eine Worst-Case Laufzeit von $\Theta(n^2)$

- **Best-Case:**

- Im Best-Case wird immer das Element als pivot verwendet, das den Median der Liste bildet, was die Partitionen immer ausbalanciert.
- Dies bedeutet eine Rekursionstiefe von $\log n$
- Pro Rekursion muss dann der Bereich immernoch mit n durchlaufen werden
- Dies bedeutet eine Best-Case Laufzeit von $\Theta(n \log n)$

- **Average-Case:**

- Im Average-Case wird ein zufälliges Element als pivot verwendet, wodurch die Partitionen im mittel gleich sind.
- Dies bedeutet eine Rekursionstiefe von $\log n$
- Pro Rekursion muss dann der Bereich immernoch mit n durchlaufen werden
- Dies bedeutet eine Average-Case Laufzeit von $\Theta(n \log n)$

Algorithmus: Quicksort

```
quicksort(A, left, right) //initial left=0, right=A.length-1

1 IF left<right THEN //more than one element
2   q=partition(A, left, right); // q partition index
3   quicksort(A, left, q); // sort left part
4   quicksort(A, q+1, right); // sort right part
```

```
partition(A, left, right) //req. left<right, ret. left..right-1

1 pivot=A[left];
2 p=left-1; q=right+1; //move from left resp. right
3 WHILE p<q DO
4   REPEAT p=p+1 UNTIL A[p]>=pivot; //left up
5   REPEAT q=q-1 UNTIL A[q]<=pivot; //right down
6   IF p<q THEN Swap(A[p],A[q]);
7 return q // A[left..q], A[q+1..right]
```

4.6 Radix Sort

```
1 import java.util.ArrayList;
2
3 class RadixSort {
4     int D = 10; // possible unique digits
5     int d; // Max amount of digits
6     ArrayList<Integer>[] buckets = new ArrayList[D];
7
8     void radixSort(int[] arr) {
9         d = amountDigits(arr);
10        for (int i = 0; i < d; i++) {
11            // for each digit in the array, 0 least significant
12            for (int j = 0; j < arr.length; j++)
13                putBucket(arr, i, j);
14            // Sorts the numbers into their buckets
15            int a = 0;
16            for (int k = 0; k < D; k++) {
17                for (int b = 0; b < buckets[k].size(); b++) {
18                    arr[a] = buckets[k].get(b);
19                    a++;
20                }
21                buckets[k].clear();
22            }
23            // Reads out the buckets in order
24        }
25    }
26
27    private void putBucket(int[] arr, int i, int j) {
28        int z = arr[j] / (int) Math.pow(D, i) % D;
29        // Gets the ith digit of the number
30        int b = buckets[z].size();
31        // size is next free index
32        buckets[z].add(b, arr[j]);
33        // puts the number in the bucket z at position b
34        // Depending on implementation might need to increase size manually
35    }
36
37    private int amountDigits(int[] arr) {
38        int max = arr[0];
39        for (int i = 1; i < arr.length; i++) {
40            if (arr[i] > max)
41                max = arr[i];
42        }
43        // Get the biggest number
44        return (int) (Math.log(max)/Math.log(D) + 1);
45        // Get the amount of digits of the number
46        // log(max)/log10(D) is equal to log_D(max)
47    }
48 }
```

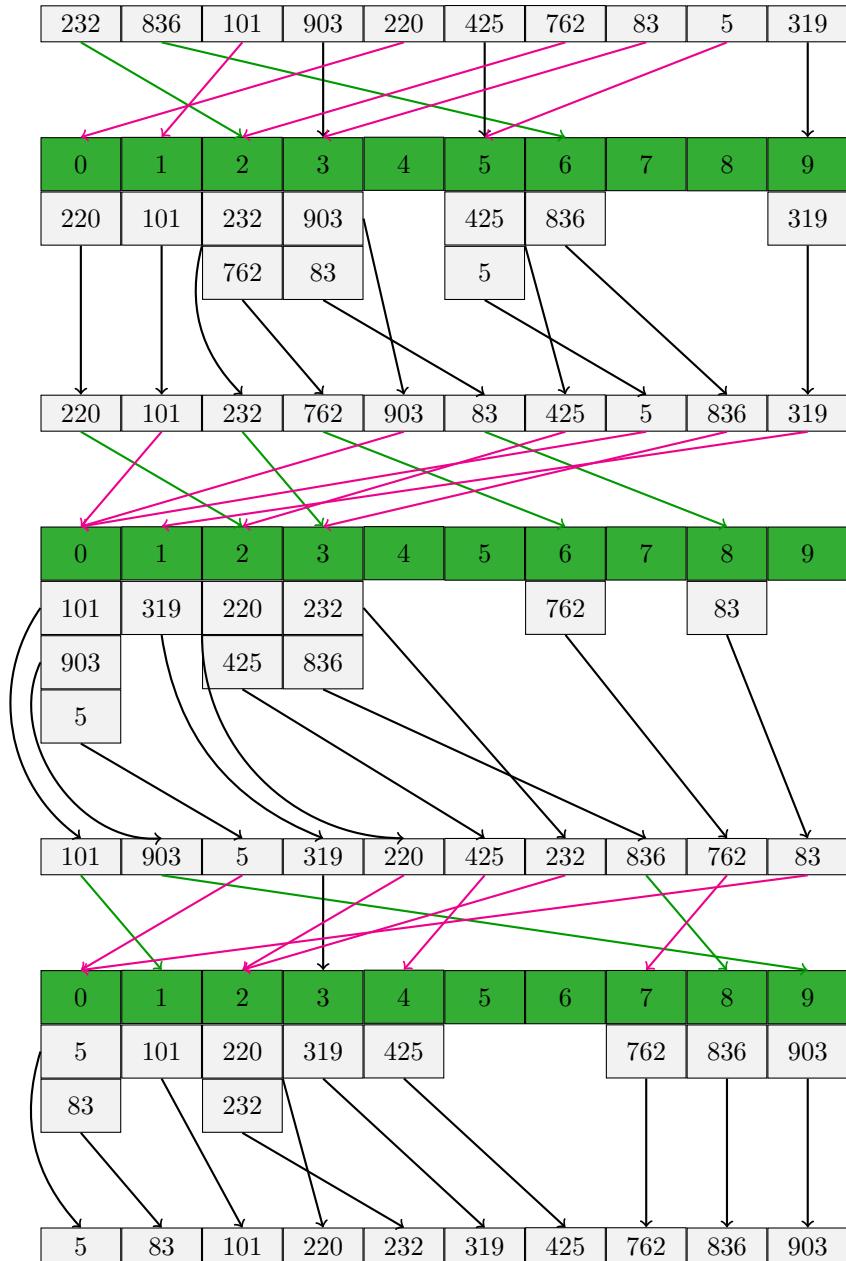
4.6 (a) Vorgehensweise

Bei RadixSort wird die Eingabe für jede Dezimalstelle sortiert. D.h., dass die Eingabe zuerst anhand von der 1er-Stelle sortiert wird, dann der 10er-Stelle, und so weiter.

Dies geschieht durch die Einordnung der Elemente in "Buckets", die jeweils einen möglichen Wert für die Dezimalstelle darstellen(z.B. {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}). Nachdem alle Werte in Buckets eingeordnet wurden, werden diese Buckets nun nach Signifikanz ausgelesen (0 ist kleiner als 9, also werden 0 zuerst ausgelesen) und nach der bearbeiteten Ziffer sortiert in die Eingabe zurückgefügt. Dadurch liegt der array1 für die Ziffer nun sortiert da.

Dies wird nun für die nächste Dezimalstelle wiederholt, wodurch die Eingabe jetzt für die ersten beiden Dezimalstellen sortiert ist. Dies wird wiederholt, bis alle Dezimalstellen durchlaufen sind, wodurch dann alle Werte sortiert sind.

4.6 (b) Visuelle Darstellung



Die Farben haben keine spezielle Bedeutung und dienen nur der Visualisierung.

4.6 (c) Komplexität

- Da bei RadixSort die Eingabe nur von der Anzahl der möglichen Ziffernvariationen D , der Eingabelänge n und die maximale Anzahl der Ziffern d abhängig ist, ist der Algorithmus für **Best-, Worst- und Average-case** gleich.
- Dieser beträgt im Allgemeinen $O(d \cdot (n + D))$
- D wird aber oft als Konstant angesehen, weshalb $O(d \cdot n)$ oft verwendet wird.
- Wenn man zusätzlich noch d als konstant ansieht so ergibt sich lineare Laufzeit $O(n)$
- Nähert sich D n an, so ergibt sich allerdings eine Laufzeit von $O(n \log n)$, da $d = \Theta(\log_D n)$ gilt.

Laufzeit

```
radixSort(A) // keys: d digits in range [0,D-1]
// B[0][], ..., B[D-1][] buckets (init: B[k].size=0)

1  FOR i=0 TO d-1 DO //0 least, d-1 most sign. digit
2      FOR j=0 TO n-1 DO putBucket(A,B,i,j); ----- O(n)
3      a=0;
4      FOR k=0 TO D-1 DO          //rewrite to array
5          FOR b=0 TO B[k].size-1 DO
6              A[a]=B[k][b]; //read out bucket in order
7              a=a+1;
8          B[k].size=0;        //clear bucket again----- O(D)
9      return A
```

Gesamlaufzeit
 $O(d \cdot (n + D))$

$O(n)$

$O(n)$
Schritte
(alles in A kopieren)

$O(D)$

$O(1)$

```
putBucket(A,B,i,j) // call-by-reference
1  z=A[j].digit[i]; // i-th digit of A[j]
2  b=B[z].size;     // next free spot
3  B[z][b]=A[j];
4  B[z].size=B[z].size+1;
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 02 Sortieren | 143

DARMSTADT Technische Universität Darmstadt

5 Grundlegende Datenstrukturen

5.1 Stacks

Stacks operieren unter dem "First in - Last out" (FILO) Prinzip. Ähnlich zu einem Kartendeck, wo die unterste (Erste Karte) die ist, die als letztes gezogen wird.

Stacks werden normalerweise mit den folgenden Funktionen erstellt:

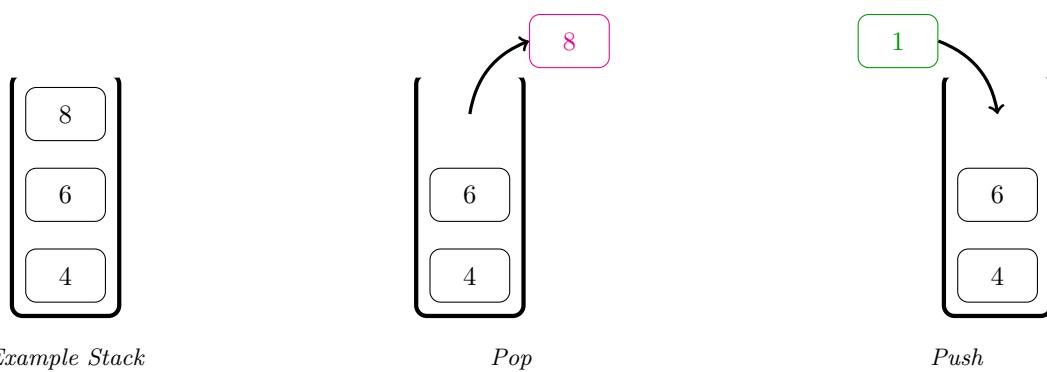
- `new(n)`: Erstellt einen neuen Stack.
- `isEmpty`: gibt an ob der Stack leer ist.
- `pop`: gibt das oberste Element des Stacks zurück und entfernt es vom Stack.
- `push(k)`: Fügt `k` auf den Stack hinzu

Eine mögliche Implementation auf Grundlage eines Arrays wäre:

```
1 class Stack {  
2     private int[] arr;  
3     private int top;  
4     Stack(int size) {  
5         arr = new int[size];  
6         top = -1;  
7         // Creates a new array with size  
8     }  
9     boolean isEmpty() {  
10        return top < 0;  
11        // Returns true if empty  
12    }  
13    int pop() { // O(1)  
14        return arr[top--];  
15        // Removes and returns the top element  
16    }  
17    void push(int k) { // O(1)  
18        arr[++top] = k;  
19        // Adds an element  
20    }  
21 }
```

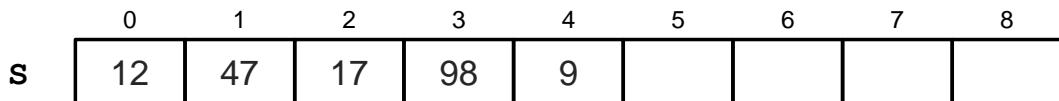
Push und Pop schmeißen Fehlermeldung wenn Stack leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wär es automatisch IndexOutOfBoundsException.

Oft werden Stacks auch mit variabler Größe implementiert. Dies kann über verschiedene Wege passieren, zum Beispiel Kopieren des arrays in einen größeren Array oder implementation über mehrere Arrays (z.B. über Linked List). Häufig wird das erstere so implementiert, dass der Array in einen Array mit doppelter Größe kopiert wird.



Stacks als Array: Algorithmen

Annahme: maximale Größe MAX
des Stacks vorher bekannt



new(S)

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

S.top

isEmpty(S)

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

pop(S)

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

push(S, k)

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

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 9

DARMSTADT Technische Universität Darmstadt

Feldarbeit: Algorithmen

RESIZE(S, m)

reserviert neuen Speicher der Größe m,
kopiert S.A um, und lässt S.A auf neuen Speicher zeigen

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, S.memsize);
8   return S.A[S.top + 1];
```

push(S, k)

```
1 S.top = S.top + 1;
2 S.A[S.top] = k;
3 IF S.top + 1 == S.memsize THEN
4   S.memsize = 2 * S.memsize;
5   RESIZE(S, S.memsize);
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 14

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptoplexity Cryptography & Complexity Theory Institute for Theoretical Cryptology www.cryptoplexity.de

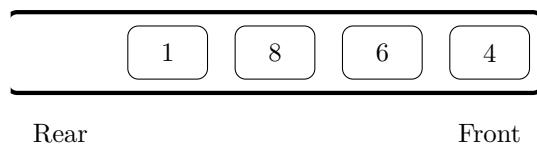
5.2 Queues

Queues funktionieren entgegengesetzt zu Stacks. Sie funktionieren nach dem FIFO-Prinzip (First in - First out). Kann als Warteschleife dargestellt werden. Die Person, die sich als erstes anstellt, kommt auch als erstes dran. Queues werden normalerweise mit den folgenden Funktionen erstellt:

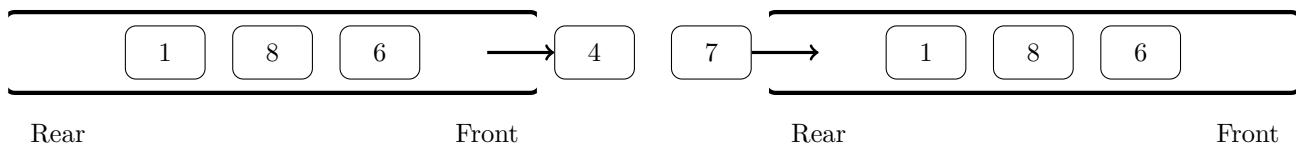
- **new(n):** Erstellt einen neuen Queue.
- **isEmpty:** gibt an ob der Queue leer ist.
- **enqueue(k):** Fügt k auf den Queue hinzu
- **dequeue:** gibt das erste Element des Queues zurück und entfernt es vom Queue.

Hier ist die Implementation für Queues wie folgt:

```
1 class Queue {  
2     private int[] arr;  
3     private int front;  
4     private int back;  
5  
6     Queue(int size) {  
7         arr = new int[size];  
8         front = -1;  
9         back = -1;  
10    }  
11  
12    boolean isEmpty() {  
13        return back == -1;  
14    }  
15  
16    boolean isFull() {  
17        return (front + 1) % arr.length == back;  
18        // If front + 1 is equal to back, the queue is full  
19        // Modulo makes this usable for cyclic arrays  
20    }  
21  
22    void enqueue(int k) { // O(1)  
23        if (isFull()) {  
24            throw new UException("Queue is full");  
25        } else {  
26            if (isEmpty())  
27                front = 0;  
28            back = (back + 1) % arr.length;  
29            // Modulo so that cyclic arrays work  
30            arr[back] = k;  
31        }  
32    }  
33  
34    int dequeue() { // O(1)  
35        if (isEmpty()) {  
36            throw new UException("Queue is empty");  
37        } else {  
38            int temp = arr[front];  
39            front = (front + 1) % arr.length;  
40            // Modulo so that cyclic arrays work  
41            if (front == back) {  
42                front = -1;  
43                back = -1;  
44            }  
45            // If front and back are equal, the queue is empty -> reset  
46            return temp;  
47        }  
48    }  
49}
```

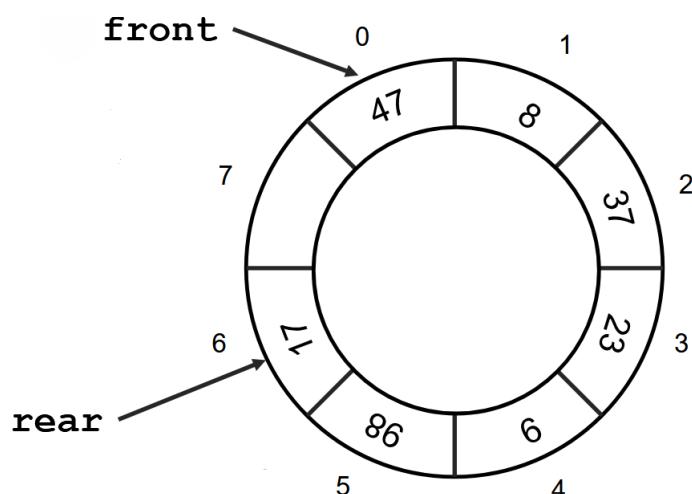


Example non-cyclic Queue



Dequeue

Enqueue



Cyclic Queue

Queues als zyklisches Array: Algorithmen

Q leer, wenn
 $front==rear+1 \bmod MAX$
und $empty==true$

Q voll, wenn
 $front==rear+1 \bmod MAX$
und $empty==false$

new(Q)

```
1 Q.A[] = ALLOCATE(MAX);  
2 Q.front=0;  
3 Q.rear=-1;  
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+1 mod MAX  
6     THEN Q.empty=true;  
7   return Q.A[Q.front-1 mod MAX];
```

enqueue(Q, k)

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

Queues durch Liste: Algorithmen

new(Q)

```
1 Q.front=nil;  
2 Q.rear=nil;
```

isEmpty(Q)

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

dequeue(Q)

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

enqueue(Q, x)

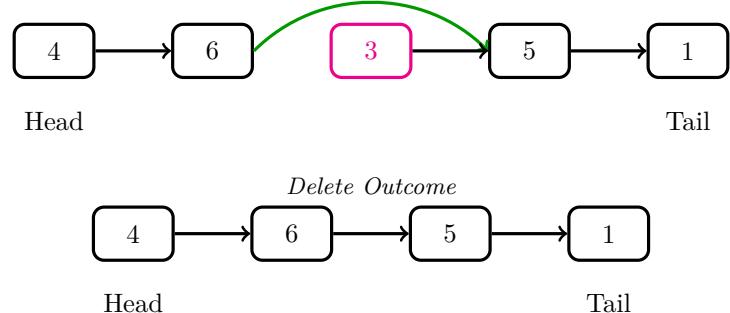
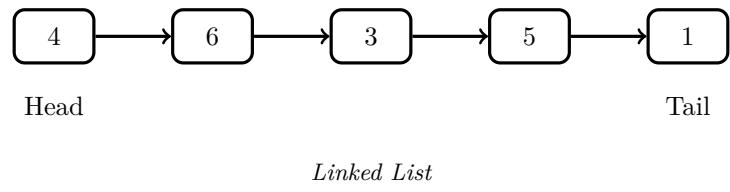
```
1 IF isEmpty(Q) THEN  
2   Q.front=x;  
3 ELSE  
4   Q.rear.next=x;  
5   x.next=nil;  
6   Q.rear=x;
```

5.3 Linked List

Eine einfache Linked List besteht aus mehreren Elementen, die jeweils immer einen Wert und eine Referenz auf das nächste Element in der Liste haben. Diese Struktur hat den Vorteil, dass sie keine festgelegte Größe hat, das Einfügen in $O(1)$ stattfindet, einfach zu implementieren ist und im Speicher nicht als Block, sondern einzelne Referenzen steht. Eine einfache Linked List kann wie folgt implementiert werden:

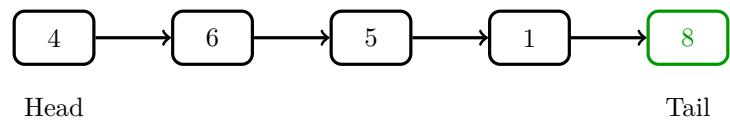
```
1 class LinkedList {
2     class LinkedElement {
3         Integer key = null;
4         LinkedElement next = null;
5
6         LinkedElement(Integer key) {
7             this.key = key;
8         }
9     }
10    LinkedElement head = null; // First element in list
11    LinkedElement tail = null; // Last element in list
12
13    void insert(int k) { // O(1)
14        LinkedElement elem = new LinkedElement(k);
15        if (head == null) {
16            head = elem;
17            tail = elem;
18        } else {
19            tail.next = elem;
20            tail = elem;
21        }
22    }
23
24
25    void delete(int k) { // O(n)
26        LinkedElement prev = null;
27        LinkedElement curr = head;
28        while (curr != null && curr.key != k) {
29            prev = curr;
30            curr = curr.next;
31        }
32        if (curr == null)
33            throw new UException("Element not found");
34
35        if (prev != null) {
36            prev.next = curr.next;
37            if (curr == tail)
38                tail = prev;
39        } else {
40            head = curr.next;
41        }
42    }
43
44    LinkedElement search(int k) { // O(n)
45        LinkedElement curr = head;
46        while (curr != null && curr.key != k)
47            curr = curr.next;
48        if (curr == null)
49            throw new UException("Element not found");
50        return curr;
51    }
52 }
```

Diese Implementation benutzt einen Head und Tail, hat aber nur Referenz für das nächste Element in der Liste. Eine alternative Implementation wäre Tail wegzulassen und den Nodes eine previous-Referenz zu geben. Damit könnte man beim Einfügen das Element vorne an den Head anzuhängen und die neue Node als Head zuzuweisen. search bleibt gleich, bei delete muss lediglich die previous Referenz angepasst werden.



Delete Quasi Outcome

Die 3 Node wird zwar nicht wirklich "gelöscht", allerdings wird die Referenz aus der Liste genommen, wodurch keine Referenz mehr auf diese Node besteht.



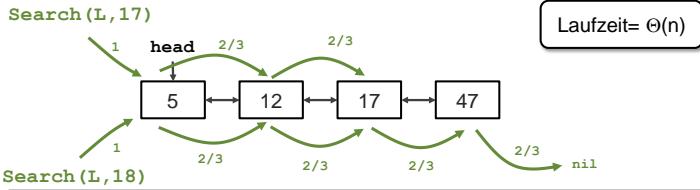
Insert of 8

8 wird an tail angehngt und wird dann zum tail

Elementare Operationen auf verketteten Listen

```
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;
```

short circuit evaluation
(wie in Java)



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 22

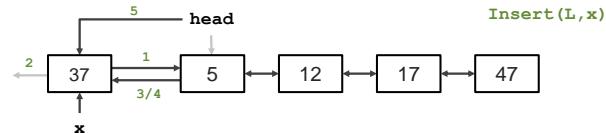
TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Elementare Operationen auf verketteten Listen

```
insert(L,x) //inserts element x in L
1 x.next=L.head;
2 x.prev=nil;
3 IF L.head != nil THEN
4   L.head.prev=x;
5   L.head=x;
```

call-by-reference
bzw. call-by-value
für Objekte wie in Java

Laufzeit = $\Theta(1)$



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 23

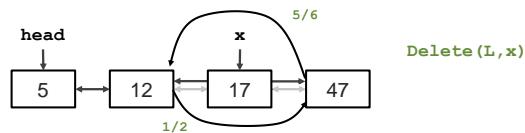
TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Elementare Operationen auf verketteten Listen

```
delete(L,x) //deletes element x from L
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 = $\Theta(1)$

Achtung: Löschen eines Wertes k kostet Zeit $\Omega(n)$



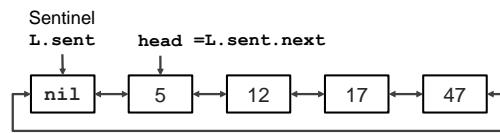
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 25

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Vereinfachung per Wächter/Sentinels

```
delete(L,x) //deletes element x from L
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;
```

Ziel:
eliminiere die
Spezialfälle für
Listenanfang/-ende



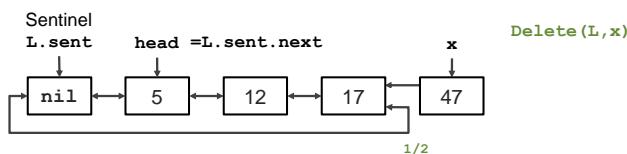
Sentinel ist „von außen“ nicht sichtbar

Leere Liste besteht nur aus Sentinel

Löschen mit Sentinels

```
deleteSent(L,x)
// deletes x from L with sentinel
1 x.prev.next=x.next;
2 x.next.prev=x.prev;
```

Andere Operationen
wie Einfügen
und Löschen
müssen auch
angepasst werden



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 27

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

5.4 Binary Search Tree

Ein Binary Search Tree ist eine Datenstruktur, die aus mehreren Nodes besteht, die jeweils pointer zu drei Nodes besitzt: Left, Right und Parent.

Hierbei repräsentiert Left und Right die Nodes, die unter der current Node stehen und Parent die, die über der current Node steht. Dabei ist im Binary Search Tree (Im Gegensatz zum normalen Search Tree) Left immer kleiner als die Node und Right immer größer gleich der Node.

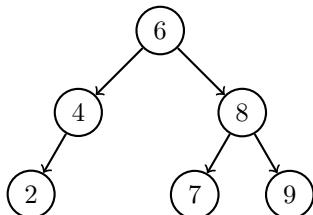
Dies erlaubt es Elemente in dem Tree schnell zu finden, da nicht alle Elemente durchlaufen werden müssen, sondern immer nur ein Pfad, bei dem das Element größer/kleiner ist.

Ein idealer Binary Search Tree ist so balanciert, dass beide Seiten des Baumes die selbe Anzahl an Knoten besitzen. Dies wäre eine ideale Höhe von $h = \log n$. Ein schlechter Binary Search Tree allerdings ist unbalanciert, so dass der Worst-Case so aussieht, dass alle Nodes jeweils maximal ein Kind haben. Dies wäre effektiv gleich einer LinkedList.

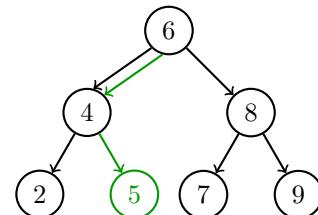
```

1 class BSTree {
2     class BSTNode {
3         Integer key;
4         BSTNode left;
5         BSTNode right;
6         BSTNode parent;
7         BSTNode(Integer k) {
8             key = k;
9         }
10    }
11    BSTNode root;
12    void insert(BSTNode z) { // Omega(1), O(h), Theta(h)
13        BSTNode x = root; // Traversal starting from the root
14        BSTNode px = null; // Parent of x, initially null
15        while(x != null) {
16            px = x;
17            if (z.key < x.key)
18                x = x.left;
19            else
20                x = x.right;
21       } // Traversing the tree until finding the insertion point
22        z.parent = px; // Sets the parent of the node to be inserted
23        if (px == null) // px only null if the tree is empty -> loop never runs -> z is root
24            root = z;
25        else if (z.key < px.key) // Key smaller -> left child
26            px.left = z;
27        else // Key bigger -> right child
28            px.right = z;
29        // May add the same node twice as it doesn't check for duplicates
30    }
}

```



Before insert

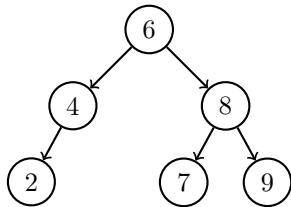


Insert 5

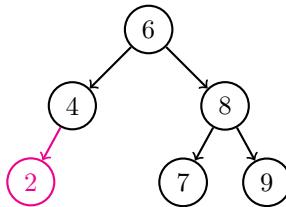
```

1 void delete(BSTNode z) { // Omega(1), O(h), Theta(log n)
2     if (z.left == null) // If z has no left, transplants the right to z's position
3         transplant(z, z.right);
4     else if (z.right == null) // If z has no right, transplants the left to z's position
5         transplant(z, z.left);
6     else { // If z has both left and right children
7         BSTNode y = z.right;
8         while (y.left != null)
9             // Finds the next biggest element of z = smallest in right subtree of z
10            y = y.left;
11         if (y.parent != z) { // If the next biggest element y is not child of z
12             transplant(y, y.right); // Transplants the right child of y to y's position
13             y.right = z.right; // The right child of y becomes the right child of z
14             y.right.parent = y; // The parent of the right child of y becomes y
15         }
16         transplant(z, y); // Transplants y to z's position
17         y.left = z.left; // The left child of y becomes the left child of z
18         y.left.parent = y; // The parent of the left child of y becomes y
19     }
20 }
21 void transplant(BSTNode u, BSTNode v) { // O(1)
22     // Transplants v to the parent of u
23     if (u.parent == null) // If u is the root, v becomes the new root
24         root = v;
25     else if (u == u.parent.left) // If u is a left child, v becomes a left child
26         u.parent.left = v;
27     else // If u is a right child, v becomes a right child
28         u.parent.right = v;
29     if (v != null) // If v is not null, v becomes a child of u's parent
30         v.parent = u.parent;
31 }

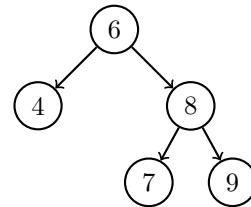
```



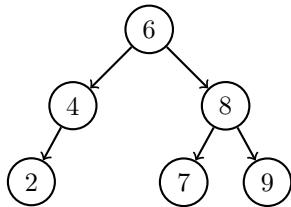
Leaf Deletion



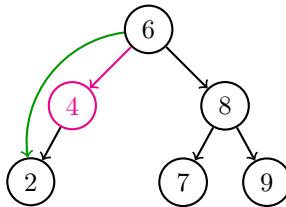
Delete 2



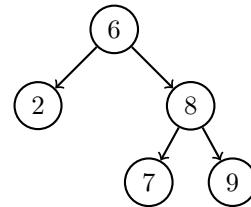
Result



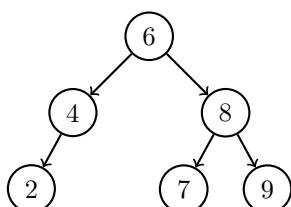
Half-Leaf Deletion



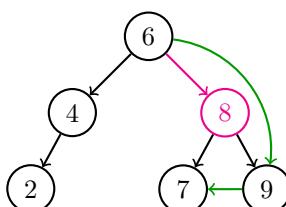
Delete 4



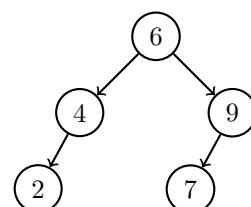
Result



Complete Node Deletion



Delete 8



Result

Leaves werden gelöscht, Half-Leaves durch Kind ersetzt, Complete Node durch Nachfolger (nächstgrößtes Element, kleinstes Element im rechten Teilbaum der Node) ersetzt.

```

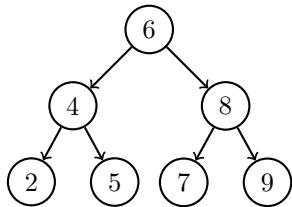
1   BSTNode iterativeSearch(int k) { // O(h), Omega(1), Theta(log n)
2       BSTNode curr = root;
3       while (curr != null && curr.key != k) {
4           if (k < curr.key)
5               curr = curr.left;
6           else
7               curr = curr.right;
8       }
9       return curr; // Returns null if element not found
10  }
11  BSTNode recursiveSearch(int k, BSTNode curr) { // O(h), Omega(1), Theta(log n)
12      if (curr == null)
13          return null;
14      if (k < curr.key)
15          return recursiveSearch(k, curr.left);
16      else if (k > curr.key)
17          return recursiveSearch(k, curr.right);
18      return curr; // Returns null if element not found
19  }

```

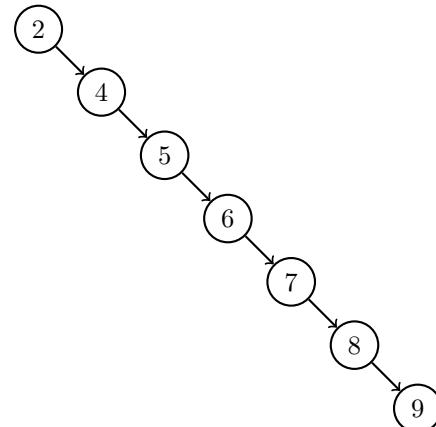
```

1   void traversal(BSTNode curr) { // O(n)
2       if (curr != null)
3           return;
4       // Any actions that should be done in a specific order can be done
5       // Here for preorder traversal
6       traversal(curr.left);
7       // Here for inorder traversal
8       traversal(curr.right);
9       // Here for postorder traversal
10      // Left and right can also be exchanged to traverse in reverse order
11  }
12 }

```

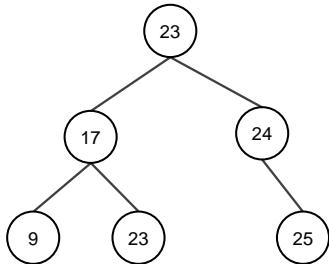


Ideal balanzierter BST ($h = \log n$)



Worst-Case unbalanzierter BST($h = n$)

Inorder-Traversieren von Binärbäumen



Beispielanwendung:
Serialisierung

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

Bei Bedarf mit „Wrapper“
`inorderTree(T)=inorder(T.root)`

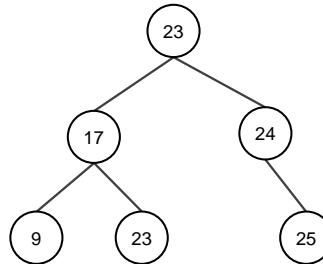
`inorder(T.root)` ergibt

9 17 23 23 24 25

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 47

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Pre- und Postorder-Traversieren von Binärbäumen (II)



`preorder(T.root)` ergibt

23 17 9 23 24 25

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

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

`postorder(T.root)` ergibt

9 23 17 25 24 23

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 51

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Iterative Suche im Binären Suchbaum

```
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);
```

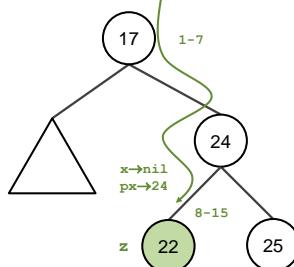
```
iterative-search(x,k) //Aufruf x=root
1 WHILE x != nil AND x.key != k DO
2   IF x.key > k THEN
3     x=x.left
4   ELSE
5     x=x.right;
6 return x;
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 70

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Einfügen im BST

insert(T,z)



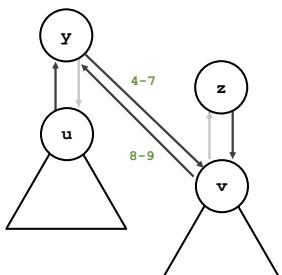
Laufzeit = $O(h)$

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

```
1 x=T.root; px=nil;
2 WHILE x != nil DO
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==nil THEN
10    T.root=z;
11 ELSE
12   IF px.key > z.key THEN
13     px.left=z;
14   ELSE
15     px.right=z;
```

Löschen: Transplantation

hängt Teilbaum v an Elternknoten von u



```
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;
```

zur Erinnerung

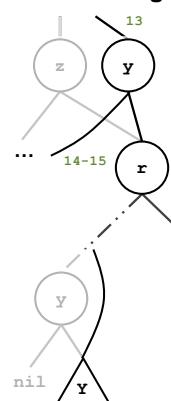
Laufzeit = $O(1)$

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 75

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Löschen: Algorithmus (IV)

Laufzeit = $O(h)$



delete(T,z)

```
1 IF z.left==nil THEN
2   transplant(T,z,z.right)
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;
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 03 Grundlegende Datenstrukturen | 79

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

6 Fortgeschrittene Datenstrukturen

6.1 Red-Black Tree

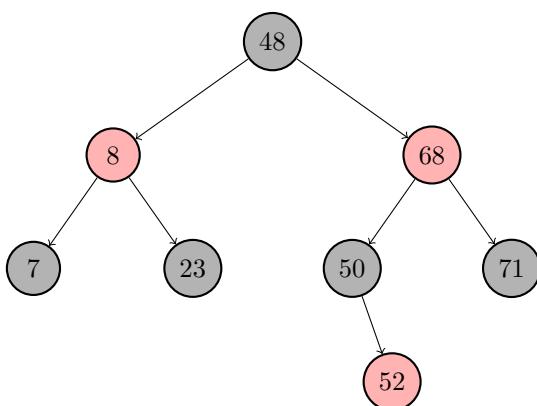
Ein Red-Black Tree ist eine Art Binary-Search Tree. Zusätzlich zu diesem besitzen die Nodes in einem RB Tree noch das Attribut `color`. Die Nodes werden also entweder als `red` oder `black` definiert. Dies dient zur Einhaltung der Red-Black-Regeln, durch die die Effizienz der Datenstruktur im Vergleich zum BST verbessert wird.
Die Regeln sind:

1. Jeder Knoten ist entweder schwarz oder rot
2. Die Wurzel ist schwarz
3. Rote Knoten haben keine Roten Kinder
4. Jeder Pfad von einem Knoten zu seinen Nachkommen besitzt die selbe Anzahl an schwarzen Knoten

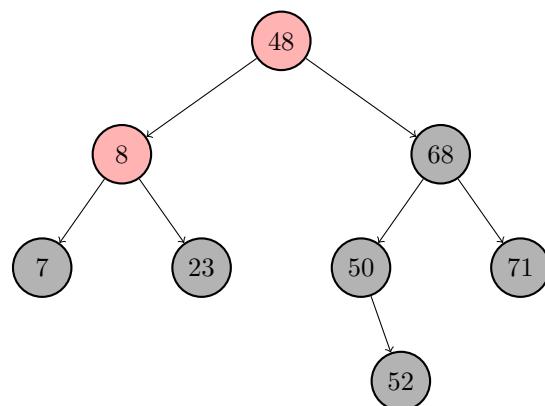
⇒ Hat ein Knoten nur ein Kind, so muss dieses Kind Rot sein, ansonsten ist die Anzahl an schwarzen Knoten auf dem Pfad unterschiedlich zu den anderen Pfaden.

Der Vorteil von RBT zu BST ist, dass während ein BST unausgewogen sein kann, was in einem Worst-Case von $h = n$ resultiert, im RBT durch die Regeln eine maximale Höhe von $h = 2 \cdot \log(n + 1)$ sichergestellt, was die Worst-Case Laufzeit der Algorithmen deutlich verbessert.

```
1 class RBTree {
2     class RBNode {
3         Integer key;
4         RBNode left;
5         RBNode right;
6         RBNode parent;
7         Color color;
8         RBNode(Integer k) {
9             key = k;
10        }
11    }
12    RBNode sent;
13    RBNode root = null;
14    RBTree() {
15        sent = new RBNode(null);
16        sent.color = Color.BLACK;
17        sent.left = sent;
18        sent.right = sent;
19        // Sentinel always points to itself ->
20        // node.parent.parent and its children will never result in null references
21    }
22    // Traversal and search are the same as BSTree
```



Richtig Konstruierter RBT



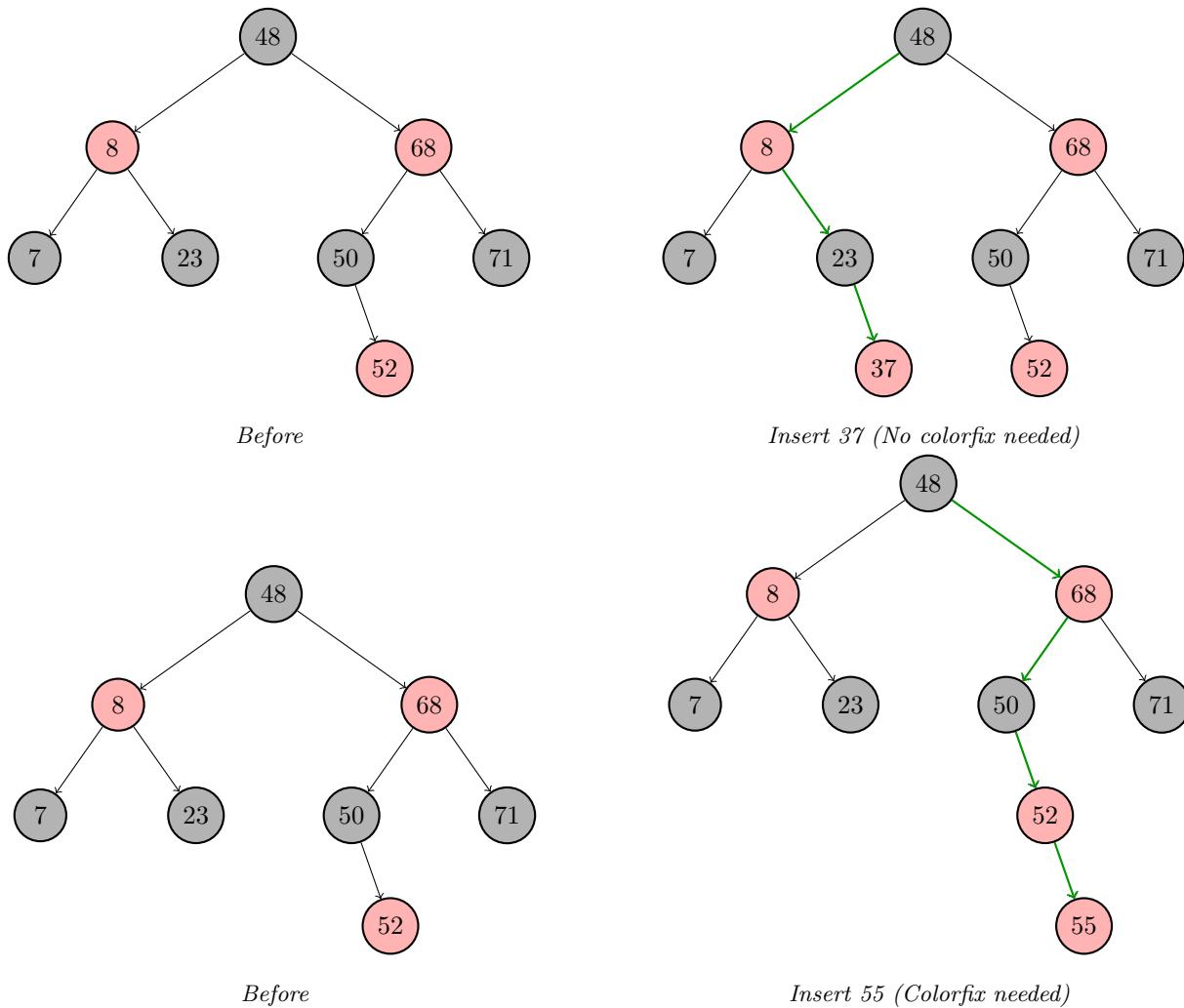
Falsch Konstruierter RBT

```

1 void insert(RBNode z) { // Omega(1), O(log n), Theta(log n)
2     // Very similar to BSTree, with addition of color and parent of sentinel instead of null
3     RBNode x = root; // Traversal starting from the root
4     RBNode px = sent; // Parent of x, initially sentinel unlike BST
5     while (x != null) {
6         px = x;
7         if (z.key < x.key)
8             x = x.left;
9         else
10            x = x.right;
11    } // Traversing the tree until finding the insertion point
12    z.parent = px; // Sets the parent of the node to be inserted
13    if (px == sent) // px only sentinel if the tree is empty -> loop never runs -> z is root
14        root = z;
15    else if (z.key < px.key) // Key smaller -> left child
16        px.left = z;
17    else // Key bigger -> right child
18        px.right = z;
19    z.color = Color.RED; // Sets color of new Node to red, will not necessarily stay red
20    fixColorsAfterInsertion(z); // Fixes colors in tree after insertion to maintain RB properties
21    // May add the same node twice as it doesn't check for duplicates
22 }

```

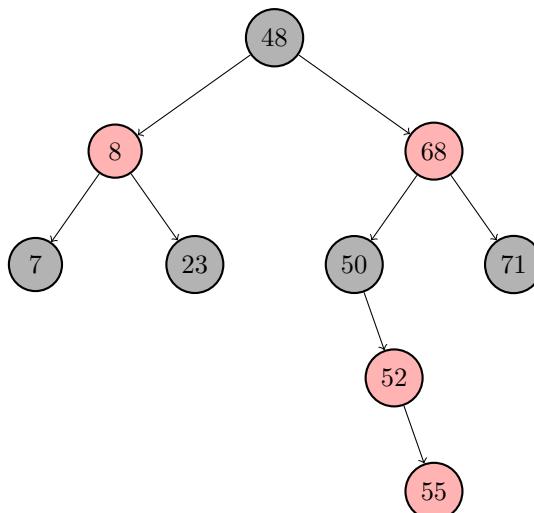
Einfügen funktioniert grundlegend gleich zu BST, allerdings wird am Ende die Farbe des neuen Knotens auf rot gesetzt und anschließend die Regeln des RBTs (falls verletzt) wieder hergestellt.



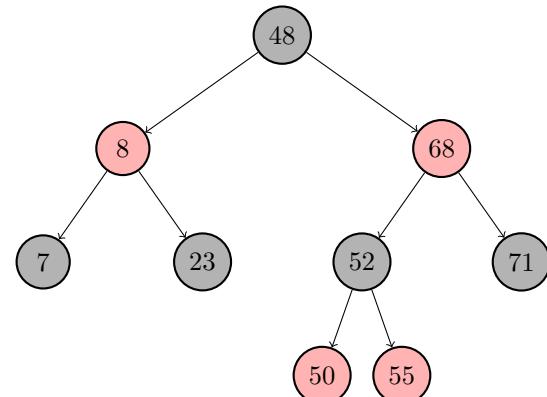
```

1 void fixColorsAfterInsertion(RBNode z) { // Omega(1), O(log n), Theta(log n)
2     while (z.parent.color == Color.RED) { // While z's parent is red
3         if (z.parent == z.parent.parent.left) { // If z's parent is a left child
4             RBNode y = z.parent.parent.right; // Gets sibling of z's parent
5             if (y != null && y.color == Color.RED) { // CASE 1: z's parent is a left child and
6                 sibling is red
7                 z.parent.color = Color.BLACK; // Set z's parent to black
8                 y.color = Color.BLACK; // Set z's uncle to black
9                 z.parent.parent.color = Color.RED; // Set z's grandparent to red
10                z = z.parent.parent; // Set z to z's grandparent
11            } else { // CASE 2: z's parent is a left child and sibling is black
12                if (z == z.parent.right) { // If z is a right child
13                    z = z.parent; // Set z to z's parent
14                    rotateLeft(z); // Rotate new z to left
15                }
16                z.parent.color = Color.BLACK; // Set z's parent to black
17                z.parent.parent.color = Color.RED; // Set z's grandparent to red
18                rotateRight(z.parent.parent); // Rotate z's grandparent to right
19            }
20        } else { // If z's parent is a right child
21            // Same as above but with right and left exchanged
22            RBNode y = z.parent.parent.left;
23            if (y != null && y.color == Color.RED) { // CASE 3: z's parent is a right child and
24                sibling is red
25                z.parent.color = Color.BLACK;
26                y.color = Color.BLACK;
27                z.parent.parent.color = Color.RED;
28                z = z.parent.parent;
29            } else { // CASE 4: z's parent is a right child and sibling is black
30                if (z == z.parent.left) {
31                    z = z.parent;
32                    rotateRight(z);
33                }
34                z.parent.color = Color.BLACK;
35                z.parent.parent.color = Color.RED;
36                rotateLeft(z.parent.parent);
37            }
38        }
39    }
40    root.color = Color.BLACK; // Set root to black, as it always should be
}

```



Fix needed at 55

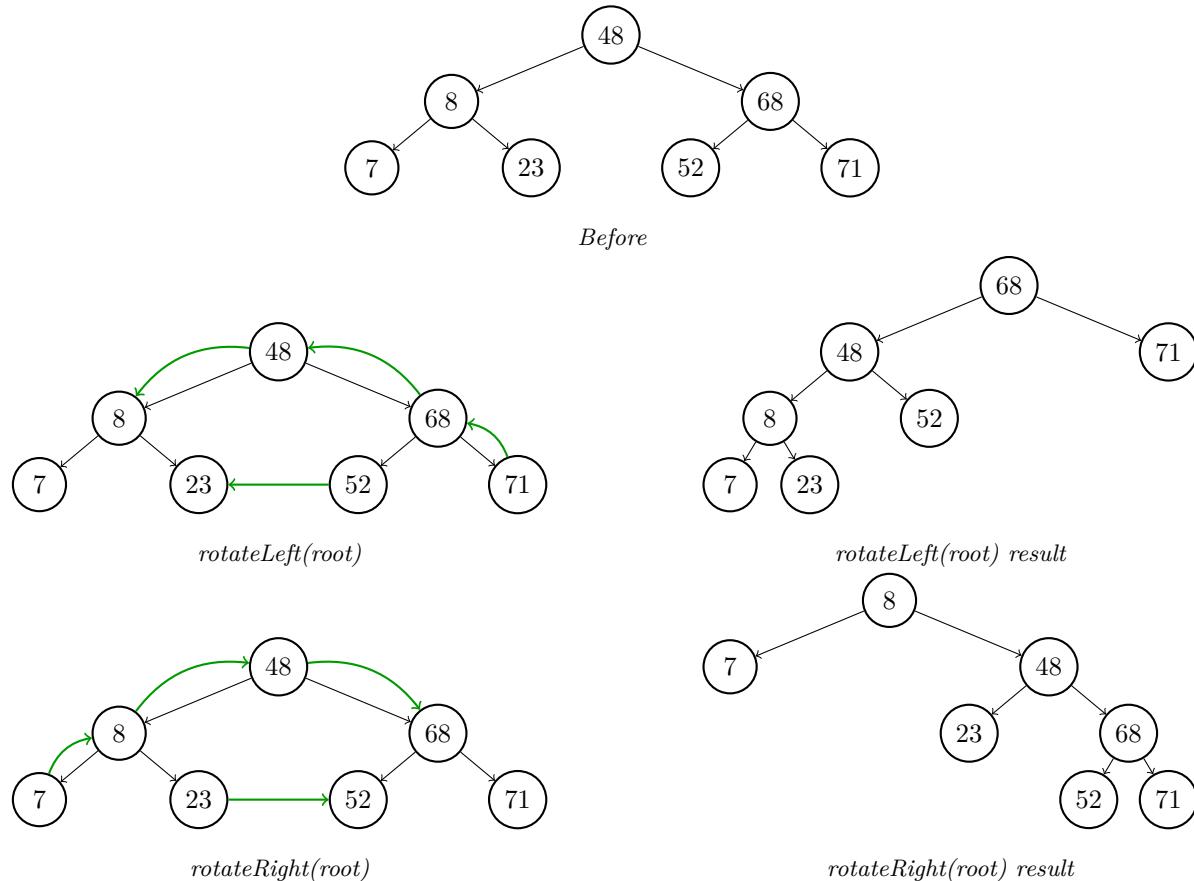


Fixed (Case 4)

Zum Wiederherstellen der RBT-Regeln muss der Baum an bestimmten Knoten rotiert werden.

```

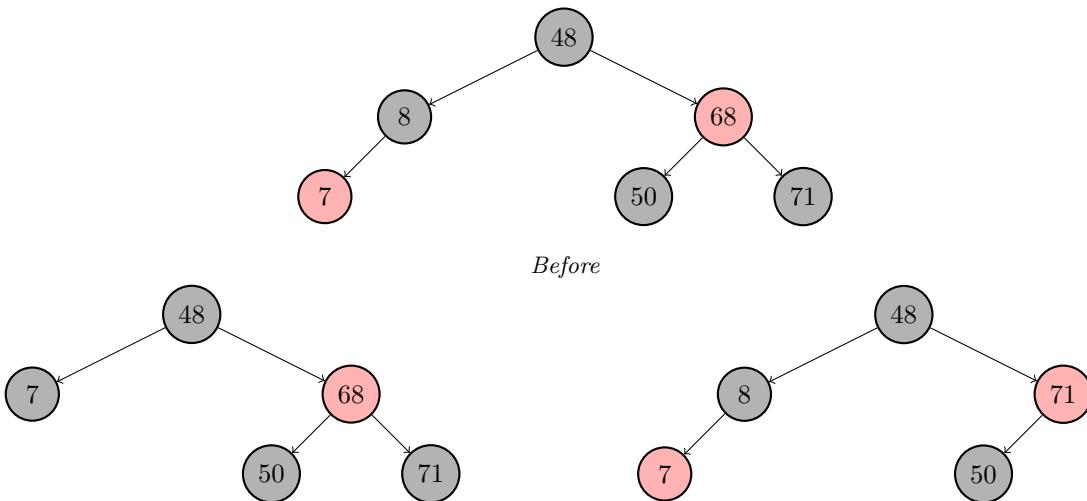
1 void rotateLeft(RBNode x) { // O(1)
2     RBNode y = x.right;
3     x.right = y.left; // Set x's right child to y's left child
4     if (y.left != null) // If y has a left child
5         y.left.parent = x; // Set y's left child's parent to x
6     y.parent = x.parent; // Set y's parent to x's parent
7     if (x.parent == sent) // If x is the root, set y to be the root
8         root = y;
9     else if (x == x.parent.left) // If x is a left child, set x's parent's left child to y
10        x.parent.left = y;
11    else // If x is a right child, set x's parent's right child to y
12        x.parent.right = y;
13    y.left = x; // Set y's left child to x
14    x.parent = y; // Set x's parent to y
15 }
16 void rotateRight(RBNode x) { // O(1)
17     // Same as rotateLeft but with right and left exchanged
18     RBNode y = x.left;
19     x.left = y.right;
20     if (y.right != null)
21         y.right.parent = x;
22     y.parent = x.parent;
23     if (x.parent == sent)
24         root = y;
25     else if (x == x.parent.right)
26         x.parent.right = y;
27     else
28         x.parent.left = y;
29     y.right = x;
30     x.parent = y;
31 }
```



```

1 void delete(RBNode z) { // Omega(1), O(log n), Theta(log n)
2     RBNode a = z.parent; // a represent node with black depth imbalance
3     int dbh = 0; // delta black height, -1 for right, 1 for left leaning
4     if (z.left == null && z.right == null) { // CASE 1: z is a leaf
5         if (z.color == Color.BLACK && z != root) { // If z is black
6             if (z == z.parent.left) // If z is a left child
7                 dbh = -1; // Set delta black height to -1
8             else // If z is a right child
9                 dbh = 1; // Set delta black height to 1
10        }
11        transplant(z, null); // Transplant null to zs parent
12    } else if (z.left == null) { // CASE 2: z only has a right child
13        RBNode y = z.right;
14        transplant(z, z.right);
15        y.color = z.color;
16    } else if (z.right == null) { // CASE 3: z only has a left child
17        RBNode y = z.left;
18        transplant(z, z.left);
19        y.color = z.color;
20    } else { // CASE 4: z has two children
21        RBNode y = z.right;
22        a = y;
23        boolean wentLeft = false;
24        while (y.left != null) { // find next biggest Node
25            a = y;
26            y = y.left;
27            wentLeft = true;
28        }
29        if (y.parent != z) { // If next biggest element is not child of z
30            transplant(y, y.right);
31            y.right = z.right;
32            y.right.parent = y;
33        }
34        transplant(z, y);
35        y.left = z.left;
36        y.left.parent = y;
37        if (y.color == Color.BLACK) {
38            if (wentLeft) // Tree imbalanced depending on y location
39                dbh = -1;
40            else
41                dbh = 1;
42        }
43        y.color = z.color;
44    }
45    if (dbh != 0) // If black height imbalance
46        fixColorsAfterDeletion(a, dbh);
47 }

```



```

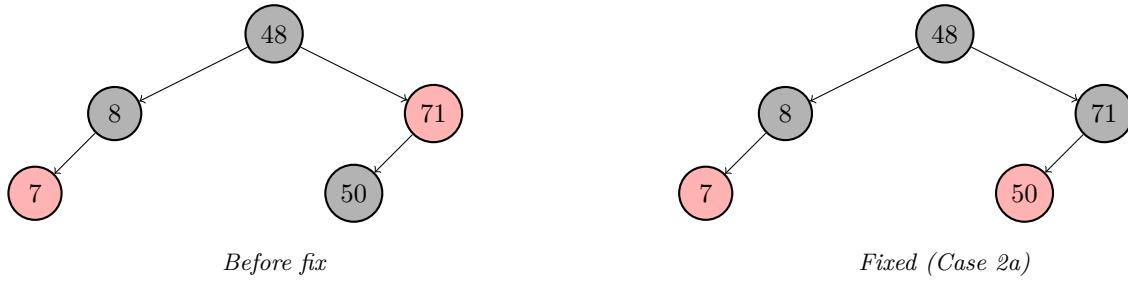
1 void fixColorsAfterDeletion(RBNode a, int dbh) { // Omega(1), O(log n), Theta(log n)
2     if (dbh == -1) { // Extra black node on the right
3         RBNode x = a.left;
4         RBNode b = a.right;
5         RBNode c = b.left; // Left child of right child of a
6         RBNode d = b.right; // Right child of right child of a
7         if (x != null && x.color == Color.RED) {
8             // Easy case: x is red
9             x.color = Color.BLACK;
10        } else if (a.color == Color.BLACK
11            && b.color == Color.RED) {
12            // Case 1: a black, b red
13            rotateLeft(a);
14            a.color = Color.RED;
15            b.color = Color.BLACK;
16            fixColorsAfterDeletion(a, dbh);
17        } else if (a.color == Color.RED
18            && b.color == Color.BLACK
19            && (c == null || c.color == Color.BLACK)
20            && (d == null || d.color == Color.BLACK)) {
21            // Case 2a: a red, b black, c and d black
22            a.color = Color.BLACK;
23            b.color = Color.RED;
24        } else if (a.color == Color.BLACK
25            && b.color == Color.BLACK
26            && (c == null || c.color == Color.BLACK)
27            && (d == null || d.color == Color.BLACK)) {
28            // Case 2b: a black, b black, c and d black
29            b.color = Color.RED;
30            if (a == a.parent.left)
31                dbh = 1;
32            else if (a == a.parent.right)
33                dbh = -1;
34            else
35                dbh = 0;
36            fixColorsAfterDeletion(a.parent, dbh);
37        } else if (b.color == Color.BLACK
38            && c != null && c.color == Color.RED
39            && (d == null || d.color == Color.BLACK)) {
40            // Case 3: a either, b black, c red, d black
41            rotateRight(b);
42            c.color = Color.BLACK;
43            fixColorsAfterDeletion(a, dbh);
44        } else if (b.color == Color.BLACK
45            && d != null && d.color == Color.RED) {
46            // Case 4: a either, b black, c either, d red
47            rotateLeft(a);
48            b.color = a.color;
49            a.color = Color.BLACK;
50            d.color = Color.BLACK;
51        }

```

```

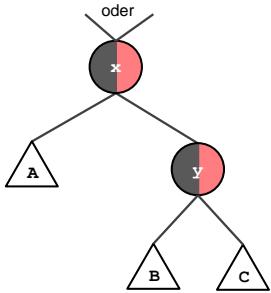
52     } else { // Extra black node on the left
53         // Same as above but with right and left exchanged
54         RBNNode x = a.right;
55         RBNNode b = a.left;
56         RBNNode c = b.right; // Right child of left child of a
57         RBNNode d = b.left; // Left child of left child of a
58         if (x != null && x.color == Color.RED) {
59             // Easy case: x is red
60             x.color = Color.BLACK;
61         } else if (a.color == Color.BLACK
62             && b.color == Color.RED) {
63             // Case 1: a black, b red
64             rotateRight(a);
65             a.color = Color.RED;
66             b.color = Color.BLACK;
67             fixColorsAfterDeletion(a, dbh);
68         } else if (a.color == Color.RED
69             && b.color == Color.BLACK
70             && (c == null || c.color == Color.BLACK)
71             && (d == null || d.color == Color.BLACK)) {
72             // Case 2a: a red, b black, c and d black
73             a.color = Color.BLACK;
74             b.color = Color.RED;
75         } else if (a.color == Color.BLACK
76             && b.color == Color.BLACK
77             && (c == null || c.color == Color.BLACK)
78             && (d == null || d.color == Color.BLACK)) {
79             // Case 2b: a black, b black, c and d black
80             b.color = Color.RED;
81             if (a == a.parent.right)
82                 dbh = 1;
83             else if (a == a.parent.left)
84                 dbh = -1;
85             else
86                 dbh = 0;
87             fixColorsAfterDeletion(a.parent, dbh);
88         } else if (b.color == Color.BLACK
89             && c != null && c.color == Color.RED
90             && (d == null || d.color == Color.BLACK)) {
91             // Case 3: a either, b black, c red, d black
92             rotateLeft(b);
93             c.color = Color.BLACK;
94             fixColorsAfterDeletion(a, dbh);
95         } else if (b.color == Color.BLACK
96             && d != null && d.color == Color.RED) {
97             // Case 4: a either, b black, c either, d red
98             rotateRight(a);
99             b.color = a.color;
100            a.color = Color.BLACK;
101            d.color = Color.BLACK;
102        }
103    }
104    // All cases except 2b mean end of the method in this or the next instance. 2b can go on tho.
105 }

```



```
1 void transplant(RBNode u, RBNode v) { // O(1)
2     // Transplants v to the position of u
3     if (u.parent == sent) // If u is the root, v becomes the new root
4         root = v;
5     else if (u == u.parent.left) // If u is a left child, v becomes a left child
6         u.parent.left = v;
7     else // If u is a right child, v becomes a right child
8         u.parent.right = v;
9     if (v != null) // If v is not null, v becomes a child of u's parent
10        v.parent = u.parent;
11    }
12 }
```

Rotation: Algorithmus (I)



```
rotateLeft(T,x) //x.right!=nil

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;
```

Laufzeit = $\Theta(1)$

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittene Datenstrukturen | 15

Aufräumen

```
fixColorsAfterInsertion(T,z)

1 WHILE z.parent.color==red DO
2   IF z.parent==z.parent.parent.left THEN
3     y=z.parent.parent.right;
4     IF y!=nil AND y.color==red THEN
5       z.parent.color=black;
6       y.color=black;
7       z.parent.parent.color=red;
8       z=z.parent.parent;
9     ELSE
10      IF z==z.parent.right THEN
11        z=z.parent;
12        rotateLeft(T,z);
13        z.parent.color=black;
14        z.parent.parent.color=red;
15        rotateRight(T,z.parent.parent);
16      ELSE
17        ... //exchange left and right
18 T.root.color=black;
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittene Datenstrukturen | 20

Einfügen

Funktioniert wie beim binären Suchbaum
(mit Sentinel)

Änderung:
Farbe des neuen Knoten auf rot setzen, dann RSB-Bedingung wieder herstellen

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

1 x=T.root; px=T.sent;
2 WHILE x != nil DO
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
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;
17 fixColorsAfterInsertion(T,z);
```

Löschen: Transplant mit Sentinels

```
transplant(T,u,v) //with Sent

1 IF u.parent==T.sent 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;
```

Zur Erinnerung:
funktioniert auch, wenn
 $v=nil$

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittene Datenstrukturen | 136

Löschen: Algorithmus (I)

```

delete(T, z)
1 a=z.parent; dsh=nil;
2 IF z.left==z.right==nil THEN // z leaf
3   IF z.color==black AND z!=T.root THEN
4     IF z.parent.left==z THEN dsh=right ELSE dsh=left;
5     transplant(T,z,nil);
6 ELSE IF z.left==nil THEN // z half leaf
7   y=z.right;
8   transplant(T,z,y);
9   y.color=z.color;
10 ELSE IF z.right==nil THEN // z half leaf
11   y=z.left;
12   transplant(T,z,y);
13   y.color=z.color;
14 ELSE ...
    
```

a Zeiger auf Knoten, in dem „tiefste“ Imbalance entstehen könnte
 $dsh = \Delta SH$ für Knoten a
 $(nil=0, left=-1, right=1)$

In den Fällen muss y.color==red sein, sonst SH-Regel verletzt; einfach y umhängen und Farbe von z kopieren

Löschen: Algorithmus (II)

```

...
14 ELSE // z has two children
15   y=z.right; a=y; wentleft=false;
16   WHILE y.left != nil DO
17     a=y; y=y.left; wentleft=true;
18   IF y.parent != z THEN
19     transplant(T,y,y.right);
20     y.right=z.right;
21     y.right.parent=y;
22     transplant(T,z,y);
23     y.left=z.left;
24     y.left.parent=y;
25   IF y.color==black THEN
26     IF wentleft THEN dsh=right ELSE dsh=left;
27     y.color=z.color;
28 IF dsh!=nil THEN fixColorsAfterDeletion(T,a,dsh);
    
```

Analog zu BST

Fallunterscheidung nach y rechtes oder linkes Kind

erzeugt imbalance, je nachdem, ob y rechtes oder linkes Kind war

Löschen: Farben korrigieren (I)

```

fixColorsAfterDeletion(T,a,dsh)
1 IF dsh==right THEN //extra black node on the right
2   x=a.left; b=a.right; c=b.left; d=b.right;
3   IF x!=nil AND x.color==red THEN
4     x.color=black;
5   ELSE IF a.color==black AND b.color==red THEN
6     rotateLeft(T,a);
7     a.color=red; b.color=black;
8     fixColorsAfterDeletion(T,a,dsh);
9   ELSE IF a.color==red AND b.color==black
10    AND (c==nil OR c.color==black)
11    AND (d==nil OR d.color==black) THEN
12     a.color=black; b.color=red;
13   ELSE IF ...
    
```

notwendig: b!=nil für dsh=right

einfacher Fall: x ist rot

Fall I: a schwarz, b rot

Fall IIa: a rot, b schwarz, c, d nicht rot

außer im Fall IIb führen rekursive Aufrufe im nächsten Schritt zum Rekursionsende

Löschen: Farben korrigieren (II)

```

fixColorsAfterDeletion(T,a,dsh)
...
11 ELSE IF a.color==black AND b.color==black
  AND (c==nil OR c.color==black)
  AND (d==nil OR d.color==black) THEN
12   b.color=red;
13   IF a==a.parent.left THEN dsh=left
14 ELSE IF a==a.parent.right THEN dsh=right ELSE dsh=nil;
15   fixColorsAfterDeletion(T,a.parent,dsh);
16 ELSE IF b.color==black AND c!=nil AND c.color==red
  AND (d==nil OR d.color==black) THEN
17   rotateRight(T,b);
18   c.color=black; b.color=black;
19   fixColorsAfterDeletion(T,a,dsh);
20 ELSE IF b.color==black AND d!=nil AND d.color==red THEN
21   rotateLeft(T,a);
22   b.color=a.color; a.color=black; d.color=black;
23 ELSE // dsh==left, extra black node on the left
24 ... //exchange left and right
    
```

Fall IIb: a, b schwarz, c, d nicht rot

Fall III: b schwarz, c rot, d nicht rot

Fall IV: b schwarz, c rot

Fall linkslastig

6.2 AVL Trees

Ein Adelson-Velsky Landis Tree ist ein BST, der sich selbst balanziert um eine Höhe von $h = \log n$ zu garantieren. Er wird so definiert, dass die Höhendifferenz von zwei Teilbäumen unter einem Knoten jeweils maximal 1 ist.

Im Vergleich zu RBT ist der AVLT in der Höhe strikter. So ist die maximale Höhe im AVL $h = 1.44 \cdot \log(n + 2)$, während er im RBT nur $2 \cdot \log n$. So haben AVLTs zwar die bessere Effizienz in Suchvorgängen, jedoch benötigen sie beim Insert oder Delete meist mehr Rotationen. Demnach bieten sich AVLTs eher bei Fällen an, wo mehr Suchvorgänge stattfinden im Vergleich zu den Insert/Delete Vorgängen. Muss der Baum oft modifiziert werden so bietet sich ein RBT besser an.

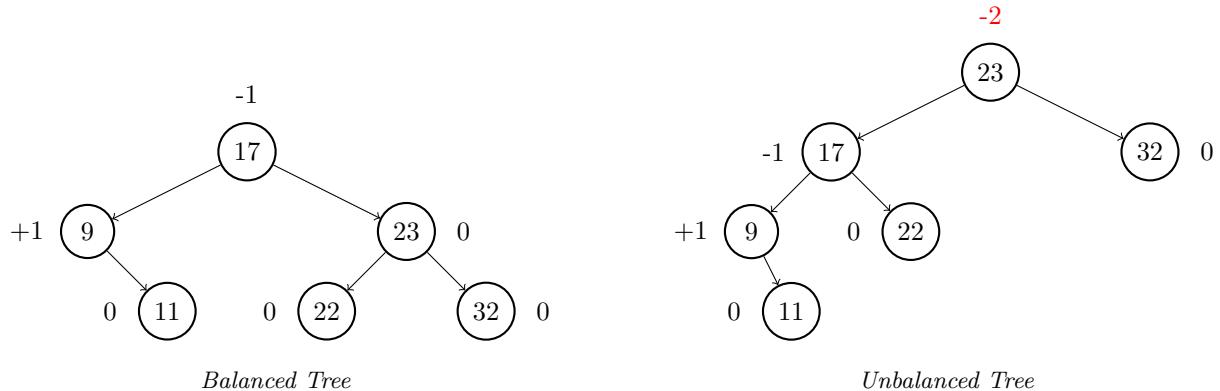
Die minimale Anzahl an Knoten in einem AVL-Tree mit Höhe h ergibt sich durch die rekursive Fibonacci-Gleichung: $N(h) = N(h - 1) + N(h - 2) + 1$ mit $N(0) = 1$ und $N(1) = 2$. (Root hat Maximalhöhe, Blätter sind $h = 0$)

```

1 class AVLTree {
2     class AVLNode {
3         Integer key;
4         int height;
5         AVLNode left;
6         AVLNode right;
7         AVLNode(Integer k) {
8             key = k;
9             height = 1;
10        }
11    }
12    AVLNode root;
13    // Search and traversal like in BST
14    int height(AVLNode n) {
15        return (n == null) ? -1 : n.height;
16    }
17    void updateHeight(AVLNode n) {
18        n.height = 1 + Math.max(height(n.left), height(n.right));
19    }
20    int getBalance(AVLNode n) {
21        return (n == null) ? 0 : height(n.right) - height(n.left);
22    }

```

Visuell gleich zu einem Best-Case ausgewogenen BST.



```

1 // Rotations work a bit different as in other Trees as we don't have parents.
2 // Does not update the tree itself -> just returns the new root of the rotated subtree
3 // Does not need to check for null reference as right.left/left.right respectively
4 // are always well defined when called in insert and delete
5 AVLNode rotateLeft(AVLNode x) { // O(1)
6     AVLNode y = x.right;
7     AVLNode z = y.left;
8     y.left = x;
9     x.right = z;
10    updateHeight(x);
11    updateHeight(y);
12    return y;
13 }
14 AVLNode rotateRight(AVLNode x) { // O(1)
15     AVLNode y = x.left;
16     AVLNode z = y.right;
17     y.right = x;
18     x.left = z;
19     updateHeight(x);
20     updateHeight(y);
21     return y;
22 }
```

Während die Function ein wenig abgeändert ist, ist das Endergebnis bei richtiger Anwendung (nicht inPlace) gleich.

```

1  AVLNode insert(AVLNode partRoot, int key) { // O(h) = O(log(n))
2      if (partRoot == null) // Found insertion point
3          return new AVLNode(key);
4      else if (key < partRoot.key) // If node smaller than partRoot -> go left
5          partRoot.left = insert(partRoot.left, key);
6      else if (key > partRoot.key) // If node bigger than partRoot -> go right
7          partRoot.right = insert(partRoot.right, key);
8      else // If node == partRoot -> throw exception
9          throw new UException("Duplicate node");
10     return fixBalance(partRoot);
11 }
12 // Rebalance every node above the inserted node.

```

Funktioniert prinzipiell gleich zu den anderen Trees, aber ist nicht in-place und muss zusätzlich noch alle subtrees balanzieren.

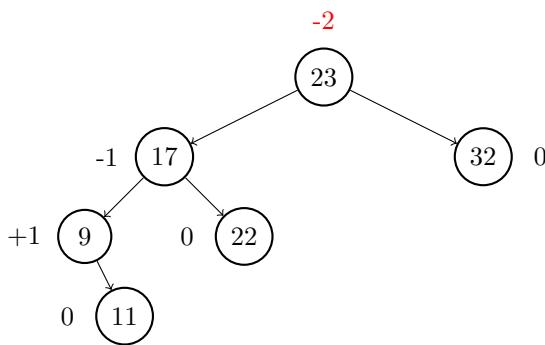
```

1  AVLNode delete(AVLNode node, int key) { // O(h) = O(log(n))
2      if (node == null) { // Node not found
3          return node;
4      } else if (key < node.key) {
5          node.left = delete(node.left, key);
6      } else if (key > node.key) {
7          node.right = delete(node.right, key);
8      } else { // Node found -> Commence deletion
9          if (node.left == null || node.right == null) // If half leaf or leaf
10             node = (node.left == null) ? node.right : node.left;
11         else { // If complete node
12             AVLNode next = node.right;
13             while (next.left != null) {
14                 next = next.left;
15             }
16             node.key = next.key;
17             node.right = delete(next, next.key);
18         }
19     }
20     return fixBalance(node); // Rebalance every node above the deleted node
21 }
```

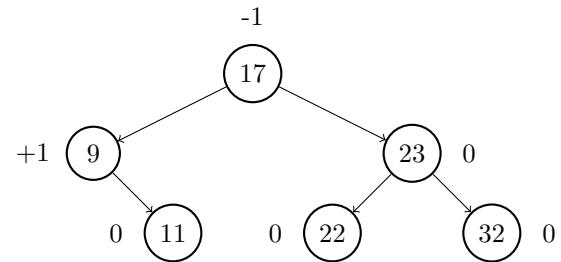
Wie bei rotate und insert, prinzipiell gleich zu den anderen Trees, aber ist nicht in-place und muss zusätzlich alle subtrees balanzieren.

```

1  AVLNode fixBalance(AVLNode z) { // O(1)
2      updateHeight(z);
3      int balance = getBalance(z);
4      if (balance > 1) { // If right heavy
5          if (height(z.right.right) > height(z.right.left)) {
6              z = rotateLeft(z);
7          } else {
8              z.right = rotateRight(z.right);
9              z = rotateLeft(z);
10         }
11     } else if (balance < -1) { // If left heavy
12         if (height(z.left.left) > height(z.left.right)) {
13             z = rotateRight(z);
14         } else {
15             z.left = rotateLeft(z.left);
16             z = rotateRight(z);
17         }
18     }
19     return z;
20 }
21 }
```



Unbalanced Tree after insert of 11



Balanced Tree after fixup

Einfügen: Laufzeit

Gesamtlaufzeit $O(h) = O(\log n)$

Laufzeit = $O(h)$

Laufzeit = $O(h)$,
da Suche nach
unbalanciertem Knoten
Richtung Wurzel in $O(h)$,
und Rebalancieren
nur einmal nötig

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

1 x=T.root; px=T.sent;
2 WHILE x != nil DO
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
10        T.root=z
11 ELSE
12     IF px.key > z.key THEN
13         px.left=z
14     ELSE
15         px.right=z;
16 fixBalanceAfterInsertion(T, z);
```

6.3 Splay Trees

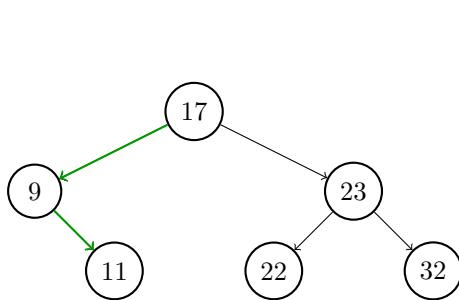
Splay Trees sind BSTs, die sich mit jedem Aufruf neu reorganisieren. Dies tun sie indem sie das betrachtete Element an die Wurzel verschieben. So ist der Splay Tree nicht unbedingt wie RBT und AVLT gut balanziert, jedoch besonders effektiv, wenn einige Elemente öfters gesucht werden als andere. Da diese Bäume sich nicht wirklich selbst balanzieren, sind Splay Trees ungeeignet für Fälle wo alle Werte ungefähr gleich viel gesucht werden, da im Durchschnitt mehr Zeit gebraucht wird ein Wert zu finden und diesen an die Wurzel zu verschieben als in anderen Trees.

```
1 class SplayTree extends BSTree {
2     //Same Nodes as BSTree
3     void splay(BSTNode z) { // O(h)
4         while(z != root) {
5             if (z.parent.parent == null) // If father is root
6                 zig(z);
7             else {
8                 if (z == z.parent.parent.left.left || z == z.parent.parent.right.right)
9                     zigZig(z);
10                else
11                    zigZag(z);
12            }
13        }
14    }
15    void zig(BSTNode z) { // O(1)
16        if (z == z.parent.left)
17            rotateRight(z.parent);
18        else
19            rotateLeft(z.parent);
20    }
21    void zigZig(BSTNode z) { // O(1)
22        if (z == z.parent.left) {
23            rotateRight(z.parent.parent);
24            rotateRight(z.parent);
25        } else {
26            rotateLeft(z.parent.parent);
27            rotateLeft(z.parent);
28        }
29    }
30    void zigZag(BSTNode z) { // O(1)
31        if (z == z.parent.left) {
32            rotateRight(z.parent);
33            rotateLeft(z.parent.parent);
34        } else {
35            rotateLeft(z.parent);
36            rotateRight(z.parent.parent);
37        }
38    }
39    //Rotate works the same as in RBTree
```

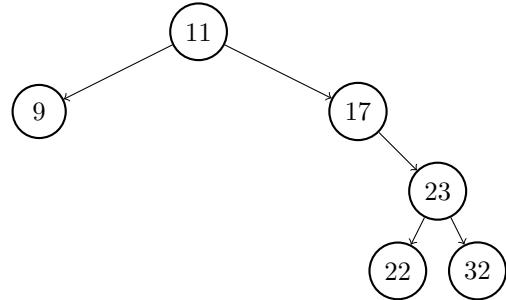
Sieht visuell praktisch wie ein normaler BST aus.

```

1  BSTNode search(int key) { // O(h), like BSTree, with additional splay
2      BSTNode x = root;
3      while(x != null && x.key != key) {
4          if (key < x.key)
5              x = x.left;
6          else
7              x = x.right;
8      }
9      if (x == null)
10         return null;
11     splay(x);
12     return root; // After splay the root is the searched node
13 }
```



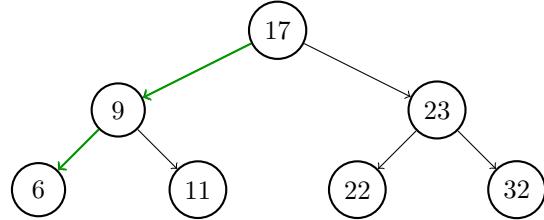
Search for 11



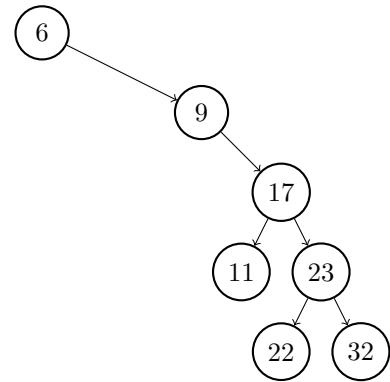
Splay 11 (zigzag)

```

1 void insert (BSTNode z) { // O(h)
2     super.insert(z); // Inserts node using BSTrees insert
3     splay(z); // Splays node to the root
4 }
```



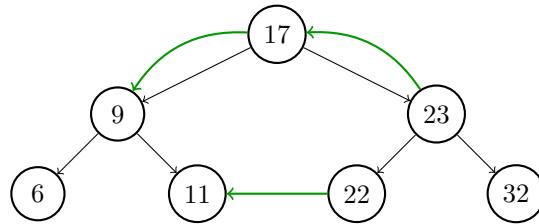
Insert 6



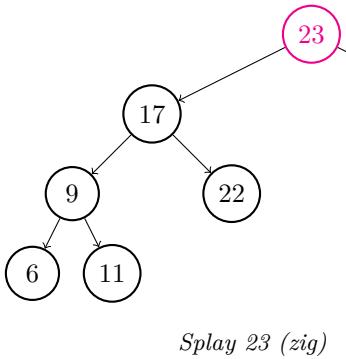
Splay 6 (zigzag)

```

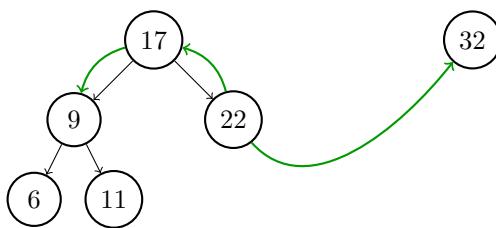
1 void delete (BSTNode z) { // O(h)
2     splay(z);
3     BSTNode r = root.right; // Save right child
4     BSTNode biggestL = root.left; // Save left child
5     root.right.parent = null; // Remove parent reference of right child of root
6     root.left.parent = null; // Remove parent reference of left child of root
7     root = biggestL; // Set root to left child
8     while (biggestL.right != null) // Get biggest node in left subtree
9         biggestL = biggestL.right;
10    splay(biggestL); // Splay biggest node -> becomes root
11    root.right = r; // Put right subtree back in place
12    r.parent = root; // Change parent of right subtree root
13 }
14 }
```



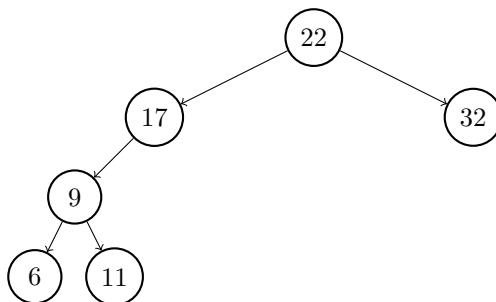
Before delete 23



Splay 23 (zig)



delete 23



Splay biggest node in left subtree (22) and append right subtree

Splay-Operation

Gesamlaufzeit $O(h)$

Laufzeit:
Bei jeder Iteration wird z mindestens einen Level nach oben rotiert

zigZig(T, z)

```

1 IF z==z.parent.left THEN
2   rotateRight(T,z.parent.parent);
3   rotateRight(T,z.parent);
4 ELSE
5   rotateLeft(T,z.parent.parent);
6   rotateLeft(T,z.parent);
    
```

zig und
zigZag analog

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittene Datenstrukturen | 75



Suchen

search(T, k)

Alternativ: „splay“ dann letzten besuchten Knoten bei erfolgloser Suche nach oben

Laufzeit $O(h)$

```

1 x=T.root;
2 WHILE x != nil AND x.key != k DO
3   IF x.key < k THEN
4     x=x.right
5   ELSE
6     x=x.left;
7   IF x==nil THEN
8     return nil
9   ELSE
10    splay(T,x);
11    return T.root;
    
```

Gesamlaufzeit $O(h)$

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittene Datenstrukturen | 79



6.4 Binary Heap Trees

Im Allgemeinen ist ein Binary Heap wie folgt definiert:

- Bis auf das unterste Level ist der Baum vollständig gefüllt und im untersten Level ist er von links befüllt.
- $\forall x \neq \text{root} : x.\text{parent}.key \geq x.key$

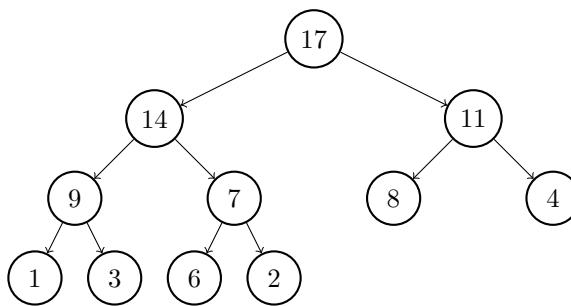
Binary Heaps unterscheiden sich von den anderen hier behandelten Trees insofern, dass sie keine BSTs sind. Das heißt, dass rechte Kinder eines Knotens nicht unbedingt größer als dieser sind und linke Kinder dieses Knotens nicht unbedingt kleiner. Sie sind so organisiert, dass sie Werte nach Ebenen sortieren. Bei Max heaps zum Beispiel steht der größte Wert in der Wurzel und der kleinste Wert irgendwo in der untersten Ebene. So ergibt sich bei Max heaps also die Eigenschaft, dass die parent node einer node immer größer ist als die node selber. Dies erlaubt einen sehr schnellen Zugriff auf das größte Element. Diese Eigenschaft kann sehr gut genutzt werden um Werte zu sortieren. Zudem sind Binary Heaps anders konstruiert als andere Trees, sie besitzen nämlich keine Nodes perse, sondern sind nur über Positionen in einem array gespeichert. Die Beziehungen zwischen den Nodes ergeben sich durch Formeln:

- **Parent:** $\text{parent}(i) = \lceil i/2 \rceil - 1$
- **Left Child:** $\text{left}(i) = 2 \cdot (i + 1) - 1$
- **Right Child:** $\text{right}(i) = 2 \cdot (i + 1)$

```

1 class BinaryMaxHeap {
2     Integer[] heap;
3     int size; // Size used to insert new elements at the first empty index
4     BinaryMaxHeap(int n) { // Creates new heap with size n
5         heap = new Integer[n];
6         size = 0;
7     }
8     BinaryMaxHeap(Integer[] arr) { // Creates heap from array
9         arrayToHeap(arr);
10    }
11    void arrayToHeap(Integer[] arr) { // Used to transfer array to heap
12        heap = arr.clone();
13        size = 0;
14        for (Integer i: arr) {
15            if (i == null) break;
16            size++;
17        }
18    }
19    int parent(int i) {
20        return (int) Math.ceil((double) i / 2) - 1;
21    }
22    int left(int i) {
23        return 2 * (i + 1) - 1;
24    }
25    int right(int i) {
26        return 2 * (i + 1);
27    }
28    boolean isEmpty() {
29        return size == 0;
30    }

```



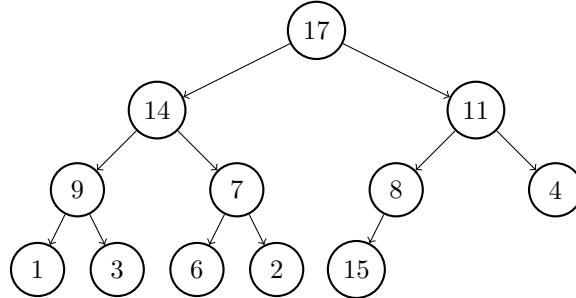
0	1	2	3	4	5	6	7	8	9	10	...
17	14	11	9	7	8	4	1	3	6	2	nil

```

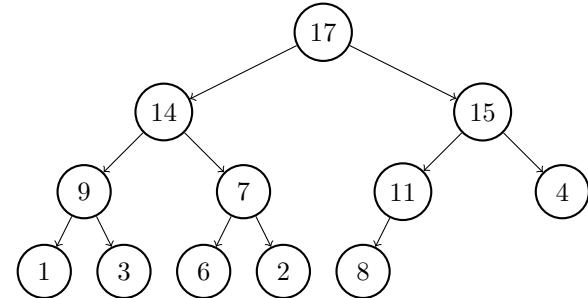
1 void insert(int k) { // O(h) = O(log n)
2     size++;
3     heap[size - 1] = k; // Add new element at first empty index
4     int i = size - 1;
5     while (i > 0 && heap[i] > heap[parent(i)]) { // Moves upward through tree
6         int temp = heap[i];
7         heap[i] = heap[parent(i)];
8         heap[parent(i)] = temp; // Swap elements if child is bigger than parent
9         i = parent(i); // Move up
10    }
11 }

```

Für min Heaps muss lediglich das Ungleichheitszeichen umgedreht werden.



Insert 15, before fixup



After fixup

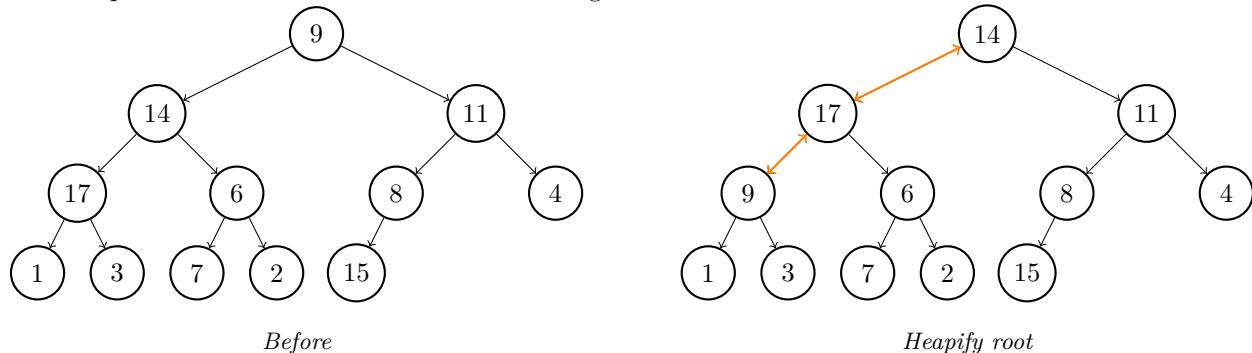
```

1 void heapify(int i) { // O(h) = O(log n)
2     // Used if heap is new unsorted array.
3     // If heap was build using insert, this should not be necessary
4     int max = i;
5     int l = left(i);
6     int r = right(i);
7     if(l < size && heap[i] < heap[l]) // If left child is bigger than i
8         max = l; // Set max to left
9     if(r < size && heap[max] < heap[r]) // if right child is bigger than max (i or left)
10        max = r; // Set max to right
11    if(max != i) { // If max is not i -> max is left or right
12        int temp = heap[i];
13        heap[i] = heap[max];
14        heap[max] = temp;
15        // Swap i and max
16        heapify(max); // max is now i
17        // Move down the tree
18    }
19}

```

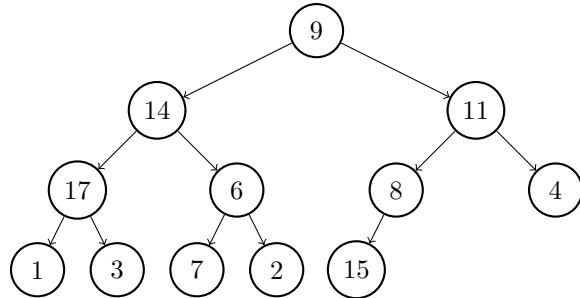
Tut essenziell das selbe wie der zweite Teil von Insert, nur in umgekehrter Reihenfolge (oben nach unten) und rekursiv.

Für Min heap muss wieder das Gleichheitszeichen umgedreht werden.

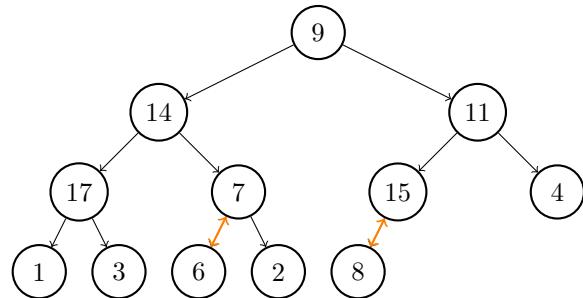


```

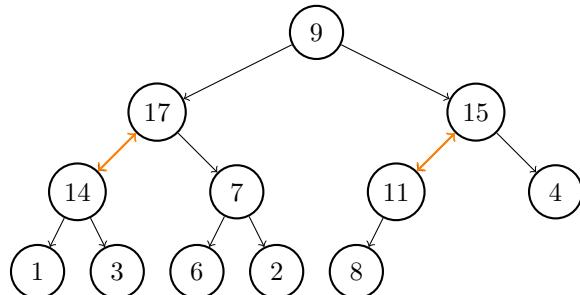
1 void buildHeap() { // O(nh) = O(n log n)
2     for(int i = parent(size - 1); i >= 0; i--)
3         heapify(i); // Calls heapify on each node starting from the second to last row
4     // Heap should now be sorted accordingly -> biggest node at root
5 }
6 int[] heapSort() { // O(nh) = O(n log n)
7     // Assumes new array for heap
8     buildHeap(); // Sorts heap accordingly
9     int[] res = new int[size]; // Create new array for sorted result
10    int i = 0;
11    while(!isEmpty())
12        res[i++] = extractMax(); // Extract max and add to array
13    return res; // Array is now sorted in reverse natural order
14 }
15 int extractMax() { // O(h) = O(log n)
16     if(isEmpty())
17         throw new UException("Underflow");
18     int max = heap[0]; // Biggest value at root
19     heap[0] = heap[size - 1]; // Sets root to last element
20     size--; // Decrease size
21     heapify(0); // Heapify new root
22     return max;
23 }
24 }
```



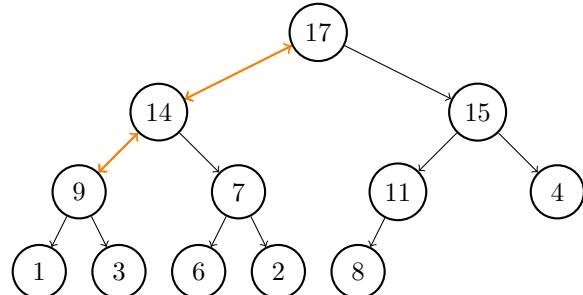
Before heapsort



heapify second to last level



heapify third to last level

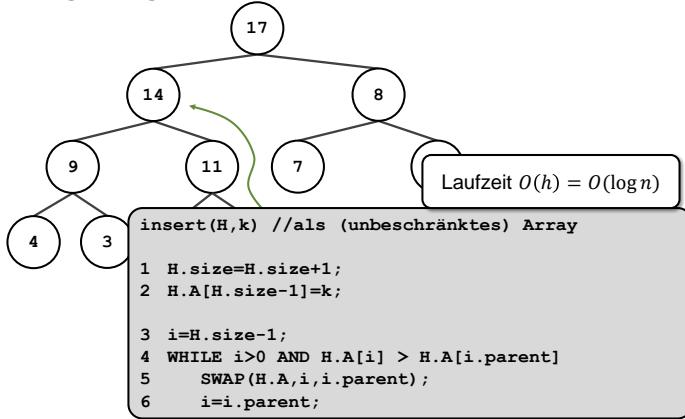


heapify last level

0	1	2	3	4	5	6	7	8	9	10	11
17	15	14	11	9	8	7	6	4	3	2	1

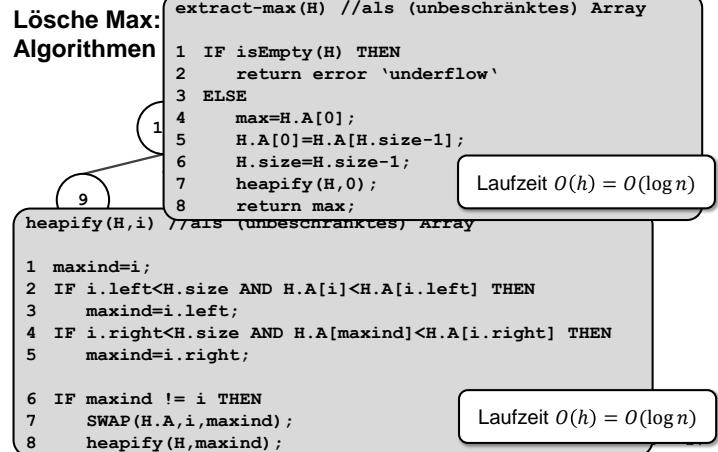
Extracted array

Einfügen: Algorithmus



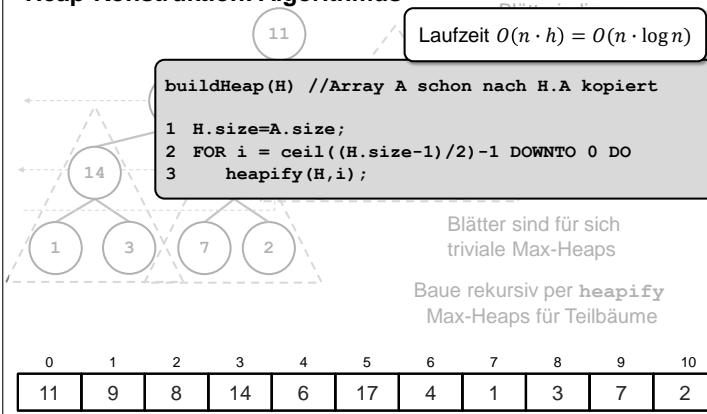
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 94

Lösche Max: Algorithmen



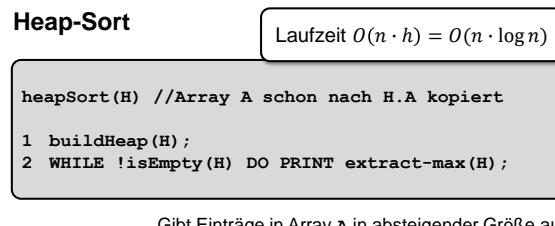
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 96

Heap-Konstruktion: Algorithmus



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 99

Heap-Sort



Gibt Einträge in Array A in absteigender Größe aus

Alternativ: speichere in jeder WHILE-Iteration
 $\max=extract\text{-}max(H)$ in $H.A[H.size]=\max$,
um sortierte Liste am Ende aufsteigend im Array A zu haben.

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 104

6.5 B-Tree

B-Trees (B hat keine festgelegte Bedeutung) unterscheiden sich sehr von den anderen Trees. Erstens sind sie keine Binary Trees. Ein B-Tree vom Grad t wird so definiert, dass

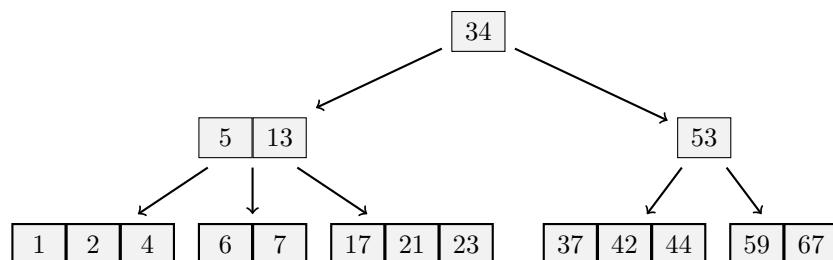
- Jede Node besitzt minimal $t - 1$ Werte und t Kinder (Außer root)
- Jede Node besitzt maximal $2 \cdot t - 1$ Werte und $2 \cdot t$ Kinder
- Die Werte innerhalb eines Knotens sind aufsteigend sortiert
- Alle Blätter haben die gleiche Höhe
- Jeder innere Knoten mit m Werten hat $m + 1$ Kinder
⇒ für alle Werte k_j in j-ten Kind gilt: $k_0 \leq key[0] \leq k_1 \leq key[1] \leq \dots \leq key[m - 1] \leq k_m$

Ein B-Tree kann somit in einem Knoten mehr als einen Wert und mehr als zwei Kinder besitzen. B-Trees balanzieren sich zudem auch selber, wodurch sie sehr effektiv Operationen in logarithmischer Laufzeit ausführen. Zudem ist durch die Anzahl der Werte die in einem Knoten gespeichert werden können und die Anzahl an Kinder die ein Knoten haben kann die Höhe des Baumes deutlich niedriger. B-Trees bieten sich so sehr für Disk-Based Operationen an, da sie die Anzahl an Disk-Access reduziert im Vergleich zu anderen Trees. Sie bieten sich also besonders für sehr große Datenbanken auf Festplatten an, sind aber im Vergleich zu den anderen Trees bei kleineren Eingaben weniger effizient. Die maximale Höhe von B Trees ist $h = \leq \log_t(\frac{n+1}{2})$.

```

1 class BTree {
2     class BNode {
3         int[] keys; // array of all keys in the node
4         int t; // degree, defines the maximum number of keys in the node
5         BNode[] children; // array of children to the node
6         int n; // current number of keys in the node, used to find the first free index
7         boolean isLeaf; // true if the node is a leaf node -> no children
8     }
9     BNode root;
10    final int t;
11    BTree(int t) {
12        root = null;
13        this.t = t;
14    }

```



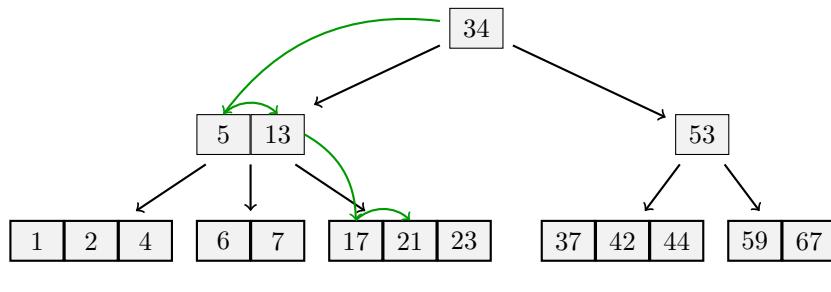
B-Tree example (Degree $t = 2$)

```

1 class BTTree {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        BNode search(int k) {
12            int i = findKey(k); // find key
13            if (inNode(i)) // If k in node
14                return this;
15            if (isLeaf) // If k not in node and node is leaf, k doesn't exist in tree
16                return null;
17            return children[i].search(k); // search for k in corresponding child
18        }
19    }
20    void search (int k) {
21        if (root == null) {
22            System.out.println("Tree is empty");
23            return;
24        }
25        root.search(k); // Search for k
26    }
27 }

```

Der Suchalgorithmus ist relativ simpel. Er durchläuft die key-Werte der Wurzel, bis es beim Element angelangt, das größer oder gleich dem Suchwert ist. Ist der Wert gleich, so gibt es den Knoten zurück. Andernfalls, wenn der Knoten keine Kinder hat, existiert der Wert nicht, ansonsten durchsucht der Algorithmus das Kind, das den Wert beinhalten sollte rekursiv.



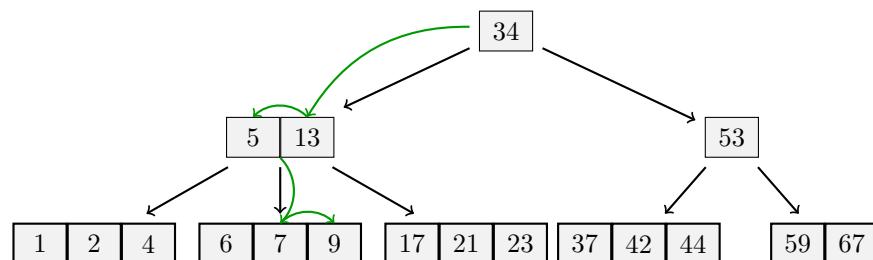
```

1 class BTREE {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        void insertNonFull(int k) { // Omega(1), O(t), Theta(t)
12            int i = n - 1; // first free index
13            if (isLeaf) { // If node has no children
14                while (i >= 0 && keys[i] > k) {
15                    keys[i + 1] = keys[i];
16                    i--;
17                } // Moves through the array and moves elements bigger than k to the right
18                // Creates insertion point
19                keys[i + 1] = k; // Inserts k
20                n++; // Increases number of keys
21            } else { // If node has children
22                while (i >= 0 && keys[i] > k) i--; // Moves through array and finds insertion point
23                if (children[i + 1].n == 2 * t - 1) { // If insertion child is full
24                    splitChild(i + 1, children[i + 1]); // Splits insertion child
25                    if (keys[i + 1] < k) // If key at insertion point is smaller than k
26                        i++; // Move insertion point one to the right
27                }
28                children[i + 1].insertNonFull(k); // Insert k into insertion child
29            }
30        }
31        void splitChild(int i, BNode y) {
32            BNode z = new BNode(y.t, y.isLeaf); // Creates new node akin to y
33            z.n = t - 1; // Has half the number of keys as the node to be split
34            for (int j = 0; j < t - 1; j++) // Copies the second half of the keys to the new node
35                z.keys[j] = y.keys[j + t];
36            if (!y.isLeaf) { // If y has children
37                for (int j = 0; j < t; j++) // Copies the second half of the children to the new node
38                    z.children[j] = y.children[j + t];
39            }
40            y.n = t - 1; // Node now has half the keys it had before
41            for (int j = n; j > i; j--) // Searches for insertion point of new child
42                children[j + 1] = children[j];
43            children[i + 1] = z; // Inserts new child
44            for (int j = n - 1; j >= i; j--) // Creates insertion point for new key
45                keys[j + 1] = keys[j];
46            keys[i] = y.keys[t - 1]; // Inserts new key
47            n++; // Increases number of keys
48        }
49    }
50    void insert(int k) {
51        if (root == null) { // If tree is empty
52            root = new BNode(t, true); // Create root
53            root.keys[0] = k; // Insert k
54            root.n = 1; // Increase number of keys
55        } else { // If tree is not empty
56            if (root.n == 2 * t - 1) { // If root is full
57                BNode s = new BNode(t, false); // Create new node
58                s.children[0] = root; // set root as first child of new node
59                s.splitChild(0, root); // Split root
60                int i = 0;
61                if (s.keys[0] < k) i++; // If only key in new node is smaller than k, move insertion
point
62                s.children[i].insertNonFull(k); // Insert k in corresponding child
63                root = s; // sets new node as root
64            } else // If root is not full
65                root.insertNonFull(k); // Simply insert
66        }
67    }
68 }

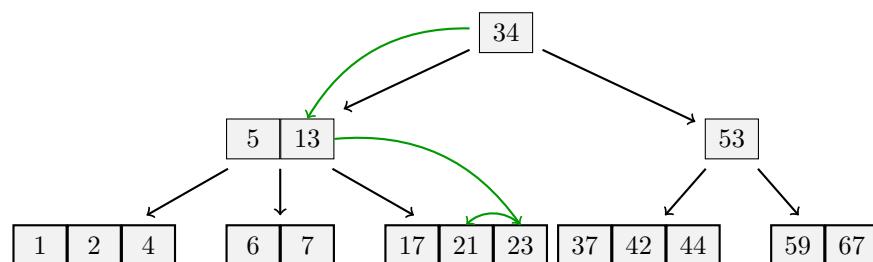
```

Prinzipiell folgt dieser Algorithmus einfach der Folge:

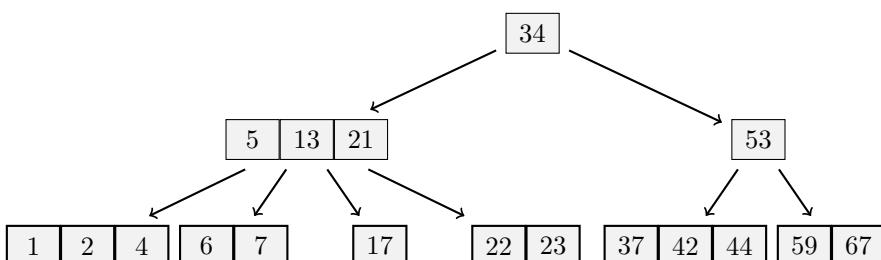
1. Finde Einfügepunkt
2. Wenn Node schon $2 \cdot t - 1$ Werte besitzt, splitte es
 - Teile Node in zwei Nodes mit je $t - 1$ Werten
 - Der mittlere Knoten wird in den Elternknoten eingefügt
 - Wenn dadurch der Elternknoten $2 \cdot t - 1$ Werte besitzt, splitte diesen rekursiv nach oben
3. Wert am Einfügepunkt einfügen



Insert 9 (Simple Case)



Before Insert 22 (Needs splitting)



Insert 22

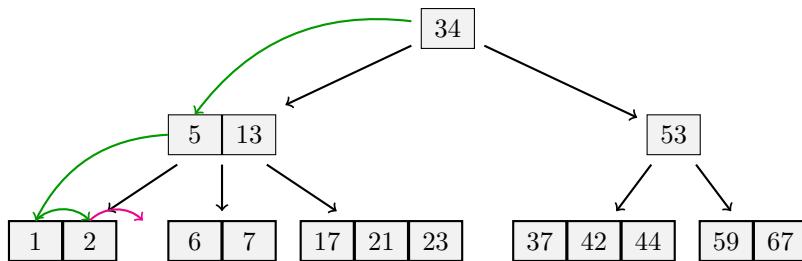
```

1 class BTREE {
2     class BNode {
3         int findKey(int k) { // Finds index of key in node, i = n if not in node
4             int i = 0;
5             while (i < n && keys[i] < k) i++;
6             return i;
7         }
8         boolean inNode(int i) {
9             return (i < n && keys[i] == 0);
10        }
11        int getPredecessor(int i) {
12            BNode child = children[i];
13            while (!child.isLeaf)
14                child = child.children[child.n];
15            return child.keys[child.n - 1];
16        }
17        int getSuccessor(int i) {
18            BNode child = children[i + 1];
19            while (!child.isLeaf)
20                child = child.children[0];
21            return child.keys[0];
22        }
23        void fill(int i) {
24            if (i != 0 && children[i - 1].n >= t) // If the previous child is filled more than half
25                borrowFromPrev(i);
26            else if (i != n && children[i + 1].n >= t) // If the next child is filled more than half
27                borrowFromNext(i);
28            else // If both children are not filled more than half
29                if (i != n) // Merge with next child
30                    merge(i);
31                else // Merge with previous child
32                    merge(i - 1);
33        }
34        void borrowFromPrev(int i) {
35            BNode child = children[i];
36            BNode sibling = children[i - 1]; // Get previous child, to be borrowed from
37            for (int j = child.n - 1; j >= 0; j--) // Move keys to the right
38                child.keys[j + 1] = child.keys[j];
39
40            if (!child.isLeaf)
41                for (int j = child.n; j >= 0; j--) // Move children to the right
42                    child.children[j + 1] = child.children[j];
43            child.keys[0] = keys[i - 1]; // Move key from parent to child
44            if (!child.isLeaf) // Move children if not child not a leaf
45                child.children[0] = sibling.children[sibling.n];
46            keys[i - 1] = sibling.keys[sibling.n - 1]; // Move key from sibling to parent
47            child.n++; // Increase number of keys in child
48            sibling.n--; // Decrease number of keys in sibling
49        }
50        void borrowFromNext(int i) {
51            BNode child = children[i];
52            BNode sibling = children[i + 1]; // Get next child, to be borrowed from
53            child.keys[child.n] = keys[i]; // Move key from parent to child
54            if (!child.isLeaf) // if child isn't a leaf move last first child of sibling to child
55                child.children[child.n + 1] = sibling.children[0];
56            keys[i] = sibling.keys[0]; // Move key from sibling to parent
57            for (int j = 1; j < sibling.n; j++) // Move keys to the left
58                sibling.keys[j - 1] = sibling.keys[j];
59            if (!sibling.isLeaf)
60                for (int j = 1; j <= sibling.n; j++) // Move children to the left
61                    sibling.children[j - 1] = sibling.children[j];
62            child.n++; // Increase number of keys in child
63            sibling.n--; // Decrease number of keys in sibling
64        }
65        void merge(int i) {
66            BNode child = children[i];
67            BNode sibling = children[i + 1];
68            child.keys[t - 1] = keys[i]; // Move key from parent to child
69            for (int j = 0; j < sibling.n; j++) // Move keys from sibling to second half of child
70                child.keys[j + t] = sibling.keys[j];
    }
}

```

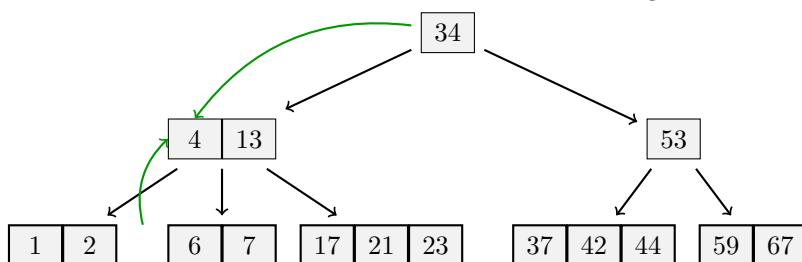
```

71         if (!child.isLeaf)
72             for (int j = 0; j <= sibling.n; j++) // Move children from sibling to second half of
73             child.children[j + t] = sibling.children[j];
74         for (int j = i + 1; j < n; j++) // Move keys to fill the gap created by moving to child
75             keys[j - 1] = keys[j];
76         for (int j = i + 2; j <= n; j--) // Move children to fill the gap created by moving to child
77             children[j - 1] = children[j];
78         child.n += sibling.n + 1;
79         n--;
80     }
81     void delete(int k) {
82         int i = findKey(k);
83         if (inNode(i)) {
84             if (isLeaf)
85                 deleteFromLeaf(i);
86             else
87                 deleteFromNonLeaf(i);
88         } else {
89             if (isLeaf)
90                 System.out.println("Key not found");
91             return;
92         }
93         boolean flag = (i == n); // if key is present in subtree of last child
94         if (children[i].n < t) // If child that contains key has less than t keys
95             fill(i); // Fill that child
96         if (flag && i > n) // If key is present in subtree of last child
97             children[i - 1].delete(k); // Delete from that subtree
98         else // If key is not present in subtree of last child
99             children[i].delete(k); // Delete from that subtree
100     }
101 }
102 void deleteFromLeaf (int i) {
103     for (int j = i + 1; j < n; j++) // Move all keys after i to the left
104         keys[j - 1] = keys[j];
105     n--; // reduce number of keys
106 }
107 void deleteFromNonLeaf (int i) {
108     int k = keys[i];
109     if (children[i].n >= t) { // If child that contains key has more than t keys
110         int pred = getPredecessor(i); // Get predecessor
111         keys[i] = pred; // Replace key with predecessor
112         children[i].delete(pred); // Delete predecessor from child
113     } else if (children[i + 1].n >= t) { // If child that contains key has more than t keys
114         int succ = getSuccessor(i); // Get successor
115         keys[i] = succ; // Replace key with successor
116         children[i + 1].delete(succ); // Delete successor from child
117     } else { // If both children have less than t keys
118         merge(i); // Merge children
119         children[i].delete(k); // Delete key from child
120     }
121 }
122 }
123 void delete(int k) {
124     if (root == null) {
125         System.out.println("Tree is empty");
126         return;
127     }
128     root.delete(k); // Delete k
129     if (root.n == 0) { // If root is empty
130         if (root.isLeaf) // If root is leaf -> tree is empty
131             root = null; // Delete root
132         else
133             root = root.children[0]; // Replace root with its child
134     }
135 }
136 }
```



Delete 4 (Simple Case)

Wenn der Knoten in einem Blatt steht kann er einfach rausgenommen werden.



Delete 5 (Replace with predecessor in child)

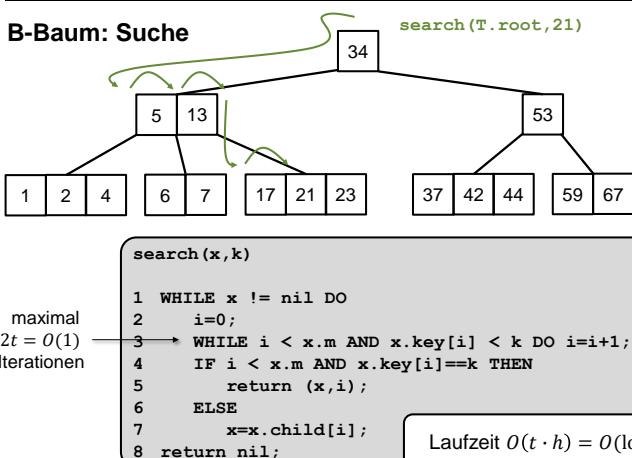
Im Fall, dass der Wert in einem inneren Knoten steht, geht der Algorithmus so vor:

1. Ersetze den Wert mit dem Vorgänger im Kind (Wenn Kind mindestens t Werte besitzt).
2. Wenn es nicht genug Werte besitzt, ersetze den Wert mit dem Nachfolger im Kind (Wenn Kind mindestens t Werte besitzt).
3. Wenn beide Kinder weniger als t Werte besitzen, verbinde die beiden Knoten und den zu löschen Wert zu einem Knoten und lösche nun den Wert aus dem Knoten

Da dieser Algorithmus bereits beim Suchen den Baum umorganisiert um unnötige Operationen zu sparen gilt zudem, dass beim Suchen, wenn die Wurzel des Unterbaums, der den Wert beinhaltet muss $t - 1$ Werte besitzt und ein unmittelbares Geschwisternode mit mindestens t Werten besitzt, man einen Wert vom Parent in den Knoten tut und diesen mit dem Wert aus dem Geschwisternode ersetzt.

Wenn beide Nodes, also die Wurzel des Unterbaumes und die Geschwisternode $t - 1$ Werte haben, verbinde sie.

B-Baum: Suche



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 117

B-Baum Einfügen (informell)

Laufzeit $O(t \cdot h) = O(\log_t n)$

insert(T, z)

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

Aktueller Knoten hat zu diesem Zeitpunkt weniger als $2t - 1$ Werte, sonst wäre er vorher geteilt worden. Beim Splitten des Kindes kann der mittlere Wert daher problemlos im aktuellen Knoten eingefügt werden.

Auch Blatt hat zu diesem Zeitpunkt weniger als $2t - 1$ Werte.

B-Baum Löschen (informell)

Laufzeit $O(t \cdot h) = O(\log_t n)$

delete(T, k)

- 1 Wenn Wurzel nur 1 Wert und beide Kinder $t-1$ Werte, verschmelze Wurzel und Kinder (reduziert Höhe um 1)
- 2 Suche rekursiv Löschposition:
- 3 Wenn zu besuchendes Kind nur $t-1$ Werte, verschmelze es oder rotiere/verschiebe
- 4 Entferne Wert k in inneren Knoten/Blatt

Aktueller Knoten hat zu diesem Zeitpunkt mindestens t Werte, sonst wäre er vorher verschmolzen worden oder es wäre rotiert worden. Beim Verschmelzen/Verschieben des Kindes kann die Anzahl der Werte im aktuellen Knoten nicht unter $t - 1$ fallen.

Entfernen aus Blatt problemlos, da mindestens t Werte. Entfernen im inneren Knoten durch Verschieben oder Verschmelzen.

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 04 Fortgeschrittenen Datenstrukturen | 133

7 Probabilistische Datenstrukturen

7.1 Deterministisch und Probabilistisch

Bisher waren alle Datenstrukturen deterministisch, d.h., dass für die selben Eingaben das Verhalten immer gleich sein wird.

Bei probabilistischen Datenstrukturen ist das Verhalten nicht nur von Eingaben abhängig, sondern auch von zufälligen Faktoren.

Aspekt	Deterministische Datenstrukturen	Probabilistische (Randomisierte) Datenstrukturen
Vorteile		
Leistungsgarantien	Bietet garantierte Worst-Case-Zeitkomplexität.	Bietet gute durchschnittliche Leistung, oft schneller in der Praxis.
Vorhersehbarkeit	Verhalten ist für die gleichen Eingaben vorhersehbar und konsistent.	Flexibler und vermeidet Worst-Case-Szenarien unter typischen Bedingungen.
Worst-Case-Behandlung	Speziell entwickelt, um Worst-Case-Szenarien zu handhaben.	Vermeidet Worst-Case-Szenarien durch probabilistische Methoden.
Einfachheit	Konzeptuell einfach mit klaren Regeln (z.B. AVL-Baum-Rotationen).	Oft einfacher in der Implementierung, ohne komplexe Ausgleichsoperationen.
Stabilität	Deterministisches Verhalten führt zu stabilen und wiederholbaren Ergebnissen.	Flexibel und widerstandsfähig gegenüber unterschiedlichen Eingabemustern.
Nachteile		
Komplexität in der Implementierung	Erfordert oft komplexe Ausgleichslogik (z.B. Rot-Schwarz-Bäume, AVL-Bäume).	Einfacher, aber schwerer probabilistisch zu analysieren.
Speicherbedarf	Kann zusätzlichen Speicher für die Speicherung von Ausgleichsinformationen erfordern.	Kann spechereffizient sein, aber einige Strukturen (z.B. Bloom-Filter) können eine geringe Fehlerquote aufweisen.
Handhabung spezifischer Eingaben	Kann bei bestimmten Eingabemustern schlecht abschneiden (z.B. Quicksort mit sortierten Eingaben).	Vermeidet schlechte Leistung bei bestimmten Eingaben durch Zufälligkeit.
Vorhersehbarkeit	Vorhersehbar und kann von einem Gegner ausgenutzt werden (z.B. Hash-Kollisionsangriffe).	Weniger vorhersehbar, wodurch die Wahrscheinlichkeit sinkt, dass gegnerische Eingaben zu einer Worst-Case-Leistung führen.
Komplexität in der Analyse	Einfacher zu analysieren und zu verstehen in Bezug auf Worst-Case-Verhalten.	Erfordert probabilistische Analyse, um Leistungsgarantien zu verstehen.
Wesentliche Unterschiede		
Leistungsgarantien	Strikte Worst-Case-Leistungsgarantien.	Konzentriert sich auf erwartete durchschnittliche Leistung.
Verhalten unter adversen Bedingungen	Kann unter adversen Bedingungen erheblich nachlassen (z.B. bestimmte Hashing-Methoden).	Robuster gegenüber adversen Bedingungen aufgrund von Zufälligkeit.
Implementierungskomplexität	Kann mehr Aufwand erfordern, um eine optimale Leistung zu gewährleisten (z.B. Ausbalancierung von Bäumen).	Typischerweise einfacher zu implementieren mit guter durchschnittlicher Leistung.
Anwendungsfälle	Geeignet für Anwendungen, bei denen Worst-Case-Leistung entscheidend ist.	Ideal für Anwendungen, bei denen die durchschnittliche Leistung wichtiger ist.

7.2 Skip-Lists

Skip-Lsts sind eine Datenstruktur, die einer Linked-List sehr ähnelt. Sie baut auf der selben grundlegenden Struktur auf, erweitert diese jedoch noch zusätzlich. So haben Elemente in einer Skip-List nicht nur `next` (und eventuell `prev`) sondern auch noch `up` und `down`.

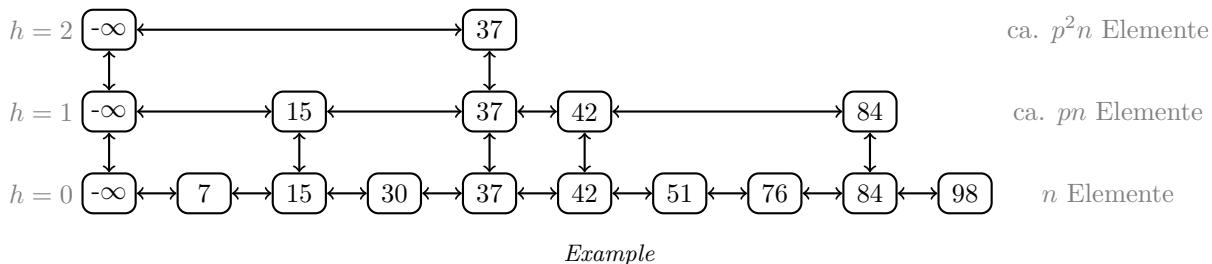
Die Struktur einer Skip-List ähnelt sogesehen mehreren aufeinandergestapelten Linked-Lists. Beim Aufbau der Skip-List wird für jedes Element zufällig ausgesucht, auf wie vielen Ebenen es abgebildet ist. Dies ergibt den Vorteil, dass diese Struktur beim `insert`, `delete` und `search` eine average time complexity von $O(\log n)$ besitzt, was sie mit balanzierten Bäumen (AVLT, RBT...) vergleichbar macht. Der Vorteil von Skip-Lsts über Bäume sind unter anderem, dass sie einfacher zu implementieren sind und weniger Speicher brauchen. Die Nachteile bilden hierbei die schlechte Worst-Case Performance von $O(n)$.

Skip-Lsts werden oft in Datenbanken genutzt um Daten nach einer spezifizierten Ordnung zu speichern, aber auch für Datensätze, die oft modifiziert werden müssen, da das anwenden von anderen Algorithmen auf Skip-Lsts oft relativ einfach ist.

```

1 class SkipList {
2     class SkipNode {
3         int key;
4         SkipNode next;
5         SkipNode prev;
6         SkipNode down;
7         SkipNode up;
8         SkipNode(int k) {
9             key = k;
10        }
11    }
12    SkipNode head; // First node in highest level
13    int height; // starts at 0 -> biggest list at height = 0
14    final double P = 0.5; // Probability of inserted node being added to express list
15    SkipList() {
16        head = new SkipNode(Integer.MIN_VALUE);
17        height = 0;
18    }

```

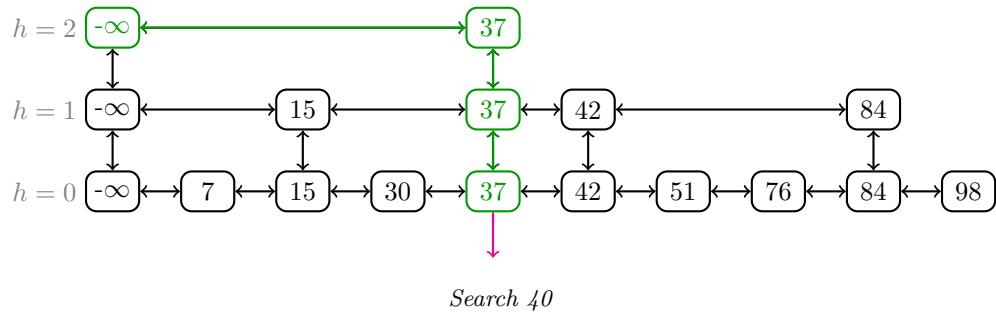
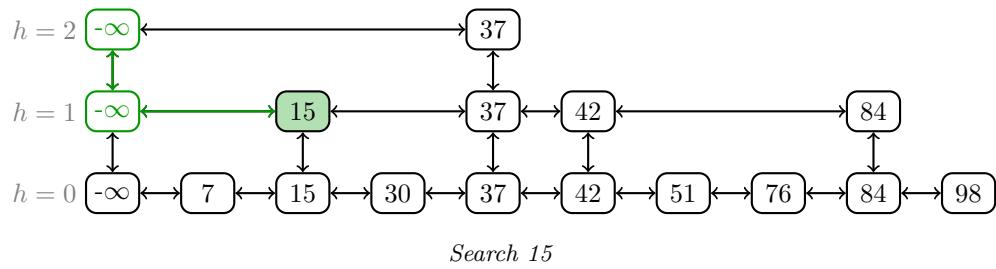
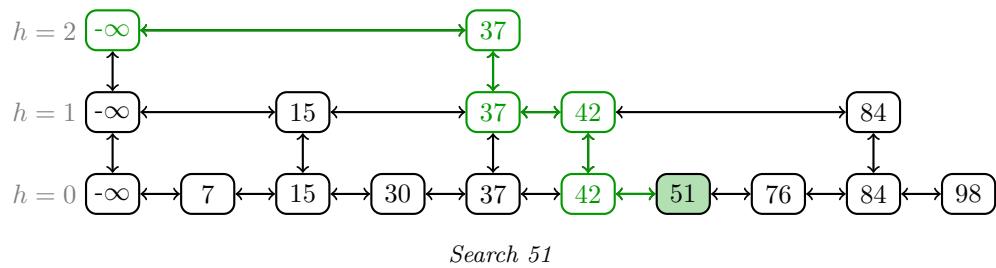


Die Anzahl der Elemente auf einer Ebene nimmt ungefähr nach dem Schema $p^h \cdot n$ ab, wobei p die Wahrscheinlichkeit und h die Höhe ist.

Der Suchalgorithmus ist relativ simpel. Er startet beim Head (Höchster Knoten des Starts) und durchläuft jedes Level in order, bis der nächste Knoten größer ist als der gesuchte Knoten / `null` ist. Dann geht er an dem Knoten nach unten und durchsucht diese Liste weiter. Dies geschieht, bis er entweder den Wert gefunden hat, in welchem Fall er den gefundenen Knoten zurück gibt, oder bis er `null` erreicht, was bedeutet, dass der Wert nicht in der Liste ist.

```

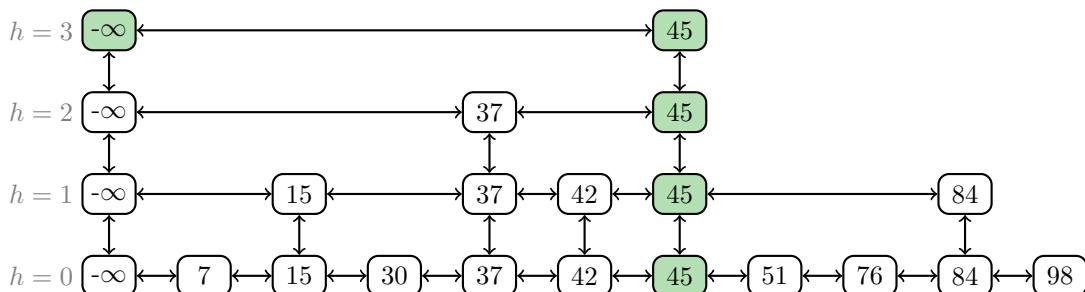
1  SkipNode search(int k) {
2      SkipNode curr = head;
3      while (curr != null) {
4          if (curr.key == k) // key found
5              return curr;
6          else if (curr.next != null && curr.next.key <= k) // If next key is less than or equal to k
7              curr = curr.next; // move along the list
8          else // If next key is greater than k
9              curr = curr.down; // move down the list
10     }
11 }
12 }
```



Einfügen in die Skip-List ist ein wenig komplizierter, aber auch grundlegend simpel. Erst einmal muss ermittelt werden auf wie viele Ebenen der Wert hinzugefügt werden muss. Dies kann über eine vom Element abhängige Formel oder auch andere Methoden zum erzeugen einer zufälligen Zahl geschehen. Nachdem dieses Level ermittelt wurde, muss, falls notwendig die Liste um die noch nicht vorhandenen Ebenen ergänzt werden. Dies geschieht dadurch, dass man die Head Node nach oben erweitert, den height-Counter erhöht und den Head auf den neuen Head aktualisiert. Nachdem dies geschehen ist wird nun die Liste wie beim search-Algorithmus durchgangen um die Einfügestelle zu ermitteln. Beim Durchlauf werden zusätzlich noch die zukünftigen Vorgängerknoten in einem Array vermerkt. Nachdem werden nun die Knoten auf den bestimmten Ebenen eingefügt.

```

1  int randomLevel() { // Used to determine in how many lists a node will be added
2      double r = Math.random();
3      int lvl = 0;
4      while (r < P) {
5          lvl++;
6          r = Math.random();
7      }
8      return lvl;
9  }
10 void insert(int k) {
11     int lvl = randomLevel();
12     while (lvl > height) { // If needed increase list height and add required heads
13         head.up = new SkipNode(Integer.MIN_VALUE);
14         head.up.down = head;
15         head = head.up;
16         height++;
17     }
18     SkipNode[] prevs = new SkipNode[height + 1]; // Holds the previous nodes in each list
19     SkipNode curr = head;
20     for (int i = height; i >= 0; i--) { // For each level starting from the highest
21         while (curr.next != null && curr.next.key < k)
22             // Loops through current list until next key is greater than or equal to k
23             curr = curr.next;
24         if (curr.key == k) return; // key already in list
25         prevs[i] = curr; // Adds current node as a predecessor to the new node
26         curr = curr.down; // Moves down in the list
27     }
28     int count = 0; // counter for number of lists in which the new node has been added
29     SkipNode dwn = null; // Holds the node that is the down node of the node in a level
30     while (count <= lvl) { // Add new nodes to lists in lvl
31         SkipNode newNode = new SkipNode(k);
32         newNode.next = prevs[count].next; // Includes the new node in the list
33         newNode.prev = prevs[count];
34         if (prevs[count].next != null) // If there is a next node
35             prevs[count].next.prev = newNode; // change previous to new node
36         prevs[count].next = newNode; // change next to new node
37         newNode.down = dwn; // connect newNode to itself on a lower level
38         if (dwn != null) // If added to more than 1 level
39             dwn.up = newNode; // connect lower level to newNode
40         dwn = newNode; // Update dwn
41         count++; // Update counter
42     }
43 }
```

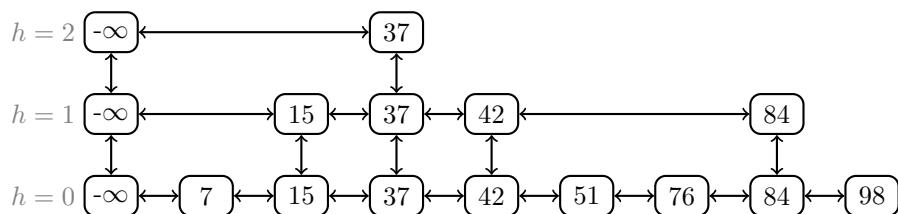


Insert 45 with randomLevel = 3

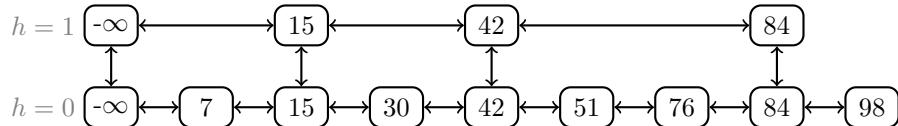
Delete ist wieder relativ simpel. Es wird erstmal die zu löschenende Node gesucht. Nachdem muss lediglich die Referenzen von den Vor- und Nachgängern abgeändert werden, sodass das Element selbst nichtmehr in der Liste ist. Dies muss nun lediglich nur für jede Ebene wiederholt werden.

```

1 void delete(int k) {
2     SkipNode node = search(k); // Find first node in list that has k
3     while (node != null) { // Moves down from the found node
4         // Remove node from list by removing its references
5         node.prev.next = node.next;
6         if (node.next != null)
7             node.next.prev = node.prev;
8         if (node.next == null
9             && node.prev.key == Integer.MIN_VALUE
10            && node.prev.down != null) {
11             // If deleted node is the last node in the list, reduce list height and update head
12             node.prev.down.up = null;
13             head = node.prev.down;
14             height--;
15         }
16         node = node.down;
17     }
18 }
19 }
```



Delete 30



Delete 37

Skip-Liste: Suchalgorithmus

```
search(L,k)

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

Laufzeit hängt von Expresslisten ab

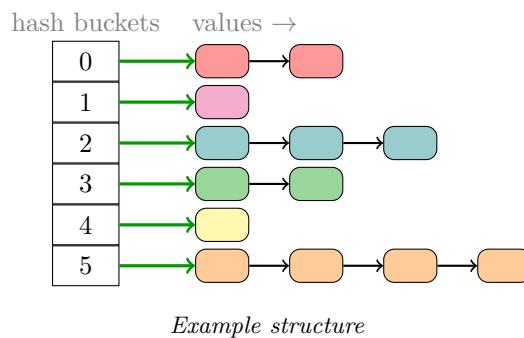
7.3 Hash Tables

Hash Tables sind eine sehr effiziente Datenstruktur, die Einfügen, Löschen und Suchen in Konstanter Zeit erlaubt. Sie funktioniert dadurch, dass sie keine Datenstruktur zum Suchen durchlaufen muss, sondern anhand des gesuchten Elements eine sogenannte Hash-Funktion berechnet. Diese wird dann auf die Länge der grundlegenden Array-Struktur komprimiert und als Index genutzt um ein Element einzufügen/zu finden.

Zwar sind die Hash-Funktionen meist so groß, dass es sehr unwahrscheinlich ist, dass zwei Objekte den gleichen Hash-Code haben, jedoch wird dieses mit der Komprimierung schwierig. Deshalb werden bei der Implementation von Hash Tables oft verschiedene Taktiken genutzt um Double Hashing einzubinden. Die wohl bekannteste ist es den Hash Table mit einer Linked List zu implementieren.

Normal sind die Aufrufe in konstanter Zeit, durch die Implementation mit Linked List verschlechtert sich dies aber im Worst-Case zu $O(n)$.

```
1 import java.util.Objects;
2
3 class HashTable <K, V> {
4     class HashNode <K, V> {
5         K key;
6         V value;
7         final int hashCode;
8         HashNode<K, V> next;
9         HashNode(K key, V value, int hashCode) {
10             this.key = key;
11             this.value = value;
12             this.hashCode = hashCode;
13         }
14     }
15     private int numBuckets;
16     private HashNode<K, V>[] buckets;
17     HashTable(int numBuckets) {
18         this.numBuckets = numBuckets;
19         this.buckets = new HashNode[numBuckets];
20     }
21     private int hashCode(K key) {
22         return Objects.hashCode(key); // 0 if key == null
23     }
24     private int getBucketIndex(K key) {
25         return Math.abs(hashCode(key)) % numBuckets; // Hashcode can be negative, index cant
26     }
```



Einfügen, Suchen und Löschen funktionieren Prinzipiell gleich zu dem in LinkedLists, nachdem der korrekte Hash Bucket gefunden wurde.

```
1 void put(K key, V value) { // Theta(1), sogar worstcase (vorne einfügen)
2     int bucketIndex = getBucketIndex(key); // search index of bucket where key should be
3     HashNode<K, V> head = buckets[bucketIndex]; // search head of bucket
4     buckets[bucketIndex] = new HashNode<>(key, value, hashCode(key)); // create new node
5     if (head != null)
6         buckets[bucketIndex].next = head; // connect new node to head
7 }
8 V get(K key) { // Theta(1), O(n)
9     int bucketIndex = getBucketIndex(key); // search index of bucket where key should be
10    HashNode<K, V> head = buckets[bucketIndex]; // search head of bucket
11    while (head != null) { // goes through the buckets list
12        if (head.key.equals(key) && head.hashCode == hashCode(key)) // if key is found
13            return head.value; // return value
14        head = head.next; // move head to next
15    }
16    return null; // If key not found -> return null
17 }
18 V remove(K key) { // Theta(1), O(n)
19     int bucketIndex = getBucketIndex(key); // search index of bucket where key should be
20     HashNode<K, V> head = buckets[bucketIndex]; // search head of bucket
21     HashNode<K, V> prev = null; // search prev of head
22     while (head != null) { // goes through the buckets list
23         if (head.key.equals(key) && head.hashCode == hashCode(key)) { // if key is found
24             if (prev == null) // if key is the head
25                 buckets[bucketIndex] = head.next; // move head to next
26             else // if key is not the head
27                 prev.next = head.next; // change references
28             return head.value; // return removed value
29         }
30         prev = head; // move prev to head
31         head = head.next; // move head to next
32     }
33     return null; // If key not found -> return null
34 }
35 }
```

7.4 Bloom Filter

Ein Bloom-Filter ist eine Datenstruktur, die benutzt wird um schnell herauszufinden, ob ein Element in einer Datenstruktur vorkommt. Sie ist ideal für große Datensätze, bei denen *false-positives* akzeptabel sind, *false-negatives* nicht. D.h. sie können zuverlässig sagen, ob ein Element vorkommt, können aber auch anschlagen, wenn ein Element nicht explizit eingefügt wurde.

Bloom Filter sind sehr effizient, da sie zum einem nicht die Elemente selbst speichern, sondern nur ihre Anwesenheit, zum anderen, da das Einfügen und Auslesen auch sehr schnell ist mit $O(k)$ mit k die Anzahl der Funktionen.

Nachteile von Bloom Filtern sind die *false-positives*, die sehr komplizierte deletion (Ein Element rauslöschen kann auch anderes rauslöschen, was *false-negatives* erzeugt) und die festgelegte Größe.

Sie werden häufig genutzt um Caches und Spam zu filtern, aber auch um bspw. zu schauen, ob ein Passwort häufig verwendet wird.

```
1 import java.util.function.Function;
2 class BloomFilter {
3     Function<String, Integer>[] functions;
4     boolean[] bloomFilter;
5     int bloomSize;
6     BloomFilter(int bloomSize, Function<String, Integer>[] functions) {
7         this.functions = functions; // functions that will be used
8         this.bloomSize = bloomSize; // size of the bloom filter
9         bloomFilter = new boolean[bloomSize]; // defaults to false
10    }
11    void exampleFunctions() {
12        functions = new Function[3];
13        functions[0] = (String s) -> s.length() % bloomSize;
14        functions[1] = (String s) -> s.charAt(0) % bloomSize;
15        functions[2] = (String s) -> s.charAt(s.length() - 1) % bloomSize;
16    }
```

Bei diesen Beispielfunktionen, könnte man z.B. **Dance** einfügen, was aber die gleichen bits wie **Dodge** belegt. Demnach würde bei der Suche für **Dodge** ein *false-positive* erzeugt werden. False-Positives werden immer wahrscheinlicher je mehr Elemente eingefügt werden und je kleiner der Bloom Filter ist. Natürlich sind sie aber auch grundlegend von den Functions, bezüglich der Komplexität, Probabilistik und Anzahl, abhängig. Diese Beispielfunktionen sind relativ schlecht, da sie simpel sind, was im Endeffekt in mehr *false-positives* resultiert. Abstraktere, komplexere Funktionen funktionieren hierbei meist besser, da sie nicht an Adjezenz der Eingaben festhalten.

Einfügen ist sehr simpel. Es müssen lediglich die Funktionen auf die Eingabe angewendet werden und die entsprechenden Bits danach umgestellt werden.

```

1 void addToBloom(String x) { // O(k), k = number of functions
2     for (Function<String, Integer> function : functions) { // for each function
3         bloomFilter[function.apply(x)] = true; // add to bloom
4     }
5 }
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Before ($bloomSize = 16$)

Insert "hello":

- $x_0 = "hello".length() \% 16 = 5$
- $x_1 = (int)'h' = 104 \% 16 = 8$
- $x_2 = (int)'o' = 111 \% 16 = 15$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1

Insert "world":

- $x_0 = "world".length() \% 16 = 5$
- $x_1 = (int)'w' = 119 \% 16 = 7$
- $x_2 = (int)'d' = 100 \% 16 = 4$

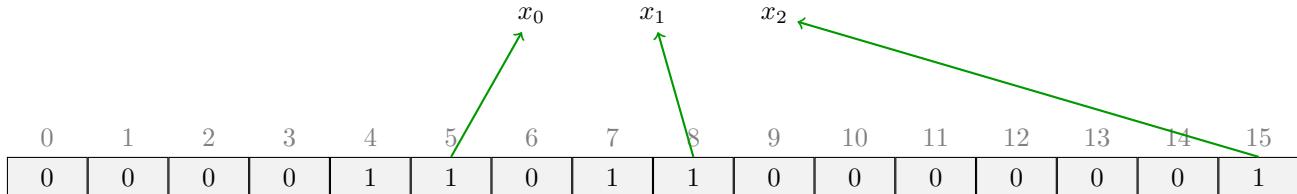
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	1	1	0	1	1	0	0	0	0	0	0	1

Das Durchsuchen des Bloom-Filters läuft fast identisch zum Einfügen ab. Es müssen wieder mittels der Funktionen die passenden Werte der Eingabe gefunden werden. Diese werden dann verwendet um zu schauen, ob alle benötigten Bits im Bloom-Filter vorhanden sind. Wenn alle da sind ist das Element im Bloom-Filter enthalten, allerdings kann dies auch wahr für Werte sein, die nicht explizit eingefügt wurden (false-positive).

```

1  boolean isInBloom(String x) { // O(k), k = number of functions
2      for (Function<String, Integer> function : functions) { // for each function
3          if (!bloomFilter[function.apply(x)]) // if not in bloom
4              return false; // One function is not in bloom -> false
5      }
6      return true; // If all functions are present in bloom -> true
7  }
8 }
```

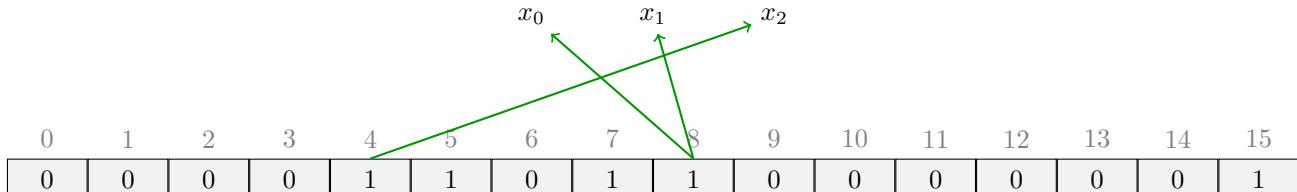
Check if "hello" exists: (true-positive)



⇒ Existenz im Bloom Filter

Check if "harassed" exists: (false-positive)

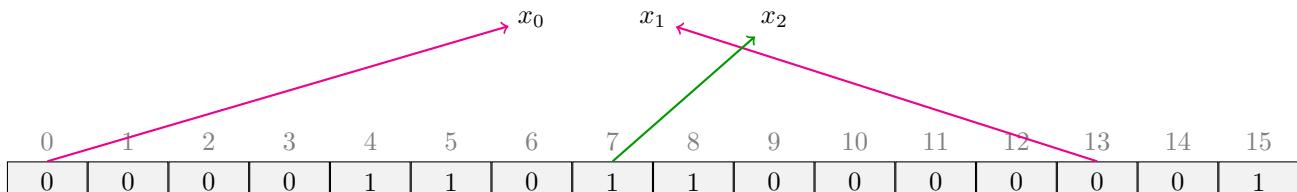
- $x_0 = \text{"harassed".length()} \% 16 = 8$
- $x_1 = (\text{int})'h' = 104 \% 16 = 8$
- $x_2 = (\text{int})'d' = 100 \% 16 = 4$



⇒ Existenz im Bloom Filter obwohl nicht spezifisch eingefügt

Check if "misunderstanding" exists: (true-negative)

- $x_0 = \text{"misunderstanding".length()} \% 16 = 0$
- $x_1 = (\text{int})'m' = 109 \% 16 = 13$
- $x_2 = (\text{int})'g' = 103 \% 16 = 7$



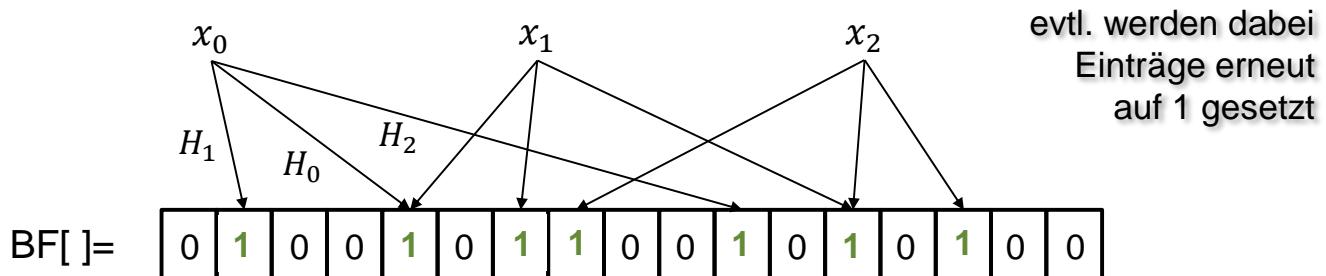
⇒ keine Existenz im Bloom Filter

Bloom-Filter: Erstellen (II)

```
initBloom(X,BF,H) //H array of functions H[j]
1  FOR i=0 TO BF.length-1 DO BF[i]=0;
2  FOR i=0 TO X.length-1 DO
3      FOR j=0 TO H.length-1 DO
4          BF[H[j](X[i])] = 1;
```

1. Initialisiere Array mit 0-Einträgen

2. Schreibe für jedes Element in jede Bit-Position $H_0(x_i), \dots, H_{k-1}(x_i)$ eine 1

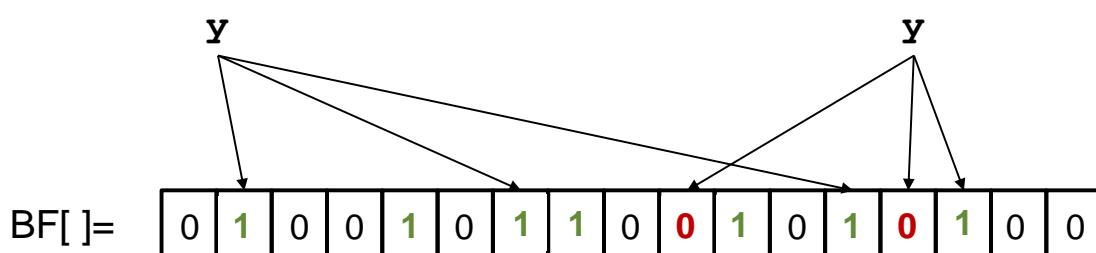


Bloom-Filter: Suchen

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

Gib an, dass y im Wörterbuch, wenn genau alle k Einträge für y in $BF=1$ sind

in Wörterbuch:



nicht in Wörterbuch:

8 Graphen Algorithmen

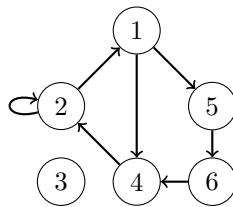
8.1 Graphen

Der abstrakter Datentyp der Graphen besteht im Grunde aus verschiedenen Knoten (vertices V), die Kanten (edges E) zu anderen Knoten haben. Sogesehen sind auch Bäume immer Graphen, wenn auch beschränkter.

8.1 (a) Gerichtete Graphen

Gerichtete Graphen sind Graphen bei denen die edges E immer nur in eine Richtung sind. D.h., dass eine Kante von Knoten u nach Knoten v **nicht** gleichzusetzen ist wie eine Kante von Knoten v nach Knoten u.

In normaler Notation werden Kanten im Schema $(u, v) \in E$ angegeben, was eine Kante von Knoten u nach Knoten v darstellt.



Example

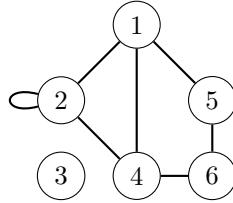
$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 4), (1, 5), (2, 1), (2, 2), (4, 2), (5, 6), (6, 4)\}$$

(3 hat keine Kanten weg oder zu sich, ist aber trotzdem Teil des Graphs)

8.1 (b) Ungerichtete Graphen

Ungerichtete Graphen unterscheiden sich von gerichteten Graphen nur in der Hinsicht, dass die Kanten $(u, v) \in E$ und $(v, u) \in E$ gleichzusetzen sind.



Example

$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{\{1, 4\}, \{1, 5\}, \{1, 2\}, \{2, 2\}, \{2, 4\}, \{4, 6\}, \{5, 6\}\}$$

(Geschweifte Klammern stellen ungerichtete Kante dar)

8.1 (c) Pfade

Ein Knoten v ist von einem Knoten u dann erreichbar, wenn es einen Pfad $(w_1, w_2, \dots, w_k) \in V^k$ für den $(w_i, w_{i+1}) \in E$ für $i = 1, 2, \dots, k - 1$ und $w_1 = u$ und $w_k = v$ gibt.

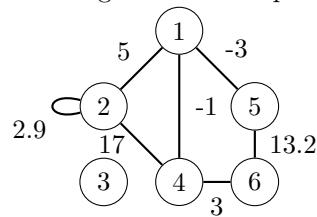
(Ein Knoten ist immer von sich selbst erreichbar (leerer Pfad $k = 1$))

Dabei ist die Länge des Pfades mit $k - 1$ = Anzahl der Kanten gegeben. $\text{shortest}(u, v) =$ Länge des kürzesten Pfades von u nach v .

8.1 (d) Gewichtete Graphen

Gewichtete Graphen haben zusätzlich bei den Kanten ein Gewicht. Dieses kann unterschiedliche Dinge, je nach Kontext bedeuten, bspw. könnte ein gewichteter Graph dazu genutzt werden Zugverbindungen darzustellen, wo die Knoten Haltestellen, die Kanten Zugverbindungen und die Gewichte die Entfernung darstellen.

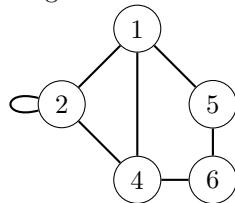
Die Notation hierzu ist $w((u, v)) = w((v, u))$



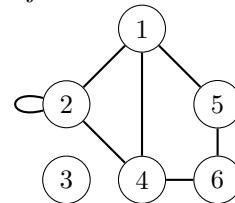
Example

8.1 (e) Zusammenhänge

Ungerichtete Graphen gelten als **zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten erreichbar ist.

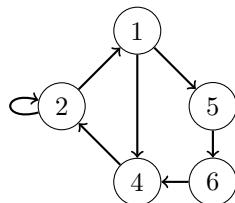


Zusammenhängend

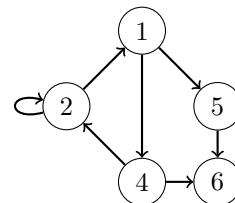


Nicht zusammenhängend

Ein gerichteter Graph gilt als **stark zusammenhängend**, wenn jeder Knoten von jedem anderen Knoten (beachte Kantenrichtung) erreichbar ist.



Zusammenhängend



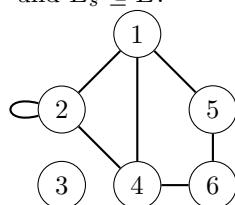
Nicht zusammenhängend

Kein Pfad von 5 nach 1. (Richtung von (4,6) geändert)

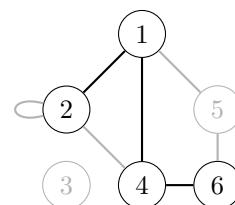
8.1 (f) Subgraphen

Ein Subgraph ist ein Graph, bei dem alle Kanten und Knoten auch in dem übergeordneten Graph liegen.

So gilt also $V_s \subseteq V$ und $E_s \subseteq E$.



Main Graph

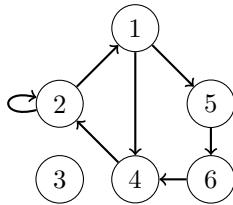


Subgraph

8.1 (g) Darstellungen

Adjazenzmatrix:

Die Adjazenzliste stellt alle Kanten von einem Knoten zu anderen Knoten in jeder Zeile dar.



	to→					
from ↓	1	2	3	4	5	6
1	0	0	0	1	1	0
2	1	1	0	0	0	0
3	0	0	0	0	0	0
4	0	1	0	0	0	0
5	0	0	0	0	0	1
6	0	0	0	1	0	0

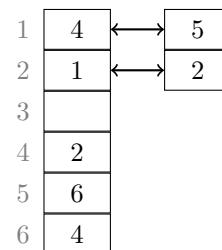
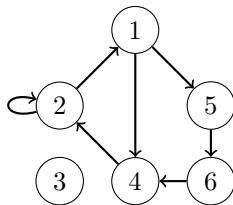
Example

Für ungerichtete Graphen ist die Adjazenzmatrix spiegelsymmetrisch zur Hauptdiagonalen.

Die Adjazenzmatrix hat die Eigenschaft, dass der Eintrag $a_{i,j}^{(m)}$ (i -te Zeile, j -te Spalte der m -ten (A^m) Potenz der Matrix A) die Anzahl der Wege von Knoten i zum Knoten j mit genau m Kanten besitzt.

Adjazenzliste:

Die Adjazenzliste stellt die Kanten von einem Knoten zu anderen Knoten als Array mit verketteten Listen dar.



Example

8.2 Breadth-First Search (BFS)

Der Breadth-First Search Algorithmus funktioniert nach dem Prinzip, dass von dem Startpunkt aus erst alle direkten Nachbarn besucht werden. Anschließend werden alle Nachbarn der direkten Nachbarn besucht und so weiter.

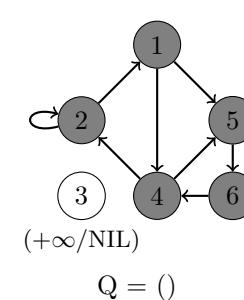
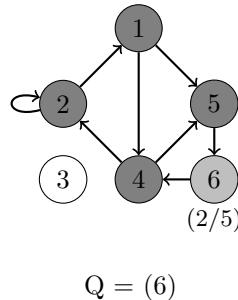
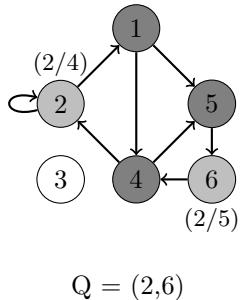
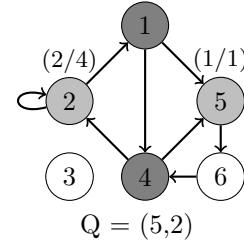
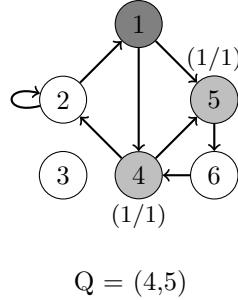
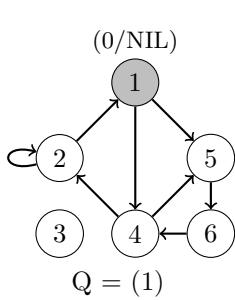
Allgemein basiert dieser Algorithmus also eher auf der Queue Mechanik, er fügt zuerst alle Werte hinzu und bearbeitet dann den zuerst hinzugefügten Wert.

```

1 // G: graph with (V,E), s: start
2 Function breadthFirstSearch(G,s):
3   // for all nodes except s:
4   ForEach u in V-{s} do
5     u.color = WHITE; u.dist = +∞; u.pred = NIL; // Initializes all nodes to appropriate values
6   s.color = GRAY; s.dist = 0; s.pred = NIL; // Initializes s at the first node
7   Q = new Queue(); // Initializes queue for saving order
8   Q.enqueue(s); // add s as first node to queue
9   // While there is an unfinished node with a path from s:
10  While !isEmpty(Q) do
11    u = Q.dequeue();
12    // For all nodes adjacent to u:
13    ForEach v in Adj(u) do
14      // If v has not been visited yet:
15      If v.color == WHITE then
16        v.color = GRAY; // Mark node as visited
17        v.dist = u.dist + 1; // set distance
18        v.pred = u; // set predecessor
19        Q.enqueue(v); // Add node to queue
20    u.color = BLACK; // Mark current node as finished

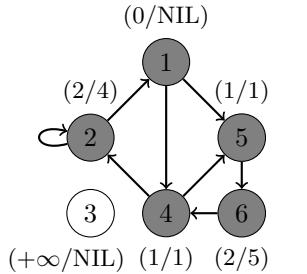
```

In diesem Code wird der Status der Knoten über Farben angegeben. WHITE bedeutet noch nicht besucht, GRAY bedeutet besucht, BLACK bedeutet beendet.

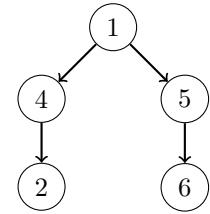


Der erste Wert in den Klammern ist der Schritt, an dem der Knoten entdeckt wurde. Der zweite Wert ist der Vorgängerknoten.

Nach dem Ausführen von BFS kann man durch die jetzt gegebenen Vorgängern einen Subgraphen ableiten.



Graph nach BFS



Abgeleiteter Subgraph

Der Subgraph ist durch $V_{pred}^s = \{v \in V | v.pred \neq \text{NIL}\} \cup \{s\}$ und $E_{pred}^s = \{(v.pred, v) | v \in V_{pred}^s - \{s\}\}$ gegeben. D.h. bedeutet, dass der Subgraph alle von s erreichbaren Knoten enthält und für jeden Knoten im Subgraph genau ein Pfad von s , der auch automatisch der kürzeste Pfad von s zu v ist, existiert.

Um alle Knoten auf dem kürzesten Pfad zwischen zwei Knoten auszugeben kann der folgende Algorithmus verwendet werden.

```

1 // G: Graph mit (V,E), s: Startknoten, v: Zielknoten
2 Function printShortestPath(G, s, v):
3     // if end == start:
4     If v == s then
5         print s;
6     // if end nodes predecessor is NIL (no path):
7     Else If v.pred == NIL then
8         print "no path from s to v";
9     Else
10        printShortestPath(G, s, v.pred); // Recursion (backwards from end to start)
11        print v;

```

BFS: Algorithmus

dist =Distanz von s
pred =Vorgängerknoten

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

1 FOREACH u in V-{s} DO
2   u.color=WHITE; u.dist=+∞; u.pred=NIL;
3 s.color=GRAY; s.dist=0; s.pred=NIL;
4 newQueue(Q);
5 enqueue(Q, s);
6 WHILE !isEmpty(Q) DO
7   u=dequeue(Q);
8   FOREACH v in adj(G, u) DO
9     IF v.color==WHITE THEN
10       v.color=GRAY; v.dist=u.dist+1; v.pred=u;
11       enqueue(Q, v);
12 u.color=BLACK;
```

WHITE =Knoten noch nicht besucht
GRAY=in Queue für nächsten Schritt
BLACK =fertig

$\text{adj}(G, u)$ = Liste aller Knoten $v \in V$ mit $(u, v) \in E$
(Reihenfolge irrelevant)

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 20



Kürzeste Pfade ausgeben

Laufzeit* = $O(|V|)$
*ohne BFS

```
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;
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 29



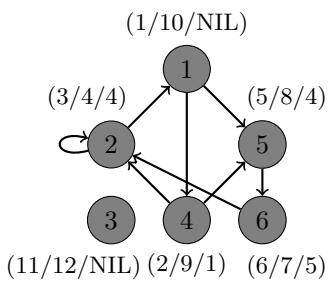
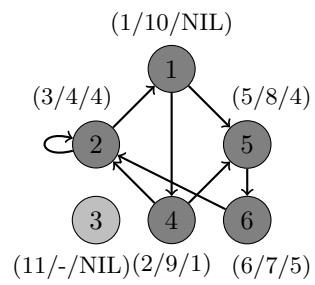
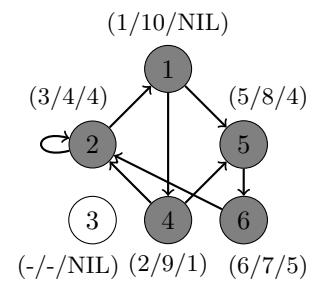
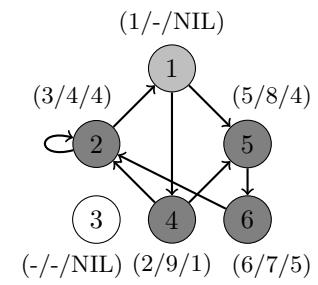
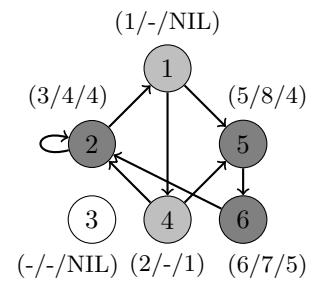
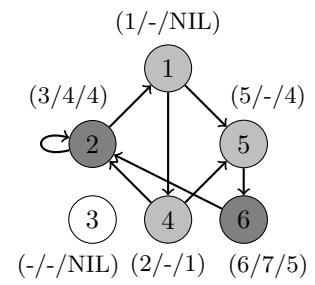
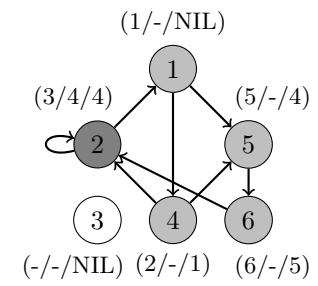
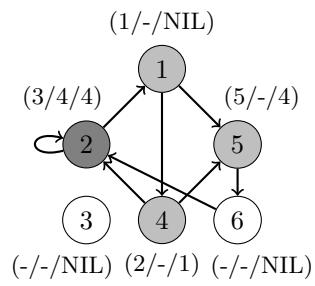
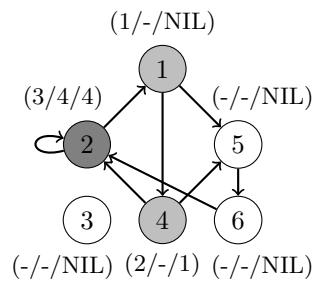
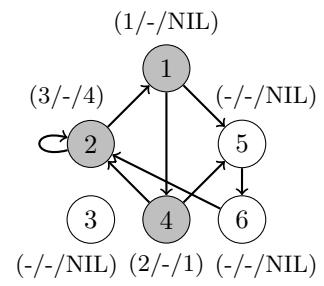
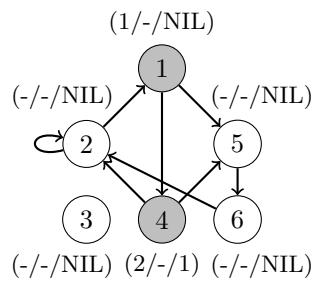
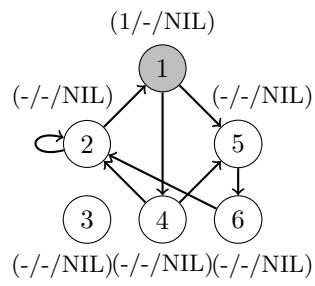
8.3 Depth-First Search (DFS)

Das Prinzip des Depth-First Search Algorithmus ist, dass im Gegensatz zu BFS nicht erst alle Nachbarn eines Knotens besucht werden, sondern immer erst ein Pfad zuende gebracht wird, bevor ein anderer besucht wird. Der Algorithmus geht also solange einen Pfad ab, bis er an einem Knoten keine neuen Knoten mehr findet, wodurch er dann einen Knoten zurück geht und von diesem den nächsten Pfad abgeht. Dies macht er solange, bis er alle Pfade abgelaufen ist.

Allgemein kann man als Unterschied zu BFS sagen, dass dieser Algorithmus mehr der Stack Datenstruktur ähnelt, da sie zuerst die Knoten merkt und dann den zuletzt hinzugefügten Knoten verarbeitet.

```
1 // G: graph with (V,E)
2 Function depthFirstSearch(G):
3     // for all nodes in the graph:
4     ForEach u in V do
5         u.color = WHITE; u.pred = NIL; // Initializes all nodes to appropriate values
6     time = 0; // Initializes time to 0
7     // for all nodes in the graph:
8     ForEach u in V do
9         // if node not visited:
10        If u.color == WHITE then
11            visit(G,u);
12 Function visit(G,u):
13    u.color = GRAY; u.disc = ++time // Mark node as visited and set discovrey time
14    // for all nodes adjacent to u:
15    ForEach v in u.adj do
16        // if node not visited:
17        If v.color == WHITE then
18            v.pred = u; // set predecessor
19            visit(G,v);
20    u.color = BLACK; u.finish = ++time; // Mark node as finished and set finish time
```

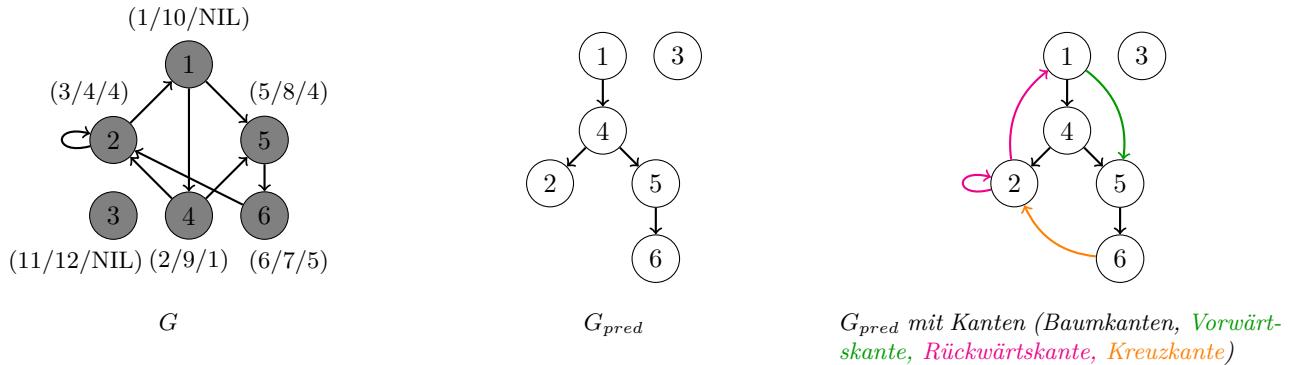
Der Algorithmus hat im Gegensatz zu BFS keinen angegebenen Startknoten, sondern geht die Knoten nach natural Order an.



(Discovery/Finish/Predecessor)

Wie bei BFS auch, kann man hier nun einen abgeleiteten Subgraphen erstellen. Unterschiedlich zu BFS aber, stellt dieser nun nicht unbedingt die kürzesten Pfade zwischen zwei Knoten dar. Zusätzlich kann dieser Subgraph genutzt werden um die einzelnen Kanten zu klassifizieren, was in den kommenden Algorithmen wichtig wird. Für die Kantenklassifizierung gilt:

- **Baumkante:** Alle Kanten in G_{pred}
- **Keine Baumkante:**
 - **Vorwärtskante:** Alle Kanten in G zu Nachkommen in G_{pred} .
 - **Rückwärtskante:** Alle Kanten in G zu Vorfahren in G_{pred} (inkl. Schleifen).
 - **Kreuzkante:** Alle Kanten, die nicht Baum-, Vorwärts, oder Rückwärtskante sind.



Man kann die Kantenarten auch schon während DFS klassifizieren. Sei (u,v) die gerade betrachtete Kante. Dann ist (u,v) :

- **Baumkante:** Wenn $v.\text{color} == \text{WHITE}$
- **Rückwärtskante:** Wenn $v.\text{color} == \text{GRAY}$
- **Vorwärtskante:** Wenn $v.\text{color} == \text{BLACK}$ und $u.\text{disc} < v.\text{disc}$
- **Kreuzkante:** Wenn $v.\text{color} == \text{BLACK}$ und $u.\text{disc} > v.\text{disc}$

Zusätzlich ist es wichtig zu wissen, dass es in ungerichteten Graphen nur Baum- und Rückwärtskanten gibt.

DFS: Algorithmus

Laufzeit = $O(|V| + |E|)$

time globale Variable

DFS-VISIT(G, u)

```

1 time=time+1;
2 u.disc=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;

```

DFS(G) // G=(V, E)

```

1 FOREACH u in V DO
2   u.color=WHITE;
3   u.pred=NIL;
4 time=0;
5 FOREACH u in V DO
6   IF u.color==WHITE THEN
7     DFS-VISIT(G,u)

```

disc = discovery time
finish=finish time

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 35



Topologisches Sortieren mittels DFS

TOPOLOGICAL-SORT(G) // G=(V, E) dag

```

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

```

Laufzeit = $O(|V| + |E|)$

da Einfügen
in Liste vorne
in Zeit $O(1)$

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 63

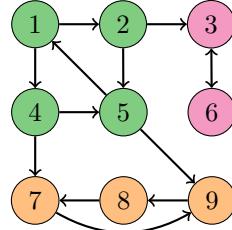


8.4 Strongly Connected Components (SCC)

Eine starke Zusammenhangskomponente (Strongly Connected Components (SCC)) ist eine Knotenmenge $C \subseteq V$ für die gilt:

- Zwischen je zwei Knoten $u, v \in C$ gibt es einen Pfad von u nach v .
- C ist maximal - Es gibt keine Menge $D \subseteq V$ mit $C \subseteq D$ für die ersteres gilt.

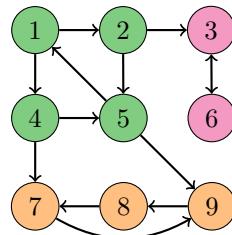
Ein Graph kann somit mehrere SCCs enthalten, diese können sich aber nicht überschneiden. Zwei SCCs C, D mit $u, v \in C$ und $w, x \in D$ mit einem Pfad $u \rightarrow w$ können also keinen Pfad $x \rightarrow v$ besitzen, ansonsten wären sie gleich.



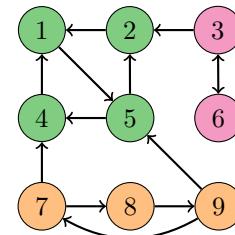
```

1 // Graph with (V,E)
2 Function stronglyConnectedComponents(G):
3   depthFirstSearch(G); // Needs to sort vertices by finish time
4   compute  $G^T$ ; // Transposed graph - Graph with  $(V, E^T)$  with all edges reversed
5   depthFirstSearchDescending( $G^T$ ); // Main loop goes through vertices by descending finish time
6   output each DFS tree in depthFirstSearchDescending() as one SCC;

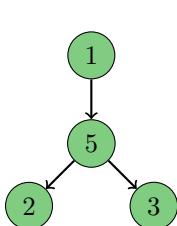
```



G



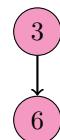
G^T - Transposed Graph(V, E^T), SCCs bleiben gleich, aber Richtungen ändern sich → Verschiedene SCCs nicht mehr miteinander verbunden



SCC 1 output



SCC 2 output



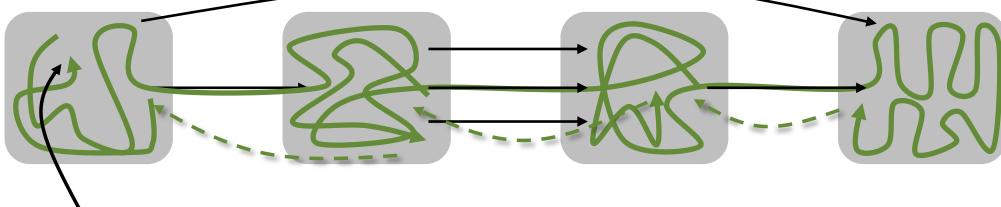
SCC 3 output

SCC Algorithmus: Idee (I)

```
SCC(G) // G=(V,E) directed graph

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

1 run DFS(G)



Knoten mit höchster
finish time liegt
in dieser SCC

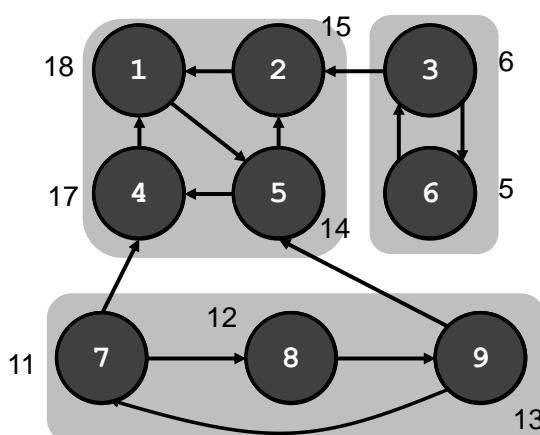
Bild nimmt zusammenhängenden Graphen an,
funktioniert aber auch allgemein, dann nacheinander
für die zusammenhängenden Subgraphen

SCC Algorithmus: Beispiel (II)

```
SCC(G) // G=(V,E) directed graph

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

umgedrehte Kanten

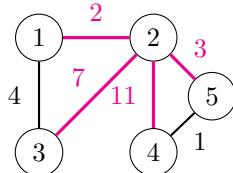


8.5 Minimale Spannbäume (MST)

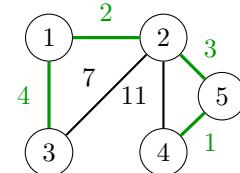
Für einen zusammenhängenden, ungerichteten, gewichteten Graph $G = (V, E)$ mit Gewichten w ist ein Subgraph $T = (V, E_T)$ ein Spannbaum, wenn T azyklisch ist und alle Knoten verbindet.

Ein Spannbaum ist also der Graph mit der minimalen Anzahl an Kanten, die aber trotzdem noch Pfade zu allen Knoten darstellen.

Ein **minimaler** Stammbaum hat zusätzlich noch die Eigenschaft, dass $w(T) = \sum_{\{u,v\} \in E_T} w(u,v)$ minimal im Vergleich zu allen anderen Spannbäumen ist. Einfach gesagt heißt das, dass das Gesamtgewicht des minimalen Spannbaums kleiner ist, als die Gesamtgewichte aller anderen Spannbäume.



Ein Spannbaum



Minimaler Spannbaum

Sichere Kanten

Eine sichere Kante ist eine Kante die in jedem Fall im minimalen Spannbaum enthalten ist. Um eine sichere Kante zu finden kann man beispielsweise den Spannbaum aufteilen. Die leichteste Kante, die dann die beiden Teilgraphen miteinander verbindet ist dann eine sichere Kante.

Anders ausgedrückt: **Sei A Teilmenge eines MST, (S, V-S) Schnitt, der A respektiert und {u,v} die leichte Kante, die den Schnitt überbrückt. Dann ist {u,v} eine sichere Kante für A.**

Hierbei gilt:

- $\{u,v\}$ sicher für A wenn $A \cup \{u,v\}$ Teilmenge eines MST
(Sicher wenn Vereinigung von A und $\{u,v\}$ eine Teilmenge des MST ist)
- Schnitt $(S, V-S)$ partitioniert Knoten des Graphen in zwei Mengen.
(Schnitt teilt die Knotenmenge in S und $V - S$)
- $\{u,v\}$ überbrückt $(S, V-S)$, wenn $u \in S$ und $v \in S$.
(Kante überbrückt Schnitt, wenn die Knoten der Kante in unterschiedlichen Teilen des Schnitts sind)
- $(S, V-S)$ respektiert A $\subseteq E$, wenn keine Kante $\{u,v\}$ aus A den Schnitt überbrückt.
(Es existiert noch keine Kante in A, die den Schnitt überbrückt)
- $\{u,v\}$ leichte Kante für $(S, V-S)$, wenn $w(u,v)$ minimal für alle Kanten, die den Schnitt überbrücken.
(Kante ist die leichteste Kante, wenn es keine leichtere Kante gibt, die den Schnitt überbrückt)

8.5 (a) Kruskals Algorithmus

Kruskals Algorithmus funktioniert nach dem Prinzip, dass erstmal für alle Knoten eine Untermenge erstellt wird. Anschließend werden alle Kanten des Graphen nach Gewicht sortiert und dann in aufsteigender Reihenfolge durchlaufen. Hierbei werden die beiden Knoten der betrachteten Kante überprüft, ob sie der gleichen Menge angehören. Falls ja, bedeutet dies, dass es schon einen Pfad im MST zwischen den beiden Knoten gibt, andernfalls wird die leichteste Kante zum MST hinzugefügt und die beiden Mengen, denen die Knoten angehören vereinigt. Wenn alle Kanten durchlaufen wurden ist der MST fertig konstruiert und kann zurückgegeben werden.

```

1 // Complexity:  $O(|E| \cdot \log |E|)$  or  $O(|E| \cdot \log |V|)$  ( $\log |E| = \Theta \log |V|$ , because  $|V| - 1 \leq |E| \leq |V|^2$ )
2 Function MSTKruskal( $G, w$ ):
3    $A = \emptyset$ ; // Empty Set of edges →represents MST
4   ForEach  $v$  in  $V$  do
5      $set(v) = \{v\}$ ; // Creates a set for each node, containing only that node
6   sort edges according to weight in increasing order;
7   // Iterate through all edges in increasing order of weight
8   ForEach  $\{u,v\} \in E$  do
9     // if the sets are the same the nodes already belong to the same set, that means that an edge
      connecting the two nodes is already in the MST →no new edge should be added to avoid
      cycles
10    If  $set(u) \neq set(v)$  then
11       $A = A \cup \{u,v\}$ ; // Add edge to MST
12      union( $G, u, v$ ); // unite the two sets
13
14  return  $A$ ; // Return MST

```

8.5 (b) Prims Algorithmus

Der Prim Algorithmus funktioniert nach dem Prinzip, dass es für jeden Knoten alle Kanten zu seinen Nachbarn durchgeht um die Kante mit dem kleinsten Gewicht zu finden. Der MST wird durch die `pred` Referenz der Knoten gebildet.

```

1 // Complexity:  $O(|E| + |V| \cdot \log |V|)$ 
2 // r is the starting node →works with any node in the graph
3 Function MSTPrim( $G, w, r$ ):
4   ForEach  $v$  in  $V$  do
5      $v.key = \infty$ ;  $v.pred = NIL$ ; // key represents lightest weight of an edge to  $v$  in the MST. MST
      not created yet →start with biggest value.
6    $r.key = -\infty$ ; // Starting nodes key is set to  $-\infty$  as its supposed to be worked on first.
7    $Q = V$ ; //  $Q$  is a queue containing all nodes.
8   // While there are still nodes that have not been worked on:
9   While !isEmpty( $Q$ ) do
10     $u = extractMin(Q)$ ; // Extract the node with the smallest key
11    // For all nodes adjacent to  $u$ :
12    ForEach  $v$  in  $Adj(u)$  do
13      // If  $v$  has not been visited yet and the weight of the edge is smaller than the current
        lightest edge to  $v$ :
14      If  $v \in Q$  and  $w(u,v) < v.key$  then
15         $v.key = w(u,v)$ ; // Update the key of  $v$ 
16         $v.pred = u$ ; // Update the predecessor of  $v$ 

```

Sowohl Kruskals als auch Prims Algorithmus funktionieren nicht für gerichtete Graphen. Sie finden zwar beide **einen** Spannbaum, aber nicht immer den **minimalen** Spannbaum.

Allgemeiner MST-Algorithmus: Idee

```
genericMST(G,w) // G=(V,E) undirected, connected graph
                  w weight function
1   A=∅
2   WHILE A does not form a spanning tree for G DO
3       find safe edge {u,v} for A
4       A = A ∪ {{u,v}}
5   return A
```

A Teilmenge der Kanten eines MST

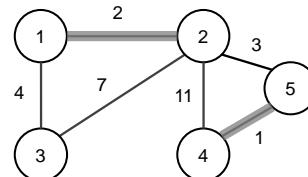
Kante $\{u,v\}$ ist sicher („safe“) für A,
wenn $A \cup \{\{u,v\}\}$ noch Teilmenge eines MST ist

Algorithmus von Kruskal

$\text{UNION}(G,u,v)$ setzt $set(w) = set(u) \cup set(v)$
für alle Knoten $w \in set(u) \cup set(v)$

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                  w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5       IF set(u) != set(v) THEN
6           A = A ∪ {{u,v}}
7           UNION(G,u,v);
8   return A
```

wenn $set(u) == set(v)$,
dann wären Knoten schon
verbunden und $\{u,v\}$ erzeugte Zyklus



Zu jedem Knoten v sei $set(v)$ Menge von mit v durch A verbundenen Knoten.
Zu Beginn ist $set(v) = \{v\}$.

$set(u), set(v)$ sind disjunkt oder identisch

Im Beispiel $set(1) = \{1,2\}, set(4) = \{4,5\}$.

Algorithmus von Prim

(Wahl des Wurzelknoten beliebig)

```
MST-Prim(G,w,r) // r root in V, MST given through v.pred values
1   FOREACH v in V DO {v.key=∞; v.pred=NULL;}
2   r.key=-∞; Q=V;
3   WHILE !isEmpty(Q) DO
4       u=EXTRACT-MIN(Q); //smallest key value
5       FOREACH v in adj(u) DO
6           IF v∈Q and w({u,v})<v.key THEN
7               v.key=w({u,v});
8               v.pred=u;
```

Idee: Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zu zusammenhängender Menge hinzu

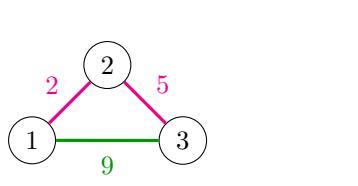
Auswahl der nächsten Kante gemäß **key**-Wert, der stets aktualisiert wird

A implizit definiert durch
 $A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$

8.6 Kürzesten Pfade

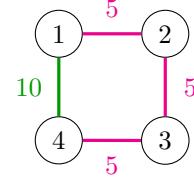
8.6 (a) Single Source Shortest Path (SSSP)

Bei den SSSP Algorithmen geht es darum, den Pfad mit dem geringsten Gewicht von einem Knoten zu einem anderen zu finden. Damit unterscheiden sie sich von den BFS Algorithmus, da dieser Algorithmus zwar den kürzesten Weg gemäß Kantenanzahl findet, jedoch Gewicht ignoriert. Zudem unterscheiden sie sich auch von den MST Algorithmen, die zwar die Wege mit den geringsten Gewichten finden, jedoch müssen alle Knoten in diesem Graph enthalten sein, wodurch sie eventuell nicht den kürzesten Weg zwischen zwei spezifischen Knoten finden.



BFS

Kürzester Pfad, aber nicht geringstes Gewicht.



MST

Spiegelt nicht den leichtesten Pfad $1 \rightarrow 4$ wieder.

Ein paar Regeln für SSSP Algorithmen sind, dass Zyklen nicht Teil des leichtesten Pfades sein können. Positiv gewichtete Zyklen würden nur Gewicht hinzufügen, was nicht gewollt ist. Negative könnten beliebig oft durchlaufen werden für einen beliebig leichten Pfad, was praktisch keinen Sinn ergibt. Negative Kanten sind okay, Negative Zyklen nicht.

Ein Teilstumpf $s \rightarrow x$ des kürzesten Pfads $s \rightarrow x \rightarrow z$ ist auch immer der kürzeste Pfad von $s \rightarrow x$.

Alle Algorithmen für SSSP funktionieren über das Konzept von **Relaxation**. Hierbei wird die Distanz eines Knotens v verringert, wenn durch eine anliegende Kante (u,v) eine kürzere Distanz möglich ist.

```

1 // G: graph with (V,E), u and v nodes, w weight function
2 Function relax(G, u, v, w):
3   // if the distance of the node v is bigger than the distance of the node u + the weight of the
   // edge (u,v)
4   If v.dist > u.dist + w(u, v) then
5     v.dist = u.dist + w(u, v); // Assign the new distance to the node v
6     v.pred = u; // Update predecessor of v

```

Die Initialisierung bleibt auch gleich:

```

1 // G: graph with (V,E), s startnode, w weight function
2 Function initSSSP(G, s, w):
3   ForEach v in V do
4     v.dist = infinity; // Set all distances to infinity
5     v.pred = NIL; // Set predecessor to NIL
6   s.dist = 0; // Set the distance of the startnode to 0

```

8.6 (b) Bellman-Ford Algorithmus

```
1 // Complexity:  $\Theta(|E| \cdot |V|)$ 
2 // G: graph with (V,E), s startnode, w weight function
3 Function BellmanFordSSSP(G, s, w):
4     initSSSP(G, s, w); // Set all distances (except s) to  $\infty$  and the predecessor to NIL
5     // Runs once for every Vertices - s since the predecessor of s is NIL  $\rightarrow$  Shortest path has at
       most  $|V|-1$  edges
6     For i = 1 to  $|V|-1$  do
7         // for every edge in the graph
8         ForEach (u,v) in E do
9             relax(G, u, v, w); // Relax every edge
10    // At this point the prev references should make up the lightest path from each node to s
11    // This does not yet check for negative cycles, which would falsify the result.
12    // Check every edge for negative cycles
13    ForEach (u,v) in E do
14        // if the edge after relaxation can still be improved, that implies a negative cycle
15        If v.dist > u.dist + w(u, v) then
16            return false; // Negative cycle  $\rightarrow$ return false
17    return true; // No negative cycle  $\rightarrow$ return true
```

8.6 (c) DAG (Directed Acyclic Graph) Algorithmus

Dieser Algorithmus hat zwar im Vergleich zu Bellman-Ford eine bessere Komplexität, ist aber nur für azyklische Graphen geeignet. Er nutzt die Eigenschaft der topologischen Reihenfolge um Kanten nicht öfter angehen zu müssen.

```
1 // Complexity:  $\Theta(|E| + |V|)$ 
2 Function DAGTopoSSSP(G, s, w):
3     initSSSP(G, s, w); // Set all distances (except s) to  $\infty$  and the predecessor to NIL
4     execute topological sorting; // topological sorted graph is graph in which each vertex u appears
       before every vertex v for all edges (u,v). Only for acyclic graphs
5     // for every vertex in topological order. Processing them in this order ensures that for a
       vertex u all vertices with edges to u have already been processed.
6     ForEach u in V (in topological order) do
7         // for every adjacent vertex v
8         ForEach v in adj(u) do
9             relax(G, u, v, w); // Relax every adjacent edge
```

8.6 (d) Dijkstra Algorithmus

Dijkstras Algorithmus funktioniert zwar auch für zyklische Graphen, jedoch hat er Probleme mit negativen Kanten. Diese können aber entfernt werden, indem man alle Kantengewichte mit dem absoluten Wert der leichtesten negativen Kante zusammenaddiert.

```
1 // Complexity:  $\Theta(|V| \cdot \log |V| + |E|)$ 
2 Function DijkstraSSSP( $G, s, w$ ):
3   initSSSP( $G, s, w$ ); // Set all distances (except  $s$ ) to  $\infty$  and the predecessor to NIL
4    $Q = V$ ; //  $Q$  is a queue containing all nodes.
5   While !isEmpty( $Q$ ) do
6      $u = \text{extractMin}(Q)$ ; // Extract the node with the smallest key
7     // For all nodes adjacent to  $u$ :
8     ForEach  $v$  in  $\text{Adj}(u)$  do
9       relax( $G, u, v, w$ ); // Relax every adjacent edge
```

8.6 (e) A* Algorithmus

Der A* Algorithmus ist eine abgewandelte Version des Dijkstra Algorithmus. Dabei geht der Algorithmus auf das Problem ein, dass der Dijkstra Algorithmus den lokal gesehenen günstigsten Schritt sucht, dabei aber die Zielrichtung ignoriert, was zu überflüssigen Operationen führen kann.

Ist in der Praxis oft schneller als Dijkstra, hat aber den selben Nachteil, dass er nicht mit negativen Kanten klarkommt. Zusätzlich ist die Spacial complexity schlechter als Dijkstra, wegen der zusätzlichen heuristic.

```
1 // Complexity:  $\Theta(|V| \cdot \log |V| + |E|)$ 
2 //  $s$  start,  $t$  target
3 Function A*SSSP( $G, s, t, w$ ):
4   initSSSP( $G, s, w$ ); // Set all distances (except  $s$ ) to  $\infty$  and the predecessor to NIL
5    $Q = V$ ; //  $Q$  is a queue containing all nodes. Additionally the nodes are sorted according to the
      // distance AND heuristic value of the vertex which ensure that the nodes closer to the target
      // are processed first.
6   While !isEmpty( $Q$ ) do
7      $u = \text{extractMin}(Q)$ ; // Extract the node with the smallest key
8     // If  $u$  is the target:
9     If  $u == t$  then
10       break;
11     // For all nodes adjacent to  $u$ :
12     ForEach  $v$  in  $\text{Adj}(u)$  do
13       relax( $G, u, v, w$ ); // Relax every adjacent edge
```

Bei Ungerichteten Graphen können zwar Bellman-Ford und Dijkstra ausgeführt werden, jedoch muss für Bellman-Ford beachtet werden, dass keine Kante ein negatives Gewicht haben kann, da diese direkt einen negativen Zyklus impliziert. Wenn man aber bei Dijkstra die Gewichte direkt alle positiv macht kann dieser bedenkslos ausgeführt werden.

Nimmt man an, dass die Kanten nicht negativ sind, ist es besser direkt Dijkstra zu nutzen.

Die Kanten $\{u,v\}$ entsprechen sogesehen den Kanten (u,v) und (v,u) . (Alle gleiches Gewicht.)

Relax!

Idee: verringere aktuelle Distanz von Knoten v , wenn durch Kante (u, v) kürzere Distanz 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;
```



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 129

Zu Beginn
Distanz = ∞
für alle Knoten $\neq s$

Bellman-Ford-Algorithmus

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
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= $\infty$ ;
3   v.pred=NULL;
4 s.dist=0;
```

prüft zusätzlich,
ob „negativer Zyklus“
erreichbar (=false)

Laufzeit = $\theta(|E| \cdot |V|)$

wegen geschachtelter
FOR-Schleifen in 2 und 3

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 130



A*-Algorithmus (II)

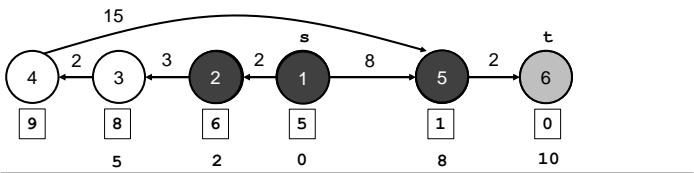
(nicht-negative Kantengewichte)

Idee: füge Heuristik hinzu, die „vom Ziel her denkt“



```
A*(G,s,t,w)
1 init(G,s,t,w);
2 Q=V; //let S=V-Q-
3 WHILE !isEmpty(Q) DO
4   u=EXTRACT-MIN(Q); +
5   IF u==t THEN break;
6   FOREACH v in adj(u) DO
7     relax(G,u,v,w);
```

jeder Knoten u bekommt
zusätzlich Wert $u.\text{heur}$ zugewiesen
(Beispiel: Abstand Luftlinie vom Ziel)
Minimum über
 $u.\text{dist} + u.\text{heur}$



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 174



Bellman-Ford: Beispiel (I)

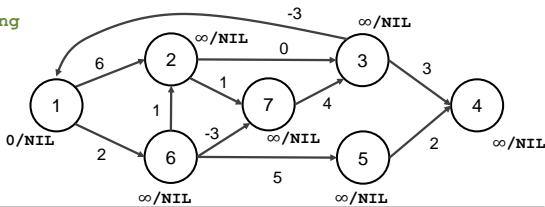
Bellman-Ford-SSSP(G, s, w)

```

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
6    IF v.dist > u.dist+w((u,v)) THEN
7      return false;
8  return true;

```

Initialisierung
für $s=1$ in 1



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 137



Bellman-Ford: Beispiel (II)

Bellman-Ford: Beispiel (II)

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
6    IF v.dist > u.dist+w((u,v)) THEN
7      return false;
8  return true;

```

Kanten in FOREACH in 3
gemäß lexikographischer
Ordnung: (1,2), (1,6), (2,3), ...

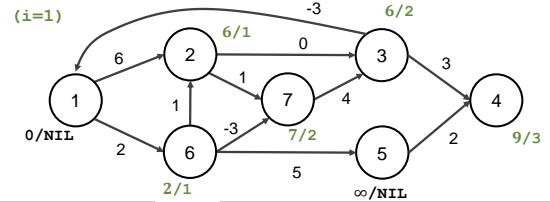
FOR-Schleife 2 (i=1)

```

relax(1,2)
relax(1,6)
relax(2,3)
relax(2,7)
relax(3,1)
relax(3,4)
relax(5,4)

```

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 138



Bellman-Ford: Beispiel (V)

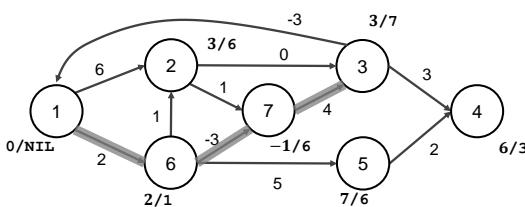
Bellman-Ford-SSSP(G, s, w)

```

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
6    IF v.dist > u.dist+w((u,v)) THEN
7      return false;
8  return true;

```

Kürzester Weg
z.B. von 1 zu 3
durch
Vorgängerwerte
gegeben



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 141



Keine Änderungen
mehr in den
folgenden Iterationen

Algorithmus gibt
true zurück

Bellman-Ford: Beispiel (IV)

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
6    IF v.dist > u.dist+w((u,v)) THEN
7      return false;
8  return true;

```

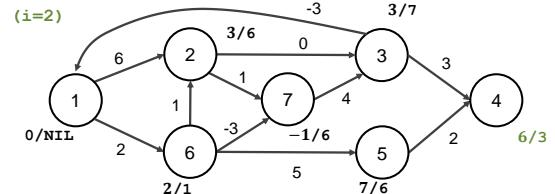
FOR-Schleife 2 (i=2)

```

relax(1,2)
relax(1,6)
relax(2,3)
relax(2,7)
relax(3,1)
relax(3,4)
relax(5,4)

```

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 140

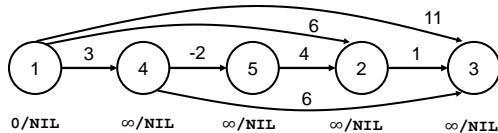


SSSP-Algorithmus für Dags: Beispiel (I)

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

```
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);
```

Initialisierung für $s=1$ in 1
und Sortieren in 2



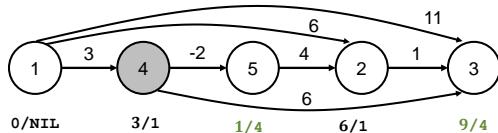
Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 144

SSSP-Algorithmus für Dags: Beispiel (II)

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

```
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);
```

3 FOREACH ($u=1$)



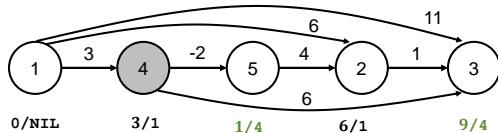
Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 145

SSSP-Algorithmus für Dags: Beispiel (III)

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

```
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);
```

3 FOREACH ($u=4$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 146

SSSP-Algorithmus für Dags: Beispiel (IV)

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

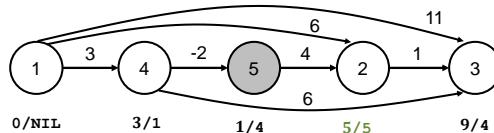
```
1 initSSSP(G,s,w);
2 execute topological sorting
3 FOREACH u in V in topological order DO
4   FOREACH v in adj(u) DO
5     relax(G,u,v,w);
```

SSSP-Algorithmus für Dags: Beispiel (IV)

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

```
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);
```

3 FOREACH ($u=5$)



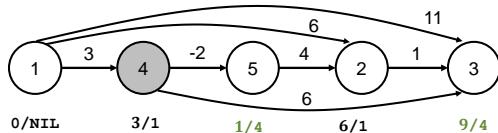
Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 147

SSSP-Algorithmus für Dags: Beispiel (V)

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

```
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);
```

3 FOREACH ($u=2$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 148

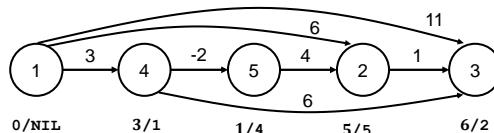
SSSP-Algorithmus für Dags: Korrektheit+Laufzeit

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

```
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);
```

Korrektheit:
Kanten auf einem
kürzesten Pfad
werden nacheinander
„gelockert“
(vgl. Bellman-Ford)

$$\text{Laufzeit} = \theta(|E| + |V|)$$



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 149

Dijkstra-Algorithmus: Beispiel (III)

Dijkstra-SSSP(G, s, w)

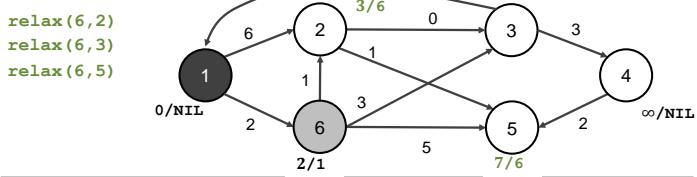
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

3 WHILE ($u=6$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 154

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

Dijkstra-Algorithmus: Beispiel (IV)

Dijkstra-Algorithmus: Beispiel (IV)

Dijkstra-SSSP(G, s, w)

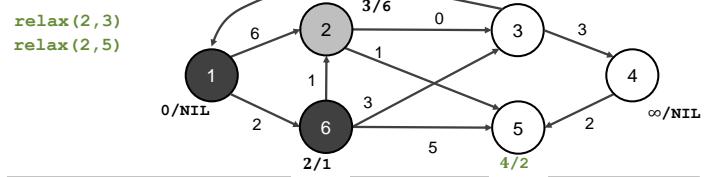
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

3 WHILE ($u=2$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 155

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

Dijkstra-Algorithmus: Beispiel (V)

Dijkstra-SSSP(G, s, w)

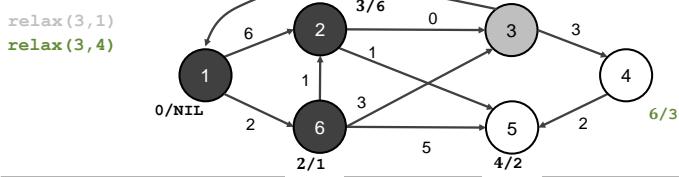
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

3 WHILE ($u=3$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 156

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

Dijkstra-Algorithmus: Beispiel (VI)

Dijkstra-SSSP(G, s, w)

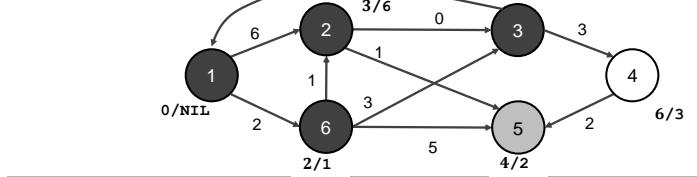
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

3 WHILE ($u=5$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 157

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

Dijkstra-Algorithmus: Beispiel (VII)

Dijkstra-SSSP(G, s, w)

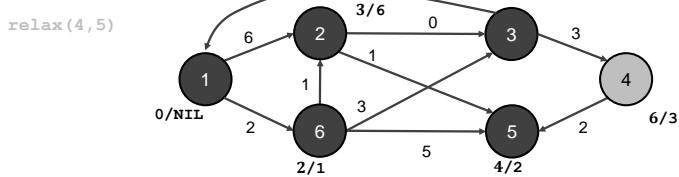
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

3 WHILE ($u=4$)



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 158

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

Dijkstra-Algorithmus: Beispiel (VIII)

Dijkstra-SSSP(G, s, w)

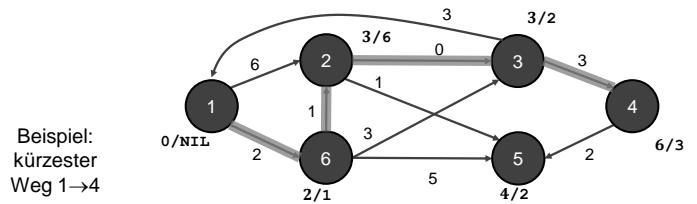
```

1  initSSSP( $G, s, w$ );
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q); //wrt. dist
5      FOREACH v in adj(u) DO
6          relax( $G, u, v, w$ );

```

Voraussetzung:
 $w((u, v)) \geq 0$
 für alle Kanten

Beispiel:
 kürzester
 Weg 1→4



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 159

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

8.7 Maximaler Fluss

Das grundsätzliche Konzept von Flüssen ist, dass Graphen sogesehen nicht nur ein Gewicht haben, sondern zwei. Dabei repräsentiert eines den aktuellen Fluss und das andere den maximalen. Bei Flussgraphen gilt dann, dass alle Knoten außer Start s und Target t gleichen eingehenden und ausgehenden Fluss haben. Formal:

Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazitätsgewicht 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 $s, t \in V$ (Quelle, Senke), sodass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist.

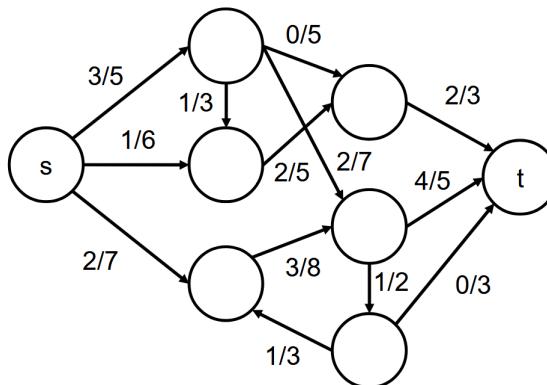
Zusätzlich ist der Fluss einer Kante minimal 0 sein kann und maximal der Kapazität der Kante entsprechen kann. Zusätzlich ist der Gesamteinfluss gleich dem Gesamt[ausfluss](#) eines Knoten. (Input = Output) Formal:

Ein Fluss $f: V \times V \rightarrow \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazitätsgewicht c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für $(u, v) \in E$. Zusätzlich gilt für $u \in V - \{s, t\}$: $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$

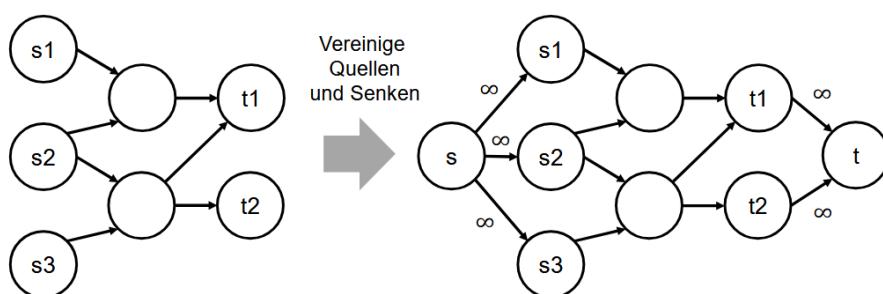
Außerdem gibt es noch den Wert eines Flusses. dieser spiegelt wieder, wie viel Fluss insgesamt von der Quelle zur Senke transportiert wird (Prinzipiell Fluss der Quelle: Ausfluss der Quelle - Einfluss der Quelle). Formal:

Der Wert $|f|$ eines Flusses $f: V \times V \rightarrow \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t ist $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$ (Ausfluss der Quelle - Einfluss der Quelle)

Der maximale Fluss beschreibt den maximalen Ausfluss der Quelle, der, unter der Beachtung der Kapazitäten in die Senke fließen kann.



$|f| = 6$, nicht maximal, da z.B. über obere Kanten noch +1 könnte.



Verschiedene Transformationen von Flüssen

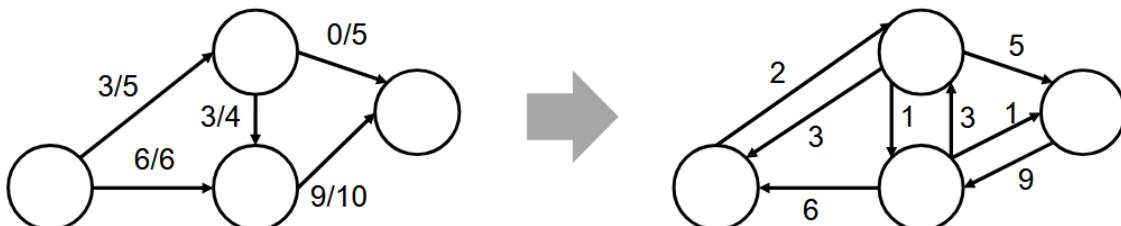
8.7 (a) Ford-Fulkerson Algorithmus

Der Ford-Fulkerson Methode zufolge wird im Flussgraphen ein Pfad von s zu t gesucht, der noch erweiterbar bzgl. des Flusses ist. Hierbei wird aber nicht der eigentlich Graph genutzt, sondern der sogenannte Restkapazitätsgraph. Dieser Graph stellt die Restkapazitäten einer Kante aufgeteilt auf Vorwärts- und Rückwärtskante dar. Dabei gilt für die Vorwärtskante $c_f(u, v) = c(u, v) - f(u, v)$ (Wie viel freie Kapazität die Kante hat), wenn $(u, v) \in E$ und die Rückwärtskante $c_f(v, u) = f(v, u)$ (Wie viel Kapazität die Kante schon besetzt), wenn $(u, v) \in E$. Andernfalls ist $c_f(u, v) = 0$.

Restkapazität:

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

Formelle Restkapazität



Restkapazitätsgraph

Im Restkapazitätsgraph ist es nun also so, dass wenn eine Kante von einem Knoten wegführt, dieser noch Kapazität übrig hat und der Fluss an dieser Kante erhöht werden kann. Demnach muss in diesem Graphen jetzt nur noch ein Pfad von s nach t gefunden werden, dessen Existenz impliziert, dass der maximale Fluss noch nicht erreicht ist. Demnach können auf diesem Pfad die Restkapazitäten angepasst werden. Formell:

Finde Pfad von s zu t in G_f und erhöhe (für Kanten in G) bzw. erniedrige (für nicht-Kanten) um Minimum $c_f(u, v)$ aller Werte auf dem Pfad in G

```

1 // Complexity:  $O(|E| \cdot u \cdot |f^*|)$ ,  $u = \max \text{ capacity}$ ,  $|f^*| = \max \text{ flow}$ 
2 //  $s$  start,  $t$  target,  $c$  capacity function
3 Function FordFulkerson( $G, s, t, c$ ):
4   ForEach  $e$  in  $E$  do
5      $e.\text{flow} = 0$ ; // Initializes all flow values to 0
6   // goes through every possible path with free capacity
7   While path  $p$  from  $s$  to  $t$  exists in  $G_f$  do
8      $c_f(p) = \min\{c_f(u, v) : (u, v) \in p\}$ ; // Minimum of all flow values on the path
9     // Changes the flow values on the path
10    ForEach  $e$  in  $p$  do
11      // If edge  $e$  in  $E \rightarrow$  total flow, else remaining capacity
12      If  $e$  in  $E$  then
13         $e.\text{flow} += c_f(p)$ ; // Add minimal flow to edge
14      Else
15         $e.\text{flow} -= c_f(p)$ ; // Subtract minimal flow from edge

```

Ford-Fulkerson-Algorithmus

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  $G_{flow}$  DO
3    $c_{flow}(p) = \min\{c_{flow}(u, v) \mid (u, v) \text{ in } p\}$ 
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow=e.flow+ $c_{flow}(p)$ 
7     ELSE
8       e.flow=e.flow- $c_{flow}(p)$ 
```

Pfadsuche z.B. per BFS oder DFS

Z.B. wenn
in jeder Iteration
der Fluss nur
um $1/u = 0.1$
erhöht wird

Laufzeit = $O(|E| \cdot u \cdot |f^*|)$

(wobei f^* maximaler Fluss
und Fluss um bis zu $1/u$ pro Iteration wächst)

Laufzeit = $O(|V| \cdot |E|^2)$

(mit Verbesserung
nach Edmonds-Karp)

Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 187

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

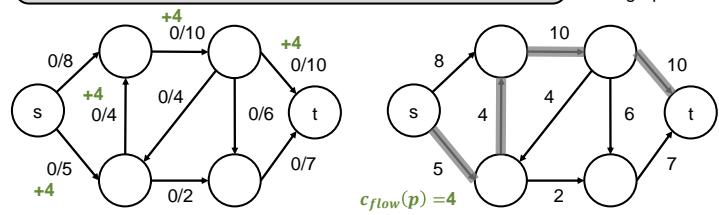
Ford-Fulkerson-Algorithmus: Beispiel (I)

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  $G_{flow}$  DO
3    $c_{flow}(p) = \min\{c_{flow}(u, v) \mid (u, v) \text{ in } p\}$ 
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow=e.flow+ $c_{flow}(p)$ 
7     ELSE
8       e.flow=e.flow- $c_{flow}(p)$ 
```

Restkapazitäts-
graph



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 188

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

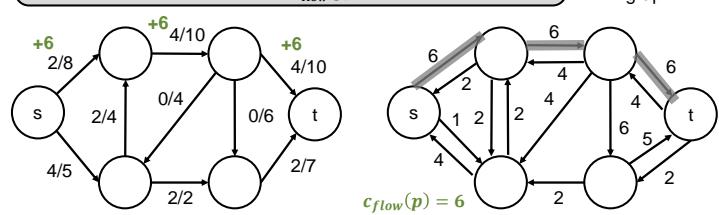
Ford-Fulkerson-Algorithmus: Beispiel (II)

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  $G_{flow}$  DO
3    $c_{flow}(p) = \min\{c_{flow}(u, v) \mid (u, v) \text{ in } p\}$ 
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow=e.flow+ $c_{flow}(p)$ 
7     ELSE
8       e.flow=e.flow- $c_{flow}(p)$ 
```

Restkapazitäts-
graph



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 189

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

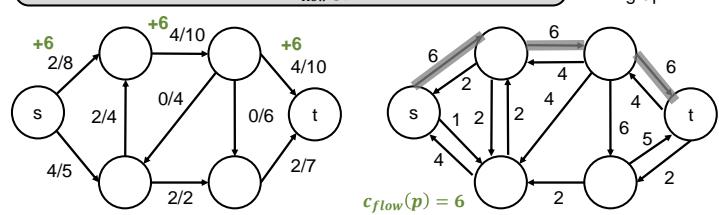
Ford-Fulkerson-Algorithmus: Beispiel (III)

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  $G_{flow}$  DO
3    $c_{flow}(p) = \min\{c_{flow}(u, v) \mid (u, v) \text{ in } p\}$ 
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow=e.flow+ $c_{flow}(p)$ 
7     ELSE
8       e.flow=e.flow- $c_{flow}(p)$ 
```

Restkapazitäts-
graph



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 190

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

Ford-Fulkerson-Algorithmus: Beispiel (IV)

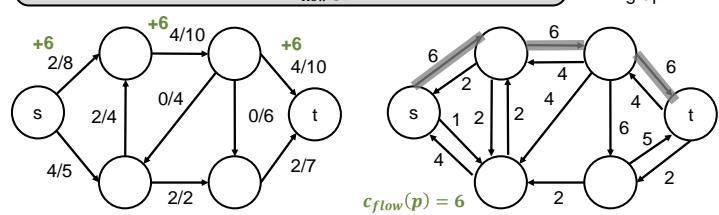
Ford-Fulkerson(G, s, t, c)

```

1 FOREACH e in E DO e.flow=0;
2 WHILE there is path p from s to t in  $G_{flow}$  DO
3    $c_{flow}(p) = \min\{c_{flow}(u, v) \mid (u, v) \text{ in } p\}$ 
4   FOREACH e in p DO
5     IF e in E THEN
6       e.flow=e.flow+ $c_{flow}(p)$ 
7     ELSE
8       e.flow=e.flow- $c_{flow}(p)$ 
```

Kein Pfad
mehr im
Restkapazitäts-
Graph

Restkapazitäts-
graph



Algorithmen und Datenstrukturen | Marc Fischlin | SS24 | 06 Graphen-Algorithmen | 191

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptophly

9 Advanced Design

9.1 Divide & Conquer

Das Divide & Conquer Prinzip ist ein Prinzip um komplexe Probleme effizient zu lösen. Dabei wird das Hauptproblem in kleiner leichter lösbare Teilprobleme zerlegt. Dies geschieht meist rekursiv. Diese kleineren Teilprobleme werden dann separat einfach gelöst und kombiniert, um das ursprüngliche Problem zu lösen. Diese Art der Problemlösung ist oft sehr effizient und bildet nicht nur in der Theorie oft eine effizientere Lösung, sondern auch praktisch, da die Teilprobleme oft parallel gelöst werden können.

9.1 (a) Fourier-Transformation

Das Allgemeine Konzept der Fourier-Transformation ist die Lösung eines Problems, welches in ungünstiger Form vorliegt. Demnach wird das Problem erst in eine geeigneter Darstellung umgewandelt, diese dann bearbeitet und anschließend zurückgewandelt. Dies kann im Endeffekt schneller sein als das Problem in der ungünstigen Form zu bearbeiten. Als Beispiel wird im folgenden die Polynommultiplikation verwendet.

Polynome liegen normalerweise in der Koeffizientendarstellung vor:

$$p(x) = p_0 + p_1x + p_2x^2 + \dots + p_{n-2}x^{n-2} + p_{n-1}x^{n-1}$$

mit $p_{n-1} \neq 0$ und $\text{grad}(p(x)) = n - 1$

Wenn so ein Polynom beispielsweise als Array der Koeffizienten gegeben ist, so kann man ein Polynom an der Stelle $x = w$ einfach mit dem folgenden Algorithmus ausrechnen:

```
1 // Complexity: Θ(n)
2 // p: Array of coefficients, w: evaluation point(x)
3 Function polyEval(p, w):
4     n = length(p);
5     result = p[n-1]; // Initializes result to last (biggest) coefficient
6     // Runs from last to first coefficient, excluding the last one as its already in result
7     For i = n - 2 down to 0 do
8         result = result * w + p[i]; // Because of the loop each coefficient gets multiplied as often as
9         it needs to
9     return result;
```

So ergibt sich essenziell:

$$p(x) = (((p_{n-1}x + p_{n-2})x + p_{n-3}) \dots p_1)x + p_0$$

Als konkretes Beispiel kann man zum Beispiel das Polynom $p(x) = 2 + 3x + 5x^2$ und $w = 2$ nehmen. Das Array würde dementsprechend dann so aussehen: $p[] = [2, 3, 5]$. So würde die letztendliche Rechenoperation so aussehen:

$$p(2) = (5 \cdot 2 + 3) \cdot 2 + 2 = 28$$

Das Multiplizieren zweier Polynome ist allerdings schon komplexer. So wäre die Multiplikation zweier Polynome vom Grad $n-1$ formell:

$$p(x) \cdot q(x) = (\sum_{i=0}^{n-1} p_i x^i) \cdot (\sum_{i=0}^{n-1} q_i x^i) = \sum_{k=0}^{2n-2} (\sum_{j=0}^k p_j \cdot q_{k-j}) x^k$$

Vom Grad $2n-2$

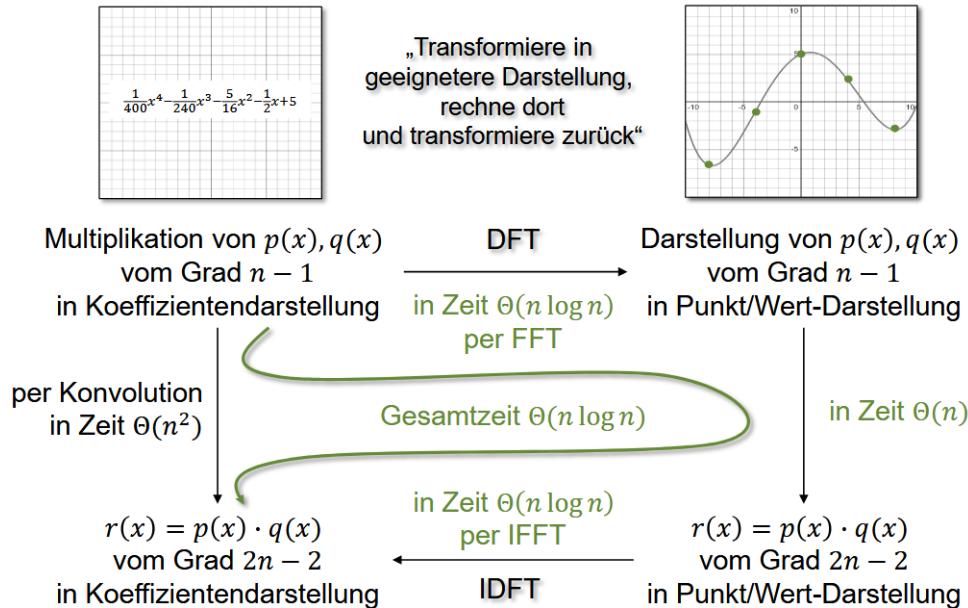
Dies kann mit dem folgenden Algorithmus dargestellt werden:

```

1 // Complexity: Θ(n2)
2 // p: Array of coefficients, q: Array of coefficients
3 Function polyMult(p, q):
4     n = length(p);
5     result = new Array(2n - 1); // Initializes result to new Array of length 2n - 1
6     // For all coefficients that are gonna be in the result
7     For k = 0 to 2n - 2 do
8         result[k] = 0; // Initializes result[k] to 0
9         // For all coefficients smaller-equal the current one
10        For j = 0 to k do
11            result[k] = result[k] + p[j] * q[k - j]; // Multiplies current result with the corresponding
12            coefficients at j in p and k-j in q
13
14    return result;

```

Dies ist aber mit $\Theta(n^2)$ relativ langsam. Mithilfe der Fourier-Transformation kann man die Rechenzeit auf $\Theta(n \log n)$ reduzieren, indem man die Koeffizientendarstellung in Punkt/Wert Darstellung umwandelt. Das Allgemeine Schema sieht dann also so aus:



Die Punkt/Wert Darstellung eines Polynoms kann auch wieder über ein Array dargestellt werden, bei dem jeder Eintrag $p[i]$ $p[i].x$ und $p[i].y$ besitzt.

So kann man in dieser Form das Produkt $r(x) = p(x) \cdot q(x)$ leicht in $\Theta(n)$ ausrechnen, indem man für jede x -Koordinate x_j in $j = 0, 1, \dots, 2n-1$ $r(x_j) = p(x_j) \cdot q(x_j)$ ausrechnet. Die x Koordinate korrespondiert dann mit dem Grad des Koeffizienten.

```

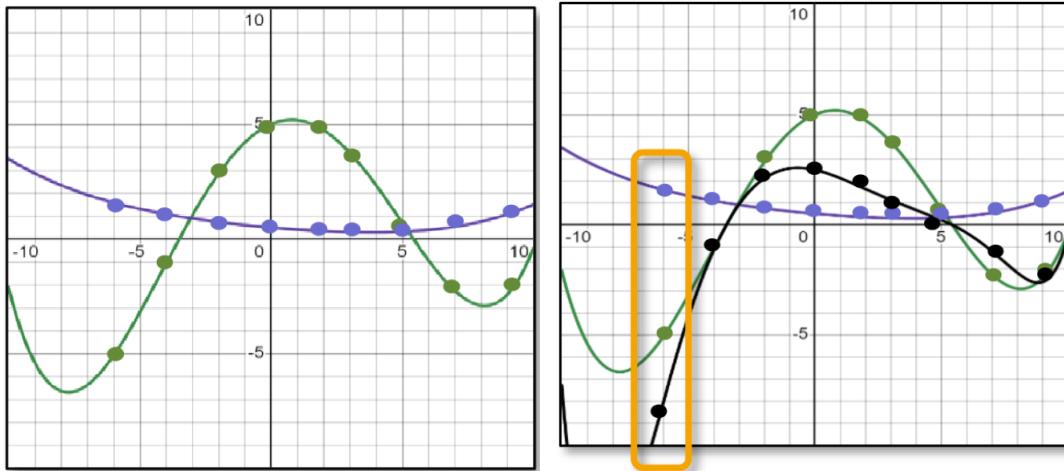
1 // Complexity: Θ(n)
2 // p: Array of coefficients, q: Array of coefficients
3 Function polyMult(p, q):
4     n = length(p);
5     result = new Array(2n - 1); // Initializes result to new Array of length 2n - 1
6     // For all coefficients that are gonna be in the result
7     For i = 0 to 2n - 2 do
8         result[i].x = p[i].x; // x stays the same
9         result[i].y = p[i].y * q[i].y; // y of the result is the product of y of p and y of q
10    return result;

```

Nun brauchen wir nur noch die Umwandlung von Koeffizientendarstellung in Punkt/Wert Darstellung und vice versa. Diese wird in der Fourier Transformation als *Discrete Fourier-Transform* und *Inverse Discrete Fourier-Transform* bezeichnet.

Die Fourier-Transformation bei Polynomen geht nach dem folgenden Schema:

1. Schreibe $p(x)$ in Form $p(x) = p_{even}(x^2) + x \cdot P_{odd}(x^2)$ mit $p_{even}(x) = p_0 + p_2x + \dots + p_{n-2}x^{(n-2)/2}$ und $p_{odd}(x) = p_1 + p_3x + \dots + p_{n-1}x^{(n-2)/2}$
p_{even} und p_{odd} sind nun jeweils Polynome von ca. halben Grad.
2. Nun können wir Werte x_j nutzen, so dass $x_j^2 = x_{j+n}^2$ für alle $j = 0, 1, \dots, n-1$ und $x_j \neq x_{j+n}$.
Somit müssen p_{odd} und p_{even} vom Grad $\frac{n-2}{2} = \frac{n}{2} - 1$ nur an n Stellen $x_0^2, x_1^2, \dots, x_{n-1}^2$ gewertet werden.
Demnach ist die Problemgröße halbiert und das Divide and Conquer Prinzip kann angewendet werden.



```

1 // Complexity: Θ(n log n)
2 // p: Array of coefficients, w: unit root (starts at 2n)
3 Function polyFFT(p, w):
4     n = length(p);
5     pEven = new Array(n/2); pOdd = new Array(n/2); // Initializes pEven and pOdd to new Arrays of
       length n/2
6     pVal = new Array(2n); pEvenVal = new Array(n); pOddVal = new Array(n); // Initializes pVal and
       pEvenVal and pOddVal to new Arrays of length 2n and n respectively
7     x = new Array(2n); x[0] = 1; // initializes x to new Array of length 2n, holds the x values of the
       coefficient
8     For i = 1 to 2n-1 do
9         x[i] = w * x[i - 1]; // Essentially does x[i] = w^i
10    // If the polynomial is constant, only needs to assign first and second coefficient. Base-Case.
11    If n == 1 then
12        pVal[0].x = x[0]; pVal[0].y = p[0].y; // Assigns first coefficient.x the corresponding value from
           x[] and the corresponding value from the polynomial
13    Else
14        // Fills pEven and pOdd with the values from p
15        For i = 0 to (n-2)/2 do
16            pEven[i] = p[2i]; pOdd[i] = p[2i + 1];
17        // Recursively call polyFFT on pEven and pOdd
18        pEvenVal = polyFFT(pEven, w*w); // Evaluates at x_j^2
19        pOddVal = polyFFT(pOdd, w*w);
20        // Essentially calculates p(x_j) = pEven(x_j^2) + x_j * pOdd(x_j^2) for all elements in pVal
21        For i = 0 to 2n-1 do
22            pVal[i].x = x[i]; // Assigns coefficient.x the corresponding value from x[]
23            pVal[i].y = pEvenVal[i % n] + x[i] * pOddVal[i % n].y;
24    return pVal;

```

Die Punkt/Wert Darstellung kann prinzipiell wie folgt dargestellt werden, wo x die Polynome für jedes x_j beinhaltet, p die Koeffizienten und y die Werte. Dabei werden in dem Ergebnis des Algorithmus die Koeffizienten ausgelassen.

$$\begin{pmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & \dots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Nun muss noch der letzte Schritt der Fourier Transformation auf Polynommultiplikation angewendet werden, die Rückwandlung in die ursprüngliche Form (Koeffizientendarstellung). So gilt für die Koeffizienten p:

$$\begin{pmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & \dots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix}^{-1} \cdot \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

Dies bedeutet also, dass wir um von der Punkt/Wert Darstellung in die Koeffizientendarstellung umzuwandeln lediglich die Inverse der Punkte mit den Werten multiplizieren müssen.

```

1 // p: Array of coefficients, Wrapper for IFFT for initial call, also renorms the result
2 Function IFFTWrapper(p):
3     result = pointKeyIFFT(p, w); // w starts at n
4     For i = 0 to n-1 do
5         result[i] = result[i] / n; // Result of DFT is normed, the coefficients however are not, therefore
6         // we have to divide the result of IFFT by n
7     return result;
8 // Complexity:  $\Theta(n \log n)$ 
9 Function pointKeyIFFT(p, w):
10    n = length(p);
11    result = new Array(n); resultevenVal = new Array(n); resultoddVal = new Array(n); // Initializes
12    // result and its values to new Array of length n
13    x = new Array(n); x[0] = 1; // initializes x to new Array of length n, holds the x values of the
14    // coefficient
15    For i = 1 to n-1 do
16        x[i] = w * x[i - 1]; // Essentially does x[i] = wi
17    // If the polynomial is constant only needs to finds first coefficient. Base-Case
18    If n == 1 then
19        result[0] = p[0].y;
20    Else
21        resulteven = new Array(n/2); resultodd = new Array(n/2); // Initializes resulteven and resultodd
22        // to new Arrays of length n/2
23        // Fill resulteven with their corresponding values from p
24        For i = 0 to (n-2)/2 do
25            resultevenVal[i] = p[2i]; resultoddVal[i] = p[2i + 1];
26        resulteven = pointKeyIFFT(resultevenVal, w*w); // Evaluates at  $x_j^2$ 
27        resultodd = pointKeyIFFT(resultoddVal, w*w);
28        For i = 0 to n-1 do
29            result[i] = resulteven[i % n] + resultodd[i % n] / x[i];
30    return result;

```

FFT – Algorithmen (I)

Wrapper, der für rekursive Aufrufe die aktuelle primitive Einheitswurzel hinzufügt:

```
FFTWrap(p, n) // n=2^k =#entries in p, k>=0
1  return FFT(p, n, w); //w 2n-th primitive root of unity
```

Zur Vereinfachung bestimmen wir $2n$ (statt $2n-1$) Punkt-Werte, so dass jeweils n Einträge im Array $p[]$ und jeweils doppelt so viele Punkt-Werte berechnet werden

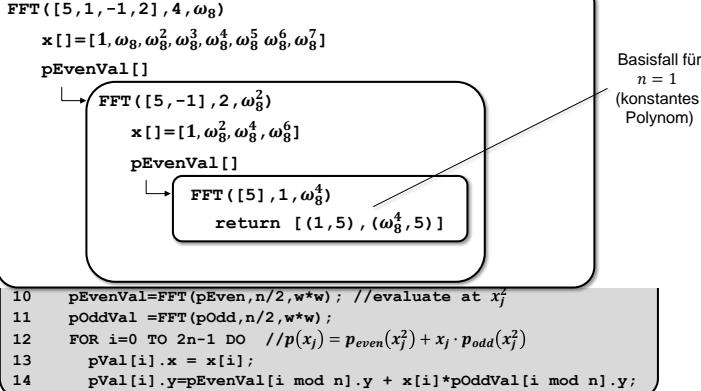
Wir benötigen, dass n Zweierpotenz ist; erreichen wir notfalls, indem wir n maximal verdoppeln und die zusätzlichen Einträge im Array $p[]$ auf 0 setzen

(Übergang $n \rightarrow 2n$ wird später in der asymptotischen Laufzeit nicht ins Gewicht fallen)

FFT – Beispiel (I)

$$p(x) = 5 + x - x^2 + 2x^3$$

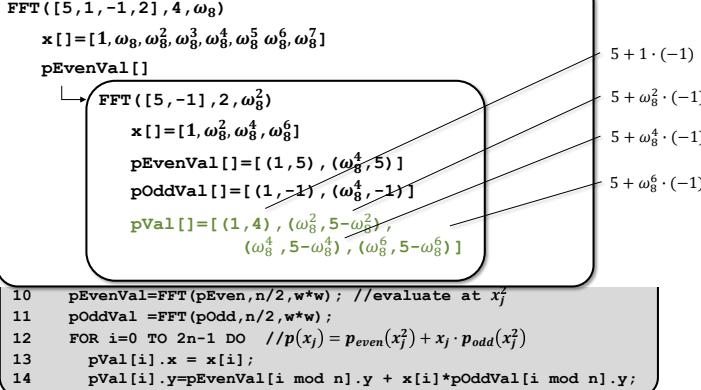
$p=[5, 1, -1, 2]$, $n=4$, $w=\omega_8$



FFT – Beispiel (III)

$$p(x) = 5 + x - x^2 + 2x^3$$

$p=[5, 1, -1, 2]$, $n=4$, $w=\omega_8$



FFT – Algorithmen (II)

```

FFT(p, n, w) // n=2^k =#entries in p, k>=0
1  pEven=ALLOC(n/2); pOdd=ALLOC(n/2);
2  pVal=ALLOC(2n); pEvenVal=ALLOC(n); pOddVal=ALLOC(n);
3  x=ALLOC(2n); x[0]=1; //input values xj
4  FOR i=1 TO 2n-1 DO x[i]=w*x[i-1]; //x[i]=wi

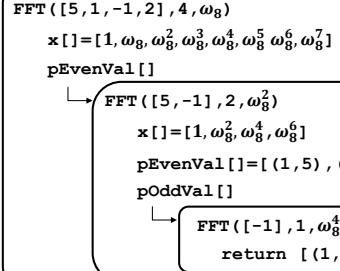
5  IF n==1 THEN //constant polynom
6      pVal[0].x=x[0]; pVal[0].y=p[0].y;
7      pVal[1].x=x[1]; pVal[1].y=p[0].y;
8  ELSE
9      FOR i=0 TO (n-2)/2 DO //create peven, podd
10     pEven[i].x=x[2i]; pOdd[i]=x[2i+1];
11     pEvenVal=FFT(pEven, n/2, w*w); //evaluate at xj2
12     pOddVal=FFT(pOdd, n/2, w*w);
13     FOR i=0 TO 2n-1 DO //p(xj) = peven(xj2) + xj · podd(xj2)
14         pVal[i].x = x[i];
15         pVal[i].y=pEvenVal[i mod n].y + x[i]*pOddVal[i mod n].y;
16  return pVal;

```

FFT – Beispiel (II)

$$p(x) = 5 + x - x^2 + 2x^3$$

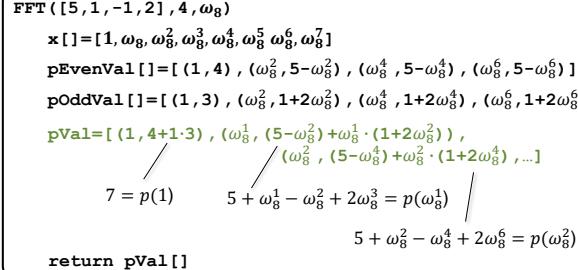
$p=[5, 1, -1, 2]$, $n=4$, $w=\omega_8$



FFT – Beispiel (IV)

$$p(x) = 5 + x - x^2 + 2x^3$$

$p=[5, 1, -1, 2]$, $n=4$, $w=\omega_8$

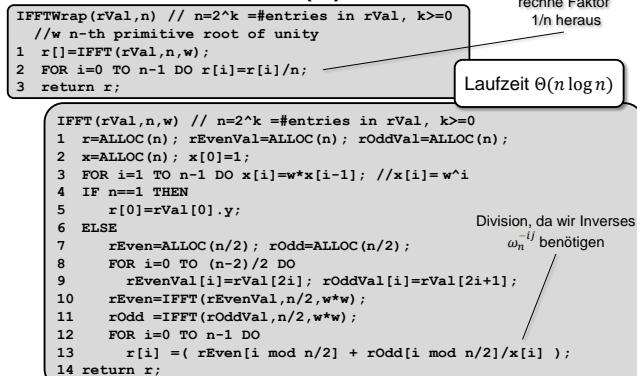


```

10 pEvenVal=FFT(pEven, n/2, w*w); //evaluate at xj2
11 pOddVal=FFT(pOdd, n/2, w*w);
12 FOR i=0 TO 2n-1 DO //p(xj) = peven(xj2) + xj · podd(xj2)
13 pVal[i].x = x[i];
14 pVal[i].y=pEvenVal[i mod n].y + x[i]*pOddVal[i mod n].y;

```

Inverse DFT berechnen (III)



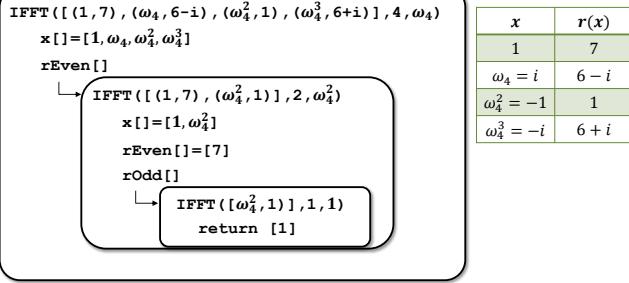
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 26



Inverse FFT: Beispiel (II)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



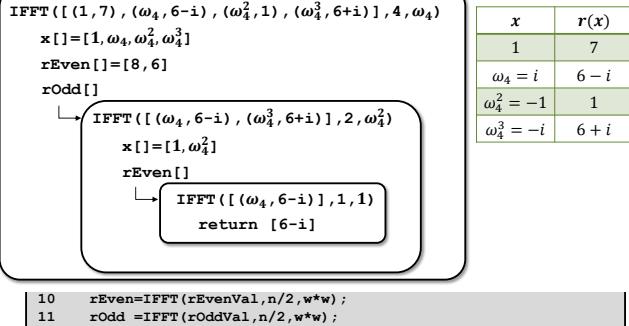
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 28



Inverse FFT: Beispiel (IV)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



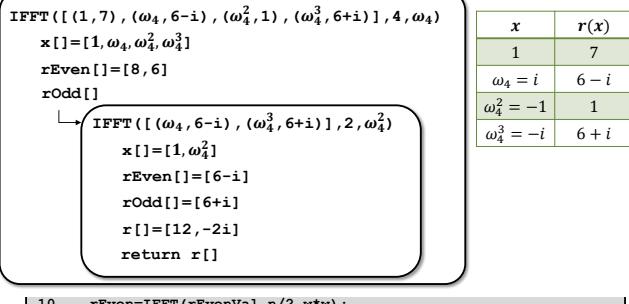
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 30



Inverse FFT: Beispiel (VI)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



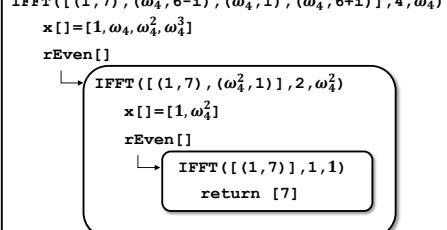
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 32



Inverse FFT: Beispiel (I)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



x	r(x)
1	7
$\omega_4 = i$	$6 - i$
$\omega_4^2 = -1$	1
$\omega_4^3 = -i$	$6 + i$

```

10 rEven=IFFT(rEvenVal,n/2,w*w);
11 rOdd =IFFT(rOddVal,n/2,w*w);
12 FOR i=0 TO n-1 DO
13   r[i] = ( rEven[i mod n/2] + rOdd[i mod n/2]/x[i] );
  
```

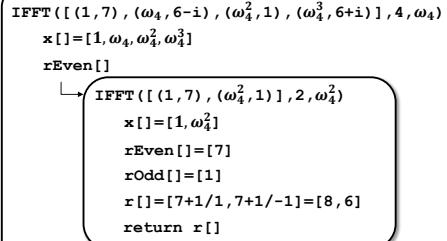
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 27



Inverse FFT: Beispiel (III)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



x	r(x)
1	7
$\omega_4 = i$	$6 - i$
$\omega_4^2 = -1$	1
$\omega_4^3 = -i$	$6 + i$

```

10 rEven=IFFT(rEvenVal,n/2,w*w);
11 rOdd =IFFT(rOddVal,n/2,w*w);
12 FOR i=0 TO n-1 DO
13   r[i] = ( rEven[i mod n/2] + rOdd[i mod n/2]/x[i] );
  
```

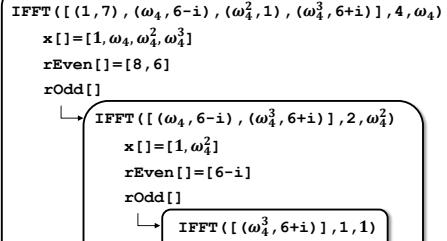
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 29



Inverse FFT: Beispiel (V)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



x	r(x)
1	7
$\omega_4 = i$	$6 - i$
$\omega_4^2 = -1$	1
$\omega_4^3 = -i$	$6 + i$

```

10 rEven=IFFT(rEvenVal,n/2,w*w);
11 rOdd =IFFT(rOddVal,n/2,w*w);
12 FOR i=0 TO n-1 DO
13   r[i] = ( rEven[i mod n/2] + rOdd[i mod n/2]/x[i] );
  
```

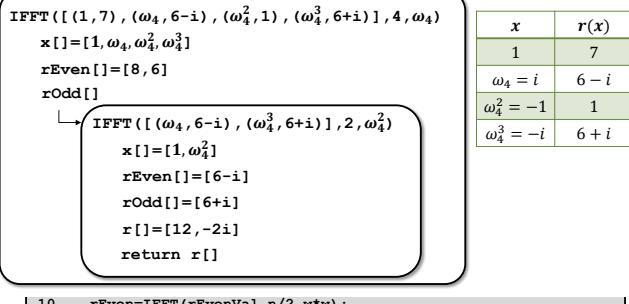
Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 31



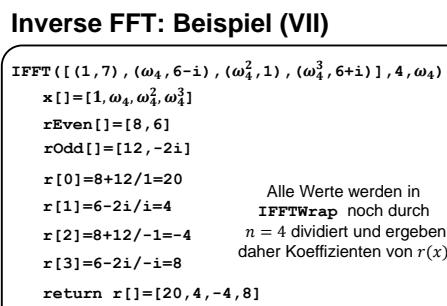
Inverse FFT: Beispiel (VI)

$$r(x) = 5 + x - x^2 + 2x^3$$

$$\omega_4 = i$$



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 32



x	r(x)
1	7
$\omega_4 = i$	$6 - i$
$\omega_4^2 = -1$	1
$\omega_4^3 = -i$	$6 + i$

```

10 rEven=IFFT(rEvenVal,n/2,w*w);
11 rOdd =IFFT(rOddVal,n/2,w*w);
12 FOR i=0 TO n-1 DO
13   r[i] = ( rEven[i mod n/2] + rOdd[i mod n/2]/x[i] );
  
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 33



9.2 Backtracking

Das grundlegende Prinzip vom Backtracking besteht darin für ein Problem zuerst die Lösung $\mathbf{x} = (x_1, x_2, \dots, x_n)$ durch Trial-and-Error zu finden, indem die Teillösung $(x_1, x_2, \dots, x_{i-1})$ von x_i ergänzt wird, bis die Gesamtlösung erreicht wird, oder nicht erreichbar ist, wodurch x_{i-1} aus der Teillösung entfernt wird.

Als Beispiel hierfür kann man zum Beispiel einen Algorithmus zum lösen von Sudoku nehmen. Dieser geht eine mögliche Lösung durch, bis er einen Widerspruch gegen die Regeln endet, an welchem Punkt er einen Schritt zurückgeht und eine andere Lösung sucht.

```
1 // Assumes 9x9 board, but can be easily modified
2 Function SolveSudoku(board):
3     // Base-Case - If board is full the solution is valid
4     If isFull(board) then
5         print board;
6     Else
7         (i,j) = findEmpty(board); // Finds the next empty spot on the board
8         For v = 1 to 9 do
9             If isAdmissible(board, i, j, v) then
10                 board[i][j] = v;
11                 SolveSudoku(board);
12             board[i][j] = 0; // Backtracks if no admissible value is found
```

9.3 Dynamic Programming

Das Prinzip des dynamischen Programmierens ist im Grunde ähnlich zu Divide & Conquer, indem es ein Problem in Teilprobleme aufteilt und diese dann rekursiv löst. Allerdings ist es hier möglich, dass die Teilprobleme überlappen. Dies führt dazu, dass in der Rekursion das selbe Probleme mehrmals behandelt werden kann. Dies wird oft durch Prozesse wie Memoization verbessert, bei der die Lösung des Problems gespeichert wird, sodass es falls es wieder auftritt schneller gelöst werden kann.

Als Beispiel für dieses Konzept kann beispielsweise die Fibonacci-Folge verwendet werden.

9.3 (a) Fibonacci

```
1 // Complexity: Θ(2n)
2 // Assumes n >= 1
3 Function FibonacciRecursive(n):
4   If n <= 2 then
5     return 1;
6   Else
7     return FibonacciRecursive(n-1) + FibonacciRecursive(n-2);
```

Hier wird noch nicht Memoization verwendet, weshalb dieser Algorithmus relativ langsam ist.

```
1 // Assumes n >= 1
2 Function FibonacciMemo(n):
3   F = new Array(n); // Creates new array that stores the results of the function for each n at i -
4   For i = 0 to n - 1 do
5     F[i] = 0;
6   return FibRecMemo(n - 1, F);
7 // Complexity: Θ(n)
8 // Assumes i >= 0
9 Function FibRecMemo(i, F):
10  // If i already has a result in F, return it
11  If F[i] != 0 then
12    return F[i];
13  // If i is 0 or 1, return 1 - Base-Case
14  If i <= 1 then
15    f = 1;
16  Else
17    f = FibRecMemo(i - 1, F) + FibRecMemo(i - 2, F); // Recursively call FibRecMemo for i - 1 and
18    i - 2
19    F[i] = f; // Add result to F
20  return f;
```

Durch die Nutzung von Memoization hat sich die Komplexität deutlich verbessert von $\Theta(2^n)$ auf $\Theta(n)$.

9.3 (b) Minimum Edit Distance

Ein anderes Beispiel ist der Minimum-Edit-Distance Algorithmus. Dieser misst die Ähnlichkeit von Texten auf, wie viele Operationen nötig sind um Texte ineinander überzuführen. Die Operationen sind dabei:

- **Insertion:** `ins(S, i, b)` fügt an Position i den Buchstaben b in String s ein
- **Deletion:** `del(S, i)` entfernt das Zeichen an Position i in String s
- **Substitution:** `sub(S, i, b)` ersetzt das Zeichen an Position i in String s durch den Buchstaben b

Der Algorithmus funktioniert hierbei so, dass er den String $X[1\dots m]$ von links nach rechts schrittweise in String $Y[1\dots n]$ umwandelt. Demnach ist $X[1\dots i]$ bereits in $Y[1\dots j]$ umgewandelt.

```
1 // Complexity: Θ(m · n)
2 // X[1..m], Y[1..n]: Strings, m, n: length of strings
3 Function MinEditDistance(X, Y, m, n):
4     D = new 2DArray(m,n); // D[i][j]: minimal amount of operations to transform the first i chars of
      // X to the first j chars of Y
5     For i = 0 to m do
6         D[i][0] = i; // First Row represents case that Y is empty: To transform the first i chars of X
                      // to an empty string, we need to delete them with i operations
7     For j = 0 to n do
8         D[0][j] = j;
9         // First Column represents case that X is empty: To transform an empty string to the first j
          // chars of Y, we need to insert them with j operations
10    For i = 1 to m do
11        For j = 1 to n do
12            // If the chars are equal, no operations are needed
13            If X[i] == Y[j] then
14                s = 0;
15            Else
16                s = 1;
17            // There are now 3 possible operations: 1. Insertion, 2. Deletion, 3. Substitution. We
              // now choose the option with the lowest cost and assign it
18            // For Substitution, the cost is dependant on D[i - 1][j - 1] plus the cost s of the
              // operation
19            // For Deletion, the first i-1 chars of X need to be transformed to the first j chars of
              // Y. Thus the cost is D[i - 1][j] + 1
20            // For Insertion, the first i chars of X need to be transformed to the first j-1 chars of
              // Y. Thus the cost is D[i][j - 1] + 1
21            D[i][j] = min(D[i - 1][j - 1] + s, D[i - 1][j] + 1, D[i][j - 1] + 1);
22    return D[m][n];
```

Minimum Edit Distance: Algorithmus

mittels dynamischer Programmierung und Memoization

```
MinEditDist(X,Y,m,n) // X=X[1..m], Y=Y[1..n]
1 D[] []=ALLOC(m,n);
2 FOR i=0 TO m DO D[i][0]=i;
3 FOR j=0 TO n DO D[0][j]=j;
4 FOR i=1 TO m DO
5   FOR j=1 TO n DO
6     IF X[i]=Y[j] THEN s=0 ELSE s=1;
7     D[i][j]=min(D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1);
8 return D[m][n];
```

Laufzeit und Speicher $\Theta(mn)$

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 61

Minimum Edit Distance: Beispiel (I)

Minimum Edit Distance: Beispiel (I)

```
X[1.. 8] = ein Test
Y[1..10] = zwei Feste
```

Initialisierung (2 und 3)

```
MinEditDist(X,Y,m,n) // X=X[1...
1 D[] []=ALLOC(m,n);
2 FOR i=0 TO m DO D[i][0]=i;
3 FOR j=0 TO n DO D[0][j]=j;
4 FOR i=1 TO m DO
5   FOR j=1 TO n DO
6     IF X[i]=Y[j] THEN s=0 ELSE s=1;
7     D[i][j]=min(D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1);
8 return D[m][n];
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 62

x[1..i]									
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1								
2	2								
3	3								
4	4								
5	5								
6	6								
7	7								
8	8								
9	9								
10	10								

x[1..i]

Minimum Edit Distance: Beispiel (II)

X[1.. 8] = ein Test

Y[1..10] = zwei Feste

i=1: FOR-Schleife für j (5-7)

j=1: X[1]=e ≠ z=Y[1] ⇒ s=1

D[1][1]=min(0+1,1+1,1+1)=1

```
MinEditDist(X,Y,m,n) // X=X[1...
1 D[] []=ALLOC(m,n);
2 FOR i=0 TO m DO D[i][0]=i;
3 FOR j=0 TO n DO D[0][j]=j;
4 FOR i=1 TO m DO
5   FOR j=1 TO n DO
6     IF X[i]=Y[j] THEN s=0 ELSE s=1;
7     D[i][j]=min(D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1);
8 return D[m][n];
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 63

x[1..i]									
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1							
2	2								
3	3								
4	4								
5	5								
6	6								
7	7								
8	8								
9	9								
10	10								

x[1..i]

Minimum Edit Distance: Beispiel (IV)

X[1.. 8] = ein Test

Y[1..10] = zwei Feste

i=1: FOR-Schleife für j (5-7)

j=3: X[1]=e = e=Y[3] ⇒ s=0

D[1][3]=min(2+0,3+1,2+1)=2

```
MinEditDist(X,Y,m,n) // X=X[1...
1 D[] []=ALLOC(m,n);
2 FOR i=0 TO m DO D[i][0]=i;
3 FOR j=0 TO n DO D[0][j]=j;
4 FOR i=1 TO m DO
5   FOR j=1 TO n DO
6     IF X[i]=Y[j] THEN s=0 ELSE s=1;
7     D[i][j]=min(D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1);
8 return D[m][n];
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 65

x[1..i]									
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1							
2	2	2							
3	3	3	2						
4	4								
5	5								
6	6								
7	7								
8	8								
9	9								
10	10								

x[1..i]

Minimum Edit Distance: Beispiel (VI)

X[1.. 8] = ein Test

Y[1..10] = zwei Feste

```
MinEditDist(X,Y,m,n) // X=X[1...
1 D[] []=ALLOC(m,n);
2 FOR i=0 TO m DO D[i][0]=i;
3 FOR j=0 TO n DO D[0][j]=j;
4 FOR i=1 TO m DO
5   FOR j=1 TO n DO
6     IF X[i]=Y[j] THEN s=0 ELSE s=1;
7     D[i][j]=min(D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1);
8 return D[m][n];
```

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 67

x[1..i]									
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1	2	3	4	5	6	7	8
2	2	2	2	3	4	5	6	7	8
3	3	2	3	3	4	5	5	6	7
4	4	3	2	3	4	5	6	6	7
5	5	4	3	3	3	4	5	5	6
6	6	5	4	4	4	5	6	7	8
7	7	6	5	5	5	5	4	5	6
8	8	7	6	6	6	5	4	5	6
9	9	8	7	7	7	6	5	4	5
10	10	9	8	8	8	8	7	6	5

x[1..i]

Minimum Edit Distance: Beispiel (VII)

Minimum Edit Distance: Beispiel (VII)

Rückwärtsschrittfolge
D[m][n] zu D[0][0] entlang
der ausgewählten Minima
(bzw. bis Rand erreicht)
gibt Operationen an:

(\v) : copy (+0)
\v : sub (+1)
→ : del
↓ : ins

(Im Allgemeinen gibt es
mehrere mögliche Sequenzen!)

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 68

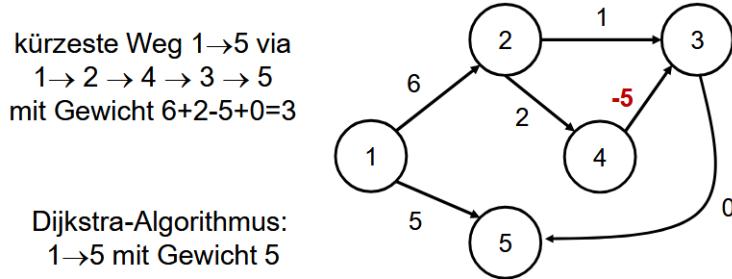
x[1..i]									
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1	2	3	4	5	6	7	8
2	2	2	2	3	4	5	6	7	8
3	3	2	3	3	4	5	5	6	7
4	4	3	2	3	4	5	5	6	7
5	5	4	3	3	3	4	5	6	7
6	6	5	4	4	4	5	6	7	8
7	7	6	5	5	5	4	5	6	7
8	8	7	6	6	6	5	4	5	6
9	9	8	7	7	7	6	5	4	5
10	10	9	8	8	8	8	7	6	5

x[1..i]

9.4 Greedy Algorithms

Das Prinzip des Greedy Algorithmus geht danach zuerst eine Lösung $x = (x_1, x_2, \dots, x_n)$ zu finden indem man der Teillösung $(x_1, x_2, \dots, x_{i-1})$ mit x_i ergänzt. Hierbei ist x_i der Teil, der Lokal am günstigsten erscheint.

Hierzu gehören beispielsweise Dijkstras und Prims Algorithmus, die für jede Iteration immer den Knoten auswählen, der in dieser Iteration die kürzeste Distanz hat, oder Kruskals Algorithmus, der in jeder Iteration immer die leichteste Kante in dieser Iteration auswählt. Das Problem bei diesen Algorithmen ist aber, dass sie nicht immer die optimale Lösung finden, da sie lokal über global nehmen.



9.4 (a) Traveling Salesperson Problem (TSP)

Im Folgenden wird nun ein Algorithmus zum Lösen des Traveling Salesperson Problems behandelt.

Das Traveling Salesperson Problem ist wie folgt definiert:

Gegeben sei ein vollständiger (Jeder Knoten hat Kante zu jedem anderen Knoten) ungerichteter Graph $G = (V, E)$ mit Kantengewichten w . Auf diesem Graphen soll nun einen Pfad mit minimalen Kantengewicht finden, der bis auf Start- und Endknoten ($v_0 = v_{i+1}$, Start = Ende) jeden Knoten genau einmal besucht, dieser Pfad wird **Tour** genannt.

Der Ansatz für den Greedy Algorithmus hier ist es für den momentanen Knoten immer die leichteste Kante zu einem noch nicht besuchten Knoten zu nehmen. Dies wird dann solange wiederholt bis alle Knoten besucht sind, woraufhin dann zum Startknoten zurückgekehrt wird.

```

1 // G: Graph with (V,E), s: startingnode, w: weight function
2 Function greedyTSP(G,s,w):
3   ForEach v in V do
4     v.color = WHITE; // Initialize all nodes to WHITE = not visited
5   tour = new Array(n); tour[0] = s; tour[0].color = GREY; // Initialize tour array, add start and set
      start to visited
6   For i = 1 to n - 1 do
7     tour[i] = extractMin(adj(tour[i-1])); // Choose next node by smallest edge
8     tour[i].color = GREY; // Set node to visited
9   return tour;
  
```

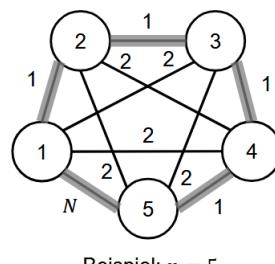
Dieser Algorithmus ist leider nicht ideal. Zwar nimmt er jedenfalls immer den kleinsten Weg, jedoch ist die Rückkehr zum Startknoten somit fest. Wenn die Kante zwischen dem letzten Knoten und dem Startknoten n groß ist mit $n \geq \sum_{(u,v) \in E} w(u, v)$ ist dieser Pfad immer suboptimal.

Greedy-Algorithmus läuft zunächst

$1 \rightarrow 2 \rightarrow \dots \rightarrow n-1 \rightarrow n$,

muss dann die „teure“ Kante $n \rightarrow 1$ nehmen, erreicht also nur Gewicht

$$1 + 1 + \dots + 1 + N = N + n - 1$$



Beispiel: $n = 5$

9.5 Metaheuristiken

In der Informatik unterscheidet man oft zwischen Heuristischen und Metaheuristischen Algorithmen. Hierbei sind heuristische Algorithmen spezifische Algorithmen, die eine gute, aber eventuell nicht optimale Lösung für ein spezielles Problem findet.

Die Metaheuristiken sind Algorithmen, die eher eine allgemeine Vorgehensweise anhand von abstrakten Problemen angeben um die Suche für beliebige Optimierungen zu leiten.

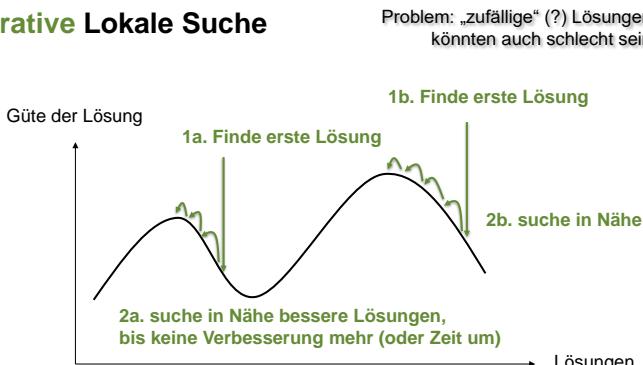
So kann für das TSP der Algorithmus erweitert werden, indem er nach lokal optimalen Minima/ Maxima sucht.

```
1 // P: Problem
2 Function HillClimbing(P, maxTime):
3     sol = initialSolution(P); // Searches for any solution (e.g. greedyTSP)
4     time = 0;
5     While time < maxTime do
6         newSol = SwitchRandom(sol); // Switches random nodes
7         If newSol.cost < sol.cost then
8             sol = newSol; // Updates solution if better
9         time++;
10    return sol;
```

Dieser Algorithmus gibt jetzt schon aus einer Auswahl an Lösungen die beste zurück, jedoch ist sie immer noch nicht ideal, da sie nur nach lokalen Minima/Maxima sucht und womöglich nicht die global beste Lösung findet. Ein Ansatz dies zu verbessern ist, den Algorithmus mehrmals laufen zu lassen, sodass er an unterschiedlichen Punkten anfängt und so optimalerweise verschiedene Maxima/Minima findet. Aber auch dies ist nicht garantiert.

In jedem Fall ist aber die Suche nach lokalen Minima/Maxima ein Metaheuristischer Algorithmus, der es erlaubt, zu optimieren.

Iterative Lokale Suche



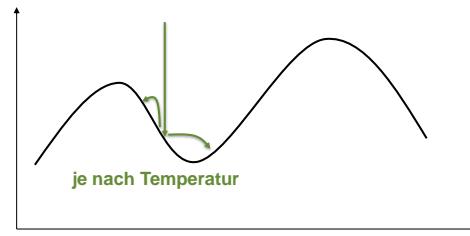
Beginne Suche nochmal von vorne, z.B. mit neuer zufälliger Lösung, akzeptiere beste gefundene Lösung

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 91

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptoplexy

Simulated Annealing

„Annealing“ in Metallverarbeitung:
Härten von Metallen durch Erhitzen auf hohe Temperatur und langsames Abkühlen



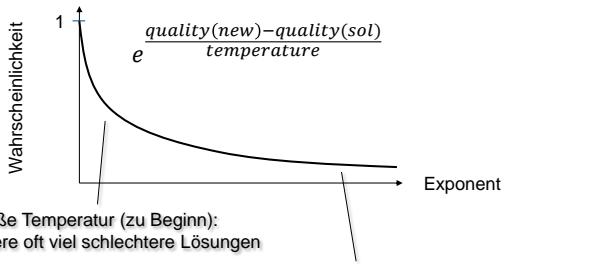
1. Temperatur zu Beginn hoch, kühl langsam ab
2. Je höher Temperatur, desto wahrscheinlicher „Sprung in schlechte Richtung“

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 92

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptoplexy

„In schlechte Richtung“ mit Wahrscheinlichkeit

Ansatz: akzeptiere auch schlechtere Lösung new mit $quality(new) < quality(sol)$ mit Wahrscheinlichkeit:



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 93

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptoplexy

Simulated Annealing

```
SimulatedAnnealing(P) // maxTime constant
// TempSched[] temparature annealing

1 sol=initialSol(P);
2 time=0;
3 WHILE time<maxTime DO
4   new=perturb(P,sol);
5   temperature=TempSched[time];
6   d=quality(P,new)-quality(P,sol); r=random(0,1);
7   IF d>0 OR r=<exp(d/temperature) THEN
8     sol=new;
9   time=time+1;
10 return sol;
```

uniformer Wert zwischen 0 und 1

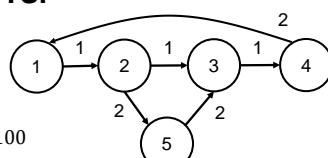
Bestimmung eines guten „Annealing schedule“ (Starttemperatur und Abnahme) betrachten wir hier in der Vorlesung nicht weiter

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 07 Fortgeschritten Algorithmen-Entwurfsmethoden | 94

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptoplexy

Simulated Annealing beim TSP

$$\text{Starttemperatur} = \max_{e \in E} \{w(e)\} = N = 1000$$



$$\text{lineare Temperaturabnahme} = N/2n = 100$$

	Aktuelle Lösung	Gewicht	Zuf. Tausch	Gew. nach Tausch	Temp	Zuf. Wert r	Exp (-d/temp)	Neue Lösung akzeptiert?
1	1→2→3→4→5→1	2N+3	2 und 3	5N	1000	0.37	0.0498	nein
2	1→2→3→4→5→1	2N+3	2 und 4	4N+2	900	0.60	0.1084	nein
3	1→2→3→4→5→1	2N+3	4 und 5	2N+4	800	0.17	0.9986	ja (Zufall)
4	1→2→3→5→4→1	2N+4	2 und 4	5N	700	0.34	0.0138	nein
5	1→2→3→5→4→1	2N+4	3 und 4	3N+3	600	0.45	0.1889	nein
6	1→2→3→5→4→1	2N+4	3 und 5	8	500	0.78	-	ja (besser)
7	1→2→5→3→4→1	8	2 und 5	2N+4	400	0.51	0.0067	nein

10 NP

10.1 Leichte Probleme

Ein Problem wird als *leicht* bezeichnet, wenn es in Polynomialzeit lösbar ist. So gilt für die Laufzeit eines leichten Algorithmus $\Theta(\sum_{i=0}^k a_i n^i) = poly(n)$, wobei a_i und k Konstanten sind. Beispiele dafür wären beispielsweise Sortieren eines Arrays (z.B. Insertion Sort $\Theta(n^2)$ oder MergeSort $\Theta(n \log n)$), Breadth-First-Search im Graphen, Minimale Spannbäume berechnen etc. Dies sind leicht zu lösende Algorithmen, die sie immer eine korrekte Lösung in Polynomialzeit ergeben.

Anders gibt es noch Algorithmen, bei denen die Lösung leicht zu prüfen ist, die Lösung selbst aber nicht unbedingt optimal, und in Polynomialzeit laufen. Dazu gehört auch Beispielsweise das Traveling Salesperson Problem.

10.2 Berechnung- & Entscheidungsprobleme

Im Allgemeinen kann man Probleme in zwei Kategorien aufteilen.

Berechnungsprobleme:

Problem, bei dem der Algorithmus eine Ausgabe berechnet. Dazu gehören zum Beispiel Algorithmen, bei denen ein Pfad in einem Graph ausgegeben wird.

Entscheidungsprobleme:

Problem, bei dem der Algorithmus eine Entscheidung berechnet. Sie geben also lediglich zurück, ob die das Problem die spezifischen Eigenschaft besitzt oder nicht. Beispielsweise ist der Algorithmus zum überprüfen ob ein gerichteter Graph stark zusammenhängend ist ein Entscheidungsproblem.

Man kann grundsätzlich jedes Berechnungsproblem in Entscheidungsproblem überführen. Wenn B in Polynomialzeit möglich ist, so ist auch E in Polynomialzeit möglich.

Als Beispiel hierfür kann der folgende Algorithmus betrachtet werden, der alle Primfaktoren einer Zahl berechnet. Ein Primfaktor ist ein Faktor einer Zahl, der eine Primzahl ist. Also ist ein Primfaktor einer Zahl ein Teiler dieser Zahl, der eine Primzahl ist.

Z.B. $12 = 2 \cdot 6 = 2 \cdot 2 \cdot 3$, 2 und 3 sind Primzahlen, dementsprechend Primfaktoren für 12.

```
1 // n > 1
2 // Calculates all prime factors of n and prints them out
3 Function factorize(n):
4   While n > 1 do
5     p = computeFactor(n); // Computes the prime factor
6     print p;
7     n = n / p;
8 Function computeFactor(n):
9   L = 1; U = n; // L and U are the bounds for the search
10  While L != U do
11    M = L + floor((U - L) / 2); // M is the middle of the search
12    // If decideFactor == 1, then m is a divisor of n (or bigger than the smallest primefactor of
13    // n). Lower upper bound U to M
13    If decideFactor(N,M) == 1 then
14      U = M;
15    // If decideFactor == 0, then m is not a divisor of n. Increase lower bound L to M + 1
16    Else
17      L = M + 1;
18 Function decideFactor(n,m):
19   // Decide whether m is a primefactor of n
20   return d; // d = 1 if m is a primefactor of n, d = 0 otherwise
```

Dieser Algorithmus stellt jetzt ein Berechnungsproblem dar, da er alle Primfaktoren einer Zahl berechnet. Die Lösung des Berechnungsproblems kann jetzt genutzt werden um das Entscheidungsproblem "Ist der kleinste Primfaktor von n maximal x?" zu lösen, indem man schaut ob der kleinste Primfaktor von n kleiner oder gleich x ist.

10.3 Komplexitätsklasse P

Ein Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse P, wenn es einen Algorithmus in Polynomialzeit A_{L_E} mit Ausgabe 0 / 1 gibt, der stets korrekt entscheidet ob P die Eigenschaft E erfüllt. Es gilt also: $\forall P : P \in L_E \Leftrightarrow A_{L_E}(P) = 1$.

Anders: Alle Entscheidungsprobleme, dessen Lösung in Zeit ausgedrückt werden kann, die ein Polynom in der Größe des Eingabewerts ist.

10.4 Komplexitätsklasse NP

Ein Entscheidungsproblem L_E ist genau dann in der Komplexitätsklasse NP, wenn es einen Algorithmus in Polynomialzeit A_{L_E} mit Ausgabe 0 / 1 gibt, der bei Eingabe eines Zeugen S_P für jede Eingabe $P \in L_E$ bzw. für jede Eingabe S_P für Eingabe $P \notin L_E$ stets korrekt entscheidet ob P die Eigenschaft E erfüllt. Es gilt also: $\forall P : P \in \exists S_P : A_{L_E}(P, S_P) = 1$ oder $\forall P : P \notin L_E \Leftrightarrow \forall S_P : A_{L_E}(P, S_P) = 0$ (Äquivalent). Hierbei steht NP für Nichtdeterministische Polynomialzeit. Anders: Ein Problem liegt in NP, wenn eine gegebene Lösung in polynomieller Zeit verifiziert werden kann, auch wenn man nicht sicher ist, ob die Lösung effizient in Polynomialzeit gefunden werden kann. (Bsp. Traveling Salesperson Problem: Kein Algorithmus bekannt, der in Polynomialzeit für alle Fälle optimale Lösung findet, jedoch kann für jede Lösung in polynomieller Zeit überprüfen ob diese korrekt ist.)

Im Allgemeinen also:

- P sind Probleme die effizient gelöst werden können
- NP sind Probleme, bei denen eine Lösung effizient verifiziert werden kann, aber unklar ist, ob sie auch effizient gefunden werden kann.

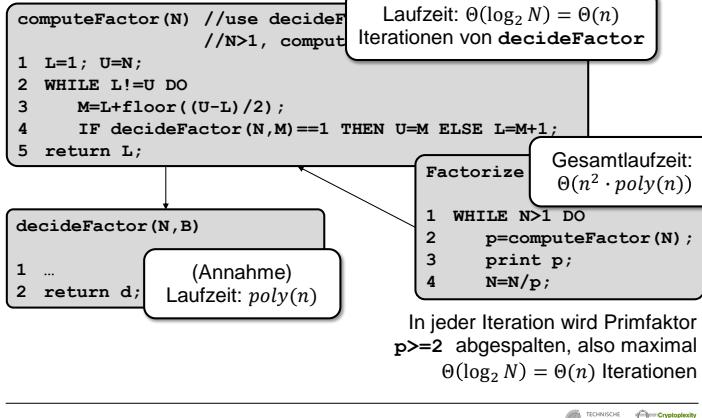
10.5 NP-Vollständigkeit

Ein Problem ist NP-Vollständig, wenn es in NP liegt und NP-schwer ist. NP-Schwer bedeutet, dass ein Problem mindestens so schwer ist wie jedes Problem in NP. Also kann jedes Problem in NP auf das NP-Schwere Problem reduziert werden. (Ein NP-schweres Problem muss nicht unbedingt in NP liegen.)

Ein NP-Vollständiges Problem, dass sich in Polynomialzeit lösen lässt impliziert, dass alle Probleme in NP in Polynomialzeit lösbar sind.

Hierbei spielt die Reduktion eine große Rolle. Nimmt man zum Beispiel einen Graphen und ein Problem P "Gibt es einen Pfad, der vom Start aus jeden Knoten besucht und zurück zum Start führt?" und Problem Q "Gibt es einen Pfad, der vom Start aus jeden Knoten besucht und zum Ziel zurück zum Start zurück mit Gewicht maximal x?". So ist das Problem P auch im Problem Q enthalten und man kann P reduzieren.

Beispiel: Faktorisieren (V)



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 08 NP | 11

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

SAT: Die Mutter aller NP-vollständigen Probleme

SAT

Gegeben: Boolesche Formel ϕ aus \wedge, \vee, \neg in n Variablen x_1, x_2, \dots, x_n
(ϕ polynomiale Komplexität in n)

Gesucht: Entscheide, ob ϕ erfüllende Belegung hat oder nicht

Beispiel: $\phi(x_1, x_2, x_3, x_4) = [\neg x_2 \vee (x_3 \wedge \neg x_4)] \wedge x_2 \wedge \neg[(x_1 \vee \neg x_2) \wedge x_4]$

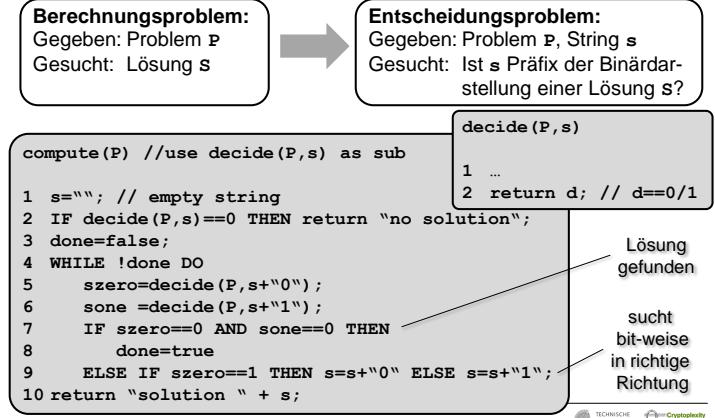
hat erfüllende Belegung
 $x_1 \leftarrow \text{false}, x_2 \leftarrow \text{true}, x_3 \leftarrow \text{true}, x_4 \leftarrow \text{false}$

Offensichtlich: **SAT** ∈ **NP** (gegebene Belegung als Zeuge, werte Formel aus)

Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 08 NP | 33

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

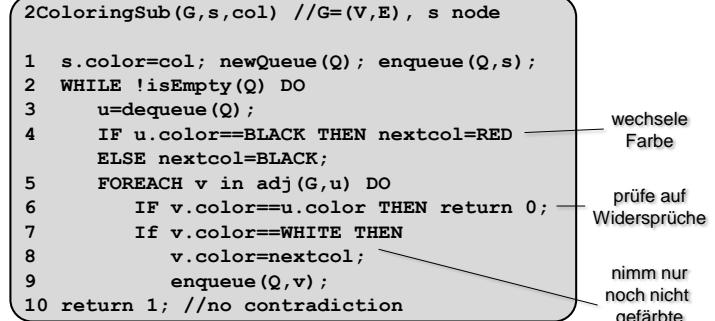
Berechnung durch Entscheidung (I)



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 08 NP | 12

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology

2-Färbbarkeit von Graphen ist (relativ) einfach (II)



Algorithmen und Datenstrukturen | Marc Fischlin | SS 24 | 08 NP | 52

TECHNISCHE UNIVERSITÄT DARMSTADT Cryptology