

AUD Reference Sheet

Frederick Wichert

Last Edited: 22. August 2020

Inhaltsverzeichnis

1 Begriffsdefinitionen	1
1.1 Allgemein	1
2 Algorithmen	2
2.1 Definition	2
2.2 Anforderungen	2
2.3 Eigenschaften	2
2.4 Effizienz von Algorithmen	2
2.5 Divide-and-Conquer	2
3 Asymptotische Komplexität	3
3.1 Allgemein	3
3.2 Asymptotische Notation	3
3.3 Schranken der asymptotischen Komplexität	3
3.4 Notationen und Definition	4
4 Analyse von Divide-and-Conquer Algorithmen	5
4.1 Allgemein	5
4.2 Methoden zur Lösung von Rekursionsgleichung	5
4.3 Mastertheorem	6
5 Sortieralgorithmen	7
5.1 Sortieralgorithmen: Theorie	7
5.2 Entwurfsprinzipien des Sortieren	7
5.3 InsertionSort	8
5.4 BubbleSort	8
5.5 SelectionSort	9
5.6 MergeSort	9
5.7 QuickSort	11
6 Datenstrukturen	13
6.1 Datenstrukturen: Theorie	13
6.2 Stacks	13
6.3 Verkettete Listen	14
6.4 Queues	14
6.5 Binärbäume - Bäume	15
6.6 Binäre-Suchbäume	16
6.7 AVL Bäume	18
6.8 Splay-Bäume	20
6.9 Binäre Max-Heaps	22
6.10 B-Bäume	24
6.11 Skip-Lists	27
6.12 Hash-Tables	29
6.13 Bloom-Filter	31

7	Rot-Schwarz-Bäume	32
7.1	Allgemeines	32
7.2	Einfügen	32
7.3	Löschen	34
7.4	Löschen: fixColorAfterDeletion	35
7.5	Löschen: Beispielgraph	37
8	Graphen	38
8.1	Allgemein	38
9	Exkurs	39
9.1	Totale Ordnung	39

1 Begriffsdefinitionen

1.1 Allgemein

Problem	Ein Problem im Sinne der Informatik enthält eine Beschreibung der Eingabe, der Ausgabe, aber keinen Übergang. Eine Probleminstanz eine bestimmte Belegung der Eingabeveriablen
Algorithmus	Ein Algorithmus ist eine endliche Folge von Rechenschritten, die eine Eingabe in eine Ausgabe umwandelt.
Datenstruktur	Eine Datenstruktur ist eine Methode, Daten abzuspeichern und zu organisieren, sowie deren Zugriff auf die Daten und die Modifikation der Daten zu erleichtern.

2 Algorithmen

2.1 Definition

Ein Algorithmus ist eine endliche Folge von Rechenschritten, die eine **Eingabe** in eine **Ausgabe** umwandelt.

2.2 Anforderungen

- Spezifizierung der Eingabe und Ausgabe
- Eindeutigkeit - Jeder Einzelschritt ist klar definiert mit festgelgter Reihenfolge
- Endlichkeit - Die Notation hat eine Endliche Länge

2.3 Eigenschaften

- Determiniertheit - Für gleiche Eingabe wird gleiche Ausgabe berechnet (mögliche, andere Zwischenzustände)
- Determinismus - Für die gleiche Eingabe ist die Ausführung und Ausgabe identisch
- Terminierung - Der Algorithmus läuft für jede Eingabe nur endlich lange
- Korrektheit - Der Algorithmus berechnet stets die spezifizierte Ausgabe, falls dieser terminiert
- Effizienz - Sparsamkeit bei Ressourcen (Zeit, Speicher, Energie etc.)

2.4 Effizienz von Algorithmen

- Effizienzfaktoren
 - Rechenzeit (Anzahl der Einzelschritte)
 - Kommunikationsaufwand (z.B. Netzwerk)
 - Speicherplatzbedarf
 - Zugriff auf Speicher (z.B. Festplatte)
- (Konkret) Laufzeit hängt von vielen Faktoren ab
 - Länge der Eingabe
 - Implementierung der Basisoperationen
 - Takt der CPU

2.5 Divide-and-Conquer

- **Prinzip:**
Zerlege das Problem und löse es direkt *oder* durch weitere Zerlegung
- **Divide:**
Teile das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des gleichen Problems darstellen
- **Conquer:**
Beherrsche die Teilprobleme rekursiv. Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme direkt
- **Combine:**
Vereinige die Lösungen der Teilprobleme zur Lösung des ursprünglichen Problems

3 Asymptotische Komplexität

3.1 Allgemein

Eine Abschätzung des (zeitlichen) Aufwands eines Algorithmus in Abhängigkeit der Eingabe in form einer Asymptote in Abhängigkeit der Eingabemenge n .

- Beispiel: Summe bis n
 - `summe=summe+i`
 - Ablauf:
 - * Variablen deklarieren
 - * 1 zur Summe addieren
 - * 2 zur Summer addieren
 - * ...
 - * n zur Sumer addieren
 - * Ausgabe der Summe
 - Alle Basisoperationen benötigen konstant viel Aufwand
 - Anzahle der Operationen wächst mit Eingabe n
 - Für liner wachsende Eingabe wächst der Aufwand auch linear
- Beispiel: Laufzeit von InsertionSort
 - Quadratische Funktion: $an^2 + bn + c$
 - Bei großem n überweigt der quadratische Anteil → Terme niedriger Ordnung werden irrelevant
 - s

3.2 Asymptotische Notation

- Wie ist das Verhalten der Laufzeit $T(n)$ für sehr große Eingaben $\in N$
 - bei kleinen Eingaben (z.B. $n = 5$) ist die reale Laufzeit mehr dominiert von den Initialisierungskosten und nicht dem Algorithmus selbst
 - Komplexität ist unabhängig von konstanten Faktoren und Summanden
 - Berücksichtigen nicht Betrachtung:
 - * Rechnergeschwindigkeit
 - * Aufwände der Initialisierung
 - Komplexitätsmessungen via Funktionsklassen ausreichend
 - * Wie verhält sich der Algorithmus für große Problemgrößen
 - * Wie verändert sich die Laufzeit, wenn die Problemgröße verdoppelt wird

Komplexitätsfunktion

Das Wachstumsverhalten reicht aus um die wichtigen Punkte eines Algorithmus zu veranschaulichen.

3.3 Schranken der asymptotischen Komplexität

Best case

- Komplexität im "bestenFall"
- Wie schnell ist der Algorithmus, wenn der Input so vorliegt, dass dieser am schnellsten terminiert?
- Beispiel: Vorsortierte Liste für (simpelen) Sortieralgorithmus

Average case

- Komplexität im "durchschnittlichenFall"
- Wie schnell ist der Algorithmus, wenn ein üblicher Input vorliegt?
- Beispiel: Zufällig verteilte Liste für (simpelen) Sortieralgorithmus

Worst case

- Komplexität im "schlechtestenFall"
- Wie schnell ist der Algorithmus, wenn der Input so vorliegt, dass dieser am spätestens terminiert?
- Beispiel: Negativ-sortierte Liste für (simpelen) Sortieralgorithmus

3.4 Notationen und Definition

Θ -Notation

Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_{>0}$

Dabei ist \mathbb{N} Die Eingabelänge und \mathbb{R} die Zeit

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

Schreibweise: $f \in \Theta(g)$ oder $f = \Theta(g)$

O-Notation - Laufzeit

Obere asymptotische Schranke - auch Laufzeit eines Algorithmus genannt

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

Sprechweise: f wächst höchstens so schnell wie g

Sprechweise: $f = O(G)$

Beachte: $f(n) = \Theta(n) \implies f(n) = O(n)$

$$\Theta(g(n)) \subseteq O(g(n))$$

Rechenregeln

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

Ω -Notation

Untere asymptotische Schranke

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

Sprechweise: f wächst mindestens so schnell wie g

Sprechweise: $f = \Omega(G)$

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

θ und ω

TODO

4 Analyse von Divide-and-Conquer Algorithmen

4.1 Allgemein

$T(n)$ ist Laufzeit eines Problems der Größe n .

Für kleines Problem, d.h. $n \leq c$ mit c ist eine Konstante, dann benötigt die Direkte Lösung konstante Zeit $\Theta(1)$.

Für sonstige n gilt: Das Aufteilen des Problems führt zu a Teilproblemen, von denen jedes die Größe $1/b$ der Größe des ursprünglichen Problems hat.

Das Lösen eines Teilproblems der Größe n/b dauert $T(n/b)$ und somit benötigen wir für das Lösen von a solcher Probleme $aT(n/b)$.

$D(n)$ ist die Zeit um das Problem aufzuteilen, $C(n)$ ist die Zeit um die Teillösungen zur Lösung des ursprünglichen Problems zusammenzufügen.

$$T(n) = \begin{cases} \Theta(1) & \text{Falls } n \leq c \\ aT(n/b) + D(n) + C(n) & \end{cases}$$

4.2 Methoden zur Lösung von Rekursionsgleichung

Substitutionsmethode: wir erraten eine Schranke und nutzen mathematische Induktion, um die Korrektheit zu beweisen.

Rekursionsbaum-Methode: wandelt die Rekursionsgleichung in einen Baum um, dessen Knoten die Kosten der Rekursion in den verschiedenen Ebenen darstellt.

Mastermethode: liefert Schranken für Rekursionsgleichung der Form: $T(n) = aT(n/b) + f(n)$

Substitutionsmethode

Zur Lösung der Rekursionsgleichung, 2 Schritte:

- Rate die Form der Lösung, z.B. durch
 - Scharfes Hinsehen
 - kurze Eingaben ausprobieren und einsetzen
- Anwendung von vollständiger Induktion, um die Konstante zu finden und zu zeigen, dass das Geratene eine Lösung ist

Rekursionsbaum-Methode

Grundidee: Stelle das ineinander-Einsetzen als Baum dar und analysiere die Kosten

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

Geratenes kann durch Substitutionsmethode überprüft werden.

Mastermethode

Allgemeine Form der Rekursionsgleichung:

$T(n) = aT(n/b) + f(n)$ mit $a \geq 1, b > 1$ und $f(n)$ eine asymptotische positive Funktion.

- Problem wird in a Teilprobleme der Größe n/b aufgeteilt.
- Lösen jedes der a Teilprobleme benötigt Zeit $T(n/b)$.
- Funktion $f(n)$ umfasst die Kosten, um das Problem in Teilprobleme aufzuteilen und um die Teillösungen zu vereinigen.

4.3 Mastertheorem

Seien $a \geq 1$ und $b > 1$ Konstanten. Sei $f(n)$ eine positive Funktion und $T(n)$ über den nicht negativen ganzen Zahlen durch die Rekursionsgleichung

$$T(n) = aT(n/b) + f(n)$$

definiert, wobei wir n/b so interpretieren, dass damit entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ gemeint ist. Dann besitzt $T(n)$ die folgenden asymptotischen Schranken:

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

Das Mastertheorem verstehen:

In jedem der 3 Fälle wird die Funktion $f(n)$ mit $n^{\log_b a}$ verglichen.

Intuition:

1. Wenn $f(n)$ olynomial kleiner ist als $n^{\log_b a}$, dann $T(n) = \Theta(n^{\log_b a})$.
2. Wenn $f(n)$ und $n^{\log_b a}$ die gleiche Größe haben, dann gilt $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Wenn $f(n)$ polinomial größer ist als $n^{\log_b a}$ und $af(n/b) \leq cf(n)$ erfüllt, dann ist $T(n) = \Theta(f(n))$.

Nicht abgedeckte Fälle sind:

1. Wenn $f(n)$ kleiner als $n^{\log_b a}$, aber nicht polinomial kleiner.
2. Wenn $f(n)$ größer als $n^{\log_b a}$, aber nicht polinomial größer.
3. Regularitätsbedingung $af(n/b) \leq cf(n)$ nicht erfüllt.

5 Sortieralgorithmen

5.1 Sortieralgorithmen: Theorie

Warum möchte man Sortieren

- Sachen Packen
- Zeitmanagement
- etc

Bedingungen

- Ausgangspunkt als Folge von Datensätzen
- Zu sortierenden Elemente heißen Schlüsselwerte (-werte)
- Ziel: Schlüsselwerte nach einem Ordnungsschlüssel zu sortieren

Bedingung → Schlüsselwerte müssen vergleichbar sein

Vergleichskriterien

- Berechnungsaufwand: $O(n)$
- Effizienz: Best Case vs. Average Case vs. Worst Case
- Speicherbedarf: → in-place: zusätzlicher Speicher von der Eingabegröße unabhängig
→ out-of-place: Speichermerhbedarf von eingabegröße abhängig
- Stabilität: Erhaltenbleiben des sekundären Sortierschlüssels

▷ Sortierverfahren abhängig von Anwendung, Unterschiedliche Zahl von Vertauschungen vs Vergleichungen

Vergleichsverfahren

Korrekttheit mittels Schleifeninvariante

- Eine Schleifeninvariante beschreibt was bei dem Durchlauf einer Schleife zu Beginn immer gegeben ist
Damit kann die Korrektheit eines Algorithmus bewiesen werden
 - ▷ Vergleichbar mit einem Induktionsbeweis
- Eine Schleifeninvariante muss 3 Eigenschaften erfüllen:
 - Initialisierung: Invariante ist vor jeder Iteration wahr
 - Fortsetzung: Wenn die Invariante vor der Schleife wahr ist dann bleibt sie auch bis zum Beginn der nächsten Iteration wahr
- Bei dem Insertionsort:
 - Zu Beginn ist die linke Teilmenge immer sortiert

5.2 Entwurfsprinzipien des Sortieren

▷ InsertionSort: inkrementelle Herangehensweise

▷ Divide-and-Conquer Ansatz (Teile-und-Beherrsche):

- Ansatz um Sortieralgorithmus zu entwerfen
- Laufzeit: im schlechtesten Fall immer noch besser als InsertionSort
- Gibt Methoden mit welchen wir die Laufzeit einfach bestimmen können

Devide-and-Conquer

Prinzip: Zerlege das Problem und löse es direkt *oder* durch weitere Zerlegung.

Divide: Teile das Problem in mehrere Teilprobleme auf, die kleinere Instanzen des gleichen Problems darstellen.

Conquer: Beherrsche die Teilprobleme rekursiv. Wenn die Teilprobleme klein genug sind, dann löse die Teilprobleme auf direktem Weg.

Combine: Vereinige die Lösung der Teilprobleme zur Lösung des ursprünglichen Problems.

Beispiel: MergeSort

5.3 InsertionSort

Grundidee

- Sortieren durch Einfügen:
- Die Linke Teilmenge ist Sortiert, und passend wird immer ein neues Element eingefügt

Eigenschaften

- Stabiler Algorithmus

Code

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

Beweis der Korrektheit

Um die Korrektheit des Algorithmuses zu bestimmen nutzen wir die Schleifeninvariante

- Initialisierung: Beginn mit $j=1$, also mit dem Teilstück bis $j-1$, besteht aus einem Element und ist Sortiert
- Fortsetzung: Zu zeigen ist, dass die Invariante bei jeder Iteration erhalten bleibt. Die for-Schleife verschiebt jedes Element der sortierten Menge um eins nach rechts bis das zu sortierende Element eingefügt werden kann. Inkrementieren von j erhält die Schleifeninvariante
- Terminierung: Abbruchbedingung for-Schleife wenn $j > A.length - 1$. Jede Iteration erhöht j . Dann bei Abbruch $j=n$ und einsetzen in Invariante liefert das Teilstück $A[0..n-1]$ welches aus den ursprünglichen Elementen besteht → Teilstück = gesamtes Feld

5.4 BubbleSort

Grundidee

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

Code

```
BUBBLE-SORT(A)
1 FOR i = 0 TO A.length - 2
2   FOR j = A.length - 1 DOWNTONTO i + 1
3     IF A[j] < A[j - 1]
4       SWAP(A[j], A[j - 1])
```

Analyse

Anzahl der Vergleiche

- es werden stets alle Elemente der Teilfolge verglichen
- unabhängig von der Sortierung

Anzahl der Vertauschungen

- Best case: 0 Vertauschungen
- Worst case: $\frac{n^2-n}{2}$ Vertauschungen

Komplexität

- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$
- Worst case: $\Theta(n^2)$

5.5 SelectionSort

Grundidee

▷ Sortieren durch direktes Auswählen

- MinSort: "wähle kleinstes Element in Array und tausche es nach vorne
- MaxSort: "wähle größtes Element in Array und tausche es nach hinten

Code

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

5.6 MergeSort

Grundidee

▷ Baut auf dem Devide-and-Conquer Prinzip auf

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

Code

Prinzip des MergeSorts

```
MERGE-SORT(A, p, r)
1  IF p < r
2      q =  $\lfloor (p+r)/2 \rfloor$ 
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Beim Halbieren wird immer abgerundet, so kann jede Teilfolge sortiert werden.

Prinzip des gesamtes Codes

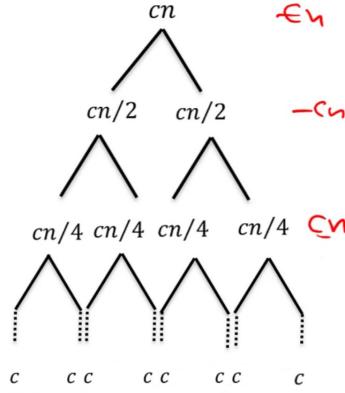
```
MERGE(A, p, q, r)           A [p .. r]   q: geteilt
1  n1 = q - p + 1
2  n2 = r - q
3  let L[0..n1] and R[0..n2] be new arrays
4  FOR i = 0 TO n1 - 1
    L[i] = A[p + i]
5  FOR j = 0 TO n2 - 1
    R[j] = A[q + j + 1]
8  L[n1] = ∞
9  R[n2] = ∞
10 i = 0
11 j = 0
12 FOR k = p TO r
13     IF L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     ELSE A[k] = R[j]
17         j = j + 1
```

Analyse

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

- **Divide:** Es wird einfach die Mitte des Feldes berechnet. Dies benötigt konstante Zeit, also $\Theta(1)$
- **Conquer:** Wir lösen zwei Teilprobleme der Größe $\frac{n}{2}$ rekursiv. Dies trägt mit $2T(\frac{n}{2})$ zur Laufzeit bei
- **Combine:** Die Prozedur MERGE auf einem Teifeld der Länge n lineare Zeit, also $\Theta(n)$

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



Daraus ergibt sich eine Komplexität von $\Theta(n \log n)$

Beweis der Korrektheit

Schleifeninvariante: Zu Beginn jeder Iteration der for-Schleife (Zeile 12-17) enthält das Teifeld $A[p \dots k - 1]$ die $k - p$ kleinsten Elemente aus $L[0 \dots n_1]$ und $R[0 \dots n_2]$ in sortierter Reihenfolge. Weiter sind $L[i]$ und $R[j]$ die kleinsten Elemente ihrer Arrays, die noch nicht zurück kopiert wurden.

Initialisierung: Vor der ersten Iteration gilt $k = p$.

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

Fortsetzung: Müssen zeigen, dass Schleifeninvariante erhalten bleibt. Dafür nehmen wir an, dass $L[i] \leq R[j]$. Dann ist $L[i]$ kleinstes Element, welches noch nicht zurück kopiert wurde. Da Array $A[p \dots k - 1]$ die $k - p$ kleinsten Elemente enthält, wird der Array $A[p \dots k]$ die $k - p + 1$ kleinsten Elemente enthalten nachdem der Wert nach der Durchführung von Zeile 14 kopiert wurde. Nach Erhöhung der Variablen k und i stellt die Schleifeninvariante für die nächste Iteration wieder her.

Wenn $L[i] > R[j]$ dann analoges Argument mit Zeilen 16-17.

Terminierung: Beim Abbruch gilt $k = r + 1$. Durch die Schleifeninvariante enthält $A[p \dots r]$ die kleinsten Elemente von $L[0 \dots n_1]$ und $R[0 \dots n_2]$ in sortierter Reihenfolge. Alle Elemente außer der Sentinels wurden korrekt zurück kopiert.

5.7 QuickSort

Grundidee

- Wähle ein Pivotelement x des Arrays
- Zerlege den Array in zwei Teilarrays
 - Erster Teilarray: Enthält alle Elemente $y \leq x$
 - Zweiter Teilarray: Enthält alle Elemente $y > x$
- Sortiere rekursiv auf den Teillisten mit Quicksort
- Einelementige Listen sind schon sortiert

Eigenschaften

- Instabiler Sortieralgorithmus
- Basiert auf Divide and Conquer

Divide-and-Conquer

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

- **Conquer:** Sortieren beider Teilarrays $A[p \dots q - 1]$ und $A[q + 1 \dots r]$ durch rekursiven Aufruf von Quicksort
- **Combine:** Da die Teilarrays bereits sortiert sind, ist keine weitere Arbeit nötig um diese zu vereinigen. $A[p \dots r]$ ist nun sortiert.

Code

```
QUICKSORT (A, p, r)
1  IF  p < r
2      q = PARTITION (A, p, r)
3      QUICKSORT (A, p, q - 1)
4      QUICKSORT (A, q + 1, r)
```

```
PARTITION (A, p, r)
1  x = A[r]
2  i = p - 1
3  FOR j = p  TO  r - 1
4      IF  A[j] ≤ x
5          i = i + 1
6      SWAP (A[i], A[j])
7  SWAP (A[i + 1], A[r])
8  RETURN i + 1
```

Im Folgenden gucken wir uns den Algorithmus genauer an ⇒

Zeile 1: Pivotelement wählen
Zeile 2: Index i setzen
Zeile 3-6: Teilarrays mit Element füllen
Zeile 7: Pivotelement tauschen
Zeile 8: Neuer Index des Pivotelements

9	10	11	12	13	14	15	16
2	8	7	1	3	5	6	4
9	10	11	12	13	14	15	16

2	8	7	1	3	5	6	4
2	8	7	1	3	5	6	4

9	10	11	12	13	14	15	16
2	1	3	4	7	5	6	8
9	10	11	12	13	14	15	16

⇒

Korrektheit von Quicksort

- **Schleifeninvariante:** Zu Beginn jeder Iteration der for-Schleife (Zeile 3-6) gilt für den Arrayindex k folgendes:
 - Ist $p \leq k \leq i$, so gilt $A[k] \leq x$.
 - Ist $i + 1 \leq k \leq j - 1$, so gilt $A[k] > x$.
 - Ist $k = r$, so gilt $A[k] = x$.
- **Initialisierung:** Vor der ersten Iteration gilt $i = p - 1$ und $j = p$. Da es keine Werte zwischen p und j gibt und es auch keine Werte zwischen $i + 1$ und $j - 1$ gibt, sind die ersten beiden Eigenschaften trivial erfüllt.
Die Zuweisung in Zeile 1 sorgt für die Erfüllung der dritten Eigenschaft.
- **Fortsetzung:** Zwei mögliche Fälle durch Zeile 4.
Wenn $A[j] > x$, dann inkrementiert die Schleife ur den Index j . Dann gilt Bedingung 2 für $A[j - 1]$ und alle anderen Einträge bleiben unverändert.
Wenn $A[j] \leq x$, dann wird index i inkrementiert und die Einträge $A[i]$ und $A[j]$ getauscht und schließlich der Index j erhöht. Wegen des Vertauschens gilt $A[i] \leq x$ und Bedingung 1 ist erfüllt. Analog gilt $A[j - 1] > x$, da das Element welches mit $A[j - 1]$ vertauscht wurde wegen der Invariante gerade größer als x ist.
- **Terminierung:** Bei der Terminierung gilt, dass $j = r$. Daher gilt, dass jeder Eintrag des Arrays zu einer der drei durch die Invariante beschriebenen Menge gehört.

6 Datenstrukturen

6.1 Datenstrukturen: Theorie

Definiton

Eine Datenstruktur ist eine Methode, Daten abzuspeichern und zu organisieren sowie den Zugriff auf die Daten und die Modifikation der Daten zu erleichtern

Besipiele

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

Begriffsunterscheidung

Abstrakter Datentyp ("was")
Beschreibt Stack mit Operationen
Datenstrukturen ("wie")
Beschreibt die Stack-Operationen als Array oder Verkettete Liste

6.2 Stacks

Operationen:

- new(S) - erzeugt neuen (leeren) Stack
- isEmpty(S) - gibt an, ob Stack S leer ist
- pop(S) - gibt oberstes Element vom Stack S zurück und löscht es, falls vorhanden
- push(S,k) - schreibt k als neues oberstes Element auf S
- → LIFO - last in, first out

Intuitive erwartete Bedingungen sind erfüllt, auch ohne algebraische Spezifikation

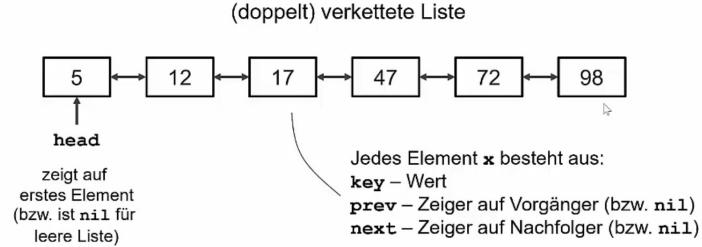
Darstellung als Array

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

Bei der Darstellung als Array kann die maximale Größe erreicht werden. Danach wird beim Hin zufügen das ganze Array in ein größeres kopiert. Da dies in $\Theta(n^2)$ liegt wird ein Faktor festgelegt. So wird beispielweise bei erreichen der maximalen Größe die Größe verdoppelt, haben wir weniger als 1/Faktor*Faktor, also weniger als ein Viertel der Elemente belegt so halbieren wir unser Array.

6.3 Verkettete Listen

Darstellung:



Operationen und Laufzeit:

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(1)$
Suchen	$\Theta(n)$

6.4 Queues

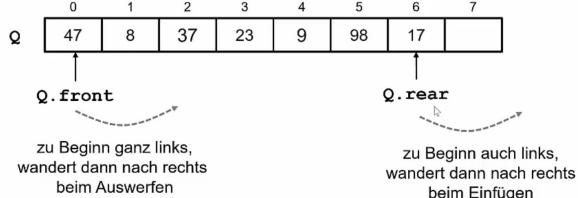
Operationen:

- `new(Q)` - erzeugt neue (leere) Queue
- `isEmpty(Q)` - gibt an, ob Queue leer ist
- `dequeue(Q)` - gibt vorderstes Element der Queue zurück und löscht es
- `enqueue(Q, k)` - schreibt k als neues hinterstes Element auf Q
- → FIFO - first in, first out

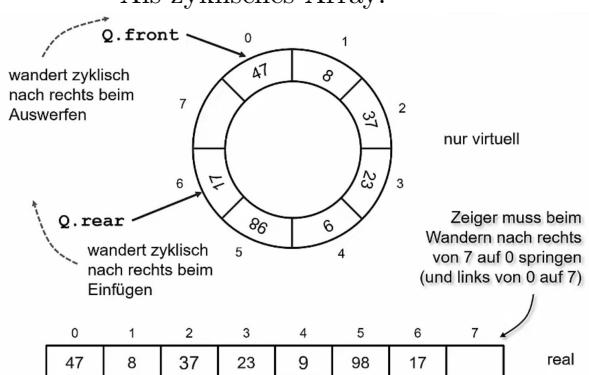
Darstellung als Array

Bei der Darstellung als Array stößt man nun auf Probleme. Es werden nun zwei Zeiger, und nicht mehr wie bei Stacks nur einer, benötigt. Da die Schlange aber nun durch das Array durchwandert bleibt sehr viel Arrayplatz ungenutzt, bzw. man bräuchte ein unendlich großes Array. Daher gibt es zwei Darstellungsmöglichkeiten:

Als Array:



Als zyklisches Array:

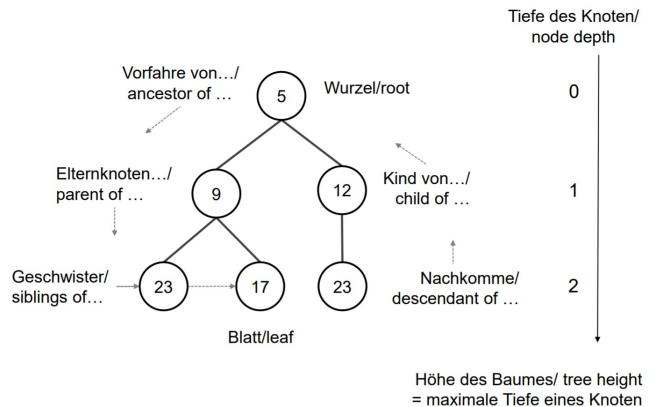


6.5 Binäräbäume - Bäume

Eigenschaften und Begrifflichkeiten

- Jeder Knoten ist eindeutig zu erreichen \Rightarrow azyklischer Graph
- Root, Leaf, Child, Descendent, Parent, Ancestor, Sibling \Rightarrow
- Halbblatt: Ein Knoten der genau ein Kind hat

Darstellung als Ungerichteter Graph



Charakteristik

Operation	Laufzeit*
Einfügen	$\Theta(1)$
Löschen	$\Theta(h)$
Suchen	$\Theta(n)$

Höhe/Tiefe

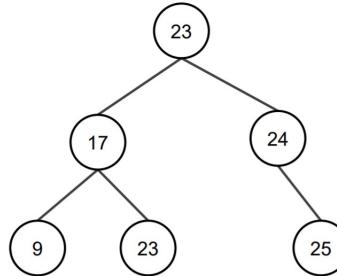
- Höhe des Leeren Baums per Konvention = -1
- Höhe (nicht leerer) Baum = max Höhe aller Teilbäume der Wurzel + 1

Inorder-Traversieren von Binäräbäumen

Das Inorder-Traversieren ist eine Möglichkeit durch alle Knoten eines Binärbaums zu Iterieren. Die Laufzeit beträgt $O(n)$. Dabei ist diese Vorgehensweise nicht eindeutig, das heißt mehrere Bäume können die gleiche Inorder-Traversierung haben.

Im allgemeinen werden die Knoten dabei in aufsteigender Reihenfolge ausgegeben.

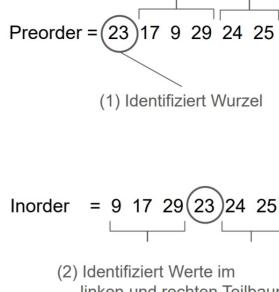
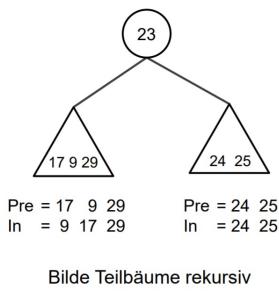
Inorder-Traversierung mit Beispiel



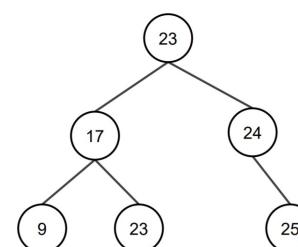
```
preorder(x)
1 IF x != nil THEN
2   print x.key;
3   preorder(x.left);
4   preorder(x.right);
```

preorder(T.root) ergibt
23 17 9 23 24 25

Bemerkung: Genauso gibt es noch Pre- und Postorder



Gilt analog für Postorder



```
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;
```

preorder(T.root) ergibt
23 17 9 23 24 25

postorder(T.root) ergibt
9 23 17 25 24 23

6.6 Binäre-Suchbäume

Eigenschaften

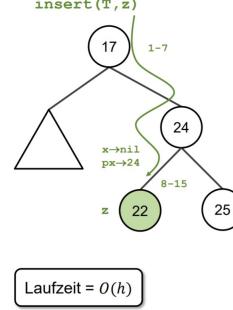
- Ein Binärersuchbaum basiert auf dem Prinzip eines Binärbaums
Hierbei gilt jedoch für jeden Knoten, dass der Wert seines rechten Kindes mindestens so groß sein muss wie der Werte des Knoten. Umgekehrt gilt dies auch für das linke Kind.
- Sei x Knoten im linken Teilbaum von $z \Rightarrow x.\text{key} \leq z.\text{key}$
- Sei y Knoten im linken Teilbaum von $z \Rightarrow y.\text{key} \geq z.\text{key}$

Suchen

- Durch die Teilsortierung, also die Bedingung eines Binärensuchbaums, kann der Suchbereich auf Teilbäume eingegrenzt werden.

```
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);
```

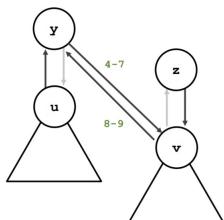
Einfügen



```
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;
```

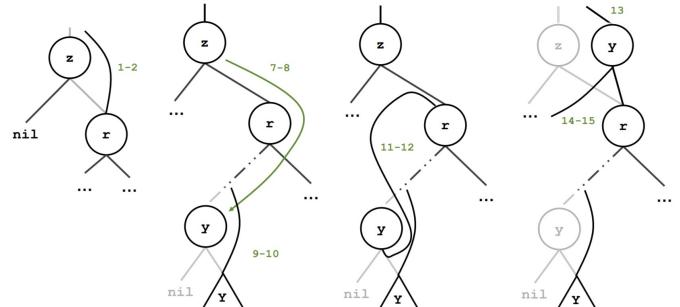
Löschen: Transplant



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

Laufzeit = O(1)

Löschen



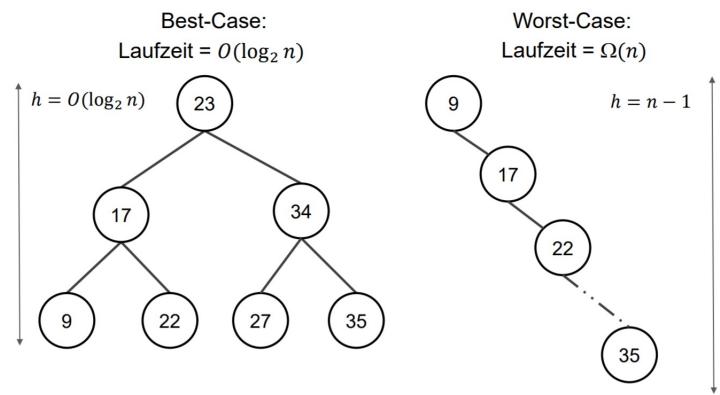
```
delete(T,z)
1 IF z.left==nil THEN
2   transplant(T,z,z.right);
3 ELSE
4   IF z.right==nil THEN
5     transplant(T,z,z.left);
6   ELSE
7     y=z.right;
8     WHILE y.left != nil DO y=y.left;
9     IF y.parent != z THEN
10      transplant(T,y,y.right);
11      y.right=z.right;
12      y.right.parent=y;
13      transplant(T,z,y);
14      y.left=z.left;
15      y.left.parent=y;
```

Charakteristik

Operation	Laufzeit*
Einfügen	O(h)
Löschen	O(h)
Suchen	O(h)

Höhe eines BST

- Da die Laufzeit abhängig von der Höhe ist, lohnt es sich hier einen weiteren Blick drauf zu werfen
- Im Worst Case beträgt die Laufzeit nun $\Omega(n)$, im Best Case $O(\log_2 n)$



6.7 AVL Bäume

Eigenschaften

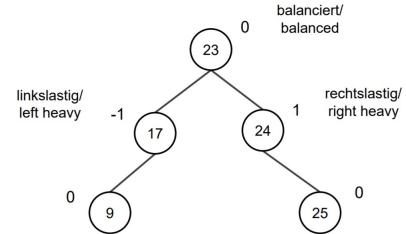
- Ein AVL (Adelson-Velski und Landis) Baum ist zunächst ein normaler Binärer Suchbaum
- Optimiert: $h \leq 2 * \log n(RBT)$ vs $h \leq 1.441 * \log n(AVL/Bume)$

Charakteristik

AVL-Bäume	
Operation	Laufzeit
Einfügen	$\Theta(\log n)$
Löschen	$\Theta(\log n)$
Suchen	$\Theta(\log n)$

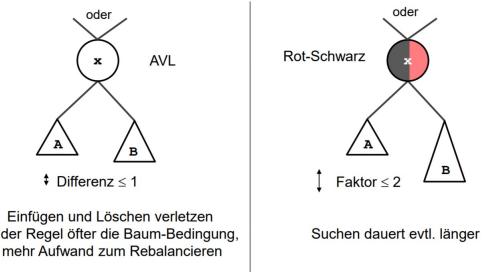
Definition

- $B(x) = Hhe(rechterTeilbaum) - Hhe(linkerTeilbaum)$
- Für alle Knoten x aus einem Baum muss $B(x) \in -1, 0, 1$
- Mit n Knoten ergibt sich: $\min(n_h) = \text{Fibonacci}_{h+2} - 1$



AVL ⊂ RBT

- AVL-Bäume sind eine Teilmenge der Rot-Schwarz-Bäume
- Da ihre Teilbaumhöhe kleiner ist, als die eines RBT, lassen sich alle AVL-Bäume als RBT darstellen



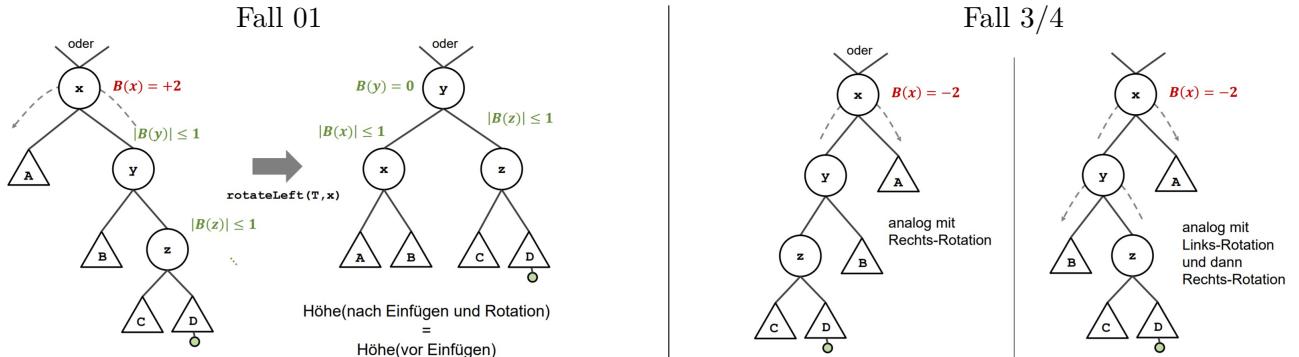
Einfügen und Löschen

- Sowohl das einfügen als auch das Löschen funktioniert gleich wie bei Binären Suchbäumen
- Am Ende beider Operationen kann es notwendig sein zu Rebalancieren

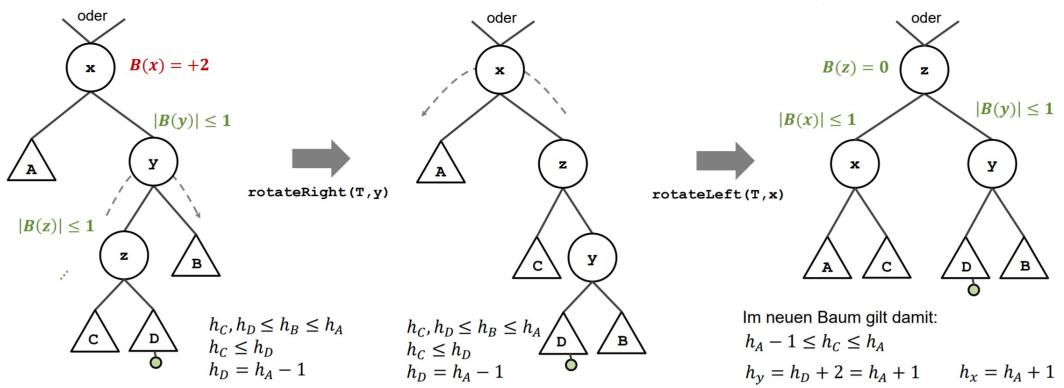
Einfügen - Rebalance: Fall 01:

Manchmal kann schon eine einfache Rotation das Gleichgewicht wiederherstellen. Da dies nicht immer der Fall ist, und die Analyse der weiteren Schritte sehr Komplex ist, belassen wir es bei diesem einfachen Fall. In allen Fällen muss nur einmal Rebalanciert werden, aber es muss die zu balancierende Stelle gefunden werden ($O(h)$)

Einfügen: Rebalancieren



Fall 02



Im neuen Baum gilt damit:

$$h_A - 1 \leq h_C \leq h_A$$

$$h_y = h_D + 2 = h_A + 1 \quad h_x = h_A + 1$$

6.8 Splay-Bäume

Eigenschaften

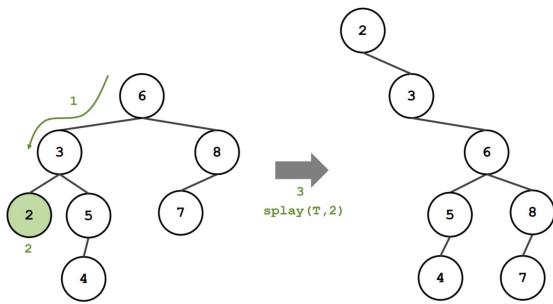
- Selbst organisierende Datenstruktur
- Beruht auf Zeitlicher Lokalität (wird ein Objekt einmal gesucht, wird es vermutlich in naher Zukunft erneut gesucht)
- Anwendungsbeispiel: SQUID - Web Cache Proxy
- *Splay – Baum $\subset BST$*
- Prinzip: Spüle neu eingefügten oder gesuchten Knoten an die Wurzel
- In der Realität umstritten ob die Implementierung sinnvoll ist

Einfügen und Suchen

- Funktioniert analog zu BST, aber mit einer Splay operation
- Bei dieser wird ein Knoten an die Wurzel gespült

Einfügen:

- Nach dem Einfügen wird das Ergebnis an die Wurzel gespült



Suchen:

- Am Ende des Suchens wird ein eventuell gefundenes Ergebnis gespült

```
search(T,k)
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;
```

Gesamtaufzeit $O(h)$

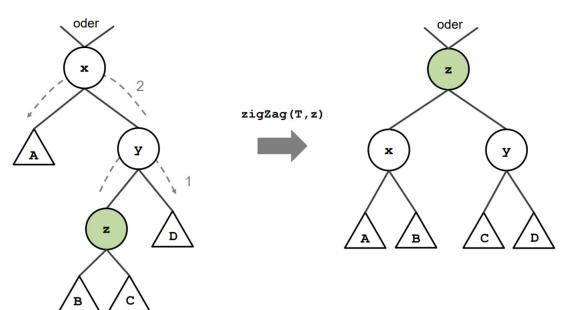
Splay

- Die Spüloperation (Splay) dient dazu einen Knoten an die Wurzel zu spülen
- Dies dient dazu den Baum im allgemeinen ausgeglichen zu halten
- Gespült wird immer nach Einfügen und Suchen, sowie vor dem Löschen
- Das Spülen basiert dabei auf drei Unterroutinen:

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

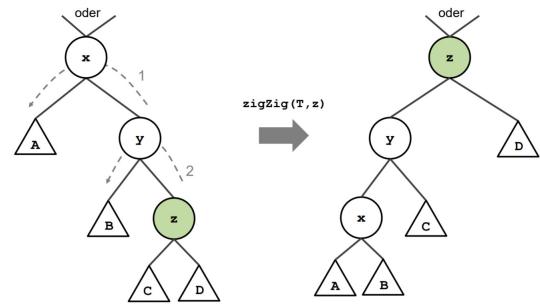
ZigZag

- 1. Rechtsrotation (Weg vom SStamm") um x.parent
- 2. Linkssrotation (Zum SStamm") um x.parent



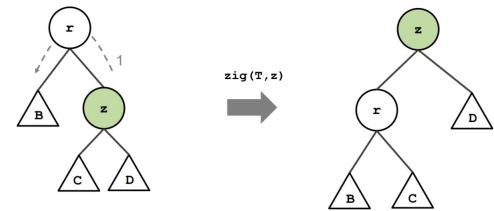
ZigZig

- 1. Linksrotation (Zum SStamm") um x.parent.parent
- 2. Linksrotation (Zum SStamm") um x.parent



Zig

- 1. Linksrotation (Zum SStamm") um x.parent



(Falls z direkt unter der Wurzel hängt)

Löschen

Schritt 1:

- Suche zu löschen Knoten und spüle ihn an die Wurzel



Schritt 2:

- Löschen von x
- Ist einer der resultierenden Teilbäume leer, sind wir fertig



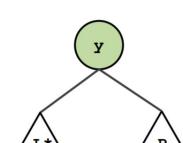
Schritt 3:

- Spüle den "größten"Knoten y in L nach oben
- Da es mindestens der größte Wert ist, hat dieser danach nur ein linkes Kind



Schritt 4:

- Hänge R an y an



6.9 Binäre Max-Heaps

Eigenschaften

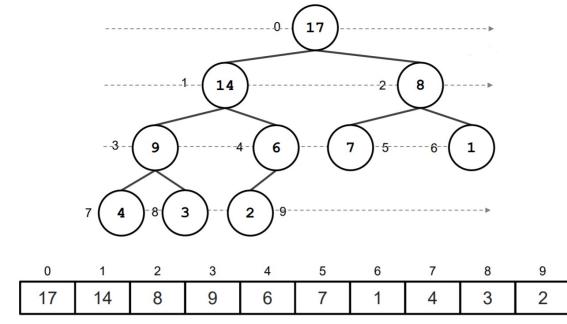
- Ein binärer Max-Heap ist ein binärer Baum (**kein Binärer Suchbaum**), der:
- (1) "bis auf das unterste Level vollständig und von Links gefüllt ist"
- (2) Für alle Knoten $x \neq T.\text{root}$ gilt, $x.\text{parent}.key \geq x.key$
- Genauso gibt es Min-Heaps, wobei sich die Bedingung umdreht

Heaps durch Arrays

- Bei der Nummerierung gehen wir von oben links nach unten rechts vor
- Anzahl Knoten in H ist gleich $H.\text{length}$
- Duale Sichtweise, als Pointer oder als Index:

Duale Sichtweise
als Pointer
oder als Array:

$$\begin{aligned} j.\text{parent} &= \left\lceil \frac{j}{2} \right\rceil - 1 \\ j.\text{left} &= 2(j + 1) - 1 \\ j.\text{right} &= 2(j + 1) \end{aligned}$$



Einfügen

- Wir fügen neue Knoten immer an das des Arrays ein
- Eventuell wird Heap Bedingung verletzt
- Dann Tauschen wir solange nach oben, bis dies nicht mehr der Fall ist

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

```
insert(H,k) //als (unbeschränktes) Array
1 H.length=H.length+1;
2 H.A[H.length-1]=k;

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

Löschen des Maximums

- Beim Löschen des Maximums wird das Maximum, also die Wurzel zurückgegeben und durch das letzte "Blatt" also das zuletzt eingefügte Element ersetzt
- Danach wird die Heap Bedingung mittels der Routine `heapify` wieder hergestellt
- Heapify sucht das Element dabei immer weiter nach unten bis die Bedingung wieder hergestellt ist. Dabei wird immer in den Teilbaum mit dem größeren Wert getauscht

```
heapify(H,i) //als (unbeschränktes) Array
1 maxind=i;
2 IF i.left<H.length AND H.A[i]<H.A[i.left] THEN
3   maxind=i.left;
4 IF i.right<H.length AND H.A[maxind]<H.A[i.right] THEN
5   maxind=i.right;
6 IF maxind != i THEN
7   SWAP(H.A,i,maxind);
8   heapify(H,maxind);
```

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

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

Heap-Konstruktion aus Array

- Aus einem Array lässt sich relativ einfach ein Heap erzeugen
- Da Einelementige Heaps trivialerweise alle Heap Bedingungen erfüllen müssen wir die Bedingung nur bei allen Elementen außer den Blättern wieder herstellen
 - Blätterindizes:
- Die Blätter Indices ergeben sich: $\lceil(n - 1)/2\rceil, \dots, n - 1$.

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

```
buildHeap(H,A) //Array A schon nach H.A kopiert
1 H.length=A.length;
2 FOR i = ceil((H.length-1)/2)-1 DOWNTO 0 DO
3     heapify(H,A,i);
```

Heap-Sort

- Mittels eines Heapes lässt sich auch sortieren
- Dabei wird der Heap z.B. erst aus einem Array gebildet, und danach werden alle Elemente wieder ins Array eingefügt
- (gibt Array in absteigender Reihenfolge zurück)

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

```
heapSort(H,A) //Array A schon nach H.A kopiert
1 buildHeap(H,A);
2 WHILE !isEmpty(H) DO PRINT extract-max(H);
```

6.10 B-Bäume

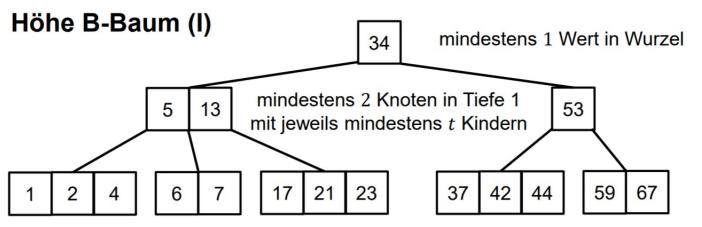
Eigenschaften

- B könnte für Balanced, Bayer-McCreight oder für Boeing stehen
- Ein B-Baum (vom Grad t) ist ein Baum, bei dem
 - (1) jeder Knoten (außer der Wurzel) mindestens $t - 1$, und maximal $2t - 1$ Werte hat
 - (2) die Werte innerhalb eines Knoten sind aufsteigend sortiert
 - (3) die Blätter alle die gleiche Höhe haben
 - (4) jeder innere Knoten mit n Werten und $n + 1$ Kindern, so dass für alle Werte k_j aus dem j -ten Kind gilt: $k_0 \leq key[0] \leq k_1 \leq \dots \leq k_{n-1} \leq key[n - 1] \leq k_n$

Höhe B-Baum

- Für größere Werte ist ein B-Baum also flacher als (vollständiger) Binärbaum

Ein B-Baum vom Grad t mit n Werten hat maximale Höhe $h \leq \log_t \frac{n+1}{2}$.



mindestens $2t^2$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern,
mindestens $2t^3$ Knoten in nächster Tiefe mit jeweils mindestens t Kindern, usw.

Und in jedem Knoten außer Wurzel mindestens $t - 1$ Werte.

Anzahl Werte n im B-Baum im Vergleich zu Höhe h :

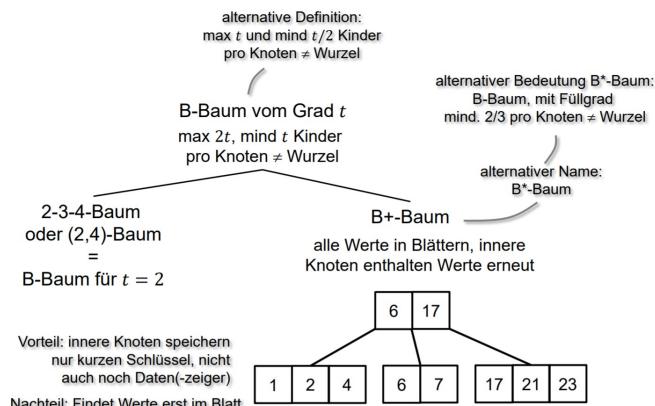
$$n \geq 1 + (t - 1) \cdot \sum_{i=1}^h 2t^{i-1} = 1 + 2(t - 1) \cdot \frac{t^{h-1}}{t-1} = 2t^h - 1$$

also: $\log_t \frac{n+1}{2} \geq h$.

Anwendung

- B-Bäume werden dort verwendet wo man im allgemeinen öfter auf ganze Blöcke von Daten zugreifen möchte
- Beispiel: Festplatten, hier wird Blockweise auf die Werte zugegriffen um gleichzeitig mehrere Indices zu bekommen
- Beispiel: Auch die MySQL Datenbank nutzt B-Bäume zur Speicherung von manchen Indices

Baumkunde



Suchen

- Das Suchen funktioniert ähnlich wie bei Binären Suchbäumen

maximal
 $2t = O(1)$
Iterationen

search(x, k)

```

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

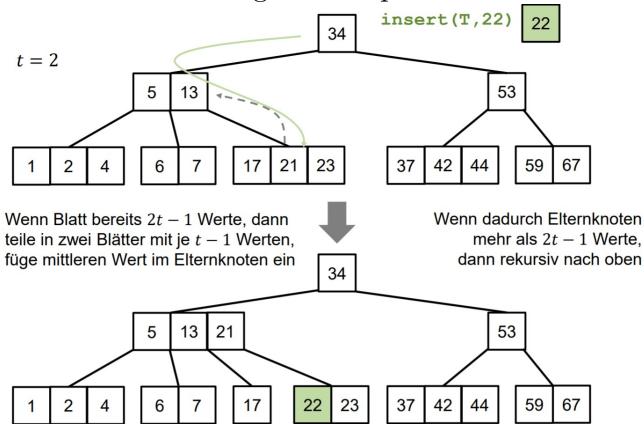
```

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

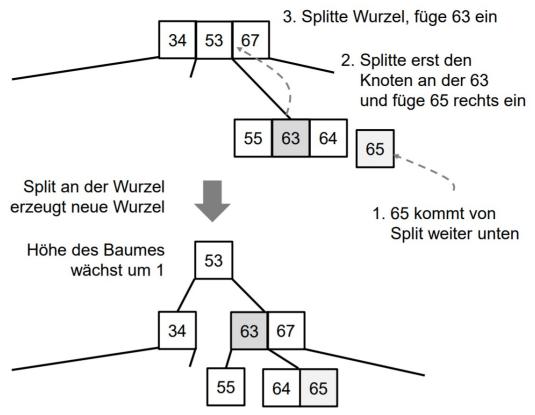
Einfügen

- Zuerst wird die Position gesucht an der eingefügt werden muss
- Einfügen erfolgt zunächst immer im Blatt
- Hat das Blatt nach Einfügen immer noch weniger als $2t - 1$ Werte sind wir fertig
- Hat der Knoten schon maximale Werte, muss der Knoten gesplittet werden
- Beim Splitten wird der Mittlere Wert in den Eltern Knoten eingefügt
- Eventuell muss hier rekursiv nach oben gegangen werden

Einfügen und Splitten



$t = 2$



Suchen, dann Splitte vs Suchen und Splitte

- Beim Einfügen wird erst gesucht, und danach wird Gesplittet
- Um das zweifache traversieren durch den Baum zu vermeiden kann man auch anders vorgehen
- bei deiser Strategie wird beim suchen gleichzeitig gesplittet

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

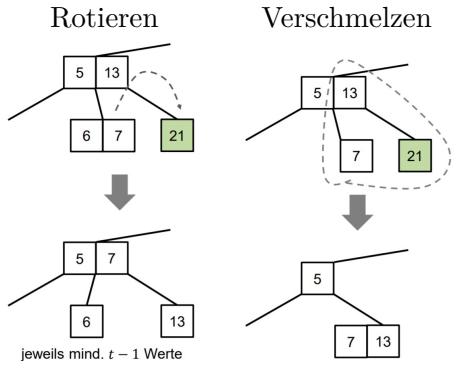
```

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

Löschen

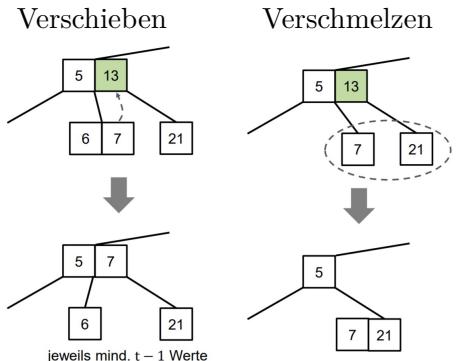
Löschen im Blatt:

- Wird in einem Blatt gelöscht, dass zu wenig Knoten hat, gibt es zwei Möglichkeiten
- (1 Rotieren) Hat der Nachbarknoten noch ausreichen Elemente, dann kann rotiert werden
- (2 Verschmelzen) Hat kein Nachbarknoten ausreichend Elemente zum Rotieren, müssen wir Verschmelzen
- Dies läuft daraus hinaus, dass der Eltern Knoten ein Element weniger hat, und zwei Kindknoten zu einem Verschmelzen
- Hierbei kann es passieren, dass dabei die Balance im Elternknoten gestört ist. Dies muss rekursiv behoben werden



Löschen im inneren Knoten:

- Wird in einem inneren Knoten gelöscht gibt es zwei Möglichkeiten
- (1 Verschieben) Wenn einer der beiden Kindknoten mehr als $t - 1$ Werte hat, kann der gelöschte Wert ersetzt werden, mit dem
 - größten Wert des linken Kindes
 - kleinsten Wert des rechten Kindes
- (2 Verschmelzen) Wenn beide Kindknoten jeweils zu wenig Elemente haben (also $t - 1$) werden diese beiden Knoten verschmolzen
- Auch hierbei kann es passieren, dass der Elternknoten zu wenige Elemente hat, hier wird dann rekursiv vorgegangen



Löschen und Splitte

Cahrakteristik

Operation	Laufzeit
Einfügen	$\Theta(\log_t n)$
Löschen	$\Theta(\log_t n)$
Suchen	$\Theta(\log_t n)$

```

delete(T, k)
  Laufzeit  $O(t \cdot h) = O(\log_t n)$ 

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
  )

```

6.11 Skip-Lists

Eigenschaften

- Skip Listen sind ein Beispiel für Randomisierte Datenstrukturen
- Dies bedeutet: Verhalten hängt von zufälligen Entscheidungen der Datenstruktur ab
- Gegensatz: deterministische Datenstrukturen - Verhalten für identische Eingaben immer gleich

Anwendungen

- Einfügen / Löschen unterstützen parallele Verarbeitung (z.B. Multi-Core-Systeme), da nur sehr lokale Änderungen (im Vergleich zum Re-Balancieren bei Bäumen)
- Dafür logarithmische Laufzeit nur um Durchschnitt

Idee von Skip-Listen

- Eine Skip Liste besteht zunächst aus einem Array, welches erstes Element auf $-\infty$ gesetzt
- Darüber hinaus gibt es eine Express-Liste
 - Eine Express-Liste wird mit einigen Elementen der Skip-Liste gefüllt
- Es werden zusätzliche Zeiger benötigt \Rightarrow

Charakteristik

Operation	Laufzeit*
Suchen	$\Omega(n)$
Löschen (Wert)	$\Omega(n)$
Einfügen	$\Omega(n)$

*Worst Case

*im Durchschnitt

Operation	Laufzeit*
Einfügen	$\Theta(\log_{1/p} n)$
Löschen	$\Theta(\log_{1/p} n)$
Suchen	$\Theta(\log_{1/p} n)$

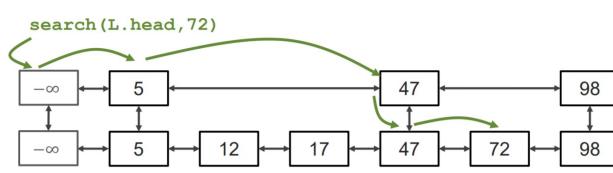
Θ -Notation versteckt (konstanten) Faktor $1/p$

Ω -Notation versteckt (konstanten) Faktor $1/p$

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

Suchen mittels Express-Listen

- Beginnend in der Express-Liste: Element gefunden? \rightarrow Element ausgeben
- Nächstes Element in Express-Liste kleiner gleich gesuchtes Element? \rightarrow Weiter nach Rechts
- Nächstes Element in Express-Liste größer als gesuchtes Element? \rightarrow Nach unten in nächste Liste

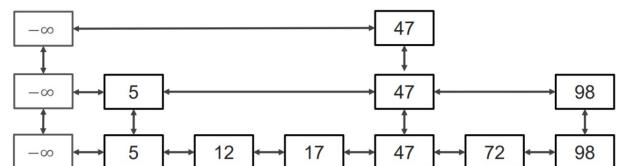


```

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;
  
```

Verbesserung

- Express-Liste ist wieder eine Skip-Liste
- Beispiel: jede Express-Liste hat die Hälfte der Elemente der vorherigen Liste
ergibt $\frac{n}{2} + \frac{n}{4} + \dots + 2 + 1 \leq n$ zusätzliche Elemente



Auswahl der Elemente für Express-Liste

- Idee: Wähle jedes Element aus Liste mit Wahrscheinlichkeit p (z.B. $p = \frac{1}{2}$) für übergeordnete Liste
- Wie viele Elemente gibt es auf den jeweiligen Ebenen?

Linearität des Erwartungswerts

$$E[X] = \sum_{i=1}^n x_i \cdot \text{Prob}[X = x_i]$$

Linearität des Erwartungswerts

Gegeben Zufallsvariablen X_1, \dots, X_n mit Erwartungswerten $E[X_i]$

Dann gilt: $E[\sum_{i=1}^n a_i \cdot X_i] = \sum_{i=1}^n a_i \cdot E[X_i]$

Bei unseren Skip-Listen:

X_i = Zufallsvariable, ob i -tes Element ausgewählt, $\text{Prob}[X_i = 1] = E[X_i] = p$

Also $E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = pn$ Elemente im Durchschnitt

Für nächste Ebene $\text{Prob}[X_i = 1] = E[X_i] = p^2$, also $p^2 n$ Elemente usw.

Erwartungswert geometrisch verteilter Zufallsvariablen

Wie oft muss man im Durchschnitt würfeln, bis eine 6 kommt?

Erwartungswert geometrisch verteilter Zufallsvariablen

Gegeben 0-1-Zufallsvariable X mit $\Pr[X = 1] = p > 0$

Zufallvariable Y zählt, wie oft man X unabhängig wiederholt, bis $X = 1$

Dann gilt: $E[Y] = \frac{1}{p}$

Also im Durchschnitt muss man 6-mal würfeln.

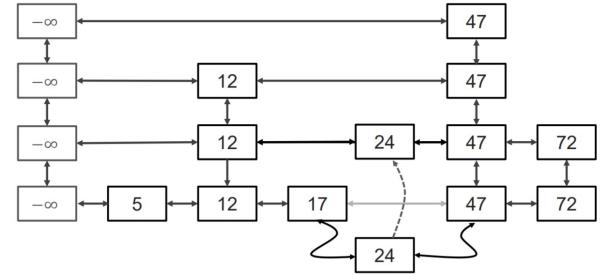
Aber: im Worst-Case würfelt man öfter!

Durchschnittliche Laufzeit für Suchen

- Im schlimmsten Fall endet unsere Suche erst auf der untersten Liste
- Im Durchschnitt machen wir nur $\frac{1}{p}$ Schritte auf jeder Ebene, bevor wir eine Ebene nach oben gegangen sind
- Wenn die Skip-Liste die Höhe h hat, brauchen wir also im Durchschnitt $\frac{1}{p} + \frac{1}{p} + \dots + \frac{1}{p} = \frac{h}{p} = O(h) = O(\log n)$

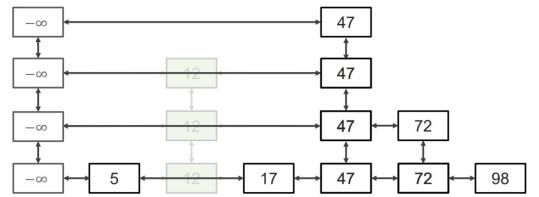
Einfügen

- Beim einfügen wird als erstes die richtige Stelle gesucht werden
- Dort wird das Element dann in die Ursprüngliche Liste eingefügt
- Danach wird das Element mit der Wahrscheinlichkeit p in die Liste obendrüüber eingefügt



Löschen

- Entfernen eines Elementes funktioniert analog wie zu LinkedLists
- Nachdem das Element gefunden ist, müssen alle Zeiger umgeschrieben werden



6.12 Hash-Tables

Eigenschaften

- Hashtables sind ein Beispiel für Randomisierte Datenstrukturen
- Hashtables sind eine gute Datenstrukturen zum Suchen und Einfügen
- Allerdings kann nicht auf Nachbarelemente zugegriffen werden, und die Elemente sind nicht sortiert
- Anwendung: Zum Beispiel bei SQL mittels Hash Indices, um Daten schneller zu ermitteln

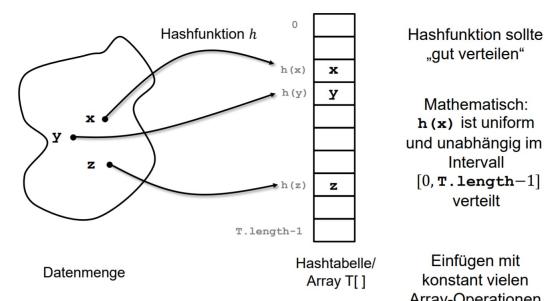
Charakteristik

Operation	Laufzeit*
Einfügen	$\Theta(1)**$
Löschen	$\Theta(1)$
Suchen	$\Theta(1)$

**sogar Worst-Case

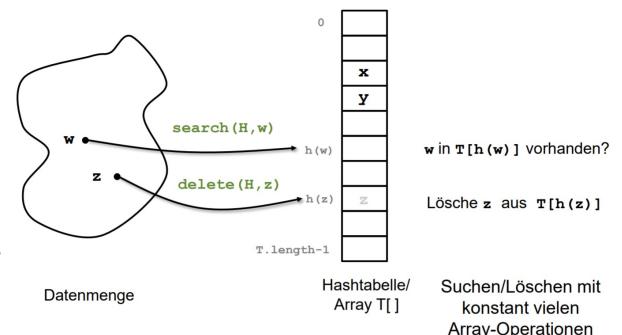
Idee von Hashtables

- Eine Hashfunktion kann als Array dargestellt werden
- Die Menge an Daten wird mittels einer Hashfunktion einsortiert
- Jeder Wert wird durch die Hashfunktion gemapped, und der daraus resultierende Wert ist der neue Arrayindex des Elements
- Damit keine Kollisionen auftreten, sollte die Hashfunktionen "gut verteilt" sein



Suchen und Löschen

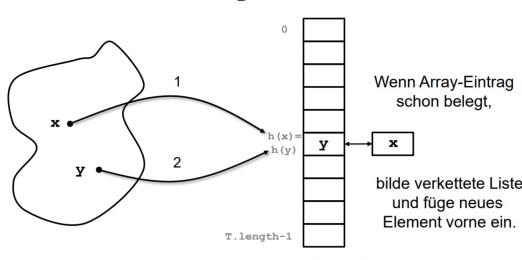
- Beim Suchen wird das gesuchte Element mit der Hashfunktion gemapped
- Befindet sich das Element in der Hashtable, ist es der zurückgegebenen Stell zu finden
- Beim Löschen wird das Element als erstes gesucht, und falls vorhanden gelöscht



Kollisionsauflösung

- Wird ein Element eingefügt kann an der Stelle schon ein Element sein (dies sollte durch eine gute Hashfunktion verhindert werden)
- Sollte es doch dazu kommen, dass mehrere Elemente von der Hashfunktionen die selbe Position zugewiesen bekommen, dann gibt es mehrere Arten damit umzugehen
- Ein Möglichkeit ist das Erstellen von LinkedLists an den Stellen der Hashtable

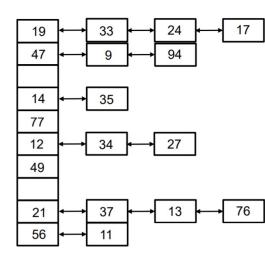
Einfügen



Es gibt weitere Arten der Kollisionsauflösung

LinkedLists

Einfügen immer noch konstante Anzahl Array-/Listen-Operation
Suchen/Löschen benötigen so viele Schritte, wie jeweilige Liste lang ist
Wenn Hashfunktion uniform verteilt, dann hat jede Liste im Erwartungswert $n/T.length$ viele Einträge



Hashtables mit Verketteten Listen: Laufzeit

- Bei uniform und unabhängig verteilten Hashwerten benötigten Suchen und Löschen im Durchschnitt $\Theta(\frac{n}{T.length})$ viele Schritte. Einfügen benötigt im Worst-Case $\Theta(1)$ viele Schritte.
- Wählt man $T.length \approx n$ ergibt sich also konstante Laufzeit (im Durchschnitt)

Hashfunktion

- Eine "gute" Hashfunktion verteilt alle Werte möglichst gleichmäßig auf den Indexbereich das Arrays ab
- Eine Möglichkeit ist die Universelle Hashfunktion (nebenstehend)
- Diese Funktion verteilt sehr gut, und jedes Element wird mit der Wahrscheinlichkeit $\frac{1}{p}$ getroffen
- Zwei Werte treffen mit einer Wahrscheinlichkeit von $\frac{1}{p^2}$ aufeinander
- Eine andere Möglichkeit sind Kryptographische Hashfunktionen wie MD5 und SHA-1 an (diese werden heute nicht mehr verwendet sondern ihre Nachfolger SHA-2, SHA-3)
- Dafür setzt man $h(x) = MD5(x) \bmod T.length$

Hashtables vs Bäume

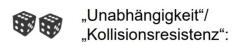
- In Hashtables lassen sich nur Suchen nach bestimmten Werten durchführen
- Um auf Nachbarwerte zuzugreifen, muss eine neue Hashfunktion berechnet werden
- Bei Bäumen oder anderen Datenstrukturen lassen sich schnell ganze Bereiche absuchen
- Des Weiteren muss die Hashtable immer größer sein als die Anzahl der Elemente (in Java empfohlen 0.75 loadfactor)

interpretiere (Binär-)Daten als
Zahlen zwischen 0 und $p - 1$,
 p prim, $p \gg T.length$
„Universelle“ Hash-Funktion:
wähle zufällige $a, b \in [0, p - 1]$, p prim, $a \neq 0$,
setze $h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod T.length$



Für alle $x, t \in [0, p - 1]$ gilt:
 $Pr_h[(a \cdot x + b) \bmod p = t] = \frac{1}{p}$

Zu gegebenen t, x, a
gibt es genau ein
 $b = (ax - t) \bmod p$ mit
 $h_{a,b}(x) = (ax + b) \bmod p = t$



Für alle $x \neq y, t_x, t_y \in [0, p - 1]$ gilt:
 $Pr_h[((ax + b) \bmod p = t_x) \neq ((ay + b) \bmod p = t_y)] = \frac{1}{p^2}$

(ohne Beweisidee)

6.13 Bloom-Filter

Eigenschaften

- Bloom-Filter sind ein weiteres Beispiel für Randomisierte Datenstrukturen
- "Speicherschonende Wörterbücher mit kleinem Fehler"

Anwendung Bloom-Filter

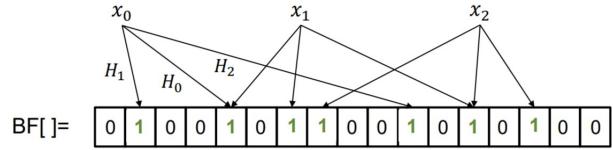
- NoSQL-Datenbanken: Abfrage für nicht-vorhandene Elemente verhindern
- Bitcoin: Prüfen von Transaktionen ohne gesamte Daten zu laden
- Früher auch Chrome-Browser: Erkennen schädlicher Webseiten

Bloom-Filter: Erstellen

- Gegeben:
 - n Elemente x_0, \dots, x_{n-1} beliebiger Komplexität
 - m Bits Speicher, üblicherweise in einem Bit-Array
 - k "gute" Hash-Funktionen H_0, \dots, H_{k-1} mit Bildbereich $0, 1, \dots, m-1$
- 1. Initialisiere Array mit 0-Einträgen
- 2. Schreibe für jedes Element in jede Bit-Position $H_0(x_j), \dots, H_{k-1}(x_j)$ eine 1
- Eventuell werden dabei Werte erneut auf 1 gesetzt

empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$
 ergibt Fehlerrate von ca. 2^{-k}
 üblicherweise $k = 5,6, \dots, 20$

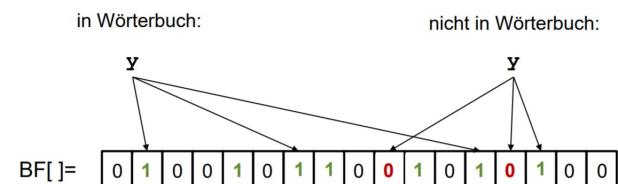
```
initBloom(X,BF,H) //H array of functions H[j]
1 FOR i=0 TO BF.length-1 DO BF[i]=0;
2 FOR i=0 TO X.length-1 DO
3   FOR j=0 TO H.length-1 DO
4     BF[H[j](X[i])]=1;
```



Suchen

- gib an, dass y im Wörterbuch, wenn genau alle k Einträge für y in $\text{BF}=1$ sind

```
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;
```



- Wenn y **nicht** im Wörterbuch, dann gibt Algorithmus evtl. trotzdem 1 zurück
- Passiert, wenn Einträge für y von anderen Werten getroffen wurden
- Daher "gute" Hashfunktion und Größe des Filters nicht zu klein

Baumstruktur
Speicherbedarf: 8.000.000 Bits (+Baumstruktur) Suchen: ca. $\log_2 100.000 \approx 17$ Elemente betrachten

Bloom-Filter
Speicherbedarf: ca. 1.000.000 Bits Suchen: $k = 7$ Mal Hashen und $k = 7$ Array-Zugriffe

7 Rot-Schwarz-Bäume

7.1 Allgemeines

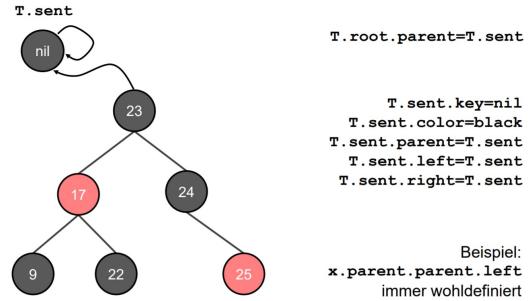
Anwendungen

- Als Anwendungen wird hier das Completely Fair Linux Scheduling
- Hierbei wird die Prozessorzeit auf die Anwendungen aufgeteilt
- Elemente werden oft gesucht, gelöscht und wieder eingefügt, daher ist eine schnelle Laufzeit vorteilhaft

RBT Bedingungen

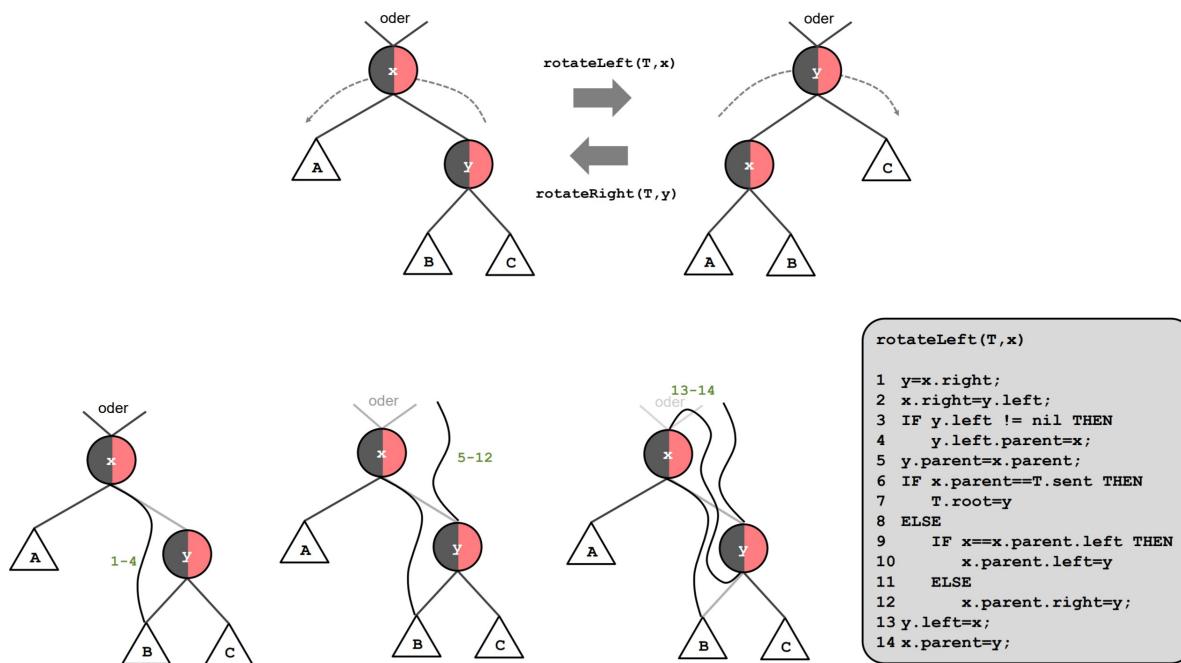
- 1. Jeder Knoten ist Rot oder Schwarz
- 2. Die Wurzel ist Schwarz
- 3. Wenn ein Knoten Rot ist, sind seine Kinder Schwarz (Nicht-Rot-Rot)
- 4. Für jeden Pfad im Teilbaum zu einem Blatt oder Halbblatt die gleiche Anzahl an Schwarzen Knoten (Schwarz-Höhe)
 - Halbblätter sind Schwarz, wenn nicht gäbe es im kinderlosen Pfad einen Schwarzen Knoten weniger

Implementierung mittels Sentinel zum Treaversieren



7.2 Einfügen

Rotation



Einfügen

```

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);

```

Einfügen: fixColorAfterInsertion

Schritt 1 (1: Farbe von Elternknoten):

- Prüfe ob Parent von z rot ist, wenn nicht sind wir fertig, da wir nur auf Roten Knoten operieren

Schritt 2 (2-3: Rechter oder Linker Teilbaum):

- Prüfen ob z in einem linken oder rechten Teilbaum ist. In beiden Fällen wird analog gehandelt
- $y = z.parent.parent.right$ (Onkel)

Schritt 3 (4-8: Wenn Onkel vorhanden und Onkel = rot):

- Siehe Fall 01
- „Äuflösen“ des Schwarzen Großvater-Knoten in seine beiden Kinder
- $z = z.parent.parent$

Schritt 4 (9-11: Einseitige Imbalance):

- Um eine Einseitige Imbalance zu bekommen wie in Fall 01, sind diese Schritte Notwendig
- Danach wird rotiert um die Imbalance wieder her zu stellen

Schritt 5 (13-15: Wenn nicht Fall 01):

- Ist nicht Fall 01 eingetreten, tritt Fall 02 ein
- $z.parent$ wird Schwarz eingefärbt, $z.parent.parent$ rot
- Danach wird einmal rotiert, siehe Fall 02

Code

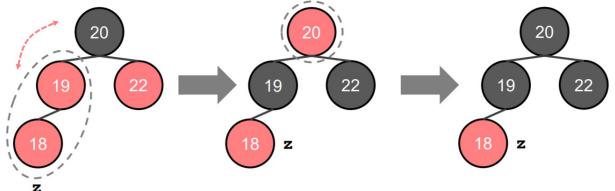
```
fixColorsAfterInsertion(T,z)
```

```

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;

```

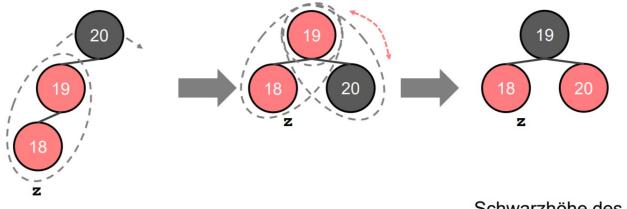
Fall 01:



Schwarzhöhe des Baumes bleibt erhalten
Achtung: Zweiter Schritt geht hier nur, weil 20 Wurzel ist

Wenn Teilbaum, dann stattdessen „rekursiv in Knoten 20 aufräumen“

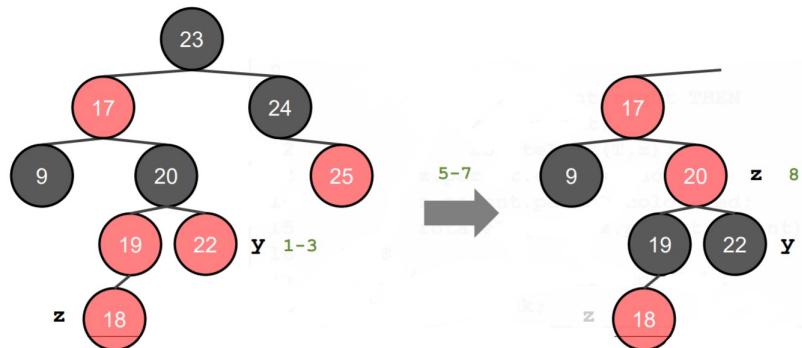
Fall 02:



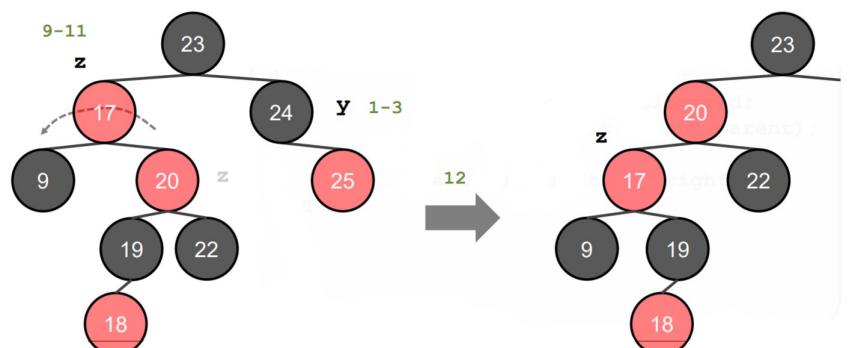
Schwarzhöhe des Baumes bleibt erhalten

Beispiel: fixColorAfterInsertion

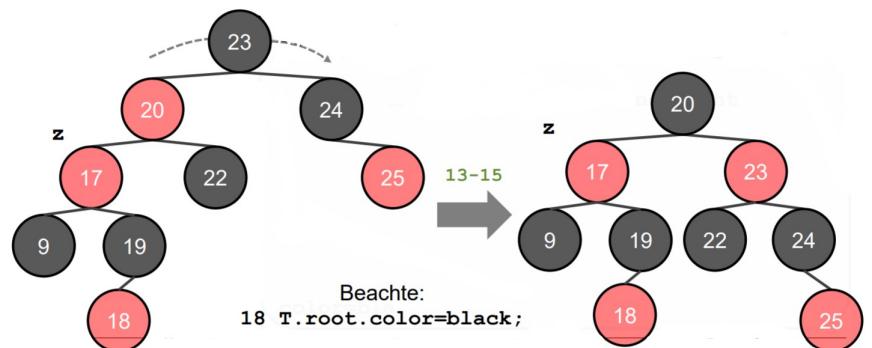
- Fall 01
- Es tritt Fall 01 ein, nach bearbeitung wird z verschoben



- Fixup für Einseitige Imbalance
- Da wir uns in einem Linken Teilbaum befinden, wollen wir eine Linksseitige Imbalance
- Daher wird einmal rotiert



- Fall 02
- Durch das rotieren befindnen wir uns nun in Fall 02, danach ist der Fixup komplett



7.3 Löschen

Allgemein

- Löschen Funktioniert wie bei BST
- Das zu löschende Element wird mit dem kleinsten Element des Rechten Teilbaums überschrieben
- Dies garantiert das Erhalten der BST Bedingung
- Das neue Element an der zu löschende Stelle wird die Farbe erbt dabei die Farbe des Alten Elements

Wann wird fixColorAfterDeletion

- Wenn der Knoten, der den gelöschten ersetzt *Rot* war, sind wir fertig
- Wenn der Knoten, der den gelöschten ersetzt *Schwarz* war, ist die Schwarzhöhe in seinem Baum zu niedrig

```

1 toBeDeleted = n
2 toReplaceDeleted = y
3 if (n.color == BLACK) {
4   if (y.color == RED) {
5     y.color = BLACK;
// y erbt die Farbe von n
6   } else {
7     DeleteCase01(y);
8   }
9 }
```

7.4 Löschen: fixColorAfterDeletion

Fall 01:

- n ist die neue Wurzel
- in diesem Fall sind wir Fertig, wenn nicht:
- DeleteCase02(n)

Bemerkung:

- Bei Fall 02, 04 und 05 nehmen wir an, dass n ein linkes Kind von a ist.

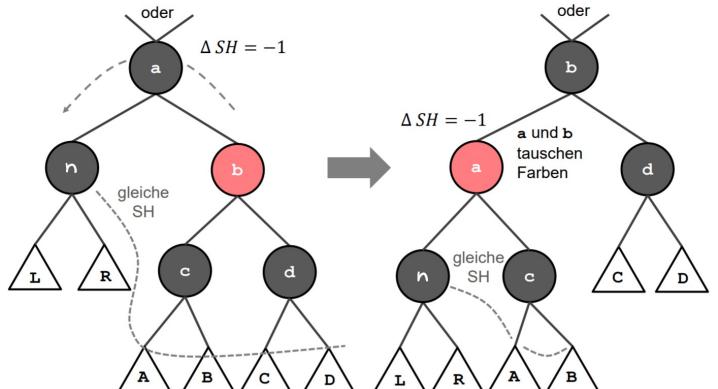
Durch Vertauschen von left und right kann man aber ein analoges Ergebniss erzielen

Fall 02:

- n ist Schwarz
- a (Parent) ist Schwarz
- b (Sibling) ist Rot

Dann \Rightarrow

- Linkssrotation um a
- Vertauschen der Farben von a und b
- Wird zu Fall 03a/b (DeleteCase03a/b(n))
- Wenn keine der beiden zutrifft zu Fall 04 (DeleteCase04(n))

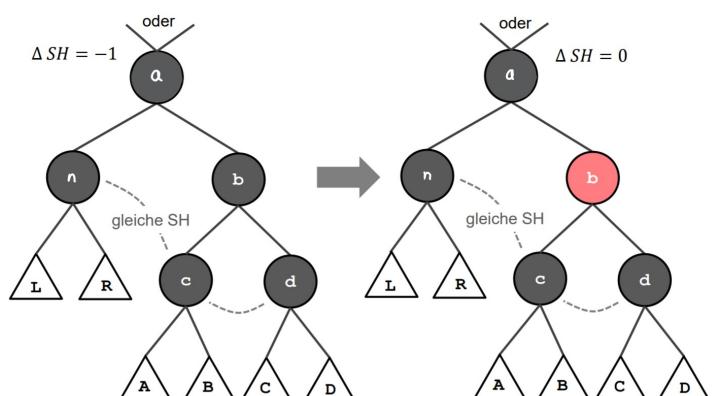


Fall 03a:

- n ist Schwarz
- a (Parent) ist Schwarz (Fall 03a)
- b (Sibling) ist Schwarz

Dann \Rightarrow

- In beiden Fällen (03a/b) wird a Schwarz und b Rot gefärbt
- Wird zu Fall 01 (DeleteCase01(n.parent))

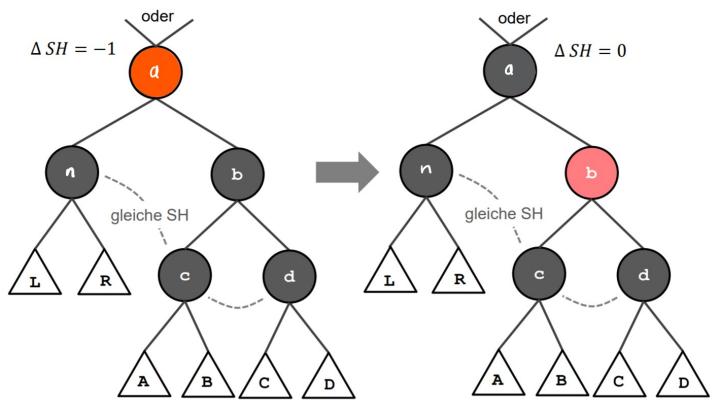


Fall 03b:

- n ist Schwarz
- a (Parent) ist Rot (Fall 03b)
- b (Sibling) ist Schwarz

Dann \Rightarrow

- In beiden Fällen (03a/b) wird a Schwarz und b Rot gefärbt
- Wenn der Fall eintritt sind wir fertig

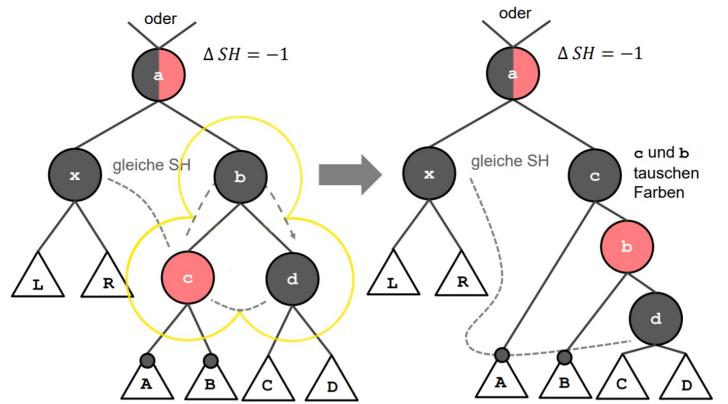


Fall 04:

- n ist Schwarz
- a (Parent) ist Rot/Schwarz
- b (Sibling) ist Schwarz
- c (b.left) ist Rot
- d (b.right) ist Schwarz

Dann \Rightarrow

- Rechtsrotation um b, dadurch kommt es zu Fall 05 (DeleteCase05(n))

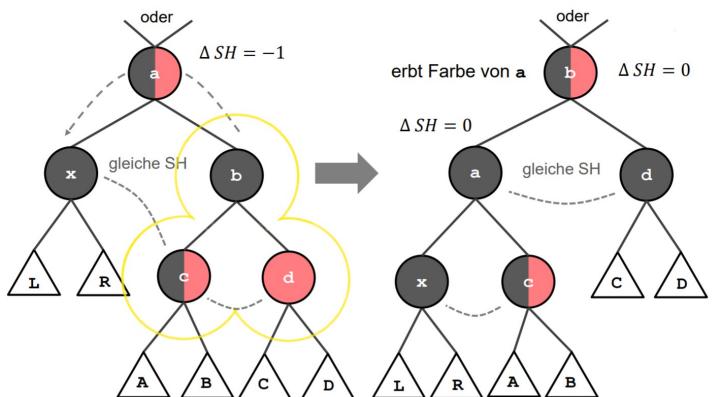


Fall 05:

- n ist Schwarz
- a (Parent) ist Rot/Schwarz
- b (Sibling) ist Schwarz
- c (b.left) ist Rot
- d (b.right) ist Schwarz

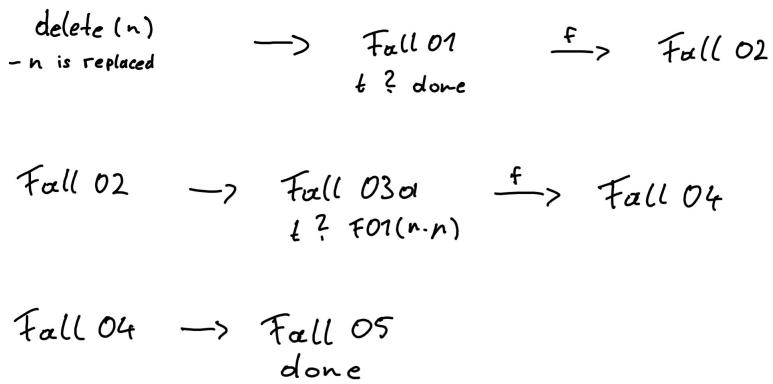
Dann \Rightarrow

- Linkssrotation um a
- a wird Schwarz und b erbt die Farbe von Schwarz



7.5 Löschen: Beispielgraph

Graph



Prozess

Fall 01:

- Zu Beginn wird immer geprüft, ob Fall 01 eintritt
- Fall 01 wird mit dem ersetzenen Knoten (kleinster Knoten im Rechten Teilbaum) aufgrufen
- Fall 01 := n ist die Wurzel
- (true): done
- (false): Fall 02(n)

Fall 02:

- Fall 02 := Sibling ist Rot
- (false/true): Fall 03a(n)

Fall 03a:

- Fall 03a := Parent ist Schwarz, Siblgn ist Schwarz
- (true): Fall 01(n.parent)
- (false): Fall 03b(n)

Fall 03b:

- Fall 03b := Parent ist Rot, Sibling ist Schwarz
- (true): done
- (false): Fall 04(n)

Fall 04:

- Fall 04 := Sibling ist Schwarz, Sibling hat ein Linkes Rotes Kind und ein Rechtes Schwarzes
- (true/false): Fall 05(n)

Fall 05:

- Fall 05 := Sibling ist Schwarz, Sibling hat ein Rechtes Rotes Kind, und ein Linkes R/S
- (true/false): done

8 Graphen

8.1 Allgemein

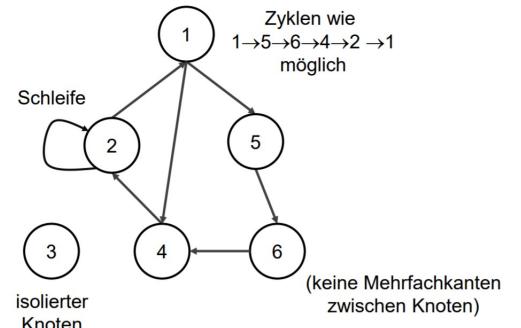
Endlich gerichtete Graphen

Ein (endlicher) gerichteter Graph $G = (V, E)$ besteht aus

- (1) einer (endlichen) Knotenmenge V ("vertices")
- (2) einer (endlichen) Kantenmenge $E \subseteq V \times V$ ("edges")
 $(u, v) \in E$: Kante von Knoten u zu v

Beispiel:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{(1, 4), (1, 5), (2, 1), (2, 2), (4, 2), (5, 6), (6, 4)\}$



Zusammenhängende Graphen

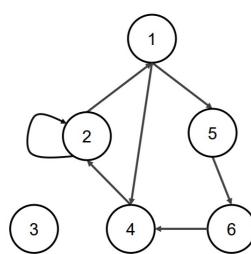
- Ungerichtete Graphen sind **zusammenhängend** wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist
- Gerichtete Graphen sind **stark zusammenhängend** wenn jeder Knoten von jedem anderen aus erreichbar ist

Subgraphen

Ein Graph $G' = (V', E')$ ist ein Subgraph des Graphen $G = (V, E)$ genau dann wenn $V' \subseteq V$ und $E' \subseteq E$

Adjazenzmatrix

- Adjazenzmatrizen sind eine Möglichkeit Kanten in einem Graph anzugeben
- Der Eintrag $a_{i,j}^{(m)}$ an der i -ten Zeile und j -ten Spalte der m -ten Potenz A^m der Adjazenzmatrix A eines Graphen gibt die Anzahl der Wege an, die von Knoten i zu Knoten j entlang von genau m Kanten führen ($m \geq 0$)
- Dabei ist A^m gegeben durch: $A^{m+1} = A^m \times A$



$$A[i,j] = \begin{cases} 1 & \text{wenn Kante von } i \text{ zu } j \\ 0 & \text{wenn keine Kante} \end{cases}$$

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

bei ungerichteten Graphen ist Matrix (spiegel-)symmetrisch zur Hauptdiagonalen

9 Exkurs

9.1 Totale Ordnung

Eine binäre Relation, \leq , auf der Menge M bildet eine Total Ordnung genau dann wenn M

- Reflexiv, $x \leq x$ für alle x,
- Transitiv, $x \leq y$ und $y \leq z$ impliziert $x \leq z$,
- Antisymmetrisch ist, $x \leq y$ und $y \leq x$ impliziert $x = y$