Algorithmen und Datenstrukturen

René Thiemann

Institute of Computer Science
University of Innsbruck

SS 2010

Übersicht

- Organisation
- Einführung
- Analyse von Algorithmen
- Suchen und Sortieren
- Bäume
- Hashverfahren
- Optimierungs-Probleme
- Graphen

Übersicht

- Organisation

Organisation

LVA 703003 VO 3

cl-informatik.uibk.ac.at/teaching/ss10/ad/

VO	Mittwochs	13:15 – 14:00	HSB 1	(RT)
	Freitags	12:15 – 13:45	HS D	
PS	Dienstags	10:15 – 11:45	RR 20	(TV, englisch)
	Dienstags	12:15 – 13:45	RR 20	(WP)
	Dienstags	12:15 – 13:45	RR 22	(RT)
	Dienstags	16:15 – 17:45	RR 20	(HS)

online Registrierung für PS – erforderlich bis zum 19. 3. 2010 um 8:00 Uhr

		Sprechstunden
Wolfgang Pausch	3S09	nach Vereinbarung
Heiko Studt	2W05	nach Vereinbarung
René Thiemann	3N01	Montags 13:00 – 15:00
Tomas Vitvar	3N07	nach Vereinbarung

Literatur

- T. Ottmann und P. Widmayer
  Algorithmen und Datenstrukturen
  Spektrum Verlag
- T. H. Cormen, C. E. Leiserson und R. L. Rivest
  Introduction to Algorithms
  MIT Press

Organisation – Vorlesung

- Vorlesungsfolien werden auf der Homepage vor der jeweiligen Vorlesung online gestellt

$\Rightarrow$ Folien zur Vorlesung mitbringen, um darin Notizen zu machen

- Ende des Semesters: schriftliche Prüfung

Organisation – Proseminar

- Jeden Mittwoch gibt es ein neues Übungsblatt auf der Webseite
- Übungsblätter sollen in 2er oder 3er Gruppen bearbeitet werden
- Lösungen werden am folgenden Dienstag im Proseminar eingesammelt
- Programmieraufgaben müssen in Java gelöst werden und bis Montag, 10 Uhr an den Proseminar Leiter geschickt werden
- Anwesenheitspflicht in Proseminaren (2x Abwesenheit geduldet)
- 2 Tests während des Semesters (keine Gruppenarbeit, keine Computer)
- Teilnahme am 1. Test $\Rightarrow$ Note wird eingetragen
- Note: $\frac{1}{3}\cdot$ 1. Test + $\frac{1}{3}\cdot$ 2. Test + $\frac{1}{3}\cdot$ Übungsblätter und Vorrechnen
- 50 % der Punkte notwendig zum Bestehen

Übersicht

- Einführung
  - Die Wahl des Algorithmus
  - Die Wahl der Datenstruktur
  - Überblick der Vorlesung

Eigenschaften eines guten Algorithmus

- korrekt (diese Vorlesung + Programmverifikation)
- effizient (diese Vorlesung)
  - Zeit
  - Platz
- wartbar (Entwurf von Softwaresystemen)

Übersicht

- Einführung
  - Die Wahl des Algorithmus
  - Die Wahl der Datenstruktur
  - Überblick der Vorlesung

Treppensteigen

- Ausgangslage: Man kann jeweils 1 oder 2 Stufen heruntergehen.
- Fragestellung: Wieviele Möglichkeiten gibt es, eine Treppe mit n Stufen herunterzusteigen?
- Bsp: $ts(\text{Kölner Dom}) = ts(533) = 177793095480941658714766079178479414784432111526800706093789579403138960940165075820050317562202766948028237512$
- Lösung: $ts(n) = \begin{cases} 1 & \text{wenn } n \leqslant 1 \\ ts(n-1) + ts(n-2) & \text{sonst} \end{cases}$
- In Java:

```java
static BigInteger ts(int n) {
    if (n <= 1) {
        return BigInteger.ONE;
    } else {
        return ts(n-1).add(ts(n-2));
    }
```

- Problem: exponentielle Laufzeit (Addition von 1en)

Verbesserter Algorithmus

```java
static BigInteger ts_fast(int n) {
    BigInteger tmp, ts_i, ts_i_min_1;
    int i = 1;
    ts_i = BigInteger.ONE;
    ts_i_min_1 = BigInteger.ONE;
    while (n > i) {
        i++;
        tmp = ts_i;
        ts_i = ts_i.add(ts_i_min_1);
        ts_i_min_1 = tmp;
    }
    return ts_i;
}
```

⇒ $\approx 5n + 4$ Ops zur Berechnung von $ts(n)$

- Frage: Wie kommt man (systematisch) auf verbesserten Algorithmus?

⇒ Algorithmen-Entwurf

Übersicht

- Einführung
  - Die Wahl des Algorithmus
  - Die Wahl der Datenstruktur
  - Überblick der Vorlesung

Algorithmen basieren immer auf Datenstrukturen

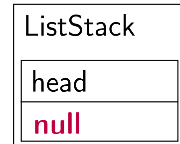
Beispiel (Keller / Stapel / Stack)

- zwei primitive Operationen:
  - **void** push(T elem) – lege elem oben auf den Stapel
  - T pop() – liefere und entferne oberstes Element des Stapels
- zwei mögliche Implementierungen
  - basierend auf verketteten Listen
  - basierend auf Arrays
- gemeinsame Schnittstelle (hier: nur Zahlen)

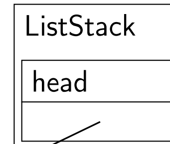
```
public interface Stack {
   void push(int n);
   int pop();
}
```

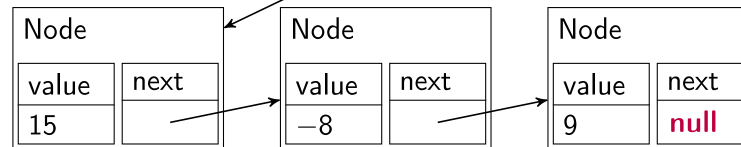
Keller mit verketteten Listen

- oberster Wert am Anfang der Liste
- für jeden Wert ein Listen-Element
- leerer Keller



Keller mit Werten 15, -8 und 9





Keller mit verketteten Listen

```
public class ListStack implements Stack {
   Node head = null; // head of stack
   public void push(int n) {
      this.head = new Node(n, this.head);
   }
   public int pop() {
      if (this.head == null) {
         throw new RuntimeException("no element");
      } else {
         int n = this.head.value;
         this.head = this.head.next;
         return n;
      }
```

```
class Node {
   int value; Node next;
   public Node(int value, Node next) {
      this.value = value;
      this.next = next;
```

kurze Einführung in Java

- Klasse = Attribute (analog zu C-struct) + Methoden

```
public class ListStack {
   Node head = null; // Attribut (initialisiert)
   public void push(int n) { ... } // Methode
}
```

- Zugriff auf Attribute und Methoden mittels ., this ist eigenes Objekt

```
this.head = this.head.next;
```

- Erzeugung von Objekten mit new

```
new int[4]; new ListStack(); new Node(n,elem);
```

- Konstruktoren werden bei Erzeugung aufgerufen

```
public Node(int value, Node next) { this.value = ... }
```

- Default-Konstruktor, wenn kein Konstruktor in Klasse

```
public ListStack() { }
```


kurze Einführung in Java

- Interfaces = Liste von erforderlichen Methoden

```
public interface Stack { public void push(int n); ... }
```

- Klasse implementiert Interface, wenn alle geforderten Methoden verfügbar sind

```
public class ListStack implements Stack {
   public void push(int n) { ... }
}
```

- Vorteil: Programme leicht wartbar, Änderung nur bei Konstruktor-Aufruf

```
Stack s = new ListStack(); s.push(5); ...
```

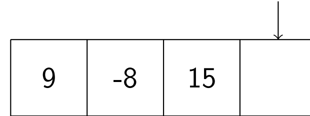
```
Stack s = new ArrayStack(); s.push(5); ...
```

- Fehlerbehandlung mittels Exceptions

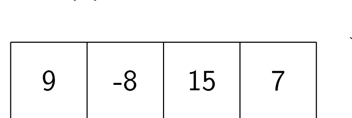
```
if (...) { throw new RuntimeException(message); ... }
```

Keller mit Arrays

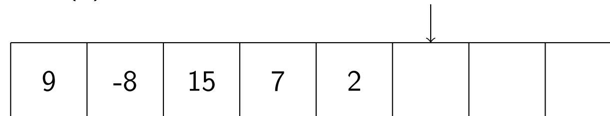
- oberster Wert am Ende des Arrays
- Index zeigt auf nächsten freien Platz
- Vergrößerung der Array-Größe bei Bedarf
- Keller mit Werten 15, -8 und 9



- push(7)



- push(2)



Keller mit Arrays

```
public class ArrayStack implements Stack {
  int[] arr = new int[1];
  int size = 0; // nr of elements
  public void push(int n) {
    if (this.size == this.arr.length) { // too small
      int[] large = new int[this.arr.length * 2];
      System.arraycopy(this.arr,0,large,0,this.size);
      this.arr = large;
    }
    this.arr[size] = n;
    size++;
  }
  public int pop() {
    if (this.size == 0) {
      throw new RuntimeException("no element");
    } else {
      this.size--;
      return this.arr[size];
    }
  }
}
```

Vergleich von Kellerimplementierungen

Angenommen, es sind bereits n Elemente im Keller.

	Liste	Array
Speicherplatz	n Knoten n Referenzen + n Zahlen	Array der Größe $\approx 1.5n$ $\approx 1.5n$ Zahlen
pop()	4 Ops	3 Ops
push()	4 Ops	$\leqslant 4 + 3n$ Ops

⇒ Listen-Implementierung Platz-sparender?

- Nein, Speicherplatz für Knoten zu teuer

⇒ Listen-Implementierung effizienter?

- Nein, da Array-Kopie nur selten benötigt
  (aber wie zeigt man so etwas?)

Übersicht

- Einführung
  - Die Wahl des Algorithmus
  - Die Wahl der Datenstruktur
  - Überblick der Vorlesung

Behandelte Themen

Analyse von Algorithmen

- $\mathcal{O}$-notation
- Rekursionsgleichungen
- amortisierte Analyse

Entwurf von Algorithmen

- Divide & Conquer
- Dynamische Programmierung

Klassische Algorithmen & Datenstrukturen

- Suchen
- Sortieren
- Bäume
- Hashtabellen
- Graphen

Übersicht

- Analyse von Algorithmen
  - $\mathcal{O}$-notation
  - Rekursionsgleichungen

Übersicht

- Analyse von Algorithmen
  - $\mathcal{O}$-notation
  - Rekursionsgleichungen

Laufzeit von Algorithmen

Definition

Sei A ein Algorithmus. Dann ist $T_A : \mathbb{N} \to \mathbb{N}$ (oder oft nur T) die Funktion, so dass $T(n)$ die maximale Laufzeit von A auf Eingaben der Größe n ist.
(Die Laufzeit wird in Anzahl der atomaren Operationen angegeben)

Anmerkung:

- Neben maximaler Laufzeit gibt es auch minimale oder durchschnittliche Laufzeit
- Analyse durchschnittlicher Laufzeit oft kompliziert
  Problem: Wie sieht durchschnittliche Eingabe gegebener Größe aus?

$\Rightarrow$ Diese Vorlesung: fast immer maximale Laufzeit

Beispiel von Laufzeit von Algorithmen

```
int n = arr.length;
for (int i=0; i < n; i++) {
    if (arr[i] == x) {
        return i;
    }
}
return -1;
```

- $T(n) = 3 + 3n$
- minimale Laufzeit: 5
- durchschnittliche Laufzeit: ?
  (wie wahrscheinlich ist es, dass x in arr vorkommt?)

Beispiel von Laufzeit von Algorithmen

```
int n = arr.length;
int last = arr[n-1];
arr[n-1] = x;
int i = 0;
while (true) {
    if (arr[i] == x) {
        break;
    }
    i++;
}
arr[n-1] = last;
return (i < n-1 ? i : (x == last ? n - 1 : -1));
```

- $T(n) = 6 + 2n$ oder $T(n) = 8 + 2n$ oder ...?
- Problem: was ist atomare Operation?
- Lösung: ignoriere konstante Faktoren, nicht relevant

Asymptotische Laufzeit

Motivation

- Genaue Laufzeit oft schwer zu berechnen
- asymptotische Laufzeit oft präzise genug

Wie verhält sich $T(n)$ bei wachsendem n?

linear? logarithmisch? quadratisch? exponentiell?

$\mathcal{O}$-Notation

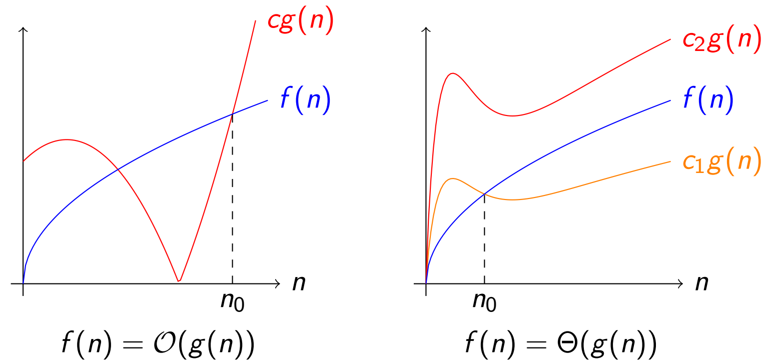
Definition

Sei g eine Funktion.

- $\mathcal{O}(g) = \{f \mid \exists n_0, c > 0 : \forall n \geqslant n_0 : f(n) \leqslant c\, g(n)\}$
  Menge aller Funktionen, die höchstens so schnell wie g wachsen
- $\Omega(g) = \{f \mid \exists n_0, c > 0 : \forall n \geqslant n_0 : c\, g(n) \leqslant f(n)\}$
  Menge aller Funktionen, die mindestens so schnell wie g wachsen
- $\Theta(g) = \{f \mid \exists n_0, c_1, c_2 > 0 : \forall n \geqslant n_0 : c_1\, g(n) \leqslant f(n) \leqslant c_2\, g(n)\}$
  Menge aller Funktionen, die genauso schnell wie g wachsen

Anstelle von $f \in \mathcal{O}(g)$ schreibt man auch $f = \mathcal{O}(g)$.

Asymptotische Laufzeit



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

Beispiel:

- $1.5\sqrt{n} = \mathcal{O}(abs(n^2 - n - 2))$, denn für $c = 0.7$ und $n_0 = 4$ gilt:

$$\forall n \geqslant n_0 : 1.5\sqrt{n} \leqslant c \cdot (abs(n^2 - n - 2))$$

- $1.5\sqrt{n} = \Theta(\sqrt{n} + 0.1n \cdot exp(4.5 - 3n))$

Komplexitätsklassen

	Bezeichnung
$\mathcal{O}(1)$	konstant
$\mathcal{O}(log(log(n)))$	doppelt logarithmisch
$\mathcal{O}(log(n))$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \cdot log(n))$	
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(n^k)$	polynomiell
$2^{\mathcal{O}(n)}$	exponentiell

Rechnen mit asymptotischen Laufzeiten

- Ω und Θ können durch $\mathcal{O}$ ausgedrückt werden:
  - $f(n) = \Omega(g(n))$ gdw. $g(n) = \mathcal{O}(f(n))$
  - $f(n) = \Theta(g(n))$ gdw. $f(n) = \mathcal{O}(g(n))$ und $g(n) = \mathcal{O}(f(n))$
- Transitivität und Reflexivität:
  - $f(n) = \mathcal{O}(g(n))$ und $g(n) = \mathcal{O}(h(n))$ impliziert $f(n) = \mathcal{O}(h(n))$
  - $f(n) = \mathcal{O}(f(n))$
- Konstante Faktoren und kleine Funktionen spielen keine Rolle
  - $\mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n))$ für alle $c > 0$
  - $f(n) = \mathcal{O}(g(n))$ impliziert $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(g(n))$
- Beispiel:

$$\begin{aligned} 5n^3 + 7n^2 - 3n + 5 &= \mathcal{O}(5n^3 + 7n^2 - 3n + 5) \\ &= \mathcal{O}(5n^3) \\ &= \mathcal{O}(n^3) \end{aligned}$$

Rechnen mit asymptotischen Laufzeiten

- Ausdrücke mit $\mathcal{O}$ auf der rechten Seite

$$T(n) = n^2 + \mathcal{O}(n)$$

  Bedeutung: es gibt eine Funktion $f(n)$, so dass $f(n) = \mathcal{O}(n)$ und $T(n) = n^2 + f(n)$

- Ausdrücke mit $\mathcal{O}$ auf der linken Seite

$$n^2 + \mathcal{O}(n) = \mathcal{O}(n^2)$$

  Bedeutung: für jede Funktion $f(n)$ mit $f(n) = \mathcal{O}(n)$ gilt $n^2 + f(n) = \mathcal{O}(n^2)$

Asymptotische Laufzeitanalyse

```
1   int n = arr.length;
2   int last = arr[n-1];
3   arr[n-1] = x;
4   int i = 0;
5   while (true) {
6       if (arr[i] == x) {
7           break;
8       }
9       i++;
10  }
11  arr[n-1] = last;
12  return (i < n-1 ? i : (x == last ? n - 1 : -1));
```

$$T(n) = \underbrace{\mathcal{O}(1)}_{1-4} + \underbrace{n \cdot \underbrace{\mathcal{O}(1)}_{6-9}}_{5-10} + \underbrace{\mathcal{O}(1)}_{11-12} = \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$$

Übersicht

- Analyse von Algorithmen
  - $\mathcal{O}$-notation
  - Rekursionsgleichungen

Divide & Conquer

D & C bezeichnet ein Entwurfsmuster für Algorithmen

1. Löse kleine Probleme direkt
2. Teile grosses Probleme in mindestens 2 Teilprobleme auf (Divide)
3. Löse Teilprobleme rekursiv
4. Füge aus Teillösungen Gesamtlösung zusammen (Conquer)

Beispiele

- Merge-Sort
- Matrix Multiplikation

Merge-Sort

1. Wenn (Teil-)Array kleiner als 1, mache nichts

1	7	10	4	12	8	2

2. Ansonsten halbiere Array

1	7	10	4

12	8	2

3. Sortiere beide Hälften rekursiv

1	4	7	10

2	8	12

4. Mische beide sortieren Arrays zu sortiertem Gesamtarray

1	2	4	7	8	10	12

Matrix Multiplikation

Aufgabe: Berechne Produkt $C = AB$ zweier $n \times n$-Matrizen A und B

1. Falls $n = 1$, dann ist $A = (a)$, $B = (b)$ und $C = (a \cdot b)$.
2. Falls $n > 1$, dann ist $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$ und $B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$, wobei jedes A_i und B_i eine $\frac{n}{2} \times \frac{n}{2}$-Matrix ist.
3. Berechne rekursiv A_1B_1, A_1B_2, A_2B_3, A_2B_4, A_3B_1, A_3B_2, A_4B_3 und A_4B_4
4. $C = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$

Laufzeit:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{falls } n = 1 \\ 8 \cdot T(\frac{n}{2}) + \Theta(n^2) & \text{sonst} \end{cases}$$

Laufzeit von D & C Algorithmen

Laufzeit von D & C Algorithmen oft durch Rekursionsgleichung gegeben.

- Merge-Sort

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

- Matrix Multiplikation

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{falls } n = 1 \\ 8 \cdot T(\frac{n}{2}) + \Theta(n^2) & \text{sonst} \end{cases}$$

Problem: finde geschlossene Form für Rekursionsgleichungen

- Merge-Sort: $T(n) = \mathcal{O}(n \cdot log(n))$
- Matrix Multiplikation: $T(n) = \mathcal{O}(n^3)$

Lösen von Rekursionsgleichungen: Raten + Induktion

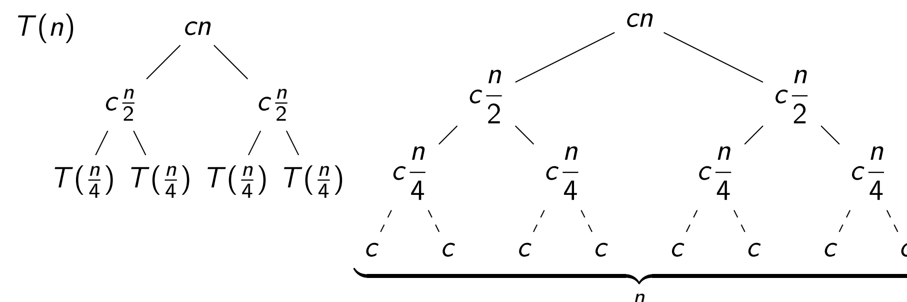
$$T(n) = \begin{cases} \mathcal{O}(1) & \text{falls } n = 1 \\ 2 \cdot T(\frac{n}{2}) + \Theta(n) & \text{sonst} \end{cases}$$

Um geschlossene Form für $T(n)$ zu erhalten, führe 2 Schritte durch:

1. Rate geschlossene Form $f(n)$
2. Führe einen Induktionsbeweis, dass $T(n) \leqslant f(n)$.

Raten mittels Rekursionsbaum

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



- Summe in jeder Ebene: cn
- Anzahl Ebenen: $log(n)$

$\Rightarrow$ Rate $T(n) = \mathcal{O}(n \cdot log(n))$

Induktionsbeweis für $T(n) = \mathcal{O}(n \cdot log(n))$

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

- Ziel: Finde geeignete Konstanten n_0, d und zeige für alle $n \geqslant n_0$:

$$T(n) \leqslant d(n \cdot log(n))$$

- Vereinfachte Annahme: $n = 2^k$ für $k \in \mathbb{N}$

$\Rightarrow$ Zeige $T(2^k) \leqslant d \cdot 2^k \cdot k$ für alle $k \geqslant k_0$ mittels Induktion

- Induktionsschritt:

$$\begin{aligned} T(2^{k+1}) &= 2 \cdot T(2^k) + c \cdot 2^{k+1} \\ \text{(IH)} &\leqslant 2 \cdot (d \cdot 2^k \cdot k) + c \cdot 2^{k+1} \\ &= d \cdot 2^{k+1} \cdot k + c \cdot 2^{k+1} \\ &= d \cdot 2^{k+1} \cdot (k+1) - d \cdot 2^{k+1} + c \cdot 2^{k+1} \\ \text{(wähle } d \geqslant c) &\leqslant d \cdot 2^{k+1} \cdot (k+1) \end{aligned}$$

Induktionsanfang

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

- Ziel: Finde geeignete Konstanten k_0, d und zeige für alle $k \geqslant k_0$:

$$T(2^k) \leqslant d \cdot 2^k \cdot k$$

- Induktionsanfang ($k = 0$): $T(2^0) = c \not\leqslant 0 = d \cdot 2^0 \cdot 0$
- Ausweg: Ungleichung nur ab k_0 verlangt. Wähle also z.B. $k_0 = 1$

$\Rightarrow$ Induktionsanfang ($k = 1$):

$$\begin{aligned} T(2^1) &= 2 \cdot T(1) + 2c \\ &= 4c \\ \text{(wähle } d \geqslant 2c) &\leqslant 2d \\ &= d \cdot 2^1 \cdot 1 \end{aligned}$$

$\Rightarrow$ Erhalte $T(2^k) \leqslant 2c \cdot 2^k \cdot k$ für alle $k \geqslant 1$, also $T(n) = \mathcal{O}(n \cdot log(n))$

Lösen von Rekursionsgleichungen: Master Theorem

Beobachtungen:

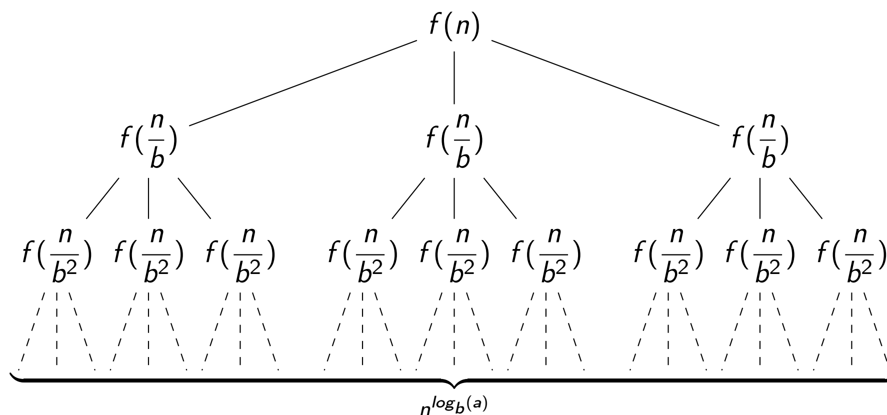
- Raten + Induktion aufwendig
- $T(1)$ ist immer $\mathcal{O}(1)$

⇒ betrachte nur rekursive Gleichung, aber im allgemeinen Fall:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

- Tiefe des Rekursionsbaums: $log_b(n)$
- Breite der Ebenen im Rekursionsbaums: $1, a, a^2, \ldots, a^{log_b(n)} = n^{log_b(a)}$
- Betrachte 3 Fälle:
  - $f(n)$ kleiner als $n^{log_b(a)}$
  - $f(n) = \Theta(n^{log_b(a)})$
  - $f(n)$ grösser als $n^{log_b(a)}$

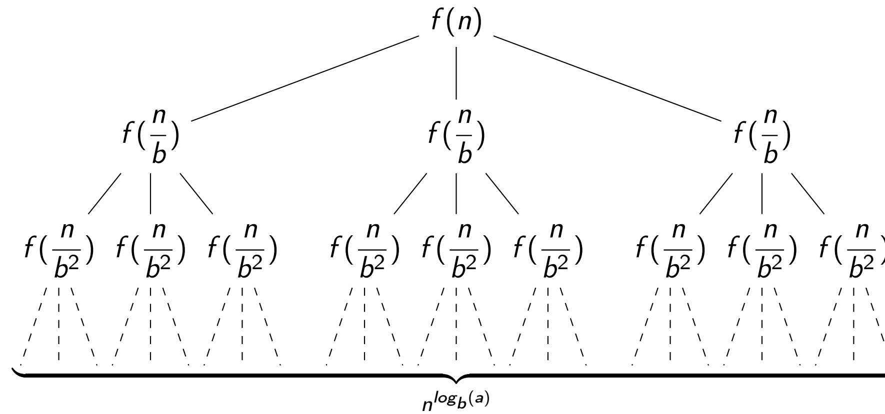
Lösen von $T(n) = aT(\frac{n}{b}) + f(n)$



⇒ Erhalte $T(n)$ durch Summation jeder Ebene:

$$T(n) = \Theta(n^{log_b(a)}) + \sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i})$$

Lösen von $T(n) = aT(\frac{n}{b}) + f(n)$

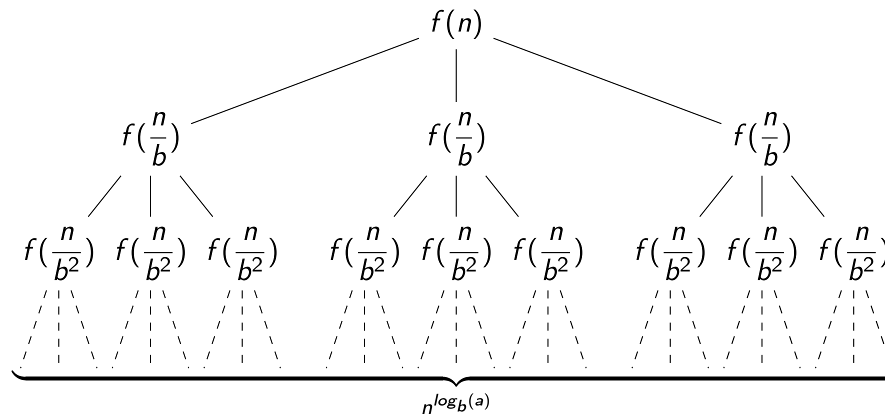


1. Fall: $f(n)$ kleiner als $n^{log_b(a)}$

$\Rightarrow$ $\sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i}) = \mathcal{O}(n^{log_b(a)})$

$\Rightarrow$ $T(n) = \Theta(n^{log_b(a)})$

Lösen von $T(n) = aT(\frac{n}{b}) + f(n)$

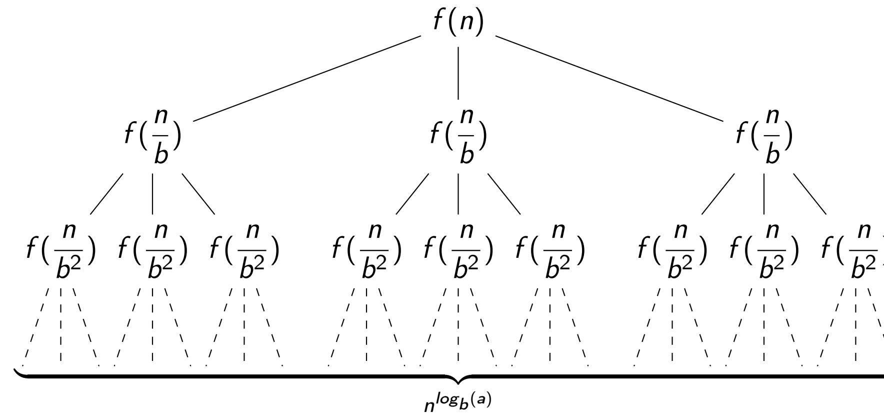


2. Fall: $f(n) = \Theta(n^{log_b(a)})$

$\Rightarrow$ $a^i f(\frac{n}{b^i}) \approx a^i \cdot (\frac{n}{b^i})^{log_b(a)} = n^{log_b(a)}$ (Summe jeder Ebene $n^{log_b(a)}$)

$\Rightarrow$ $T(n) = \Theta(n^{log_b(a)} \cdot log(n))$

Lösen von $T(n) = aT(\frac{n}{b}) + f(n)$



3. Fall: $f(n)$ größer als $n^{log_b(a)}$

$\Rightarrow$ $\Theta(n^{log_b(a)}) + \sum_{i=0}^{log_b(n)-1} a^i f(\frac{n}{b^i}) = \mathcal{O}(f(n)) + \mathcal{O}(f(n)) = \mathcal{O}(f(n))$

$\Rightarrow$ $T(n) = \Theta(f(n))$

Master Theorem

Theorem

Seien $a \geqslant 1$ und $b > 1$ Konstanten. Sei $\epsilon > 0$. Sei

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

1. *Wenn $f(n) = \mathcal{O}(n^{log_b(a)-\epsilon})$, dann $T(n) = \Theta(n^{log_b(a)})$*
2. *Wenn $f(n) = \Theta(n^{log_b(a)})$, dann $T(n) = \Theta(n^{log_b(a)} \cdot log(n))$*
3. *Wenn $f(n) = \Omega(n^{log_b(a)+\epsilon})$, dann $T(n) = \Theta(f(n))$, falls es zudem $c < 1$ und n_0 gibt, so dass $a \cdot f(\frac{n}{b}) \leqslant c \cdot f(n)$ für alle $n \geqslant n_0$*

Anwendung des Master Theorems

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

1. Wenn $f(n) = \mathcal{O}(n^{log_b(a)-\epsilon})$, dann $T(n) = \Theta(n^{log_b(a)})$
2. Wenn $f(n) = \Theta(n^{log_b(a)})$, dann $T(n) = \Theta(n^{log_b(a)} \cdot log(n))$
3. Wenn $f(n) = \Omega(n^{log_b(a)+\epsilon})$, dann $T(n) = \Theta(f(n))$, falls es zudem $c < 1$ und n_0 gibt, so dass $a \cdot f(\frac{n}{b}) \leqslant c \cdot f(n)$ für alle $n \geqslant n_0$

- Merge-Sort: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$
  $\Rightarrow$ $a = b = 2$, $log_b(a) = 1$, $\Theta(n) = \Theta(n^{log_b(a)})$.
  $\Rightarrow$ $T(n) = \Theta(n^{log_b(a)} \cdot log(n)) = \Theta(n \cdot log(n))$ (Fall 2)
- Matrix Multiplikation: $T(n) = 8 \cdot T(\frac{n}{2}) + \Theta(n^2)$
  $\Rightarrow$ $a = 8$, $b = 2$, $log_b(a) = 3$, $\Theta(n^2) = \mathcal{O}(n^{3-1}) = \mathcal{O}(n^{log_b(a)-\epsilon})$ für $\epsilon = 1$
  $\Rightarrow$ $T(n) = \Theta(n^{log_b(a)}) = \Theta(n^3)$ (Fall 1)

Zusammenfassung

- Vereinfachung der Laufzeitabschätzung: ignoriere Konstanten
  - $\mathcal{O}$-Notation
  - asymptotische Analyse
- Divide & Conquer Algorithmen
- Lösen von Rekursionsgleichungen
  - Raten + Induktion
  - Master-Theorem

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Motivation

Warum suchen?

- Fundamentale Fragestellung
  - Suche Datensatz zu Schlüssel
    (Anschrift von Kunde, Preis von Produkt, Tel.-Nr. von Oma)
  - Suche von Texten die Wörter enthalten
    (Bücher, in denen "Algorithmus" vorkommt)
  - Suche Beweis zu Theorem

Warum sortieren?

- Suche in unsortierten Feldern: $\mathcal{O}(n)$
- Suche in sortierten Feldern: $\mathcal{O}(log(n))$
- Präsentation von Ergebnissen
  (Medaillenspiegel, ...)

Was suchen und sortieren?

- Oft: Datensätze mit Schlüssel
```
class Entry { int telNr; String name; ... }
```
  - Sortiere nach Telefonnummer oder nach Name
  - Suche nach Eintrag zu gegebener Nummer oder Name
  - gewünschte Methoden
    - `void sort(Entry[] a)`
    - `int index(Entry[] a, int nr)`
    - `int index(Entry[] a, String name)`
- Hier: Datensatz = Schlüssel, Schlüssel = Zahl

  ⇒ gewünschte Methoden
    - `void sort(int[] a)`
    - `int index(int[] a, int x)`

  ⇒ einfachere Präsentation der Algorithmen
  - vorgestellte Algorithmen einfach auf komplexe Datensätze erweiterbar

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Suchverfahren - Unsortiere Arrays

- lineare Suche (Folien 27 & 28), $\mathcal{O}(n)$
- optimal: angenommen, es werden weniger als n Elemente betrachtet,

$\Rightarrow$ es gibt ein Element x, das nicht betrachtet wurde

$\Rightarrow$ x könnte das gesuchte Element gewesen sein

$\Rightarrow$ Verfahren inkorrekt

Suchverfahren - Binäre Suche

- Voraussetzung: sortiertes Array

```
int left = 0;
int right = a.length - 1;
while (left <= right) {
  int middle = (left + 1) / 2 + right/2;
  if (x < a[middle]) {
    right = middle - 1;
  } else if (x == a[middle]) {
    return middle;
  } else {
    left = middle + 1;
  }
}
return -1;
```

- jeder Schleifendurchlauf: Halbierung von right $-$ left $\Rightarrow \mathcal{O}(log(n))$

Suchverfahren - Interpolationssuche

- Voraussetzung: sortiertes Array
- Beispiel Telefonbuch mit n Einträgen
  - Suche nach "Thiemann"
  - ⇒ binäre Suche wählt $a[\frac{1}{2} \cdot n] \approx$ "*Moser*" zum Vergleich
  - Mensch (+ Interpolations-Suche) sind geschickter: vergleiche mit $a[\frac{4}{5} \cdot n] \approx$ "Sternagel" oder "Winkler"
  - ⇒ nutze erwartete Distanz um schneller ans Ziel zu kommen
- falls Schlüssel Zahlen, vergleiche mit Element an Position

$$\text{left} + \frac{\text{x} - \text{a[left]}}{\text{a[right]} - \text{a[left]}} \cdot (\text{right} - \text{left})$$

- Bsp.: left $= 5$, right $= 100$, a[left] $= 20$, a[right] $= 1500$, x $= 99$

$$5 + \frac{99 - 20}{1500 - 20} \cdot (100 - 5) = 5 + \frac{79}{1480} \cdot 95 = 5 + 0.053 \cdot 95 = 10$$

Suchverfahren - Interpolationssuche

- falls Werte im Array gleichverteilt:

  durchschnittliche Laufzeit von $\mathcal{O}(log(log(n)))$
- doppelt-logarithmische Laufzeit erst bei großen Eingaben relevant (konstanter Faktor höher als bei binärer Suche)
- worst-case: $\mathcal{O}(n)$

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Fundamentale Eigenschaft von Sortierverfahren

- Komplexität (zwischen $\mathcal{O}(n)$ und $\mathcal{O}(n^2)$)
- insitu / inplace (zusätzlicher Speicherbedarf von Elementen konstant)
- stabil (relative Anordnung wird beibehalten bei gleichen Schlüsseln)
- Verhalten auf vorsortierten Daten
- sequentiell (benötigt nur sequentiellen Zugriff auf Daten, kein Random-Access)
- intern oder extern
  - intern: gesamtes Feld muss in Hauptspeicher passen
  - extern: nur Teile der Feldes im Hauptspeicher

Elementare Sortierverfahren

Beispiel

- Bubble-Sort (sortieren durch Vertauschen von Nachbarn)
- Insertion-Sort (sortieren durch iteriertes sortiertes Einfügen)
- Minimum/Maximum-Sort (sortieren durch Auswahl)

Gemeinsame Eigenschaften

- insitu
- sequentiell
- stabil
- worst-case Komplexität: $\mathcal{O}(n^2)$

Insertion Sort

- Idee: Füge schrittweise alle Elemente von vorne nach hinten sortiert ein
- Sortiertes einfügen: Fange hinten an und verschiebe kleine Elemente, um Platz zu schaffen

Insertion Sort

```
static void insertionSort(int[] a) {
    for (int i=1; i<a.length; i++) {
        int ai = a[i];
        int j = i;
        while (j > 0) {
            if (a[j-1] > ai) {
                a[j] = a[j-1];
            } else {
                break;
            }
            j--;
        }
        a[j] = ai;
    }
}
```


Unterschiede

	# Vergleiche		# a[i] = ...		$T(n)$	
	best	worst	best	worst	best	worst
Bubble-Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion-Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Minimum-Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

Anmerkungen

- alle obigen Verfahren haben worst-case Komplexität von $\mathcal{O}(n^2)$

$\Rightarrow$ nutze bessere Verfahren
(elementare Verfahren leicht zu implementieren $\Rightarrow$ werden verwendet)

- Beispiel Merge-Sort: $\mathcal{O}(n\ log(n))$, stabil, nicht insitu

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Heap-Sort Eigenschaften

- Komplexität: $\Theta(n\,log(n))$ (best- und worst-case)
- insitu
- nicht stabil
- nicht sequentiell

Heaps

Definition

Array $a[0]\ldots a[n-1]$ ist ein Heap gdw.

$$\forall 0 \leqslant i < n : a[i] \geqslant a[2i+1] \wedge a[i] \geqslant a[2i+2]$$

(falls $2i+1$ und $2i+2$ größer als n, wird nichts verlangt)

Beispiel

kein Heap:

0	7	2	8	5	-3	10	8	1

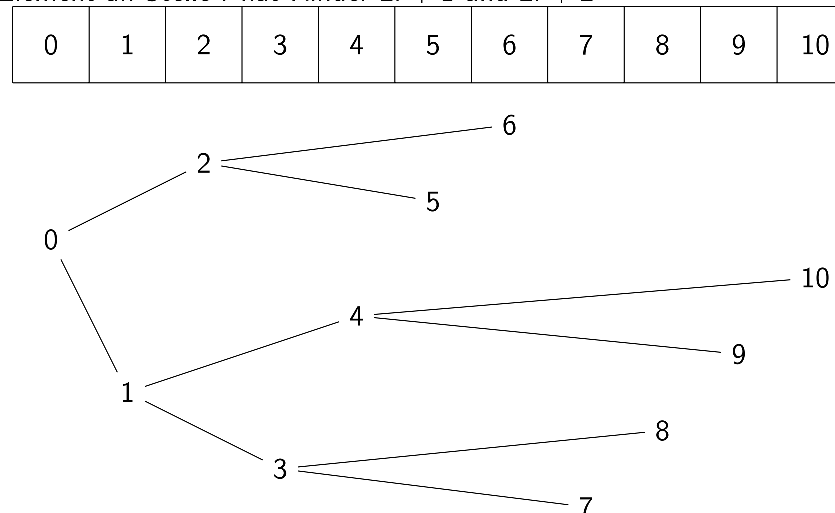
Heap:

10	8	2	8	5	-3	0	7	1

Heaps als Binär-Bäume

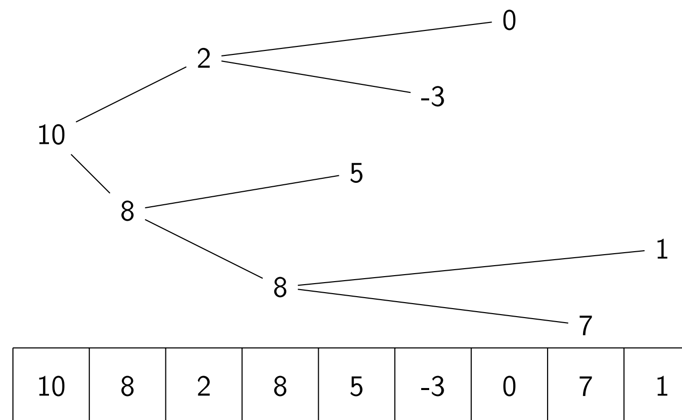
- Fasse jedes Array-Element als Knoten auf
- Element an Stelle i hat Kinder $2i+1$ und $2i+2$

0	1	2	3	4	5	6	7	8	9	10



Heaps als Binär-Bäume

- Array ist Heap gdw. jeder Knoten größer als beide Kinder



10	8	2	8	5	-3	0	7	1

Heap-Sort

- Beobachtung: in Heap ist größtes Element in $a[0]$
- Heap-Sort:
  1. stelle aus Array Heap her
  2. für alle $i = n - 1 \dots 1$:
     - vertausche $a[0]$ mit $a[i]$
     - stelle Heap her (bzgl. um 1 verkürzten Array)
- Wesentliche Operation: `downHeap(int a[], int i, int n)`
  - Annahme: $a[i + 1] \dots a[n - 1]$ erfüllt Heap-Eigenschaft
  - Nachher: $a[i] \dots a[n - 1]$ erfüllt Heap-Eigenschaft

Heap-Sort

- Beobachtung: in Heap ist größtes Element in $a[0]$
- Heap-Sort:
  1. stelle aus Array Heap her
  2. für alle $i = n - 1 \dots 1$:
     - vertausche $a[0]$ mit $a[i]$
     - stelle Heap her (bzgl. um 1 verkürzten Array)
- Wesentliche Operation: `downHeap(int a[], int i, int n)`
  - Annahme: $a[i + 1] \dots a[n - 1]$ erfüllt Heap-Eigenschaft
  - Nachher: $a[i] \dots a[n - 1]$ erfüllt Heap-Eigenschaft
  - Algorithmus:
    falls an Position i Eigenschaft verletzt,
    vertausche $a[i]$ mit $max(a[2i + 1], a[2i + 2])$
    und fahre an entsprechender Position fort

Heap-Sort

- Heap-Sort:
  1. stelle aus Array Heap her:
     für alle $i = \frac{n}{2} - 1 \dots 0$:
     `downHeap(a, i, n);`
  2. für alle $i = n - 1 \dots 1$:
     - vertausche $a[0]$ mit $a[i]$
     - stelle Heap her: `downHeap(a, 0, i);`
- Wesentliche Operation: `downHeap(int a[], int i, int n)`
  - Annahme: $a[i + 1] \dots a[n - 1]$ erfüllt Heap-Eigenschaft
  - Nachher: $a[i] \dots a[n - 1]$ erfüllt Heap-Eigenschaft

Beispiel

downHeap im Detail

```
static void downHeap(int[] a, int i, int n) {
  while (true) {
    int l = 2*i + 1;
    int r = 2*i + 2;
    int max = l < n && a[l] > a[i] ? l : i;
    max = r < n && a[r] > a[max] ? r : max;
    if (max == i) {
      break;
    } else {
      swap(a, max, i);
      i = max;
    }
  }
}
```

Beweise von Programmen mit Schleifen

```
before;
while (cond) {
    loop;
}
```

Um Eigenschaft φ zu beweisen, nutze Schleifeninvariante ψ

- Zeige, dass ψ bei Eintritt der Schleife gilt:
  nach Abarbeitung von before gilt ψ
- Zeige, dass bei jedem Durchlauf der Schleife ψ erhalten bleibt:
  wenn ψ und cond gelten, dann gilt ψ auch nach Ausführung von loop

$\Rightarrow$ nach Beendigung der Schleife gilt ψ
(und auch $\neg$ cond, falls Schleife nicht durch **break** beendet)

$\Rightarrow$ zeige, dass aus ψ und Abbruchbedingung Eigenschaft φ folgt

Korrektheit von downHeap

- Beobachtung: Heap-Eigenschaft für $a[i] \ldots a[n-1]$ erfüllt gdw.
  - Heap-Eigenschaft im linken (unteren) Teilbaum von $a[i]$ erfüllt
  - Heap-Eigenschaft im rechten (oberen) Teilbaum von $a[i]$ erfüllt
  - Heap-Eigenschaft für $a[i]$ erfüllt
- Annahme: $a[i+1] \ldots a[n-1]$ erfüllt Heap-Eigenschaft
- Zeige nach `downHeap(int a[], int i, int n)`: $a[i] \ldots a[n-1]$ erfüllt Heap-Eigenschaft
- Zeige dazu Schleifeninvariante:
  nur $a[i]$ kann gegen Heap-Eigenschaft verstoßen
  - Invariante beim Start von downHeap wegen Annahme erfüllt
  - $a[2i+1] = max(a[i], a[2i+1], a[2i+2])$
    - $\Rightarrow$ downHeap tauscht $a[i]$ mit $a[2i+1]$
    - $\Rightarrow$ Heap-Eigenschaft an Position i, einzige mögliche Verletzung an $2i+1$
    - downHeap setzt i auf $2i+1 \Rightarrow$ Invariante erfüllt
  - $a[2i+2] = max(\ldots)$: analog
- Beendung von downHeap nur falls $a[i] = max(a[i], a[2i+1], a[2i+2])$

$\Rightarrow$ Mit Schleifeninvariante Heap-Eigenschaft erfüllt

Korrektheit von Heap-Sort

1. Erstellung des Heaps:
   für alle $i = \frac{n}{2} - 1 \dots 0$: `downHeap(a, i, n);`
2. für alle $i = n - 1 \dots 1$: ...

1. Erstellung des Heaps:
   - Invariante: $a[i+1] \dots a[n-1]$ ist Heap
   - zu Beginn: $a[\frac{n}{2}] \dots a[n-1]$ ist Heap, da keine Nachfolger
   - Schleifendurchlauf: folgt aus Korrektheit von downHeap und Invariante

Korrektheit von Heap-Sort

1. Erstellung des Heaps
2. für alle $i = n - 1 \dots 1$:
   `swap(a,0,i); downHeap(a, 0, i);`

2. Abbau des Heaps:
   - Invariante:
     - 2.a $a[0] \dots a[i]$ ist Heap
     - 2.b $a[i+1] \dots a[n-1]$ sortiert
     - 2.c Heap-Elemente kleiner als sortierte Elemente: $a[0], \dots, a[i] \leqslant a[i+1]$
   - Start:
     - 2.a: $a[0] \dots a[n-1]$ ist Heap wegen Korrektheit von 1.
     - 2.b und 2.c trivial erfüllt
   - Schleifendurchlauf:
     - 2.b erfüllt, da zuvor 2.b und 2.c
     - 2.c erfüllt, da zuvor 2.c und $a[0]$ grösstes Element von Heap $a[0] \dots a[i]$
     - 2.a erfüllt, da nur $a[0]$ Heap-Eigenschaft verletzt + downHeap korrekt
   - Nach Ende von Heapsort: $i = 0$, wegen 2.b und 2.c Array sortiert

Komplexität von Heap-Sort

- Position i auf Level ℓ ($lvl(i) = \ell$) gdw. Abstand von i zur Wurzel $= \ell$
- Beobachtungen:
  - Array der Länge n hat $k = log(n)$ als höchstes Level
  - Level ℓ hat maximal 2^ℓ Elemente
  - Laufzeit von `downheap(a, i, n)` : $\approx k - lvl(i) = lvl(n) - lvl(i)$
- Folgerungen:

$$\text{Laufzeit Erstellung} \approx \sum_{i=0}^{\frac{n}{2}} k - lvl(i) \leqslant \sum_{\ell=0}^{k} 2^\ell (k - \ell)$$

$$= \sum_{\ell=0}^{k} 2^{k-\ell} \ell = 2^k \sum_{\ell=0}^{k} \frac{\ell}{2^\ell} \leqslant n \sum_{\ell=0}^{\infty} \frac{\ell}{2^\ell} = 2n$$

$$\text{Laufzeit 2. Phase} \approx \sum_{i=1}^{n-1} lvl(i) - lvl(0) \leqslant \sum_{i=1}^{n} log(n) = n\,log(n)$$

$\Rightarrow$ Heap-Sort hat eine Komplexität von $\mathcal{O}(n\,log(n))$

Zusammenfassung Heap-Sort

- Worst-case Komplexität von $\mathcal{O}(n\,log(n))$ (average-case auch $\mathcal{O}(n\,log(n))$)
- insitu (im Gegensatz zu Merge-Sort)
- kein Ausnutzen von Vorsortierung (Erweiterung: Smooth-Sort $\mathcal{O}(n)$)
- nicht stabil (im Gegensatz zu Merge-Sort)
- nicht sequentiell (im Gegensatz zu Merge-Sort)

$\Rightarrow$ Caches werden nicht (gut) ausgenutzt

$\Rightarrow$ es gibt Verfahren, die sich in der Praxis besser verhalten

$\Rightarrow$ Quick-Sort

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Quick-Sort

- D & C Algorithmus
- insitu, nicht stabil, weitestgehend sequentiell
- best- und average-case: $\mathcal{O}(n\,log(n))$, worst-case: $\mathcal{O}(n^2)$
- Idee:
  - um Bereich $a[l] \dots a[r]$ mit $l < r$ zu sortieren, wähle Pivot-Element v
  - partitioniere $a[l] \dots a[r]$ so, dass
    - linker Bereich: Elemente $\leqslant v$
    - rechter Bereich: Elemente $\geqslant v$
    - dazwischen: Element v
    - bei Bedarf, vertausche Elemente aus linkem und rechten Bereich
  - sortiere linken Bereich und rechten Bereich rekursiv

```
1   static void quickSort(int[] a, int l, int r) {
2     if (l < r) {
3       int p = pivot(a,l,r); // l <= p <= r
4       int v = a[p];
5       int i = l - 1;
6       int j = r;
7       swap(a,p,r);
8       while (true) {
9         do i++; while (a[i] < v);
10        do j--; while (a[j] > v);
11        if (i >= j) {
12          break;
13        }
14        swap(a,i,j);
15      }
16      swap(a,r,i);
17      quickSort(a,l,i-1);
18      quickSort(a,i+1,r);
19    }
20  }
```


Beispiel (Pivot = rechtes Element)

Analyse

Fehlerquellen:

```
7          swap(a,p,r);
8          while (true) {
9             do i++; while (a[i] < v);
10            do j--; while (a[j] > v);
```

- gefährlicher Zugriff auf $a[i]$ und $a[j]$ in Zeilen 9 und 10 (Indizes nicht beschränkt)
- Zeile 9 unkritisch, da spätestens für $i = r$ Schleifenabbruch
- Zeile 10 kritisch, falls Pivot kleinstes Element ist

⇒ Vorverarbeitung erforderlich:

```
static void quickSort(int[] a) {
   if (a.length > 0) swap(a, 0, minimum(a));
   quickSort(a, 1, a.length - 1);
}
```


Laufzeit

- $T(0) = T(1) = 1$
- $T(n) = T(links) + T(rechts) + \Theta(n)$

Spezialfälle:

- Partitionierung spaltet konstanten Bereich ab: $|links| \leqslant c$ oder $|rechts| \leqslant c$

⇒ $T(n) \geqslant T(n - c - 1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$

⇒ falls Pivot Element immer grösstes/kleinstes Element: $\Theta(n^2)$

- Partitionierung in 2 Hälften:

⇒ $T(n) \leqslant 2\,T(\frac{n}{2}) + \Theta(n) \Rightarrow T(n) = n\,log(n)$

Folgerung: worst-case $\Theta(n^2)$, best-case: $\Theta(n\,log(n))$
(Insertion-Sort: worst-case $\Theta(n^2)$, best-case: $\Theta(n)$)

⇒ Quick-Sort schlechter als Insertion-Sort?

- Nein: average-case Quick-Sort: $\Theta(n\,log(n))$, Insertion-Sort: $\Theta(n^2)$

Beispiel = mittleres Element

Average-Case Quick-Sort

Annahmen über $a[1] \dots a[n]$:

- Werte unterschiedlich $\Rightarrow$ $n!$ verschiedene Permutationen
- Array "zufällig": jede Permutation hat gleiche Wahrscheinlichkeit

Beobachtungen:

- Pivotelement k.-kleinstes Element:
  $T(n) = T(k-1) + T(n-k) + \Theta(n)$
- Array zufällig $\Rightarrow$ gleiche Wahrscheinlichkeit für jedes $1 \leqslant k \leqslant n$
- Partitionierung von zufälligen Arrays liefert wieder zufällige Arrays

$\Rightarrow$ durchschnittliche Laufzeit:

$$T(n) = \frac{1}{n} \cdot \sum_{k=1}^{n} (T(k-1) + T(n-k)) + \Theta(n) = \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n)$$

(unter Benutzung von $T(0) = 0$)

Average-Case Quick-Sort

Erhalte b so dass für alle $n \geqslant 2$ gilt:

$$T(n) \leqslant \frac{2}{n} \sum_{k=1}^{n-1} T(k) + bn$$

Zeige $T(n) \leqslant cn\, log(n)$ für $n \geqslant 2$ per Induktion für hinreichend großes c

- $n \leqslant 2$ einfach durch Wahl von c beweisbar
- $n > 2$: o.B.d.A. n gerade (Fall n ungerade analog)

$$\begin{aligned}
T(n) &\leqslant \frac{2}{n} \sum_{k=1}^{n-1} T(k) + bn \\
\text{(ind.)} \quad &\leqslant \frac{2c}{n} \sum_{k=1}^{n-1} k\, log(k) + bn \\
&\leqslant \frac{2c}{n} \left(\sum_{k=1}^{\frac{n}{2}} k\, log(k) + \sum_{k=1}^{\frac{n}{2}-1} (\frac{n}{2} + k)\, log(\frac{n}{2} + k) \right) + bn
\end{aligned}$$

Average-Case Quick-Sort

$$\begin{aligned}
T(n) \leqslant \ldots &\leqslant \frac{2c}{n} \left(\sum_{k=1}^{\frac{n}{2}} k\, log(k) + \sum_{k=1}^{\frac{n}{2}-1} (\frac{n}{2} + k)\, log(\frac{n}{2} + k) \right) + bn \\
&\leqslant \frac{2c}{n} \left(\sum_{k=1}^{\frac{n}{2}} k\, (log(n) - 1) + \sum_{k=1}^{\frac{n}{2}-1} (\frac{n}{2} + k)\, log(n) \right) + bn \\
&= \frac{2c}{n} \left(\frac{n^2 + 2n}{8} (log(n) - 1) + \frac{3n^2 - 6n}{8} log(n) \right) + bn \\
&= cn\, log(n) - c\, log(n) - \frac{cn}{4} - \frac{c}{2} + bn \\
(n \geqslant 2) \quad &\leqslant cn\, log(n) - \frac{cn}{4} + bn \\
(\text{wähle } c \geqslant 4b) \quad &\leqslant cn\, log(n)
\end{aligned}$$

Wahl des Pivot Elements

- immer rechter Rand
  - Vorteil: Zeile 7 kann entfernt werden
  - Nachteil: $\Theta(n^2)$ auf sortierten Arrays
- mittleres Element $(\frac{l+r}{2})$
  - Vorteil: $\mathcal{O}(n\,log(n))$ auf sortierten Arrays
- Median-of-three: Median von $a[l], a[\frac{l+r}{2}], a[r]$
  - Vorteil: $\mathcal{O}(n\,log(n))$ auf sortierten Arrays, wahrscheinlich gute Partition
  - Nachteil: Mehraufwand, um Pivot Element zu berechnen

  (für alle obigen Verfahren gibt es Eingaben mit quadratischer Laufzeit)
- Median von $a[l], \dots, a[r]$
  - Vorteil: perfekte Partitionierung
  - Nachteil: naives Verfahren für Median-Berechnung: $\mathcal{O}(n\,log(n))$
    $\mathcal{O}(n)$-Verfahren aufwendig, hohe Konstante
- Zufälliges Element aus $a[l], \dots, a[r]$
  - Vorteil: für jede Eingabe erwartete Laufzeit von $\Theta(n\,log(n))$

Zusammenfassung

- Quick-Sort ist effizienter Sortieralgorithmus
- parametrisch in Wahl der Pivot-Funktion
- deterministisch meistens: average-case $\Theta(n\,log(n))$, worst-case $\Theta(n^2)$
- Median: worst-case $\Theta(n\,log(n))$, aufwendig
- zufällig: worst-case $\Theta(n\,log(n))$ erwartete Laufzeit
- in der Praxis: schneller als Heap-Sort
- insitu, nicht stabil, fast sequentiell (Partitionierung: ja, Rekursion: nein)

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Motivation

- Bislang: Sortierverfahren basieren nur auf Schlüsselvergleich

⇒ diese Verfahren können im worst-case nicht besser als $\Theta(n\,log(n))$ sein!

- Alternative: nutze Struktur der Schlüssel

⇒ lineare Sortierverfahren

Bucket-Sort

- Annahme: Menge der Schlüssel endlich, # Schlüssel = m
- Idee:
  1. nutze m Eimer, die zu Beginn leer sind
  2. iteriere über Array, werfe Wert mit Schlüssel i in entsprechenden Eimer
  3. iteriere über Eimer von kleinen zu großen Schlüsseln und fülle Array von links nach rechts
- Komplexität: $\Theta(n + m)$
- Problem: Größe der Eimer
  - pessimistische Annahme: m Eimer der Größe $n \Rightarrow$ hoher Platzbedarf
  - dynamisch wachsende Eimer (Stacks, Queues) $\Rightarrow$ geringerer Platzbedarf
  - statt Eimer nur Größe von Eimer $\Rightarrow$ noch geringerer Platzbedarf

Bucket-Sort

Eingabe: Array a der Länge n

1. nutze Array cnt_ind der Größe m, initialisiert mit 0en
2. iteriere über a
   für Wert a[i] mit Schlüssel k inkrementiere cnt_ind[k]
   (cnt_ind enthält nun Größe der Eimer)
3. iteriere über cnt_ind und speichere jetzt Start-Indizes
   (cnt_ind[k] = i, cnt_int[k+1] = j
   $\Rightarrow$ Elemente mit Schlüssel k müssen an Positionen i,...,j−1)
4. iteriere über a und schreibe Elemente direkt an passende Position

Komplexität:

- Laufzeit: Schritt 1 und 3: $\mathcal{O}(m)$, Schritt 2 und 4: $\mathcal{O}(n)$
- Speicher: Schritt 1: m, Schritt 4: n (zweites Array erforderlich)

Beispiel (n = 12, m = 8)

Bucket-Sort

Implementierung:

```
void bucketSort(int[] a, int[] b, int mask, int shift)
```

- sortiere a, speichere in b
- mask $= 2^k - 1$
- Schlüssel für Zahl x: (x >>> shift) & mask
  (entferne shift Bits von rechts, Schlüssel = die rechten k Bits)

Eigenschaften:

- $\Theta(n + mask)$
- stabil, nicht insitu

```
static void bucketSort(int[] a,int[] b,int mask,int shift) {
   int[] cnt_ind = new int[mask+1];
   int n = a.length;
   // count
   for (int i=0; i<n; i++) {
      int key = (a[i] >>> shift) & mask;
      cnt_ind[key]++;
   }
   // compute start-indices
   cnt_ind[mask] = n - cnt_ind[mask];
   for (int j=mask-1; j >= 0; j--) {
      cnt_ind[j] = cnt_ind[j+1] - cnt_ind[j];
   }
   // sort
   for (int i=0; i<n; i++) {
      int key = (a[i] >>> shift) & mask;
      int index = cnt_ind[key];
      b[index] = a[i];
      cnt_ind[key] = index+1;
   }
}
```


Radix-Sort

Idee:

- wiederholtes Bucket-Sort
- sortiere erst nach letzter Ziffer,
- dann nach vorletzter Ziffer,
- ...
- und zum Schluss nach erster Ziffer

$\Rightarrow$ $\Theta(\#\text{Stellen} \cdot (n + \#\text{Ziffern}))$

- Korrektheit benötigt insitu-Implementierung von Bucket-Sort

Beispiel (16-bit Zahlen mit 4-bit Ziffern)

Übersicht

- Suchen und Sortieren
  - Suchen
  - Sortierverfahren
  - Heap-Sort
  - Quick-Sort
  - Bucket- und Radix-Sort
  - Quick-Select

Suchen von kleinsten Elementen

Aufgabe: bestimme i.-kleinstes Element in Array a
äquivalent: bestimme a[i], nachdem a sortiert wurde

- Beispiel
  - 0.-kleinstes Element von a = kleinstes Element von a
  - 2.-kleinstes Element von [7,2,5,10000,8] = 7
- Algorithmen
  - sortiere a, liefere a[i] $\Rightarrow \mathcal{O}(n \cdot log(n))$
  - bestimme $i+1$ mal das Minimum $\Rightarrow \Theta((i+1) \cdot n)$
    (oder, wenn $i > \frac{n}{2}$: bestimme $n - i$ mal das Maximum)
  - $\Rightarrow$ $\mathcal{O}(n^2)$, falls i nicht am Rand
  - Quick-Select: $\mathcal{O}(n)$

Quick-Select

```
void quickSelect(int[] a, int l, int r, int x)
```

- Annahme: l $\leqslant$ x $\leqslant$ r
- Ziel: vertausche Elemente von a[l].. a[r] so, dass nachher an Position x das Element steht, was auch an Position x bei Sortierung von a[l].. a[r] stehen würde
- Idee: analog Quick-Sort
  - falls l $\geqslant$ r ist nichts zu tun
  - sonst wähle Pivot-Element v = a[p] mit l $\leqslant$ p $\leqslant$ r
  - Partioniere a[l].. a[r] in linken und rechten Teil
  - $\Rightarrow$ erhalte Position i für Pivot-Element
  - falls i == x ist nichts mehr zu tun
  - falls x < i, quickSelect(a, l, i−1, x)
  - falls x > i, quickSelect(a, i+1, r, x)
  - $\Rightarrow$ jedes Mal nur ein rekursiver Aufruf im Gegensatz zu Quick-Sort

$\Rightarrow$ Komplexität: $\mathcal{O}(n^2)$ bei ungünstiger Wahl des Pivot-Elements

```
static void quickSelect(int[] a, int l, int r, int x) {
    while (l < r) {
        int p = pivotSelect(a,l,r);
        int v = a[p]; // Partition: von hier bis ...
        int i = l - 1;
        int j = r;
        swap(a,p,r);
        while (true) {
            do i++; while (a[i] < v);
            do j--; while (a[j] > v);
            if (i >= j) {
                break;
            }
            swap(a,i,j);
        }
        swap(a,r,i); // ... hier gleich zu quickSort
        if (i > x) {
            r = i-1;
        } else if (i == x) {
            return;
        } else {
            l = i+1;
```


Stopper benötigt (wie bei Quick-Sort)

```
static void quickSelect(int[] a, int x) {
    if (x < 0 || x >= a.length) {
        throw new RuntimeException("no x-th element");
    }
    swap(a, 0, minimum(a));
    if (x > 0) {
        quickSelect(a, 1, a.length-1, x);
    }
}
```

- Nach Ausführung von quickSelect(a,x) gilt:

$$a[x] = sort(a)[x]$$

Verbesserung der Pivot-Funktion

- Idee: nutze Quick-Select um Median von a[l].. a[r] zu bestimmen

⇒ quickSelect (a, l, r, x) führt zum Aufruf von
quickSelect (a, l, r, (l+1)/2 + r/2)

⇒ Nicht-Terminierung, Veränderung der Idee notwendig

- Neue Idee: nutze Quick-Select für ungefähr gleichmäßige Aufteilung (so dass nicht immer nur konstanter Teil abgespalten wird)

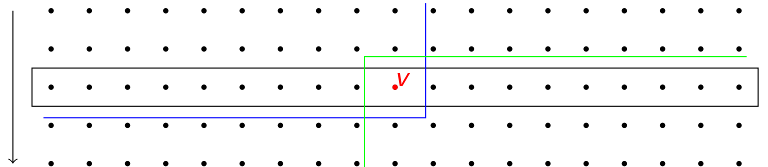
$$|\text{links}| \geqslant c \cdot (r - l) \qquad \text{und} \qquad |\text{rechts}| \geqslant c \cdot (r - l) \qquad \text{für } 0 < c \leqslant \frac{1}{2}$$

Ungefähr gleichmäßige Aufteilung: Median von Medianen

- Angenommen, a[l] .. a[r] hat n Elemente, $m = \lfloor \frac{n}{5} \rfloor$
- Unterteile a[l] .. a[r] in m Teil-Arrays der Größe 5 (und ignoriere verbleibende $n - 5 \cdot m \leqslant 4$ Elemente)
- Sortiere jedes der m Teil-Arrays in konstanter Zeit ($\mathcal{O}(n)$)

⇒ Mediane der Teil-Arrays in $\mathcal{O}(1)$ verfügbar

- Bestimme den Median v der m Mediane mittels Quick-Select
- Nehme diesen Wert als Pivot-Element



⇒ $|\text{links}| \geqslant 3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n}{5} \rfloor \rceil - 1 \geqslant \frac{3}{10} \cdot n - 3$ und $|\text{rechts}| \geqslant \frac{3}{10} \cdot n - 3$

$3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n}{5} \rfloor \rceil - 1 \geqslant \frac{3}{10} \cdot n - 3$

Induktion über n mit 10er Schritten:

- Basisfälle:

n	0	1	2	3	4
$3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n}{5} \rfloor \rceil - 1$	-1	-1	-1	-1	-1
$\frac{3}{10} \cdot n - 3$	-3	-2.7	-2.4	-2.1	-1.8

n	5	6	7	8	9
$3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n}{5} \rfloor \rceil - 1$	2	2	2	2	2
$\frac{3}{10} \cdot n - 3$	-1.5	-1.2	-0.9	-0.6	-0.3

- Induktionsschritt von n auf $n + 10$:

$$\begin{aligned} 3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n+10}{5} \rfloor \rceil - 1 &= 3 + 3 \cdot \lceil \frac{1}{2} \cdot \lfloor \frac{n}{5} \rfloor \rceil - 1 \\ \text{(IH)} \quad &\geqslant 3 + \frac{3}{10} \cdot n - 3 \\ &= \frac{3}{10} \cdot (n + 10) - 3 \end{aligned}$$

Lineare Laufzeit von Quick-Select

Erhalte Rekursionsgleichung

$$\begin{aligned} T(n) &\leqslant \underbrace{max(T(|\text{links}|), T(|\text{rechts}|))}_{\text{rekursiver Aufruf}} + \underbrace{T(\lfloor \frac{n}{5} \rfloor)}_{\text{Median der M.}} + \underbrace{a \cdot n}_{\text{Partition + Sortierung}} \\ &\leqslant T(n - (\frac{3}{10} \cdot n - 3)) + T(\lfloor \frac{n}{5} \rfloor) + a \cdot n \\ &\leqslant T(\frac{7}{10} \cdot n + 3) + T(\frac{n}{5}) + a \cdot n \end{aligned}$$

- lineare Laufzeit, da $\frac{7}{10} + \frac{1}{5} = \frac{9}{10} < 1$
- (wenn Summe = 1 wäre, erhalte $\Theta(n \cdot log(n))$)

Einsatz von Quick-Select

- optimales Pivot-Element (Median) für Quick-Sort:

```
int pivot = quickSelect(a, left, right, (left+right)/2);
```

⇒ Quick-Sort garantiert in $\mathcal{O}(n \cdot log(n))$

- größere Konstanten: schlechtere Laufzeit als Heap-Sort
- Optimierung: quickSelect führt schon Partitionierung durch

⇒ nicht mehr notwendig in Quick-Sort

```
static void quickSortMedian(int[] a, int l, int r) {
    if (l < r) {
        int m = (l+1)/2 + r/2;
        quickSelect(a, l, r, m);
        quickSortMedian(a,l,m-1);
        quickSortMedian(a,m+1,r);
    }
}
```

- trotzdem: wegen großer Konstante schlechtere Laufzeit als Heap-Sort

Zusammenfassung

- Suchverfahren
  - lineare Suche $\mathcal{O}(n)$ (keine Sortierung notwendig)
  - binäre Suche $\mathcal{O}(log(n))$, Interpolations-Suche $\mathcal{O}(log(log(n)))$
- Sortierverfahren
  - Heap-Sort: worst-case $\mathcal{O}(n\, log(n))$
  - Quick-Sort: average-case $\mathcal{O}(n\, log(n))$, worst-case $\mathcal{O}(n^2)$ (Pivot-Funktion)
  - Bucket- und Radix-Sort: worst-case $\mathcal{O}(n)$, Ausnutzen von Struktur von Schlüsseln
- Selektionsverfahren (i.-kleinstes Element)
  - Quick-Select: worst-case $\mathcal{O}(n)$

  ⇒ optimierter Quick-Sort in $\mathcal{O}(n \cdot log(n))$

Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

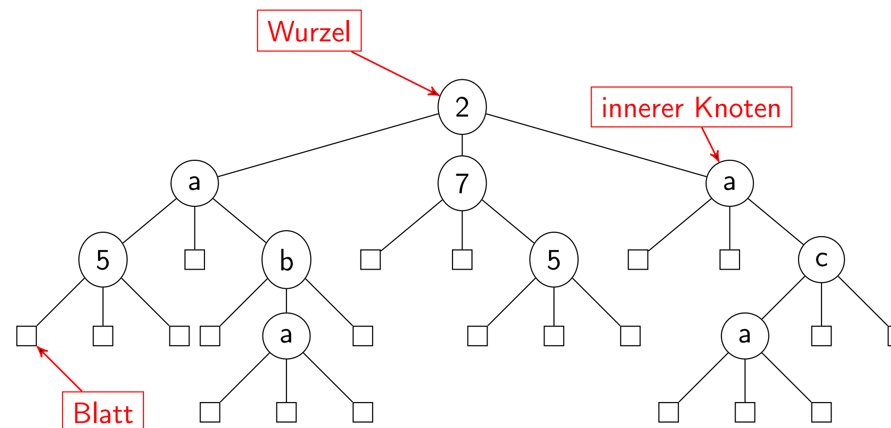
Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

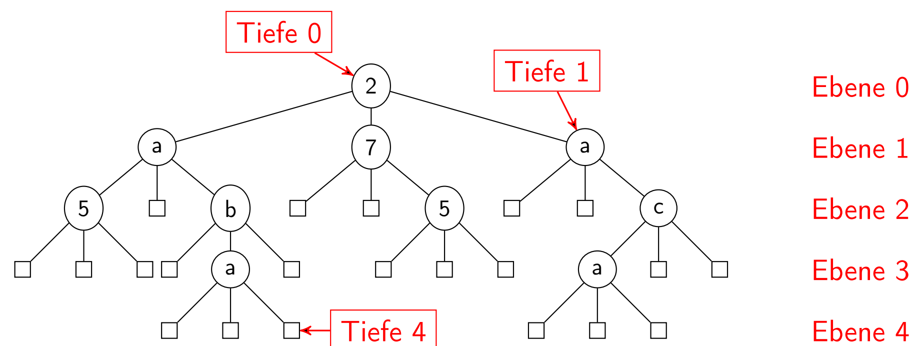
Was ist ein Baum?

- häufig genutzte Datenstruktur: Datenbanken, Dateisystem, ...
- besteht aus Knoten
  - Knoten hat Verweise auf Kinder
    - 0 oder k Kinder: Baum hat Ordnung k
    - Bäume der Ordnung 1: Listen
    - Bäume der Ordnung 2: Binärbäume
    - Bäume der Ordnung > 2: Vielwegbäume
  - n ist Elternknoten von k gdw. k Kind von n ist
  - Knoten ohne Kinder: Blätter
  - Knoten mit Kindern: innerer Knoten
  - Knoten kann Daten beinhalten
    häufig: Daten nur in Blättern, oder Daten nur in inneren Knoten
  - Ausgezeichneter Wurzelknoten (Wurzel)
  - Bäume sind azyklisch
    - jeder Knoten außer der Wurzel hat genau einen Elternknoten
    - Wurzel hat keinen Elternknoten
  - Bäume wachsen von oben nach unten

Baum der Ordnung 3 mit Werten in inneren Knoten



Tiefen, Höhen, Ebenen



- Tiefe eines Knotens = Abstand zur Wurzel
- Ebene n = alle Knoten mit Tiefe n
- Höhe eines Baums = maximale Tiefe
- Beispiel-Baum hat Höhe 4

Java Generics

- Daten in Bäumen, Listen, ... oft unterschiedlich: Integer, Strings, ...

```
class List_Integer {
   Node_Integer head;
   void insert(Integer data) {
      this.head = new Node_Integer(data, this.head);
   }
}
class Node_Integer {
   Integer data;
   Node_Integer next;
   Node_Integer(Integer data, Node_Integer next) {
      this.data = data;
      this.next = next;
   }
}
```

```
      List_Integer list_integer = new List_Integer();
      list_integer.insert(5);
```



```
class List_String {
   Node_String head;
```

Java Generics: Klassen haben als Parameter Datentyp

- Daten in Bäumen, Listen, ... oft unterschiedlich: Integer, Strings, ...

```
class List<D> {
   Node<D> head;
   void insert(D data) {
      this.head = new Node<D>(data, this.head);
   }
}
class Node<D> {
   D data;
   Node<D> next;
   Node(D data, Node<D> next) {
      this.data = data;
      this.next = next;
   }
}
```

```
      List<String> list_string = new List<String>();
      list_string.insert("text");
```


Java Generics

Instantiierung durch beliebige Datentypen (außer primitiven Datentypen)

```
      List<Integer> list_integer = new List<Integer>();
      list_integer.insert(5);
      List<String> list_string = new List<String>();
      list_string.insert("text");
      List<List<String>> list = new List<List<String>>();
      list.insert(list_string);
```

Auch Methoden können generisch sein

```
   static <D> D firstElement(List<D> list) {
      if (list.head == null) {
         throw new RuntimeException("no such element");
      } else {
         return list.head.data;
      }
   }
```

Java Generics

- generische Klassen brauchen immer Typparameter in spitzen Klammern
  `List<Integer> someList = new List<Integer>();`
- Generics werden in fast allen Container-Klassen verwendet
  - List<D>
  - Set<D>
  - Tree<D>
  - Map<K,D>
    Abbildung von Schlüsseln vom Typ K auf Datensätze vom Typ D
- In nicht-statischen Methoden sind Typparameter der Klasse verfügbar

```
class SomeClass<D> {
  D someMethod(D someParam) { ... } //okay
  static D someStaticMethod(D someParam) { ... } //not okay
  static <E> E otherStaticMethod(E someParam) { ... } //okay
}
```


Implementierungen von Bäumen mit generischem Datentyp

wesentlicher Unterschied: Speicherung von Kind-Verweisen

- für Bäume kleiner Ordnung: explizite Namen

```
class TreeAlternativeA<D> {
   NodeA<D> root;
}
class NodeA<D> {
   NodeA<D> firstChild;
   NodeA<D> secondChild;
   NodeA<D> thirdChild;
   D data;
}
```

Implementierungen von Bäumen mit generischem Datentyp

- Arrays (falls Anzahl Kinder nicht stark variieren)

```
class TreeAlternativeB<D> {
   NodeB<D> root;
}
class NodeB<D> {
   NodeB<D>[] children;
   D data;
}
```

- Listen (falls Anzahl Kinder variieren)

```
class TreeAlternativeC<D> {
   NodeC<D> root;
}
class NodeC<D> {
   List<NodeC<D>> children;
   D data;
}
```


Implementierungen von Bäumen mit generischem Datentyp

- manchmal separate Klassen für Blätter und innere Knoten

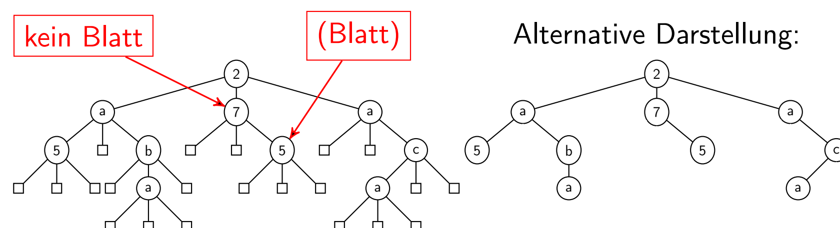
⇒ benötigt Objekt-Orientierung mit Vererbung

- falls in Blättern keine Daten

⇒ jedes Blatt ist gleich

⇒ repräsentiere Blätter durch `null`

- in dieser Konstellation werden dann manchmal innere Knoten mit nur Blättern als Nachfolgern auch als Blatt bezeichnet!



Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

Wörterbücher / Dictionaries / Maps

- Wörterbuch ist Abbildung von Schlüsseln auf Datensätze
- 4 grundlegende Operationen
  - Auslesen von Daten mittels Schlüssel (get)
  - Einfügen / Überschreiben von Datensätzen (put)
  - Löschen von Datensatz mittels Schlüssel (remove)
  - Iteration über alle Einträge (iterator)
- Entsprechendes Interface (Vereinfachung: Schlüssel = **int**)

```
public interface Dictionary<D> {
  D get(int key);
  void put(int key, D data);
  void remove(int key);
  Iterator<D> iterator();
}
```

Iteratoren

- Iterator liefert nach und nach alle Werte einer Datenstruktur
- 2 grundlegende Operationen
  - gibt es weiteres Element? (hasNext)
  - liefere nächstes Element (next)
- optional auch Modifikationsmöglichkeiten
  - entferne zuletzt geliefertes Element (remove)
  - füge an momentaner Position Element ein (insert)
- entsprechendes Interface spezialisiert auf Wörterbücher

```
public interface Iterator<D>
      extends java.util.Iterator<Entry<D>> {
   boolean hasNext();
   Entry<D> next();
   void remove();
}
```

```
public class Entry<D> {
   public int key;
   public D data;
}
```


Wörterbuch Implementierungen

Datenstruktur	get	put	remove
sortiertes Array	$\mathcal{O}(log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
verkettete Liste	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Bäume	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$	$\mathcal{O}(log(n))$

Übersicht

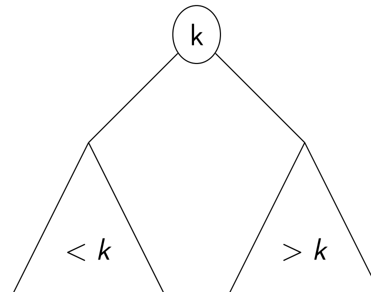
- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

Binäre Suchbäume

- Bäume der Ordnung 2 (linker und rechter Teilbaum)
- Werte in inneren Knoten, bestehend aus Schlüssel k und Datensatz d
- Blätter enthalten keine Daten

⇒ Blatt = Knoten, bei dem beide Kinder **null** sind

- Sortierung: für jeden inneren Knoten mit Schlüssel k sind
  - Schlüssel im linken Teilbaum kleiner als k
  - Schlüssel im rechten Teilbaum größer als k



Datenstruktur Knoten

```
class Node<D> {
   Node<D> left;
   Node<D> right;
   Node<D> parent;
   int key;
   D data;
   public Node(int key, D data,
         Node<D> left, Node<D> right, Node<D> parent) {
      this.key = key;
      this.data = data;
      this.left = left;
      this.right = right;
      this.parent = parent;
   }
}
```


Datenstruktur Binärbaum

```
public class BinTree<D> implements Dictionary<D> {
   Node<D> root;
}
```

Einfügen von 10, 4, 8, 15, 7, -2, 4, 6, 11, 3, 18, 12, 9, 2, 13

Einfügen von Datensatz d unter Schlüssel k

- Starte von Wurzel
- Suche Knoten mit Schlüssel k
  - k = Schlüssel des aktuellen Knotens ⇒ suche beendet
  - k < Schlüssel des aktuellen Knotens ⇒ suche im linken Teilbaum
  - k > Schlüssel des aktuellen Knotens ⇒ suche im rechten Teilbaum
  - beende Suche spätestens in Blatt
- 3 Alternativen für Suchergebnis
  - Suche liefert Knoten mit Schlüssel k ⇒ überschreibe Daten mit d
  - Suche liefert Knoten n mit Schlüssel ungleich k ⇒ erzeuge neuen Knoten (k,d) und füge diesen direkt unter n ein
  - Suche liefert **null** ⇒ Baum war leer, erzeuge initiale Wurzel (k,d)

Suche

```
Node<D> find(int key) {
    Node<D> current = this.root;
    Node<D> previous = null;
    while (current != null) {
        previous = current;
        if (key < current.key) {
            current = current.left;
        } else if (key == current.key) {
            return current;
        } else {
            current = current.right;
        }
    }
    return previous;
}
```


Einfügen

```
public void put(int key, D data) {
    Node<D> node = find(key);
    if (node == null) { // root is null
        this.root = new Node<D>(key,data,null,null,null);
    } else if (node.key == key) { // overwrite
        node.data = data;
    } else { // new node below node
        Node<D> newNode = new Node<D>(key,data,null,null,node);
        if (key < node.key) {
            node.left = newNode;
        } else {
            node.right = newNode;
        }
    }
}
```

Laufzeit Suchen / Einfügen

- Suchen: $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist
- Einfügen = Suchen + $\mathcal{O}(1)$
- Baum mit n Elementen: Höhe zwischen $log_2(n)$ und n
- Einfügen von n Werten in zufälliger Reihenfolge $\Rightarrow$ erwarte Höhe: $1.386 \cdot log_2(n)$

Einfügen von -2, 3, 4, 6, 7, 8, 10, 11

Löschen eines Datensatzes bzgl. Schlüssel k

- Suche Knoten n mit Schlüssel k
- Falls k im Baum vorhanden, lösche n

```
public void remove(int key) {
    Node<D> node = find(key);
    if (node != null && node.key == key) {
        removeNode(node);
    }
}
```

- Verbleibendes Problem: Löschen von Knoten

Löschen von 18, 9, 11, 3, 10, 2, 12, 8, 15, 7, -2, 4, 6, 13

Löschen von Knoten n mit Schlüssel k

3 Fälle

1. Löschen von Blättern

⇒ einfach

2. Löschen von Knoten mit nur einem Kind

⇒ ersetze zu löschenden Knoten durch Kind

3. Löschen von Knoten mit zwei Kindern

⇒ vertausche Knoten n mit rechtestem Knoten r im linken Teilbaum (r hat größten Schlüssel unter allen Schlüsseln kleiner k)

⇒ Sortierungsbedingung an der ursprünglichen Position von n gewährleistet, bis auf den nach unten getauschten Knoten n

⇒ da rechtester Knoten kein Blatt ist, kann nun Fall 2 benutzt werden, um Knoten n zu löschen

Laufzeit Löschen

- Suchen: $\mathcal{O}(h)$, wobei h die Höhe des Baumes ist
- removeNode $= \mathcal{O}(1) +$ maximum $= \mathcal{O}(h)$

⇒ Suchen, Einfügen und Löschen: $\mathcal{O}(h)$

- Einfügen von n Werten in zufälliger Reihenfolge

⇒ erwarte Höhe: $1.386 \cdot log_2(n)$

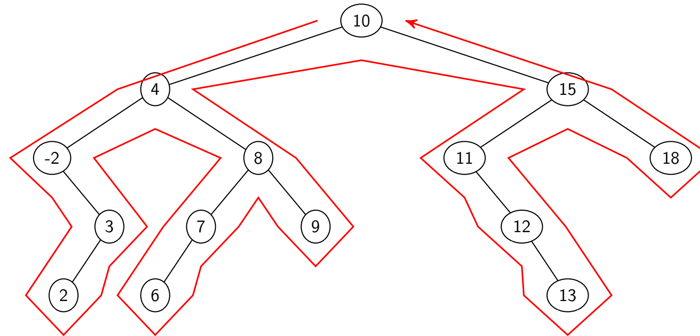
- Einfügen von n Werten in zufälliger Reihenfolge, dann $\geqslant n^2$ mal: (einfügen, löschen)

⇒ erwarte Höhe: $\Theta(\sqrt{n})$
Problem: löschen macht Bäume rechtslastig
(löschen mit 2 Kindern ersetzt Knoten immer durch kleinere Werte)

- besserer Ansatz für löschen mit 2 Kindern: abwechselnd
  - größter Schlüssel im linken Teilbaum
  - kleinster Schlüssel im rechten Teilbaum

⇒ empirisch bessere Balancierung, analytisch bisher nicht gelöst

Iteration über Bäume



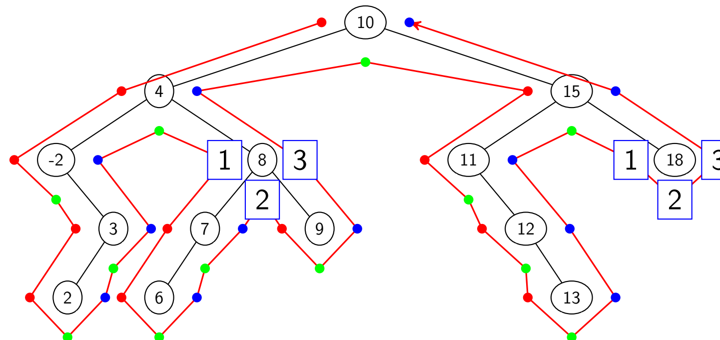
- 3 verschiedene Reihenfolgen
  - Preorder: erst Knotendaten, dann Teilbäume

  ⇒ 10, 4, -2, 3, 2, 8, 7, 6, 9, 15, 11, 12, 13, 18
  - Postorder: erst Teilbäume, dann Knotendaten

  ⇒ 2, 3, -2, 6, 7, 9, 8, 4, 13, 12, 11, 18, 15, 10
  - Inorder (für Binärbäume): links, Daten, rechts

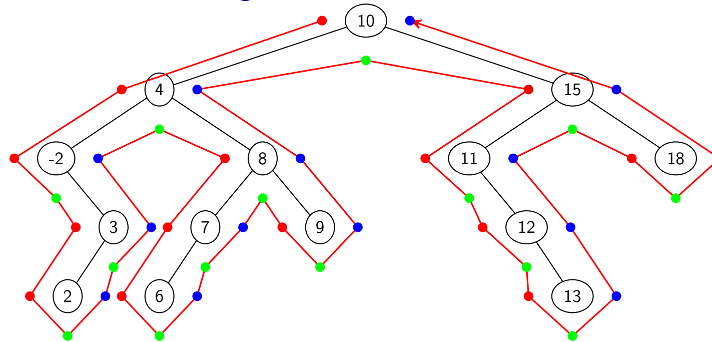
  ⇒ -2, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 15, 18

Iteration über Binärbäume



- Jeder Knoten wird dreimal besucht
  1. Pfad kommt von oben
  2. Pfad kommt vom linken Teilbaum
  3. Pfad kommt vom rechten Teilbaum
- Alle Traversierungsarten laufen entlang des roten Pfads
  - Preorder: Ausgabe des Knotens im Schritt 1
  - Inorder: Ausgabe des Knotens im Schritt 2
  - Postorder: Ausgabe des Knotens im Schritt 3

Iteration entlang des roten Pfades



- benötige nur aktuellen Knoten und Eingangsrichtung, um nächsten Knoten zu bestimmen

⇒

```
public class TreeIterator<D> implements Iterator<D> {
    int direction;    // from which direction ...
    Node<D> current; // ... did we enter current node
    int order;        // traversal order
```


Idee des Iterators

- laufe alle Haltepunkte (farbige Punkte) entlang
- `void walkNext(int stopAt)` läuft bis zum nächsten Haltepunkt
  - stopAt bei welcher Richtung soll gestoppt werden
    - UP = PREORDER = 1
    - LEFT = INORDER = 2
    - RIGHT = POSTORDER = 3
    - NONE = 0, mache mindestens einen Schritt, wechsele dann zur Traversierungsordnung
  - current steht anschließend auf Knoten, der ausgegeben werden soll

```
public boolean hasNext() {
    return current != null;
}
public Entry<D> next() {
    if (current == null) {
        throw new NoSuchElementException();
    } else {
        Entry<D> entry = new Entry<D>(current.key, current.data);
        walkNext(NONE);
        return entry;
    }
}
```

Initialisierung

```
public class TreeIterator<D> implements Iterator<D> {
   int direction;    // from which direction ...
   Node<D> current;  // ... did we enter current node
   int order;        // traversal order

   public TreeIterator(Node<D> root, int order) {
      this.order = order;
      this.current = root;
      this.direction = UP;
      walkNext(order);
   }
```

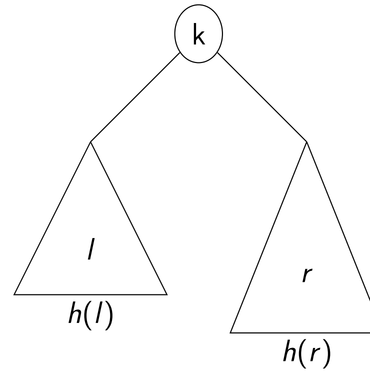
```
public class BinTree<D> implements Dictionary<D> {
   Node<D> root;
   public Iterator<D> iterator() {
      return new TreeIterator<D>(this.root, TreeIterator.INORDER);
   }
}
```


Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

AVL-Baum

- Binärer Suchbaum ist AVL-Baum gdw. für jeden inneren Knoten

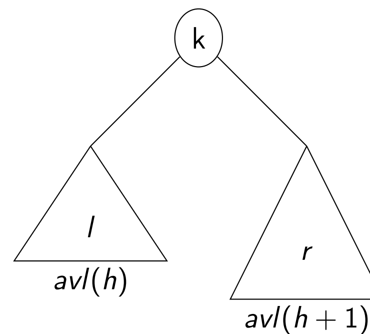


die Höhendifferenz $|h(l) - h(r)| \leqslant 1$ ist

Anzahl Knoten eines AVL-Baums

Sei $avl(h)$ minimale Anzahl der Blätter eines AVL-Baum mit Höhe h
(hier: **null**-Blätter $\Rightarrow$ Anzahl innerer Knoten = Anzahl Blätter - 1)

- $avl(0) = 2$ nur Wurzel
- $avl(1) = 3$ Wurzel und ein Kind
- $avl(h+2) = avl(h) + avl(h+1)$



$\Rightarrow avl(h) = fib(h+3)$

Höhe eines AVL-Baums

$$
\begin{aligned}
fib(h) &= \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+1} \right) \\
&\approx \frac{1+\sqrt{5}}{2 \cdot \sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h} \\
&= 0,7236\ldots \cdot (1,618\ldots)^{h}
\end{aligned}
$$

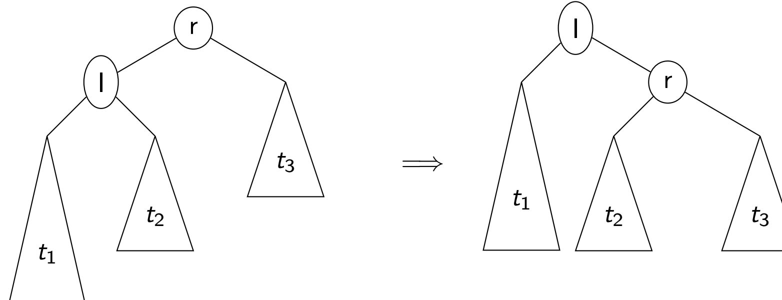
⇒ da $avl(h) = fib(h+3)$, wächst die Anzahl der Knoten eines AVL-Baums der Höhe h exponentiell in h

⇒ die Höhe eines AVL-Baums mit n Knoten ist logarithmisch in n

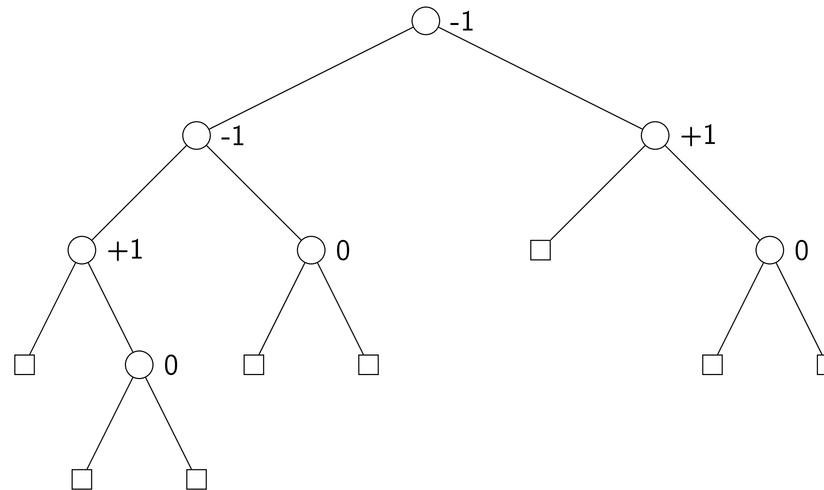
⇒ Suchen, Einfügen, Löschen in AVL-Bäumen: $\mathcal{O}(log(n))$

Erzeugung von AVL-Bäumen

- leerer Baum ist AVL-Baum
- modifiziere einfügen und löschen so, dass AVL-Eigenschaft erhalten bleibt
- wesentliche Hilfsmittel:
  - speichere Höhendifferenz hdiff $\in \{-1, 0, 1\}$ als Attribut in Knoten
  - führe Rotationen durch, um Höhendifferenz wiederherzustellen
  - rechts-Rotation von r: Sortierung bleibt erhalten

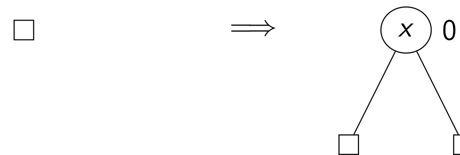


Höhendifferenzen

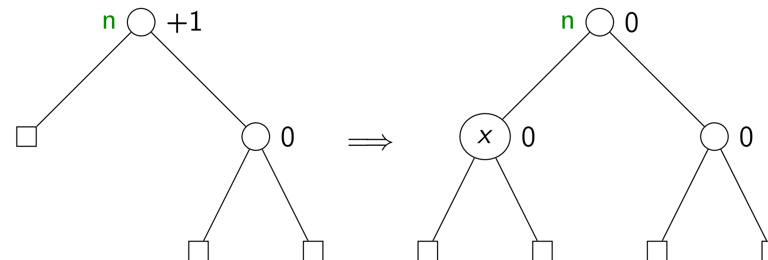


Einfügen von x in AVL-Bäume: 4 Fälle

- leerer Baum:

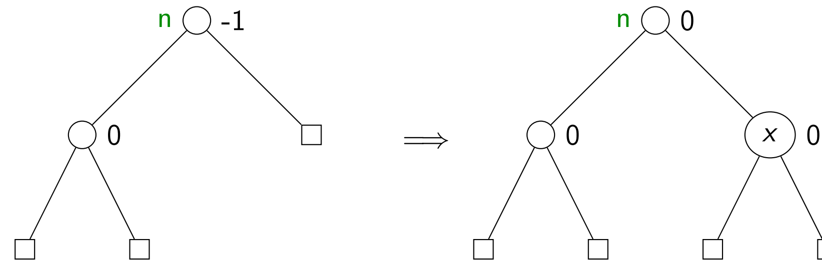


- suche endet in Knoten n, der rechtslastig ist (n. hdiff = +1):



Einfügen von x in AVL-Bäume: 4 Fälle

- suche endet in Knoten n, der linkslastig ist (n. hdiff $= -1$):

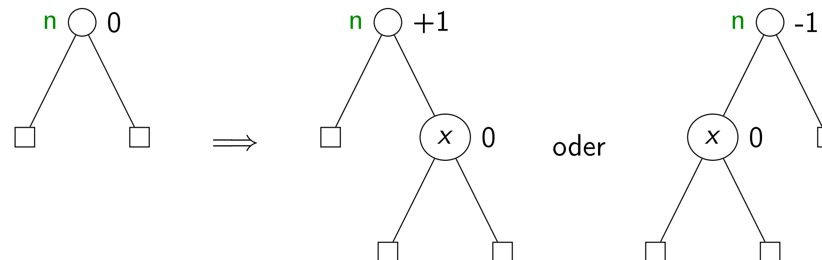


$\Rightarrow$ Beim Einfügen mit n. hdiff $\in \{-1, +1\}$ bleibt Höhe von n gleich

$\Rightarrow$ AVL-Eigenschaft bleibt erhalten

Einfügen von x in AVL-Bäume: 4 Fälle

- suche endet in Knoten n, der balanciert ist (n. hdiff $= 0$):



$\Rightarrow$ beim Einfügen wird Höhe des Teilbaums mit Wurzel n vergrößert

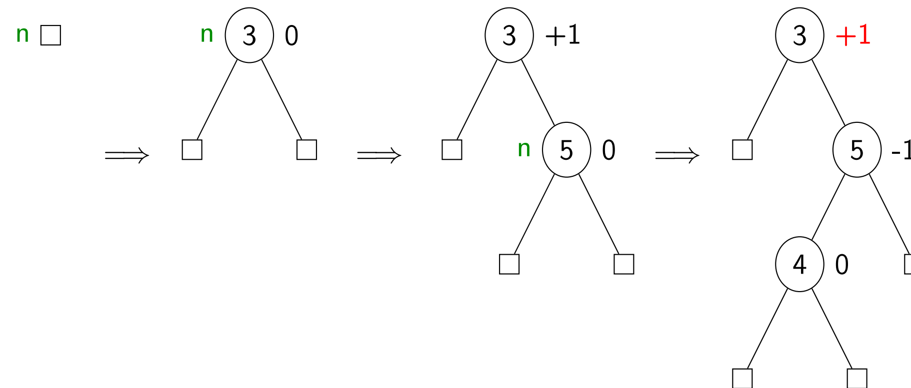
$\Rightarrow$ Überprüfung bzw. Wiederherstellung der AVL-Eigenschaft notwendig

$\Rightarrow$ Aufruf der Methode **void** balGrow(AVLNode<D> n)

- nimmt an, dass n. hdiff $\in \{-1, +1\}$ vor Aufruf gilt
- nimmt an, dass Höhe des Teilbaums mit Wurzel n um 1 gewachsen ist
- bricht spätestens ab, falls Wurzel erreicht

Notwendigkeit von balGrow

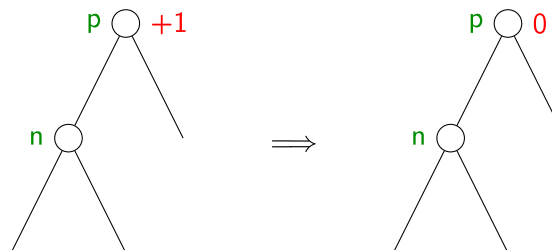
Einfügen von 3, 5, 4



das Einfügen der 4 unterhalb des Knoten mit der 5

- invalidiert die hdiff-Einträge
- hinterläßt einen Baum, der nicht die AVL-Eigenschaft erfüllt

balGrow(n): n ist linker Sohn von p und p.hdiff $= +1$



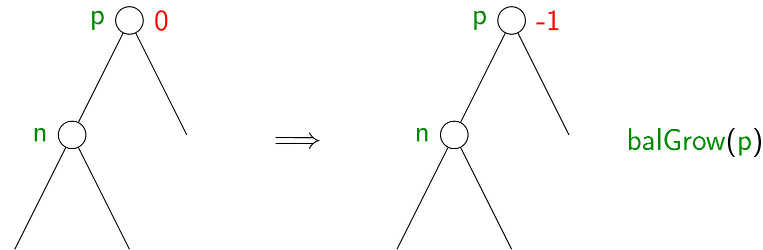
- +1 ist p.hdiff vor Einfügen des neuen Knotens im linken Teilbaum
- Aufruf-Bedingung besagt, dass linker Teilbaum um 1 in der Höhe gewachsen ist

⇒ nach Einfügen sind linker und rechter Teilbaum von p gleich hoch

⇒ Baum mit Wurzel p bleibt gleich hoch

⇒ balGrow(n) ist fertig

balGrow(n): n ist linker Sohn von p und p. hdiff $= 0$



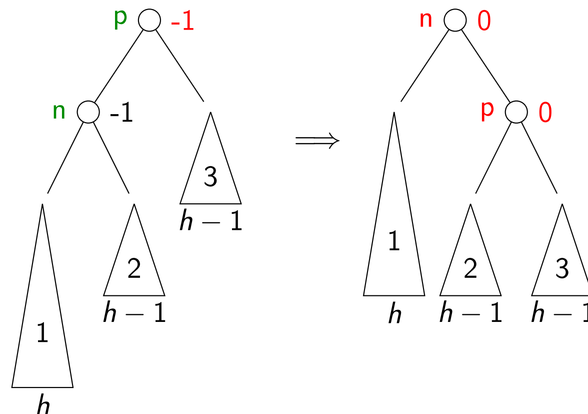
- 0 ist p. hdiff vor Einfügen des neuen Knotens im linken Teilbaum
- Aufruf-Bedingung besagt, dass linker Teilbaum um 1 in der Höhe gewachsen ist

$\Rightarrow$ nach Einfügen ist linker Teilbaum von p um 1 höher

$\Rightarrow$ Baum mit Wurzel p ist um 1 höher geworden

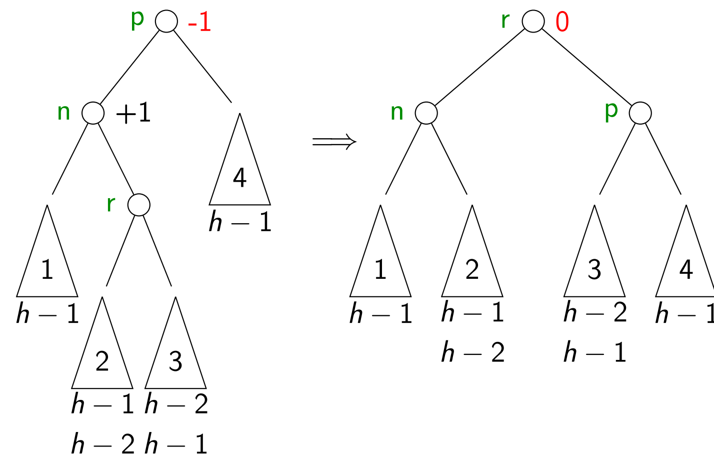
$\Rightarrow$ Aufruf von balGrow(p)

n ist linker Sohn von p, p. hdiff $= -1$, n. hdiff $= -1$



- Ursprüngliche Höhe von Baum mit Wurzel p: $h + 1$
- nach Einfügen: Höhe des Baums mit Wurzel n: $h + 1$
- Rechts-Rotation notwendig für AVL-Bedingung
- Höhe von Baum nach Rotation: $h + 1$

n ist linker Sohn von p, p. hdiff $= -1$, n. hdiff $= +1$



- Ursprüngliche Höhe von Baum mit Wurzel p: $h + 1$
- Doppel-Rotation für AVL-Bedingung: rotateLeft (n); rotateRight (p);
- Höhe von Baum nach Rotation: $h + 1$

balGrow(n)

- n ist rechter Sohn von p: komplett symmetrisch

⇒ Laufzeit: $\mathcal{O}(h) = \mathcal{O}(log(n))$

⇒ Anzahl Rotationen: 1 (Doppel-)Rotation

⇒ Einfügen in AVL-Bäumen: $\mathcal{O}(log(n))$

Beispiel: Einfügen von 9, 2, 1, 10, 0, 4, 6, 3, 7, 8

Löschen in AVL-Bäumen

- analog zum Einfügen: wie Löschen in binären Suchbäumen + Rebalancierung
- **void** balShrink(n)
  - Aufruf, falls n.hdiff = 0 und Höhe von Teilbaum mit Wurzel n durch Löschen um 1 verringert
  - Laufzeit: $\mathcal{O}(h) = \mathcal{O}(log(n))$
  - Anzahl Rotationen: $\leqslant h$ (Doppel-)Rotationen
  - $\Rightarrow$ Löschen in AVL-Bäumen: $\mathcal{O}(log(n))$

Zusammenfassung AVL-Bäume

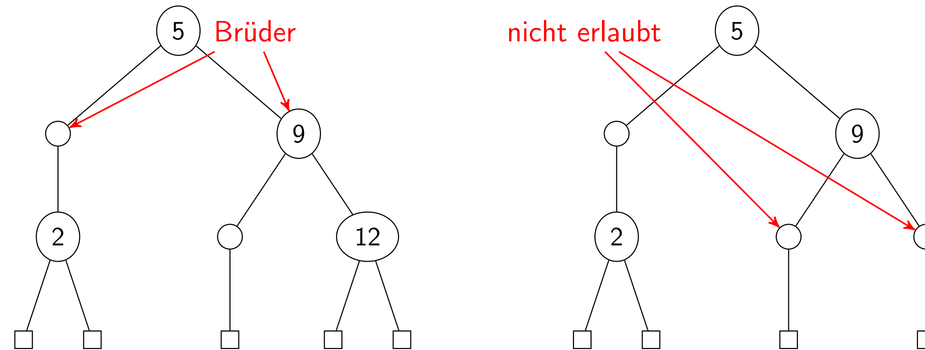
- binäre Suchbäume
- Knoten haben zusätzliches Attribut: Höhendifferenz
- AVL-Bedingung: Höhendifferenz in jedem Knoten maximal 1
- Rotationen um AVL-Bedingung zu gewährleisten
- Suchen, Einfügen und Löschen: $\mathcal{O}(log(n))$

Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

Bruder-Baum

- zwei Knoten sind Brüder gdw. sie den gleichen Elternknoten haben
- Baum ist Bruder-Baum gdw.
  - jeder innere Knoten 1 oder 2 Kinder hat, (unäre und binäre Knoten)
  - jeder binäre Knoten ein Schlüssel speichert,
  - jeder unäre Knoten einen binären Bruder hat,
  - alle Blätter auf der gleichen Ebene sind, und
  - die binären Knoten sind geordnet: "links < Schlüssel < rechts"



Suchen in Bruder-Bäumen

- Suche ist wie in Binärbäumen
- einzige Ausnahme: unäre Knoten werden direkt übersprungen

$\Rightarrow$ Kosten Suche: $\mathcal{O}(h)$ für Brüder-Bäume der Höhe h

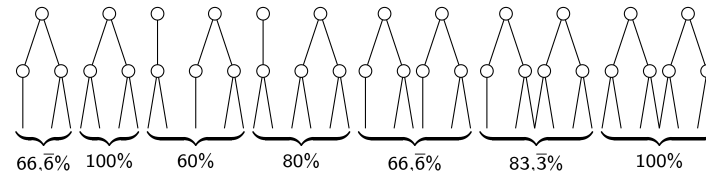
Höhe von Bruder-Bäumen

- falls zuviele unäre Knoten vorkommen

⇒ Höhe linear in Knotenanzahl, wenig Schlüssel

- Bedingung "jeder unäre Knoten hat binären Bruder" verhindert dies
  - wie bei AVL-Bäumen kann die minimale Anzahl Blätter in Brüder-Bäumen über die Fibonacci-Funktion (*fib*(Höhe + Konstante)) ausgedrückt werden

  ⇒ Höhe von Brüder-Bäumen ist logarithmisch in Knoten-Anzahl
  - mindestens 60 % der innerer Knoten sind binär

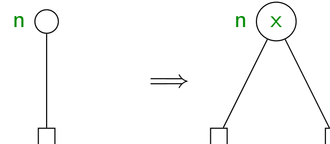


⇒ Höhe von Brüder-Bäumen ist logarithmisch in Schlüssel-Anzahl

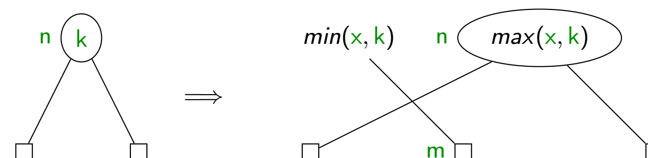
Einfügen in Bruder-Bäume

- leerer Baum: trivial
- nicht-leerer Baum analog zu Einfügen in binäre Suchbäume
  - finde Knoten n, unterhalb dessen Schlüssel x eingefügt werden soll
  - Bedingung "Blätter auf gleicher Ebene" ⇒ Kinder von n sind Blätter

  ⇒ 2 Fälle
    - n ist unärer Knoten:



    - n ist binärer Knoten:

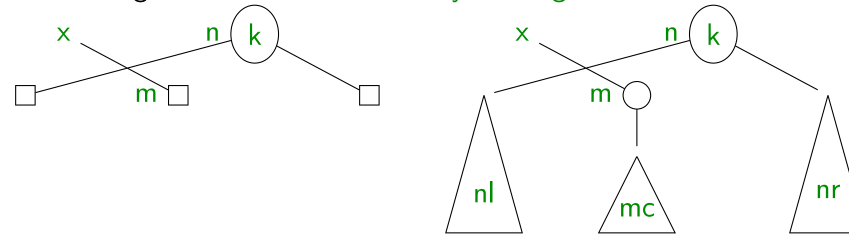


⇒ Aufruf von up(n,m,min(x,k))

up(n,m,x)

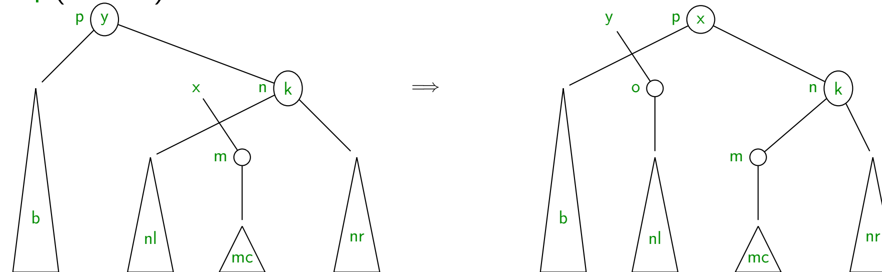
Voraussetzungen:

- n ist binärer Knoten in einem Bruder-Baum
- n. left und n. right sind Wurzeln von Bruder-Bäumen (also Blätter oder binäre Knoten)
- m ist Blatt oder unärer Knoten und das Kind von m ist Bruder-Baum
- die Höhen von m, n. left , und n. right sind gleich
- Anordnung: n. left $< x < m <$ n.key $<$ n. right



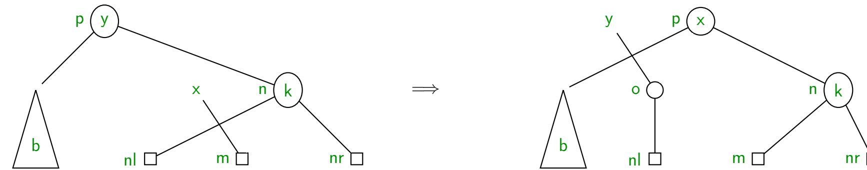
- nach Ausführung: erhalte Bruder-Baum, der um x und m ergänzt ist
- Idee: up kann x—m-Baum einhängen, wenn es unären Bruder oder Elternknoten gibt

up(n,m,x): Fall n hat binären linken Bruder



- Sortierung bleibt erhalten
- da nl und nr Bruder-Bäume gilt: Wurzel von nl und nr binär
- Wurzel von nr binär $\Rightarrow$ m bekommt binären Bruder $\Rightarrow$ n bleibt Bruder-Baum
- Einfügen auf höhere Ebene verschoben, rufe anschließend up(p,o,y) auf
- b hat gemäß Fall binäre Wurzel, n bleibt binär $\Rightarrow$ rekursiver Aufruf ok

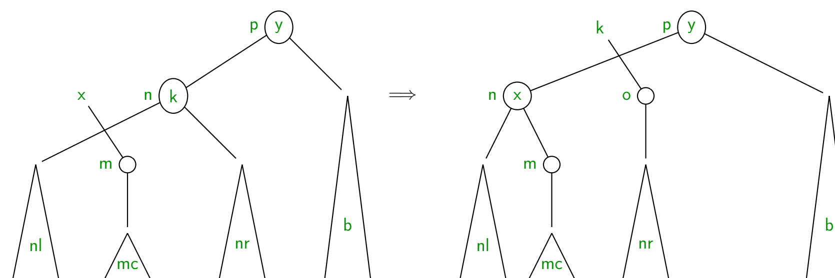
up(n,m,x): Fall n hat binären linken Bruder, m ist Blatt



- genauso wie voriger Fall, ersetze einfach nl und nr durch Blätter

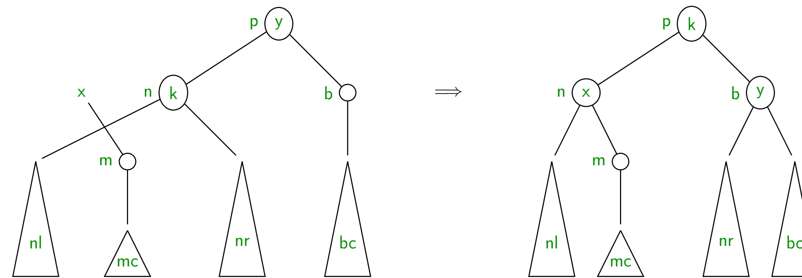
⇒ im Folgenden wird "m ist Blatt"-Fall nicht mehr explizit aufgeführt

up(n,m,x): Fall n hat binären rechten Bruder



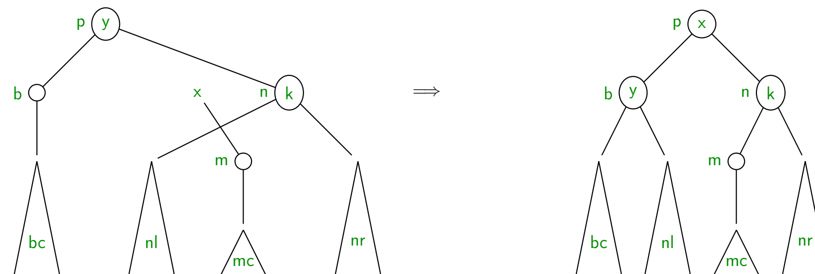
- analog voriger Fall:
- Sortierung und Bruder-Baum-Eigenschaft bleiben erhalten
- Einfügen auf höhere Ebene verschoben, rufe anschließend up(p,o,k) auf

up(n,m,x): Fall n hat unären rechten Bruder



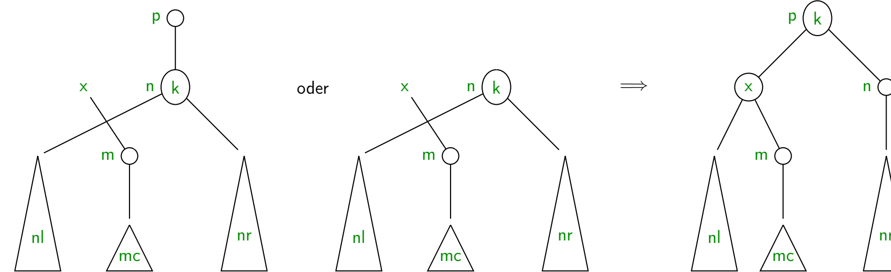
- Bruder b wird binär
- kein rekursiver Aufruf nötig

up(n,m,x): Fall n hat unären linken Bruder



- analog unärer rechter Bruder
- kein rekursiver Aufruf erforderlich

up(n,m,x): Fall n hat keinen Bruder



- n hat keinen Bruder ⇒ n hat unären Elternknoten oder n ist Wurzel
- kein rekursiver Aufruf erforderlich

⇒ Bruder-Bäume wachsen an der Wurzel

Beispiel: Einfügen von 10, 3, 2, 11, 1, 4, 6, 13, 8, 9, 5, 12, 7

Einfügen und Löschen in Bruder-Bäumen

- Einfügen: Suchen + up, Aufwand: $\mathcal{O}(log(n)) + \mathcal{O}(log(n))$
- Fragestellung: wie viel Zusatzaufwand benötigt up?
  - Laufzeit von up $\approx$ Anzahl eingefügter Knoten
    (jeder Fall, in dem up rekursiv aufgerufen wird, erzeugt 1 neuen Knoten)
  - bekannt: Bruder-Baum mit n Schlüsseln hat maximal $\frac{5}{3} \cdot n$ Knoten

  $\Rightarrow$ Ausführen von up führt bei n Einfüge-Operationen zu insgesamt maximal $\frac{5}{3} \cdot n$ (rekursiven) Aufrufen von up

  $\Rightarrow$ für jedes Einfügen ist durchschnittliche Ausführungszeit von up konstant (es wird meistens unten eingefügt)

  $\Rightarrow$ durchschnittliche Zeit für Einfügen: $\underbrace{\mathcal{O}(log(n))}_{\text{Suchen}} + \underbrace{\mathcal{O}(1)}_{\text{up}}$
- Löschen: Suchen + delete, Aufwand: $\mathcal{O}(log(n)) + \mathcal{O}(log(n))$
  - delete arbeitet ähnlich wie up

Zusammenfassung: Bruder-Bäume

- balancierte Bäume mit 1-2 Kindern, Blätter auf gleicher Ebene
- Suchen, Einfügen und Löschen in logarithmischer Zeit
- keine Rotationen wie in AVL-Bäumen
- Verallgemeinerungen:
  - a-b Bäumen (zwischen a und b viele Kinder)
  - B-Bäume (sind spezielle m-$2m$-Bäume, oft großes m in der Praxis)

Übersicht

- Bäume
  - Grundlagen
  - Wörterbücher
  - Binäre Suchbäume
  - AVL-Bäume
  - Bruder-Bäume
  - Splay-Bäume

Motivation

bekannte Bäume:

- binäre Suchbäume
  - \+ Speicherplatz optimal
  - \+ einfach zu implementieren (insgesamt 120 Zeilen)
  - − zu Listen degradierte Bäume möglich ⇒ Zugriffskosten: $\mathcal{O}(n)$
- AVL- und Bruder-Bäume
  - − zusätzlicher Speicherbedarf (hdiff bzw. unäre Knoten)
  - − aufwendige Fallanalysen in der Implementierung (z.B. alleine up: 100 Zeilen)
  - \+ Zugriffskosten: $\mathcal{O}(log(n))$

kommender Abschnitt:

- Splay-Bäume
  - \+ Speicherbedarf optimal (keine Zusatzinformationen in Knoten)
  - o einfache Fallanalysen in der Implementierung (insgesamt 160 Zeilen)
  - \+ amortisierte Zugriffskosten: $\mathcal{O}(log(n))$
  - \+ selbst-organisierend: oft geforderte Schlüssel im oberen Teil des Baums

Grundidee

- erlaube beliebige Bäume, also auch z.B. zu Listen entartete Bäume
- restrukturiere Bäume, so dass häufig zugegriffene Elemente nach oben wandern

$\Rightarrow$ Restrukturierung auch beim Suchen, nicht nur beim Einfügen und Löschen

- falls Baum zu Liste entartet ist, hat Suche Kosten von $\Theta(n)$

$\Rightarrow$ $\mathcal{O}(log(n))$ für jeden Zugriff nicht realisierbar

- Lösung: mittlere Kosten
  - bislang: n Operationen, die jede $\mathcal{O}(f(n))$ kostet
    $\Rightarrow$ Gesamtkosten: $\mathcal{O}(n \cdot f(n))$
  - umgekehrt: Gesamtkosten von n Operationen: $\mathcal{O}(n \cdot f(n))$
    $\Rightarrow$ jede Operation im Mittel: $\mathcal{O}(f(n))$
  - Beispiel: $(n-1) \times \mathcal{O}(1) + 1 \times \mathcal{O}(n)$ erzeugt mittlere Kosten von $\mathcal{O}(1)$

Hauptoperation: splay(root,x)

Arbeitsweise von splay:

- suche nach Schlüssel x im Teilbaum mit Wurzel root $\Rightarrow$ erhalte Knoten n (mit Schlüssel x, oder unter dem x eingefügt werden müsste)
- führe dann Rotationen (zig, zig-zig und zig-zag) durch, bis Knoten n an der Wurzelposition steht

Mittels splay sind Wörterbuch-Operationen einfach:

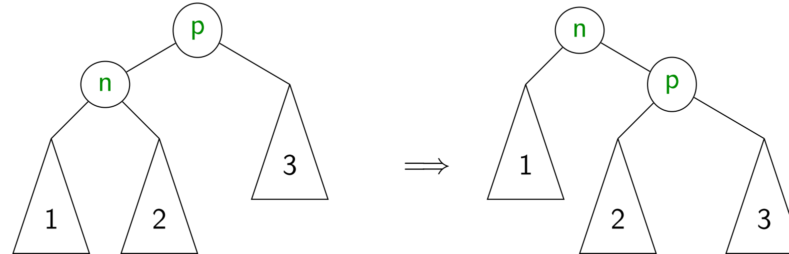
$\Rightarrow$ Suche nach x: splay(root,x) und vergleiche x mit Schlüssel in Wurzel

$\Rightarrow$ Löschen von x: splay(root,x) und lösche Wurzel-Knoten, falls dieser Schlüssel x enthält

$\Rightarrow$ Einfügen von x: splay(root,x) und erzeuge neue Wurzel, falls Wurzel nicht Schlüssel x enthält

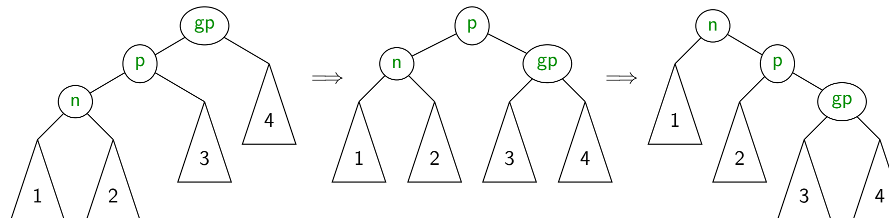
splay-Rotationen: zig

- wird durchgeführt, wenn der Vater p von n die Wurzel ist
- zig ist links- oder rechts-Rotation um p, so dass n zur Wurzel wird



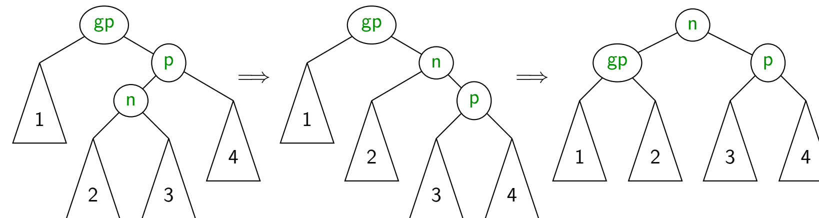
splay-Rotationen: zig-zig

- wird durchgeführt, wenn der Vater gp vom Vater p von n existiert und sowohl p und n beides linke Söhne oder beides rechte Söhne sind
- zig-zig ist doppelte rechts- oder doppelte links-Rotation, so dass n um 2 Ebenen nach oben wandert
- es wird erst um gp gedreht, dann um p (nicht umgekehrt)



splay-Rotationen: zig-zag

- wird durchgeführt, wenn der Vater gp vom Vater p von n existiert und n und p linker und rechter Sohn oder rechter und linker Sohn sind
- zig-zag ist rechts-links oder links-rechts Rotation, so dass n um 2 Ebenen nach oben wandert
- es wird erst um p gedreht, dann um gp (im Gegensatz zu zig-zig)



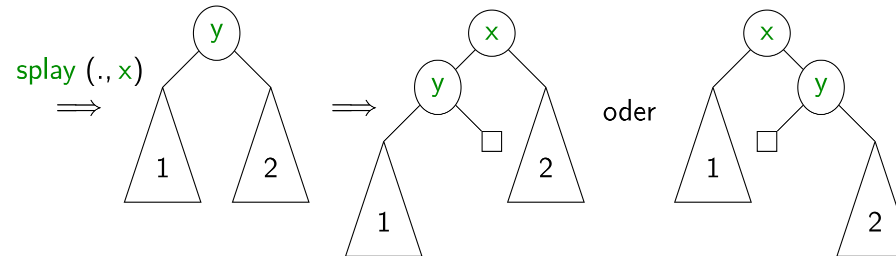
Suchen in Splay-Bäume

Suche nach x: splay(root,x) und vergleiche x mit Schlüssel in Wurzel

```
public D get(int key) {
   splay(this.root, key);
   if (this.root != null && this.root.key == key) {
      return this.root.data;
   } else {
      return null;
   }
}
```

Einfügen in Splay-Bäume

Einfügen von x: splay (root, x) und erzeuge neue Wurzel, falls für Wurzel y nach splay gilt: $y \neq x$



Beispiel: E1, E2, ..., E7, S1, S2, ... S7

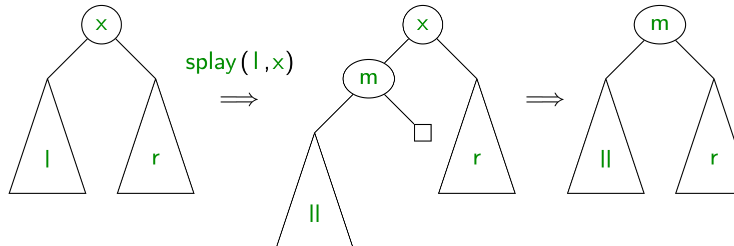
Löschen in Splay-Bäume

Löschen von x: splay(root,x) und lösche Wurzel-Knoten, falls dieser Schlüssel x enthält

- Fall 1: Wurzel hat keinen linken Teilbaum ⇒ mache rechten Teilbaum zur Wurzel
- Fall 2: Wurzel hat linken Teilbaum l
  - bekanntes Vorgehen: tausche x mit größtem Wert in l, lösche x in l
  - Splay-Bäume nutzen folgende Alternative:
    - splay(l,x)

    ⇒ größter Wert von l steht nun in Wurzel von l

    ⇒ Löschen wird danach einfach



Beispiel: E10, E9, ..., E1, S9, L9, L8, ..., L1, L10

Kostenanalyse Splay-Bäume

Beobachtungen:

- degenerierte Bäume können entstehen
- ⇒ Zugriff auf unterste Ebene ist teuer, aber
  - um degenerierte Bäume entstehen zu lassen, gab es zuvor viele günstige Operationen
  - nach dem Zugriff entsteht Baum, bei dem spätere Zugriffe wieder billiger sind
  - im Mittel sind Zugriffe $\mathcal{O}(log(n))$ (Warum? Wie zeigt man das?)

Amortisierte Analyse (Idee)

- Ziel: mittlere Laufzeit von $f(n)$
- gegeben ist Operationen-Folge $o_1, \ldots, o_m$
- verwalte Guthaben
- addiere nach und nach Kosten der Operationen $o_1, \ldots, o_m$
  - Kosten von $o_i \leqslant f(n) \Rightarrow$ addiere Differenz zum Guthaben
  - Kosten von $o_i > f(n) \Rightarrow$ subtrahiere Differenz vom Guthaben

Amortisierte Analyse (Details)

- gegeben ist Operationen-Folge $o_1, \ldots, o_m$
- Guthaben G: Funktion, die jedem Zustand eine Zahl zuordnet
  - G_0: initiales Guthaben
  - G_i: Guthaben der Struktur nach Abarbeitung von $o_1, \ldots, o_i$
- Kosten
  - t_i: (reale) Kosten der Operation o_i — schwankend
  - a_i: amortisierte Kosten der Operation o_i — nicht schwankend
    (reale Kosten plus Differenz der Guthaben)

$$a_i = t_i + G_i - G_{i-1}$$

- $G_0 \leqslant G_m \Rightarrow$ amortisierte Kosten bilden Schranke für reale Kosten

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + G_0 - G_m \leqslant \sum_{i=1}^{m} a_i$$

- ⇒ falls alle $a_i = \mathcal{O}(f(n))$ sind, ist die mittlere Laufzeit von $t_i = \mathcal{O}(f(n))$
- amortisierte Analyse: finde geeignetes G und zeige, dass $a_i = \mathcal{O}(f(n))$

Amortisierte Analyse für Splay-Bäume

- Kosten der Wörterbuch-Operationen ≈ Kosten von splay(root,x)
- Kosten von splay(root,x):
  - angenommen, x befindet sich auf Ebene h
  - ⇒ splay (., x) führt h Rotationen aus
  - ⇒ Suche nach x hat Kosten $\mathcal{O}(h)$
  - ⇒ Anzahl Rotationen ist sinnvolles Maß für Kosten von splay
- reale Kosten von splay(root,x):

$$\max(1, \text{Rotationen von splay(root,x)})$$

- Guthaben-Funktion G:
  - s(n): Anzahl Schlüssel im Teilbaum mit Wurzel n
  - r(n): Rang(n) $= log_2(s(n))$
  - $r(x) = r$(Knoten mit Schlüssel x)
  - G(tree) = Summe aller r(n)

Beispiel: (am.) Kosten beim Suchen nach 10,8,6,1,5,3,2,1

Amortisierte Analyse für Splay-Bäume

Lemma (Zugriffs-Lemma)

Die amortisierten Kosten von splay(root, x) sind höchstens
$3 \cdot (r(root) - r(x)) + 1$

Korollar

Für einen Baum mit höchstens N Knoten sind die amortisierten Kosten von splay(root, x) in $\mathcal{O}(log(N))$

Beweis

$$\begin{aligned}
& \text{amortisierte Kosten von splay(root, x)} \\
\leqslant\ & 3 \cdot (r(\text{root}) - r(\text{x})) + 1 \\
\leqslant\ & 3 \cdot r(\text{root}) + 1 \\
=\ & 3 \cdot log_2(s(\text{root})) + 1 \\
\leqslant\ & 3 \cdot log_2(N) + 1 \qquad (\text{da } s(\text{root}) \leqslant N)
\end{aligned}$$

Auf dem Weg zum Zugriffs-Lemma

Lemma (Zugriffs-Lemma)

amortisierte Kosten von splay(root, x) sind $\leqslant 3 \cdot (r(root) - r(x)) + 1$

Fall 1: Suche nach x liefert Wurzel

- ⇒ splay führt keine Rotation durch ⇒ reale Kosten von splay $= 1$
- ⇒ amortisierte Kosten = reale Kosten + Guthaben-Differenz $= 1 + 0 = 1$
- ⇒ amortisierte Kosten $= 1 = 3 \cdot 0 + 1 = 3 \cdot (r(\text{root}) - r(\text{x})) + 1$

Fall 2: Suche nach x liefert Knoten n ungleich Wurzel

- splay setzt sich aus vielen Rotationen zusammen
- ⇒ berechne amortisierte Kosten von splay über Abschätzung der amortisierten Kosten der Rotationen
  - zig: $\leqslant 1 + 3 \cdot (r'(\text{n}) - r(\text{n}))$
  - zig-zig und zig-zag: $\leqslant 3 \cdot (r'(\text{n}) - r(\text{n}))$

  (n: Knoten aus der Abbildung $\quad r(\text{n})$: Rang von n vor Rotation
  $r'(\text{n})$: Rang von n nach Rotation)

Auf dem Weg zum Zugriffs-Lemma

Lemma (Zugriffs-Lemma)

amortisierte Kosten von splay(root, x) sind $\leqslant 3 \cdot (r(\text{root}) - r(x)) + 1$

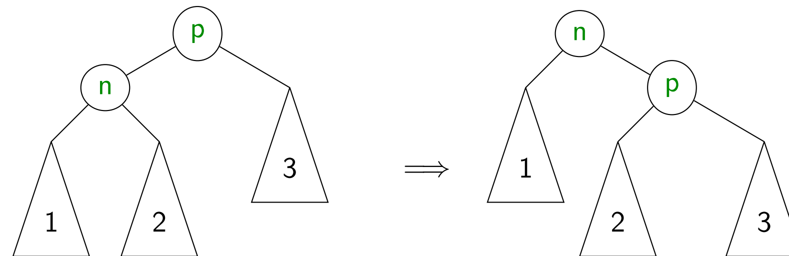
Fall 2: Knoten n mit Schlüssel x ist nicht die Wurzel

- amortisierten Kosten der Rotationen
  - zig: $\leqslant 1 + 3 \cdot (r'(\text{n}) - r(\text{n}))$
  - zig-zig und zig-zag: $\leqslant 3 \cdot (r'(\text{n}) - r(\text{n}))$

$\Rightarrow$ amortisierte Kosten von splay(root, x)

	$3 \cdot (r^{(1)}(\text{n}) - r(\text{n}))$	(zig-zig oder zig-zag)
$+$	$3 \cdot (r^{(2)}(\text{n}) - r^{(1)}(\text{n}))$	(zig-zig oder zig-zag)
$+$	$3 \cdot (r^{(3)}(\text{n}) - r^{(2)}(\text{n}))$	(zig-zig oder zig-zag)
$+$	$\ldots$	
$+$	$3 \cdot (r^{(k)}(\text{n}) - r^{(k-1)}(\text{n}))(+1)$	(zig oder zig-zig oder zig-zag)
$\leqslant$	$3 \cdot (r^{(k)}(\text{n}) - r(\text{n})) + 1$	
$=$	$3 \cdot (r^{(k)}(\text{root}) - r(\text{n})) + 1$	(da n zur Wurzel geworden ist)
$=$	$3 \cdot (r(\text{root}) - r(\text{x})) + 1$	(da Anzahl Knoten unverändert)

amortisierte Kosten von zig $\leqslant 1 + 3 \cdot (r'(\text{n}) - r(\text{n}))$



- reale Kosten: 1
- $G_{nachher} - G_{vorher} = r'(\text{p}) - r(\text{p}) + r'(\text{n}) - r(\text{n})$

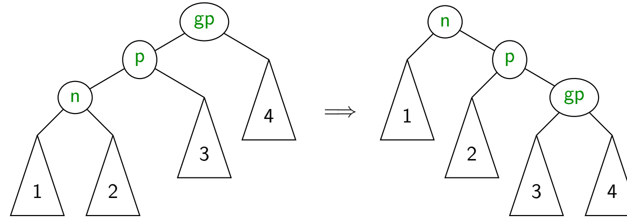
$\Rightarrow$ amortisierte Kosten $= 1 + r'(\text{p}) - \underline{r(\text{p})} + \underline{r'(\text{n})} - r(\text{n})$

$= 1 + \underline{r'(\text{p})} - r(\text{n})$

$\leqslant 1 + \underline{r'(\text{n}) - r(\text{n})}$

$\leqslant 1 + 3 \cdot (r'(\text{n}) - r(\text{n}))$

amortisierte Kosten von zig-zig $\leqslant 3 \cdot (r'(n) - r(n))$

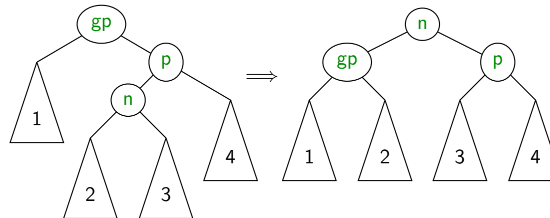


- amortisierte Kosten $= 2 + r'(gp) - \underline{r(gp)} + r'(p) - r(p) + \underline{r'(n)} - r(n)$

$= 2 + r'(gp) + \underline{r'(p)} - r(p) - r(n)$
$\leqslant 2 + r'(gp) + r'(n) - \underline{r(p)} - r(n)$
$\leqslant 2 + r'(gp) + r'(n) - 2 \cdot r(n)$
$= 2 + \underline{r(n)} + \underline{r'(gp)} + r'(n) - 3 \cdot r(n)$
$= \underline{2 + log_2(s(n)) + log_2(s'(gp))} + r'(n) - 3 \cdot r(n)$
$\leqslant 2 \cdot \underline{log_2(s'(n))} + r'(n) - 3 \cdot r(n)$
$= 3 \cdot (r'(n) - r(n))$

- Hilfslemma: $a + b \leqslant c \Rightarrow 2 + log_2(a) + log_2(b) \leqslant 2 \cdot log_2(c)$

amortisierte Kosten von zig-zag $\leqslant 3 \cdot (r'(n) - r(n))$



- amortisierte Kosten $= 2 + r'(gp) - \underline{r(gp)} + r'(p) - r(p) + \underline{r'(n)} - r(n)$

$= 2 + r'(gp) + r'(p) - \underline{r(p)} - r(n)$
$\leqslant 2 + \underline{r'(gp)} + \underline{r'(p)} - 2 \cdot r(n)$
$= \underline{2 + log_2(s'(gp)) + log_2(s'(p))} - 2 \cdot r(n)$
$\leqslant 2 \cdot \underline{log_2(s'(n))} - 2 \cdot r(n)$
$= 2 \cdot \underline{(r'(n) - r(n))}$
$\leqslant 3 \cdot (r'(n) - r(n))$

- Hilfslemma: $a + b \leqslant c \Rightarrow 2 + log_2(a) + log_2(b) \leqslant 2 \cdot log_2(c)$

amortisierte Kosten der Wörterbuch-Operationen

- bislang bewiesen: amortisierte Kosten von splay: $\mathcal{O}(log(N))$
- zeige nun, dass Suche, Einfügen und Löschen diese Komplexität haben (für Bäume mit maximal N Einträgen)

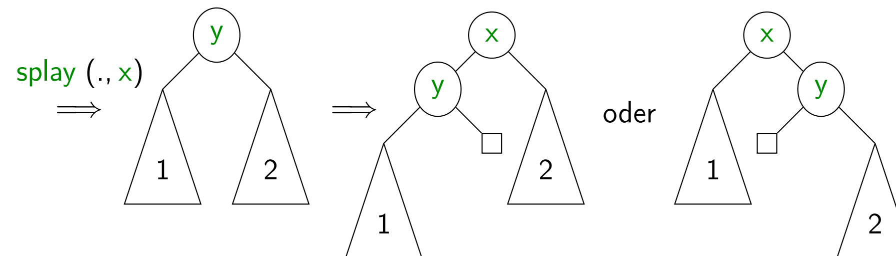
Suchen: splay(root,x) und vergleiche x mit Schlüssel in Wurzel

$\Rightarrow$ amortisierte Kosten von Suchen

$=$ amortisierte Kosten von splay und Vergleich

$= \mathcal{O}(log(N)) + \mathcal{O}(1) = \mathcal{O}(log(N))$

amortisierte Kosten von Einfügen

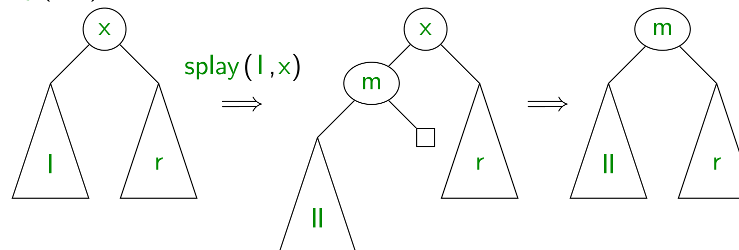


- erhalte drei Bäume:
  - t_1 vor Einfügen
  - t_2 nach splay
  - t_3 nach Einfügen
- amortisierte Kosten = am. Kosten $t_1 \Rightarrow t_2$ + am. Kosten $t_2 \Rightarrow t_3$
- am. Kosten $t_1 \Rightarrow t_2$ = am. Kosten splay $= log(N)$
- am. Kosten $t_2 \Rightarrow t_3 = 1 + G(t_3) - G(t_2)$
  $\leqslant 1 + r(x) = 1 + log_2(s(x)) \leqslant 1 + log_2(N) = \mathcal{O}(log(N))$

amortisierte Kosten von Löschen

Löschen von x: splay(root,x) und lösche Wurzel-Knoten, falls dieser Schlüssel x enthält

- Fall 1: Wurzel hat keinen linken Teilbaum ⇒ mache rechten Teilbaum zur Wurzel
- Fall 2: Wurzel hat linken Teilbaum l
  - splay(l,x)



- in beiden Fällen: am. Kosten von $\leqslant 2 \cdot$ am. Kosten splay $= \mathcal{O}(log(N))$ (das eigentliche Löschen nach den Splays senkt sogar die am. Kosten, da Guthaben abgehoben wird: $G_{nachher} < G_{vorher}$)

amortisierte Kosten von Wörterbuch-Operationen

- jede der Operationen Suchen, Einfügen und Löschen hat amortisierte Kosten von $\mathcal{O}(log(N))$, falls Baum maximal N Einträge hat

⇒ am. Kosten einer Folge von m WB-Operationen: $\mathcal{O}(m \cdot log(N))$, falls zu keinem Zeitpunkt der Baum mehr als N Einträge hat

⇒ am. Kosten einer Folge von m WB-Operationen: $\mathcal{O}(m \cdot log(N + m))$, falls der initiale Baum N Einträge hat

- Erinnerung: reale Kosten = amortisiere Kosten + $G_{vorher} - G_{nachher}$

⇒ reale Kosten einer Folge von m WB-Operationen, falls der initiale Baum N Einträge hat

$$\mathcal{O}(m \cdot log(N + m) + G_{vorher}) = \mathcal{O}(m \cdot log(N + m) + N \cdot log(N))$$

⇒ wenn $m \geqslant N$ (z.B. falls initialer Baum leer): reale Kosten $\mathcal{O}(m \cdot log(m))$

⇒ wenn $m \geqslant N$ und Baum nie mehr als N Einträge hat: reale Kosten $\mathcal{O}(m \cdot log(N))$

Zusammenfassung Splay-Bäume und amortisierte Analyse

- Splay-Bäume benötigen nur eine komplexe Operation: splay

⇒ einfach zu implementieren

- optimaler Speicherverbrauch: keinerlei Zusatzinformation in Knoten
- schwankende Zugriffskosten zwischen $\mathcal{O}(1)$ und $\mathcal{O}(n)$, aber im Mittel: $\mathcal{O}(log(n))$
- Hilfsmittel: amortisierte Analyse
  - amortisierte Kosten = Kosten + $\text{Guthaben}_{\text{nachher}}$ - $\text{Guthaben}_{\text{vorher}}$
  - amortisierte Kosten schwanken nicht so sehr
  - Analyse der amortisierten Kosten + Analyse der Guthaben-Differenz

  ⇒ erhalte Aussagen über reale Kosten
- selbst-organisierend: oft geforderte Schlüssel wandern in oberen Bereich des Baums

Übersicht

- Hashverfahren
  - Grundlagen
  - Hashverfahren mit Verkettung
  - Offene Adressierung
  - Wahl der Hashfunktion

Übersicht

- Hashverfahren
  - Grundlagen
  - Hashverfahren mit Verkettung
  - Offene Adressierung
  - Wahl der Hashfunktion

Motivation

- bislang: Wörterbuch-Zugriffe bestenfalls $\mathcal{O}(log(n))$
  (mittels balancierter und sortierter Bäume)
  - $\Rightarrow$ bei großen Datenmengen und vielen Zugriffen evtl. zu langsam
  - $\Rightarrow$ Ordnung auf Schlüsseln notwendig
- Wunsch: $\mathcal{O}(1)$, keine Ordnung erforderlich
- Realisierung: Hashverfahren

Idee

- zur Speicherung: Array $a[0] \dots a[m-1]$
- Schlüsselmengen:
  - $\mathcal{K}$: alle möglichen Schlüssel ($\approx$ Datentyp; z.B. Zahlen, Strings, ...)
  - $K \subseteq \mathcal{K}$: Schlüssel für konkretes Wörterbuch (z.B. Namen in Innsbruck)
- Annahme: Hashfunktion $h : \mathcal{K} \to \{0, \dots, m-1\}$
- Speichere Eintrag mit Schlüssel k an Stelle $h(k)$ im Array ($h(k)$ ist Hashadresse)
- $k \neq k'$ sind Synonym gdw. $h(k) = h(k')$
- falls keine Synonyme in K (Wahrscheinl. $\leqslant 50\%$, wenn $|K| \geqslant \sqrt{\frac{\pi m}{2}}$)

$\Rightarrow$ jeder Eintrag kann an Hashadresse gespeichert werden

- falls Synonyme $k, k' \in K$ (wahrscheinlicher Fall): Adresskollision

$\Rightarrow$ Spezialbehandlung erforderlich

- Gutes Hashverfahren
  - minimiert Kollisionen durch geschickte Wahl von Hashfunktion
  - löst Kollisionen geschickt auf

Hashfunktionen in Java

- für Hashverfahren benötige 2 Methoden
  - Gleichheit von Elementen: **boolean** equals(Object other)
  - Hashfunktion (abhängig von Array-Größe m)

  $\Rightarrow$ Aufspaltung in Java
    1. **int** hashCode() liefert Hashwert für Objekt (unabhängig von m)
    2. Hashverfahren bildet dann aus hashCode() mit m die Hashfunktion

    $\Rightarrow$ Beispiel: h(k) = k.hashCode() % m
- equals und hashCode sind Methoden der Klasse Object, sind also immer verfügbar
- wichtig: equals und hashCode müssen konsistent sein

  k1.equals(k2) $\Rightarrow$ k1.hashCode() == k2.hashCode()

Beispiel: Strings, ignoriere Groß/Klein-Schreibung

```
public class MyString {
   String s;
   public MyString(String s) {
      this.s = s;
   }
   public int hashCode() {
      return s.toLowerCase().hashCode();
   }
   public boolean equals(Object other) {
      if (other instanceof MyString) {
         return
            ((MyString)other).s.equalsIgnoreCase(this.s);
      } else {
         return false;
      }
   }
   public String toString() {
      return s;
   }
}
```


Berechnung von Hashwerten

- Hashverfahren rufen hashCode häufig auf

⇒ Effizienz von hashCode ist wichtig

⇒ Idee: merke Hashwert statt immer neu zu berechnen
  - Alternative 1: berechne Hashwert direkt im Konstruktor

  ⇒ Hashwert wird immer berechnet, auch wenn Objekt nicht für Hashverfahren genutzt wird
  - Alternative 2: berechne Hashwert bei erster Anfrage

  ⇒ geringer Overhead, da jedesmal noch überprüft werden muss, ob Hashwert gültig ist

Beispiel: Strings, Alternative 1

```
public class MyString {
    String s;
    int hashCode;
    public MyString(String s) {
        this.s = s;
        this.hashCode = s.toLowerCase().hashCode();
    }
    public int hashCode() {
        return this.hashCode;
    }
    ...
}
```


Beispiel: Strings, Alternative 2

```
public class MyString {
    String s;
    int hashCode;
    boolean hashValid;
    public MyString(String s) {
        this.s = s;
        this.hashValid = false;
    }
    public int hashCode() {
        if (!hashValid) {
            this.hashCode = s.toLowerCase().hashCode();
            this.hashValid = true;
        }
        return this.hashCode;
    }
    ...
}
```

Hashfunktion und Hashwerte

- Java:
  - Hashfunktion h indirekt über hashCode und Größe m gegeben
  - Gleichheit kann selbst mittels equals definiert werden
- oft auf Folien:
  - h direkt definiert (für gegebenes m)
  - Schlüssel sind Zahlen (Gleichheit = gleiche Zahl)

$\Rightarrow$ einfachere Präsentation

Übersicht

- Hashverfahren
  - Grundlagen
  - Hashverfahren mit Verkettung
  - Offene Adressierung
  - Wahl der Hashfunktion

Hashverfahren mit Verkettung – open hashing

- speichere an jeder Array-Position (verkettete) Liste von Einträgen
- $a[i]$ speichert alle Einträge mit $h(k) = i$

⇒ Suchen von Eintrag zu Schlüssel k
- durchlaufe Liste $a[h(k)]$ um nach Eintrag $(k, data)$ zu suchen

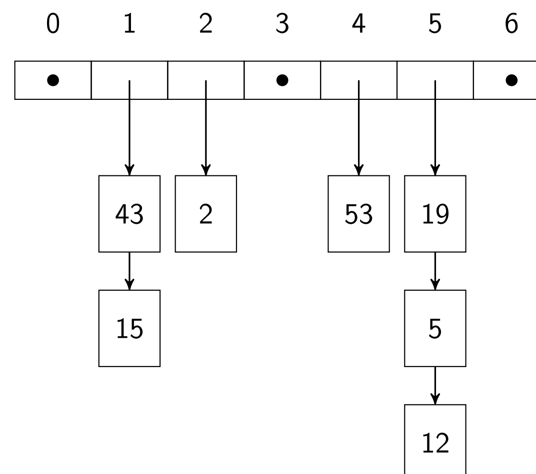
⇒ 2 Fälle
- Suche erfolgreich ⇒ gebe *data* zurück
- Suche erfolglos ⇒ gebe **null** zurück

⇒ Löschen von Eintrag zu Schlüssel k
- durchlaufe Liste $a[h(k)]$ um nach Eintrag $(k, data)$ zu suchen und lösche diesen, wenn vorhanden

⇒ Einfügen von Eintrag $(k, data)$
- durchlaufe Liste $a[h(k)]$ um nach Eintrag $(k, data')$ zu suchen

⇒ 2 Fälle
- Suche erfolgreich ⇒ überschreibe $data'$ mit $data$
- Suche erfolglos ⇒ füge $(k, data)$ in $a[h(k)]$ ein

Einfügen von 12,53,5,15,12,2,19,43, $h(k) = k \% 7$



In Java

```
class Node<K,D> {

    K key;
    D data;
    Node<K,D> next;

    Node(K key, D data, Node<K,D> next) {
        this.key = key;
        this.data = data;
        this.next = next;
    }

}
```


Konstruktor und Suchen

```
public class LinkedHashArray<K,D> {
    Node<K,D>[] array;
    public LinkedHashArray(int capacity) {
        this.array = new Node[capacity];
    }
    int h(K key) {
        int hk = key.hashCode() % array.length;
        return hk < 0 ? hk + array.length : hk;
    }
    public D get(K key) {
        Node<K,D> node = array[h(key)];
        while (node != null) {
            if (node.key.equals(key)) {
                return node.data;
            }
            node = node.next;
        }
        return null;
    }
```

Einfügen

```
public void put(K key, D data) {
    int hk = h(key);
    Node<K,D> node = array[hk];
    while (node != null) {
        if (node.key.equals(key)) {
            node.data = data;
            return;
        } else {
            node = node.next;
        }
    }
    array[hk] = new Node<K,D>(key, data, array[hk]);
}
```


Analyse

- Suchen, Einfügen und Löschen: Komplexität entspricht der Suche
- unterscheide 2 Fälle (wenn schon n Einträge vorhanden sind)
  - C_n: Erwartungswert von # betrachteter Elemente in erfolgreicher Suche
  - C'_n: Erwartungswert von # betrachteter Elemente in erfolgloser Suche
- Annahme: h verteilt die Schlüssel K gleichmässig auf $0, \ldots, m-1$

⇒ Wahrscheinlichkeit für $h(k) = j$ ist stets $\frac{1}{m}$ unabhängig von j

- Analyse von C'_n:
  - C'_n = erwartete Listenlänge = $\frac{n}{m}$
  - Wert $\alpha = \frac{n}{m}$ nennt man auch Belegungsfaktor
- Analyse von C_n:
  - da Suche erfolgreich, wird im Schnitt nur zur Hälfte der Liste gelaufen
  - es wird aber mindestens mit einem Element verglichen

  ⇒ $C'_n \approx 1 + \frac{\text{erwartete Listenlänge}}{2} = 1 + \frac{\alpha}{2}$

⇒ C_n und $C_{n'}$ sind unabhängig von n (aber abhängig von α)

⇒ gleiche Zugriffskosten bei 9 Einträgen im Array der Länge 10, und bei 900000 im Array der Länge 1000000

Folgerungen

Wahl von m wichtig

- $m \gg n$

$\Rightarrow$ Platzverschwendung

- $m \ll n$

$\Rightarrow$ $\alpha \approx n \Rightarrow \mathcal{O}(n)$

- $m \approx n$

$\Rightarrow$ $\alpha \approx 1 \Rightarrow \mathcal{O}(1)$, keine Platzverschwendung

Übersicht

- Hashverfahren
  - Grundlagen
  - Hashverfahren mit Verkettung
  - Offene Adressierung
  - Wahl der Hashfunktion

Offene Adressierung – open addressing – closed hashing

- bisher: Array-Einträge = Listen
- nun: pro Array-Position maximal ein Datensatz

$\Rightarrow$ alle Daten im Array $\Rightarrow$ Name: closed hashing
  - $\Rightarrow$ Array von Größe m kann maximal $n = m$ Einträge speichern
  - $\Rightarrow$ $\alpha \leqslant 1$
  - $\Rightarrow$ neue Art der Kollisionsbehandlung erforderlich

- Idee: falls Platz $h(k)$ belegt und $\alpha < 1$, suche anderen freien Platz

$\Rightarrow$ Sondierungsfolge: Liste von möglichen Plätzen zur Speicherung von k
  - keine eindeutige Adresse mehr $\Rightarrow$ Name: offene Adressierung
  - eigene Sondierungsfolge der Länge m für jeden Schlüssel
  - $\Rightarrow$ Folge $s(0, k), s(1, k), \ldots, s(m-1, k)$
  - freie Plätze werden gefunden: jede Position vorhanden in Folge

$$(h(k)+s(0,k)) \,\%\, m, (h(k)+s(1,k)) \,\%\, m, \ldots, (h(k)+s(m-1,k)) \,\%\, m$$

- Ziel: beim Einfügen möglichst früh in Positionsfolge freien Platz finden

$\Rightarrow$ Wahl der Sondierungsfolge beeinflusst Güte des Hashverfahren

Wörterbuch-Operationen

Einfügen

- Suche nach Schlüssel an Positionen $h(k) + s(i, k)$ für $i = 0, 1, \ldots$
- Abbruch, falls Schlüssel gefunden $\Rightarrow$ Daten überschreiben
- Abbruch, falls freier Platz gefunden $\Rightarrow$ erzeuge dort neuen Eintrag

Suchen

- Suche nach Schlüssel an Positionen $h(k) + s(i, k)$ für $i = 0, 1, \ldots$
- Abbruch, falls Schlüssel gefunden $\Rightarrow$ Daten zurückgeben
- Abbruch, falls freier Platz gefunden $\Rightarrow$ Schlüssel nicht vorhanden

Löschen

- Suche nach Schlüssel an Positionen $h(k) + s(i, k)$ für $i = 0, 1, \ldots$
- Abbruch, falls Schlüssel gefunden $\Rightarrow$ Eintrag löschen
- Abbruch, falls freier Platz gefunden $\Rightarrow$ Schlüssel nicht vorhanden
- Löschen erzeugt Löcher in Sondierungsfolgen $\Rightarrow$ spezielle Markierung erforderlich: unterscheide FREE, OCCUPIED, REMOVED

Lineares Sondieren

- einfachste Sondierungsfolge: $s(i, k) = i$

⇒ für Schlüssel k erhalte Folge

$$h(k), h(k)+1, h(k)+2, \ldots, m-1, 0, 1, \ldots, h(k)-1$$

oder äquivalent

$$h(k), (h(k)+1) \,\%\, m, (h(k)+2) \,\%\, m, \ldots, (h(k)+m-1) \,\%\, m$$

Beispiel: $h(k) = k \,\%\, 11$, lineares Sondieren

Operationsfolge:
E4, E29, E42, E17, E65, E32, E98, E12, E24, L32, L65, S98, E44

Offene Adressierung: Code

```
public class OpenAddressingHashArray<K,D> {
    static final byte FREE = 0;
    static final byte OCCUPIED = 1;
    static final byte REMOVED = 2;
    K[] keys;
    D[] data;
    byte[] status;
```

Alternative: statt K[] und D[] nutze Entry<K,D>[]

Offene Adressierung: Code

```
    public D get(K key) {
        int hk = h(key);
        for (int i=0; i<keys.length; i++) {
            int pos = (hk + s(key, i)) % keys.length;
            switch (status[pos]) {
            case FREE:
                return null;
            case OCCUPIED:
                if (this.keys[pos].equals(key)) {
                    return this.data[pos];
                }
            }
        }
        return null;
    }
```

Offene Adressierung: Fehlerhafter Code

```
public void put(K key, D data) {
    int hk = h(key);
    for (int i=0; i<keys.length; i++) {
        int pos = (hk + s(key, i)) % keys.length;
        switch (status[pos]) {
        case REMOVED:
        case FREE:
            status[pos] = OCCUPIED;
            this.keys[pos] = key;
            this.data[pos] = data;
            return;
        case OCCUPIED:
            if (this.keys[pos].equals(key)) {
                this.data[pos] = data;
                return;
            }
        }
    }
    throw new RuntimeException("no space left");
}
```


Lineares Sondieren: Eigenschaften

+ einfach zu implementieren
− primäres Clustering:
  - betrachte Array

0	1	2	3	4	5	6
			3	53	17	

  ⇒ W. für Einfügen an Position 6: $\frac{4}{7}$, für Positionen 0,1,2: je $\frac{1}{7}$
  ⇒ lange belegte Blöcke wachsen stärker als Kurze
  ⇒ Einfügen findet erst spät freien Platz (besonders wenn α nahe 1)
  ⇒ Suche nach diesen Elementen auch ineffizient

- genauere Analyse für lineares Sondieren:

$$C_n \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad C'_n \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

α	C_n	C'_n
0, 5	1, 5	2, 5
0, 9	5, 5	20, 5
0, 95	10, 5	200, 5

Verbesserung: Quadratisches Sondieren

- Sondierungsfolge $s(i,k) = \lceil \frac{i}{2} \rceil^2 \cdot (-1)^{i+1}$

$$0, 1, -1, 4, -4, 9, -9, \ldots$$

- Problem: werden alle Positionen getroffen

⇒ ja, falls m Primzahl ist, und $m-3$ durch 4 teilbar ($m = 7, 11, 19, \ldots$)

+ kein primäres Clustering

− sekundäres Clustering:
  - Beobachtung: $s(i,k)$ unabhängig von k
  - ⇒ Synonyme k und k' durchlaufen gleiche Sondierungsfolge
  - ⇒ Synonyme behindern sich auch in Ausweichplätzen

- genauere Analyse für quadratisches Sondieren:

$$C_n \approx 1 + log\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \qquad C'_n \approx \frac{1}{1-\alpha} - \alpha + log\left(\frac{1}{1-\alpha}\right)$$

α	C_n	C'_n
0, 5	1, 44	2, 19
0, 9	2, 85	11, 40
0, 95	3, 52	22, 05

Beispiel: $h(k) = k$ % 11, quadratisches Sondieren

Operationsfolge:
E4, E29, E42, E17, E65, E32, E98, E12, E24, L32, L65, S98, E44

Verbesserung: Double-Hashing

- lineares Sondieren: primäres und sekundäres Clustering
- quadratisches Sondieren: sekundäres Clustering
- Grund für sekundäres Clustering: $s(i, k)$ ignoriert k

$\Rightarrow$ verbesserte Idee: Sondierungsfolge hängt von k ab, so dass Synonyme k und k' unterschiedliche Sondierungsfolgen bekommen

- Ideal-Fall: uniformes Sondieren: $s(i, k)$ ist beliebige Permutation von $0, \ldots, m-1$, so dass
  - Wahrscheinlichkeit für jede Permutationen gleich groß ist bzgl. K
  - $\Rightarrow$ insbesondere für Synonyme k, k' ist Wahl der Permutation unabhängig
- Problem: Implementierung / Realisierung von uniformen Sondieren

$\Rightarrow$ Double-Hashing:

- Idee: nutze zweite Hashfunktion h' zur Erzeugung der Permutation

$$s(i, k) = i \cdot h'(k)$$

Double Hashing: $s(i, k) = i \cdot h'(k)$

- 1. Anforderung: keine sekundäre Häufung

$\Rightarrow$ h und h' sollten unabhängig sein:

$$p[h(k) = h(k') \text{ und } h'(k) = h'(k')] = p[h(k) = h(k')] \cdot p[h'(k) = h'(k')]$$

($p[\ldots]$: Wahrscheinlichkeit, dass … eintrifft)

$\Rightarrow$ für Synonyme k, k' bzgl. h gilt: die Wahrscheinlichkeit, dass k, k' auch Synonyme bzgl. h' sind, ist $\frac{1}{m'}$, wobei h' insgesamt m' verschiedene Werte annimmt

- 2. Anforderung: $s(0, k), \ldots, s(m-1, k)$ liefert Permutation von $0, \ldots, m-1$ (modulo m)

$\Rightarrow$ $h'(k) \neq 0$

$\Rightarrow$ $h'(k)$ darf kein Vielfaches von m sein

$\Rightarrow$ Wähle $h(k) = k \ \% \ m$ und $h'(k) = 1 + (k \ \% \ (m-2))$ $\Rightarrow$ $h'(k) \in \{1, \ldots, m-1\}$

$\Rightarrow$ für diese Wahl von h und h' gilt: wenn m Primzahl ist, sind h und h' unabhängig $\Rightarrow$ 1. Anforderung auch erfüllt

Beispiel: $h(k) = k \% 11$, $h'(k) = 1 + k \% 9$

Operationsfolge:
E4, E29, E42, E17, E65, E32, E98, E12, E24, L32, L65, S98, E44

Uniformes Sondieren und Double Hashing

- genauere Analyse für uniformes Sondieren:

$$C_n \approx \frac{1}{\alpha} \cdot log\left(\frac{1}{1-\alpha}\right) \qquad C'_n \approx \frac{1}{1-\alpha}$$

α	C_n	C'_n
0,5	1,39	2
0,9	2,56	10
0,95	3,15	20

- falls h und h' unabhängig sind, gelten ungefähr die gleichen Resultate auch für Double Hashing

Zusammenfassung: Hashverfahren

- Hashverfahren mit Verkettung
  - feste Adressen, Array beinhaltet Listen
- offene Adressierung
  - finde freien Platz über Sondierungsfolgen
    - lineares Sondieren: $s(i,k) = i$

    $\Rightarrow$ primäres und sekundäres Clustering
    - quadratisches Sondieren: $s(i,k) = \lceil \frac{i}{2} \rceil^2 \cdot (-1)^i$

    $\Rightarrow$ kein primäres, aber sekundäres Clustering

    $\Rightarrow$ $h(k) = k \,\%\, m$ für Primzahl m, so dass $m-3$ durch 4 teilbar
    - Double Hashing: $s(i,k) = i \cdot (1 + k \,\%\, (m-2))$

    $\Rightarrow$ weder primäres noch sekundäres Clustering

    $\Rightarrow$ $h(k) = k \,\%\, m$ für Primzahl m
  - Kosten von WB-Operationen groß, wenn $\alpha \approx 1$
  - jede Array-Position speichert max. einen Eintrag $\Rightarrow \alpha \leqslant 1$

  $\Rightarrow$ dynamische Anpassung von m notwendig, um $\alpha \approx 1$ zu verhindern
- Kosten von Wörterbuch-Operationen: abhängig von α, nicht von n

Zusammenfassung: Hashverfahren

Nicht behandelt, aber möglich

- Iteratoren
  - ohne Optimierung: Gesamtdurchlauf in $\mathcal{O}(n+m)$

  $\Rightarrow$ wenn $n \ll m$ ist dies zu teuer

  $\Rightarrow$ Optimierung (speichere Zusatzinformationen): Gesamtdurchlauf in $\mathcal{O}(n)$
- dynamische Anpassung der Array-Größe
  - für verkettete Hashverfahren, wenn α zu groß wird
  - für offene Addressierung, wenn α nahe 1 wird
  - aber auch wenn α zu klein wird, um Platzverschwendung zu vermeiden
  - einfaches Verfahren:
    - erzeuge neues Array mit angepasster Größe

    $\Rightarrow$ ändert Hashfunktion $\Rightarrow$ Positionen können nicht übernommen werden
    - füge alle Daten von altem Array mittels put in neues Array ein

    amortisierte Kosten: $\mathcal{O}(1)$

Übersicht

- Hashverfahren
  - Grundlagen
  - Hashverfahren mit Verkettung
  - Offene Adressierung
  - Wahl der Hashfunktion

Güte einer Hashfunktion

- üblicherweise: h.key = key.hashCode() % m wobei m = array.length
- bislang: key war Integer, und key.hashCode() = key

⇒ zentrale Frage: wie sollte man hashCode() für andere Datentypen definieren?

- `int hashCode() { return 0; }` ist einfach, führt aber immer zu Kollisionen

⇒ nicht jede Hashfunktion ist gut

- Erinnerung: K = genutzte Schlüssel, $\mathcal{K}$ = Menge aller Schlüssel
- Ziel: Schlüssel K werden gleichmäßig auf alle Integer-Werte abgebildet
- normalerweise ist K nicht bekannt

⇒ bilde $\mathcal{K}$ gleichmäßig auf Integer ab

⇒ versuche auch zu erreichen, dass wahrscheinlich für jedes $K \subseteq \mathcal{K}$ gleichmäßige Verteilung auftritt

Mögliche Probleme

- Ziel: Schlüssel K werden gleichmäßig auf $\{0, \ldots, m-1\}$ abgebildet
- Problem: Auch wenn h alle Schlüssel $\mathcal{K}$ gleichmäßig verteilt, gilt dies nicht für K
- Beispiel: 64-bit long Zahlen auf 32-bit Hashwerte
  - falls h(long_number) = 32 führende Bits $\Rightarrow$ alle kleine Zahlen führen auf h(klein) = 0
  - falls h(long_number) = 32 letzte Bits $\Rightarrow$ ebenso schlecht: vordere Bits werden ignoriert

  $\Rightarrow$ gesamte Daten sollten in den Hashwert einfließen
- Beispiel: Variablen-Namen auf 32-bit Hashwerte
  - falls h(var_name) = Summe aller Charakter

  $\Rightarrow$ oft benutzte Namen führen zu Kollisionen
  Beispiel: 2 Vektoren im drei-dimensionalen Raum (x1,x2,x3,y1,y2,y3) haben bereits 2 Kollisionen

Universelles Hashing

- Problem: wenn h festliegt, gibt es immer Schlüsselmenge $K \subseteq \mathcal{K}$ mit vielen Kollisionen

$\Rightarrow$ es gibt schlechte Eingaben K auf denen Hashverfahren langsam werden

$\Rightarrow$ Lösung analog randomisierter Quick-Sort:
wähle Hashfunktion h zufällig (aus Menge von Hashfunktionen $\mathcal{H}$),
so dass es keine schlechte Schlüsselmenge K mehr gibt

- Menge von Hashfunktionen $\mathcal{H}$ heißt universell gdw.

$$\forall k, k' \in \mathcal{K}, k \neq k' : \quad \frac{|\{h \in \mathcal{H} \mid h(k) = h(k')\}|}{|\mathcal{H}|} \leqslant \frac{1}{m}$$

d.h., für alle Schlüsselpaare k, k' führt höchtens der m-te Teil aller Hashfunktionen in $\mathcal{H}$ zur Kollision

Universelles Hashing

- Kollisionsfunktion δ

$$\delta(k, k', h) = \begin{cases} 1, & h(k) = h(k') \wedge k \neq k' \\ 0, & \text{sonst} \end{cases}$$

- Ausweitung auf Menge von Schlüsseln:

$$\delta(k, K, h) = \sum_{k' \in K} \delta(k, k', h)$$

- Ausweitung auf Menge von Hashfunktionen:

$$\delta(k, k', \mathcal{H}) = \sum_{h \in \mathcal{H}} \delta(k, k', h)$$

- $\mathcal{H}$ ist universell gdw. $\delta(k, k', \mathcal{H}) \leqslant \frac{|\mathcal{H}|}{m}$ für alle k, k'

Güte von universellem Hashing

- Ziel: Wahrscheinlichkeit für Kollisionen gering für beliebiges $K \subseteq \mathcal{K}$.
- Berechne dazu Erwartungswert von $E[\delta(k, K, h)]$ gemittelt über $h \in \mathcal{H}$ (Anzahl der Kollisionen von k mit Schlüsseln in K)

$$\begin{aligned} E[\delta(k, K, h)] &= \frac{\sum_{h \in \mathcal{H}} \delta(k, K, h)}{|\mathcal{H}|} \\ &= \frac{\sum_{h \in \mathcal{H}} \sum_{k' \in K} \delta(k, k', h)}{|\mathcal{H}|} \\ &= \frac{\sum_{k' \in K} \sum_{h \in \mathcal{H}} \delta(k, k', h)}{|\mathcal{H}|} \\ &= \frac{\sum_{k' \in K} \delta(k, k', H)}{|\mathcal{H}|} \\ &\leqslant \frac{\sum_{k' \in K} \frac{|\mathcal{H}|}{m}}{|\mathcal{H}|} = \frac{|K|}{m} \end{aligned}$$

$\Rightarrow$ für jedes K werden Schlüssel gleichmäßig auf $\{0, \ldots, m-1\}$ verteilt

Existenz von universellem $\mathcal{H}$

- bislang: $\mathcal{H}$ universell $\Rightarrow$ keine schlechten Schlüsselmengen K
- bislang offen: gibt es so ein $\mathcal{H}$ überhaupt?
- Antwort: ja, mit zwei Einschränkungen
  - Schlüssel sind Zahlen: $\mathcal{K} = \{0, \ldots, p-1\}$
  - p ist Primzahl
- für $a \in \{1, \ldots, p-1\}$ und $b \in \{0, \ldots, p-1\}$ definiere

$$h_{a,b}(k) = ((ak + b) \mod p) \mod m$$

Theorem

Die Menge $\mathcal{H} = \{h_{a,b} \mid 1 \leqslant a < p, 0 \leqslant b < p\}$ ist universell.

Anwendbarkeit des Theorems

$$h_{a,b}(k) = \underbrace{((ak + b) \mod p)}_{\text{hashCode(k)}} \mod \underbrace{m}_{\text{arr.length}}$$

- Vorteil: m kann beliebig gewählt werden

$\Rightarrow$ Hashtabellen brauchen also keine Primzahl als Größe

- 2. Einschränkung: $|\mathcal{K}| = p$ ist Primzahl.
  - oft nicht gegeben
  - $\Rightarrow$ wähle einfach p als etwas größere Primzahl als $|\mathcal{K}|$
  - Anmerkung: p braucht nur einmalig gefunden werden
- 1. Einschränkung: $\mathcal{K}$ sind Zahlen $\{0, \ldots, p-1\}$
  - oft nicht gegeben
  - da Daten Bitstrings sind, interpretiere einfach diese Bitstrings als Zahlen
- Wahl von a und b sollte zufällig sein
  - statische Variablen, die zufällig beim Programm-Start belegt werden

In Java: Beispiel: Long

- `class Long { long value; ... }`
- `static BigInteger p = new BigInteger("27248918038870668403");`
  (generiert durch `BigInteger.probablePrime(65, new Random())`)
- ```
  static Random rnd = new Random();
  static BigInteger a = BigInteger.valueOf(rnd.nextLong()).
                                   add(BigInteger.ONE);
  static BigInteger b = BigInteger.valueOf(rnd.nextLong());
  ```
- ```
  int hashCode() {
    BigInteger val = BigInteger.valueOf(this.value);
    val = a.multiply(val).add(b).mod(p);
    return val.intValue();
  }
  ```


Alternativen

- Universelles Hashing ist realisierbar in Java ...
- ... aber unter Umständen zu teuer

⇒ einfachere Varianten:

1. wähle Primzahl innerhalb **long**, führe alle Operationen auf **long** durch
2. falls immer noch zu teuer, nutze kein universelles Hashing, aber betrachte zumindest alle Bits

⇒ Kombination des langen Bitstrings mittels XOR (^)

⇒ Beispiel Long: `return (int)(this.value ^ (this.value >>> 32));`

Zusammenfassung: Hashfunktionen

- Wahl der Hashfunktion ist wichtig
- Hashfunktion sollte alle Daten berücksichtigen
- einfache Hashfunktionen: mittels XOR
- universelle Menge von Hashfunktionen
  - randomisierte Wahl der Hashfunktion
  - $\Rightarrow$ keine schlechten Schlüsselmengen K

Übersicht

- Optimierungs-Probleme
  - Dynamische Programmierung
  - Greedy-Algorithmen

Optimierungs-Probleme – Anwendungsszenarien

- Beispiele von Optimierungs-Problemen
  - Logistik: optimale Beladung eines LKWs, Schiffs
  - Produktion: optimale Auslastung von Maschinen, kürzeste Zeit für Fertigstellung
  - Navigationssysteme: optimale Wegstrecken
  - DNA-Analyse
  - ...
- Struktur von Optimierungs-Problemen
  - Problemstellung mit vielen Lösungen
  - Jede Lösung hat einen gewissen Wert / Kosten
  - Aufgabe: finde optimale Lösung mit maximalem Wert / minimalen Kosten

Übersicht

- Optimierungs-Probleme
  - Dynamische Programmierung
  - Greedy-Algorithmen

Ein Optimierungs-Problem

- Aufwand der Matrix-Multiplikation $C^{d,d''} = A^{d,d'} \times B^{d',d''}$: $\mathcal{O}(dd'd'')$

$$\forall 1 \leqslant i \leqslant d, 1 \leqslant j \leqslant d'' : c_{ij} = \sum_{k=1\dots d'} a_{ik} b_{kj}$$

- Optimierungs-Problem Ketten-Multiplikation:
- Gegeben Folge von n Matrizen $A_1, \dots, A_n$, Dimension $A_i = d_{i-1} \times d_i$, finde Klammerung, die Gesamtaufwand zur Berechnung von $A_1 \dots A_n$ minimiert
- Beispiel: 4 Matrizen $A_1, \dots, A_4$, Dimensionen $\vec{d} = (100, 20, 1000, 5, 80)$
  Aufwand für $(A_1 A_2)(A_3 A_4)$:
  $\underbrace{100 \cdot 20 \cdot 1000}_{A_1 \times A_2} + \underbrace{100 \cdot 1000 \cdot 80}_{A_1 A_2 \times A_3 A_4} + \underbrace{1000 \cdot 5 \cdot 80}_{A_3 \times A_4} = 10.400.000$

  Aufwand für $A_1((A_2 A_3)A_4)$:
  $\underbrace{100 \cdot 20 \cdot 80}_{A_1 \times A_2 A_3 A_4} + \underbrace{20 \cdot 1000 \cdot 5}_{A_2 \times A_3} + \underbrace{20 \cdot 5 \cdot 80}_{A_2 A_3 \times A_4} = 268.000$

Ketten-Multiplikation

- Dimension von $A_i \dots A_j$: $d_{i-1} \times d_j$

$\Rightarrow$ Aufwand zur Multiplikation von $A_i \dots A_k \times A_{k+1} \dots A_j$: $d_{i-1} d_k d_j$

$\Rightarrow$ Rekursionsgleichung für Gesamtkosten $c(i,j)$ der Berechnung von $A_i \dots A_j$ bei gegebener Klammerung:

$$c(i,j) = \begin{cases} 0, & \text{falls } i = j \\ c(i,k) + c(k+1,j) + d_{i-1} d_k d_j, & \text{falls } (A_i \dots A_k)(A_{k+1} \dots A_j) \end{cases}$$

- Naiver Algorithmus zur Minimierung des Gesamtaufwands:
  - Probiere alle Klammerungen aus
  - Bereche für jede Klammerung $c(1,n)$ und gebe Klammerung mit minimalem Wert aus
- Problem: Anzahl Klammerungen: $\Omega(2^n)$

Auf dem Weg zur Lösung

⇒ Rekursionsgleichung für Gesamtkosten $c(i,j)$ der Berechnung von $A_i \dots A_j$ bei gegebener Klammerung:

$$c(i,j) = \begin{cases} 0, & \text{falls } i = j \\ c(i,k) + c(k+1,j) + d_{i-1}d_k d_j, & \text{falls } (A_i \dots A_k)(A_{k+1} \dots A_j) \end{cases}$$

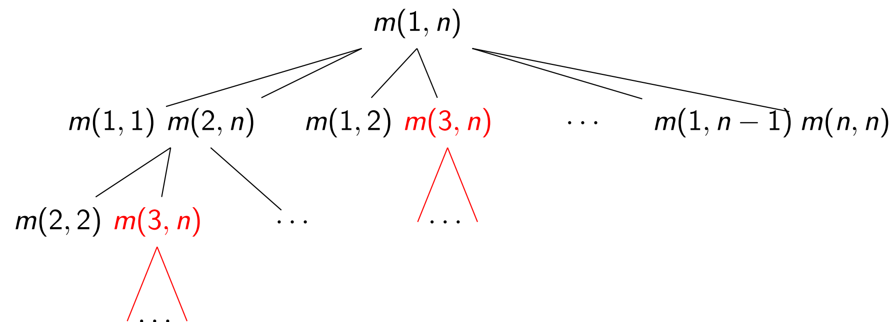
- Bestimme aus Rekursionsgleichung direkt die minimalen Kosten $m(i,j)$

$$m(i,j) = \begin{cases} 0, & \text{falls } i = j \\ \min_{i \leqslant k < j}\{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\}, & \text{falls } i < j \end{cases}$$

- Divide & Conquer (direkte Umsetzung der Rekursionsgleichung) berechnet $m(1,n)$, aber leider in exponentieller Zeit

Probleme von Divide & Conquer

- bisher: Divide & Conquer hat Problem in unabhängige Probleme aufgespalten (ohne Überlappung)
- hier: Divide & Conquer führt zu vielen Überlappungen



⇒ Laufzeit für n Matrizen $\geqslant 2 \cdot$ (Laufzeit für $n-2$ Matrizen)

⇒ exponentielle Laufzeit

Lösung: Dynamische Programmierung

- Idee: Berechne und speichere Werte $m(i,j)$
  (Es gibt nur n^2 viele Paare i,j)
- während Divide & Conquer von oben-nach unten rechnet (top-down), nutze bottom-up Ansatz
- dynamische Programmierung = bottom-up Berechnung + Speicherung der Teilergebnisse
- bottom-up für dieses Beispiel:

$$m(i,j) = \begin{cases} 0, & \text{falls } i = j \\ \min_{i \leqslant k < j}\{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\}, & \text{falls } i < j \end{cases}$$

  - starte mit Werten $m(i,j)$ für die $i = j$ gilt
  - erhöhe Differenz $j - i$ schrittweise
  - $\Rightarrow$ Werte $m(i,k)$ und $m(k+1,j)$ sind bekannt zur Berechnung von $m(i,j)$
  - stoppe Verfahren, sobald $j - i = n - 1$ erreicht und lese $m(1,n)$ ab

Beispiel: $\vec{d} = (100, 20, 1000, 5, 80)$

	j = 1	2	3	4
i = 1	0	2000000	110000	150000
2		0	100000	108000
3			0	400000
4				0

- $j - i = 1$
  - $m(1,2) = \min\{m(1,1) + m(2,2) + d_0 \cdot d_1 \cdot d_2\}$
  - $m(2,3) = \min\{m(2,2) + m(3,3) + d_1 \cdot d_2 \cdot d_3\}$
  - $m(3,4) = \min\{m(3,3) + m(4,4) + d_2 \cdot d_3 \cdot d_4\}$
- $j - i = 2$
  - $m(1,3) = \min\{m(1,1)+m(2,3)+d_0\cdot d_1\cdot d_3, m(1,2)+m(3,3)+d_0\cdot d_2\cdot d_3\}$
  - $m(2,4) = \min\{m(2,2)+m(3,4)+d_1\cdot d_2\cdot d_4, m(2,3)+m(4,4)+d_1\cdot d_3\cdot d_4\}$
- $j - i = 3$
  - $m(1,4) = \min\{m(1,1)+m(2,4)+d_0\cdot d_1\cdot d_4, m(1,2)+m(3,4)+d_0\cdot d_2\cdot d_4, m(1,3) + m(4,4) + d_0 \cdot d_3 \cdot d_4\}$
- Optimale Klammerung: $(A_1(A_2A_3))A_4$

Matrix-Multiplikation

$$m(i,j) = \begin{cases} 0, & \text{falls } i = j \\ \min_{i \leqslant k < j}\{m(i,k) + m(k+1,j) + d_{i-1}d_k d_j\}, & \text{falls } i < j \end{cases}$$

- Rekursionsgleichung berechnet $m(1,n)$
- Problem: Wie bekommt man daraus die Klammerung?
- Lösung: Speichere zusätzlich die Indizes k, die zur Minimierung von $m(i,j)$ führten (in Einträgen $s(i,j)$)

$\Rightarrow$ $A_i \dots A_j = (A_i \dots A_{s(i,j)})(A_{s(i,j)+1} \dots A_j)$ ist optimale Klammerung

Matrix-Multiplikation in Java

```
int n = d.length - 1;
int[][] m = new int[n+1][n+1]; // so m[i,i] = 0
int[][] s = new int[n+1][n+1];
for (int diff=1; diff < n; diff++) { // diff = j-i
  for (int i = 1; i+diff <=n; i++) {
    int j = i + diff;
    int min = Integer.MAX_VALUE;
    int dij = d[i-1]*d[j];
    for (int k = i; k < j; k++) {
      int cost = m[i][k] + m[k+1][j] + dij*d[k];
      if (cost < min) {
        s[i][j] = k;
        min = cost;
      }
    }
    m[i][j] = min;
  }
} // Laufzeit: O(n^3)
```

Dynamische Programmierung

vom Optimierungs-Problem zum Programm

1. Charakterisiere Struktur einer optimalen Lösung
2. Definiere Wert einer optimalen Lösung rekursiv
3. Berechne Wert einer optimalen Lösung (bottom-up)
4. Speichere evtl. zusätzliche Daten, um aus berechneten Daten eine optimale Lösung konstruieren zu können

wann dynamisches Programmieren?

- starkes Indiz: Rekursions-Gleichung in 2. hat Überlappungen
- notwendige Bedingung: Problem hat optimale Teilstruktur:

  optimale Lösung enthält optimale Lösungen für Teilproblem

  Beispiel Matrix: wenn $(A_1(A_2A_3))A_4$ optimal geklammert ist, dann ist auch $A_1(A_2A_3)$ optimal geklammert

1. Charakterisiere Struktur einer optimalen Lösung

Matrix Beispiel

- 1. Idee: Berechne nur optimale Klammerung für $A_i \dots A_n$ (rechte Seite ist fest)

⇒ stelle fest, dass optimale Klammerung von $A_i \dots A_n$ auch mittels $A_i \dots A_k$ und $A_{k+1} \dots A_n$ für $i < k < n$ möglich ist

⇒ benötige optimale Klammerungen für $A_i \dots A_j$

⇒ adaptiere Idee, und versuche Struktur von optimalen Lösungen für Teilproblem $A_i \dots A_j$ zu finden

Auswirkungen

- zu viele Teilprobleme ⇒ zu große Tabellen
- zu wenig Teilprobleme ⇒ Verfahren kann auf benötigte Werte nicht zugreifen
- Beispiel:
  1. Idee: $\mathcal{O}(n)$, aber leider zu wenig Informationen
  adaptierte Idee: $\mathcal{O}(n^2)$, ausreichend Informationen

2. Definiere Wert einer optimalen Lösung

Matrix Beispiel

- Struktur: zerlege $A_i \dots A_j$ in $B = A_i \dots A_k$ und $C = A_{k+1} \dots A_n$, bestimme optimale Teillösungen für B und C, wähle k optimal

$\Rightarrow$ stelle fest, dass man nicht nur die Klammerung von B und C braucht, um k zu bestimmen, sondern vor allem die Kosten

$\Rightarrow$ stelle Rekursionsgleichung für Kosten einer optimalen Klammerung auf

3. Berechne Wert einer optimalen Lösung

- Berechnung bottom-up
  - zentrale Frage: in welcher Reihenfolge werden Werte benötigt?
  - bottom-up Algorithmus muss sicherstellen, dass erforderliche Werte vorhanden
  - Beispiel Matrix: berechne Einträge $m(i, j)$ nach aufsteigender Differenz $j - i$
    und nicht `for(int i = 1..n) { for(int j=i..n) { ... }}`
- Berechnung top-down
  - Reihenfolge der Aufrufe wird automatisch richtig erfolgen
  - aber die Überlappung führt zu exponentieller Laufzeit
- Berechnung top-down mit Memoization
  - rekursiver Ansatz, automatisch richtige Reihenfolge
  - Memoization
    - wenn Teilergebnis vorhanden, speichere dieses in einer Tabelle
    - wenn Ergebnis berechnet werden soll, schaue zuerst in Tabelle, ob es dort schon vorhanden ist $\Rightarrow$ wenn ja, gebe einfach dieses zurück

4. Konstruiere optimale Lösung aus berechneten Daten

- Beispiel Matrix: Werte $m(i,j)$ liefern nicht direkt die Lösung
- zwei Ansätze
  - Zeit-sparend: speichere weitere Hilfsinformationen, die Konstruktion der Lösung erleichtern ($s(i,j)$)

  $\Rightarrow$ $\mathcal{O}(n)$ Zeit zur Konstruktion, $\mathcal{O}(n^2)$ zusätzlicher Platz
  - Platz-sparend: ohne Hilfsinformationen, suche erneut nach einem k, so dass $m(i,j) = m(i,k) + m(k+1,j) + d_{i-1}d_kd_j$

  $\Rightarrow$ $\mathcal{O}(n^2)$ Zeit zur Konstruktion

Beispiel: DNA Analyse

- Untersuche Ähnlichkeit zweier DNA-Sequenzen ACCGAGTTCGGACTTA... und CCATTGCTCCCATTACGG...
- Mögliches Maß für Ähnlichkeit: Länge einer gemeinsamen Teilsequenz

$\Rightarrow$ LCS-Problem (longest common sequence):
  - gegeben zwei Strings s und t
  - bestimme gemeinsame Teilsequenz beider Strings mit maximaler Länge
- Anzahl der Teilsequenzen von s und t: $2^{|s|}$ und $2^{|t|}$

$\Rightarrow$ naiver Ansatz, alle Teilsequenzen zu konstruieren, ist ineffizient
- dynamische Programmierung: $\mathcal{O}(|s| \cdot |t|)$

LCS 1: Bestimmung der Struktur einer optimalen Lösung

LCS 2: Aufstellen einer Rekursiongleichung

LCS 3: Berechnung der Länge einer optimalen Lösung

LCS 4: Konstruktion einer optimalen Lösung

Optimierung der entstehenden Programme

- Umsetzung der Rekursionsgleichung in Programm meist einfach, wenn für jedes Teilresultat eigener Tabelleneintrag existiert
- Oft nicht alle Teilresultate während des gesamten Ablaufs erforderlich

⇒ häufig kann Platzbedarf optimiert werden

Beispiel LCS

- Eingabe: 2 Genom-Sequenzen einer Taufliege, Länge je ≈ 200.000.000

⇒ zusätzlicher Speicherbedarf bei $|s| \cdot |t| \approx 40.000 TB$

- Beobachtung: zur Berechnung von `l[i][.]` werden nur `l[i+i][.]` und `l[i][.]` benötigt

⇒ Benötige nur Speicherplatz der Größe $2 \cdot |t|$: 2 Zeilen der gesamten Tabelle

⇒ insgesamter Speicherbedarf: $4 \cdot 200.000.000 \approx 0.8 GB$

⇒ DNA-Analyse vollständig im Hauptspeicher eines Standard-PCs möglich

Zusammenfassung dynamische Programmierung

- dynamische Programmierung für Optimierungs-Problem einsetzbar, wenn Problem optimale Teilstruktur aufweist
- bottom-up Berechnung verhindert Mehrfachberechnung des gleichen Teilproblems im Falle von Überlappungen
- systematischer Entwurf in 4 Schritten
- Konstruktion einer optimalen Lösung: Platz- oder Zeit-optimiert
- Optimierung der entstehenden Programme oft möglich

Übersicht

- Optimierungs-Probleme
  - Dynamische Programmierung
  - Greedy-Algorithmen

Vorgehen bei Optimierungs-Problemen

- Optimierungs-Problem mit optimaler Teilstruktur

⇒ dynamische Programmierung führt zu Programm

- Eigenschaft der dynamischen Programmierung
  - Speicherplatz intensiv
  - Entscheidung wird aufgrund von (mehreren) optimalen Teillösungen getroffen

  ⇒ großer Aufwand
- neuer Ansatz: Greedy-Algorithmen
  - treffe gierige Entscheidung lokal (ohne Wissen der Güte von Teilproblemen)
  - gierig: wähle lokal beste Möglichkeit bzgl. Optimierungsproblem
  - nach Entscheidung, löse eindeutiges Teilproblem rekursiv
  - top-down Verfahren

  ⇒ effizient, platz-sparend, einfach zu implementieren

Probleme bei Greedy-Algorithmen

- dynamische Programmierung: direkte Umsetzung der Rekursionsgleichung in Code

$\Rightarrow$ Korrektheit leicht einzusehen

- Greedy-Algorithmus: trifft lokale gierige Entscheidung

$\Rightarrow$ Korrektheit fraglich: Liefert Greedy-Algorithmus das Optimum?

- 3 Mögliche Konsequenzen
  - Korrektheit konnte gezeigt werden $\Rightarrow$ nutze Greedy-Algorithmus
  - Korrektheit konnte widerlegt werden $\Rightarrow$ verwerfe Greedy-Algorithmus und nutze dynamische Programmierung
  - Korrektheit konnte widerlegt werden, aber man kann zeigen, dass Resultate vom Greedy-Algorithmus nicht weit vom Optimum abweichen

  $\Rightarrow$ nutze vielleicht trotzdem Greedy-Algorithmus
    - besser ein ungefähres Ergebnis in 1 Stunde als ein genaues in 100 Tagen
    - der Routenplaner sollte auf meinem PDA laufen, dynamische Programmierung hätte zu hohen Speicherbedarf

  $\Rightarrow$ Bereich der approximativen Algorithmen, nicht in dieser Vorlesung

Beispiel: Aktivitäten-Auswahl-Problem

- Gegeben: eine Liste von Aktivitäten $S = a_1 \dots a_n$ repräsentiert durch Startzeiten s_i und Endzeiten f_i, wobei $s_i < f_i$
- alle Aktivitäten benötigen gleich Ressource (Sporthalle, Mietauto, ...)
- Aktivitäten a_i und a_j sind kompatibel gdw. $f_i \leqslant s_j$ oder $f_j \leqslant s_i$
- $A = \{a_{i_1}, \dots, a_{i_k}\} \subseteq S$ ist gültige Auswahl, wenn je zwei Aktivitäten in A kompatibel sind
- Aktivitäten-Auswahl-Problem: bestimme gültige Auswahl, so dass $|A|$ maximal ist
- Beispiel:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

$A = \{a_3, a_7, a_{11}\}$ ist gültig, aber nicht maximal, denn
$A' = \{a_1, a_4, a_8, a_{11}\}$ ist auch gültig (und auch maximal)

Lösung mittels dynamischer Programmierung

- identifiziere Teilprobleme, nutze dazu Mengen

$$S_{ij} = \{a_k \in S \mid f_i \leqslant s_k \text{ und } f_k \leqslant s_j\}$$

  anschaulich: S_{ij} = Menge der Aktivitäten, die nach Ende von a_i anfangen und vor Start von a_j aufhören

$\Rightarrow$ jedes $a_k \in S_{ij}$ ist zu a_i und a_j kompatibel

- Vereinfachte Handhabung: fiktive Aktivitäten a_0 und a_{n+1} mit $f_0 = 0$ und $s_{n+1} = \infty$

$\Rightarrow$ $S = S_{0,n+1}$, Indizes i, j in S_{ij} reichen von 0 bis $n+1$

- Vereinfachte Handhabung: Endwerte f_i sind aufsteigend sortiert

$\Rightarrow$ $S_{ij} = \varnothing$ falls $i \geqslant j \Rightarrow$ betrachte nur S_{ij} mit $0 \leqslant i < j \leqslant n+1$

- Sei A_{ij} eine maximale Lösung des Aktivitäten-Auswahl-Problems für S_{ij}

Lösung mittels dynamischer Programmierung

- falls $S_{ij} = \varnothing \Rightarrow A_{ij} = \varnothing$,
- falls $S_{ij} \neq \varnothing$, dann wähle ein $a_k \in S_{ij}$ und definiere $A_{ij} = \{a_k\} \cup A_{ik} \cup A_{kj}$ so, dass $|\{a_k\} \cup A_{ik} \cup A_{kj}|$ maximal wird (falls $a_k \in S_{ij}$ dann gilt $i < k < j$)
- erhalte Rekursionsgleichung für Kardinalität $c(i,j)$ von A_{ij}:

$$c(i,j) = \begin{cases} 0, & \text{falls } S_{ij} = \varnothing \\ \max\{c(i,k) + c(k,j) + 1 \mid a_k \in S_{ij}\}, & \text{sonst} \end{cases}$$

- erhalte daraus direkt den Algorithmus:
  - $\mathcal{O}(n^2)$ Teilprobleme $\Rightarrow \mathcal{O}(n^2)$ Platz
  - Suche nach optimalem k: $\mathcal{O}(n) \Rightarrow \mathcal{O}(n^3)$ Zeit

Auf dem Weg zum Greedy-Algorithmus

$$c(i,j) = \begin{cases} 0, & \text{falls } S_{ij} = \varnothing \\ \max\{1 + c(i,k) + c(k,j) \mid a_k \in S_{ij}\}, & \text{sonst} \end{cases}$$

- betrache $S_{ij} \neq \varnothing$ und sei a_m eine Aktivität mit frühester Endzeit in S_{ij}:

$$f_m = \min\{f_k \mid a_k \in S_{ij}\}$$

$\Rightarrow$ 1. Aktivität kommt in einer maximaler Menge A_{ij} vor

$\Rightarrow$ 2. $S_{im} = \varnothing$, also $A_{im} = \varnothing$ und $c(i,m) = 0$

- aus 1. und 2. erhalte vereinfachte Rekursionsgleichung:

$$A_{ij} = \{a_m\} \cup A_{mj} \qquad \text{bzw.} \qquad c(i,j) = 1 + c(m,j)$$

1. Aktivität a_m kommt in maximaler Menge A_{ij} vor

$$f_m = \min\{f_k \mid a_k \in S_{ij}\}$$

- Nutze Austauschprinzip: Zeige, dass man in optimaler Lösung ohne a_m eine Aktivität findet, mit der man a_m tauschen kann, so dass Optimalität erhalten bleibt
- angenommen, A_{ij} enthält a_m nicht
- da $\{a_m\}$ gültige Lösung ist, und A_{ij} optimal ist, ist $A_{ij} \neq \varnothing$

$\Rightarrow$ existiert $a_k \in A_{ij}$ mit minimaler Endzeit in A_{ij}, $a_k \neq a_m$

- definiere $A'_{ij} = A_{ij} \setminus \{a_k\} \cup \{a_m\}$

$\Rightarrow$ $|A'_{ij}| = |A_{ij}|$ und A'_{ij} ist gültig, da $f_m \leqslant f_k$

$\Rightarrow$ A'_{ij} ist maximale gültige Lösung, die a_m enthält

2. $S_{im} = \varnothing$

$$f_m = \min\{f_k \mid a_k \in S_{ij}\}$$

- Angenommen, $a_k \in S_{im}$

$\Rightarrow$ $f_i \leqslant s_k < f_k \leqslant s_m < f_m$ und da $a_m \in S_{ij}$ gilt zudem $f_m \leqslant s_j$

$\Rightarrow$ $a_k \in S_{ij}$ und $f_k < f_m$

$\Rightarrow$ Widerspruch, zur Minimalität von f_m

Nutzen der vereinfachten Rekursionsgleichungen

$$A_{ij} = \begin{cases} \varnothing, & \text{falls } S_{ij} = \varnothing \\ \{a_m\} \cup A_{mj}, & \text{wobei } a_m \text{ minimale Endzeit in } S_{ij} \text{ hat} \end{cases}$$

- Erhalte drei Vorteile gegenüber dynamischer Programmierung:
  - keine Überlappung $\Rightarrow$ auch top-down Berechnung ist effizient
  - rekursive Aufruf-Struktur verändert j nicht

  $\Rightarrow$ statt $\Theta(n^2)$ Teilprobleme S_{ij} erhalte nur $\Theta(n)$ Teilprobleme $S_{i,n+1}$
  - m kann lokal bestimmt werden, ohne Teillösungen zu kennen
- Gesamt:
  - dynamische Programmierung: $\mathcal{O}(n^3)$ Zeit + $\mathcal{O}(n^2)$ zusätzlicher Platz
  - Greedy-Algorithmus: $\mathcal{O}(n)$ Zeit + $\mathcal{O}(1)$ zusätzlicher Platz
  - bei beiden evtl. zusätzliche Kosten von $\mathcal{O}(n \cdot log(n))$ für Sortierung

Aktivitäten-Auswahl in Java

```
public static void activity(int[] s, int[] f) {
    if (s.length == 0) return;
    int n = s.length;
    System.out.print("1 "); // m for A_{0,n+1} = 1
    int i = 1; // we are producing A_{i,n+1}
    // and look for next m
    for (int m = 2; m <= n; m++) {
        if (s[m-1] >= f[i-1]) { // index shift for Java
            i = m;
            System.out.print(i + " ");
        }
    }
    System.out.println();
}
```


Warum Greedy? In welcher Beziehung?

- Vorgestelltes Verfahren wählt Aktivität mit minimaler Endzeit
- Warum ist das "gierig"?
  ⇒ Maximierung des verbleibender Freiraum für spätere Aktivitäten
- Andere "gierige" Strategien:
  - Wähle Aktivität mit geringster Dauer
  ⇒ Maximierung des verbleibender Freiraum für andere Aktivitäten
  - Wähle Aktivität mit frühester Startzeit
  ⇒ sobald Ressource frei wird, kann nächst-mögliche Aktivität beginnen
  - Wähle Aktivität mit möglichst vielen anderen kompatiblen Aktivitäten

  alle diese anderen Strategien klingen auch gierig, führen aber nicht zu optimalen Lösungen

⇒ obwohl ein gegebener Greedy-Algorithmus einfach zu implementieren und effizient ist, ist die Erstellung nicht einfach, da es viele "gierige" Auswahl-Möglichkeiten gibt, die nicht zum Optimum führen (und es zudem unklar ist, ob es überhaupt eine gibt)

Entwurf von Greedy-Algorithmen

- Aktivitäten-Auswahl Beispiel ging über Umweg dynamische Programmierung
  (um den Unterschied zwischen den beiden Algorithmen-Arten zu verdeutlichen)
- Greedy-Algorithmen Entwurf in 3 Schritten
  - Lösen des Optimierungs-Problems, wobei jeweils eine (gierige) Auswahl getroffen wird, so dass ein Teilproblem verbleibt
  - Zeige, dass es immer optimale Lösung gibt, die der gierigen Auswahl entspricht
  - Zeige, dass sich aus optimaler Lösung des Teilproblems in Kombination mit der gierigen Auswahl eine optimale Lösung des Gesamtproblems ergibt
    (man kann dabei Voraussetzen, dass optimale Lösung des Teilproblems durch Greedy-Algorithmus erzeugt worden ist $\Rightarrow$ Induktionsbeweis)

Komplexeres Beispiel: Huffmann-Codierungen

- Anwendung: (verlustfreie) Komprimierung von Dateien
- Komprimierung hat zahlreiche Anwendungen: Archivierung, Datei-Übertragung, Fax, ...
- Komprimierungs-Problem: gegeben ein String s über Alphabet Σ mit $|\Sigma| = n$, finde eine Binär-Codierung c von s, so dass $|c|$ möglichst klein wird
- Lösungsmöglichkeiten:
  - feste Bitlänge: jedes Zeichen wird in $\lceil log(n) \rceil$ Bits codiert

  $\Rightarrow$ Länge von c ist $|s| \cdot \lceil log(n) \rceil$
  - variable Bitlänge: z.B. Zeichen e häufiger als Zeichen x

  $\Rightarrow$ nutze für e kurzen Code, für x längeren

  $\Rightarrow$ nutze Präfix-Codes für Eindeutigkeit:

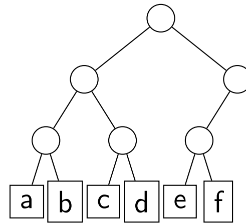
  kein Code eines Zeichens ist Präfix eines Codes für ein anderes Zeichen

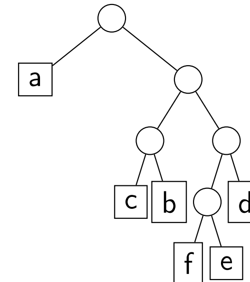
  Beispiel: $e = 1$, $h = 00$, $x = 01$ ist Präfixcode, $e = 1$, $h = 0$, $x = 01$ ist keiner

Präfix-Codes

- jeder Code läßt sich als binärer Baum darstellen (links = 0, rechts = 1)
- Präfix-Code, falls alle Zeichen in Blättern
- Beispiel: Text mit 100.000 Zeichen über $\Sigma = \{a,b,c,d,e,f\}$

	a	b	c	d	e	f
Häufigkeit in $\|s\|$	45k	13k	12k	16k	9k	5k
Codewort fester Länge	000	001	010	011	100	101
Codewort variabler Länge	0	101	100	111	1101	1100





Code-Länge $= 100k \cdot 3$ $\quad 45k \cdot 1 + 13k \cdot 2 + \cdots + 5k \cdot 4 = 224k$

Codierung = Optimierungs-Problem

- Aufgabe:
  - Gegeben eine Häufigkeitstabelle $f(c)$ für jedes Zeichen $c \in \Sigma$
  - Finde einen Baum T, so dass der resultierende Code minimale Länge hat
- Notationen:
  - $d_T(c)$: Tiefe/Ebene des Knotens c im Baum T

  $\Rightarrow$ $d_T(c)$ = Anzahl Bits für Codierung des Zeichens c
  - $B(T)$: Länge des entstehendes Codes bei Nutzung des Baums T

$$B(T) = \sum_{c \in \Sigma} f(c) \cdot d_T(c)$$

- Brute-Force Lösung: generiere alle Bäume (die einem Präfix-Code von Σ entsprechen) und merke den Baum T mit minimalem Wert $B(T)$

Rekursiver Ansatz zur Lösung

- Rekursion über $|\Sigma|$
- wenn $|\Sigma| \geqslant 2$, dann gibt es zwei Zeichen c und c', die unterschieden werden müssen

⇒ erstelle inneren Knoten n über c und c'

⇒ der neue Knoten n kann jetzt als eigenes Zeichen angesehen werden

⇒ erhalte kleineres Alphabet $\Sigma' = \Sigma \cup \{n\} \setminus \{c, c'\}$

⇒ Häufigkeit von n ist $f(n) = f(c) + f(c')$

- Angenommen wir erhalten Baum T' als Lösung für $|\Sigma'|$, dann erhalte Lösung für $|\Sigma|$, indem man das Zeichen n durch den inneren Knoten n mit Kindern c und c' ersetzt

$$\Rightarrow \quad \begin{aligned} B(T) &= B(T') - f(n) \cdot d_{T'}(n) + f(c) \cdot d_T(c) + f(c') \cdot d_T(c') \\ &= B(T') - f(n) \cdot d_{T'}(n) + (f(c) + f(c')) \cdot (1 + d_{T'}(n)) \\ &= B(T') + f(c) + f(c') \end{aligned}$$

Greedy-Algorithmus: Huffman-Codierung

- $B(T) = B(T') + f(c) + f(c')$
  (Länge des Codes für Σ = Länge des Codes für Σ' + $f(c)$ + $f(c')$)
- Treffe gierige Auswahl: Wähle c und c' mit kleinsten Häufigkeiten

⇒ Schritt 1 (Lösen des Optimierungsproblems mit einem Greedy-Algorithmus) ist fertig:

  - Solange noch mehr als Symbol in Σ:
  - Füge inneren Knoten über Symbolen c und c' mit minimaler Häufigkeit ein
  - Betrachte diesen inneren Knoten als neues Symbol mit Häufigkeit $f(c) + f(c')$

Huffman-Codierung: Korrektheit

- Schritt 2: Zeige, dass es optimale Lösung gibt, die der gierigen Auswahl entspricht

Lemma

Seien Σ und eine Häufigkeits-Funktion f gegeben. Seien $c, c' \in \Sigma$ mit minimaler Häufigkeit. Dann existiert ein optimaler Präfix-Code, dargestellt durch Baum T, bei dem c und c' Brüder sind.

Beweis des Lemmas mittels Austauschprinzip.

Huffman-Codierung: Korrektheit

- Schritt 3: Zeige, dass sich aus optimaler Lösung des Teilproblems in Kombination mit der gierigen Auswahl eine optimale Lösung ergibt.

Lemma

Sei n der neue Knoten / das neue Zeichen, dass anstelle von c und c' verwendet wird. Sei T' eine optimale Lösung für $\Sigma' = \Sigma \cup \{n\} \setminus \{c, c'\}$ mit $f(n) = f(c) + f(c')$. Dann ist T optimal, wobei T durch T' entsteht, indem man n durch inneren Knoten und Blätter c und c' ersetzt.

Huffman-Codierung: Optimalität

Theorem

Der Greedy-Algorithmus für Huffman-Codierungen erzeugt einen optimalen Präfix-Code.

- Abschwächung: obwohl Optimalität vorliegt, gibt es bessere Komprimierungen
  - 1. Grund: man kann z.B. Alphabet anpassen: nutze "ver", "sch", "st" als einzelne Zeichen
  - 2. Grund: kodiere "aaaaaaaaaa" als "10 mal a"
- Aber: nachdem Alphabet festliegt, ist Huffman optimal

Huffman-Codierung in der Praxis

- da Huffman-Codierungen einfach zu kodieren/entkodieren sind und zudem Optimalität gewährleistet ist, findet dieses Verfahren auch in der Praxis Einsatz (jpeg, mpeg, Fax, ...)
- Beispiel: Fax-Protokoll
- jede Zeile wird kodiert als Wort über $\Sigma = \{is, iw \mid 1 \leqslant i \leqslant 1728\}$

  ■■■■□■■□□□■■■□■■

  4s 1w 2s 2w 3w 3s 1w 2s
- der resultierende String wird dann mittels eines Präfix-Codes codiert
- dazu wurde eine Huffmann-Codierung genutzt (um den Baum nicht übertragen zu müssen, wird beim Fax-Protokoll ein fester Baum gewählt)
- Wahl von f_{fax} ergab sich durch Mittelung über Häufigkeitsfunktionen vieler Dokumente

Zusammenfassung: Greedy-Algorithmen

- Greedy-Algorithmen sind einfacher und effizienter als dynamische Programmierung
  - Entscheidung wird lokal getroffen, ohne Lösung von Teilproblemen zu kennen
  - nur ein entstehendes Teilproblem $\Rightarrow$ keine Tabellen
- Entwurf in 3 Schritten
- Schwierigkeit: Korrektheit der Algorithmen
- Beispiele
  - Aktivitäten-Auswahl Problem
  - Huffman-Codierung

Übersicht

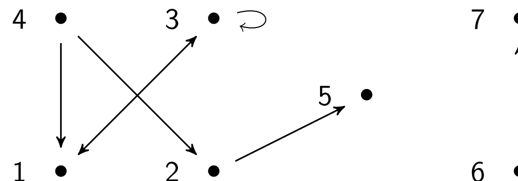
- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Warum Graphen?

- betrachte folgende Probleme ...
  - Was ist schnellster Weg von Innsbruck nach Rom?
  - Was ist kürzester Weg um alle Kunden zu besuchen?
  - Wieviel Abwasser kann eine Kanalisation verkraften?
  - Wie kann ich die Menge produzierter Waren optimieren?
  - Wie viele Router werden durch *ping www.foo.de* mindestens passiert?
- alle diese Probleme lassen sich mit Hilfe von Graphen modellieren und mit Hilfe von Graph-Algorithmen lösen

Was sind Graphen?

- ein gerichteter Graph ist ein Paar $G = (V, E)$ mit
  - V: Menge von Knoten (vertices)
  - $E \subseteq V \times V$: Menge der Kanten (edges)
- ein Paar $(v, v') \in E$ heißt Kante mit Anfangsknoten v und Endknoten v'
- wenn $e = (v, v') \in E$ dann gilt:
  - v und v' sind adjazent / benachbart
  - v ist Vorgänger von v' und v' ist Nachfolger von v
  - v ist mit e inzident (und auch e mit v)
  - v' ist mit e inzident (und auch e mit v')
- Beispiel: $V = \{1, 2, 3, 4, 5, 6, 7\}$,
  $E = \{(1, 3),\ (2, 5),\ (4, 2),\ (3, 3),\ (4, 1),\ (3, 1),\ (6, 7)\}$

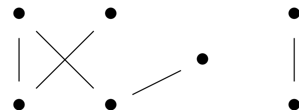


Weitere Notationen über Graphen

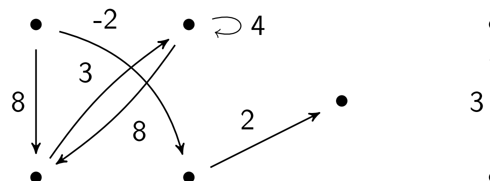
- $G = (V, E)$ ist Teilgraph von $G' = (V', E')$, falls $V \subseteq V'$ und $E \subseteq E'$
- für $G = (V, E)$ und $V' \subseteq V$ induziert V' den Teilgraphen $(V', E \cap V' \times V')$
- $G - V'$ ist der durch $V \setminus V'$ induzierte Teilgraph
  für einzelne Knoten v ist $G - v$ definiert als $G - \{v\}$
- Eingangsgrad eines Knotens: $indeg(v) = |\{v' \mid (v', v) \in E\}|$
- Ausgangsgrad eines Knotens: $outdeg(v) = |\{v' \mid (v, v') \in E\}|$
- $v_0 v_1 \dots v_k$ ist Weg der Länge k gdw. $\forall 1 \leqslant i \leqslant k : (v_{i-1}, v_i) \in E$
- v' ist von v erreichbar ($v \to_E^* v'$) gdw. es Weg $v \dots v'$ gibt
- Weg ist einfach, wenn kein Knoten doppelt vorkommt
- Weg $v_0 \dots v_k$ ist Zyklus, wenn $k > 0$ und $v_0 = v_k$
- Graph ist azyklisch, wenn es keine Zyklen in G gibt

Varianten von Graphen

- $G = (V, E)$ ist ungerichtet, falls
  - $(v, v') \in E$ gdw. $(v', v) \in E$
  - $(v, v) \notin E$
- Beispiel:



- $G = (V, E)$ ist (Kanten)-gewichtet, falls es eine zusätzlich eine Funktion $w : E \to \mathbb{N}, \mathbb{R}, \dots$ gibt
- Beispiel:



Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Datenstruktur für Graphen

- es gibt unterschiedliche Möglichkeiten, Graphen darzustellen
- jede Datenstruktur hat Vor- und Nachteile

$\Rightarrow$ Wahl der Datenstruktur abhängig von benötigen Graph-Operationen:
  - teste zu v und v', ob $(v, v') \in E$
  - liefere / iteriere über alle Vorgänger von v
  - liefere / iteriere über alle Nachfolger von v
  - entferne Knoten / Kante
  - füge Knoten / Kante hinzu

und natürlich abhängig vom Speicherverbrauch

Eine Vereinfachung

- im Normalfall: Knoten sind Städte, Personen, ... Objekte vom Typ T
- hier: Knoten $V = \{0, \ldots, n-1\}$ oder $V = \{1, \ldots, n\}$
- dies kann immer erreicht werden:
  - nummeriere alle Objekt-Knoten mit Zahlen
  - speichere T[]: von Knoten-Nummer auf Objekt
  - speichere HashMap<T,Integer>: von Objekt auf Knoten-Zahl

Möglichkeit 1: Adjazenzmatrix

- Idee: Stelle Graph mit n Knoten, $V = \{1, \ldots, n\}$ als $n \times n$-Matrix dar
- $a_{ij} = \begin{cases} 1, & \text{wenn } (i,j) \in E \\ 0, & \text{sonst} \end{cases}$
- Speicherbedarf: $\mathcal{O}(|V|^2)$

⇒ sehr hoch, wenn es nur wenige Kanten gibt
- Testen, ob $(v, v') \in E$: $\mathcal{O}(1)$
- Iterieren über Vorgänger / Nachfolger von v: $\mathcal{O}(|V|)$

⇒ schlecht, wenn es nur wenige Vorgänger / Nachfolger von v gibt
- Entfernen / Hinzufügen von Kanten: $\mathcal{O}(1)$
- Entfernen / Hinzufügen von Knoten: $\mathcal{O}(|V|^2)$

- Insgesamt: Adjazenzmatrizen eignen sich vor allem dann, wenn es viele Kanten gibt, wenn also $|E| = \Theta(|V|^2)$ ist, und wenn V konstant ist

Möglichkeit 2: Adjazenzliste

- Idee:
  - Verwalte die Knoten als Array
  - Speichere zu jedem Knoten die Menge der Nachfolger als Liste
  - in Java: List<Integer>[] graph
- Speicherbedarf: $\mathcal{O}(|V| + |E|)$
- Testen, ob $(v, v') \in E$: $\mathcal{O}(outdeg(v))$
- Iterieren über Nachfolger von v: $\mathcal{O}(outdeg(v))$
- Iterieren über Vorgänger von v: $\mathcal{O}(|V| + |E|) \Rightarrow$ schlecht
- Entfernen / Hinzufügen von Kante (v, v'): $\mathcal{O}(outdeg(v))$
- Hinzufügen von Knoten: $\mathcal{O}(|V|)$
- Entfernen von Knoten: $\mathcal{O}(|V| + |E|) \Rightarrow$ schlecht

- Insgesamt: Adjazenzlisten sind zu bevorzugen, wenn es wenige Kanten gibt; Vorgänger-Bestimmung ist aufwändig

Möglichkeit 3: Erweiterungen von Adjazenzlisten

- Idee:
  - Speichere zu jedem Knoten die Menge der Nachfolger und Vorgänger als Liste
  - in Java: Pair<List<Integer>,List<Integer>>[] graph
- Speicherbedarf: $\mathcal{O}(|V| + |E|)$, höher als bei einfachen Adjazenzlisten
- Testen, ob $(v, v') \in E$: $\mathcal{O}(outdeg(v))$
- Iterieren über Nachfolger von v: $\mathcal{O}(outdeg(v))$
- Iterieren über Vorgänger von v: $\mathcal{O}(indeg(v))$
- Hinzufügen von Kante (v, v'): $\mathcal{O}(outdeg(v))$
- Entfernen von Kante (v, v'): $\mathcal{O}(outdeg(v) + indeg(v'))$
- Hinzufügen von Knoten: $\mathcal{O}(|V|)$
- Entfernen von Knoten v:
  $\mathcal{O}(|V| + \sum_{(v,v') \in E} indeg(v') + \sum_{(v',v) \in E} outdeg(v'))$

- Insgesamt: erweiterte Adjazenzlisten sind zu bevorzugen, wenn man effizienten Zugriff auf Vorgänger braucht; ansonsten sind einfache Adjazenzlisten effizienter

Möglichkeit 4: dynamische Adjazenzlisten

- Idee:
  - Beobachtung: Löschen von Knoten hinterlässt Lücken in Nummerierung
  - $\Rightarrow$ schlecht für bisherige Implementierungen, da Zugriff über Array-Position
  - $\Rightarrow$ verwalte Knoten nicht in Array, sondern in Liste oder mittels Hashing
  - in Java:
    ```
    HashMap<Integer, Pair<List<Integer>,List<Integer>>> graph
    ```
  - $\Rightarrow$ einfaches Löschen von Knoten, Einfügen von Knoten effizienter
- Speicherbedarf: $\mathcal{O}(|V| + |E|)$, aber höher als bei statischem Array
- Iterieren, Kantenzugriff: wie im statischen Fall
- Hinzufügen von Knoten: $\mathcal{O}(1)$ amortisiert
- Entfernen von Knoten v:
  $\mathcal{O}(\sum_{(v,v') \in E} \mathit{indeg}(v') + \sum_{(v',v) \in E} \mathit{outdeg}(v'))$ amortisiert
- Insgesamt: dynamische Adjazenzlisten sind zu bevorzugen, wenn man häufig Knoten einfügt und löscht; ansonsten sind statische Adjazenzlisten Platz-sparender und effizienter

Möglichkeit 5: dynamische Adjazenzlisten mittels Hashing

- Idee:
  - Speichere zu jedem Knoten die Menge der Nachfolger (und Vorgänger) als dynamisch wachsendes HashSet (garantiere $\frac{1}{2} \leqslant \alpha \leqslant 2$)
  - in Java:
    ```
    HashMap<Integer,Pair<HashSet<Integer>,HashSet<Integer>>> g
    ```
- Speicherbedarf: $\mathcal{O}(|V| + |E|)$, aber höher als bei Listen
- Testen, ob $(v, v') \in E$: $\mathcal{O}(1)$ im Mittel
- Iterieren über Nachfolger von v: $\mathcal{O}(\mathit{outdeg}(v))$
- Iterieren über Vorgänger von v: $\mathcal{O}(\mathit{indeg}(v))$
- Hinzufügen von Kante (v, v'): $\mathcal{O}(1)$ amortisiert, im Mittel
- Entfernen von Kante (v, v'): $\mathcal{O}(1)$ amortisiert, im Mittel
- Hinzufügen von Knoten: $\mathcal{O}(1)$
- Entfernen von Knoten v: $\mathcal{O}(\mathit{outdeg}(v) + \mathit{indeg}(v))$ amortisiert, im Mittel
- Insgesamt: alle Operationen haben optimale asymptotische Laufzeit; hoher Verwaltungsaufwand erzeugt hohe Konstanten $\Rightarrow$ wenn *max-indeg* und *max-outdeg* klein, sind Listen schneller

Weitere Möglichkeiten

- speichern der Nachfolger-Liste als Array

$\Rightarrow$ schlecht, wenn E nicht konstant

$\Rightarrow$ schnellerer Zugriff als auf Listen, weniger Platzbedarf

- speichern der Nachfolger-Liste als sortiertes Array

$\Rightarrow$ schnellerer Test auf $(v, v') \in E$ durch binäre Suche

- ...

Zusammenfassung

- Viele Datenstrukturen für Graphen
- Wähle geeignete Datenstruktur abhängig von benötigten Operationen
- Frage-Stellungen
  - Ist V konstant?
  - Ist E konstant?
  - Benötigt man oft Vorgänger? Oder Nachfolger? Oder beides?
  - Wie viele Kanten gibt es? $\mathcal{O}(|V|)$? $\mathcal{O}(|V|^2)$?

Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Topologisches Sortieren

- Topologische Sortierung stellt eine Ordnung auf den Knoten her
- formal: topologische Sortierung ist bijektive Abbildung
  $ord : V \to \{1, \dots, |V|\}$, so dass für alle Knoten v und w gilt:

$$\text{wenn } (v, w) \in E \text{ dann } ord(v) < ord(w)$$

- anschaulich: Ordnungsziffern entlang eines Weges werden immer größer
- für Weg $v_0 \dots v_k$ der Länge k gilt $ord(v_0) \leqslant ord(v_k)$
  (genauer: $ord(v_0) \leqslant ord(v_k) - k$)
- es existiert topologische Sortierung gdw. Graph azyklisch ist

$\Rightarrow$ Algorithmus für topologische Sortierung testet, ob Graph azyklisch ist

Konstruktion einer topologischen Sortierung

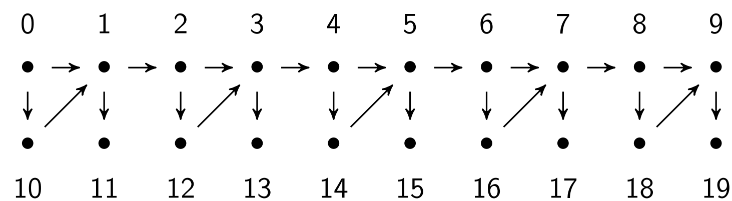
- falls topologische Sortierung existiert, gibt es einen Knoten v mit $ord(v) = 1 \Rightarrow indeg(v) = 0$
- falls Knoten v mit $indeg(v) = 0$ existiert, kann man für diesen o.B.d.A. $ord(v) = 1$ setzen

$\Rightarrow$ Verfahren für topologische Sortierung
  - $i := 0$; $n := |V|$
  - Solange es Knoten v mit $indeg(v) = 0$ gibt:
    - $i := i + 1$; $ord(v) := i$
    - $G := G - v$
  - Falls $i = n$, habe topologische Sortierung erstellt
  - Ansonsten gibt es keine topologische Sortierung

- Problem: Wie findet man Knoten mit $indeg(v) = 0$?
- Lösung: Starte von beliebigen Knoten, gehe rückwärts solange wie möglich, maximal $|V|$ Schritte

$\Rightarrow$ Komplexität ist $\mathcal{O}(n^2)$

Beispiel



	0	1	2	3	4	5	6	7	8	9
ord										

	10	11	12	13	14	15	16	17	18	19
ord										

Verbesserte Version

- Finden der *indeg* = 0-Knoten zu kostspielig

⇒ Speichere alle *indeg* = 0-Knoten

- Entfernen eines Knoten ⇒ verringere *indeg* der Nachfolger um 1

⇒ einzige Möglichkeit, dass neue *indeg* = 0-Knoten entstehen

- verbesserter Algorithmus:
  - $i := 0$
  - für alle Knoten v setze $indeg[v] := indeg(v)$
  - für alle Knoten v mit $indeg[v] = 0$: $indegZero.push(v)$
  - solange *indegZero* nicht-leer
    - $v := indegZero.pop()$
    - $i := i + 1$; $ord(v) := i$
    - für alle Nachfolger v' von v:
      - $indeg[v'] := indeg[v'] - 1$
      - wenn $indeg[v'] = 0$ dann $indegZero.push(v')$
  - *ord* ist topologische Sortierung gdw. $i = |V|$
- Laufzeit: $\mathcal{O}(|V| + |E|)$

Korrektheit

- sei O die Menge der Knoten, die schon eine Ordnungsziffer bekommen haben
- Invariante:
  1. *ord* ist eine topologische Sortierung des von O induzierten Teilgraphen
  2. $|O| = i$
  3. für alle $v \notin O$ ist $indeg[v] = indeg(v, G - O)$
  4. *indegZero* enthält alle Knoten aus $G - O$ mit Eingangsgrad 0
  5. für alle $v \in O$ gilt: $ord(v) \leqslant i$
  6. es gibt keine Kanten (v, w) mit $v \notin O$ und $w \in O$
- aus Invariante folgt Behauptung:
  - wenn nach Abschluss $i = |V|$ ist, dann gilt $O = V$ wegen 2 und wegen 1 ist damit eine topologische Sortierung für G gefunden
  - wenn nach Abschluss $i < |V|$ ist, dann gilt $O \subset V$ wegen 2 und damit enthält wegen 4 der nicht-leere Graph $G - O$ nur Knoten mit Eingangsgrad > 0; folglich gibt es keine topologische Sortierung für $G - O$ und damit auch nicht für G

Zusammenfassung

- topologische Sortierung kann genutzt werden, um Graph auf Zyklen zu testen
- bei geeigneter Datenstruktur in $\mathcal{O}(|V| + |E|)$ implementierbar
- (Randbemerkung für Übung: Kanten sind topologisch sortiert, wenn gilt: wann immer (v, w) vor (v', w') in topologischer Reihenfolge kommt, dann muss $ord(v) \leqslant ord(v')$ gelten)

Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Breitensuche

- startet Suche bei einem Knoten v
- dient zum Test der Erreichbarkeit $v \rightarrow_E^* w$?
- findet zudem kürzeste Wege von v zu allen anderen Knoten
- Laufzeit: $\mathcal{O}(|V| + |E|)$ (optimal)

Breitensuche mit Start v

- Färbt Knoten in drei Farben
  - weiß: Knoten noch nicht gefunden
  - grau: Knoten gefunden, aber Nachfolger noch nicht untersucht
  - schwarz: Knoten gefunden und über Nachfolger iteriert
- Speichere graue Knoten in Queue
- Speichere zu jedem (grauen und schwarzen) Knoten seine Distanz zu v
- Speichere zu jedem (grauen und schwarzen) Knoten w seinen Vorgänger
  (der Vorgänger, der w erstmalig entdeckt hat)

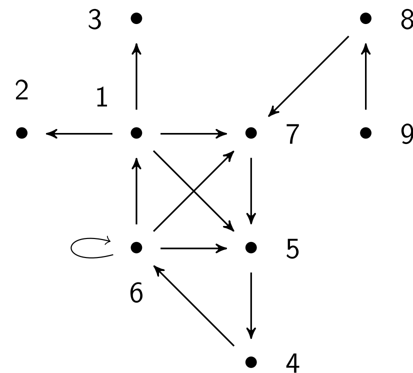
$\Rightarrow$ erhalte Breitensuchbaum

Algorithmus

Breitensuche von Knoten v

- $todo$ = leere Queue
- setze $color[w]$ = weiß für alle Knoten w
- setze $distance[w] = \infty$ für alle Knoten w
- $color[v]$ = grau
- $distance[v] = 0$
- $parent[v] = null$
- $todo.enqueue(v)$
- solange $todo$ nicht leer
  - $w = todo.dequeue()$
  - für alle Nachfolger u von w mit $color[u]$ = weiß
    - $color[u]$ = grau
    - $distance[u] = distance[w] + 1$
    - $parent[u] = w$
    - $todo.enqueue(u)$
  - $color[w]$ = schwarz

Beispiel



Eigenschaften

- definiere Distanz zwischen Knoten v und w als $\delta(v, w)$: Länge des kürzestes Pfades von v nach w, falls dieser existiert oder ∞, sonst
- nach Ausführung von Breitensuche mit Startknoten v gilt:
  - $\delta(v, w) = \mathit{distance}[w]$
  - der Breitensuchbaum enthält die kürzesten Wege von v zu allen erreichbaren Knoten w
- wenn $\delta(v, w) = \mathit{distance}[w]$ gilt, dann ist die Breitensuchbaum-Aussage sofort erfüllt, da $\mathit{parent}[u] = w$ zu einem Pfad für u führt, der genau um 1 länger ist als der Pfad für w, und da $\delta(v, u) = \mathit{distance}[u] = \mathit{distance}[w] + 1 = \delta(v, w) + 1$

Korrektheit

- zeige zur Korrektheit von $\delta(v, w) = \mathit{distance}[w]$ zunächst, dass $\delta(v, w) \leqslant \mathit{distance}[w]$
- zeige dazu Invariante: zu jedem Zeitpunkt gilt: $\delta(v, w) \leqslant \mathit{distance}[w]$
  - Invariante gilt zu Beginn der Schleife, da $\delta(v, v) = 0 = \mathit{distance}[v]$ und $\mathit{distance}[w] = \infty$ für alle $w \neq v$
  - sei nun u ein beliebiger Nachfolger von Knoten w, für den die Distanz-Zuweisung $\mathit{distance}[u] = \mathit{distance}[w] + 1$ ausgeführt wird
    - da $(w, u) \in E$, gilt $\delta(v, u) \leqslant \delta(v, w) + 1$
    - aufgrund der Invariante gilt $\delta(v, w) \leqslant \mathit{distance}[w]$
    - $\Rightarrow$ $\delta(v, u) \leqslant \delta(v, w) + 1 \leqslant \mathit{distance}[w] + 1 = \mathit{distance}[u]$

Korrektheit Teil 2

- um $\delta(v, w) \geqslant \mathit{distance}[w]$ zu zeigen, benötigt man zunächst Hilfsaussagen über die Elemente in der Queue

Lemma

Die folgende Invariante ist erfüllt:
Sei todo $= (w_1, \dots, w_k)$. *Dann gilt* $\mathit{distance}[w_i] \leqslant \mathit{distance}[w_{i+1}]$ *für alle* $1 \leqslant i < k$ *und* $\mathit{distance}[w_k] \leqslant \mathit{distance}[w_1] + 1$.

- Das Lemma besagt, dass zu jedem Zeitpunkt
  - die Elemente in der Queue nach aufsteigender Distanz sortiert sind
  - der Distanz-Unterschied in der gesamten Queue maximal 1 ist

Korrektheit Teil 3

Theorem

Nach Breitensuche mit Startknoten v gilt:

- $\mathit{distance}[w] = \delta(v, w)$ *für alle Knoten w*
- *Der Breitensuchbaum, der durch parent repräsentiert wird, enthält genau die kürzesten Pfade von v zu allen erreichbaren Knoten w*

Zusammenfassung Breitensuche

- Nützlich, um kürzeste Wege von gegebenen Startknoten zu berechnen
- optimale Komplexität von $\mathcal{O}(|V| + |E|)$
- wesentliches Hilfsmittel: Queues

Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Tiefensuche

- ähnlich der Breitensuche
  - Laufzeit: $\mathcal{O}(|V| + |E|)$
  - Färbung: weiß $\longrightarrow$ grau $\longrightarrow$ schwarz
  - *todo*-Menge von grauen Knoten
  - Erzeugung von Tiefensuchbaum
- ... aber auch unterschiedlich
  - *todo* ist Stack, nicht Queue $\Rightarrow$ nutze normale Rekursion
  - Zeitstempel beim Finden und Verlassen von Knoten
  - Start von jedem unbesuchten Knoten

  $\Rightarrow$ mehrere Tiefensuchbäume (Tiefensuchwald)

Algorithmus

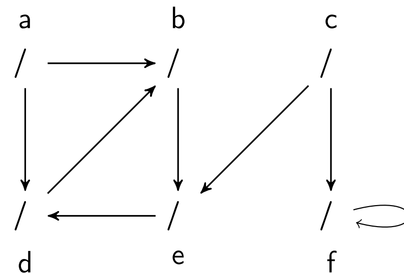
Tiefensuche(v)

- $color[v]$ = grau
- $time$++; $d[v] = time$
- für alle Nachfolger u von v: wenn $color[u]$ = weiß, dann
  - $parent[u] = v$
  - Tiefensuche(u)
- $color[v]$ = schwarz
- $time$++; $f[v] = time$

Tiefensuche

- für alle Knoten v: $color[v]$ = weiß
- $time = 0$
- für alle Knoten v: wenn $color[v]$ = weiß, dann
  - $parent[v] = null$
  - Tiefensuche(v)

Beispiel



Klammerungstheorem

- Angenommen, wir nutzen öffnende Klammer (u, wenn u erstmalig bearbeitet wird (also zur Zeit $d[u]$), und wir nutzen schließende Klammer u), wenn die Bearbeitung von u abgeschlossen ist, dann ergibt sich ein wohlgeformter Klammerausdruck
- im Beispiel: (a(b(e(dd)e)b)a)(c(ff)c)
- formaler ausgedrückt:

Theorem

Seien u und v zwei Knoten. Dann gilt einer der 3 Fälle

- *die Intervalle $[d[u], f[u]]$ und $[d[v], f[v]]$ sind disjunkt, und u und v sind in verschiedenen Bäumen des Tiefensuchwalds*
- *das Intervall $[d[u], f[u]]$ ist vollständig in $[d[v], f[v]]$ enthalten, und u ist Nachfahre von v im Tiefensuchwald*
- *das Intervall $[d[v], f[v]]$ ist vollständig in $[d[u], f[u]]$ enthalten, und v ist Nachfahre von u im Tiefensuchwald*

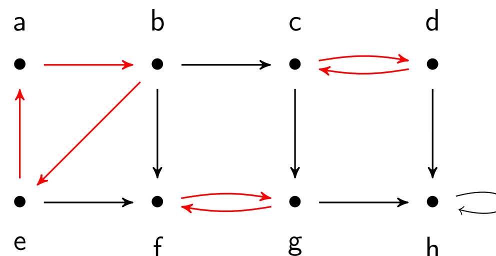
$\Rightarrow$ v ist Nachfolger von u im Tiefensw. gdw. $d[u] < d[v] < f[v] < f[u]$

Klassifizierung von Kanten

- jede der Kanten $(u, v) \in E$ hat einen von 4 Typen
  - (u, v) ist Baumkante, wenn (u, v) auch im Tiefensuchwald vorkommt
  - (u, v) ist Rückwärtskante, wenn u Nachfolger von v in einem Tiefensuchbaum ist
  - (u, v) ist Vorwärtskante, wenn (u, v) keine Baumkante ist und v Nachfolger von u in einem Tiefensuchbaum ist
  - (u, v) ist Querkante, wenn (u, v) eine Verbindung zwischen zwei Tiefensuchbäumen herstellt
- Kante (u, v) kann leicht klassifiziert werden, je nach dem welche Farbe v hatte, als von u der Nachfolger v betrachtet wurde
  - v war weiß: (u, v) ist Baumkante
  - v war grau: (u, v) ist Rückwärtskante
  - v war schwarz: (u, v) ist Vorwärtskante oder Querkante (Vorwärtskante: $d[u] < d[v]$, Querkante: $d[u] > d[v]$)
- Tiefensuche ermöglicht Test auf Zyklen: Graph azyklisch gdw. keine Rückwärtskanten

Starke Zusammenhangskomponenten

- $C \subseteq V$ ist Zusammenhangskomponente von V gdw. jeder Knoten aus C von jedem Knoten aus C in vom C induzierten Teilgraphen erreichbar ist
- C ist starke Zusammenhangskomponente (SCC) gdw. C maximale Zusammenhangskomponente ist (alle $C' \supset C$ sind nicht Zusammenhangskomponenten)
- die Menge der SCCs eines Graphen sind eindeutig und bilden eine Partition von V



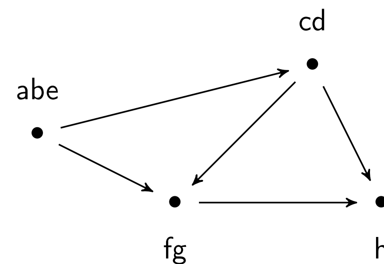
$V = \{a, b, e\} \cup \{c, d\} \cup \{f, g\} \cup \{h\}$

Kosarajus Algorithmus zur Berechnung von SCCs

- führe Tiefensuche aus, erhalte Endzeiten $f[u]$
- drehe alle Kanten um
- führe Tiefensuche aus, iteriere über Knoten nach absteigenden Endzeiten $f[u]$
  (beginne also mit Knoten mit höchster Endzeit aus erster Tiefensuche)
- die Knoten jedes Tiefensuchbaums der zweiten Tiefensuche bilden eine SCC

Eigenschaften von Kosarajus Algorithmus

- optimale Laufzeit: $\mathcal{O}(|V| + |E|)$
- einfach zu implementieren
- korrekt
- liefert SCCs in topologischer Reihenfolge



Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

Das Problem der kürzesten Wege

- gegeben ein Graph mit Kanten-Gewichten $w \approx$ Entfernung
- Fragestellungen:
  - gegeben a und b, was ist der kürzeste Weg von a nach b?
  - gegeben a, was sind die kürzesten Weg von a zu allen anderen Knoten b?
  - was sind die kürzesten Wege zwischen allen Knoten a und b?
- in gewichteten Graphen definiere Distanz zwischen a und b als

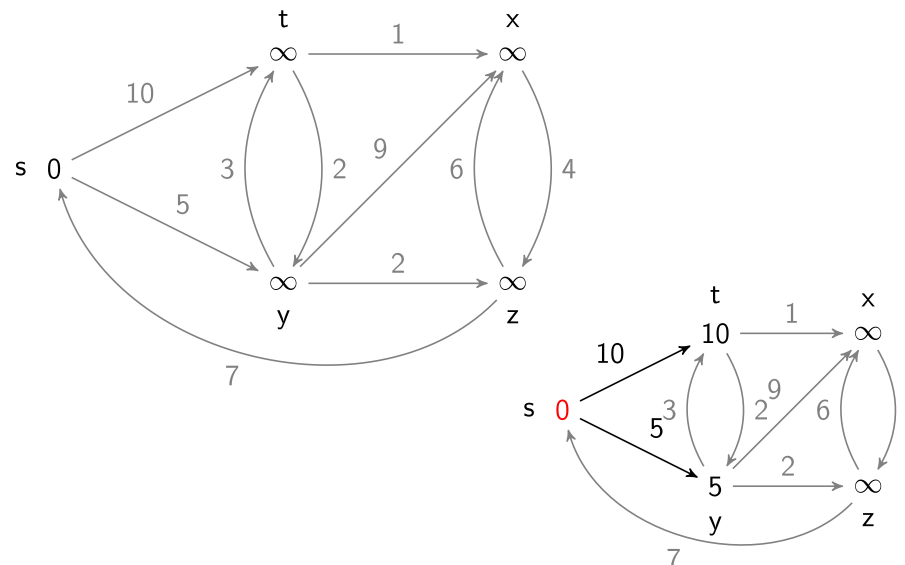
$$\delta(a, b) = \min\{ \sum_{0 \leqslant i < n} w(v_i, v_{i+1}) \mid v_0 v_1 \dots v_n \text{ ist Weg von } a \text{ nach } b\}$$

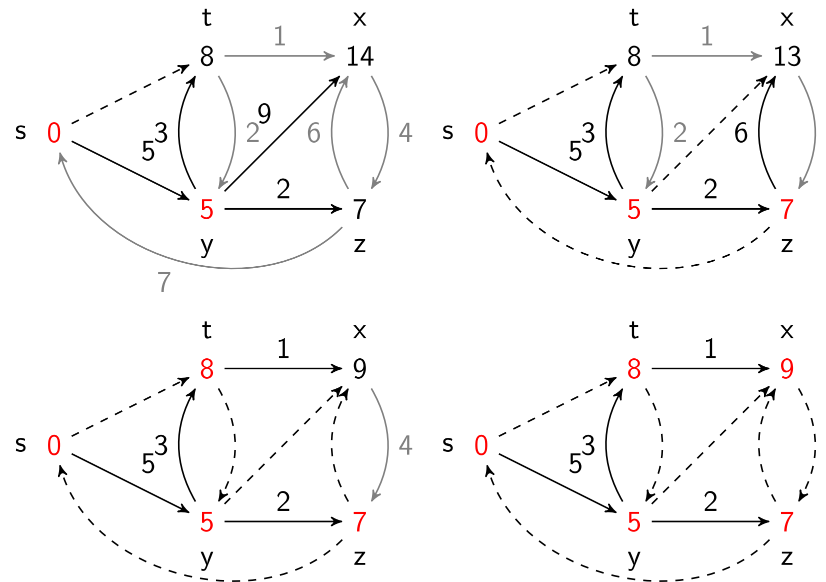
ein Weg, bei dem das Minimum erreicht wird, ist ein kürzester Weg

Algorithmus von Dijkstra

- Gegeben: gewichteter Graph (V, E, w) und Startknoten $s \in V$
- Berechnet: kürzeste Wege zu allen anderen Knoten
- Algorithmus:
  1. für alle Knoten v: setze $d[v] = \begin{cases} 0, & \text{falls } v = s \\ \infty, & \text{sonst} \end{cases}$
  2. $queue = V$
  3. solange *queue* nicht leer
     - entferne Knoten u mit minimalem $d[u]$ aus *queue*
     - für alle Nachfolger v von u: $d[v] = min(d[v], d[u] + w(u, v))$

Beispiel





Analyse von Dijkstra's Algorithmus

- Laufzeit
  - jede Kante wird genau einmal betrachtet: $\mathcal{O}(|E|)$
  - jeder Knoten wird genau einmal betrachtet: $\mathcal{O}(|V|)$
  - in jeder Iteration wird das Minimum aus Queue entnommen

  $\Rightarrow$ $\mathcal{O}(|V|)$, falls Queue als Liste implementiert

  $\Rightarrow$ $\mathcal{O}(log(|V|))$, falls Queue als balancierter Baum oder Heap implementiert

$\Rightarrow$ Insgesamt: $\mathcal{O}(|V| \cdot log(|V|) + |E|)$ bei geeigneter Datenstruktur

- Dijkstra ist Greedy-Algorithmus, in jedem Schritt wird lokales Optimum gewählt

Korrektheit von Dijkstra's Algorithmus

- wenn $w(v, v') \geqslant 0$ für alle Kanten (v, v'), dann gilt die Invariante:
  - wenn $u \notin queue$, dann ist $d[u] = \delta(s, u)$
- aus Invariante folgt direkt Korrektheit

Theorem

Sei $G = (V, E, w)$ ein gewichteter Graph mit $w(v, v') \geqslant 0$ für alle $(v, v') \in E$. Nach Abschluß des Dijkstra-Algorithmus gilt $d[v] = \delta(s, v)$ für alle $v \in V$.

Anwendungsmöglichkeiten von Dijkstra's Algorithmus

- Routenplaner, ...
- Routenplaner benötigt normalerweise nur kürzesten Weg von a nach b

$\Rightarrow$ Berechnung von Wegen zu allen Zielen b mit Dijkstra erscheint unnötig

- aber: bislang sind keine Algorithmen bekannt, die kürzeste Wege von a nach b asymptotisch effizienter berechnen
- alternative Algorithmen für kürzeste Wege:
  - kürzeste Wege in azyklischen Graphen ($\mathcal{O}(|V| + |E|)$)
  - kürzeste Wege in Graphen mit negativen Kantengewichten (Algorithmus von Bellman-Ford: $\mathcal{O}(|V| \cdot |E|)$)

Kürzeste Wege in azyklischen Graphen

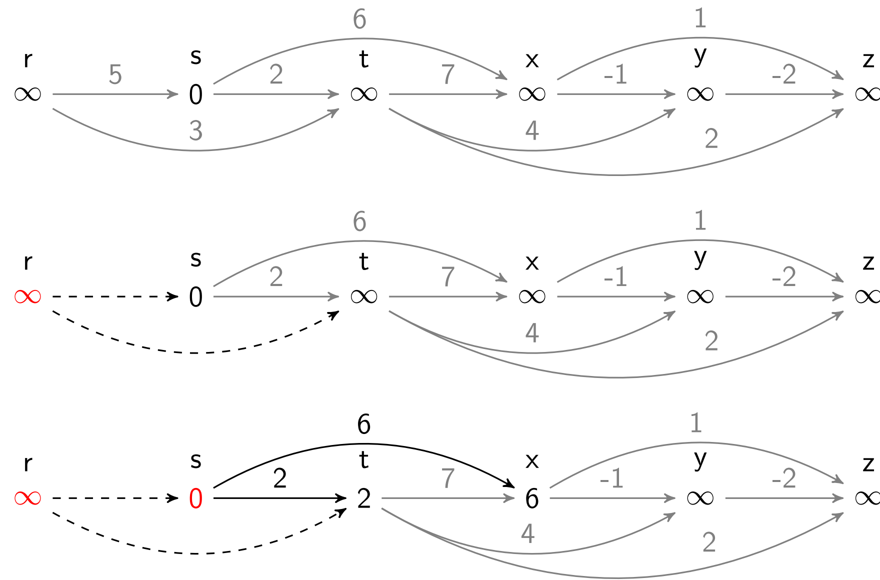
- Gegeben: gewichteter azyklischer Graph (V, E, w) und Startknoten $s \in V$
- Berechnet: kürzeste Wege zu allen anderen Knoten
- Algorithmus:
  1. erzeuge topologische Sortierung
  2. setze $d[v] = \begin{cases} 0, & \text{falls } v = s \\ \infty, & \text{sonst} \end{cases}$
  3. für alle Knoten u in topogischer Reihenfolge
     - für alle Nachfolger v von u: $d[v] = min(d[v], d[u] + w(u, v))$
- Laufzeit: $\mathcal{O}(|V| + |E|)$ für Schritte 1 und 3, $\mathcal{O}(|V|)$ für Schritt 2

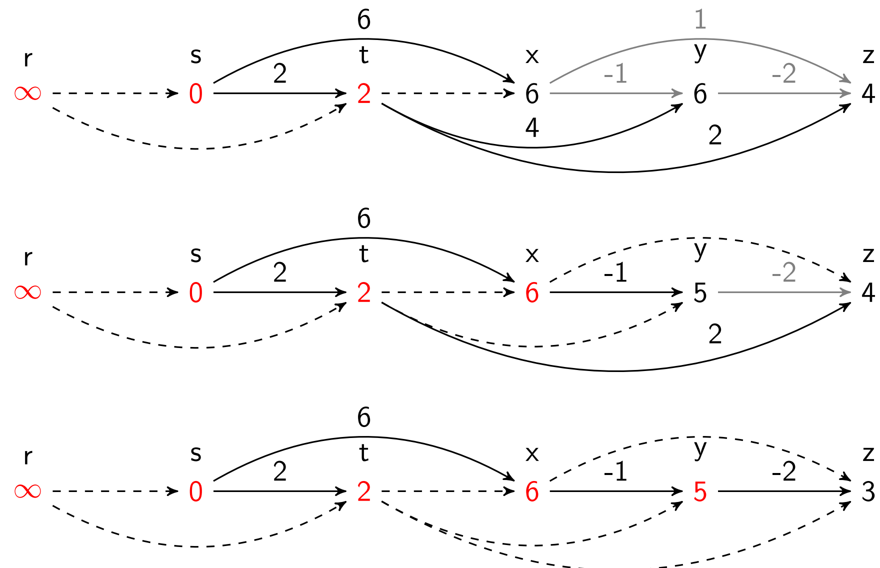
Kürzeste Wege in azyklischen Graphen

Theorem

Sei $G = (V, E, w)$ ein gewichteter azyklischer Graph. Nach Abschluß des Algorithmus gilt $d[v] = \delta(s, v)$ für alle $v \in V$.

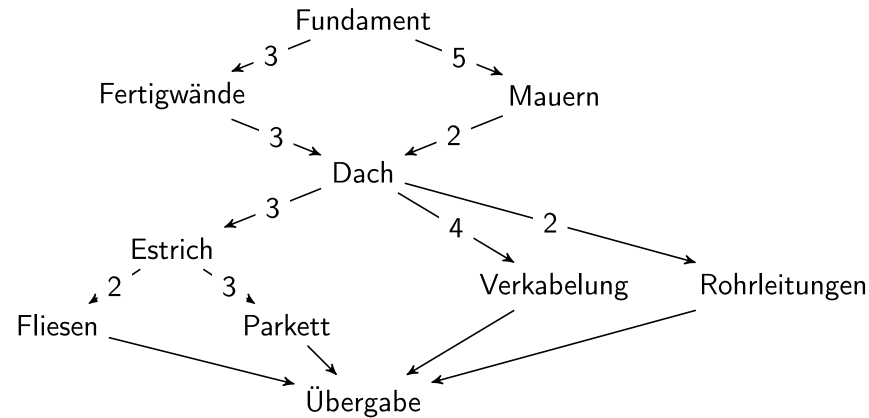
Beispiel





Anwendung

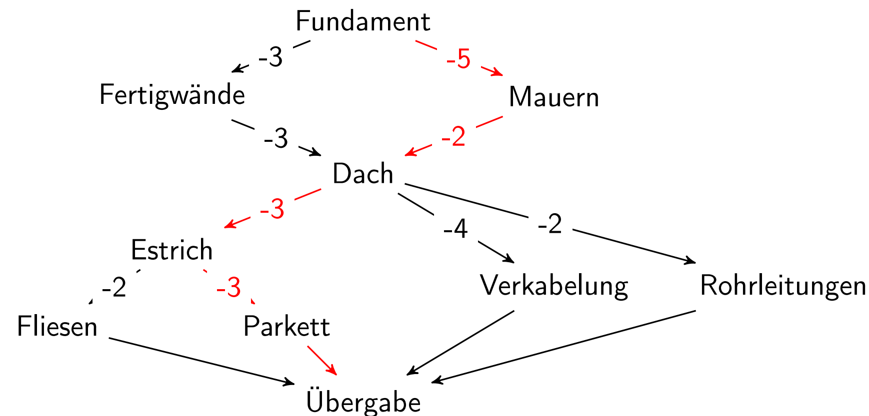
- Aufgabenplanung
- Beispiel: Hausbau



- Werbeslogan: Wir bauen Ihnen jedes Haus in X(?) Wochen!

Aufgabenplanung

- Gesucht: Maximale Ausführungszeit, d.h. längster Weg
- Lösung mittels kürzesten Wegen in azyklischen Graphen: negiere Kosten



$\Rightarrow$ $w(v, v') < 0$ kann sinnvoll sein, geht nicht mit Dijkstra

Zusammenfassung

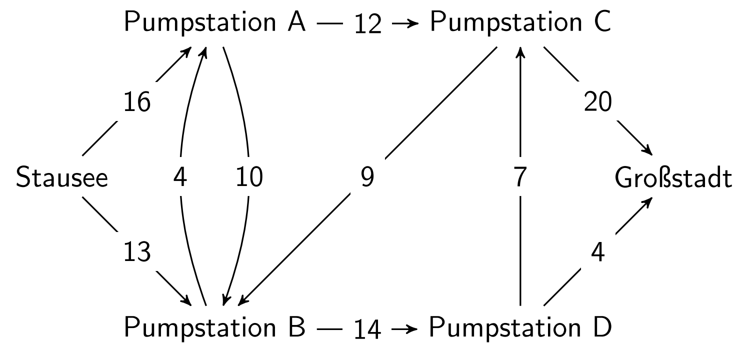
- es gibt verschiedene Verfahren um kürzeste Wege zu bestimmen
- hier nur Verfahren mit festgelegtem Startknoten betrachtet
  - kürzeste Wege in azyklischen Graphen: $\mathcal{O}(|V| + |E|)$, w beliebig
  - kürzeste Wege in beliebigen Graphen (Dijkstra): $\mathcal{O}(|V| \cdot log(|V|) + |E|)$, w nicht-negativ
- weitere Verfahren:
  - Bellman-Ford: kürzeste Wege von gegebenem Startknoten in beliebigen Graphen: $\mathcal{O}(|V| \cdot |E|)$, w beliebig
  - Floyd-Warshall: Verfahren für alle kürzesten Wege ($\mathcal{O}(|V|^3)$)

Übersicht

- Graphen
  - Datenstrukturen für Graphen
  - Topologisches Sortieren
  - Breitensuche
  - Tiefensuche
  - Kürzeste Wege
  - Flüsse in Netzwerken

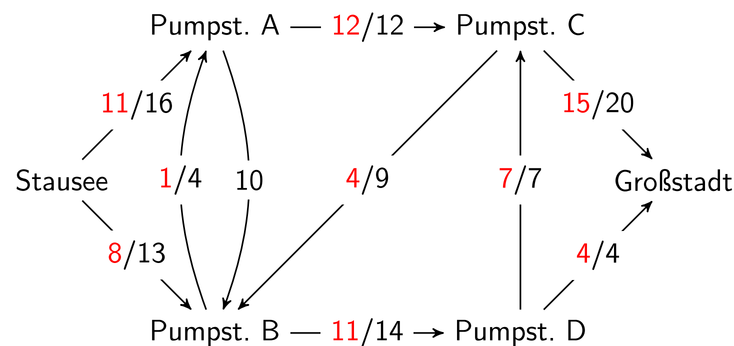
Flussnetzwerke

- Flussnetzwerk ist Graph $G = (V, E)$
  - jede Kante (u, v) hat nicht-negative Kapazität $c(u, v)$
  - es gibt zwei ausgezeichnete Knoten: Quelle s und Senke t
- Beispiel Wasserversorgung



Flüsse

- Fluss ist Funktion $f : (V, V) \to \mathbb{N}$
- Erfüllt 3 Bedingungen
  - Kapazitätsbeschränkung: für alle v, v': $f(v, v') \leqslant c(v, v')$
  - Asymmetrie: für alle v, v': $f(v, v') = -f(v', v)$
  - Flusserhaltung: für alle $v \in V \setminus \{s, t\}$: $\sum_{u \in V} f(v, u) = 0$

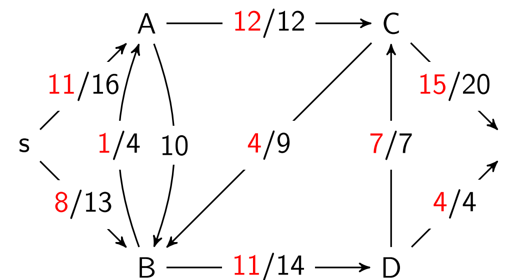


- Wert eines Flusses ist $|f| = \sum_{v \in V} f(s, v)$

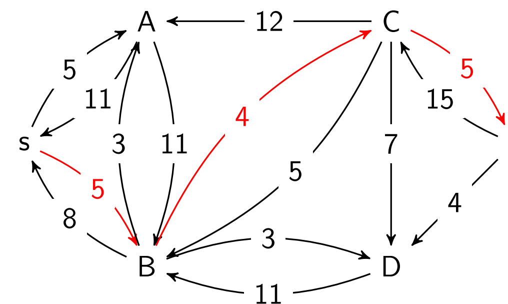
Flussanalyse

- gegeben ein Flussnetzwerk, bestimme maximalen Fluss (mit höchstem Wert)
- nutze dazu Ford-Fulkerson Methode (iterierte Flussverbesserung)
    - starte mit irgendeinem Fluss, z.B. mit f_0, wobei $f_0(u, v) = 0$
    - bestimme Restnetzwerk
        - Restkapazität $c_f(u, v) = c(u, v) - f(u, v)$ (wieviel kann man Fluss auf Kante erhöhen, bevor Kapazität ausgereizt ist)
        - Restnetzwerk $G_f = (V, E_f)$ mit $(u, v) \in E_f$ gdw. $c_f(u, v) > 0$
        - E_f wird auch als Menge der Restkanten bezeichnet
    - falls Weg von s nach t in G_f vorhanden (Erweiterungspfad), bestimme minimale Kapazität m in G_f auf diesem Weg
    - erweitere Fluss in G auf Erweiterungspfad um m und beginne nächste Iteration

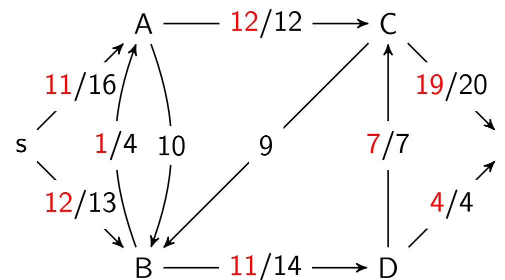
- Flussnetzwerk



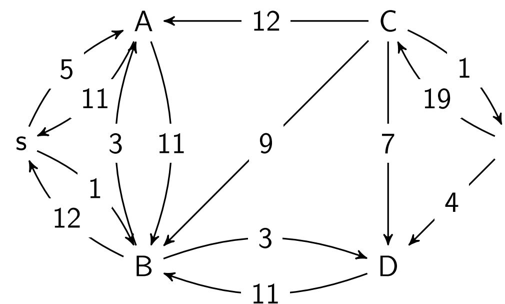
- Restnetzwerk



- erweiterter Fluss



- neues Restnetzwerk



Eigenschaften der Ford-Fulkerson Methode

Theorem

- *die Methode terminiert*
- *nach Abschluss der Methode ist ein maximaler Fluss gefunden*
- *Laufzeit:* $\mathcal{O}(|E| \cdot m)$*, wobei* m *der Wert eines maximalen Flusses ist*

Korrektheit der Ford-Fulkerson Methode

- Schnitt eines Flussnetzwerkes $G = (V, E)$ ist Partitionierung $V = S \uplus T$, so dass $s \in S$ und $t \in T$.
- Nettofluss des Schnitts: $f(S, T) = \sum_{u \in S, v \in T} f(u, v)$
- Kapazität des Schnitts: $c(S, T) = \sum_{u \in S, v \in T} c(u, v)$

⇒ Nettofluss ist Wert des Flusses:

$$
\begin{aligned}
f(S,T) &= f(S,V) - f(S,S) \\
&= f(S,V) && (f(S,S) = 0 \text{ wegen Asymmetrie}) \\
&= f(s,V) + f(S-s,V) \\
&= f(s,V) && (f(S-s,V) = 0 \text{ wegen Flusserhaltung}) \\
&= |f|
\end{aligned}
$$

⇒ Wert eines Flusses ist wegen Kapazitätsbeschränkung durch Kapazität eines beliebigen Schnitts beschränkt: $|f| = f(S, T) \leqslant c(S, T)$

Korrektheit der Ford-Fulkerson Methode

Theorem (maximaler-Fluss = minimaler-Schnitt)

Sei f ein Fluss in $G = (V, E)$. Dann sind folgende Aussagen äquivalent:

1. *f ist maximaler Fluss in G*
2. *Das Restnetzwerk G_f enthält keine Erweiterungspfade.*
3. *Es gilt $|f| = c(S, T)$ für einen Schnitt $V = S \uplus T$.*

Beweis.

$3 \Rightarrow 1$ Da $|f'| \leqslant c(S, T)$ für alle Flüsse f', ist f maximal wegen 3

$1 \Rightarrow 2$ Angenommen, G_f enthält Erweiterungspfad ⇒ erhalte Fluss f' in G_f und Fluss $f + f'$ in G mit $|f + f'| = |f| + |f'| > |f| \Rightarrow \lightning$ zu 1

$2 \Rightarrow 3$
- Definiere $S = \{v \in V \mid v \text{ von } s \text{ in } G_f \text{ erreichbar}\}$, $T = V \setminus S$
- Wegen 2 ist $s \in S$ und $t \in T \Rightarrow V = S \uplus T$ ist Schnitt
- Für $u \in S$ und $v \in T$ gilt $f(u, v) = c(u, v)$, da andernfalls $(u, v) \in E_f$
- Da Nettowert = Wert des Flusses: $|f| = f(S, T) = c(S, T)$

■

Effizientere Berechnung von maximalem Fluss

- Wunsch: Verfahren mit Laufzeit abhängig von $|V|$ und $|E|$, aber nicht vom c
- Einfache Verbesserung: Edmonds-Karp-Verfahren: $\mathcal{O}(|V| \cdot |E|^2)$
- Komplexere Verfahren: Push-Relabel-Algorithmen: $\mathcal{O}(|V|^2 \cdot |E|)$ und $\mathcal{O}(|V|^3)$

Edmonds-Karp-Algorithmus

- Nutze Ford-Fulkerson Methode
- Nutze zur Suche der Erweiterungspfade eine Breitensuche

Lemma

Sei $v \in V \setminus \{s, t\}$. Im Laufe des Edmonds-Karp-Algorithmus steigt der Abstand von s zu v in G_f monoton an (bei Durchführung jeder Flusserweiterung)

(wenn während des Algorithmus erst Fluss f und später Fluss f' genutzt wird, dann gilt $\delta_f(s, v) \leqslant \delta_{f'}(s, v)$)

Komplexität des Edmonds-Karp-Algorithmus

Theorem

Es dauert maximal $\mathcal{O}(|V| \cdot |E|)$ viele Iterationen, bis der Edmonds-Karp-Algorithmus den maximalen Fluss liefert.

- Kante auf Erweiterungspfad p heisst kritisch, gdw. diese Kante minimale Kapazität auf p hat
- bei Flusserw. verschwinden kritischen Kanten von p im Restnetzwerk
- jede Kante $(u, v) \in E$ kann höchstens $\frac{|V|}{2}$ mal kritisch werden
  - sei $(u, v) \in E$; sei f der Fluss, wenn (u, v) erstmalig kritisch wird
  - da Erweiterungspfade kürzeste Pfade sind, gilt $\delta_f(s, v) = \delta_f(s, u) + 1$
  - (u, v) ist erst wieder im Restnetzwerk, wenn (v, u) in einem Erweiterungspfad vorkommt (beim Fluss f')
  - wegen kürzester Pfade gilt wiederum: $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$
  - mit Lemma: $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geqslant \delta_f(s, v) + 1 = \delta_f(s, u) + 2$
  - da $\delta_*(s, u)$ maximal $|V|$ ist, kann (u, v) maximal $\frac{|V|}{2}$ mal kritisch sein

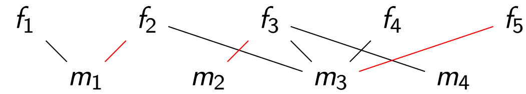
$\Rightarrow$ es gibt $\mathcal{O}(|V| \cdot |E|)$ viele kritische Kanten während Algorithmus

Erweiterungen von Flussnetzwerken

- bislang: Flussnetzwerke mit einer Quelle und einer Senke
- Erweiterung: mehrere Quellen und mehrere Senken
- Beispiel: 3 Großstädte werden über 2 Stauseen versorgt
- Nutze gleiche Verfahren wie bisher zur Analyse, indem man Superquelle und Supersenke einführt:
  - Sei $G = (V, E)$ und c gegeben, seien $\{s_1, \dots, s_m\} \subseteq V$ Quellen und $\{t_1, \dots, t_n\} \subseteq V$ Senken
  - Bilde daraus neues Flussnetzwerk mit einer Quelle und Senke
    - $G' = (V \uplus \{s, t\}, E \uplus E')$
    - $E' = \{(s, s_i) \mid 1 \leqslant i \leqslant m\} \cup \{(t_i, t) \mid 1 \leqslant i \leqslant n\}$
    - $c'(x, y) = \begin{cases} c(x, y), & \text{falls } x, y \in V \\ \infty, & \text{falls } (x = s \text{ und } y = s_i) \text{ oder } (x = t_i \text{ und } y = t) \end{cases}$
    - Quelle in G' ist s, Senke in G' ist t

Anwendung von Flussnetzwerken

- bipartites Matching-Problem (Hochzeits-Problem)
- zwei Knotensätze: Frauen und Männer
- ungerichtete Kanten: $(f, m) \in E$ gdw. f und m mögen sich gegenseitig
- Ziel: spiele Verkuppler, so dass möglichst viele Pärchen entstehen



- Lösung mittels Flussnetzwerken (alle Kapazitäten: 1)