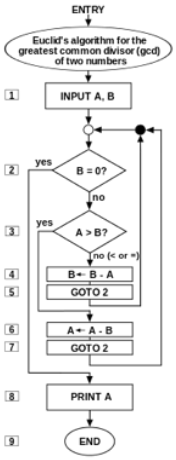
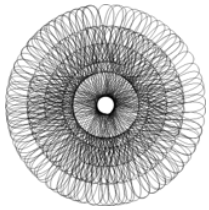
Algorithmen und Komplexität

Stephan Schulz
stephan.schulz@dhbw-stuttgart.de

Jan Hladik
jan.hladik@dhbw-stuttgart.de





DHBW Stuttgart
Duale Hochschule Baden-Württemberg

Inhaltsverzeichnis

Semesterübersicht

- Was sind Algorithmen?
- Wie kann man die Komplexität von Algorithmen beschreiben?
  - Platzbedarf
  - Zeitbedarf
- Mathematische Werkzeuge zur Komplexitätsanalyse
  - Z.B. Rekurrenzrelationen
- Klassifikation von Algorithmen
  - Z.B. Brute Force, Greedy, Divide&Conquer, Dynamic Programming
  - Ansätze zur Algorithmenentwicklung
- Algorithmen und Datenstrukturen
  - Arrays
  - Listen
  - Suchbäume
  - Tries
  - Hashes
  - Graphen

Sonstiges zum Kurs

- Begleitendes Labor *Angewandte Informatik*
  - Algorithmentwicklung in C
- Webseiten zum Kurs:
  `http://wwwlehre.dhbw-stuttgart.de/~sschulz/algo2015.html`
  `http://wwwlehre.dhbw-stuttgart.de/~hladik/Algorithmen/`
  - Folienskript zur Vorlesung
  - Aufgaben zum Labor
  - Musterlösungen

Literatur

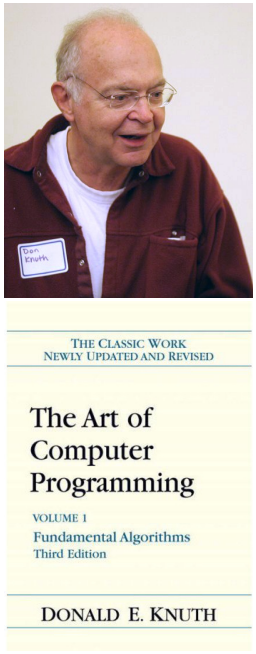
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
  - 3. Auflage 2009, moderner Klassiker (der Titel lügt)

Literatur

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
  - 3. Auflage 2009, moderner Klassiker (der Titel lügt)
- Robert Sedgewick, Kevin Wayne: *Algorithms*
  - 4. Auflage 2011, moderner Klassiker
  - Ähnlich auch als *Algorithms in C/Java/C++*

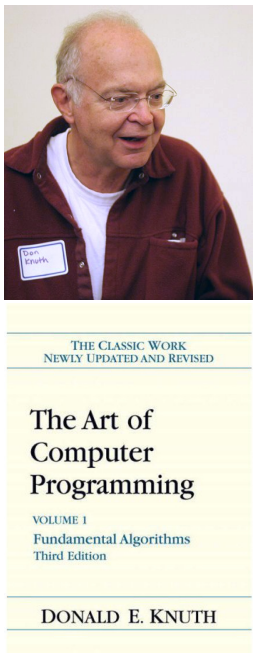
Literatur

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
  - 3. Auflage 2009, moderner Klassiker (der Titel lügt)
- Robert Sedgewick, Kevin Wayne: *Algorithms*
  - 4. Auflage 2011, moderner Klassiker
  - Ähnlich auch als *Algorithms in C/Java/C++*
- Donald Knuth: *The Art of Computer Programming*
  - Die Bibel, seit 1962, Band 1 1968 (3. Auflage 1997), Band 4a 2011, Band 5 geplant für 2020 (!)



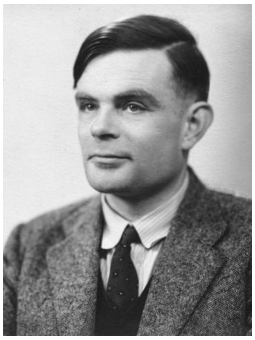
Literatur

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
  - 3. Auflage 2009, moderner Klassiker (der Titel lügt)
- Robert Sedgewick, Kevin Wayne: *Algorithms*
  - 4. Auflage 2011, moderner Klassiker
  - Ähnlich auch als *Algorithms in C/Java/C++*
- Donald Knuth: *The Art of Computer Programming*
  - Die Bibel, seit 1962, Band 1 1968 (3. Auflage 1997), Band 4a 2011, Band 5 geplant für 2020 (!)
- Niklaus Wirth: *Algorithmen und Datenstrukturen*
  - Deutschsprachiger Klassiker, 1991



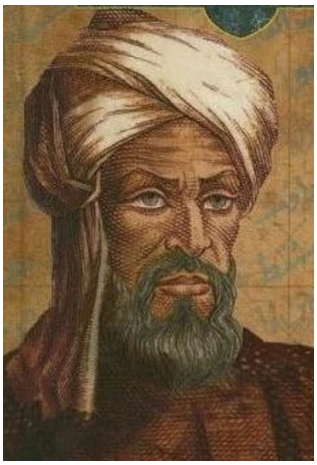
Informelle Definition: Algorithmus

- Ein Algorithmus ist ein Verfahren zur Lösung eines Problems oder einer Problemklasse
- Ein Algorithmus. . .
  - . . . überführt eine Eingabe in eine Ausgabe
  - . . . besteht aus endlich vielen Einzelschritten
  - . . . ist auch für Laien durchführbar
  - Jeder Einzelschritt ist wohldefiniert, ausführbar, und terminiert nach endlicher Zeit
  - Gelegentliche Forderung: Der Algorithmus terminiert (problematisch)
- Formalisierung: z.B. Turing-Maschine



Der Begriff *Algorithmus*

Der Begriff *Algorithmus*



Mohammed al-Chwarizmi

- Mathematiker, Astronom, Geograph
- geboren ca. 780 nahe des Aralsees (heute Usbekistan)
- gestorben ca. 850 in Bagdad
- zahlreiche Werke
  - *Algoritmi de numero Indorum*
  - Rechenverfahren
  - Algebra

Einige Klassen von Algorithmen

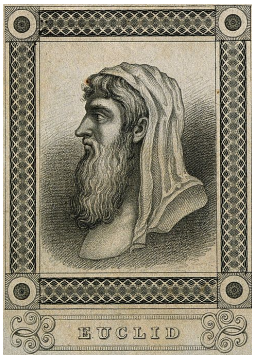
- Suchalgorithmen
  - Schlüssel in Datenbank
  - Pfad in Umgebung/Landkarte/Graph
  - Beweis in Ableitungsraum
- Sortierverfahren
  - Total
  - Topologisch
- Optimierungsverfahren
  - Spielpläne
  - Kostenminimierung
- Komprimierung
  - Lossy
  - Lossless
- Mathematische Algorithmen
  - Faktorisierung
  - Größter gemeinsamer Teiler
  - Gauß-Verfahren
- ...

Beispiel: Euklids GGT-Algorithmus

- Problem: Finde den größten gemeinsamen Teiler (GGT) (*greatest common divisor, GCD*) für zwei natürliche Zahlen *a* und *b*.
- Also: Eingabe $a, b \in \mathbb{N}$
- Ausgabe: $c \in \mathbb{N}$ mit folgenden Eigenschaften:
  - *c* teilt *a* ohne Rest
  - *c* teilt *b* ohne Rest
  - *c* ist die größte natürliche Zahl mit diesen Eigenschaften
- Beispiele:
  - $\text{ggt}(12, 9) = 3$
  - $\text{ggt}(30, 15) = 15$
  - $\text{ggt}(25, 11) = 1$

Beispiel: Euklids GGT-Algorithmus

- Berchnung des GGT in Euklids Worten:
  Εἰ δὲ οὐ μετρεῖ ὁ **ΓΔ** τὸν **ΑΒ**, τῶν **ΑΒ**, **ΓΔ** ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεταί τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται: εἰ δὲ μή,



Euklid von Alexandria (ca. 3 Jh. v. Chr.), *Elemente*

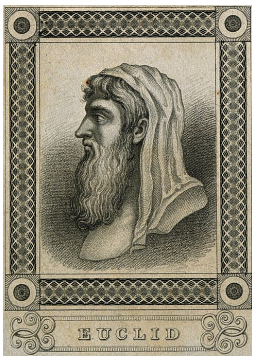
Beispiel: Euklids GGT-Algorithmus

- Berchnung des GGT in Euklids Worten:

  Εἰ δὲ οὐ μετρεῖ ὁ **ΓΔ** τὸν **ΑΒ**, τῶν **ΑΒ**, **ΓΔ** ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεταί τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται: εἰ δὲ μή,

  Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.

  Elemente, Buch VII, Behauptung 2



Euklid von Alexandria (ca. 3 Jh. v. Chr.), *Elemente*

Beispiel: Euklids GGT-Algorithmus

Euklids Algorithmus moderner:

- Gegeben: Zwei natürliche Zahlen a und b
- Wenn $a = b$: Ende, der GGT ist a
- Ansonsten:
  - Sei c die absolute Differenz von a und b.
  - Bestimme den GGT von c und dem kleineren der beiden Werte a und b

Der *Pharos* von Alexandria, Bild: Emad Victor Shenouda

Übung: Euklids GGT-Algorithmus

- Algorithmus
  - Gegeben: Zwei natürliche Zahlen a und b
  - Wenn $a = b$: Ende, der GGT ist a
  - Ansonsten: Sei c die absolute Differenz von a und b.
  - Bestimme den GGT von c und dem kleineren der beiden Werte a und b
- Aufgabe: Bestimmen Sie mit Euklids Algorithmus die folgenden GGTs. Notieren Sie die Zwischenergebnisse.
  - $\text{ggt}(16, 2)$
  - $\text{ggt}(36, 45)$
  - $\text{ggt}(17, 2)$
  - $\text{ggt}(121, 55)$
  - $\text{ggt}(2, 0)$

Spezifikation von Algorithmen

Algorithmen können auf verschiedene Arten beschrieben werden:

- Informeller Text
- Semi-Formaler Text
- Pseudo-Code
- Konkretes Programm in einer Programmiersprache
- Flussdiagramm
- ...

Euklid als (semi-formaler) Text

- Algorithmus: Größter gemeinsamer Teiler
- Eingabe: Zwei natürliche Zahlen a, b
- Ausgabe: Größter gemeinsamer Teiler von a und b

1. Wenn a gleich 0, dann ist das Ergebnis b. Ende.
2. Wenn b gleich 0, dann ist das Ergebnis a. Ende.
3. Wenn a größer als b ist, dann setze a gleich $a - b$.
   Mache mit Schritt 3 weiter.
4. Wenn b größer als a ist, dann setze b gleich $b - a$.
   Mache mit Schritt 3 weiter.
5. Ansonsten: a ist gleich b, und ist der gesuchte GGT. Ende.

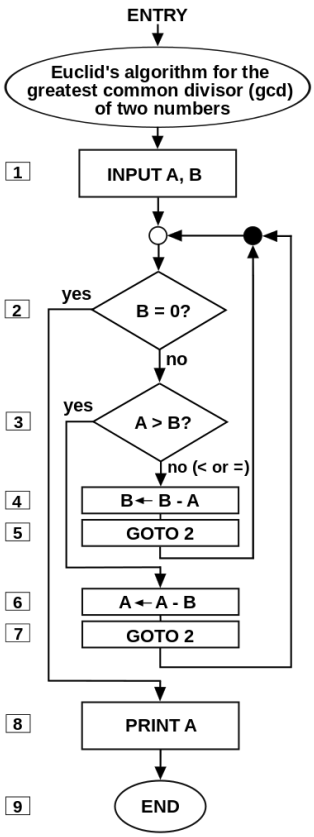
Euklid als (Pseudo-)Code

```
def euclid_gcd(a, b):
    """
    Compute the Greatest Common Divisor of two numbers, usin
    Euclid's naive algorithm.
    """
    if a==0:
        return b
    if b==0:
        return a
    while a!=b:
        if a>b:
            a=a-b
        else:
            b=b-a
    return a
```

Euklid Rekursiv

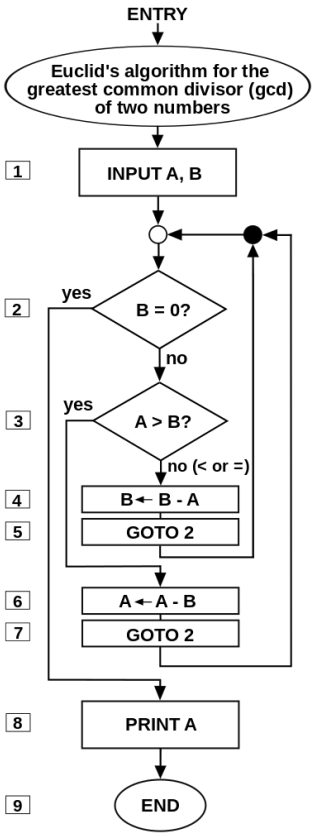
```
def euclid_gcdr(a, b):
    """
    Compute the Greatest Common Divisor of two numbers, usin
    Euclid's naive algorithm.
    """
    if a==0:
        return b
    if b==0:
        return a
    if a>b:
        return euclid_gcdr(a-b,b)
    else:
        return euclid_gcdr(b,b-a)
```


Panta Rhei



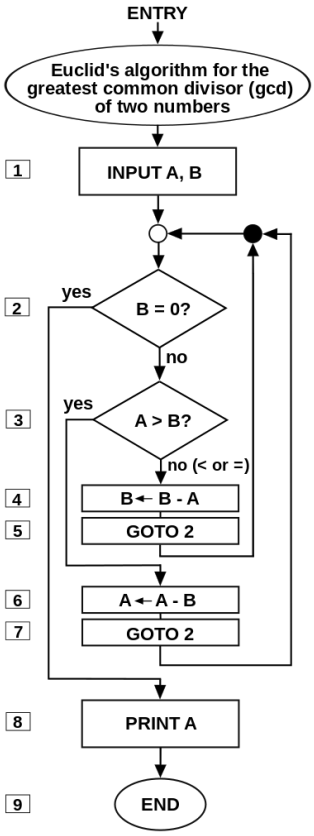
- Flussdiagramm: Graphische Visualisierung des Algorithmus

Panta Rhei



- Flussdiagramm: Graphische Visualisierung des Algorithmus
- Wer findet den Fehler?

Panta Rhei



- Flussdiagramm: Graphische Visualisierung des Algorithmus
- Wer findet den Fehler?

Was passiert bei $A = 0, B \neq 0$?

Übung: Euklids Worst Case

- Wie oft durchläuft der Euklidsche Algorithmus im schlimmsten Fall für ein gegebenes $a + b$ die Schleife? Begründen Sie Ihr Ergebnis!

Ende Vorlesung 1

Das Geheimnis des Modulus

- Der modulo-Operator ermittelt den Divisionsrest bei der ganzzahligen Division:
  - Sei z.B. $z = nq + r$
  - Dann ist

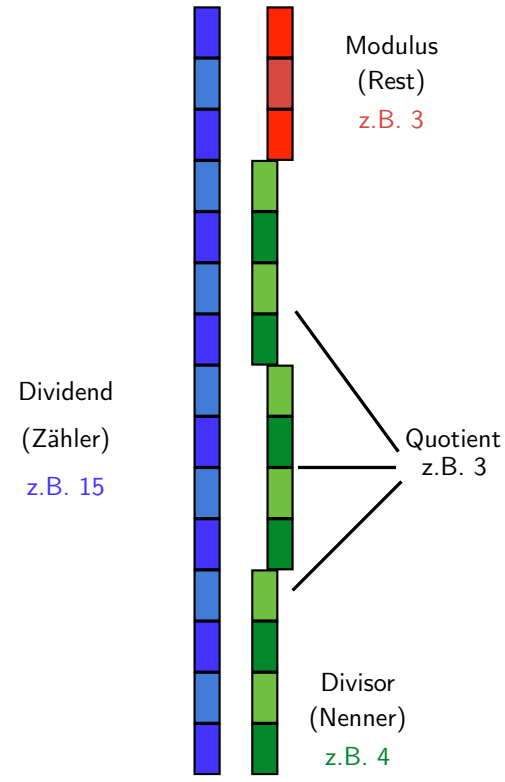
    $$z/n = q \text{ mit Rest } r$$

  - oder auch:

    $$z/n = q \text{ und } z\%n = r$$

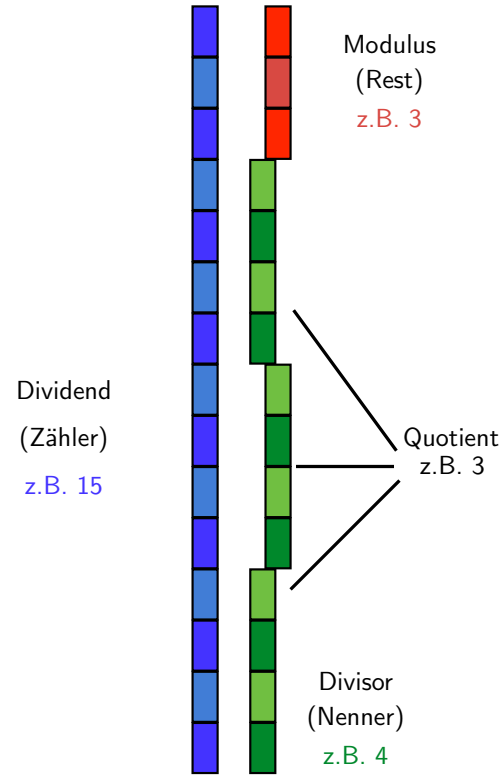
  - Alternative Schreibweise:

    $$z \operatorname{div} n = q \text{ und } z \bmod n = r$$



Modulus Teil 2

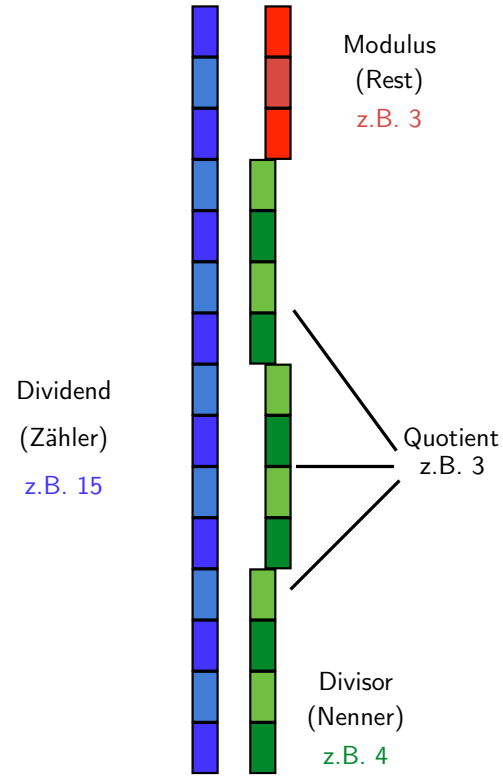
- Eigenschaften:
  - Der Divisionsrest modulo n liegt zwischen 0 und n
  - Aufeinanderfolgende Zahlen kleiner n haben aufeinanderfolgende Divisionsreste
  - Aufeinanderfolgende Zahlen haben meistens aufeinanderfolgende Divisionsreste (Ausnahme: Die größere ist glatt durch n teilbar)
- Verwendung:
  - Kryptographie (RSA)
  - Hashing
  - “Faire” Verteilung auf n Töpfe



Modulus Teil 3

Beispiele

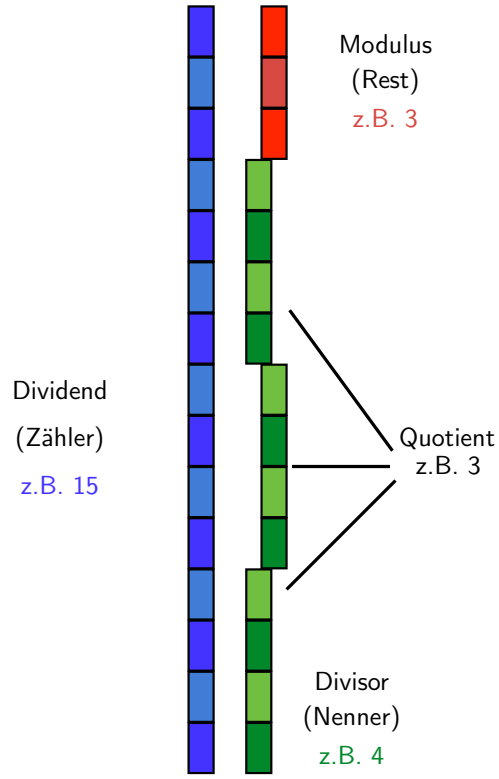
Divident	Divisor	Quotient	Modulus
0	3	0	0
1	3	0	1
2	3	0	2
3	3	1	0
4	3	1	1
5	3	1	2
6	3	2	0
25	17	1	8
26	17	1	9
27	17	1	10
34	17	2	0
35	17	2	1



Modulus Teil 3

Beispiele

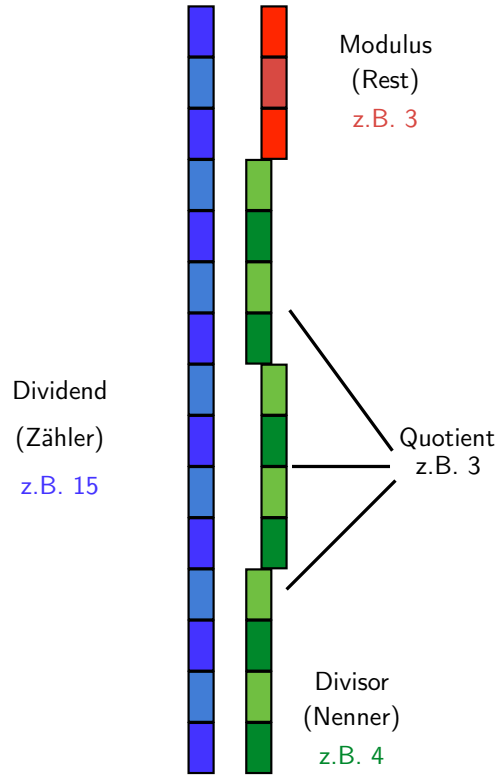
Divident	Divisor	Quotient	Modulus
0	3	0	0
1	3	0	1
2	3	0	2
3	3	1	0
4	3	1	1
5	3	1	2
6	3	2	0
25	17	1	8
26	17	1	9
27	17	1	10
34	17	2	0
35	17	2	1
37	1		



Modulus Teil 3

Beispiele

Divident	Divisor	Quotient	Modulus
0	3	0	0
1	3	0	1
2	3	0	2
3	3	1	0
4	3	1	1
5	3	1	2
6	3	2	0
25	17	1	8
26	17	1	9
27	17	1	10
34	17	2	0
35	17	2	1
37	1	37	0



Übung: Modulus

11 mod 15	=	19 mod 23	=	52 mod 2	=
82 mod 12	=	54 mod 29	=	66 mod 10	=
44 mod 26	=	12 mod 16	=	23 mod 15	=
96 mod 20	=	26 mod 15	=	87 mod 17	=
93 mod 26	=	64 mod 14	=	68 mod 20	=
99 mod 14	=	15 mod 25	=	36 mod 23	=
34 mod 19	=	28 mod 27	=	46 mod 14	=
71 mod 24	=	84 mod 24	=	62 mod 20	=
76 mod 27	=	21 mod 20	=	38 mod 17	=
96 mod 23	=	36 mod 14	=	44 mod 13	=
35 mod 25	=	72 mod 29	=	32 mod 7	=

Übung: Modulus

11 % 15 =

Übung: Modulus

11 % 15	=	11	19 % 23	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=							

23

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	

23

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=							

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=	8	44 % 13	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=	8	44 % 13	=	5
35 % 25	=							

23

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=	8	44 % 13	=	5
35 % 25	=	10	72 % 29	=				

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=	8	44 % 13	=	5
35 % 25	=	10	72 % 29	=	14	32 % 7	=	

Übung: Modulus

11 % 15	=	11	19 % 23	=	19	52 % 2	=	0
82 % 12	=	10	54 % 29	=	25	66 % 10	=	6
44 % 26	=	18	12 % 16	=	12	23 % 15	=	8
96 % 20	=	16	26 % 15	=	11	87 % 17	=	2
93 % 26	=	15	64 % 14	=	8	68 % 20	=	8
99 % 14	=	8	15 % 25	=	15	36 % 23	=	13
34 % 19	=	16	28 % 27	=	1	46 % 14	=	4
71 % 24	=	23	84 % 24	=	12	62 % 20	=	2
76 % 27	=	22	21 % 20	=	1	38 % 17	=	4
96 % 23	=	4	36 % 14	=	8	44 % 13	=	5
35 % 25	=	10	72 % 29	=	14	32 % 7	=	4

Geschafft!

GGT-Algorithmus von Euklid

- Algorithmus: Größter gemeinsamer Teiler
- Eingabe: Zwei natürliche Zahlen a, b
- Ausgabe: Größter gemeinsamer Teiler von a und b
  1. Wenn a gleich 0, dann ist das Ergebnis b. Ende.
  2. Wenn b gleich 0, dann ist das Ergebnis a. Ende.
  3. Wenn a größer als b ist, dann setze a gleich $a - b$. Mache mit Schritt 3 weiter.
  4. Wenn b größer als a ist, dann setze b gleich $b - a$. Mache mit Schritt 3 weiter.
  5. Ansonsten: a ist gleich b, und ist der gesuchte GGT. Ende.

Analyse: Euklids GGT-Algorithmus

Sei *o.B.d.A* a größer als b und sei g der ggt(a, b)

- Dann gilt: $a = mg$ und $b = ng$ für $m, n \in \mathbb{N}$ und $m > n$
- Nach einem Schritt ist also $a = (m - n)g$ und $b = ng$
  - g teilt immer noch a und b (Korrektheit!)
  - Wenn m groß gegen n ist, dann durchläuft der Algorithmus viele Schritte, bis $a \leq b$ gilt

Analyse: Euklids GGT-Algorithmus

Sei *o.B.d.A* a größer als b und sei g der ggt(a, b)

- Dann gilt: $a = mg$ und $b = ng$ für $m, n \in \mathbb{N}$ und $m > n$
- Nach einem Schritt ist also $a = (m - n)g$ und $b = ng$
  - g teilt immer noch a und b (Korrektheit!)
  - Wenn m groß gegen n ist, dann durchläuft der Algorithmus viele Schritte, bis $a \leq b$ gilt

Geht das auch schneller?

25

Euklid Schneller

Sei *o.B.d.A* a größer als b und sei g der ggt(a, b)

- Dann gilt: $a = mg$ und $b = ng$ für $m, n \in \mathbb{N}$ und $m > n$
- Nach einem Schritt ist also $a = (m - n)g$ und $b = ng$
  - g teilt immer noch a und b (Korrektheit!)
  - Wenn m groß gegen n ist, dann durchläuft der Algorithmus viele Schritte, bis $a \leq b$ gilt
- Beobachtung: Es wird so lange immer wieder b von a abgezogen, bis $a \leq b$ gilt!
  - Sei im folgenden a' der Originalwert von a
  - Wenn wir b i-mal von a abziehen, so gilt also: $a' = ib + a$
  - In anderen Worten: a ist der Divisionsrest von a'/b!

Euklid Schneller

Sei *o.B.d.A* a größer als b und sei g der ggt(a, b)

- Dann gilt: $a = mg$ und $b = ng$ für $m, n \in \mathbb{N}$ und $m > n$
- Nach einem Schritt ist also $a = (m - n)g$ und $b = ng$
  - g teilt immer noch a und b (Korrektheit!)
  - Wenn m groß gegen n ist, dann durchläuft der Algorithmus viele Schritte, bis $a \leq b$ gilt
- Beobachtung: Es wird so lange immer wieder b von a abgezogen, bis $a \leq b$ gilt!
  - Sei im folgenden a' der Originalwert von a
  - Wenn wir b i-mal von a abziehen, so gilt also: $a' = ib + a$
  - In anderen Worten: a ist der Divisionsrest von a'/b!

Wir können also die wiederholten Subtraktionen durch eine Division mit Restberechnung ersetzen!

Euklid Schneller in (Pseudo)-Code

```
def euclid_gcd2(a, b):
    """ Compute the Greatest Common Divisor of two numbers,
        using an improved version od Euclid's  algorithm.
    """
    while a!=b:
        if a==0:
            return b
        if b==0:
            return a
        if a>b:
            a=a%b
        else:
            b=b%a
    return a
```


Übung: Euklid Schneller

- Aufgabe: Bestimmen Sie mit dem verbesserten Algorithmus die folgenden GGTs. Notieren Sie die Zwischenergebnisse.
  - ggt(16, 2)
  - ggt(36, 45)
  - ggt(17, 2)
  - ggt(121, 55)
  - ggt(89, 55)

```
def euclid_gcd2(a, b):
    while a!=b:
        if a==0:
            return b
        if b==0:
            return a
        if a>b:
            a=a%b
        else:
            b=b%a
    return a
```

Übung: Datenstrukturen

- Zur Lösung eines gegebenen Problems kann es verschieden effiziente Algorithmen geben
  - Oft wesentlich: Geschickte Organisation der Daten durch geeignete Datenstrukturen
- Übung: Finden Sie die zu den 5 Namen in Spalte 1 gehörende Ziffernfolge in Liste 1. Stoppen Sie Ihre Zeit!

Spalte 1
Dummy 1
Dummy 2
Dummy 3
Dummy 4
Dummy 5

Übung: Datenstrukturen

- Zur Lösung eines gegebenen Problems kann es verschieden effiziente Algorithmen geben
  - Oft wesentlich: Geschickte Organisation der Daten durch geeignete Datenstrukturen
- Übung: Finden Sie die zu den 5 Namen in Spalte 1 gehörende Ziffernfolge in Liste 1. Stoppen Sie Ihre Zeit!

Spalte 1
Stone, Jo
Pierce, Jaime
Nunez, Glenda
Hawkins, Mona
Massey, Harold

Übung: Datenstrukturen

- Zur Lösung eines gegebenen Problems kann es verschieden effiziente Algorithmen geben
  - Oft wesentlich: Geschickte Organisation der Daten durch geeignete Datenstrukturen
- Übung: Finden Sie die zu den 5 Namen in Spalte 2 gehörende Ziffernfolge in Liste 2. Stoppen Sie Ihre Zeit!

Spalte 2
Mcdonald, Jeffrey
Palmer, Katie
Pierce, Jaime
Schmidt, Tami
French, Erica

Übung: Datenstrukturen

- Zur Lösung eines gegebenen Problems kann es verschieden effiziente Algorithmen geben
  - Oft wesentlich: Geschickte Organisation der Daten durch geeignete Datenstrukturen
- Übung: Finden Sie die zu den 5 Namen in Spalte 3 gehörende Ziffernfolge in Liste 3. Stoppen Sie Ihre Zeit!

Spalte 3
Sims, Helen
Obrien, Kim
Curry, Courtney
Brewer, Marcella
Thornton, Dwight

Datenstrukturen?

- Liste 1: Unsortiertes Array
  - Lineare Suche
- Liste 2: Sortiertes Array
  - Binäre Suche (oder auch auch “gefühlte Interpolationssuche”)
- Liste 3: Sortiertes Array mit pre-Hashing
  - Ditto, aber mit besserem Einstiegspunkt

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”
— Linus Torvalds

Datenstrukturen?

- Liste 1: Unsortiertes Array
  - Lineare Suche
- Liste 2: Sortiertes Array
  - Binäre Suche (oder auch auch “gefühlte Interpolationssuche”)
- Liste 3: Sortiertes Array mit pre-Hashing
  - Ditto, aber mit besserem Einstiegspunkt

Wahl der geeigneten Datenstruktur ermöglicht bessere/effizientere Suche!

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”
— Linus Torvalds

Komplexität von Algorithmen

- Fragestellung: Wie teuer ist ein Algorithmus?
- Konkreter:
  - Wie viele (elementare) Schritte braucht er für einen gegebene Eingabe?
  - Wie viel Speicherplatz bracht er für einen gegebene Eingabe?

Komplexität von Algorithmen

- Fragestellung: Wie teuer ist ein Algorithmus?
- Konkreter:
  - Wie viele (elementare) Schritte braucht er für einen gegebene Eingabe?
  - Wie viel Speicherplatz bracht er für einen gegebene Eingabe?
- Allgemeiner:
  - Wie viele elementare Schritte braucht ein Algorithmus für Eingaben einer bestimmten Länge?
    - . . . im Durchschnitt?
    - . . . schlimmstenfalls?
  - Wie viel Speicher braucht ein Algorithmus für Eingaben einer bestimmten Länge?
    - . . . im Durchschnitt?
    - . . . schlimmstenfalls?

Effizienz und Auswahl von Algorithmen

- Was ist der beste Algorithmus für ein gegebenes Problem?
  - Verschiedene Algorithmen können sehr verschiedene Komplexität haben (Beispiel: Euklid!)
- Kriterien:
  - Performanz auf erwarteten Eingabedaten!
  - Performanz im Worst-Case
  - Anforderungen der Umgebung (Echtzeit? Begrenzter Speicher?)

Effizienz und Auswahl von Algorithmen

- Was ist der beste Algorithmus für ein gegebenes Problem?
  - Verschiedene Algorithmen können sehr verschiedene Komplexität haben (Beispiel: Euklid!)
- Kriterien:
  - Performanz auf erwarteten Eingabedaten!
  - Performanz im Worst-Case
  - Anforderungen der Umgebung (Echtzeit? Begrenzter Speicher?)
  - (Nachweisbare) Korrektheit!

Effizienz und Auswahl von Algorithmen

- Was ist der beste Algorithmus für ein gegebenes Problem?
  - Verschiedene Algorithmen können sehr verschiedene Komplexität haben (Beispiel: Euklid!)
- Kriterien:
  - Performanz auf erwarteten Eingabedaten!
  - Performanz im Worst-Case
  - Anforderungen der Umgebung (Echtzeit? Begrenzter Speicher?)
  - (Nachweisbare) Korrektheit!

Eleganz und Einfachheit sind schwer zu quantifizieren, aber ebenfalls wichtig!

Ende Vorlesung 2

Komplexitätsfragen

Die Zeit, die ein Computer für eine Operation braucht, hängt ab von

- Art der Operation (Addition ist einfacher als Logarithmus)
- Speicherort der Operanden (Register, Cache, Hauptspeicher, Swap)
- Länge der Operanden (8/16/32 Bit)
- Taktrate des Prozessors
- Programmiersprache / Compiler

Komplexitätsfragen

Die Zeit, die ein Computer für eine Operation braucht, hängt ab von

- Art der Operation (Addition ist einfacher als Logarithmus)
- Speicherort der Operanden (Register, Cache, Hauptspeicher, Swap)
- Länge der Operanden (8/16/32 Bit)
- Taktrate des Prozessors
- Programmiersprache / Compiler

Diese Parameter hängen ihrerseits ab von

- Prozessormodell
- Größe von Cache und Hauptspeicher
- Laufzeitbedingungen
  - Wie viele andere Prozesse?
  - Wie viel freier Speicher?

Komplexitätsfragen

Die Zeit, die ein Computer für eine Operation braucht, hängt ab von

- Art der Operation (Addition ist einfacher als Logarithmus)
- Speicherort der Operanden (Register, Cache, Hauptspeicher, Swap)
- Länge der Operanden (8/16/32 Bit)
- Taktrate des Prozessors
- Programmiersprache / Compiler

Diese Parameter hängen ihrerseits ab von

- Prozessormodell
- Größe von Cache und Hauptspeicher
- Laufzeitbedingungen
  - Wie viele andere Prozesse?
  - Wie viel freier Speicher?

Exakte Berechnung ist extrem aufwendig und umgebungsabhängig

Komplexität abstrakt

Um von den genannten Problemen zu abstrahieren, d.h. Komplexitätsberechnungen praktikabel und umgebungsunabhängig zu gestalten, definiert man:

- eine Zuweisung braucht 1 Zeiteinheit
- eine arithmetische Operation braucht 1 ZE
  - moderne Prozessoren arbeiten mit 64-Bit-Integers
  - Vereinfachung ist legitim für Zahlen von -9 bis +9 Trillionen
  - Ausnahmen für extrem große Zahlen ($\leadsto$ Kryptographie)
- ein Vergleich (`if, while, for`) braucht 1 ZE

Komplexität und Eingabe (1)

Bei (fast) allen Algorithmen hängt die Laufzeit von der Größe der Eingabe ab

- Suchen/Sortieren: Anzahl der Elemente
- Matrix-Multiplikation: Dimensionen der Matrizen
- Graph-Operationen: Anzahl der Knoten/Kanten

Komplexität und Eingabe (1)

Bei (fast) allen Algorithmen hängt die Laufzeit von der Größe der Eingabe ab

- Suchen/Sortieren: Anzahl der Elemente
- Matrix-Multiplikation: Dimensionen der Matrizen
- Graph-Operationen: Anzahl der Knoten/Kanten

$\rightsquigarrow$ **Komplexität kann sinnvoll nur als Funktion angegeben werden, die von der Größe der Eingabe abhängt.**

Komplexität und Eingabe (1)

Größe wird meistens abstrakt beschrieben:

- eine Zahl benötigt 1 Größeneinheit
- ein Buchstabe benötigt 1 GE
- Elemente von komplexen Strukturen (Knoten/Kanten/...) benötigen 1 GE

Für spezielle Algorithmen interessieren uns auch speziellere Größenmaße

- Z.B. nur Anzahl der Knoten in einem Graphen
- Z.B. lineare Größe einer quadratischen Matrix

Beispiel: Komplexität

Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen

Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```


Beispiel: Komplexität

Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen

Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

- Schleife z: $n \cdot 4$

Beispiel: Komplexität

Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen

Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

- Schleife z: $n \cdot 4$
- Schleife y:
  $n \cdot (3 + n \cdot 4) = 3 \cdot n + 4 \cdot n^2$

Beispiel: Komplexität

Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen

Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

- Schleife z: $n \cdot 4$
- Schleife y:
  $n \cdot (3 + n \cdot 4) = 3 \cdot n + 4 \cdot n^2$
- Schleife x:
  $n \cdot (1 + 3 \cdot n + 4 \cdot n^2) =$
  $n + 3 \cdot n^2 + 4 \cdot n^3$

Beispiel: Komplexität

Beispiel: Matrix-Multiplikation

Eingabe zwei $n \times n$-Matrizen

Ausgabe Matrizenprodukt ($n \times n$-Matrix)

```
def matrix_mult(a,b):
    for x in range(n):
        for y in range(n):
            sum=0
            for z in range(n):
                sum=sum+a[x,z]*b[z,y]
            c[x,y]=sum
    return c
```

- Schleife z: $n \cdot 4$
- Schleife y: $n \cdot (3 + n \cdot 4) = 3 \cdot n + 4 \cdot n^2$
- Schleife x: $n \cdot (1 + 3 \cdot n + 4 \cdot n^2) = n + 3 \cdot n^2 + 4 \cdot n^3$
- Funktion matrix_mult(): $4n^3 + 3n^2 + n + 1$

Übung: Komplexität

Ein Dozent verteilt n Klausuren an n Studenten. Er verwendet die folgenden Verfahren:

1. Er geht zum ersten Studenten, vergleicht dessen Namen mit denen auf jeder Klausur, und gibt dem Studenten seine Klausur, sobald er sie gefunden hat. Anschließend macht er beim nächsten Studenten weiter.
2. Der Dozent nimmt die erste Klausur, liest den Namen auf der Klausur und gibt die Klausur dem Studenten, der sich meldet.

Berechnen Sie, wie groß der Aufwand der Verteilung in Abhängigkeit von n bei jedem Verfahren ist. Machen Sie dabei die folgenden Annahmen:

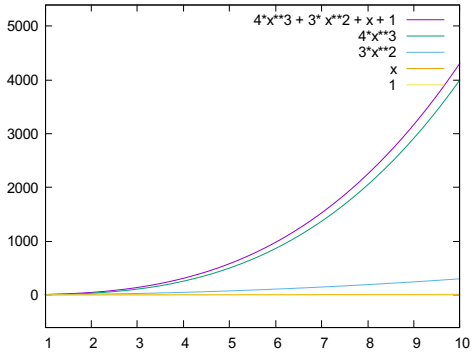
- Der Vergleich von zwei Namen dauert eine ZE.
- In einem Stapel der Größe n ist die gesuchte Klausur an Position $\lceil n/2 \rceil$.
- Das Übergeben der Klausur an den entsprechenden Studenten (Variante 2) hat konstanten Aufwand von einer ZE.

Komplexität für große Eingaben

Der Term $4n^3 + 3n^2 + n + 1$ ist unhandlich.
Wie verhält sich der Term für große Werte von n?

Komplexität für große Eingaben

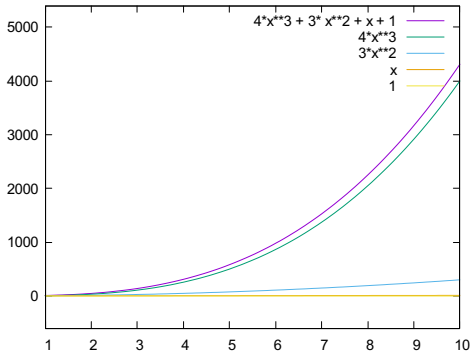
Der Term $4n^3 + 3n^2 + n + 1$ ist unhandlich.
Wie verhält sich der Term für große Werte von n?



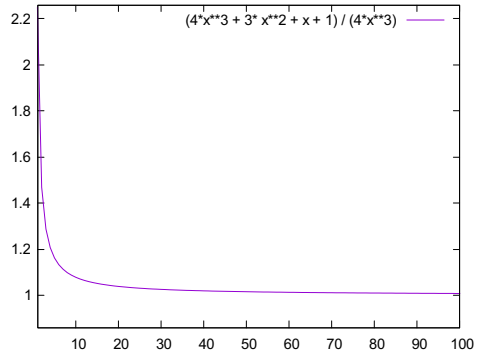
- für $n > 5$ ist der Unterschied zwischen $4n^3 + 3n^2 + n + 1$ und $4 \cdot n^3$ irrelevant

Komplexität für große Eingaben

Der Term $4n^3 + 3n^2 + n + 1$ ist unhandlich.
Wie verhält sich der Term für große Werte von n?

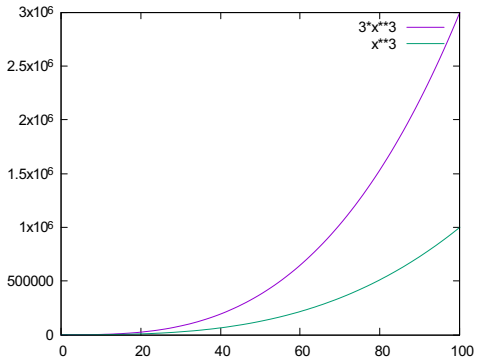


- für $n > 5$ ist der Unterschied zwischen $4n^3 + 3n^2 + n + 1$ und $4 \cdot n^3$ irrelevant



- der Abstand wird zwar größer, ...
- ... aber das Verhältnis konvergiert gegen 1
- bei Polynomen ist nur der größte Exponent interessant $\rightsquigarrow 4n^3$

Weitere Vereinfachungen



$3n^3$ und n^3 verhalten sich „ähnlich“:

- Verdoppelung von $n \rightsquigarrow$ Verachtfachung der Funktion
- $3n^3 / n^3$ konvergiert gegen einen konstanten Wert (nämlich 3)

- Außerdem: konstante Faktoren oft abhängig von Implementierungs-/Sprach-/Compiler-Details
  - `a = 2 * 3 + 4;` $\rightsquigarrow$ 3 Schritte
  - `a = 2 * 3; a = a + 4;` $\rightsquigarrow$ 4 Schritte
- $\rightsquigarrow$ Vernachlässigung von konstanten Faktoren
- Umstritten!
  - Sedgewick entwickelt eigene Notation, die konstante Faktoren berücksichtigt
  - Verbesserung eines Algorithmus um Faktor 10 ist spürbar

$\mathcal{O}$-Notation

$\mathcal{O}$-Notation

Für eine Funktion f bezeichnet $\mathcal{O}(f)$ die Menge aller Funktionen g mit

$$\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$

$\mathcal{O}$-Notation

$\mathcal{O}$-Notation

Für eine Funktion f bezeichnet $\mathcal{O}(f)$ die Menge aller Funktionen g mit

$$\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$

- Ab einer bestimmten Zahl n ist $g(n)$ kleiner-gleich $c \cdot f(n)$ für einen konstanten Faktor c.
- $\mathcal{O}(f)$ ist die Menge aller Funktionen, die nicht schneller wachsen als f
- Statt $g \in \mathcal{O}(f)$ sagt man oft „g ist $\mathcal{O}(f)$“.
  „Der Aufwand des Matrix-Multiplikations-Algorithmus ist $\mathcal{O}(n^3)$.“

Ende Vorlesung 3

Beispiele: $\mathcal{O}$-Notation

- n^2 ist $\mathcal{O}(n^3)$
- $3 \cdot n^3$ ist $\mathcal{O}(n^3)$
- $4n^3 + 3n^2 + n + 1$ ist $\mathcal{O}(n^3)$
- $n \cdot \sqrt{n}$ ist $\mathcal{O}(n^2)$

Beispiele: $\mathcal{O}$-Notation

- n^2 ist $\mathcal{O}(n^3)$
- $3 \cdot n^3$ ist $\mathcal{O}(n^3)$
- $4n^3 + 3n^2 + n + 1$ ist $\mathcal{O}(n^3)$
- $n \cdot \sqrt{n}$ ist $\mathcal{O}(n^2)$

Vorsicht: $\mathcal{O}$-Notation

In der Literatur wird $g \in \mathcal{O}(f)$ oft geschrieben als $g = \mathcal{O}(f)$.

Beispiele: $\mathcal{O}$-Notation

- n^2 ist $\mathcal{O}(n^3)$
- $3 \cdot n^3$ ist $\mathcal{O}(n^3)$
- $4n^3 + 3n^2 + n + 1$ ist $\mathcal{O}(n^3)$
- $n \cdot \sqrt{n}$ ist $\mathcal{O}(n^2)$

Vorsicht: $\mathcal{O}$-Notation

In der Literatur wird $g \in \mathcal{O}(f)$ oft geschrieben als $g = \mathcal{O}(f)$.
Dieses $=$ ist nicht symmetrisch: $n = \mathcal{O}(n^2)$, aber $n^2 \neq \mathcal{O}(n)$.

Beispiele: $\mathcal{O}$-Notation

- n^2 ist $\mathcal{O}(n^3)$
- $3 \cdot n^3$ ist $\mathcal{O}(n^3)$
- $4n^3 + 3n^2 + n + 1$ ist $\mathcal{O}(n^3)$
- $n \cdot \sqrt{n}$ ist $\mathcal{O}(n^2)$

Vorsicht: $\mathcal{O}$-Notation

In der Literatur wird $g \in \mathcal{O}(f)$ oft geschrieben als $g = \mathcal{O}(f)$.
Dieses $=$ ist nicht symmetrisch: $n = \mathcal{O}(n^2)$, aber $n^2 \neq \mathcal{O}(n)$.
Besser: $g \in \mathcal{O}(f)$ oder g ist $\mathcal{O}(f)$.

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f $\quad f \in \mathcal{O}(f)$

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f $\quad f \in \mathcal{O}(f)$

$g \in \mathcal{O}(f) \quad \Rightarrow \quad c \cdot g \in \mathcal{O}(f)$ Konstanter Faktor

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f		f	$\in \mathcal{O}(f)$	
$g \in \mathcal{O}(f)$	$\Rightarrow$	$c \cdot g$	$\in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$	$\Rightarrow$	$g + h$	$\in \mathcal{O}(f)$	Summe

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f		f	$\in \mathcal{O}(f)$	
$g \in \mathcal{O}(f)$	$\Rightarrow$	$c \cdot g$	$\in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$	$\Rightarrow$	$g + h$	$\in \mathcal{O}(f)$	Summe
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(g)$	$\Rightarrow$	h	$\in \mathcal{O}(f)$	

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f		f	$\in \mathcal{O}(f)$	
$g \in \mathcal{O}(f)$	$\Rightarrow$	$c \cdot g$	$\in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$	$\Rightarrow$	$g + h$	$\in \mathcal{O}(f)$	Summe
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(g)$	$\Rightarrow$	h	$\in \mathcal{O}(f)$	Transitivität

Rechenregeln für $\mathcal{O}$-Notation

Für jede Funktion f		f	$\in \mathcal{O}(f)$	
$g \in \mathcal{O}(f)$	$\Rightarrow$	$c \cdot g$	$\in \mathcal{O}(f)$	Konstanter Faktor
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(f)$	$\Rightarrow$	$g + h$	$\in \mathcal{O}(f)$	Summe
$g \in \mathcal{O}(f) \wedge h \in \mathcal{O}(g)$	$\Rightarrow$	h	$\in \mathcal{O}(f)$	Transitivität
$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R}$	$\Rightarrow$	g	$\in \mathcal{O}(f)$	Grenzwert

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} =$

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5}$

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5} \Rightarrow f \in \mathcal{O}(g)$

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5} \Rightarrow f \in \mathcal{O}(g)$
  - $\lim_{n\to\infty} \frac{g(n)}{f(n)} =$

44

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5} \Rightarrow f \in \mathcal{O}(g)$
  - $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{5}{3}$

Grenzwertbetrachtung

- Grenzwertregel:

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \in \mathbb{R} \Rightarrow g \in \mathcal{O}(f)$$

- Anschaulich:
  - Wenn der Grenzwert existiert, dann steigt g langfristig höchstens um einen konstanten Faktor schneller als f
  - Spezialfall: Wenn der Grenzwert 0 ist, dann steigt f um mehr als einen konstanten Faktor schneller als g
- Beispiel:
  - $f(n) = 3n + 2, g(n) = 5n$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3}{5} \Rightarrow f \in \mathcal{O}(g)$
  - $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{5}{3} \Rightarrow g \in \mathcal{O}(f)$

44

Ein nützliches Resultat der Analysis

Regel von l'Hôpital

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Ein nützliches Resultat der Analysis

Regel von l'Hôpital

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Beispiel

$$\lim_{x\to\infty} \frac{10 \cdot x}{x^2}$$

Ein nützliches Resultat der Analysis

Regel von l'Hôpital

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Beispiel

$$\lim_{x\to\infty} \frac{10 \cdot x}{x^2} = \lim_{x\to\infty} \frac{10}{2 \cdot x}$$

Ein nützliches Resultat der Analysis

Regel von l'Hôpital

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

Beispiel

$$\lim_{x\to\infty} \frac{10 \cdot x}{x^2} = \lim_{x\to\infty} \frac{10}{2 \cdot x} = \lim_{x\to\infty} \frac{0}{2} = 0$$

Übung: $\mathcal{O}$-Bestimmung

- Finden Sie das kleinste $k \in \mathbb{N}$ mit $n \cdot \log n \in \mathcal{O}(n^k)$
- Finden Sie das kleinste $k \in \mathbb{N}$ mit $n \cdot (\log n)^2 \in \mathcal{O}(n^k)$
- Finden Sie das kleinste $k \in \mathbb{N}$ mit $2^n \in \mathcal{O}(n^k)$
- Ordnen Sie die folgenden Funktionen nach Komplexität

$$n^2 \quad \sqrt{n} \quad n \cdot 2^n \quad \log(\log(n)) \quad 2^n \quad n^{10} \quad 1,1^n \quad n^n \quad \log n$$

Anmerkung:

- $\log_b n = \frac{\ln n}{\ln b} = \frac{1}{\ln b} \cdot \ln n = c \cdot \ln n$
- $\rightsquigarrow$ die Basis der Logarithmus-Funktion bewirkt nur eine Änderung um einen konstanten Faktor.
- $\rightsquigarrow$ die Basis ist für die $\mathcal{O}$-Betrachtung vernachlässigbar

Ende Vorlesung 4

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} =$$

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} = \lim_{n\to\infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}}$$

47

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} = \lim_{n\to\infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^{(n-1)}}{n^{(n-1)}}$$

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} = \lim_{n\to\infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^{(n-1)}}{n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^n}{n^n}$$

47

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} = \lim_{n\to\infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^{(n-1)}}{n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^n}{n^n} = 0$$

Ergänzung

Frage: Sei $f(n) = n^n$, $g(n) = n \cdot 2^n$. Gilt $f \in O(g)$ oder $g \in O(f)$?

- Antwort ist nicht offensichtlich!
- Idee: Betrachte die Ableitungen:
  - $f'(n) = n^n((\ln n) + 1) = (\ln n)n^n + n^n = (\ln n)f(n) + f(n)$
  - $g'(n) = ((\ln 2)n + 1)2^n = (\ln 2)n2^n + 2^n = (\ln 2)g(n) + \frac{g(n)}{n}$
- Also:
  - g' steigt nicht wesentlich schneller als g: $g' \in O(g)$
  - Aber: f' steigt wesentlich schneller als f: $f' \notin O(f)$
- Alternativ:

$$\lim_{n\to\infty} \frac{n \cdot 2^n}{n^n} = \lim_{n\to\infty} \frac{n \cdot 2 \cdot 2^{(n-1)}}{n \cdot n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^{(n-1)}}{n^{(n-1)}} = \lim_{n\to\infty} \frac{2 \cdot 2^n}{n^n} = 0$$

Also: $g \in O(f)$ und $f \notin O(g)$. n^n wächst echt schneller als $n \cdot 2^n$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

$$n! \quad \in \quad \mathcal{O}(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n)$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

$$\begin{aligned} n! &\in \mathcal{O}(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n) \\ &\in \mathcal{O}(c \cdot \tfrac{\sqrt{n}}{e^n} \cdot n^n) \end{aligned}$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

$$\begin{aligned} n! &\in \mathcal{O}(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n) \\ &\in \mathcal{O}(c \cdot \tfrac{\sqrt{n}}{e^n} \cdot n^n) \\ &\in \mathcal{O}(n^n) \end{aligned}$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

$$\begin{aligned} n! &\in \mathcal{O}(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n) \\ &\in \mathcal{O}(c \cdot \tfrac{\sqrt{n}}{e^n} \cdot n^n) \\ &\in \mathcal{O}(n^n) \\ &\in \mathcal{O}(e^{n \log n}) \end{aligned}$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

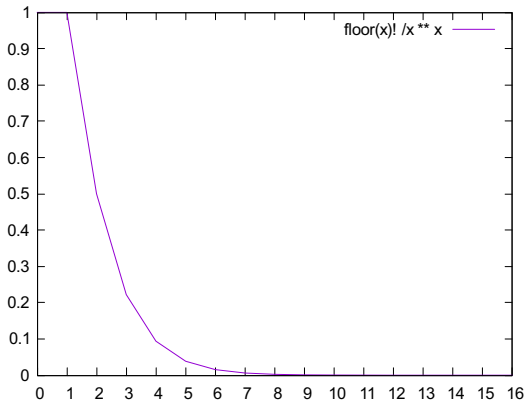
$$\begin{aligned} n! &\in \mathcal{O}(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n) \\ &\in \mathcal{O}(c \cdot \tfrac{\sqrt{n}}{e^n} \cdot n^n) \\ &\in \mathcal{O}(n^n) \\ &\in \mathcal{O}(e^{n \log n}) \\ &\notin \mathcal{O}(e^{c \cdot n}) \text{ für irgendein } c \end{aligned}$$

Noch ein nützliches Resultat der Analysis

Stirling'sche Formel

$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

$$\begin{aligned} n! &\in \mathcal{O}(\sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n) \\ &\in \mathcal{O}(c \cdot \tfrac{\sqrt{n}}{e^n} \cdot n^n) \\ &\in \mathcal{O}(n^n) \\ &\in \mathcal{O}(e^{n \log n}) \\ &\notin \mathcal{O}(e^{c \cdot n}) \text{ für irgendein } c \end{aligned}$$



Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung

Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung
$\mathcal{O}(n!)$	„faktoriell“	alle Permutationen	Travelling Salesman Brute-Force-Sortieren

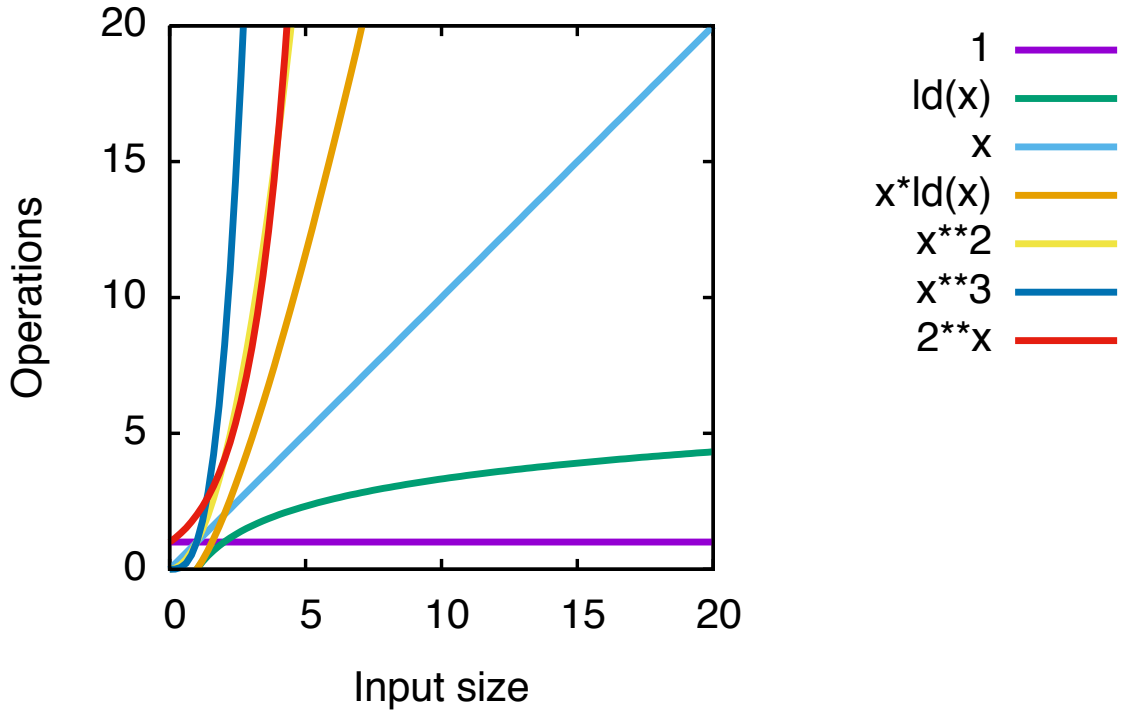
Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung
$\mathcal{O}(n!)$	„faktoriell“	alle Permutationen	Travelling Salesman Brute-Force-Sortieren
$\mathcal{O}(n^n)$		alle Folgen der Länge n	

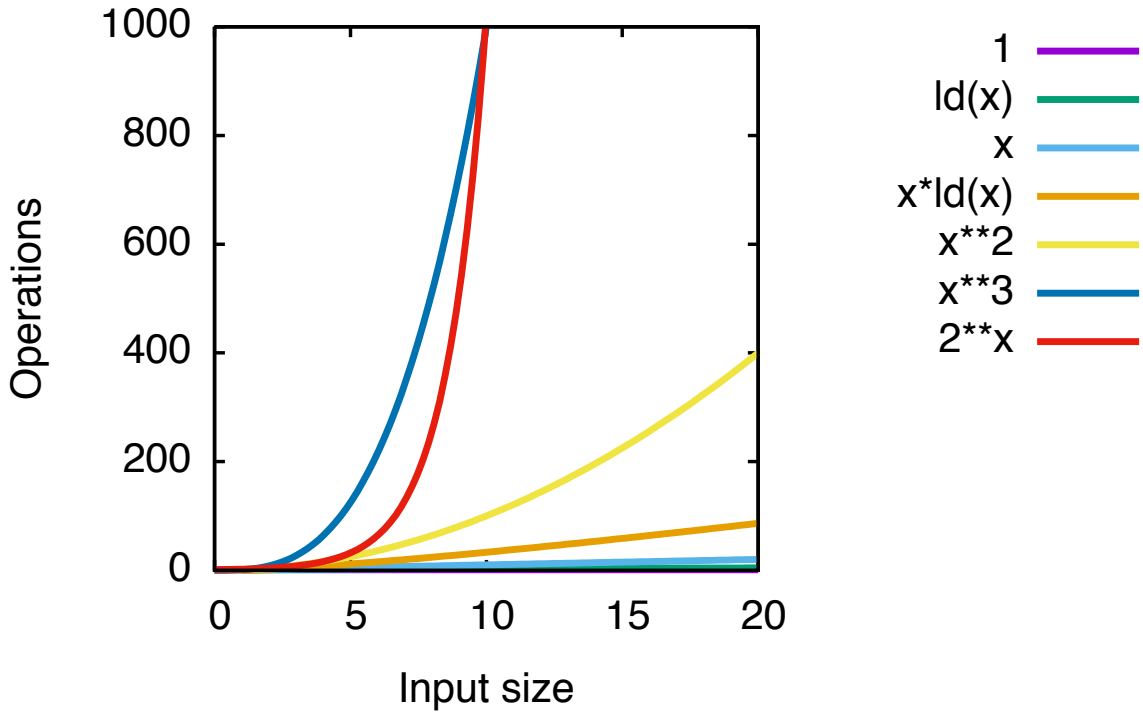
Komplexitätsklassen verschiedener Funktionen

Ordnung	Bezeichnung	Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch“ „super-linear“	divide and conquer	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung
$\mathcal{O}(n!)$	„faktoriell“	alle Permutationen	Travelling Salesman Brute-Force-Sortieren
$\mathcal{O}(n^n)$		alle Folgen der Länge n	
$\mathcal{O}(2^{2^n})$	doppelt exponentiell	binärer Baum mit exponentieller Tiefe	

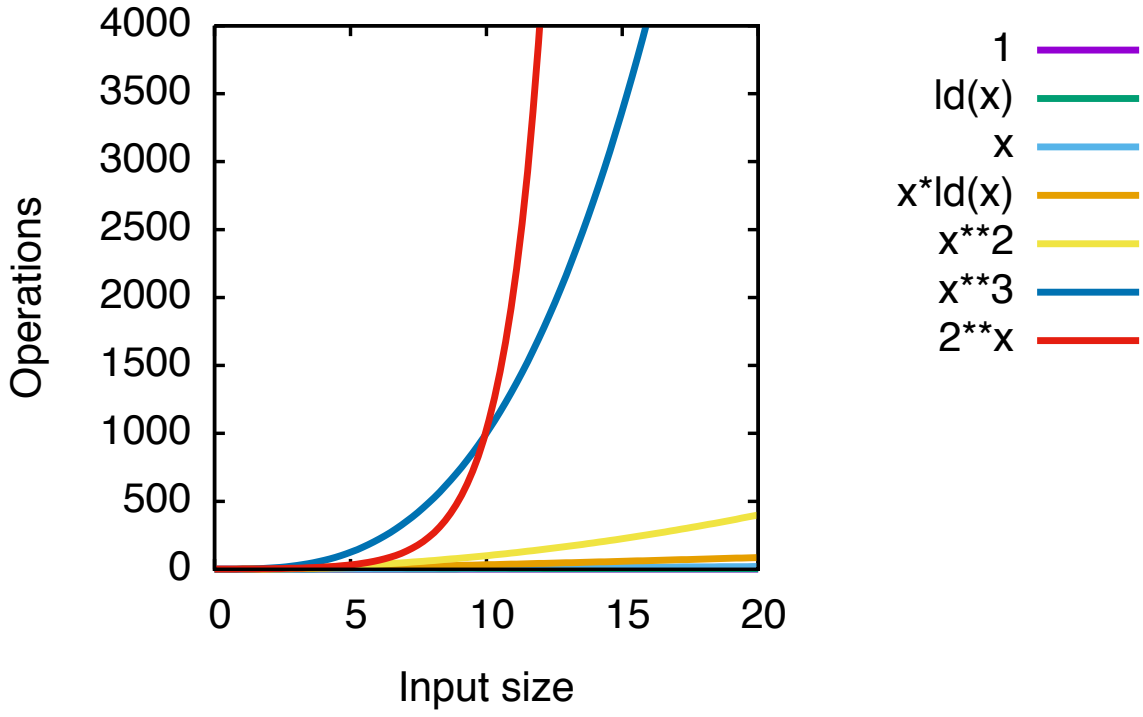
Komplexität anschaulich



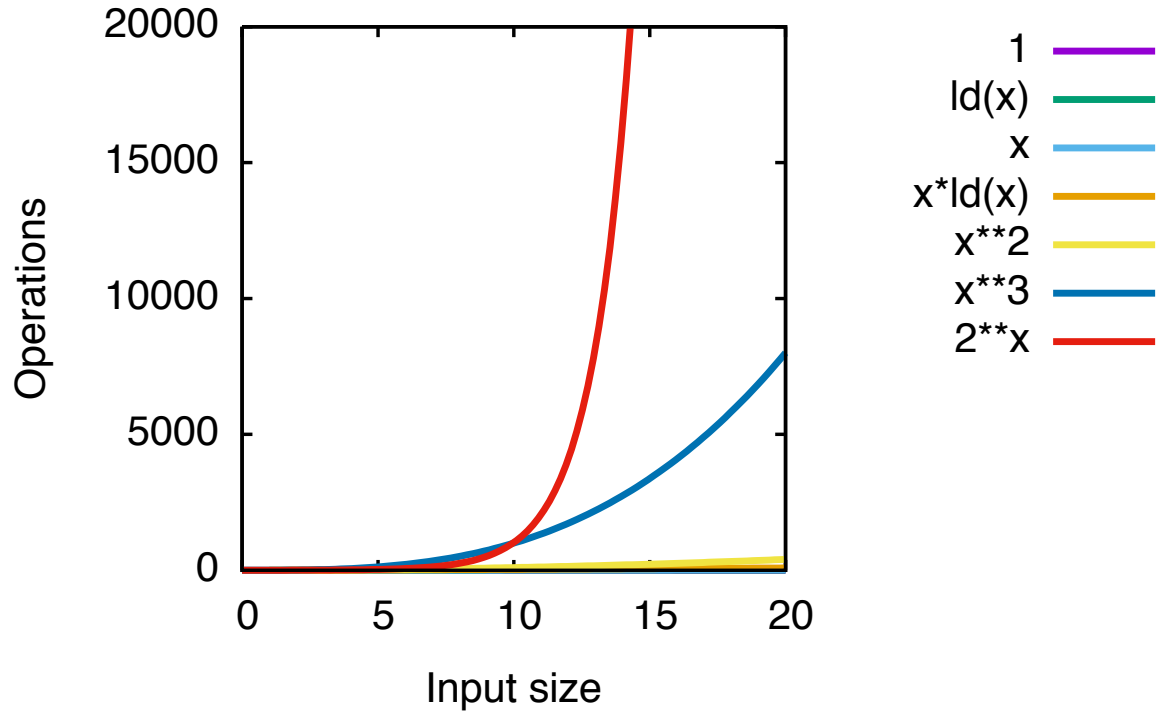
Komplexität anschaulich



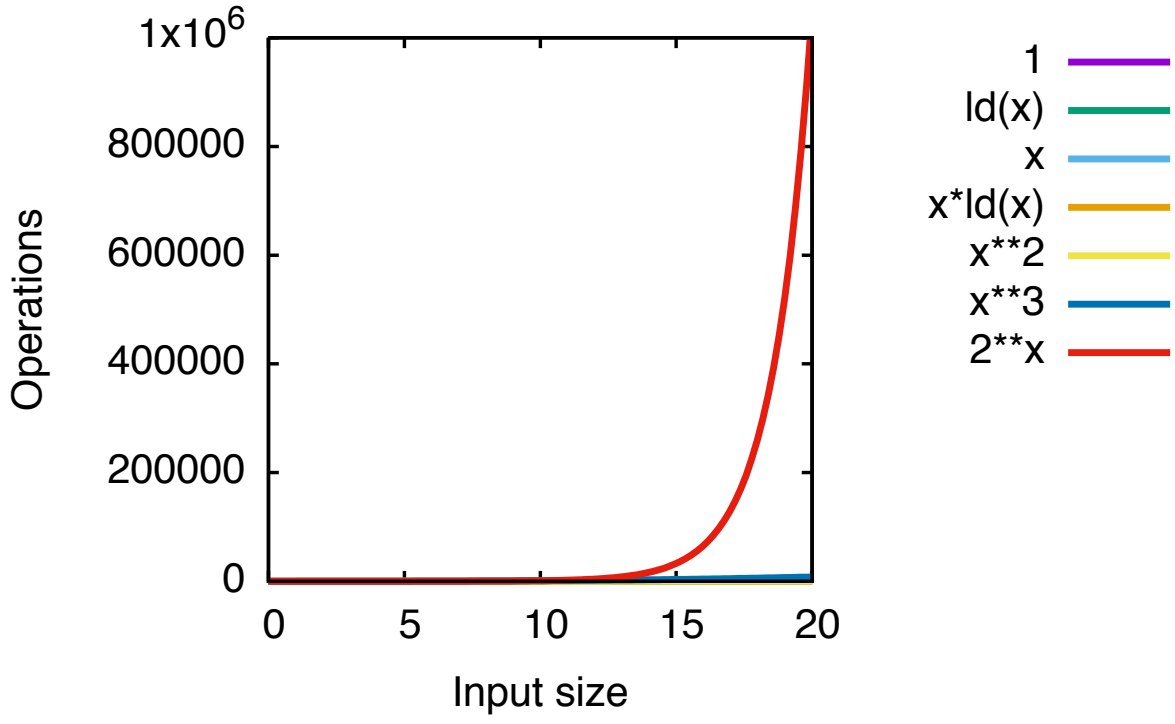
Komplexität anschaulich



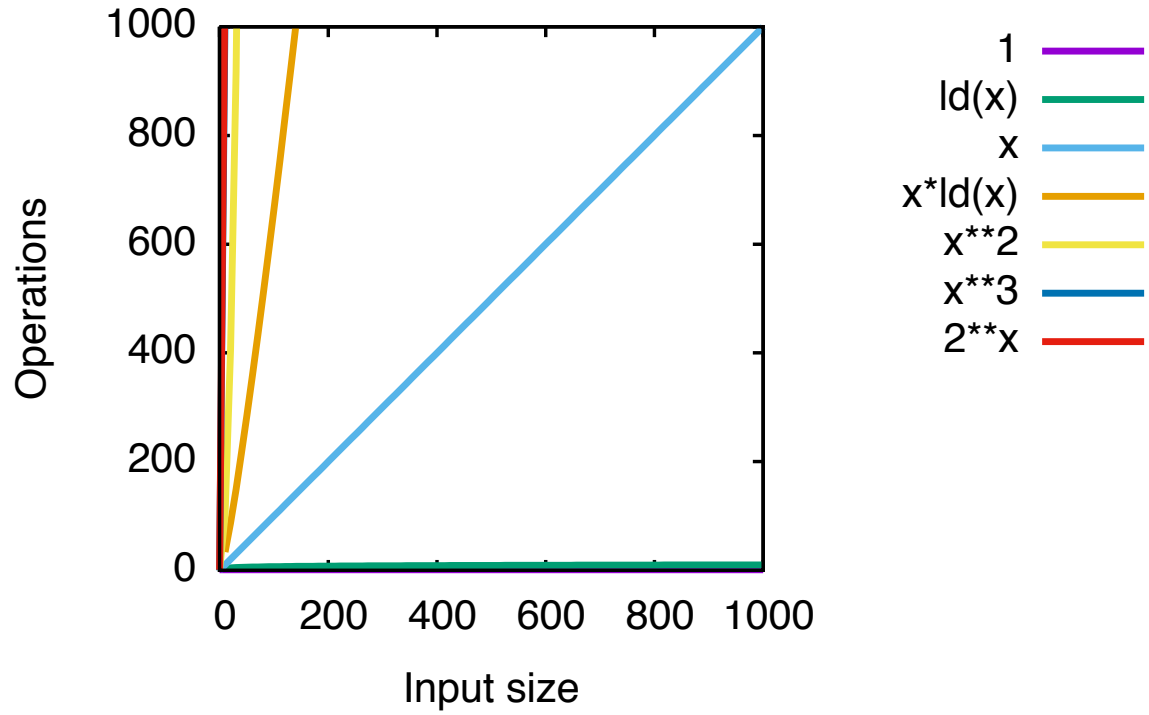
Komplexität anschaulich



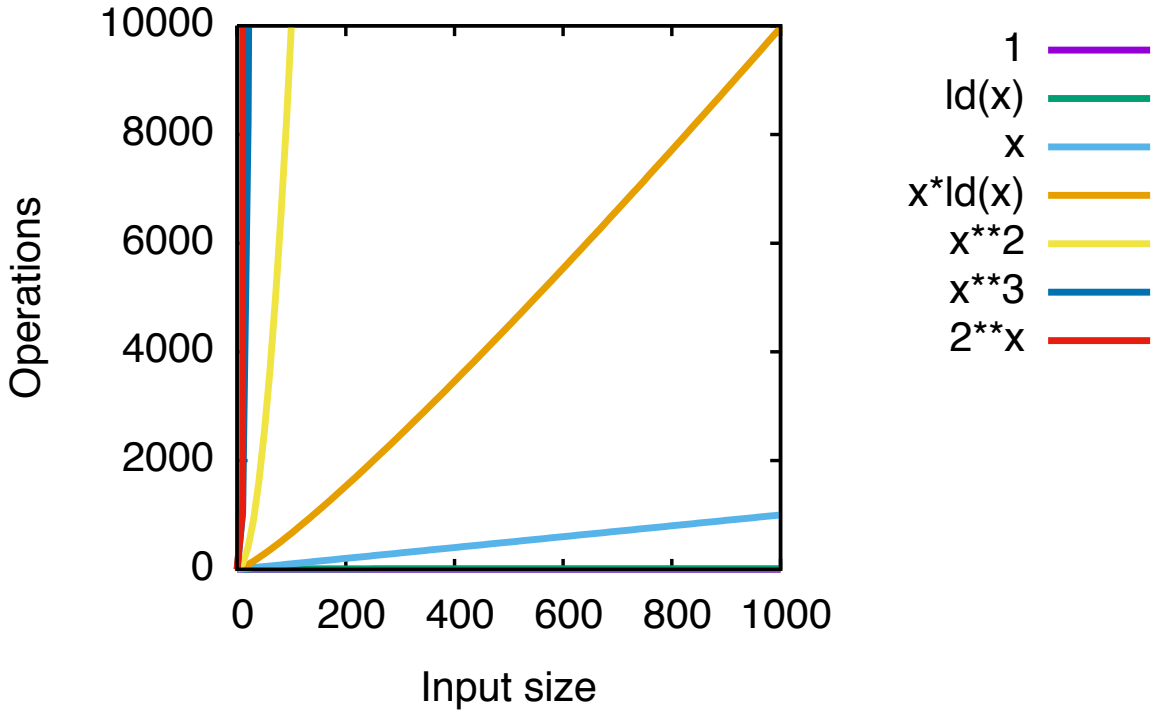
Komplexität anschaulich



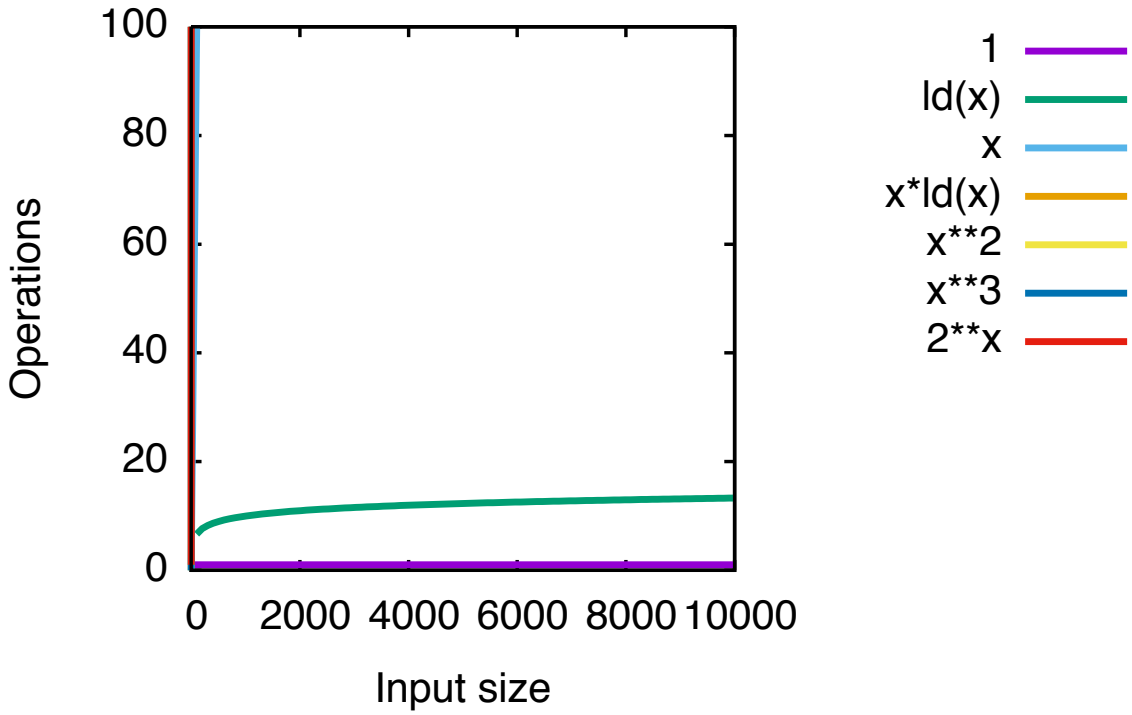
Komplexität anschaulich



Komplexität anschaulich



Komplexität anschaulich



Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
ld(n)	Overflow	Overflow	Overflow	Overflow

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
ld(n)	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\mathrm{ld}(n)$	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB
$n \cdot \mathrm{ld}(n)$	350 MB	17 GB	900 GB	1.95 TB

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\mathrm{ld}(n)$	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB
$n \cdot \mathrm{ld}(n)$	350 MB	17 GB	900 GB	1.95 TB
n^2	100 KB	700 KB	6 MB	29 MB

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\mathrm{ld}(n)$	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB
$n \cdot \mathrm{ld}(n)$	350 MB	17 GB	900 GB	1.95 TB
n^2	100 KB	700 KB	6 MB	29 MB
n^3	2.1 KB	8.4 KB	33 KB	95 KB

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\mathrm{ld}(n)$	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB
$n \cdot \mathrm{ld}(n)$	350 MB	17 GB	900 GB	1.95 TB
n^2	100 KB	700 KB	6 MB	29 MB
n^3	2.1 KB	8.4 KB	33 KB	95 KB
2^n	33 B	39 B	45 B	50 B

Was geht?

- Moderne Prozessoren schaffen ca. 10 Milliarden Instruktionen pro Sekunde
- Welche Eingabegröße kann vom Prozessor verarbeitet werden?
  - Annahme: 1 Byte = 1 GE, 1 Instruktion = 1 ZE

Zeitlimit	1s	1m	1h	1d
Komplexität				
$\mathrm{ld}(n)$	Overflow	Overflow	Overflow	Overflow
n	10 GB	600 GB	36 TB	864 TB
$n \cdot \mathrm{ld}(n)$	350 MB	17 GB	900 GB	1.95 TB
n^2	100 KB	700 KB	6 MB	29 MB
n^3	2.1 KB	8.4 KB	33 KB	95 KB
2^n	33 B	39 B	45 B	50 B
2^{2^n}	5 B	5,3 B	5,5 B	5,6 B

Mr. Universe vs. Log-Man

Mr. Universe vs. Log-Man

Geschätzte 10^{87} Elementarteilchen
im beobachtbaren Universum

Mr. Universe vs. Log-Man

Geschätzte 10^{87} Elementarteilchen
im beobachtbaren Universum

$$100000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$

Mr. Universe vs. Log-Man

Geschätzte 10^{87} Elementarteilchen im beobachtbaren Universum

$$100000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$

$$\log_2(10^{87}) \approx 289$$

Mr. Universe vs. Log-Man

Geschätzte 10^{87} Elementarteilchen im beobachtbaren Universum

$$100000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$
$$000000000000000000000$$

$$\log_2(10^{87}) \approx 289$$

Ein logarithmischer Algorithmus könnte das Universum $\approx$ 35 Millionen Mal pro Sekunde durchsuchen!

Übung: Algorithmentypen und Komplexität

Fibonacci-Zahlen

- $f(1) = 1$
- $f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$

Erste 10 Zahlen: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

1. Schreiben Sie zwei Funktionen (Pseudocode oder Sprache ihrer Wahl), von denen eine $f(n)$ rekursiv , die andere die selbe Funktion iterativ berechnet.

Übung: Algorithmentypen und Komplexität

Fibonacci-Zahlen

- $f(1) = 1$
- $f(2) = 1$
- $f(n) = f(n-1) + f(n-2)$

Erste 10 Zahlen: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

1. Schreiben Sie zwei Funktionen (Pseudocode oder Sprache ihrer Wahl), von denen eine $f(n)$ rekursiv , die andere die selbe Funktion iterativ berechnet.
2. Bestimmen Sie dann für jede Funktion die Komplexität ($\mathcal{O}$-Notation).

Ende Vorlesung 5

Optimierung des rekursiven Fib-Algorithmus

- Die naive rekursive Funktion ist ineffizient
  - Beispiel: Berechnung von `fib(10)`
  - 1 x `fib(9)`, 2 x `fib(8)`, 3 x `fib(7)`, 5 x `fib(6)`, 8 x `fib(5)`, ...
- Rekursives Programmieren ist eleganter als iteratives
- Oft ist es nicht (leicht) möglich, eine Funktion iterativ zu berechnen (z.B. Ackermann-Funktion)
- Wie kann man den rekursiven Algorithmus auf $\mathcal{O}(n)$ bringen?

Optimierung des rekursiven Fib-Algorithmus

- Die naive rekursive Funktion ist ineffizient
  - Beispiel: Berechnung von `fib(10)`
  - 1 x `fib(9)`, 2 x `fib(8)`, 3 x `fib(7)`, 5 x `fib(6)`, 8 x `fib(5)`, ...
- Rekursives Programmieren ist eleganter als iteratives
- Oft ist es nicht (leicht) möglich, eine Funktion iterativ zu berechnen (z.B. Ackermann-Funktion)
- Wie kann man den rekursiven Algorithmus auf $\mathcal{O}(n)$ bringen?

Ansatz: Zwischenspeichern von Teilergebnissen

- speichere jedes berechnete Ergebnis $f(n)$
- berechne nur Ergebnisse, die noch nicht gespeichert sind
- f ist $\mathcal{O}(n)$

Optimierung des rekursiven Fib-Algorithmus

- Die naive rekursive Funktion ist ineffizient
  - Beispiel: Berechnung von `fib(10)`
  - 1 x `fib(9)`, 2 x `fib(8)`, 3 x `fib(7)`, 5 x `fib(6)`, 8 x `fib(5)`, ...
- Rekursives Programmieren ist eleganter als iteratives
- Oft ist es nicht (leicht) möglich, eine Funktion iterativ zu berechnen (z.B. Ackermann-Funktion)
- Wie kann man den rekursiven Algorithmus auf $\mathcal{O}(n)$ bringen?

Ansatz: Zwischenspeichern von Teilergebnissen

- speichere jedes berechnete Ergebnis $f(n)$
- berechne nur Ergebnisse, die noch nicht gespeichert sind
- f ist $\mathcal{O}(n)$

```
def fibd(n) :
  if n==1:
    return 1
  elif n==2:
    return 1
  elif fib[n]==0:
    fib[n]=fibd(n-1)+fibd(n-2)
  return fib[n]
```


Übung: Fibonacci dynamisch

```
def fibd(n) :
  if n==1:
    return 1
  elif n==2:
    return 1
  elif fib[n]==0:
    fib[n]=fibd(n-1)+fibd(n-2)
  return fib[n]
```

- Führen Sie den Algorithmus für $n = 7$ aus
  - Sie können davon ausgehen, dass das Array `fib` geignet groß und in allen Elementen mit `0` initialisiert ist

Übung: Fibonacci dynamisch

```
def fibd(n) :
  if n==1:
    return 1
  elif n==2:
    return 1
  elif fib[n]==0:
    fib[n]=fibd(n-1)+fibd(n-2)
  return fib[n]
```

- Führen Sie den Algorithmus für $n = 7$ aus
  - Sie können davon ausgehen, dass das Array `fib` geignet groß und in allen Elementen mit `0` initialisiert ist
- Bonus-Frage: Wie lange würde das initialisieren des Arrays `fib` dauern?

Dynamisches Programmieren: Technik

Die skizzierte Optimierung für die Fibonacci-Funktion wird dynamisches Programmieren genannt.

Dynamisches Programmieren: Technik

Die skizzierte Optimierung für die Fibonacci-Funktion wird dynamisches Programmieren genannt.

- Führe komplexes Problem auf einfache Teilprobleme zurück
- berechne Lösungen der Teilprobleme
  - speichere Teillösungen
- rekonstruiere Gesamtlösung aus Teillösungen

Dynamisches Programmieren: Anwendungskriterien

Überlappende Teilprobleme	Dasselbe Problem taucht mehrfach auf (Fibonacci)
Optimale Substruktur	Globale Lösung setzt sich aus lokalen Lösungen zusammen

Dynamisches Programmieren: Anwendungskriterien

Überlappende Teilprobleme	Dasselbe Problem taucht mehrfach auf (Fibonacci)
Optimale Substruktur	Globale Lösung setzt sich aus lokalen Lösungen zusammen

Beispiel: Routing

- Der optimale Weg von Stuttgart nach Frankfurt ist 200km lang.
- Wenn der optimale Weg von Konstanz nach Frankfurt über Stuttgart führt,
  dann benötigt er 200km plus die Länge des optimalen Wegs von Konstanz nach Stuttgart

Übung: Dynamisches Programmieren

Für eine natürliche Zahl n können 3 verschiedene Operationen durchgeführt werden:

1. subtrahiere 1 von n
2. teile n durch 2, wenn n durch 2 teilbar ist
3. teile n durch 3, wenn n durch 3 teilbar ist

Finden Sie für ein gegebenes n die minimale Anzahl von Schritten, die nötig ist, um n zu 1 zu überführen. Vergleichen Sie die Komplexität des Brute-Force-Ansatzes mit dem des Dynamischen Programmierens.

Übung: Dynamisches Programmieren

Für eine natürliche Zahl n können 3 verschiedene Operationen durchgeführt werden:

1. subtrahiere 1 von n
2. teile n durch 2, wenn n durch 2 teilbar ist
3. teile n durch 3, wenn n durch 3 teilbar ist

Finden Sie für ein gegebenes n die minimale Anzahl von Schritten, die nötig ist, um n zu 1 zu überführen. Vergleichen Sie die Komplexität des Brute-Force-Ansatzes mit dem des Dynamischen Programmierens.

Ein Greedy-Algorithmus entscheidet sich immer für denjenigen Schritt, der ihn dem Ziel am nächsten bringt (in diesem Fall: die Zahl am meisten reduziert). Führt der Greedy-Algorithmus in diesem Fall zur besten Lösung?

Grenzen des Dynamischen Programmierens

DP ist nicht auf jedes Problem anwendbar!

Grenzen des Dynamischen Programmierens

DP ist nicht auf jedes Problem anwendbar!

Travelling Salesman Problem

Aufgabe: Rundreise durch Städte 1–5, minimiere die Gesamtstrecke.
Wenn zum Besuchen der Städte 1–4 die beste Reihenfolge
$1 \to 2 \to 3 \to 4 \to 1$ ist, kann die beste Reihenfolge für 1–5 auch
$1 \to 3 \to 2 \to 4 \to 5 \to 1$ sein.

Grenzen des Dynamischen Programmierens

DP ist nicht auf jedes Problem anwendbar!

Travelling Salesman Problem

Aufgabe: Rundreise durch Städte 1–5, minimiere die Gesamtstrecke.
Wenn zum Besuchen der Städte 1–4 die beste Reihenfolge
$1 \to 2 \to 3 \to 4 \to 1$ ist, kann die beste Reihenfolge für 1–5 auch
$1 \to 3 \to 2 \to 4 \to 5 \to 1$ sein.

Sortieren

Aufgabe: Sortiere die Liste $(5, 3, 1, 2, 4)$
Die Sortierung der Liste $(5, 3, 1)$ ist $(1, 3, 5)$.
Diese ist keine Teilliste der sortierten Gesamtliste $(1, 2, 3, 4, 5)$.

Grenzen des Dynamischen Programmierens

DP ist nicht auf jedes Problem anwendbar!

Travelling Salesman Problem

Aufgabe: Rundreise durch Städte 1–5, minimiere die Gesamtstrecke.
Wenn zum Besuchen der Städte 1–4 die beste Reihenfolge
$1 \to 2 \to 3 \to 4 \to 1$ ist, kann die beste Reihenfolge für 1–5 auch
$1 \to 3 \to 2 \to 4 \to 5 \to 1$ sein.

Sortieren

Aufgabe: Sortiere die Liste $(5, 3, 1, 2, 4)$
Die Sortierung der Liste $(5, 3, 1)$ ist $(1, 3, 5)$.
Diese ist keine Teilliste der sortierten Gesamtliste $(1, 2, 3, 4, 5)$.

Grund: Globale Lösung setzt sich nicht direkt aus lokalen Lösungen zusammen.

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

60

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

$$a(1, m) = m + 2$$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

60

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

$$\begin{aligned} a(1, m) &= m + 2 \\ a(2, m) &= 2m + 3 \end{aligned}$$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

$$\begin{aligned} a(1, m) &= m + 2 \\ a(2, m) &= 2m + 3 \\ a(3, m) &= 8 \cdot 2^m - 3 \end{aligned}$$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

60

Exkurs: Ackermann-Funktion

Warum überhaupt rekursiv programmieren?

$a(n, m)$

- $a(0, m) = m + 1$
- $a(n + 1, 0) = a(n, 1)$
- $a(n + 1, m + 1) = a(n, a(n + 1, m))$

$$
\begin{aligned}
a(1, m) &= m + 2 \\
a(2, m) &= 2m + 3 \\
a(3, m) &= 8 \cdot 2^m - 3 \\
a(4, m) &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{m+3 \text{ mal}} - 3
\end{aligned}
$$

- rekursiv sehr einfach zu implementieren
- iterativ?
  - welche Werte für n und m werden zur Berechnung $a(4, 2)$ benötigt?
  - selbst wenn die richtigen Werte gefunden sind, ist das Programm schwer verständlich

Ende Vorlesung 6

Komplexität rekursiver Programme

Berechnung von m^n

Iteratives Programm:

```
def pi(m,n):
    p = 1
    for x in range(n):
        p = p * m
    return p
```

Komplexität rekursiver Programme

Berechnung von m^n

Iteratives Programm:

```
def pi(m,n):
    p = 1
    for x in range(n):
        p = p * m
    return p
```

- Zähle verschachtelte `for`-Schleifen
- Bestimme maximalen Wert der Zählvariable
- $\mathcal{O}(n)$

Komplexität rekursiver Programme

Berechnung von m^n

Iteratives Programm:

```
def pi(m,n):
    p = 1
    for x in range(n):
        p = p * m
    return p
```

- Zähle verschachtelte `for`-Schleifen
- Bestimme maximalen Wert der Zählvariable
- $\mathcal{O}(n)$

Rekursives Programm:

```
def pr(m,n):
    if n==0:
        return 1
    else:
        return m * pr(m,n-1)
```

Komplexität rekursiver Programme

Berechnung von m^n

Iteratives Programm:

```
def pi(m,n):
    p = 1
    for x in range(n):
        p = p * m
    return p
```

- Zähle verschachtelte `for`-Schleifen
- Bestimme maximalen Wert der Zählvariable
- $\mathcal{O}(n)$

Rekursives Programm:

```
def pr(m,n):
    if n==0:
        return 1
    else:
        return m * pr(m,n-1)
```

- Wie Komplexität bestimmen?

Rekurrenzrelationen

```
def pr(m,n):
    if n==0:
        return 1
    else:
        return m * pr(m,n-1)
```

Rekurrenzrelationen

```
def pr(m,n):
  if n==0:
    return 1
  else:
    return m * pr(m,n-1)
```

Definiere $r(n)$ als Anzahl der ZE, die zur Berechnung von `pr(m,n)` benötigt werden.

- $r(0) = 2$
- $r(n) = 3 + r(n-1)$ für $n > 0$
- $r(n) \approx 3 \cdot n$

Rekurrenzrelationen

```
def pr(m,n):
  if n==0:
    return 1
  else:
    return m * pr(m,n-1)
```

Definiere $r(n)$ als Anzahl der ZE, die zur Berechnung von `pr(m,n)` benötigt werden.

- $r(0) = 2$
- $r(n) = 3 + r(n-1)$ für $n > 0$
- $r(n) \approx 3 \cdot n \in \mathcal{O}(n)$

Rekurrenzrelationen

```
def pr(m,n):
  if n==0:
    return 1
  else:
    return m * pr(m,n-1)
```

Definiere $r(n)$ als Anzahl der ZE, die zur Berechnung von `pr(m,n)` benötigt werden.

- $r(0) = 2$
- $r(n) = 3 + r(n-1)$ für $n > 0$
- $r(n) \approx 3 \cdot n \in \mathcal{O}(n)$

```
def pe(m,n):
  if n==0:
    return 1
  else:
    p = pe(m, n // 2)
    if n % 2 == 0:
      return p*p
    else:
      return p*p*m
```


Rekurrenzrelationen

```
def pr(m,n):
  if n==0:
    return 1
  else:
    return m * pr(m,n-1)
```

Definiere $r(n)$ als Anzahl der ZE, die zur Berechnung von `pr(m,n)` benötigt werden.

- $r(0) = 2$
- $r(n) = 3 + r(n-1)$ für $n > 0$
- $r(n) \approx 3 \cdot n \in \mathcal{O}(n)$

```
def pe(m,n):
  if n==0:
    return 1
  else:
    p = pe(m, n // 2)
    if n % 2 == 0:
      return p*p
    else:
      return p*p*m
```

- $r(0) = 2$
- $r(n) = 6 + r(\lfloor \frac{n}{2} \rfloor)$ für $n > 0$
- $r(n) \approx 6 \cdot \log_2 n$

Divide and Conquer

Die effiziente Lösung teilt das Problem (n-fache Multiplikation) in 2 Hälften und muss im nächsten Schritt nur ein halb so großes Problem lösen.

Divide and Conquer

Die effiziente Lösung teilt das Problem (n-fache Multiplikation) in 2 Hälften und muss im nächsten Schritt nur ein halb so großes Problem lösen.

Divide-and-Conquer-Algorithmus

Ein Algorithmus, der

- ein Problem im mehrere Teile aufspaltet,
- die Teilprobleme (rekursiv) löst und
- die Teillösungen zu einer Gesamtlösung kombiniert.

Übung: Divide-and-Conquer-Suche

Schreiben Sie einen effizienten Algorithmus zur Suche in einem sortierten Array, der auf Divide and Conquer beruht.

- Gehen Sie davon aus, dass das Array von $0 - (n-1)$ besetzt ist

Komplexität von Divide-and-Conquer-Algorithmen

Master-Theorem

$$f(n) = \underbrace{a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Kombination zur Gesamtlösung}} \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

wobei $a \in \mathbb{N}^{\geq 1}, b \in \mathbb{N}^{\geq 2}, d \in \mathbb{R}^{\geq 0}$. Dann gilt:

1. $a < b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(n^d)$
2. $a = b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(\log_b n \cdot n^d)$
3. $a > b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(n^{\log_b a})$

Anschaulich:

1. Kombination dominiert Rekursion
2. Kombination und Rekursion haben gleichermaßen Einfluss auf Gesamtkomplexität
3. Rekursion dominiert Kombination

Beispiel: Master-Theorem

$$f(n) = a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor) + c(n) \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

Formel für effiziente Potenzberechnung:

$$f(n) = f(\left\lfloor \frac{n}{2} \right\rfloor) + 6$$

Beispiel: Master-Theorem

$$f(n) = a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor) + c(n) \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

Formel für effiziente Potenzberechnung:

$$f(n) = f(\left\lfloor \frac{n}{2} \right\rfloor) + 6$$

$a = 1, b = 2, d = 0 \quad \Rightarrow \quad 1 = 2^0 \quad \Rightarrow \quad$ Fall 2

Beispiel: Master-Theorem

$$f(n) = a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor) + c(n) \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

Formel für effiziente Potenzberechnung:

$$f(n) = f(\left\lfloor \frac{n}{2} \right\rfloor) + 6$$

$a = 1, b = 2, d = 0 \quad \Rightarrow \quad 1 = 2^0 \quad \Rightarrow \quad \text{Fall 2} : f(n) \in \mathcal{O}(\log_b n \cdot n^d)$

Beispiel: Master-Theorem

$$f(n) = a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor) + c(n) \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

Formel für effiziente Potenzberechnung:

$$f(n) = f(\left\lfloor \frac{n}{2} \right\rfloor) + 6$$

$a = 1, b = 2, d = 0 \quad \Rightarrow \quad 1 = 2^0 \quad \Rightarrow \quad \text{Fall 2} : f(n) \in \mathcal{O}(\log_b n \cdot n^d)$

$$f(n) \in \mathcal{O}(\log_2 n \cdot n^0) = \mathcal{O}(\log n)$$

Übung/Hausaufgabe: Master-Theorem

1. Berechnen Sie anhand des Master-Theorems die Komplexität des Algorithmus zur binären Suche.
2. Wenden Sie das Master-Theorem auf die folgenden Rekurrenz-Gleichungen an:
   1. $f(n) = 4 \cdot f(\frac{n}{2}) + n$
   2. $f(n) = 4 \cdot f(\frac{n}{2}) + n^2$
   3. $f(n) = 4 \cdot f(\frac{n}{2}) + n^3$

Ende Vorlesung 7

Weitere Notationen

$g \in \mathcal{O}(f)$ g wächst höchstens so schnell wie f

$\lim_{x \to \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$

Weitere Notationen

$g \in \mathcal{O}(f)$ g wächst höchstens so schnell wie f
$\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$

Ω-, Θ-, ∼-Notation

$g \in \Omega(f)$ g wächst mindestens so schnell wie f
$\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$

$g \in \Theta(f)$ g wächst genau so schnell wie f,
bis auf einen konstanten Faktor
$\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$

$g \sim f$ g wächst genau so schnell wie f,
ohne konstanten Faktor (Sedgewick)
$\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$

Übung: $\mathcal{O}, \Omega, \Theta, \sim$

- Betrachten Sie folgende Funktionen:
  - $h_1(x) = x^2 + 100x + 3$
  - $h_2(x) = x^2$
  - $h_3(x) = \frac{1}{3}x^2 + x$
  - $h_4(x) = x^3 + x$

$g \in \mathcal{O}(f)$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$
$g \in \Omega(f)$: $\lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$
$g \in \Theta(f)$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$
$g \sim f$: $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$

Vervollständigen Sie die Tabelle. Zeile steht in Relation . . . zu Spalte:

	h_1	h_2	h_3	h_4
h_1	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$		
h_2		$\mathcal{O}, \Omega, \Theta, \sim$		
h_3			$\mathcal{O}, \Omega, \Theta, \sim$	
h_4				$\mathcal{O}, \Omega, \Theta, \sim$

Übung: $\mathcal{O}, \Omega, \Theta, \sim$

- Betrachten Sie folgende Funktionen:
  - $h_1(x) = x^2 + 100x + 3$
  - $h_2(x) = x^2$
  - $h_3(x) = \frac{1}{3}x^2 + x$
  - $h_4(x) = x^3 + x$

$$g \in \mathcal{O}(f): \lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$$
$$g \in \Omega(f): \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$$
$$g \in \Theta(f): \lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$$
$$g \sim f: \lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$$

Vervollständigen Sie die Tabelle. Zeile steht in Relation . . . zu Spalte:

	h_1	h_2	h_3	h_4
h_1	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_2	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_3	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}$
h_4	Ω	Ω	Ω	$\mathcal{O}, \Omega, \Theta, \sim$

Arrays

Arrays/Felder

- Ein Array (deutsch: *Feld*, der Begriff ist aber mehrdeutig) ist eine Datenstruktur zur Speicherung einer Anzahl von gleichartigen Datensätzen
- Eindimensionale Standard-Arrays:
  - Zugriff auf die Elemente erfolgt über einen ganzahligen Index
  - Arrays haben festgelegte Größe (z.B. n Elemente)
  - Indices in C laufen von 0 bis $n-1$
    - Andere Sprachen erlauben z.T. Indices von $1-n$ oder von $m-(m+n-1)$
  - C hat keine Sicherheitsgurte
    - Keine Gültigkeitsprüfung beim Zugriff
    - Effizienz vor Sicherheit

Arrays im Speicher

- Array-Elemente werden im Speicher sequentiell abgelegt
  - Kein Speicheroverhead pro Element!
- Die Adresse von Element i errechnet sich aus der Basisadresse b des Arrays, und der Größe eines einzelnen Elements s: $addr(i) = b + i * s$
- Damit gilt: Zugriff auf ein Element über den Index hat (unten den üblichen Annahmen) Kostenfunktion $\mathcal{O}(1)$
  - Zugriffskosten sind unabhängig von i
  - Zugriffskosten sind unabhängig von der Größe des Arrays

Arrays im Speicher

- Array-Elemente werden im Speicher sequentiell abgelegt
  - Kein Speicheroverhead pro Element!
- Die Adresse von Element i errechnet sich aus der Basisadresse b des Arrays, und der Größe eines einzelnen Elements s: $addr(i) = b + i * s$
- Damit gilt: Zugriff auf ein Element über den Index hat (unten den üblichen Annahmen) Kostenfunktion $\mathcal{O}(1)$
  - Zugriffskosten sind unabhängig von i
  - Zugriffskosten sind unabhängig von der Größe des Arrays
- Im Prinzip ist der Speicher eines modernen Rechners ein Array von Bytes
  - Adressen-Transformation ist lineare Abbildung

Beispiel: Wegpunkt-Tabelle eines Luftraums

- Basiseigenschaften:
  - Geographische Länge (`double`, 8 B)
  - Geographische Breite(`double`, 8 B)
  - Name des Wegpunkte (`char[4]`, 4 Bytes)
- In C:

```
typedef struct waypoint
{
    double lon;
    double lat;
    char   name[4];
}waypoint;

waypoint wps[1024];
```

Implementierung im Speicher

Index	lat	lon	name
0	60.3124	17.0122	WUXA
1	61.9811	17.9212	FARI
2	59.1373	18.1192	PIRI
3	62.3212	16.9981	BALA
4	60.0134	19.1966	KORU
. . .			
1023	0	0	\0\0\0\0

Implementierung im Speicher

Index	lat	lon	name
0	60.3124	17.0122	WUXA
1	61.9811	17.9212	FARI
2	59.1373	18.1192	PIRI
3	62.3212	16.9981	BALA
4	60.0134	19.1966	KORU
. . .			
1023	0	0	\0\0\0\0

Addr	Value
base	lat1
base+4	lat2
base+8	lon1
base+12	lon2
base+16	name
base+20	lat1
base+24	lat2
base+28	lon1
base+32	lon2
base+36	name
base+40	lat1
base+44	lat2
base+48	lon1
base+52	lon2
base+56	name
base+60	lat1
base+64	lat2
base+68	lon1
base+72	lon2
base+76	name
base+80	lat1
base+84	lat2
base+88	lon1
base+92	lon2
base+96	name
. . .	. . .
base+20476	name

Implementierung im Speicher

Index	lat	lon	name
0	60.3124	17.0122	WUXA
1	61.9811	17.9212	FARI
2	59.1373	18.1192	PIRI
3	62.3212	16.9981	BALA
4	60.0134	19.1966	KORU
. . .			
1023	0	0	\0\0\0\0

```
&wps = base
   &wps[0] = base
      &wps[0].lat  = base
      &wps[0].lon  = base+8
      &wps[0].name = base+16
   &wps[1] = base+20
      &wps[1].lat  = base+20
      &wps[1].lon  = base+28
      &wps[1].name = base+36
   &wps[2] = base+40
      &wps[2].lat  = base+40
      &wps[2].lon  = base+48
      &wps[2].name = base+56
. . .
   &wps[k] = base+k*20
      &wps[k].lat  = base+k*20
      &wps[k].lon  = base+k*20+8
      &wps[k].name = base+k*20+16
```

Addr	Value
base	lat1
base+4	lat2
base+8	lon1
base+12	lon2
base+16	name
base+20	lat1
base+24	lat2
base+28	lon1
base+32	lon2
base+36	name
base+40	lat1
base+44	lat2
base+48	lon1
base+52	lon2
base+56	name
base+60	lat1
base+64	lat2
base+68	lon1
base+72	lon2
base+76	name
base+80	lat1
base+84	lat2
base+88	lon1
base+92	lon2
base+96	name
. . .	. . .
base+20476	name

Verwendung von Arrays

- Eigenständig
  - Für Daten, die weitgehend statisch sind
  - Für Daten, die eine überschaubare Maximalgröße haben (müssen)
  - Für mathematische Objekte (Vektoren, Matrizen)

Verwendung von Arrays

- Eigenständig
  - Für Daten, die weitgehend statisch sind
  - Für Daten, die eine überschaubare Maximalgröße haben (müssen)
  - Für mathematische Objekte (Vektoren, Matrizen)
- Als unterliegende Basisdatenstruktur für komplexere Datentypen
  - Listen (verwende Indexwerte zur Verkettung)
  - Bäume (verwende Indexwerte zur Verzeigerung)
  - Heaps (durch geschickte Organisation ohne(!) Verzeigerung)
  - Stacks (mit Indexwert als Stackpointer)

Arrays zur Datenhaltung

- Arrays können als Tabellen für die Datenhaltung verwendet werden
  - Typischerweise ein Schlüssel per Array-Element
  - Zusätzliche Daten, die mit dem Schlüssel assoziiert sind
- Beispiele:
  - Meßreihen: Zeit (Schlüssel) und Temperatur
  - Bücher: ISBN (Schlüssel), Autor, Titel, Jahr ...
  - Studenten: Matrikelnummer, Name, Email, Notenlisten ...
- Umsetzung:
  - Array der Größe n mit $k < n$ Einträgen
  - Zähler zeigt auf das erste freie Element

Ende	9

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
12	
14	
15	

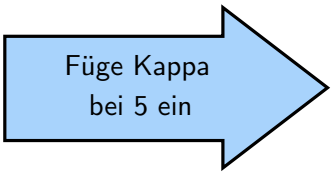
Standard-Operationen

- Iterieren über alle Array-Elemente
- Einfügen eines Elements
  - Am Ende
  - In der Mitte
- Löschen eines Eintrags
  - Am Ende
  - In der Mitte
  - Mit einem bestimmten Wert
- Suchen eines Eintrags
  - In unsortiertem Array
  - In sortiertem Array
- Sortieren (eigene Vorlesung)

Beispiel: Einfügen an Indexposition

Ende	9

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
12	
14	
15	



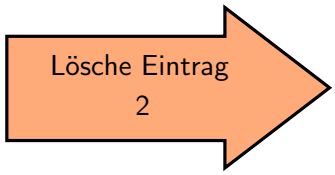
Ende	10

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Kappa
6	Zeta
7	Eta
8	Theta
9	Iota
10	
11	
12	
12	
14	
15	

Beispiel: Löschen an Indexposition

Ende	10

Index	Wert
0	Alpha
1	Beta
2	Gamma
3	Delta
4	Epsilon
5	Kappa
6	Zeta
7	Eta
8	Theta
9	Iota
10	
11	
12	
12	
14	
15	



Ende	9

Index	Wert
0	Alpha
1	Beta
2	Delta
3	Epsilon
4	Kappa
5	Zeta
6	Eta
7	Theta
8	Iota
9	
10	
11	
12	
12	
14	
15	

Übung

- Betrachten Sie ein Array `arr` von `n` Elementen der Größe `m` GE, von denen die ersten `k` benutzt sind
- Löschen
  - Entwickeln Sie einen Algorithmus, der ein Element an der Stelle `u` entfernt
  - Wie viele ZE benötigt der Algorithmus?
- Einfügen (Mitte)
  - Entwickeln Sie einen Algorithmus, der ein Element an der Stelle `u` einfügt
  - Wie viele ZE benötigt der Algorithmus?
- Einfügen (Ende)
  - Entwickeln Sie einen Algorithmus, der ein Element an der Stelle `k` einfügt
  - Wie viele ZE benötigt der Algorithmus?

Suchen im Array

- Lineare Suche:

```
def find_key(arr, n, key):
    for i in range(0,n):
        if arr[i].key() == key:
            return i
    return -1
```

  - Komplexität: $\mathcal{O}(n)$
- Bei sortierten Array besser:
  - Binäre Suche
  - $\mathcal{O}(\log n)$

Suchen im Array

- Lineare Suche:

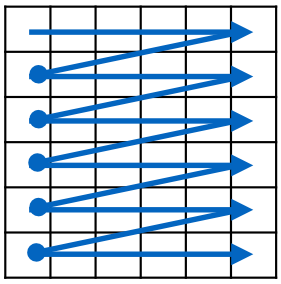
```
def find_key(arr, n, key):
    for i in range(0,n):
        if arr[i].key() == key:
            return i
    return -1
```

  - Komplexität: $\mathcal{O}(n)$
- Bei sortierten Array besser:
  - Binäre Suche
  - $\mathcal{O}(\log n)$
  - Tradeoff: Sortiertes Einfügen ist $\mathcal{O}(n)$, unsortiertes ist $\mathcal{O}(1)$

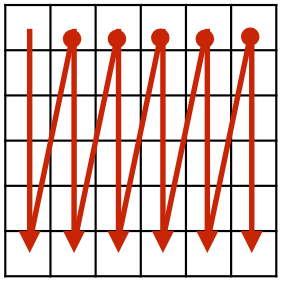
Zweidimensionale Arrays

- Arrays können auf den zwei- und mehrdimensionalen Fall verallgemeinert werden
  - Zeilen- oder Spalten liegen sequentiell in Speicher
- C-Familie, Python: Row-major order
  - `int array[5][9];`
  - Array mit 5 Zeilen a 9 Einträge (Spalten)
  - `array[2][7]`: 7. Element der 2. Zeile
- Fortran, OpenGL,. . . : Colum-major order
  - `REAL, DIMENSION(9,5) :: A`

Row-major order
Zeilenweise Ablage



Column-major order
Spaltenweise Ablage



Ende Vorlesung 8

Array Scorecard 1: Zugriff per Index

Voraussetzung:

- Standard-Array fester Größe m
- Dicht gefüllt mit $n < m$ Elementen (`a[0]-a[n-1]` besetzt)
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Kostenfunktion
Zugriff per Index	$\mathcal{O}(1)$
Einfügen an Positon i	$\mathcal{O}(n)$
Löschen an Positon i	$\mathcal{O}(n)$
Array sortieren	$\mathcal{O}(n \log n)$

Array Scorecard 2: Schlüsselverwaltung

Voraussetzung:

- Standard-Array fester Größe m
- Dicht gefüllt mit $n < m$ Elementen (`a[0]-a[n-1]` besetzt)
- Einträge sind Schlüssel, Schlüsselmenge ist sortierbar
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Unsortiertes Array	Sortiertes Array
Schlüssel finden	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Schlüssel einfügen	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Schlüssel bedingt einfügen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel löschen	$\mathcal{O}(1)$ [1]	$\mathcal{O}(n)$
Array sortieren	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$

[1] Trick: Letztes Element im Array zieht in die Lücke um!

Listen

Listen

- Verkettete Listen sind eine dynamische Datenstruktur zum Speichern einer Menge von Elementen
  - Listen bestehen aus Knoten oder Zellen und Zeigern oder Pointern, die diese miteinander verbinden
  - Lineare Listen: Jede Zelle hat höchstens einen Vorgänger und höchstens einen Nachfolger
  - Jede Zelle außer der ersten hat genau einen Vorgänger
  - Jede Zelle außer der letzten hat genau einen Nachfolger

Listen

- Verkettete Listen sind eine dynamische Datenstruktur zum Speichern einer Menge von Elementen
  - Listen bestehen aus Knoten oder Zellen und Zeigern oder Pointern, die diese miteinander verbinden
  - Lineare Listen: Jede Zelle hat höchstens einen Vorgänger und höchstens einen Nachfolger
  - Jede Zelle außer der ersten hat genau einen Vorgänger
  - Jede Zelle außer der letzten hat genau einen Nachfolger
- Listeninhalte können homogen oder heterogen sein
  - Homogen: Alle Zellen haben eine *Nutzlast* vom selben Typ
  - Heterogen: Zellen haben verschiedene Nutzlasten
    - Implementierung typischerweise pseudo-homogen mit Pointern auf die tatsächliche Nutzlast

Listen

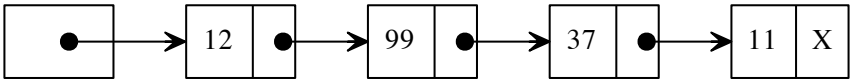
- Verkettete Listen sind eine dynamische Datenstruktur zum Speichern einer Menge von Elementen
  - Listen bestehen aus Knoten oder Zellen und Zeigern oder Pointern, die diese miteinander verbinden
  - Lineare Listen: Jede Zelle hat höchstens einen Vorgänger und höchstens einen Nachfolger
  - Jede Zelle außer der ersten hat genau einen Vorgänger
  - Jede Zelle außer der letzten hat genau einen Nachfolger
- Listeninhalte können homogen oder heterogen sein
  - Homogen: Alle Zellen haben eine *Nutzlast* vom selben Typ
  - Heterogen: Zellen haben verschiedene Nutzlasten
    - Implementierung typischerweise pseudo-homogen mit Pointern auf die tatsächliche Nutzlast
- Historisch:
  - Prozedurale Sprachen nutzen primär Arrays
  - Funktionale Sprachen nutzen primär Listen
    - LISP - LISt Processing
    - Scheme, Caml, Haskell, ...

Einfach und doppelt verkettete Listen

- Zwei wesentliche Varianten
  - Einfach verkettete Listen (“singly linked lists”)
    - Jeder Zelle zeigt auf ihren Nachfolger
  - Doppelt verkettete Listen (“doubly linked lists”)
    - Jede Liste hat zwei Pointer auf Vorgänger und Nachfolger

Einfach verkettete Listen

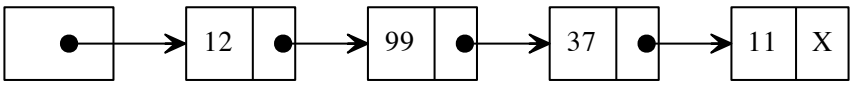
- Listen bestehen aus Listenzellen
  - Nutzlast (Wert(e) oder Pointer)
  - Verweis auf Nachfolger (Pointer)
- Lineare Listen sind rekursiv definiert
  - Die leere Liste ist eine Liste
    - Repräsentiert als Ende-Marker (`NULL/nil/'()`)
  - Eine Listenzelle, deren Nachfolger eine Liste ist, ist eine Liste



- Besonders effiziente Operationen:
  - Einfügen am Anfang (`cons`)
  - Ausfügen am Anfang (`cdr`)

Einfach verkettete Listen

- Listen bestehen aus Listenzellen
  - Nutzlast (Wert(e) oder Pointer)
  - Verweis auf Nachfolger (Pointer)
- Lineare Listen sind rekursiv definiert
  - Die leere Liste ist eine Liste
    - Repräsentiert als Ende-Marker (`NULL/nil/'()`)
  - Eine Listenzelle, deren Nachfolger eine Liste ist, ist eine Liste

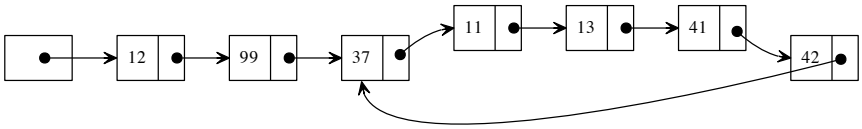


- Besonders effiziente Operationen:
  - Einfügen am Anfang (`cons`)
  - Ausfügen am Anfang (`cdr`)

Siehe auch Labor

Zyklenerkennung

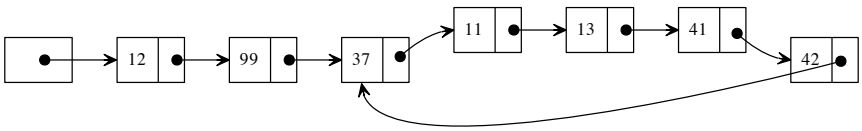
- Zyklische Listen sind keine echten Listen
  - Siehe Definition oben!
  - Sie können aber aus den selben Zellen aufgebaut werden



- Frage: Wie kann man zyklische Listen erkennen?
  - Ideen?

Zyklenerkennung

- Zyklische Listen sind keine echten Listen
  - Siehe Definition oben!
  - Sie können aber aus den selben Zellen aufgebaut werden



- Frage: Wie kann man zyklische Listen erkennen?
  - Ideen?
  - Behauptung: Das geht in $\mathcal{O}(n)$ Zeit und mit $\mathcal{O}(1)$ Speicher!

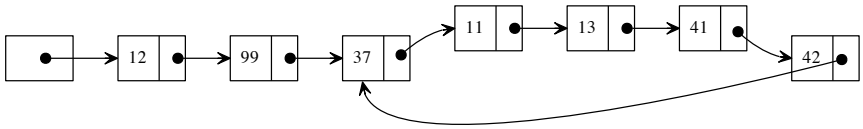
Hase und Igel

- Zyklenerkennung nach (nach Knuth) Floyd
  - Eingabe: Pointer auf eine Listenzelle
  - Ausgabe: `True`, falls Zyklus, `False` sonst

```
def list_is_cyclic(l):
    hase = l
    igel = l
    while True:
        hase = hase.succ
        if not hase:
            # Liste ist zyklenfrei
            return False
        hase = hase.succ
        if not hase:
            return False
        igel = igel.succ
        if hase == igel:
            # Liste hat Zyklus
            return True
```

Übung: Hase und Igel

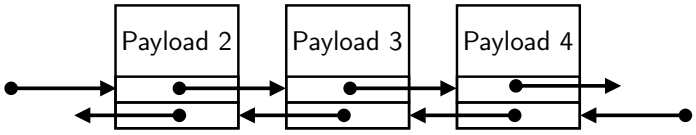
- Spielen Sie den Algorithmus von Hase und Igel mit folgener Liste



  durch
  - Nach wie vielen Schritten wird der Zyklus erkannt?
- Entwerfen Sie je ein weiteres Beispiel und spielen Sie es durch.
- Können Sie allgemein angeben, nach wie vielen Schritten der Algorithmus im schlimmsten Fall abbricht?

Doppelt verkettete Listen

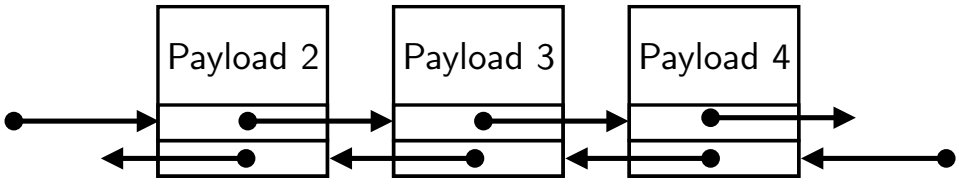
- Doppelt verkettete Listenzellen haben zwei Pointer
  - Auf den Vorgänger
  - Auf den Nachfolger



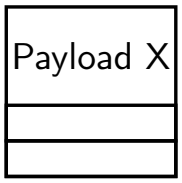
- Entsprechend ist die Liste vorne und hinten verankert
- Vorteile:
  - Einfügen ist vorne und hinten in $\mathcal{O}(1)$ möglich
  - Einfügen ist vor und hinter einem beliebigen gegebenen Element einfach möglich
  - Ausfügen einer Zelle ist einfach und in $\mathcal{O}(1)$ möglich

Beispiel: Einfügen in doppelt verkettete Listen

- Einfügen vor Zelle X: *Aufschneiden* und Einsetzen

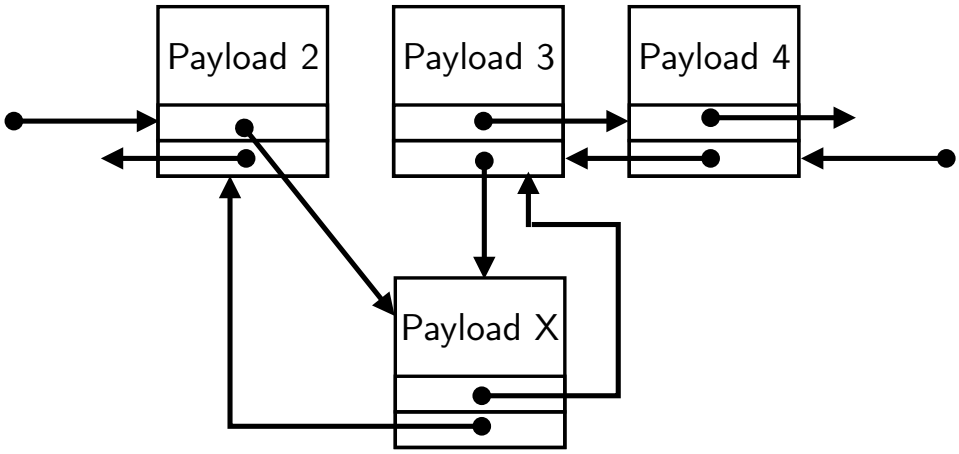


Einfügen X vor 3:



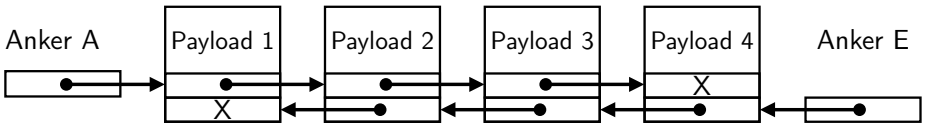
Beispiel: Einfügen in doppelt verkettete Listen

- Einfügen vor Zelle X: *Aufschneiden* und Einsetzen



Verankerung von doppelt verketteten Listen (1)

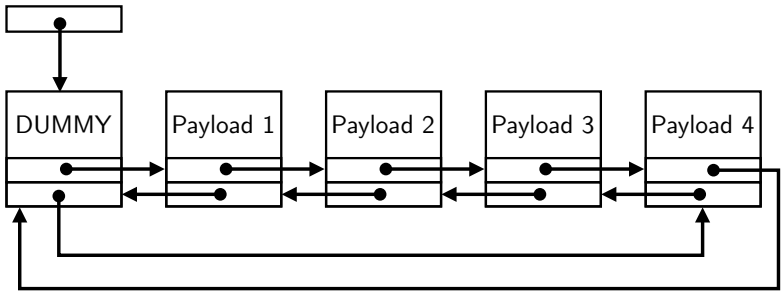
- Methode 1: Zwei Anker



  - Zeiger auf erstes und letztes Element
  - Vorteil: Minimaler Speicheroverhead
  - Nachteile:
    - Bei Funktionsaufrufen müssen beide Anker übergeben werden
    - Viele Spezialfälle beim Einfügen und Ausfügen von Randzellen

Verankerung von doppelt verketteten Listen (2)

- Methode 2: Zyklische Liste



  - Ein Zeiger auf Dummy-Zelle
  - Zellen bilden eine Ring-Struktur
  - Jede Zelle hat echten Vorgänger und Nachfolger
  - Vorteile:
    - Keine Spezialfälle
    - Liste wird durch einen Pointer repräsentiert
    - Keine Pointer auf Pointer notwendig ...
  - Nachteil:
    - Etwas mehr Speicheroverhead (ca. 1 Payload)

Übung: Einfügen und Ausfügen

- Gehen Sie von einem Listenzellentyp mit den Feldern `l.pred` (Vorgänger) und `l.succ` (Nachfolger) aus und nehmen Sie doppelt verkettete Listen mit zyklischer Struktur und einfachem Anker an
- Geben Sie Pseudo-Code für folgende Operationen an:
  - Einfügen einer Zelle `n` nach einer Zelle `l`, die bereits in der Liste ist, die an der Dummy-Zelle `d` verankert ist
  - Ausfügen einer Zelle `z`, die in einer Liste ist, die an der Dummy-Zelle `t` verankert ist

Listen vs. Arrays

Listen	**Arrays**
Elemente liegen unabhängig im Speicher	Elemente müssen im Speicher aufeinander folgen
Länge dynamisch	Länge in der Regel fest
Direkter Zugriff nur auf Randelemente	Freier Zugriff auf alle Elemente
Teilen und Vereinigen einfach	Teilen und Vereinigen erfordert Kopieren
Suche immer sequentiell	Bei sortierten Arrays binäre Suche mögliche
Einfügen/Ausfügen von Elementen billig	Einfügen/Ausfügen bei Erhalten der Ordnung relativ teuer

Listen Scorecard 1

Voraussetzung:

- Einfach und doppelt verkettere Listen
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Einfach v.	Doppelt v.
Zugriff an Position i	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Einfügen an Anfang	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Einfügen an Ende	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Löschen am Anfang	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Löschen am Ende	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Löschen einer gegebenen Zelle	$\mathcal{O}(n)$[2]	$\mathcal{O}(1)$
Liste sortieren	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

[2] Bei $\star\star$-Programmierung $\mathcal{O}(1)$ möglich.

Listen Scorecard 2: Schlüsselverwaltung

Voraussetzung:

- Einträge sind Schlüssel
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Einfach v.	Doppelt v.
Schlüssel finden	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Schlüssel bedingt einfügen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel löschen	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Listen Scorecard 2: Schlüsselverwaltung

Voraussetzung:

- Einträge sind Schlüssel
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Einfach v.	Doppelt v.
Schlüssel finden	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Schlüssel bedingt einfügen	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Schlüssel löschen	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Aber beachte: Doppelt verkettete Listen haben doppelte Konstanten und doppelten Speicher-Overhead!

Ende Vorlesung 9

Sortieralgorithmen

Sortieren

Eingabe:

- Folge A von Daten $(a_0, a_1, a_2, \ldots, a_{k-1})$
- Ordnungsrelation $<_A$ auf Daten

Ausgabe:

- Permutation von A: $(a_{s_0}, a_{s_1}, a_{s_2}, \ldots, a_{s_{k-1}})$
- $a_{s_i} <_A a_{s_{i+1}}$ für alle $i \in \{0, \ldots, k-2\}$

Sortieren

Eingabe:

- Folge A von Daten $(a_0, a_1, a_2, \ldots, a_{k-1})$
- Ordnungsrelation $<_A$ auf Daten

Ausgabe:

- Permutation von A: $(a_{s_0}, a_{s_1}, a_{s_2}, \ldots, a_{s_{k-1}})$
- $a_{s_i} <_A a_{s_{i+1}}$ für alle $i \in \{0, \ldots, k-2\}$

In der Vorlesung beispielhaft:

- Daten: natürliche Zahlen
- Ordnung: $<_{\mathbb{N}}$

Sortieren

Eingabe:

- Folge A von Daten $(a_0, a_1, a_2, \ldots, a_{k-1})$
- Ordnungsrelation $<_A$ auf Daten

Ausgabe:

- Permutation von A: $(a_{s_0}, a_{s_1}, a_{s_2}, \ldots, a_{s_{k-1}})$
- $a_{s_i} <_A a_{s_{i+1}}$ für alle $i \in \{0, \ldots, k-2\}$

In der Vorlesung beispielhaft:

- Daten: natürliche Zahlen
- Ordnung: $<_{\mathbb{N}}$

In der Praxis:

- Daten: Datensätze (Records) mit Zahl, String, Datum, ...
- Ordnung: $<_{\mathbb{N}}$, $<_{\mathbb{R}}$, lexikographisch, temporal, ...
- Sortierschlüssel: zum Sortieren verwendete Komponente

Übung: Lexikographische Ordnung

Schreiben Sie eine Funktion `lexgreater (left, right)`, die für zwei Strings `left` und `right` (ohne deutsche Sonderzeichen) entscheidet, ob `left` $>_{\text{lex}}$ `right` gilt (Rückgabewert 1) oder nicht (Rückgabewert 0).

Klassifikation von Sortieralgorithmen

- Verwendete Datenstrukturen
  - **Arrays** Beliebiger Zugriff, Ein-/Ausfügen teuer
  - **Listen** Sequentieller Zugriff, Ein-/Ausfügen günstig
  - **Bäume** „Alles $\mathcal{O}(\log n)$“, oft als Zwischenstufe

Klassifikation von Sortieralgorithmen

- Verwendete Datenstrukturen

Arrays	Beliebiger Zugriff, Ein-/Ausfügen teuer
Listen	Sequentieller Zugriff, Ein-/Ausfügen günstig
Bäume	„Alles $\mathcal{O}(\log n)$“, oft als Zwischenstufe

- Verhältnis der benötigten Operationen

Vergleiche	Test der Ordnung von zwei Schlüsseln
Zuweisungen	Ändern von Elementen der Folge

Klassifikation von Sortieralgorithmen

- Verwendete Datenstrukturen

Arrays	Beliebiger Zugriff, Ein-/Ausfügen teuer
Listen	Sequentieller Zugriff, Ein-/Ausfügen günstig
Bäume	„Alles $\mathcal{O}(\log n)$“, oft als Zwischenstufe

- Verhältnis der benötigten Operationen

Vergleiche	Test der Ordnung von zwei Schlüsseln
Zuweisungen	Ändern von Elementen der Folge

- Benötigter zusätzlicher Speicher

in-place	zusätzlicher Speicherbedarf ist $\mathcal{O}(1)$
out-of-place	zusätzlicher Speicherbedarf ist $\mathcal{O}(n)$

Klassifikation von Sortieralgorithmen

- Verwendete Datenstrukturen

Arrays	Beliebiger Zugriff, Ein-/Ausfügen teuer
Listen	Sequentieller Zugriff, Ein-/Ausfügen günstig
Bäume	„Alles $\mathcal{O}(\log n)$“, oft als Zwischenstufe

- Verhältnis der benötigten Operationen

Vergleiche	Test der Ordnung von zwei Schlüsseln
Zuweisungen	Ändern von Elementen der Folge

- Benötigter zusätzlicher Speicher

in-place	zusätzlicher Speicherbedarf ist $\mathcal{O}(1)$
out-of-place	zusätzlicher Speicherbedarf ist $\mathcal{O}(n)$

- Stabilität: Auswirkung auf Elemente mit gleichem Schlüssel

stabil	relative Reihenfolge bleibt erhalten
instabil	relative Reihenfolge kann sich ändern

Beispiel: Indirektes Sortieren

- Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer

Beispiel: Indirektes Sortieren

- Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer
- Lösung: Indirektes Sortieren mit Permutationsindex

1	D
2	B
3	A
4	E
5	C

Beispiel: Indirektes Sortieren

- Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer
- Lösung: Indirektes Sortieren mit Permutationsindex

1	D
2	B
3	A
4	E
5	C

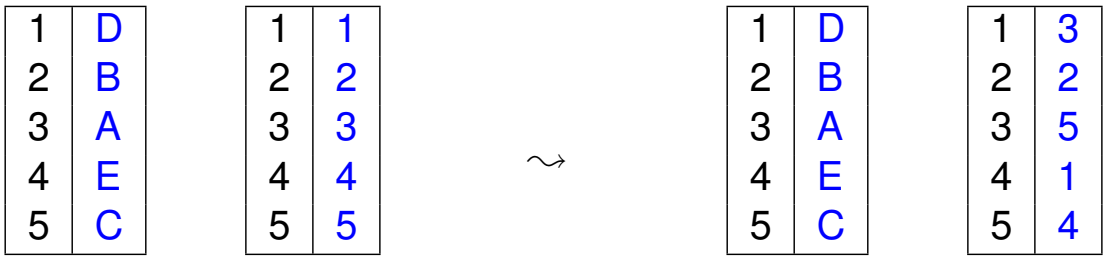
1	1
2	2
3	3
4	4
5	5

Beispiel: Indirektes Sortieren

- Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer
- Lösung: Indirektes Sortieren mit Permutationsindex

Beispiel: Indirektes Sortieren

- Problem: Bei großen Datensätzen ist das Vertauschen von Einträgen sehr teuer
- Lösung: Indirektes Sortieren mit Permutationsindex



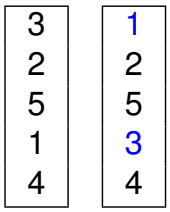
- Jeder Lesezugriff benötigt einen zusätzlichen Lesezugriff auf das neue Array
- Typischerweise akzeptabel, wenn der Index im Hauptspeicher ist

Beispiel: In-place und Out-of-place

In-place

Beispiel: In-place und Out-of-place

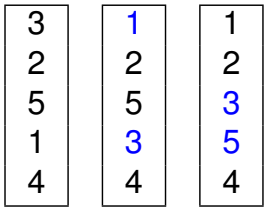
In-place



105

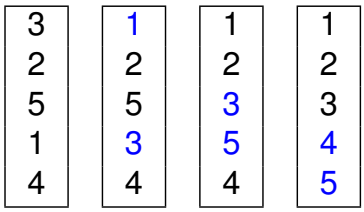
Beispiel: In-place und Out-of-place

In-place



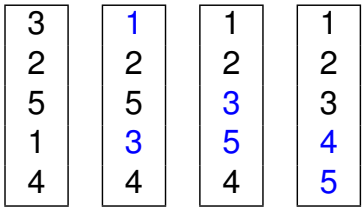
Beispiel: In-place und Out-of-place

In-place

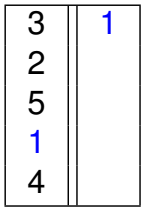


Beispiel: In-place und Out-of-place

In-place

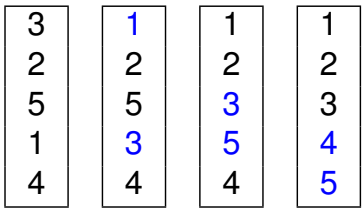


Out-of-place

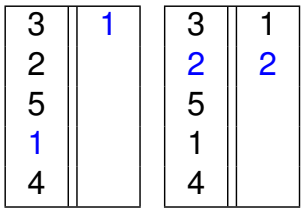


Beispiel: In-place und Out-of-place

In-place

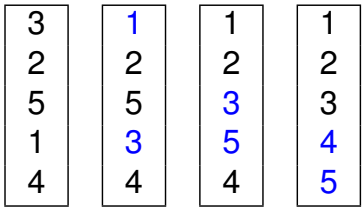


Out-of-place

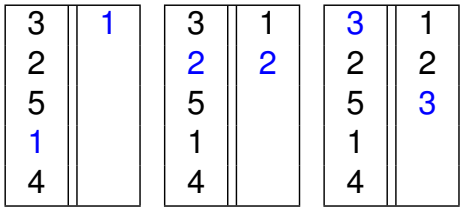


Beispiel: In-place und Out-of-place

In-place



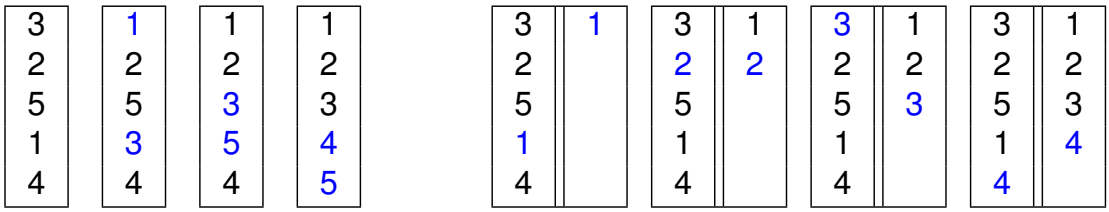
Out-of-place



Beispiel: In-place und Out-of-place

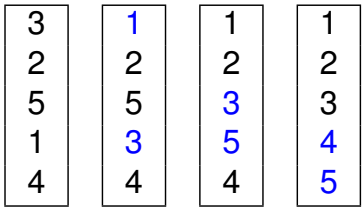
In-place

Out-of-place

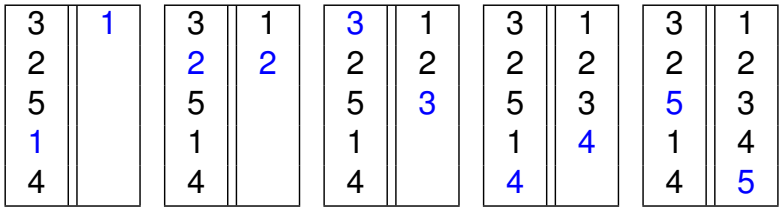


Beispiel: In-place und Out-of-place

In-place

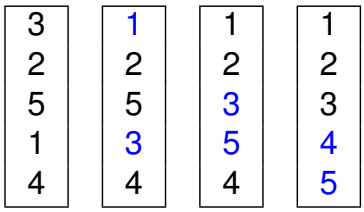


Out-of-place

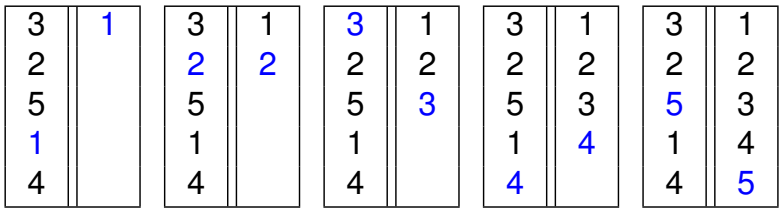


Beispiel: In-place und Out-of-place

In-place



Out-of-place



Vorteil Zusätzlicher Speicher $\mathcal{O}(1)$

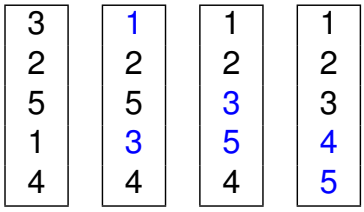
Nachteil Nur Austausch von 2 Elementen

Vorteil Komplexere Tausch-Operationen möglich ($\rightsquigarrow$ Mergesort)

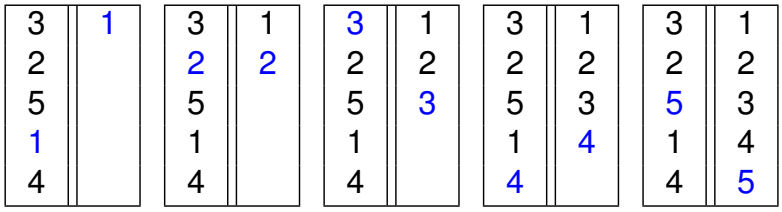
Nachteil Zusätzlicher Speicher $\mathcal{O}(n)$

Beispiel: In-place und Out-of-place

In-place



Out-of-place



Vorteil	Zusätzlicher Speicher $\mathcal{O}(1)$	Vorteil	Komplexere Tausch-Operationen möglich ($\rightsquigarrow$ Mergesort)
Nachteil	Nur Austausch von 2 Elementen	Nachteil	Zusätzlicher Speicher $\mathcal{O}(n)$

Bei Listen oder indirektem Sortieren ist der Unterschied zwischen In-place und Out-of-place gering, da nur Zeiger / Indizes verändert werden.

Beispiel: Stabiles und instabiles Sortieren

Original

Name	Kurs
Anna	C
Anton	B
Bert	A
Christoph	C
Conny	A
Dirk	B
Ernie	A
Frank	C
Grit	B

Stabil

Instabil

Beispiel: Stabiles und instabiles Sortieren

Original

Name	Kurs
Anna	C
Anton	B
Bert	A
Christoph	C
Conny	A
Dirk	B
Ernie	A
Frank	C
Grit	B

Stabil

Name	Kurs
Bert	A
Conny	A
Ernie	A
Anton	B
Dirk	B
Grit	B
Anna	C
Christoph	C
Frank	C

Instabil

Beispiel: Stabiles und instabiles Sortieren

Original

Name	Kurs
Anna	C
Anton	B
Bert	A
Christoph	C
Conny	A
Dirk	B
Ernie	A
Frank	C
Grit	B

Stabil

Name	Kurs
Bert	A
Conny	A
Ernie	A
Anton	B
Dirk	B
Grit	B
Anna	C
Christoph	C
Frank	C

Instabil

Name	Kurs
Ernie	A
Bert	A
Conny	A
Grit	B
Dirk	B
Anton	B
Christoph	C
Anna	C
Frank	C

Beispiel: Stabiles und instabiles Sortieren

Original

Name	Kurs
Anna	C
Anton	B
Bert	A
Christoph	C
Conny	A
Dirk	B
Ernie	A
Frank	C
Grit	B

Stabil

Name	Kurs
Bert	A
Conny	A
Ernie	A
Anton	B
Dirk	B
Grit	B
Anna	C
Christoph	C
Frank	C

Instabil

Name	Kurs
Ernie	A
Bert	A
Conny	A
Grit	B
Dirk	B
Anton	B
Christoph	C
Anna	C
Frank	C

- Einfache Sortierverfahren sind meist stabil, effiziente oft instabil
- Ob Instabilität ein Nachteil ist, hängt von Anwendung ab
  - behebbar durch zusätzliches eindeutiges Feld in Datenstruktur
  - bei eindeutigen Werten (z.B. Primärschlüsseln) unproblematisch

Einfache Sortierverfahren: Selection Sort

1. finde kleinstes Element $a_{\min}$ der Folge $(a_0, \ldots a_{k-1})$
2. vertausche $a_{\min}$ mit a_0
3. finde kleinstes Element $a_{\min}$ der Folge $(a_1, \ldots a_{k-1})$
4. vertausche $a_{\min}$ mit a_1
5. ...

Selection Sort auf Array

```
def sel_sort(arr):
    arrlen = len(arr)
    # i is the position to fill
    for i in range(arrlen-1):
        # Find smallest unsorted
        min_i = i
        for j in range(i+1, arrlen):
            if arr[j] < arr[min_i]:
                min_i = j
        # Swap to correct position
        arr[min_i],arr[i] = \
            arr[i], arr[min_i]
    return arr
```


Selection Sort auf Array

```
def sel_sort(arr):
    arrlen = len(arr)
    # i is the position to fill
    for i in range(arrlen-1):
        # Find smallest unsorted
        min_i = i
        for j in range(i+1, arrlen):
            if arr[j] < arr[min_i]:
                min_i = j
        # Swap to correct position
        arr[min_i],arr[i] = \
            arr[i], arr[min_i]
    return arr
```

Bestimmen Sie:

- Anzahl der Vergleiche
- Anzahl der Integer-Zuweisungen
- Anzahl der Daten-Zuweisungen
- In-/Out-of-Place
- Stabilität

Selection Sort auf Liste

```
def selsortlist (l):
  res = NULL
  while len(l) > 0:
    i = l
    mini = i
    while (i.succ != NULL)
      i = i.succ
      if i.data < mini.data:
        mini = i
    remove(l, mini)
    append(res, mini)
  return res
```


Selection Sort auf Liste

```
def selsortlist (l):
  res = NULL
  while len(l) > 0:
    i = l
    mini = i
    while (i.succ != NULL)
      i = i.succ
      if i.data < mini.data:
        mini = i
    remove(l, mini)
    append(res, mini)
  return res
```

Bestimmen Sie:

- Anzahl der Vergleiche
- Anzahl der Integer- oder Pointer-Zuweisungen
- Anzahl der Daten-Zuweisungen
- Stabilität

Zusammenfassung Selection Sort

- Prinzip: Vertausche kleinstes Element der unsortierten Liste mit ihrem ersten Element
- Zeit-ineffizient: Andere Elemente bleiben unberührt
- Platz-effizient: Dreieckstausch zwischen erstem und kleinstem Element
- einfach zu implementieren
- In-place
- stabil
- $\mathcal{O}(n^2)$

Einfache Sortierverfahren: Insertion Sort

Out-of-place-Version verwendet zwei Folgen

- Eingabe *In*
- Ausgabe *Out* (anfangs leer)

Einfache Sortierverfahren: Insertion Sort

Out-of-place-Version verwendet zwei Folgen

- Eingabe *In*
- Ausgabe *Out* (anfangs leer)

1. verwende erstes Element von *In* als erstes Element von *Out*
2. füge zweites Element von *In* an korrekte Position in *Out*
3. füge drittes Element von *In* an korrekte Position in *Out*
4. ...

Einfache Sortierverfahren: Insertion Sort

Out-of-place-Version verwendet zwei Folgen

- Eingabe *In*
- Ausgabe *Out* (anfangs leer)

1. verwende erstes Element von *In* als erstes Element von *Out*
2. füge zweites Element von *In* an korrekte Position in *Out*
3. füge drittes Element von *In* an korrekte Position in *Out*
4. ...

In-place-Version betrachtet nach Schritt *i* die ersten *i* Elemente als *Out*, die restlichen als *In*

Insertion Sort auf Array

```
def ins_sort(arr):
    arrlen = len(arr)
    # Array arr[0]-arr[i-1] is sorted
    for i in range(1,arrlen-1):
        # Move # arr[i] into
        # the right position
        j = i
        while j>0 and arr[j]<arr[j-1]:
            arr[j],arr[j-1] = \
                arr[j-1], arr[j]
            j=j-1
    return arr
```


Insertion Sort auf Array

```
def ins_sort(arr):
    arrlen = len(arr)
    # Array arr[0]-arr[i-1] is sorted
    for i in range(1,arrlen-1):
        # Move # arr[i] into
        # the right position
        j = i
        while j>0 and arr[j]<arr[j-1]:
            arr[j],arr[j-1] = \
                arr[j-1], arr[j]
            j=j-1
    return arr
```

Bestimmen Sie:

- Anzahl der Vergleiche
- Anzahl der Integer-Zuweisungen
- Anzahl der Daten-Zuweisungen
- In-/Out-of-Place
- Stabilität

Insertion Sort auf Liste

```
def inssortlist (l)
  while len(l) > 0:
    first = l
    l = l.succ
    insert (res, first)
  return res

def insert (l, el)
  if el.data < l.data
    el.succ = l
    l = el
  else
    rest = l
    while rest != NULL and rest.data <= el.data
      if rest.succ = NULL
        rest.succ = el
      else
        if rest.succ.data > el
          el.succ = rest.succ
          rest.succ = el
      rest = rest.succ
```


Insertion Sort auf Liste

```
def inssortlist (l)
  while len(l) > 0:
    first = l
    l = l.succ
    insert (res, first)
  return res

def insert (l, el)
  if el.data < l.data
    el.succ = l
    l = el
  else
    rest = l
    while rest != NULL and rest.data <= el.data
      if rest.succ = NULL
        rest.succ = el
      else
        if rest.succ.data > el
          el.succ = rest.succ
          rest.succ = el
      rest = rest.succ
```

Bestimmen Sie:

- Anzahl der Vergleiche
- Anzahl der Integer-Zuweisungen
- Anzahl der Daten-Zuweisungen
- Stabilität

Zusammenfassung Insertion Sort

- Prinzip: Füge Elemente der Reihe nach an richtiger Position ein
- Zeit-ineffizient:
  - Für jede Einfüge-Operation wird gesamte Liste durchlaufen
  - Im Array müssen alle folgenden Elemente verschoben werden
- Platz-effizient: Nur ein Element als Zwischenspeicher
- einfach zu implementieren
- In-place
- stabil
- $\mathcal{O}(n^2)$

Übung: Bubble Sort

Der Sortieralgorithmus *Bubble Sort* für eine Folge S funktioniert wie folgt:

1. durchlaufe S von Anfang bis Ende
   - wann immer $a_i > a_{i+1}$ gilt, vertausche a_i mit a_{i+1}
2. wiederhole Schritt 1 solange, bis keine Vertauschungen mehr vorkommen

Aufgaben:

1. Sortieren Sie die Folge $A = (3, 5, 2, 4, 1)$ mit Bubble Sort.
2. Welche Komplexität hat Bubble Sort ($\mathcal{O}$-Notation)?

Übung: Bubble Sort

Der Sortieralgorithmus *Bubble Sort* für eine Folge S funktioniert wie folgt:

1. durchlaufe S von Anfang bis Ende
   - wann immer $a_i > a_{i+1}$ gilt, vertausche a_i mit a_{i+1}
2. wiederhole Schritt 1 solange, bis keine Vertauschungen mehr vorkommen

Aufgaben:

1. Sortieren Sie die Folge $A = (3, 5, 2, 4, 1)$ mit Bubble Sort.
2. Welche Komplexität hat Bubble Sort ($\mathcal{O}$-Notation)?
3. Ein Nachteil von Bubble Sort liegt darin, dass kleine Elemente, die am Ende der Liste stehen, nur sehr langsam an die richtige Position „blubbern“. *(Cocktail) Shaker Sort* versucht, dieses Problem zu lösen, indem die Folge abwechselnd vom Anfang und vom Ende her durchlaufen wird. Führt dies zu einer niedrigeren Komplexität?

Ineffizienz der einfachen Sortierverfahren

- Selection Sort
  - gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - Folge besteht aus vollständig sortiertem und vollständig unsortiertem Teil
  - Informationen über nicht-minimale Elemente werden vergessen

Ineffizienz der einfachen Sortierverfahren

- Selection Sort
  - gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - Folge besteht aus vollständig sortiertem und vollständig unsortiertem Teil
  - Informationen über nicht-minimale Elemente werden vergessen
- Insertion Sort
  - Einfügen hat linearen Aufwand
  - Array: alle folgenden Elemente verschieben
  - Liste: korrekte Position muss durch lineare Suche gefunden werden, obwohl Teilliste bereits sortiert ist

Ineffizienz der einfachen Sortierverfahren

- Selection Sort
  - gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - Folge besteht aus vollständig sortiertem und vollständig unsortiertem Teil
  - Informationen über nicht-minimale Elemente werden vergessen
- Insertion Sort
  - Einfügen hat linearen Aufwand
  - Array: alle folgenden Elemente verschieben
  - Liste: korrekte Position muss durch lineare Suche gefunden werden, obwohl Teilliste bereits sortiert ist
- Bubble / Shaker Sort
  - gesamte Folge wird durchlaufen, um ein Element an korrekten Platz zu verschieben
  - zusätzliche Vertauschungen

Effiziente Sortierverfahren

- Divide and Conquer
  - Teile Folge in zwei Teile
  - sortiere Teile rekursiv
  - kombiniere sortierte Teile
  - Reihenfolge der Schritte unterschiedlich
    - Quicksort: Gesamtliste nach Größe organisieren $\rightarrow$ teilen $\rightarrow$ Rekursion
    - Mergesort: teilen $\rightarrow$ Rekursion $\rightarrow$ Teillösungen kombinieren
- Heapsort: spezielle Datenstruktur
  - partiell sortierter Baum
  - partielles Sortieren erfordert $\log n$ Schritte

Quicksort: Prinzip

Eingabe: Folge S

1. Wenn $|S| \leq 1$: fertig
2. Wähle Pivot-Element $p \in S$
   - Pivot: Dreh- und Angelpunkt
   - idealerweise: Mittlere Größe
   - teile Folge in zwei Teilfolgen $S_<$ und $S_\geq$
     - $\forall a \in S_< : a < p$
     - $\forall a \in S_\geq : a \geq p$
3. Sortiere $S_<$ und $S_\geq$ mittels Quicksort

Übung: Partitionieren $S_<$ und $S_\geq$

Betrachten Sie die Folge $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$

- Wählen Sie das Element in der Mitte der Folge als Pivot-Element p und teilen Sie die Folge in zwei Teilfolgen $S_<$ und $S_\geq$, so dass alle Elemente in $S_<$ kleiner als das Pivot sind, und alle Elemente in $S_\geq$ größer sind.
- Wierderholen Sie obige Aufgabe, aber mit dem ersten Folgenelement als Pivot-Element
- Wiederholen Sie die erste Aufgabe, aber erledigen Sie die Aufgabe nur durch vertauschen von Elementen, nicht durch kopieren in eine neue Liste

Partitionieren in $S_<$ und $S_\geq$

Idee für einen Algorithmus

- Eingabe: Folge $S = (s_0, s_1, \ldots s_n)$
- Pivot-Element $p = s_k \in S$
- Vorgehen
  1. Vertausche Pivot-Element und letztes Element
  2. Durchlaufe S mit zwei Zählern su und sp - initial sind beide 0
  3. su wird bei jedem Schleifendurchlauf erhöht
  4. Wenn $s_{su} < p$, dann vertausche s_{sp} und s_{su} und erhöhe sp
  5. Abbruch, wenn $su = n - 1$
  6. Tausche das Pivot-Element s_n und s_{sp}
- Ergebnis: $(s_0, \ldots s_{sp-1}, p, s_{sp+1}, \ldots s_n)$, so dass $s_0 - s_{sp-1} < p$ und $s_{sp+1} - s_n \geq p$

Quicksort: Code

```
def q_part(arr, low, high):
    pivotindex = (low+high)/2
    pivotvalue = arr[pivotindex]
    # Move pivot out of place
    arr[high], arr[pivotindex] = arr[pivotindex], arr[high]
    sp = low
    for su in range(low, high):
        if arr[su]<pivotvalue:
            arr[sp],arr[su] = arr[su], arr[sp]
            sp=sp+1
    arr[sp],arr[high] = arr[high],arr[sp]
    return sp

def q_sort(arr, lo, hi):
    if lo < hi:
        pivotindex = q_part(arr, lo, hi)
        q_sort(arr, lo, pivotindex-1)
        q_sort(arr, pivotindex+1, hi)
```


Übung: Quicksort

- Betrachten Sie die Folge $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$
- Sortieren Sie diese mit dem Quicksort-Algorithmus
  1. Wählen Sie als Pivot immer das erste Element der Folge
  2. Wählen Sie als Pivot immer das Element in der Mitte der Folge

Ende Vorlesung 11

Übung: Analyse Quicksort

1. Was ist der Best Case für Quicksort?
   - Welche Komplexität hat Quicksort dann? (Tip: Master-Theorem)
2. Was ist der Worst Case für Quicksort?
   - Welche Komplexität hat Quicksort dann?
3. Ist Quicksort stabil . . .
   - . . . bei Verwendung von LL-Pointern?
   - . . . bei Verwendung von LR-Pointern?
4. Arbeitet Quicksort in-place?

Warum ist Quicksort effizient?

- QS sortiert zunächst grob, dann immer feiner
- Analogie mit Spielkarten:
  1. nach rot und schwarz aufteilen
  2. nach Herz und Karo bzw. Pik und Kreuz aufteilen
  3. nach 7–10 und B–A aufteilen
  4. . . .
- Elemente, zwischen denen einmal ein Pivot lag, werden nie mehr verglichen
- Ineffizienz von Selection / Insertion Sort wird vermieden:
  - kein wiederholtes Durchlaufen derselben Folge
  - jede Information wird genutzt
  - `arr[i] < pivot` $\rightsquigarrow$ links; `arr[i] > pivot` $\rightsquigarrow$ rechts

Vor- und Nachteile von Quicksort

Vorteile

- in der Praxis oft effizientestes Sortierverfahren (wenn optimiert)
  - Oft Standard-Sortierverfahren in C und Java (`qsort()`)
- In-place

Vor- und Nachteile von Quicksort

Vorteile

- in der Praxis oft effizientestes Sortierverfahren (wenn optimiert)
  - Oft Standard-Sortierverfahren in C und Java (`qsort()`)
- In-place

Nachteile

- Auswahl des Pivot-Elements entscheidend für Effizienz
  - größtes oder kleinstes $\leadsto$ worst-case
  - Problem bei fast sortierten Folgen und Auswahl des ersten oder letzten Elements als Pivot
  - Abhilfe: median-of-three: Median von erstem, letztem, mittlerem Element (aber auch keine Garantie)
- Ineffizient für kurze Folgen
  - Overhead durch Rekursion und Zeigerverwaltung
  - Abhilfe: Verwende für kurze Folgen einfaches Sortierverfahren

Divide and Conquer

- Quicksort teilt die zu sortierende Folge nach Größe
  - Ideal: Hälfte kleine Elemente, Hälfte große Elemente
  - Problem: Wir raten den trennenden Wert $\leadsto$ keine Garantie!

Divide and Conquer

- Quicksort teilt die zu sortierende Folge nach Größe
  - Ideal: Hälfte kleine Elemente, Hälfte große Elemente
  - Problem: Wir raten den trennenden Wert $\leadsto$ keine Garantie!
- Alternative: Teile die zu sortierende Folge nach Position(en)
  - Kann gleichmäßige Teilung garantieren
  - Aber: Sortierte Teilergebnisse können nicht einfach aneinandergehängt werden

Divide and Conquer

- Quicksort teilt die zu sortierende Folge nach Größe
  - Ideal: Hälfte kleine Elemente, Hälfte große Elemente
  - Problem: Wir raten den trennenden Wert $\leadsto$ keine Garantie!
- Alternative: Teile die zu sortierende Folge nach Position(en)
  - Kann gleichmäßige Teilung garantieren
  - Aber: Sortierte Teilergebnisse können nicht einfach aneinandergehängt werden

Dieser Ansatz führt zu Mergesort!

Ende Vorlesung 12

Mergesort: Prinzip

Eingabe: Folge S

1. Wenn $|S| = 1$: gib S zurück
2. Teile S in zwei gleich lange Folgen L und R
3. Sortiere L und R (rekursiv)
4. Vereinige L und R zu S':
   1. solange L oder R nicht leer sind:
   2. $m := min(l_1, r_1)$
   3. entferne m aus L bzw. R
   4. hänge m an S' an
5. gib S' zurück

Mergesort: Code

```
mrg_sort(arr):
    if len(arr) <= 1:
        return arr
    arr1 = arr[:len(arr)/2]
    arr2 = arr[len(arr)/2:]
    arr1 = mrg_sort(arr1)
    arr2 = mrg_sort(arr2)
    e1 = 0; e2 = 0
    for i in range(len(arr)):
        if e1 >= len(arr1):
            arr[i] = arr2[e2]; e2 = e2+1
        elif e2 >= len(arr2):
            arr[i] = arr1[e1]; e1 = e1+1
        elif arr1[e1] <= arr2[e2]:
            arr[i] = arr1[e1]; e1 = e1+1
        else:
            arr[i] = arr2[e2]; e2 = e2+1
    return arr
```

- `arr`: Zu sortierendes Array
- `arr1`: Erste Hälfte
- `arr2`: Erste Hälfte
- `e1`: Index von aktuellem Element von `arr1`
- `e2`: Index von aktuellem Element von `arr2`
- `i`: Position des nächsten Elements in der gemergten Liste

Übung: Mergesort

Sortieren Sie die Folge $S = (2, 17, 5, 3, 9, 4, 11, 13, 5)$ mit Mergesort.

Wenn sich eine Folge nicht in zwei gleich große Hälften teilen lässt, erhält die erste (linke) Teilfolge das zusätzliche Element.

Mergesort auf Liste

Unterschiede zum Array:

- Halbieren der Liste elementweise
  - „eins links, eins rechts“
  - effizienter als Halbieren in der Mitte (2 Durchläufe)
- Mischen der sortierten Listen allein durch Zeiger-Manipulation
  - wie bei Insertion Sort
  - kein Overhead durch zusätzliches Array

Mergesort: Analyse

1. Was ist der Best/Worst Case für Mergesort?
2. Was ist die Komplexität?
3. Ist Mergesort stabil, . . .
   - . . . wenn ein Array in der Mitte geteilt wird?
   - . . . wenn eine Liste elementweise in zwei Listen aufgeteilt wird?
4. Arbeitet Mergesort in-place?

Warum ist Mergesort effizient?

- Mergesort sortiert zuerst im Kleinen, dann im Großen
- Da die Teillisten sortiert sind, wird das kleinste Element in $\mathcal{O}(1)$ gefunden
- Effiziente Nutzung von Information: Elemente werden nur mit ähnlich großen verglichen (Listenanfänge)
- Kein wiederholtes Durchlaufen derselben Folge

Optimierung von Mergesort

- Nachteil: Ineffizienz durch Rekursion bei kurzen Folgen
  - Abhilfe: Verwende Insertion Sort für kurze Folgen
- Nachteil: Overhead durch Rekursion: Sortieren beginnt erst auf maximaler Rekursionstiefe
  - Abhilfe: Bottom-up Mergesort (iterative Variante)
    1. Betrachte jedes Element als ein-elementige Liste
    2. Mische je zwei benachbarte Listen
    3. Mische benachbarte zwei-elementige Listen
    4. Mische benachbarte vier-elementige Listen
    5. ...
  - Eliminiert Overhead durch Rekursion
  - Einziger Unterschied: Jede Liste (bis auf die letzte) hat Länge 2^i statt $\frac{n}{2^i}$

Vor- und Nachteile von Mergesort

Vorteile

- auch im Worst Case effizient
- Stabilität einfach erreichbar

Vor- und Nachteile von Mergesort

Vorteile

- auch im Worst Case effizient
- Stabilität einfach erreichbar

Nachteile

- bei Arrays: Zusätzliches Array zum Mischen nötig
- Etwa um Faktor 2 langsamer als effiziente Implementierung von Quicksort (S. Skiena: Algorithm Design Manual, 2009)

Heaps und Heapsort

Heaps als Datenstruktur

Definition: Heap

Ein (binärer) Heap ist ein (fast) vollständiger binärer Baum, in dem für jeden Knoten gilt, dass er in einer definierten Ordnungsrelation zu seinen Nachfolger steht.

- Max-Heap: Jeder Knoten ist $\geq$ als seine Nachfolger
- Min-Heap: Jeder Knoten ist $\leq$ als seine Nachfolger

Heaps als Datenstruktur

Definition: Heap

Ein (binärer) Heap ist ein (fast) vollständiger binärer Baum, in dem für jeden Knoten gilt, dass er in einer definierten Ordnungsrelation zu seinen Nachfolger steht.

- Max-Heap: Jeder Knoten ist $\geq$ als seine Nachfolger
- Min-Heap: Jeder Knoten ist $\leq$ als seine Nachfolger

- Fast vollständiger Binärbaum:
  - Alle Ebenen bis auf die unterste sind vollständig
  - Die unterste Ebene ist von von links durchgehend besetzt
- Finden des größten Elements mit $\mathcal{O}(1)$ möglich

Heaps als Datenstruktur

Definition: Heap

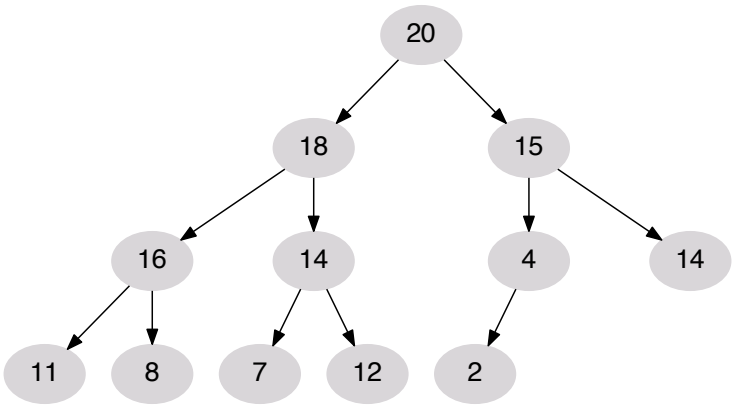
Ein (binärer) Heap ist ein (fast) vollständiger binärer Baum, in dem für jeden Knoten gilt, dass er in einer definierten Ordnungsrelation zu seinen Nachfolger steht.

- Max-Heap: Jeder Knoten ist $\geq$ als seine Nachfolger
- Min-Heap: Jeder Knoten ist $\leq$ als seine Nachfolger

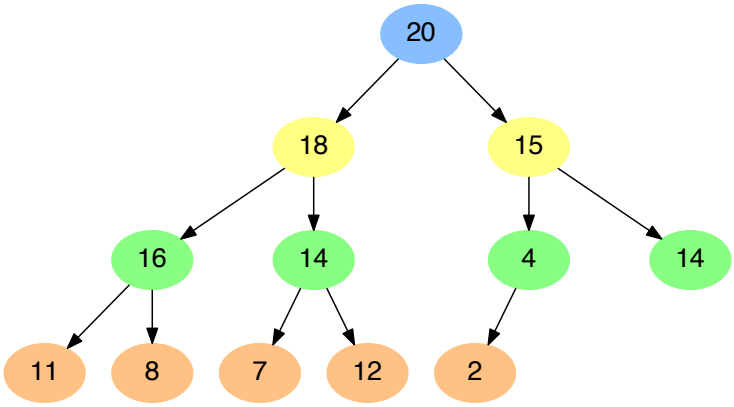
- Fast vollständiger Binärbaum:
  - Alle Ebenen bis auf die unterste sind vollständig
  - Die unterste Ebene ist von von links durchgehend besetzt
- Finden des größten Elements mit $\mathcal{O}(1)$ möglich

Achtung: Die Datenstruktur Heap ist etwas anderes, als der Heap (von dynamischem Speicher), der z.B. von `malloc()` und `free()` verwaltet wird!

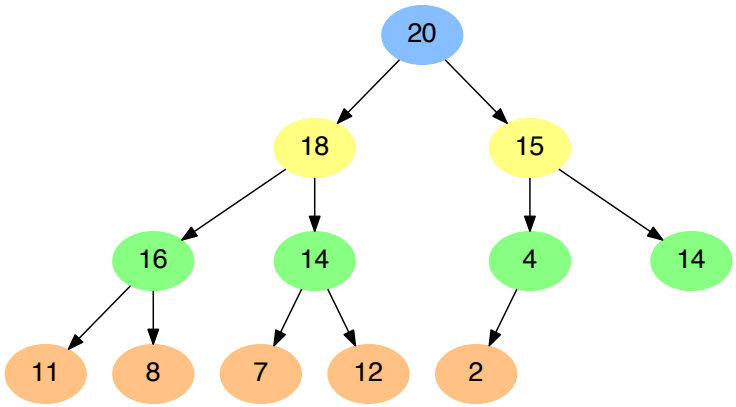
Ein Heap



Ein Heap



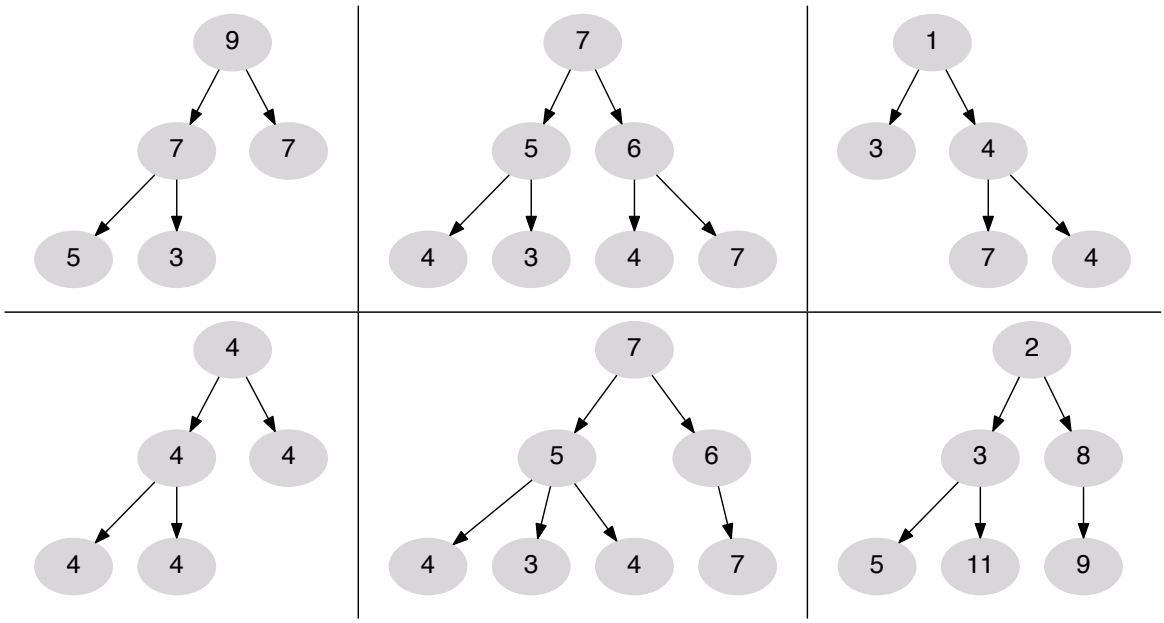
Ein Heap



- Binärbaum:
  - Jeder Knoten hat maximal zwei Nachfolger
- Fast vollständig:
  - Alle Ebenen außer der letzten sind gefüllt
  - Auf der letzten Ebene fehlen Knoten nur rechts
- Max-Heap:
  - Jeder Knoten ist größer oder gleich seinen Nachfolgern

Übung: Charakterisierung von Heaps

Welcher der folgenden Bäume ist ein Max-Heap? Welcher ist ein Min-Heap? Begründen Sie Ihre Entscheidung!



Anwendungen für Heaps

- Priority-Warteschlangen (Queues)
  - Verwaltung von Aufgaben mit Prioritäten
  - Als Selbstzweck oder in anderen Algorithmen
    - CPU scheduling
    - Event handling
    - Auswahl von Klauseln zur Resolution
- Heapsort
  - Effizientes In-Place Sortierverfahren
  - Garantiert $O(n \log n)$

Übung: Heaps bauen

- Betrachten Sie die folgende Menge von Wörtern:
  $W = \{da, bd, ab, aa, b, ac, cb, ba, d, bc, dd\}$
- Bauen Sie 3 verschiedene Max-Heaps für die Wörter aus W.
- Verwenden sie die lexikographische Ordnung auf Wörtern
  (also z.B. $dd > da$, $da > d$, ...)

Übung: Heaps bauen

- Betrachten Sie die folgende Menge von Wörtern:
  $W = \{da, bd, ab, aa, b, ac, cb, ba, d, bc, dd\}$
- Bauen Sie 3 verschiedene Max-Heaps für die Wörter aus W.
- Verwenden sie die lexikographische Ordnung auf Wörtern
  (also z.B. $dd > da$, $da > d$, ...)

Ende Vorlesung 13

Wichtige Operationen auf Heaps

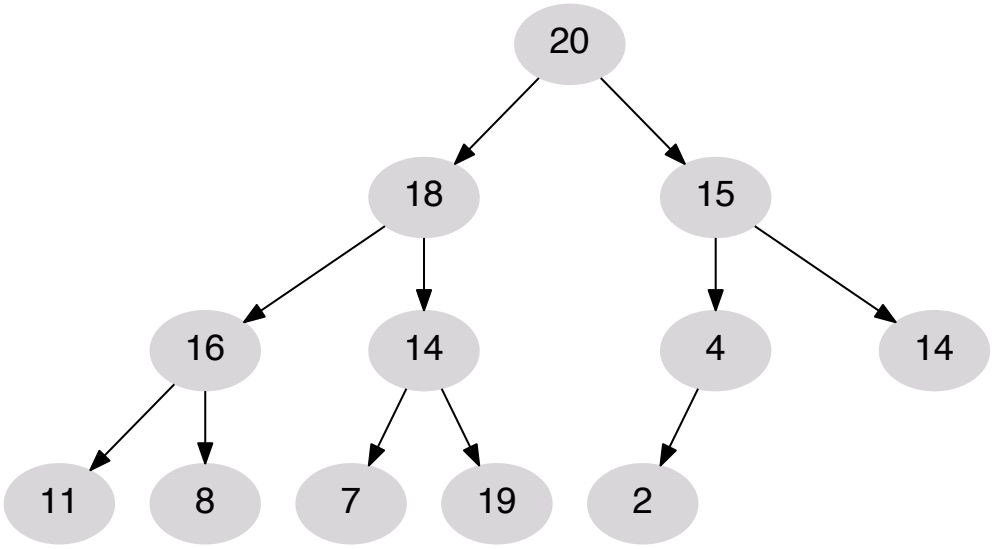
Wir gehen im folgenden immer von Max-Heaps aus. Für Min-Heaps gelten die entsprechenden Aussagen analog!

- `find_max`: Finde das maximale Element eines Heaps
  - Möglich in $\mathcal{O}(1)$
- `heapify`: Stelle die Heap-Eigenschaft eines fast vollständigen Binärbaums her
  - `bubble_up`: Lasse einen großen Knoten nach oben steigen
  - `bubble_down`: Lasse einen kleinen Knoten nach unten sinken
- `extract_max`: Entferne das maximale Element eines Heaps und gib es zurück
  - Möglich in $\mathcal{O}(\log n)$
- `insert`: Füge ein neues Element in den Heap ein
  - Möglich in $\mathcal{O}(\log n)$

Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

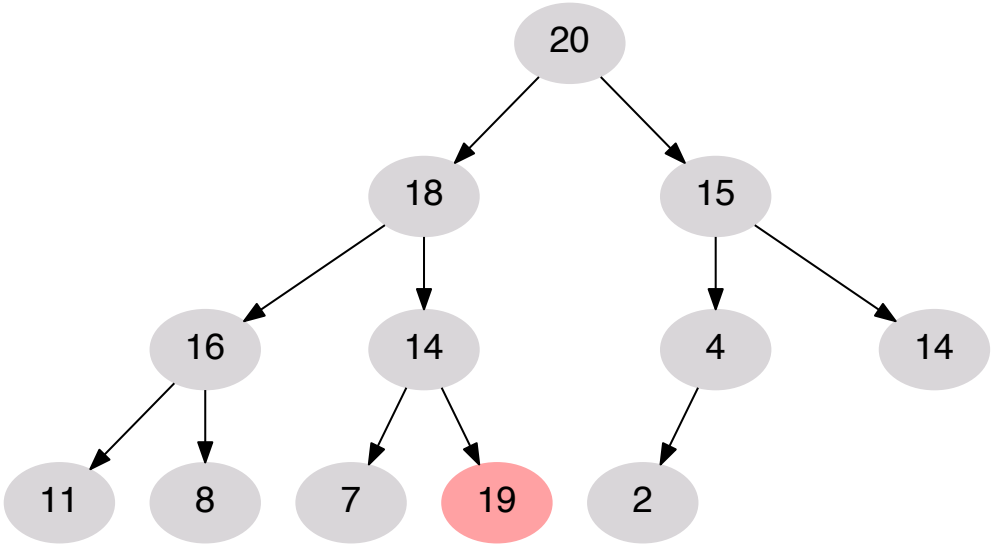
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

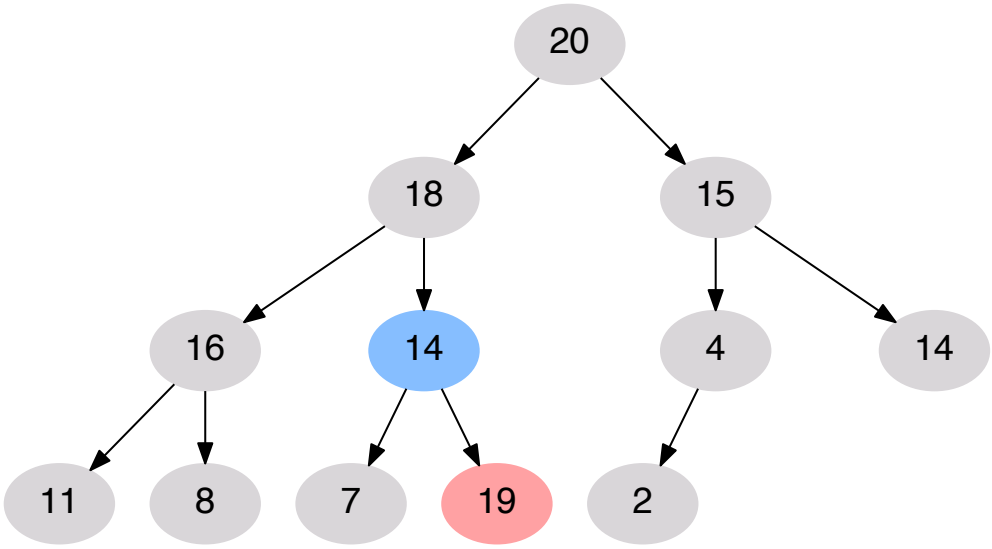
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

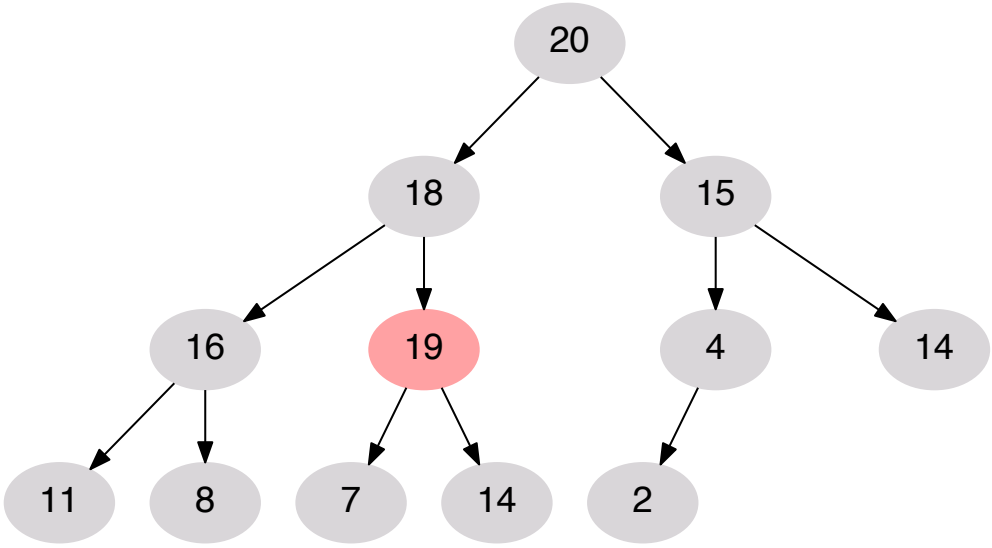
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

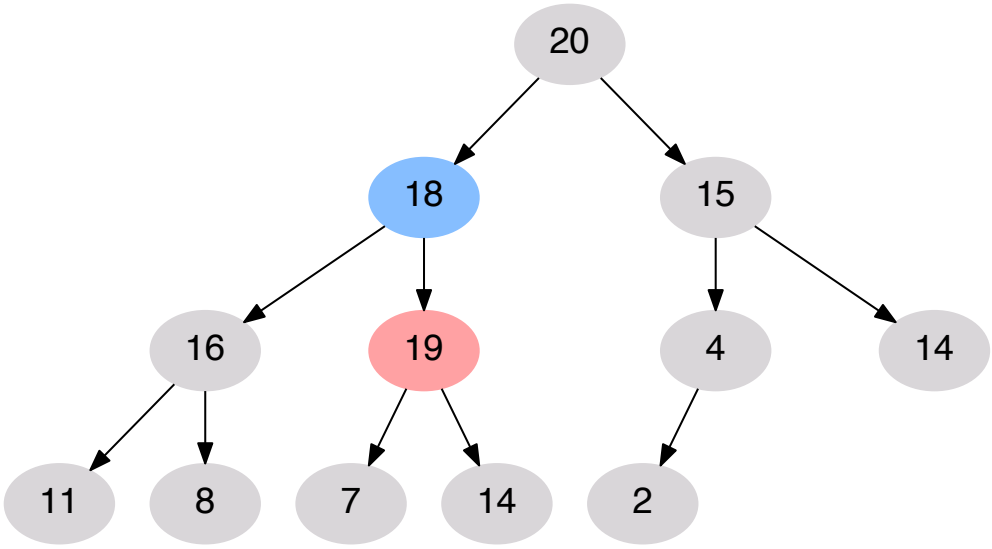
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

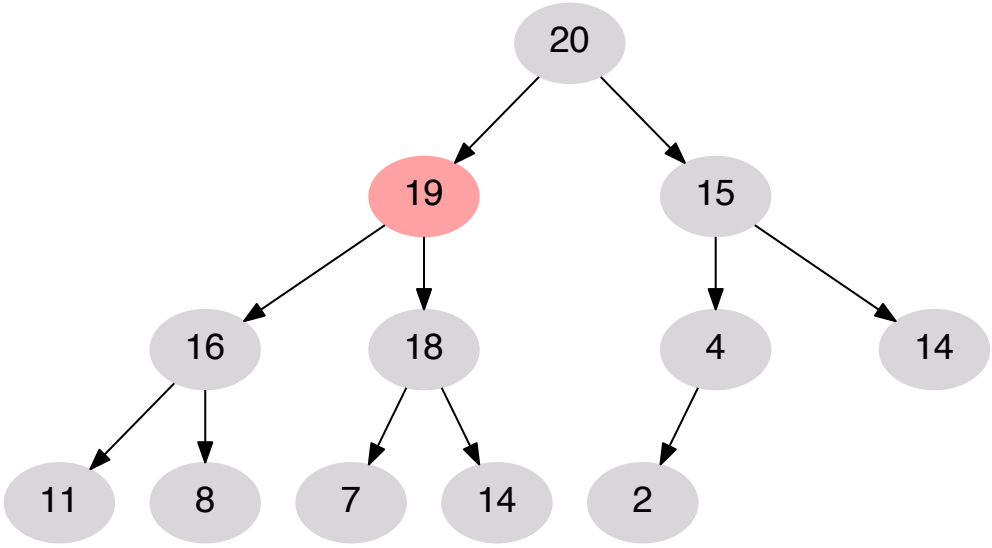
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

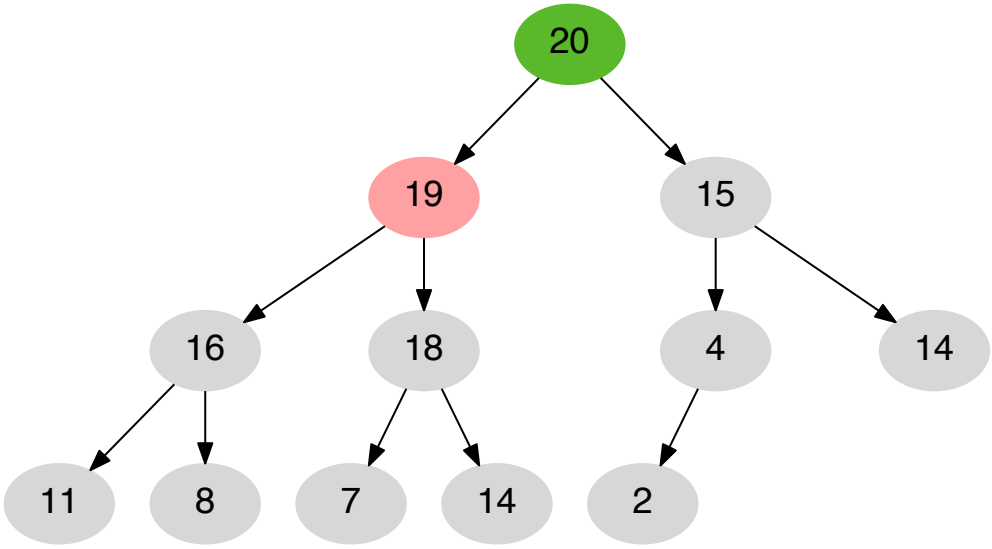
- Idee: Tausche großen Knoten mit Elternteil



Bubble-Up

Repariere Heap-Eigenschaft, wenn ein Knoten zu tief liegt

- Idee: Tausche großen Knoten mit Elternteil



Einfügen in Heaps

- Wir müssen zwei Eigenschaften erhalten:
  - Shape (fast vollständiger Binärbaum)
  - Heap (Kinder sind nie größer als die Eltern)

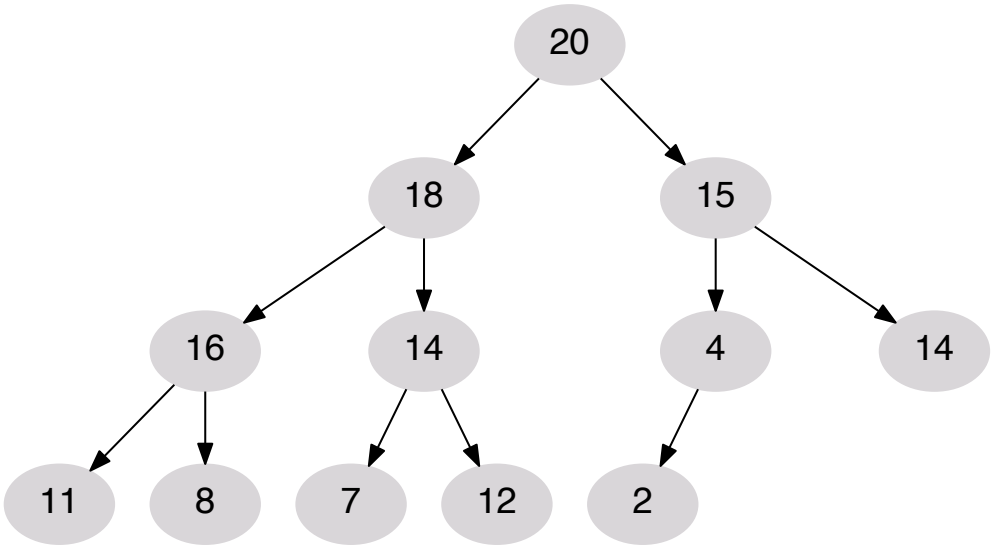
Einfügen in Heaps

- Wir müssen zwei Eigenschaften erhalten:
  - Shape (fast vollständiger Binärbaum)
  - Heap (Kinder sind nie größer als die Eltern)
- Einfügen eines neuen Elements:
  - Füge Element am Ende des Heaps ein (linkester freier Platz auf der untersten Ebene)
    - Damit: Shape-Eigenschaft ist erhalten!
    - Heap-Eigenschaft ist i.A. verletzt
  - Dann: Bubble-up des neuen Elements
    - Dadurch: Wiederherstellung der Heap-Eigenschaft

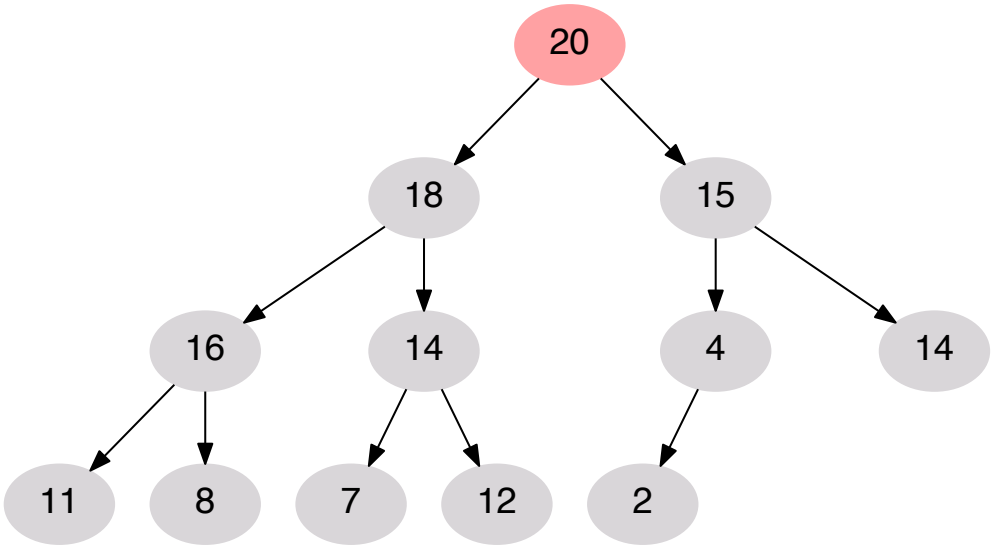
Größtes Element entfernen und Bubble-Down

- Ziel: Größtes Element (Wurzel) aus dem Heap Entfernen
- Idee: Ersetzte größtes Element durch letztes Element
  - Shape-Eigenschaft bleibt erhalten
  - Lasse neue Wurzel nach unten sinken
- Nach-unten-sinken: Bubble-down
  - Wenn Knoten $\geq$ als alle Kinder: Abbruch
  - Sonst: Tausche Knoten mit seinem größten Kind
  - Wiederhole, bis Abbruch (Kinder sind kleiner oder Knoten ist Blatt)

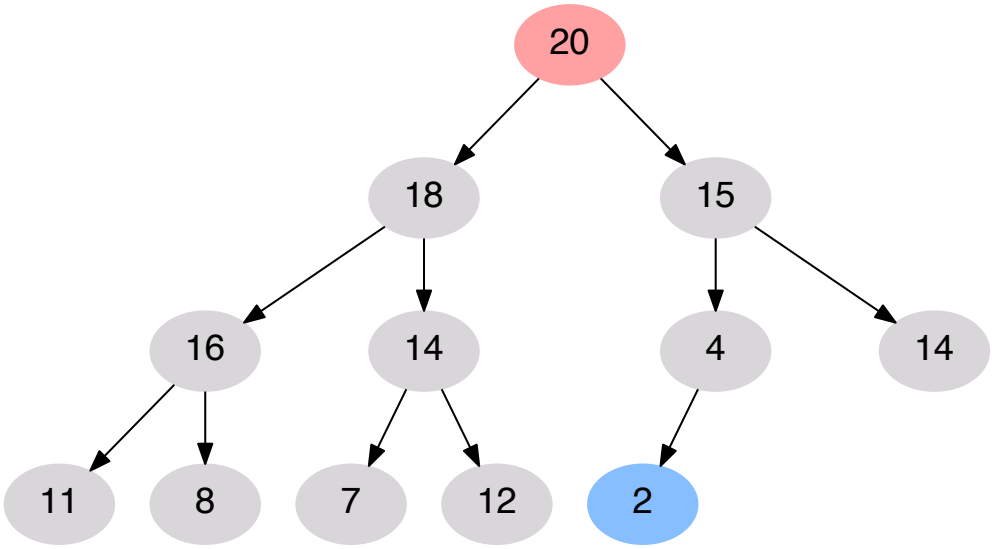
Wurzel extrahieren



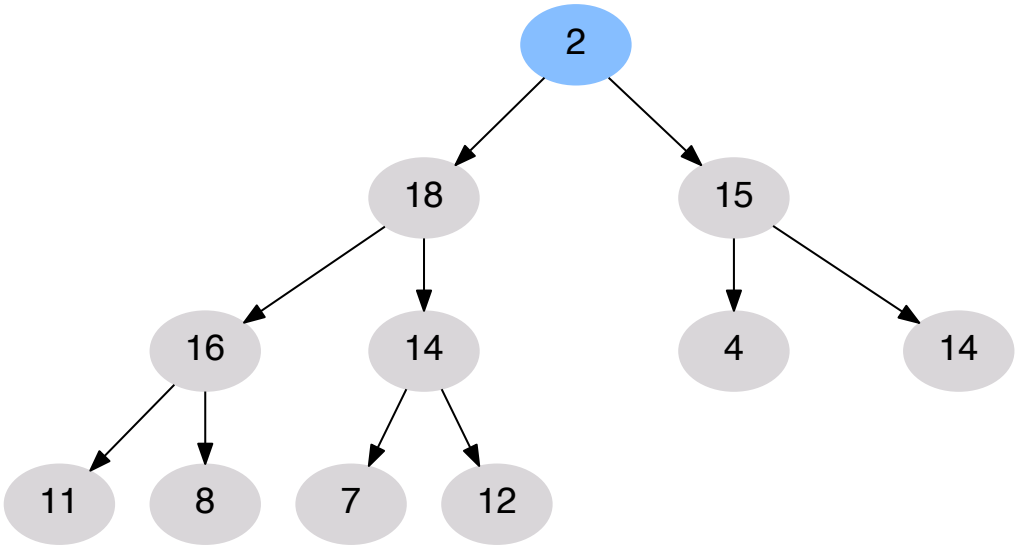
Wurzel extrahieren



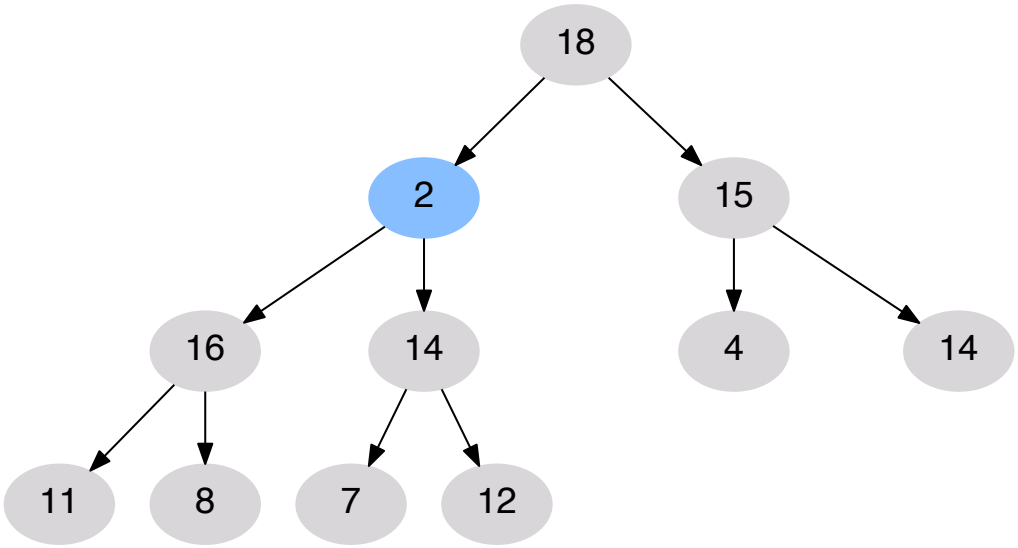
Wurzel extrahieren



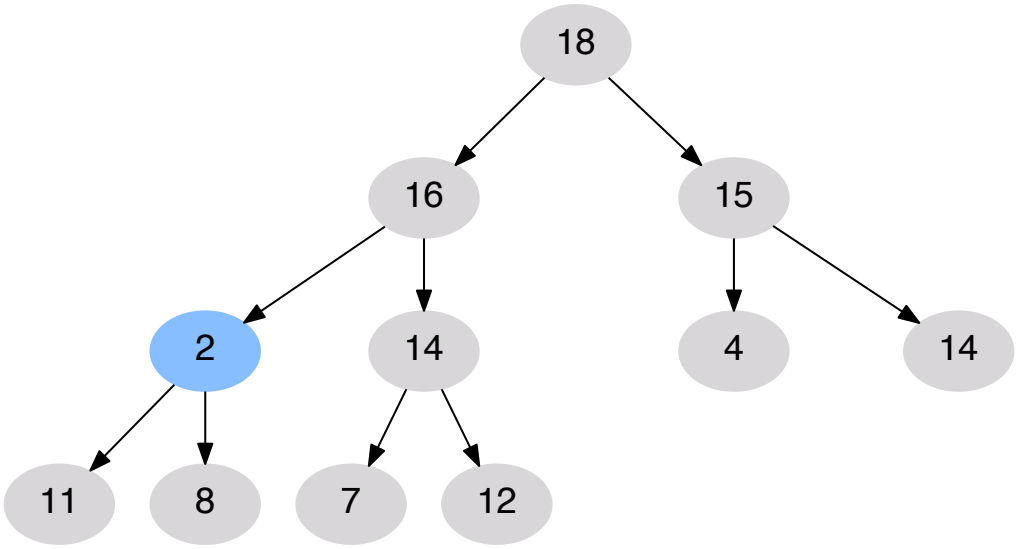
Wurzel extrahieren



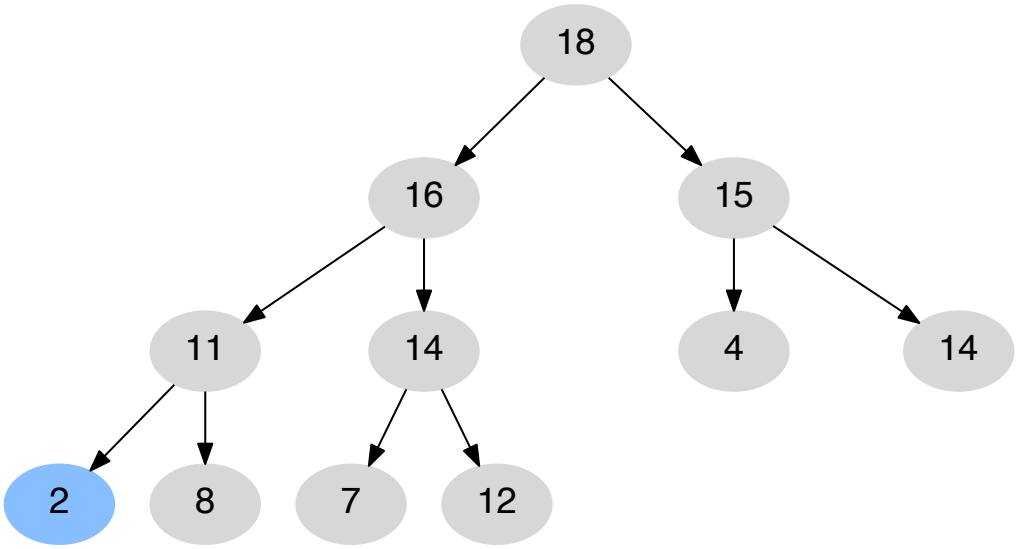
Wurzel extrahieren



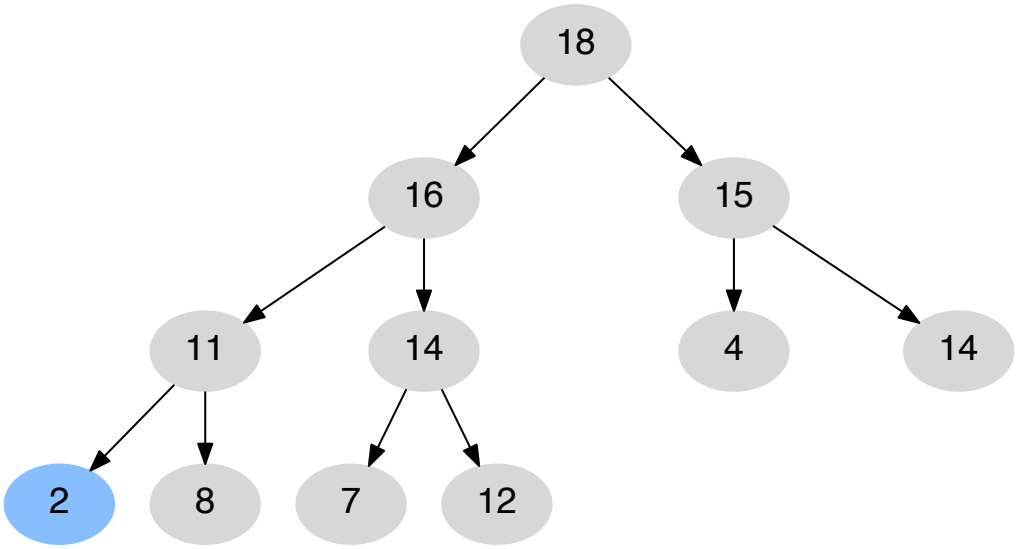
Wurzel extrahieren



Wurzel extrahieren

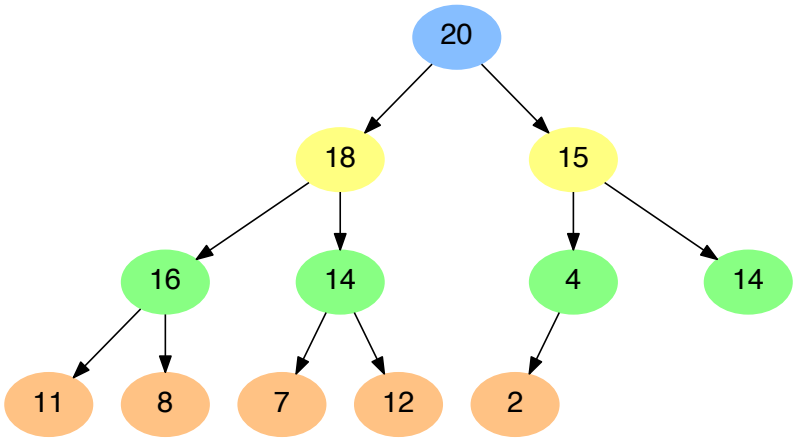


Wurzel extrahieren

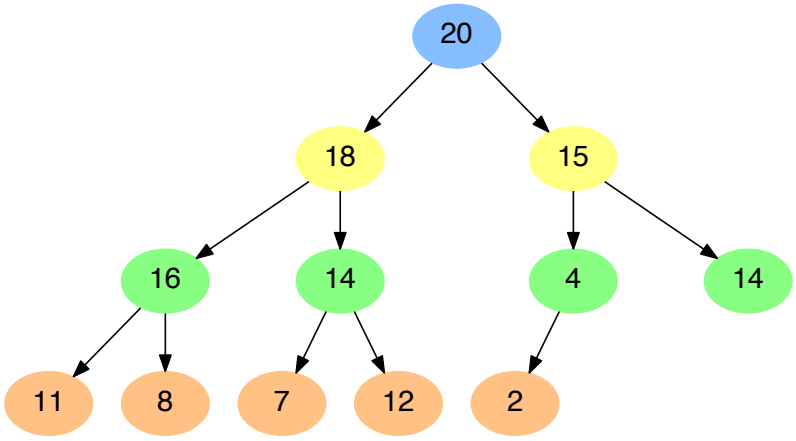


Heap-Eigenschaft wieder hergestellt!

Heaps als Arrays

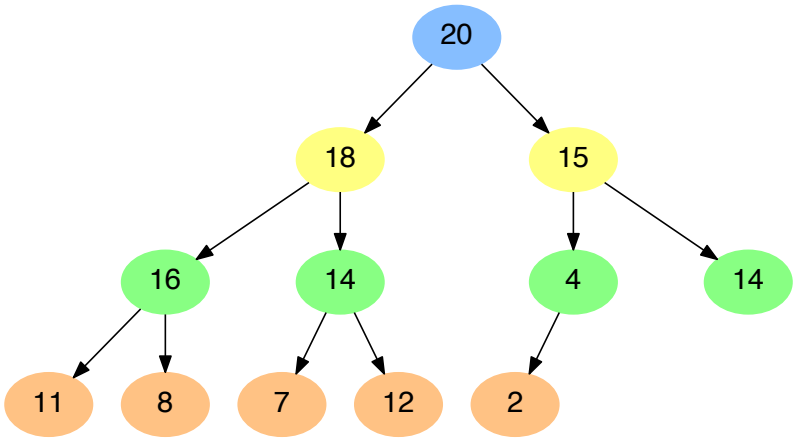


Heaps als Arrays



Idx	Wert	Tiefe
0	20	Ebene 1
1	18	Ebene 2
2	15	
3	16	Ebene 3
4	14	
5	4	
6	14	
7	11	Ebene 4
8	8	
9	7	
10	12	
11	2	
12		
13		
14		

Heaps als Arrays



$$\begin{aligned} \text{lchild}(i) &= 2i+1 \\ \text{rchild}(i) &= 2i+2 \\ \text{parent}(i) &= \left\lfloor \frac{i-1}{2} \right\rfloor \end{aligned}$$

Idx	Wert	Tiefe
0	20	Ebene 1
1	18	Ebene 2
2	15	
3	16	Ebene 3
4	14	
5	4	
6	14	
7	11	Ebene 4
8	8	
9	7	
10	12	
11	2	
12		
13		
14		

Basis-Operationen: Bubble-up

```
def bubble_up(node, arr)
    if node == 0
        # node is root
        return
    parent = (node-1) // 2
    if arr[node] > arr[parent]
        arr[node], arr[parent] = \
          arr[parent], arr[node]
        bubble_up (parent, arr)
```

- Falls `node` größer als `parent(node)`, vertausche `node` und `parent(node)`
- Wiederhole den Schritt, bis er fehlschlägt
  - Fehlschlag: `node` ist Wurzel
  - Fehlschlag: `node` ist $\leq$ seinem Elternknoten

Basis-Operationen: Bubble-down

```
def bubble_down(node,arr,end)
    lci = 2*node+1
    # left child index
    if lci < end
        # node is not a leaf
        gci = lci
        # greater child index
        rci = lci + 1
        if rci < end
            if arr[rci] > arr[lci]
                gci = rci
        if arr[gci] > arr[node]
            arr[node],arr[gci] = \
                arr[gci],arr[node]
            bubble_down(gci, arr, end)
```

- Falls `node` kleiner als eines seiner Kinder ist, vertausche `node` mit dem größten Kind
- Wiederhole den Schritt, bis er fehlschlägt
  - Fehlschlag: `node` ist Blatt
  - Fehlschlag: `node` ist $\geq$ seinen Kindern

Basis-Operationen: Heapify

```
def heapify(arr):
    end = len(arr)
    last = (end - 1) // 2
    # last inner node
    for node in range(last,-1,-1):
        bubble_down(arr,node,end)
```

- lasse jeden inneren Knoten nach unten sinken
- beginne mit unterster Ebene
- Komplexität: $\frac{n}{2} \cdot (\log n - 1) \quad \leadsto \mathcal{O}(n \cdot \log n)$

Übung: Operationen auf Heaps

Gegeben sei die Menge $W = \{da, bd, ab, aa, b, ac, cb, ba, d, bc, dd\}$.

Erzeugen Sie aus W einen Heap:

1. Beginnen Sie mit einem leeren Array, fügen Sie die Elemente der Reihe nach ein und bringen Sie sie mittels `bubble_up` an die richtige Position.
2. Beginnen Sie mit dem unsortierten Array und führen Sie für diesen die Funktion `heapify` durch.

Zählen Sie hierbei jeweils die Vertauschungs-Operationen.

Hinweis: Die Übung wird einfacher, wenn Sie das Array bereits in Form eines fast vollständigen Binärbaums aufschreiben!

Heapsort: Prinzip

Eingabe: Zu sortierende Folge als Array A

Heapsort: Prinzip

Eingabe: Zu sortierende Folge als Array A

1. Transformiere A in Max-Heap (`heapify`)
2. Vertausche Wurzel (a_0) mit letztem Element des Arrays (Heap endet jetzt mit vorletztem Element)
3. Lasse neue Wurzel herabsinken (`bubble_down`)
4. Vertausche neue Wurzel mit vorletztem Element (Heap endet jetzt mit drittletztem Element)
5. Lasse neue Wurzel herabsinken
6. ...

Heapsort: Prinzip

Eingabe: Zu sortierende Folge als Array A

1. Transformiere A in Max-Heap (`heapify`)
2. Vertausche Wurzel (a_0) mit letztem Element des Arrays (Heap endet jetzt mit vorletztem Element)
3. Lasse neue Wurzel herabsinken (`bubble_down`)
4. Vertausche neue Wurzel mit vorletztem Element (Heap endet jetzt mit drittletztem Element)
5. Lasse neue Wurzel herabsinken
6. ...

Im Prinzip ist Heapsort ein *Selection Sort*, bei dem immer das größte Element effizient selektiert wird.

Heapsort: Code

```
def heapsort(arr):
    last = len(arr)-1
    heapify(arr)
    for i in range(last,0,-1):
        arr[0],arr[i] = \
          arr[i],arr[0]
        bubble_down(arr,0, i)
    return arr
```

- `arr`: zu sortierendes Array
- `i`: Grenze zwischen Heap und sortiertem Array

Warum ist Heapsort effizient?

- Entfernen des größten Elements: $\mathcal{O}(1)$
- Wiederherstellung der Heap-Eigenschaft:
  - Vergleiche erfolgen nur entlang eines Pfades Wurzel $\rightarrow$ Blatt
  - Maximale Pfadlänge: $\mathcal{O}(\log n)$
  - Elemente auf verschiedenen Pfaden werden nicht verglichen

Heapsort: Vor- und Nachteile

Vorteile

- In-place
- $\mathcal{O}(n \log n)$ auch im worst case

Heapsort: Vor- und Nachteile

Vorteile

- In-place
- $\mathcal{O}(n \log n)$ auch im worst case

Nachteile

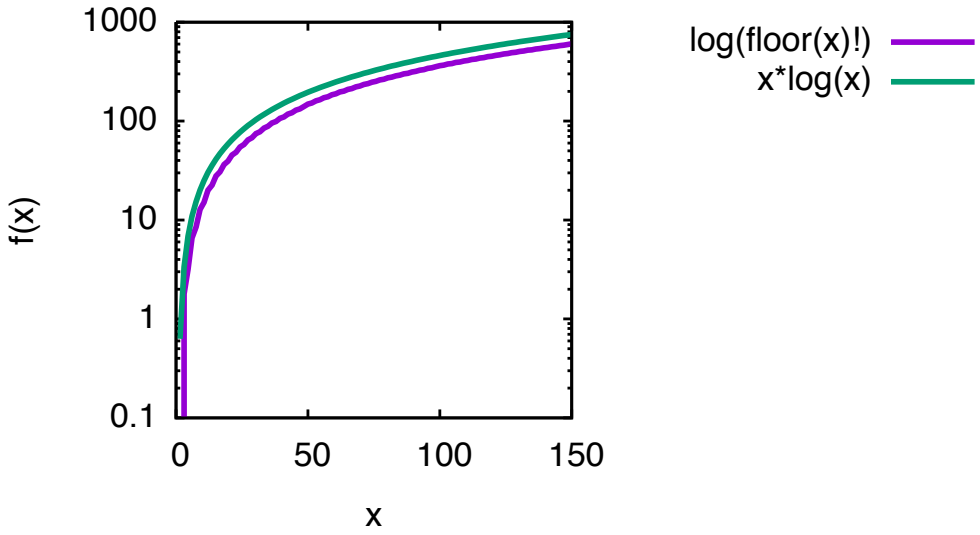
- $2 \cdot n \log n$ Schritte:
  - 1-mal `heapify` $\leadsto$ $n \log n$
  - n-mal `bubble_down` $\leadsto$ $n \log n$
- funktioniert nur auf Arrays gut
- instabil

Heapsort: Zusammenfassung

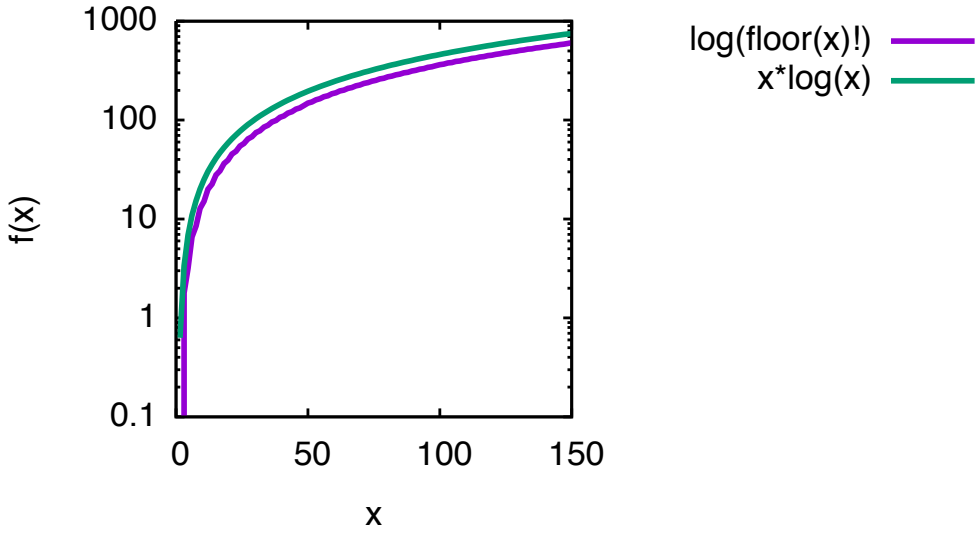
- Betrachte Array als Max-Heap
- Entferne sukzessive größtes Element
- Stelle danach Heap-Eigenschaft wieder her ($\mathcal{O}(\log n)$)
- in-place
- instabil
- $\mathcal{O}(n \log n)$
- erfordert Arrays

Sortieren – Abschluss

Mathematischer Fakt



Mathematischer Fakt



$$\log(n!) \in \Theta(n \log n)$$

Folgerung aus Stirling'scher Formel (Cormen, Leiserson, Rivest)

Sortieren: Untere Schranke für Komplexität

- alle effizienten Sortieralgorithmen haben Komplexität $\mathcal{O}(n \log n)$
- geht es noch besser, oder ist Sortieren $\Theta(n \log n)$?
- Offensichtliche untere Schranke für Vergleiche: $\lceil \frac{n}{2} \rceil$
  (sonst würde ein Element nicht verglichen)
- Kann man $\mathcal{O}(n)$ erreichen?
  - Eingabe hat n Elemente
  - Ausgabe ist Permutation der Eingabe: $n!$ Möglichkeiten
    (jede Permutation kann die richtige sein)
  - Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - m Vergleiche $\quad\rightsquigarrow\quad$ 2^m mögliche Ausgaben
  - $2^m \geq n!$

Sortieren: Untere Schranke für Komplexität

- alle effizienten Sortieralgorithmen haben Komplexität $\mathcal{O}(n \log n)$
- geht es noch besser, oder ist Sortieren $\Theta(n \log n)$?
- Offensichtliche untere Schranke für Vergleiche: $\lceil \frac{n}{2} \rceil$
  (sonst würde ein Element nicht verglichen)
- Kann man $\mathcal{O}(n)$ erreichen?
  - Eingabe hat n Elemente
  - Ausgabe ist Permutation der Eingabe: $n!$ Möglichkeiten
    (jede Permutation kann die richtige sein)
  - Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - m Vergleiche $\quad\rightsquigarrow\quad$ 2^m mögliche Ausgaben
  - $2^m \geq n! \quad\Rightarrow\quad m \geq \log(n!)$

Sortieren: Untere Schranke für Komplexität

- alle effizienten Sortieralgorithmen haben Komplexität $\mathcal{O}(n \log n)$
- geht es noch besser, oder ist Sortieren $\Theta(n \log n)$?
- Offensichtliche untere Schranke für Vergleiche: $\lceil \frac{n}{2} \rceil$
  (sonst würde ein Element nicht verglichen)
- Kann man $\mathcal{O}(n)$ erreichen?
  - Eingabe hat n Elemente
  - Ausgabe ist Permutation der Eingabe: $n!$ Möglichkeiten
    (jede Permutation kann die richtige sein)
  - Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - m Vergleiche $\quad\rightsquigarrow\quad$ 2^m mögliche Ausgaben
  - $2^m \geq n! \quad\Rightarrow\quad m \geq \log(n!) \quad\Leftrightarrow\quad m \geq n \log n$ (Stirling)

Sortieren: Untere Schranke für Komplexität

- alle effizienten Sortieralgorithmen haben Komplexität $\mathcal{O}(n \log n)$
- geht es noch besser, oder ist Sortieren $\Theta(n \log n)$?
- Offensichtliche untere Schranke für Vergleiche: $\lceil \frac{n}{2} \rceil$
  (sonst würde ein Element nicht verglichen)
- Kann man $\mathcal{O}(n)$ erreichen?
  - Eingabe hat n Elemente
  - Ausgabe ist Permutation der Eingabe: $n!$ Möglichkeiten
    (jede Permutation kann die richtige sein)
  - Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - m Vergleiche $\quad\rightsquigarrow\quad$ 2^m mögliche Ausgaben
  - $2^m \geq n! \quad\Rightarrow\quad m \geq \log(n!) \quad\Leftrightarrow\quad m \geq n \log n$ (Stirling)
- mindestens $n \log n$ Vergleiche für Folge der Länge n nötig!

Sortieren: Untere Schranke für Komplexität

- alle effizienten Sortieralgorithmen haben Komplexität $\mathcal{O}(n \log n)$
- geht es noch besser, oder ist Sortieren $\Theta(n \log n)$?
- Offensichtliche untere Schranke für Vergleiche: $\lceil \frac{n}{2} \rceil$
  (sonst würde ein Element nicht verglichen)
- Kann man $\mathcal{O}(n)$ erreichen?
  - Eingabe hat n Elemente
  - Ausgabe ist Permutation der Eingabe: $n!$ Möglichkeiten
    (jede Permutation kann die richtige sein)
  - Unterschiedliche Ausgaben resultieren aus Ergebnis der Vergleiche
  - m Vergleiche $\quad\rightsquigarrow\quad$ 2^m mögliche Ausgaben
  - $2^m \geq n! \quad\Rightarrow\quad m \geq \log(n!) \quad\Leftrightarrow\quad m \geq n \log n$ (Stirling)
- mindestens $n \log n$ Vergleiche für Folge der Länge n nötig!

Sortieren einer Folge der Länge n ist bestenfalls $\Theta(n \log n)$.

Sortieren: Zusammenfassung

Einfache Verfahren

- vergleichen jedes Paar von Elementen
- bearbeiten in jedem Durchlauf nur ein Element
- verbessern nicht die Position der anderen Elemente
- $\mathcal{O}(n^2)$

Sortieren: Zusammenfassung

Einfache Verfahren

- vergleichen jedes Paar von Elementen
- bearbeiten in jedem Durchlauf nur ein Element
- verbessern nicht die Position der anderen Elemente
- $\mathcal{O}(n^2)$

Effiziente Verfahren

- Unterschiedliche Ansätze:
  - erst grob, dann fein: Quicksort
  - erst im Kleinen, dann im Großen: Mergesort
  - spezielle Datenstruktur: Heapsort
- Effizienzgewinn durch
  - Vermeidung unnötiger Vergleiche
  - effiziente Nutzung der in einem Durchlauf gesammelten Information
  - Verbessern der Position mehrerer Elemente in einem Durchlauf
- $\mathcal{O}(n \log n)$ *(as good as it gets)*

Ende Vorlesung 14

Logarithmen

Logarithmus: Grundideen

Definition: Logarithmus

Seien $x, b \in \mathbb{R}^{\geq 0}$ positive reele Zahlen, sei $b \neq 1$ und sei $x = b^y$. Dann heißt y der Logarithmus von x zur Basis b.

- Wir schreiben: $y = \log_b(x)$.

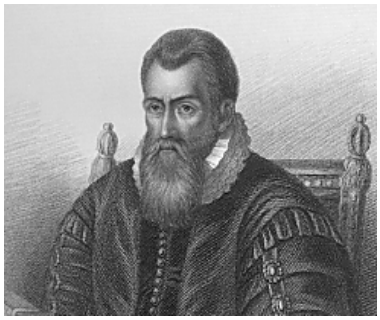
Logarithmus: Grundideen

Definition: Logarithmus

Seien $x, b \in \mathbb{R}^{\geq 0}$ positive reele Zahlen, sei $b \neq 1$ und sei $x = b^y$. Dann heißt y der Logarithmus von x zur Basis b.

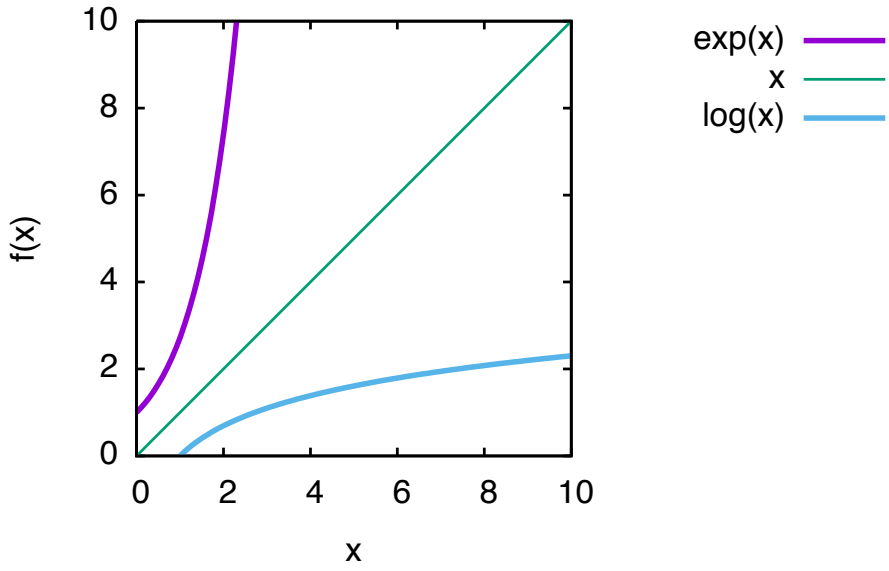
- Wir schreiben: $y = \log_b(x)$.

- Der Logarithmus ist die Umkehrfunktion zur Potenzierung
  - Also: $b^{\log_b(x)} = \log_b(b^x) = x$
  - Die Graphen von b^x und $log_b(x)$ gehen durch Spiegelung an der Diagonalen ineinander über
- Die Klammern lassen wir oft weg:
  - $y = log_b x$

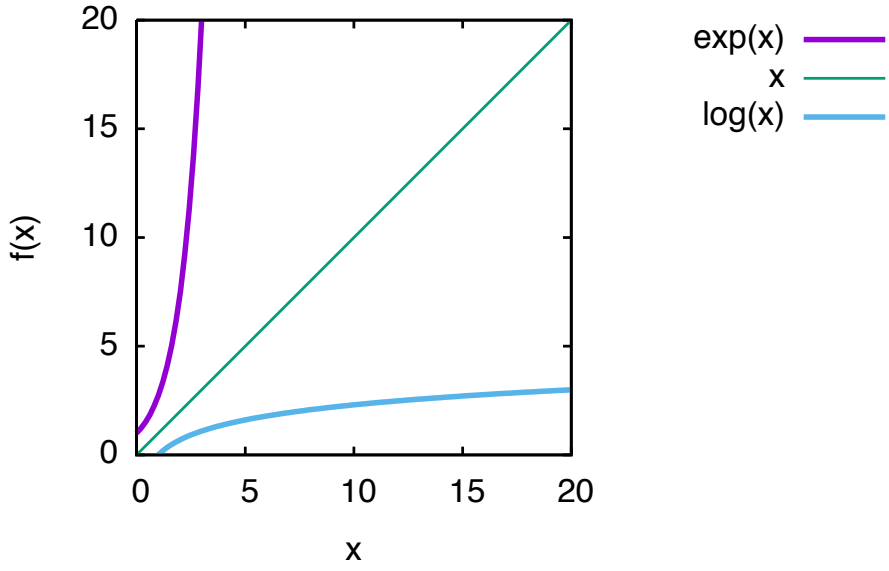


John Napier, 1550-1617, Erfinder des Begriffs Logarithmus

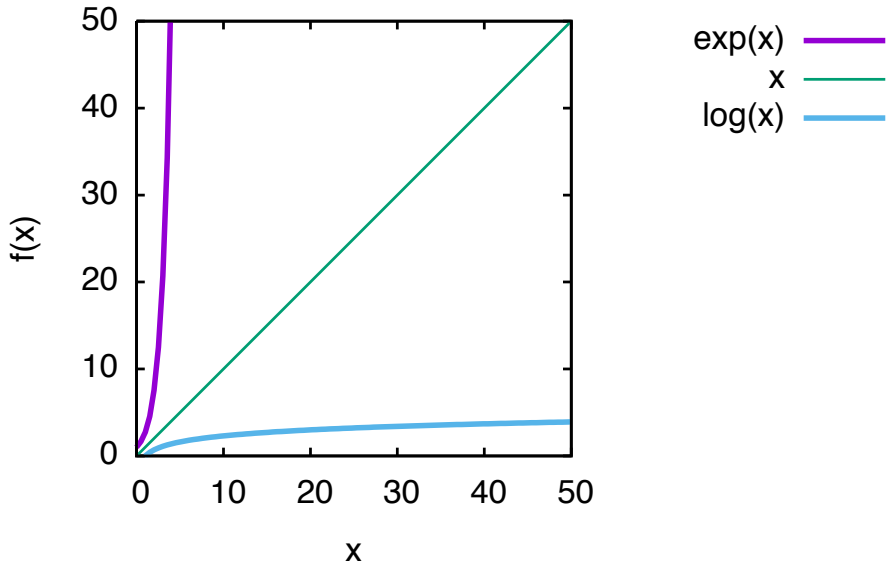
Logarithmus illustriert



Logarithmus illustriert



Logarithmus illustriert



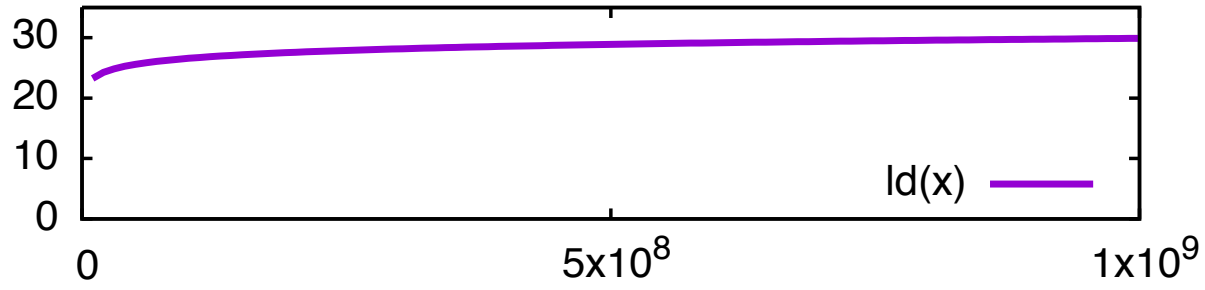
Logarithmus: Basen

- Wichtige Basen b
  - 1 ist als Basis verboten (warum?)
  - 2 ist die für Informatiker wichtigste Basis (warum?)
  - $e \approx 2.71828\ldots$ hat die Eigenschaft, dass $\frac{d}{dx}e^x = (e^x)' = e^x$ gilt
    - Also: Beim Ableiten kommt "das Selbe" raus!
    - Die Funktion $\log_e x$ heißt auch *natürlicher Logarithmus* und wird $\ln(x)$ geschrieben
  - 10 ist auf den meisten Taschenrechnern der Default
- Wir können verschiedene Basen ineinander überführen

$$\log_a x = \frac{\log_b x}{\log_b a}$$

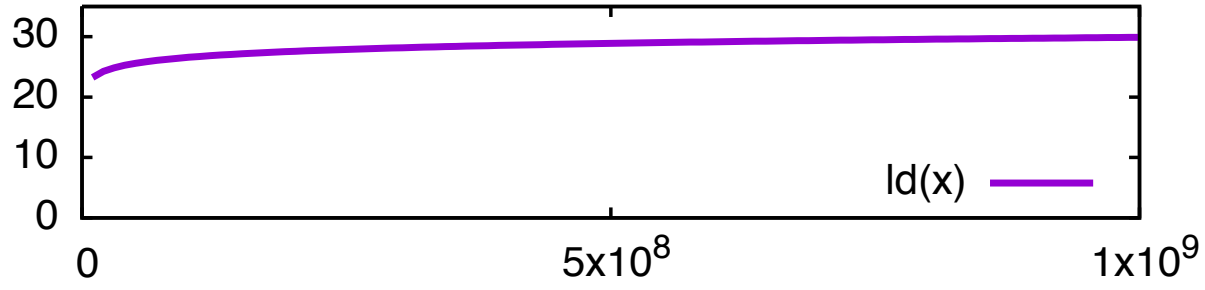
  - Also: $\log_a x = \frac{1}{\log_b a} \log_b x = c \log_b x$
  - Damit: $\mathcal{O}(\log_a x) = \mathcal{O}(c \log_b x) = \mathcal{O}(log_b x)$
    Die Basis ist für $\mathcal{O}$ irrelevant!

Logarithmen: Wachstum



- “Für große n ist $\log n$ annähernd konstant”
  - $\log_2 1024 = 10$ und $\log_2 2048 = 11$
  - $\log_2 1048576 = 20$ und $\log_2 2097152 = 21$

Logarithmen: Wachstum



- “Für große n ist $\log n$ annähernd konstant”
  - $\log_2 1024 = 10$ und $\log_2 2048 = 11$
  - $\log_2 1048576 = 20$ und $\log_2 2097152 = 21$

Algorithmen mit logarithmischer Laufzeit sind in der Regel sehr effizient und skalieren auf sehr große Eingaben!

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$

165

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$
- Wie oft kann ich eine Menge M in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$
- Wie oft kann ich eine Menge M in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?
  - Antwort: $\log_2 |M|$
  - Beispiel: Binäre Suche!

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$
- Wie oft kann ich eine Menge M in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?
  - Antwort: $\log_2 |M|$
  - Beispiel: Binäre Suche!
- Wie hoch ist ein vollständiger Binärbaum mit $n = 2^k - 1$ Knoten? Mit anderen Worten: Wie lang ist der längste Ast eines solchen Baums?

Logarithmen: Anwendungen

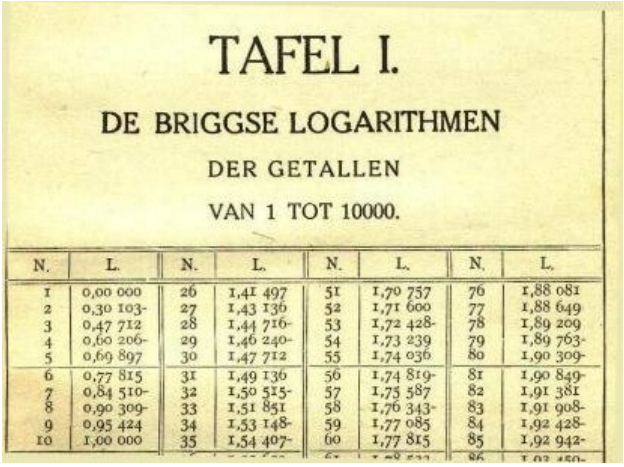
- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$
- Wie oft kann ich eine Menge M in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?
  - Antwort: $\log_2 |M|$
  - Beispiel: Binäre Suche!
- Wie hoch ist ein vollständiger Binärbaum mit $n = 2^k - 1$ Knoten? Mit anderen Worten: Wie lang ist der längste Ast eines solchen Baums?
  - Antwort: $\log_2(n+1)$

Logarithmen: Anwendungen

- Wie oft kann ich eine Zahl x halbieren, bis sie 1 oder kleiner wird?
  - Antwort: $\log_2 x$
- Wie oft kann ich eine Menge M in zwei fast gleichmächtige Teilmengen (+/- 1 Element) teilen, bis jede höchstens ein Element hat?
  - Antwort: $\log_2 |M|$
  - Beispiel: Binäre Suche!
- Wie hoch ist ein vollständiger Binärbaum mit $n = 2^k - 1$ Knoten? Mit anderen Worten: Wie lang ist der längste Ast eines solchen Baums?
  - Antwort: $\log_2(n+1) \approx \log_2 n$
  - Beispiel: Heap

Logarithmen: Ausgewählte Rechenregeln

- Traditionell wichtig: $\log_b(x \cdot y) = \log_b x + \log_b y$
  - Begründung: $b^x \cdot b^y = b^{x+y}$
  - Anwendung: Kopfrechnen mit großen Zahlen
  - Dafür: *Logarithmentafeln*



TAFEL I.

DE BRIGGSE LOGARITHMEN

DER GETALLEN

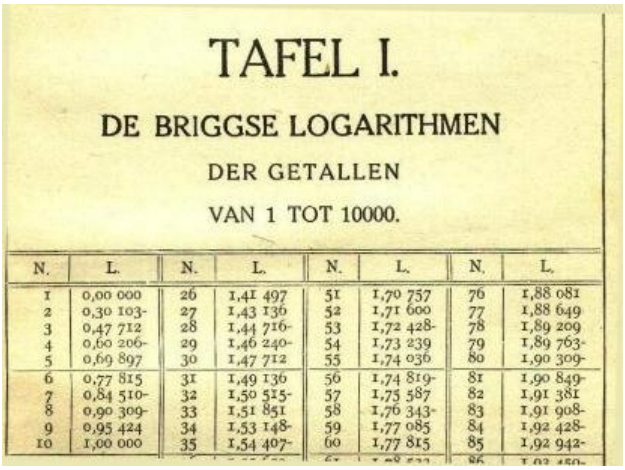
VAN 1 TOT 10000.

N.	L.	N.	L.	N.	L.	N.	L.
1	0,00 000	26	1,41 497	51	1,70 757	76	1,88 081
2	0,30 103-	27	1,43 136	52	1,71 600	77	1,88 649
3	0,47 712	28	1,44 716-	53	1,72 428-	78	1,89 209
4	0,60 206-	29	1,46 240-	54	1,73 239	79	1,89 763-
5	0,69 897	30	1,47 712	55	1,74 036	80	1,90 309-
6	0,77 815	31	1,49 136	56	1,74 819-	81	1,90 849-
7	0,84 510-	32	1,50 515-	57	1,75 587	82	1,91 381
8	0,90 309-	33	1,51 851	58	1,76 343-	83	1,91 908-
9	0,95 424	34	1,53 148-	59	1,77 085	84	1,92 428-
10	1,00 000	35	1,54 407-	60	1,77 815	85	1,92 942-

Logarithmentafeln gibt es seit 1588 (zeitgleich schickte Phillip II die spanische Armada gegen Elisabeth I)

Logarithmen: Ausgewählte Rechenregeln

- Traditionell wichtig:
  $\log_b(x \cdot y) = \log_b x + \log_b y$
  - Begründung: $b^x \cdot b^y = b^{x+y}$
  - Anwendung: Kopfrechnen mit großen Zahlen
  - Dafür: *Logarithmentafeln*
  - Mini-Übung: Berechnen Sie $2 \cdot 26$ und $81/3$ mit Hilfe der Logarithmentafel



TAFEL I.

DE BRIGGSE LOGARITHMEN

DER GETALLEN

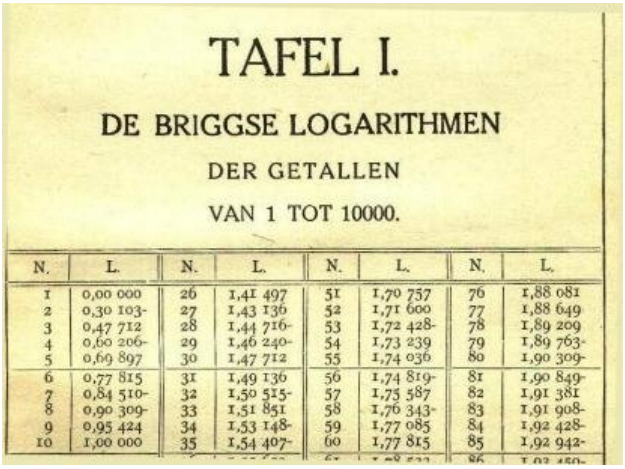
VAN 1 TOT 10000.

N.	L.	N.	L.	N.	L.	N.	L.
1	0,00 000	26	1,41 497	51	1,70 757	76	1,88 081
2	0,30 103-	27	1,43 136	52	1,71 600	77	1,88 649
3	0,47 712	28	1,44 716-	53	1,72 428-	78	1,89 209
4	0,60 206-	29	1,46 240-	54	1,73 239	79	1,89 763-
5	0,69 897	30	1,47 712	55	1,74 036	80	1,90 309-
6	0,77 815	31	1,49 136	56	1,74 819-	81	1,90 849-
7	0,84 510-	32	1,50 515-	57	1,75 587	82	1,91 381
8	0,90 309-	33	1,51 851	58	1,76 343-	83	1,91 908-
9	0,95 424	34	1,53 148-	59	1,77 085	84	1,92 428-
10	1,00 000	35	1,54 407-	60	1,77 815	85	1,92 942-

Logarithmentafeln gibt es seit 1588 (zeitgleich schickte Phillip II die spanische Armada gegen Elisabeth I)

Logarithmen: Ausgewählte Rechenregeln

- Traditionell wichtig:
  $\log_b(x \cdot y) = \log_b x + \log_b y$
  - Begründung: $b^x \cdot b^y = b^{x+y}$
  - Anwendung: Kopfrechnen mit großen Zahlen
  - Dafür: *Logarithmentafeln*
  - Mini-Übung: Berechnen Sie $2 \cdot 26$ und $81/3$ mit Hilfe der Logarithmentafel
- $\log_b x^r = r \cdot \log_b x$
  - Iterierter Anwendung der vorherigen Regel



TAFEL I.

DE BRIGGSE LOGARITHMEN

DER GETALLEN

VAN 1 TOT 10000.

N.	L.	N.	L.	N.	L.	N.	L.
1	0,00 000	26	1,41 497	51	1,70 757	76	1,88 081
2	0,30 103-	27	1,43 136	52	1,71 600	77	1,88 649
3	0,47 712	28	1,44 716-	53	1,72 428-	78	1,89 209
4	0,60 206-	29	1,46 240-	54	1,73 239	79	1,89 763-
5	0,69 897	30	1,47 712	55	1,74 036	80	1,90 309-
6	0,77 815	31	1,49 136	56	1,74 819-	81	1,90 849-
7	0,84 510-	32	1,50 515-	57	1,75 587	82	1,91 381
8	0,90 309-	33	1,51 851	58	1,76 343-	83	1,91 908-
9	0,95 424	34	1,53 148-	59	1,77 085	84	1,92 428-
10	1,00 000	35	1,54 407-	60	1,77 815	85	1,92 942-

Logarithmentafeln gibt es seit 1588 (zeitgleich schickte Phillip II die spanische Armada gegen Elisabeth I)

Logarithmen: Übung

- Kopfberechnen Sie:
  - $\log_2 1$
  - $\log_2 1000$ (in etwa)
  - $\log_2 2000000$ (in etwa)
- Bestimmen Sie
  - $\log_8 2$
  - $\log_2 \frac{1}{8}$
  - $\log_8 x$ zur Basis 2

Definition: Tiefe eines Baums

Die Tiefe eines Baumes ist die Anzahl der Knoten auf seinem längsten Ast.
Formal:

- Die Tiefe des leeren Baums ist 0.
- Die Tiefe eines nichtleeren Baums ist $1 + max(\text{Tiefe der Kinder der Wurzel})$.

- Zeigen Sie: Ein Binärbaum mit n Knoten hat mindestens Tiefe $\lceil \log_2(n+1) \rceil$

Schlüssel und Werte

Dictionaries

- Ziel: Dynamische Assoziation von Schlüsseln und Werten
  - Symboltabellen
  - Dictionaries
  - Look-up tables
  - Assoziative Arrays
  - ...
- Prinzip: Speichere Paare von Schlüsseln und Werten
  - Z.B. Wort und Liste von Webseiten, auf denen es vorkommt
  - Z.B. Matrikelnummer und Stammdatensatz
  - Z.B. Datum und Umsatz
  - Z.B. KFZ-Kennzeichen und Wagenhalter
  - Z.B. Name und Mitarbeiterstammdaten
  - Häufig: Schlüssel (String) und Pointer auf beliebige Daten
  - ...

Operationen auf Dictionaries

- Wichtige Operationen auf Dictionaries
  - Anlegen eine leeren Dictionaries
  - Einfügen von Schlüssel/Wert-Paaren
  - Löschen von Schlüssel/Wert-Paaren (normalerweise anhand eines Schlüssels)
  - Finden von Schlüssel/Wert-Paaren anhand eines Schlüssels
  - Geordnete Ausgabe aller Schlüssel/Wert Paare
- Zu klärende Frage: Mehrfachschlüssel

Operationen auf Dictionaries

- Wichtige Operationen auf Dictionaries
  - Anlegen eine leeren Dictionaries
  - Einfügen von Schlüssel/Wert-Paaren
  - Löschen von Schlüssel/Wert-Paaren (normalerweise anhand eines Schlüssels)
  - Finden von Schlüssel/Wert-Paaren anhand eines Schlüssels
  - Geordnete Ausgabe aller Schlüssel/Wert Paare
- Zu klärende Frage: Mehrfachschlüssel
  - Mehrfachschlüssel verbieten
  - Ein Schlüssel, Liste/Menge von Werten
  - Mehrfachschlüssel erlauben

Übung: Dictionaries

- Wie können Sie den *abstrakten Datentyp* Dictionary mit den bekannten Datenstrukturen realisieren?
  - Betrachten Sie verschiedene Optionen!
- Welchen Komplexitäten haben die Kernoperationen?
  - Nehmen Sie an, dass das Dictionary n Schlüssel enthält

Übung: Dictionaries

- Wie können Sie den *abstrakten Datentyp* Dictionary mit den bekannten Datenstrukturen realisieren?
  - Betrachten Sie verschiedene Optionen!
- Welchen Komplexitäten haben die Kernoperationen?
  - Nehmen Sie an, dass das Dictionary n Schlüssel enthält

Ende Vorlesung 15

Binäre Suchbäume

Binäre Suchbäume

Definition: Binärer Suchbaum

Eine binärer Suchbaum ist ein Binärbaum mit folgenden Eigenschaften:

- Die Knoten des Baums sind mit Schlüsseln aus einer geordneten Menge K beschriftet
- Für jeden Knoten N gilt:
  - Alle Schlüssel im linken Teilbaum von N sind kleiner als der Schlüssel von N
  - Alle Schlüssel im rechten Teilbaum von N sind größer als der Schlüssel von N

Binäre Suchbäume

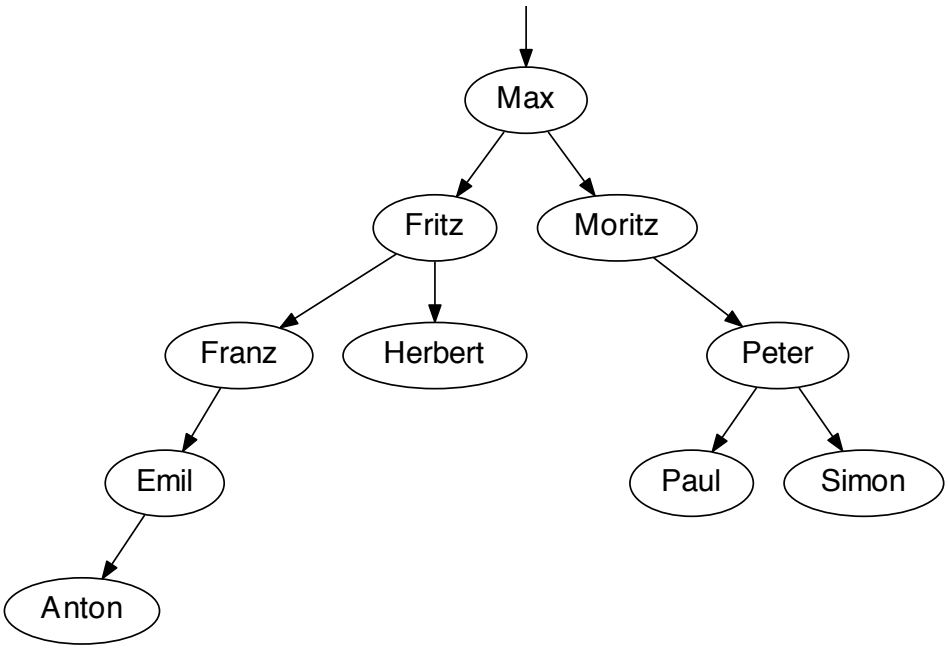
Definition: Binärer Suchbaum

Eine binärer Suchbaum ist ein Binärbaum mit folgenden Eigenschaften:

- Die Knoten des Baums sind mit Schlüsseln aus einer geordneten Menge K beschriftet
- Für jeden Knoten N gilt:
  - Alle Schlüssel im linken Teilbaum von N sind kleiner als der Schlüssel von N
  - Alle Schlüssel im rechten Teilbaum von N sind größer als der Schlüssel von N

- Geordnete Menge K:
  - Auf K ist eine Ordnungsrelation $>$ definiert
  - Wenn $k_1, k_2 \in K$, dann gilt enweder $k_1 > k_2$ oder $k_2 > k_1$ oder $k_1 = k_2$

Binärer Suchbaum – Beispiel

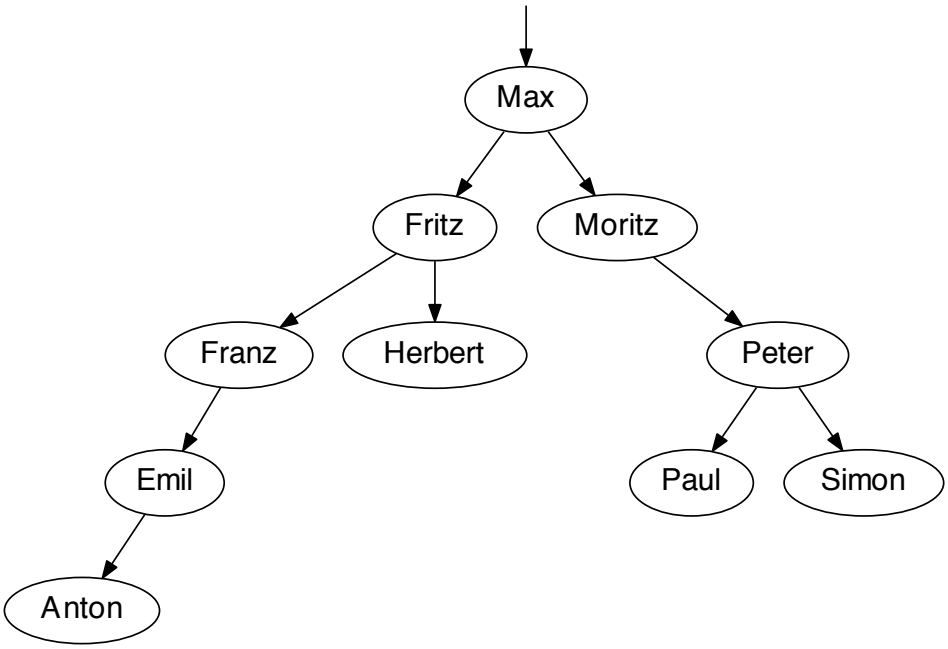


Schlüsselmenge K: Strings mit normaler alphabetischer Ordnung

Suche im Suchbaum

- Gegeben: Baum B mit Wurzel W, Schlüssel k
- Algorithmus: Suche k in B
  - Wenn B leer ist: Ende, k ist nicht in B
  - Wenn $W.key > k$: Suche im linken Teilbaum von B
  - Wenn $W.key < k$: Suche im rechten Teilbaum von B
  - Sonst: $W.key = k$: Ende, gefunden

Binärer Suchbaum – Suche



- Wie finde ich raus, ob “Kurt” im Baum ist?
- Wie finde ich raus, ob “Emil” im Baum ist?

Übung: Komplexität der Suche

- Wie ist die Zeitkomplexität der Suchoperation in einem Baum mit *n* Knoten...
  - . . . im schlimmsten Fall?
  - . . . im besten Fall?

Einfügen

- Gegeben: Baum B mit Wurzel W, Schlüssel k
- Gesucht: Baum B', der aus B entsteht, wenn k eingefügt wird
- Idee:

Einfügen

- Gegeben: Baum B mit Wurzel W, Schlüssel k
- Gesucht: Baum B', der aus B entsteht, wenn k eingefügt wird
- Idee:
  - Suche nach k
  - Falls k nicht in B ist, setze es an der Stelle ein, an der es gefunden worden wäre

Einfügen

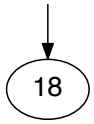
- Gegeben: Baum B mit Wurzel W, Schlüssel k
- Gesucht: Baum B', der aus B entsteht, wenn k eingefügt wird
- Idee:
  - Suche nach k
  - Falls k nicht in B ist, setze es an der Stelle ein, an der es gefunden worden wäre
- Implementierung z.B. funktional:
  - Wenn B leer ist, dann ist ein Baum mit einem Knoten mit Schlüssel k der gesuchte Baum
  - Ansonsten:
    - Wenn $W.key > k$: Ersetze den linken Teilbaum von B durch den Baum, der entsteht, wenn man k in ihn einfügt
    - Wenn $W.key < k$: Ersetze den rechten Teilbaum von B durch den Baum, der entsteht, wenn man k in ihn einfügt
    - Ansonsten: k ist schon im Baum

Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein

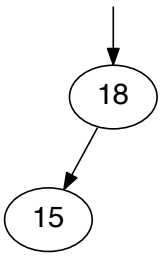
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



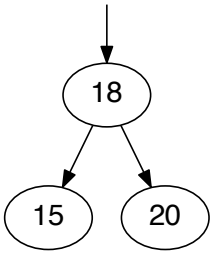
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



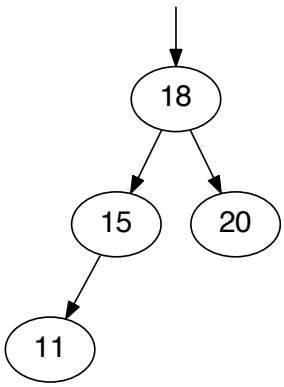
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



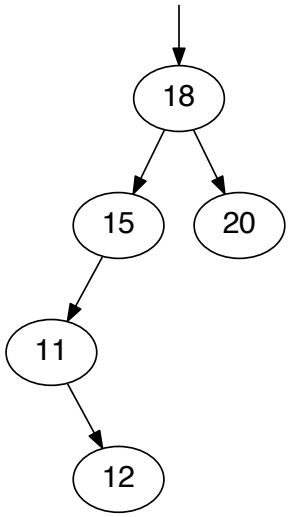
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



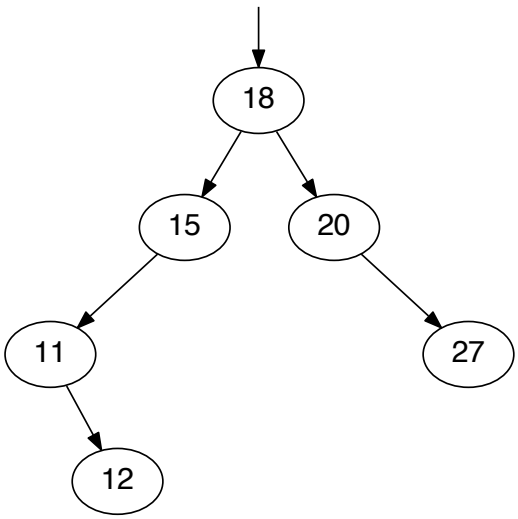
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



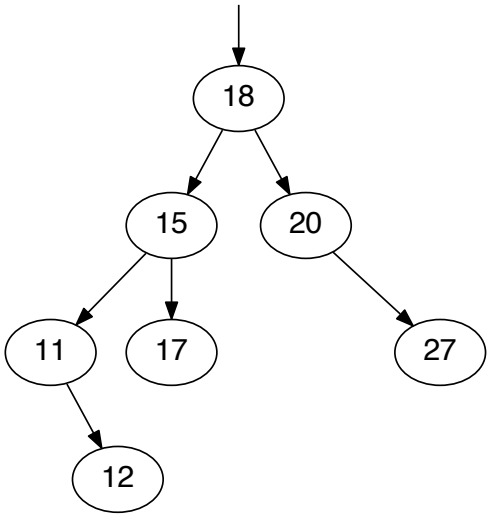
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



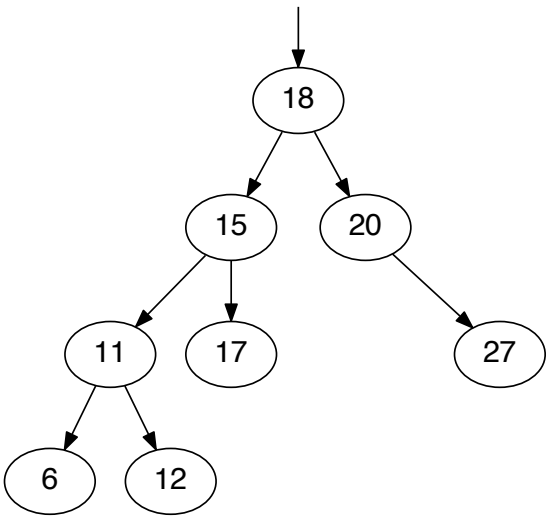
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



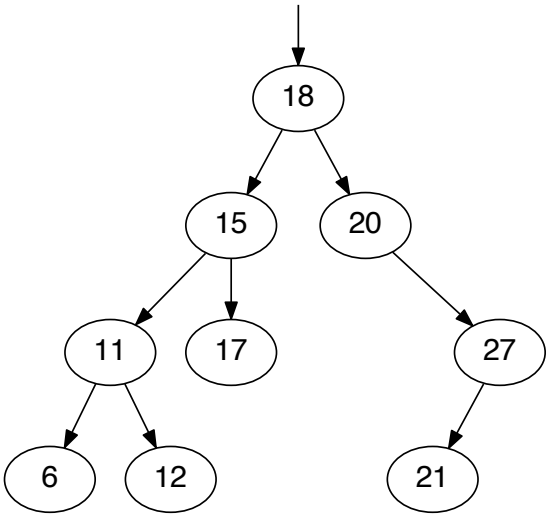
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



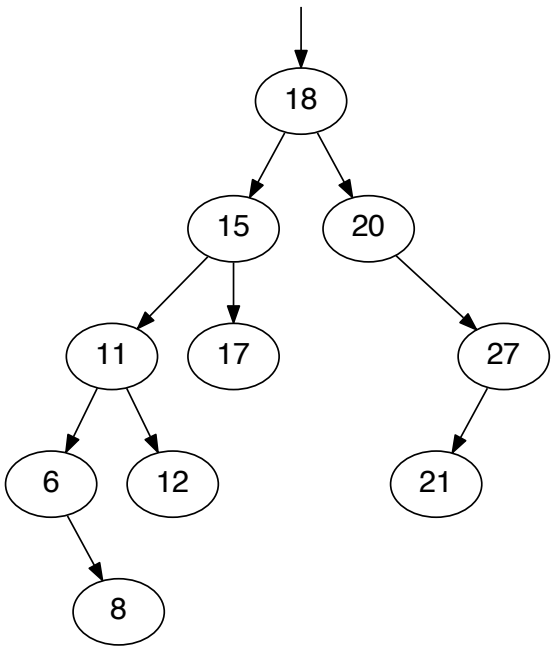
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



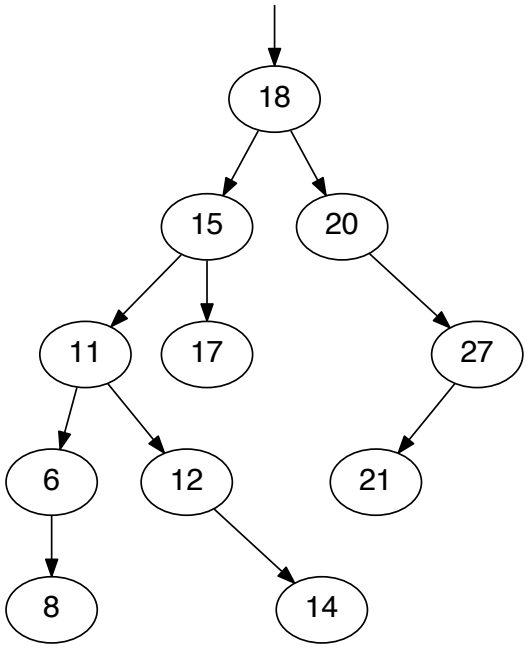
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



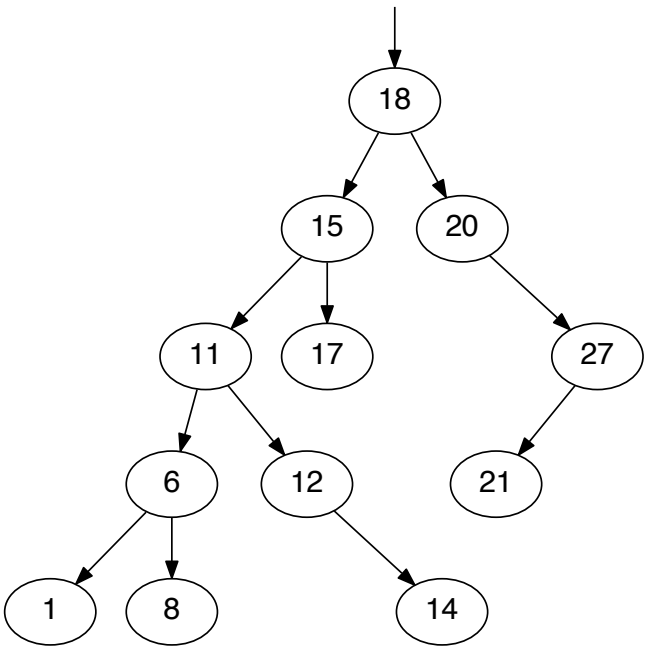
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



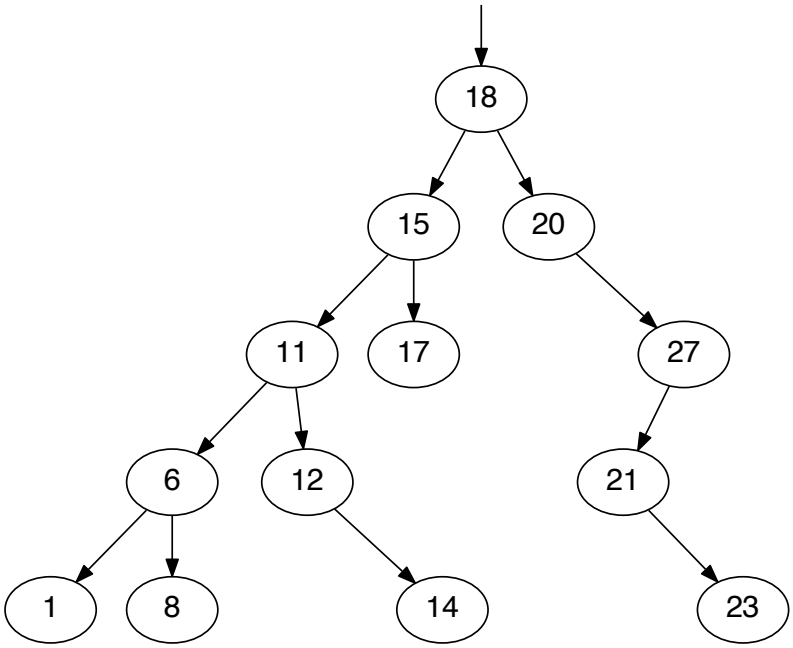
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



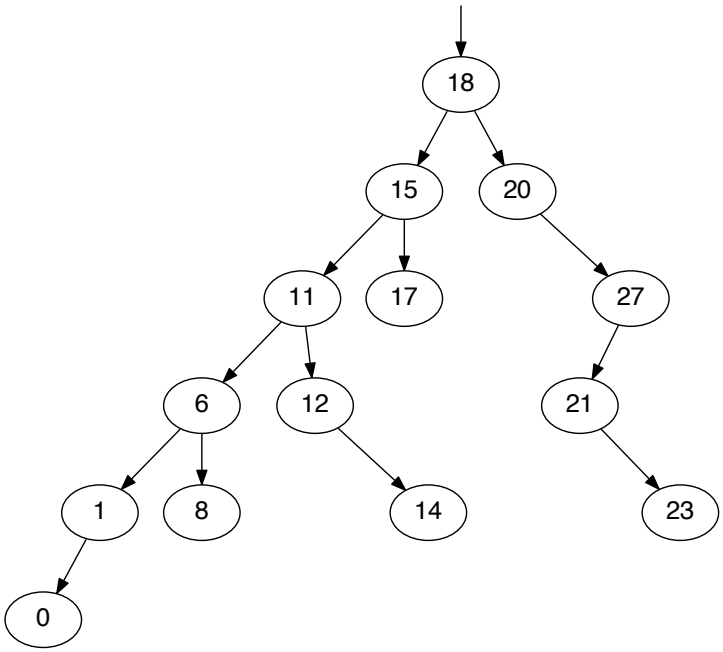
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



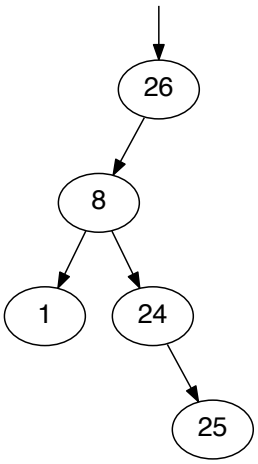
Einfügen: Beispiel

Füge $K = (18, 15, 20, 11, 12, 27, 17, 6, 21, 8, 14, 1, 23, 0)$ in dieser Reihenfolge in einen Anfangs leeren Baum ein



Übung: Einfügen

- Fügen Sie die Schlüssel $K = (15, 0, 3, 5, 4, 27, 14, 21, 28, 6)$ in dieser Reihenfolge in den gegebenen binären Suchbaum ein



Implementierung: Datentyp

```
class TreeNode(object):
    """
    Binary tree node
    """
    def __init__(self, key, lchild = None, rchild = None):
        self.key    = key
        self.lchild = lchild
        self.rchild = rchild
```

- Der leere Baum wird durch `None` repräsentiert
- In C:
  - `struct` mit `key` und Pointern `lchild`, `rchild`
  - Der leere Baum ist `NULL`

Implementierung: Suchen

```
def find(tree, key):
    if tree:
        if key < tree.key:
            return find(tree.lchild, key)
        if key > tree.key:
            return find(tree.rchild, key)
        return tree
    else:
        return None
```

- Rückgabe: Knoten mit dem gesuchten Schlüssel oder `None`

Implementierung: Einfügen

```
def insert(tree, key):
    if not tree:
        return TreeNode(key)
    else:
        if key < tree.key:
            tree.lchild = insert(tree.lchild, key)
        elif key > tree.key:
            tree.rchild = insert(tree.rchild, key)
        else:
            print "Error: Duplicate key"
        return tree
```

- Funktionaler Ansatz: Die Funktion gibt den neuen Baum zurück

Geordnete Ausgabe

- Aufgabe: Alle Schlüssel in der geordneten Reihefolge ausgeben
  - Analog: Jede Operation, die über alle Schlüssel geordnet iteriert
- Idee:
  - Gib den linken Teilbaum (rekursiv) aus
  - Gib den Schlüssel der Wurzel aus
  - Gib den rechten Teilbaum (rekursiv) aus
- Was ist die Abbruchbedingung der Rekursion?

Übung: Durchlaufen

- Wie ist die Komplexität der *geordneten Ausgabe*?
  - . . . im *best case*
  - . . . im *worst case*

Diskussion: Löschen

- Frage: Wie können wir einen Knoten aus dem Baum löschen?

Diskussion: Löschen

- Frage: Wie können wir einen Knoten aus dem Baum löschen?
- Diskussionsgrundlage:
  - Fall 1: Knoten ist ein Blatt
  - Fall 2: Knoten hat einen Nachfolger
  - Fall 3: Knoten hat zwei Nachfolger

Diskussion: Löschen

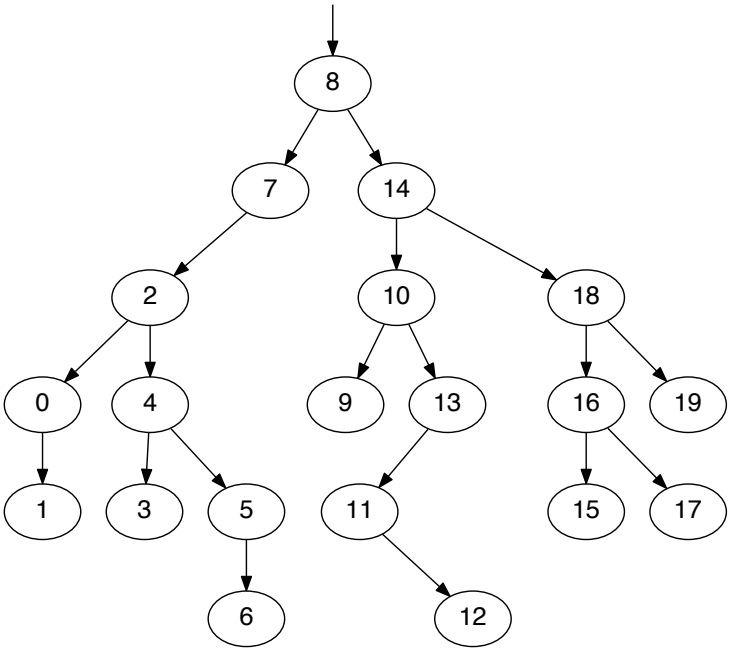
- Frage: Wie können wir einen Knoten aus dem Baum löschen?
- Diskussionsgrundlage:
  - Fall 1: Knoten ist ein Blatt
  - Fall 2: Knoten hat einen Nachfolger
  - Fall 3: Knoten hat zwei Nachfolger
- Was ist die Komplexität einer Löschoperation?

Ende Vorlesung 16

Löschen in binären Suchbäumen (1)

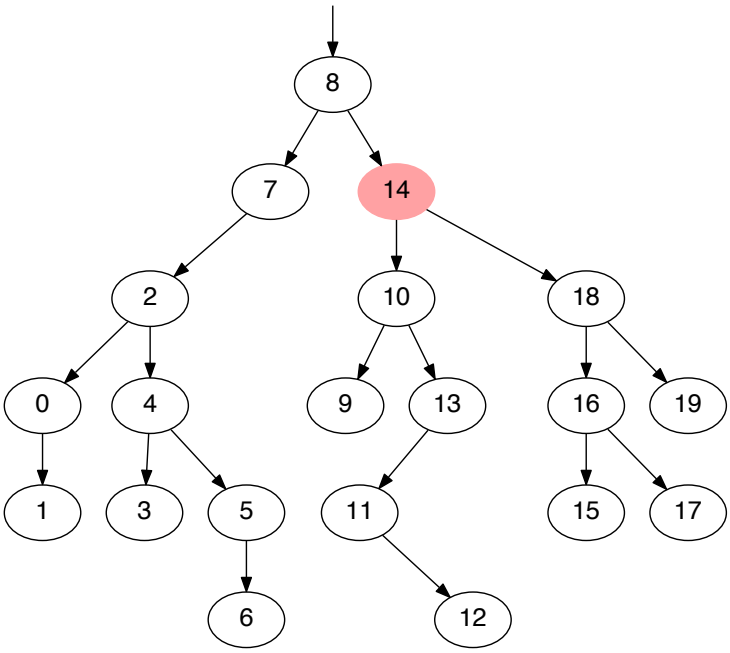
- Problem: Entferne einen Knoten K mit gegebenen Schlüssel k aus dem Suchbaum
  - . . . und erhalte die Binärbaumeigenschaft
  - . . . und erhalte die Suchbaumeigenschaft
- Fallunterscheidung:
  - Fall 1: Knoten hat keinen Nachfolger
    - Lösung: Schneide Knoten ab
    - Korrektheit: Offensichtlich
  - Fall 2: Knoten hat einen Nachfolger
    - Lösung: Ersetze Knoten durch seinen einzigen Nachfolger
    - Korrektheit: Alle Knoten in diesem Baum sind größer (bzw. kleiner) als die Knote im Vorgänger des gelöschten Knotens
  - Fall 3: Knoten hat zwei Nachfolger
    - Lösung?

Löschen: Beispiel



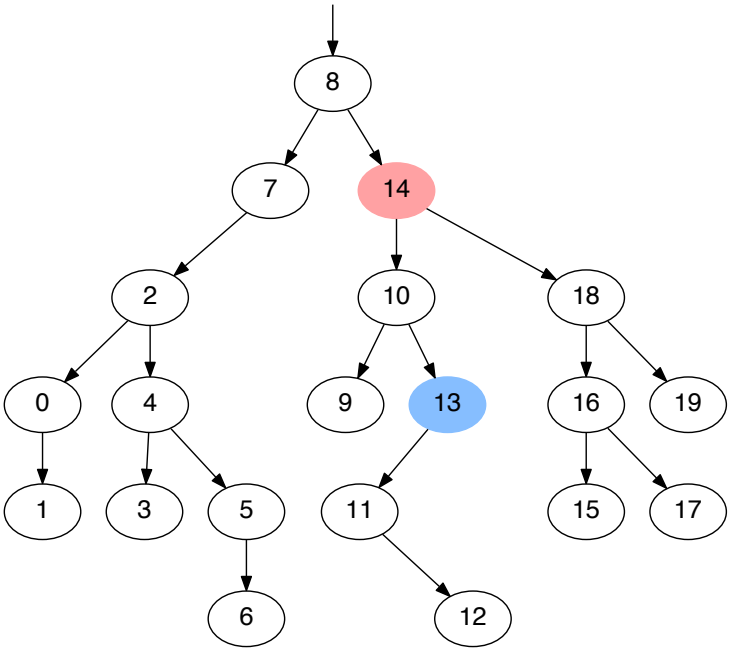
Aufgabe: Lösche Knoten 14

Löschen: Beispiel



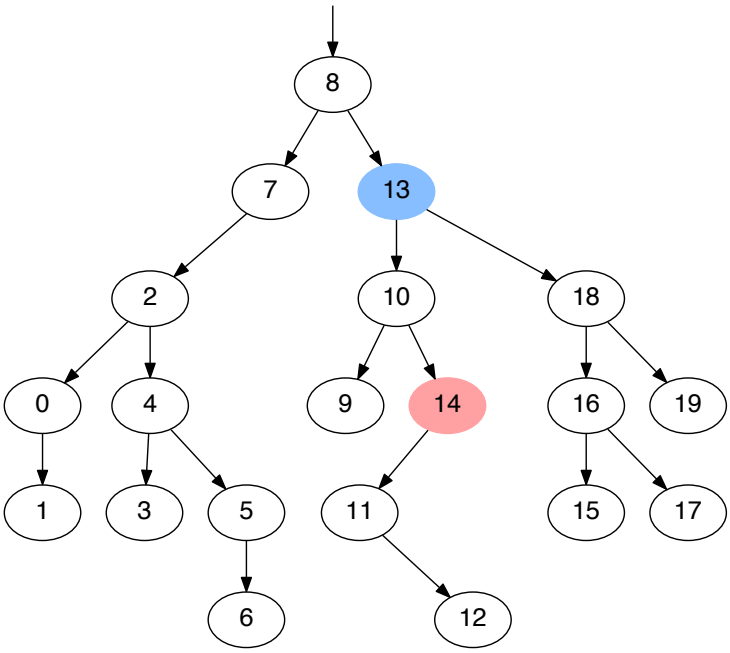
Aufgabe: Lösche Knoten 14

Löschen: Beispiel



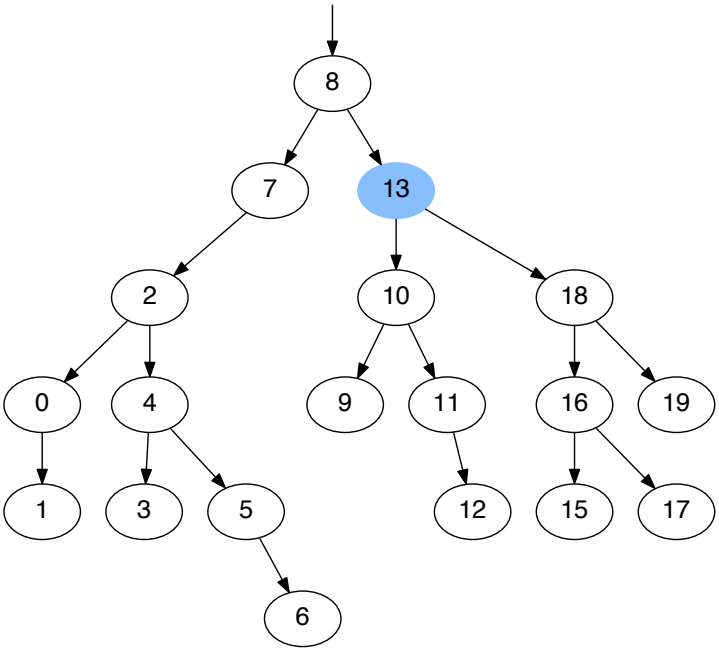
Aufgabe: Lösche Knoten 14

Löschen: Beispiel



Aufgabe: Lösche Knoten 14

Löschen: Beispiel



Aufgabe: Lösche Knoten 14

Löschen in binären Suchbäumen (2)

- Problem: Entferne einen Knoten K mit gegebenen Schlüssel k aus dem Suchbaum
- Fall 3: Knoten K hat zwei Nachfolger
  - Lösung:
    - Suche größten Knoten G im linken Teilbaum
    - Tausche G und K (oder einfacher: Ihre Schlüssel/Werte)
    - Lösche rekursiv k im linken Teilbaum von (nun) G

Löschen in binären Suchbäumen (2)

- Problem: Entferne einen Knoten K mit gegebenen Schlüssel k aus dem Suchbaum
- Fall 3: Knoten K hat zwei Nachfolger
  - Lösung:
    - Suche größten Knoten G im linken Teilbaum
    - Tausche G und K (oder einfacher: Ihre Schlüssel/Werte)
    - Lösche rekursiv k im linken Teilbaum von (nun) G
  - Anmerkungen
    - Wie viele Nachfolger hat G ursprünglich?

Löschen in binären Suchbäumen (2)

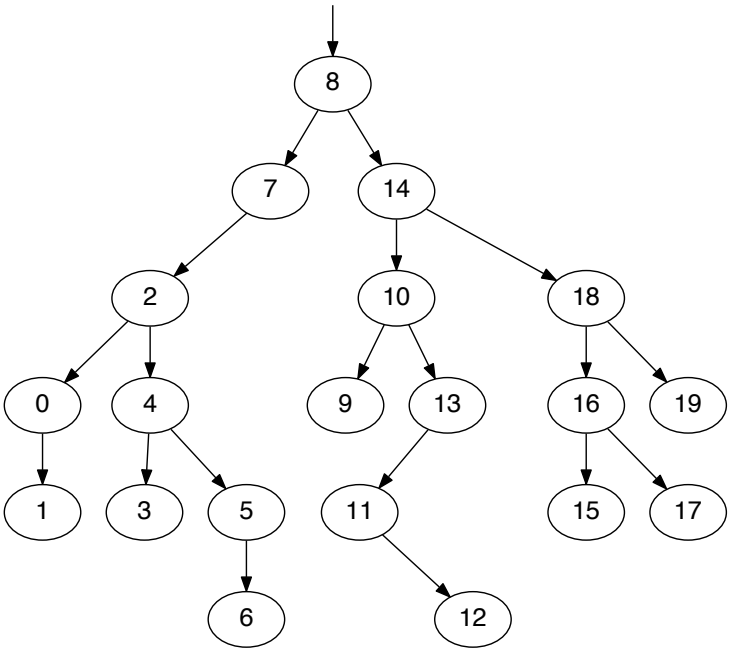
- Problem: Entferne einen Knoten K mit gegebenen Schlüssel k aus dem Suchbaum
- Fall 3: Knoten K hat zwei Nachfolger
  - Lösung:
    - Suche größten Knoten G im linken Teilbaum
    - Tausche G und K (oder einfacher: Ihre Schlüssel/Werte)
    - Lösche rekursiv k im linken Teilbaum von (nun) G
  - Anmerkungen
    - Wie viele Nachfolger hat G ursprünglich?
    - Also: Fall 3 kommt höchstens ein Mal pro Löschvorgang vor

Löschen in binären Suchbäumen (2)

- Problem: Entferne einen Knoten K mit gegebenen Schlüssel k aus dem Suchbaum
- Fall 3: Knoten K hat zwei Nachfolger
  - Lösung:
    - Suche größten Knoten G im linken Teilbaum
    - Tausche G und K (oder einfacher: Ihre Schlüssel/Werte)
    - Lösche rekursiv k im linken Teilbaum von (nun) G
  - Anmerkungen
    - Wie viele Nachfolger hat G ursprünglich?
    - Also: Fall 3 kommt höchstens ein Mal pro Löschvorgang vor
    - Komplexität: $\mathcal{O}(\log n)$ (wir folgen nur einem Ast)

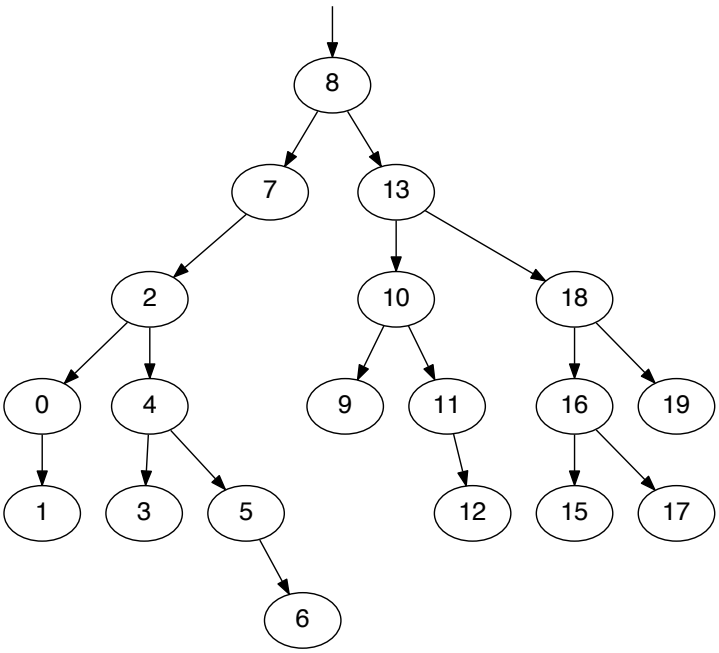
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



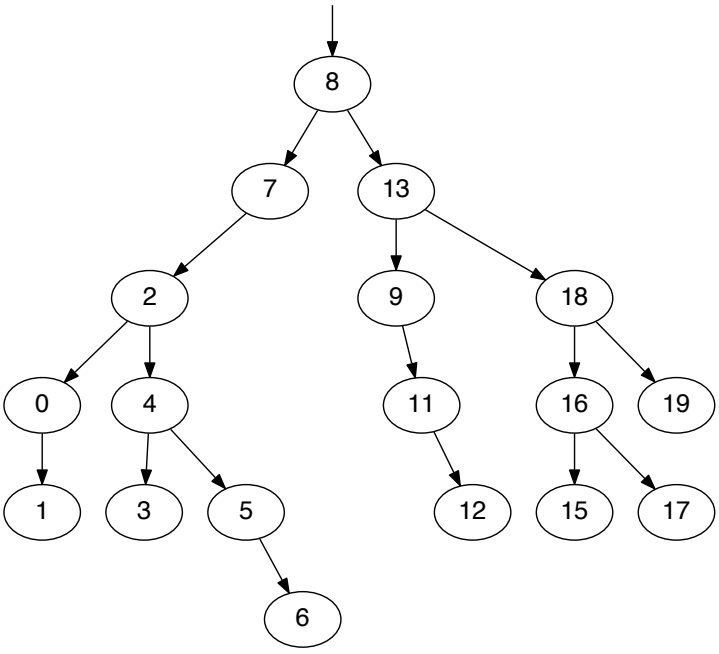
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



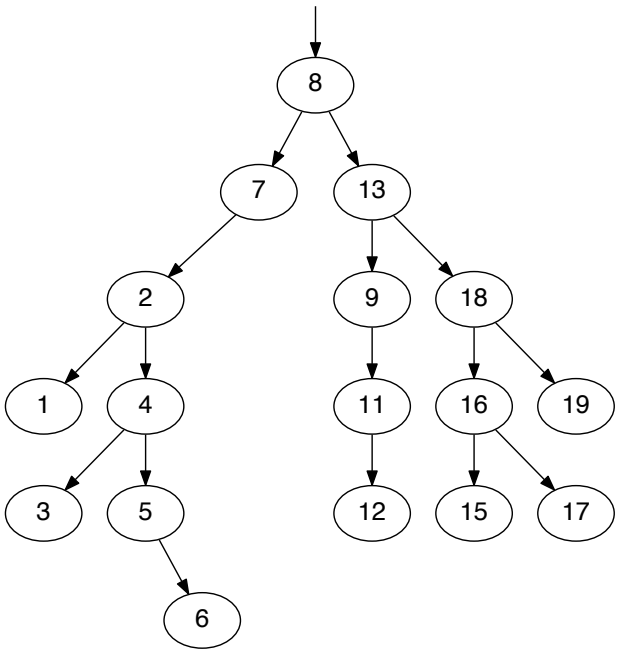
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



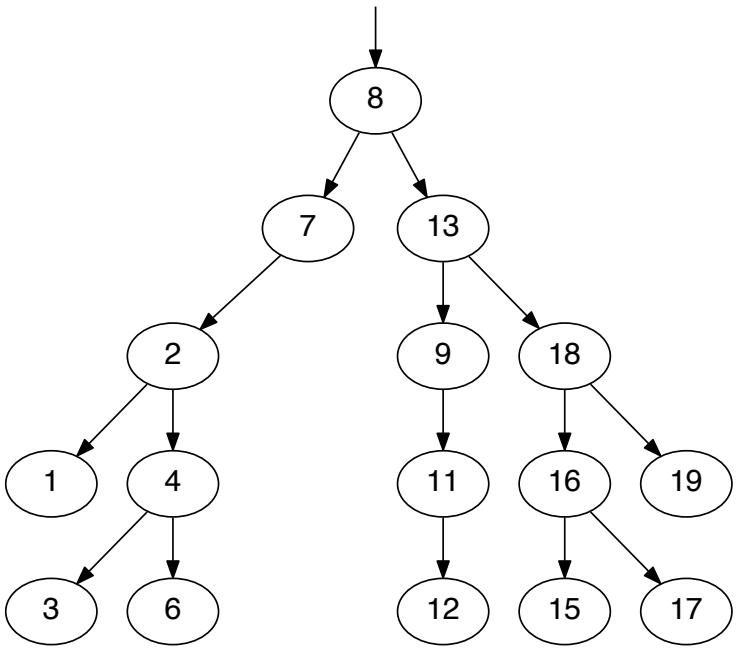
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



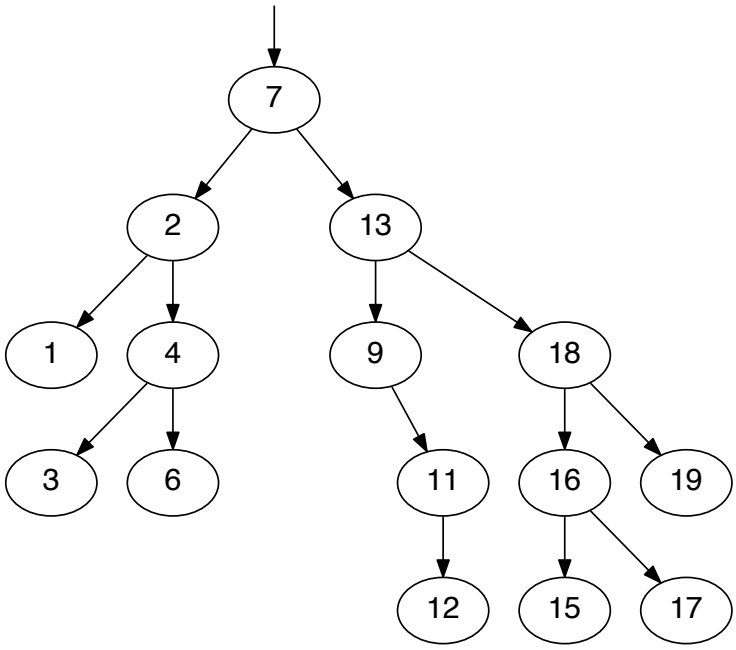
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



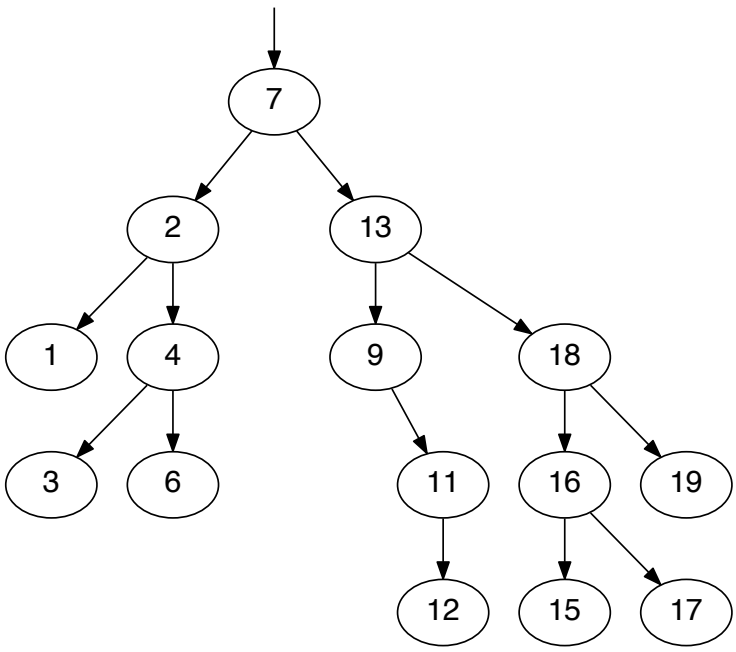
Löschen: Beispiel

Wir löschen: (14 , 10 , 0 , 5 , 8)



Übung: Löschen

Löschen Sie die Knoten $(4, 7, 12, 6, 11)$ in dieser Reihenfolge



Löschen: Implementierung (1)

Finde größten Knoten in einem Baum und gib ihn zurück

```
def find_max(tree):
    while tree.rchild:
        tree = tree.rchild
    return tree
```


Löschen: Implementierung (2)

```
def delete(tree, key):
    if not tree:
        print "Error: Key does not exist"
    if key < tree.key:
        tree.lchild = delete(tree.lchild, key)
    elif key > tree.key:
        tree.rchild = delete(tree.rchild, key)
    else:
        if tree.lchild and tree.rchild:
            max_node = find_max(tree.lchild)
            tmp = max_node.key
            max_node.key = tree.key
            tree.key = tmp
            tree.lchild = delete(tree.lchild, key)
        elif tree.lchild:
            return tree.lchild
        elif tree.rchild:
            return tree.rchild
        else:
            return None
    return tree
```

Binärer Suchbaum: Scorecard

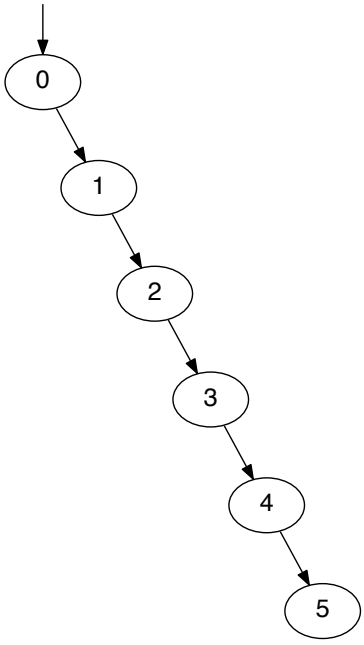
Voraussetzung:

- Baum hat n Knoten
- Kostenfunktion betrachtet Durchschnittsfall

Operation	Kosten
Leeren Baum anlegen	$\mathcal{O}(1)$
Schlüssel finden	$\mathcal{O}(\log n)$
Schlüssel einfügen	$\mathcal{O}(\log n)$
Schlüssel bedingt einfügen	$\mathcal{O}(\log n)$
Schlüssel löschen	$\mathcal{O}(\log n)$
Sortierte Ausgabe:	$\mathcal{O}(n)$

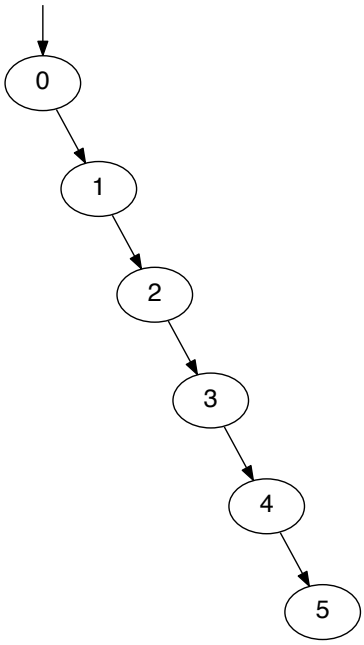
Unbalancierte Bäume

- Problem: Binärbäume können entarten
  - Z.B. durch Einfügen einer sortierten Schlüsselmenge
  - Z.B. durch systematisches Ausfügen von kleinen/großen Schlüsseln
  - Worst case: Baum wird zur Liste
    - Viele Operationen kosten dann $\mathcal{O}(n)$



Unbalancierte Bäume

- Problem: Binärbäume können entarten
  - Z.B. durch Einfügen einer sortierten Schlüsselmenge
  - Z.B. durch systematisches Ausfügen von kleinen/großen Schlüsseln
  - Worst case: Baum wird zur Liste
    - Viele Operationen kosten dann $\mathcal{O}(n)$



- Lösung: Rebalancieren von Bäumen!
  - Relativ billig ($\mathcal{O}(1)/\mathcal{O}(\log n)$ pro Knoten)
  - Kann gutartigen Baum garantieren!

Balancierte Bäume

- Idee: An jedem Knoten sollen rechter und linker Teilbaum ungefähr gleich groß sein
  - Divide-and-Conquer funktioniert nahezu optimal
  - “Alles ist $\mathcal{O}(\log n)$” z.B. mit dem Master-Theorem

Balancierte Bäume

- Idee: An jedem Knoten sollen rechter und linker Teilbaum ungefähr gleich groß sein
  - Divide-and-Conquer funktioniert nahezu optimal
  - “Alles ist $\mathcal{O}(\log n)$” z.B. mit dem Master-Theorem
- Größenbalancierter Binärbaum
  - Beide Teilbäume haben in etwa ähnlich viele Knoten
  - Optimal für Divide-and-Conquer

Balancierte Bäume

- Idee: An jedem Knoten sollen rechter und linker Teilbaum ungefähr gleich groß sein
  - Divide-and-Conquer funktioniert nahezu optimal
  - “Alles ist $\mathcal{O}(\log n)$” z.B. mit dem Master-Theorem
- Größenbalancierter Binärbaum
  - Beide Teilbäume haben in etwa ähnlich viele Knoten
  - Optimal für Divide-and-Conquer
- Höhenbalancierter Binärbaum
  - Beide Teilbäume haben in etwa die gleiche Höhe
  - Gut genug für $\log n$
  - Einfacher zu erreichen

Balancierte Bäume

- Idee: An jedem Knoten sollen rechter und linker Teilbaum ungefähr gleich groß sein
  - Divide-and-Conquer funktioniert nahezu optimal
  - “Alles ist $\mathcal{O}(\log n)$” z.B. mit dem Master-Theorem
- Größenbalancierter Binärbaum
  - Beide Teilbäume haben in etwa ähnlich viele Knoten
  - Optimal für Divide-and-Conquer
- Höhenbalancierter Binärbaum
  - Beide Teilbäume haben in etwa die gleiche Höhe
  - Gut genug für $\log n$
  - Einfacher zu erreichen
- (Zugriffs-)Gewichtbalancierter Binärbaum
  - Die Wahrscheinlichkeit, in einen der beiden Bäume absteigen zu müssen, ist etwa gleich groß
  - Hängt von der zu erwartenden Verteilung der Schlüssel ab
  - Zu kompliziert für heute - bei Interesse Stichwort *Splay Trees*

Diskussion

- Wie komme ich von einem unbalancierten Baum zu einem balancierten Baum?

Diskussion

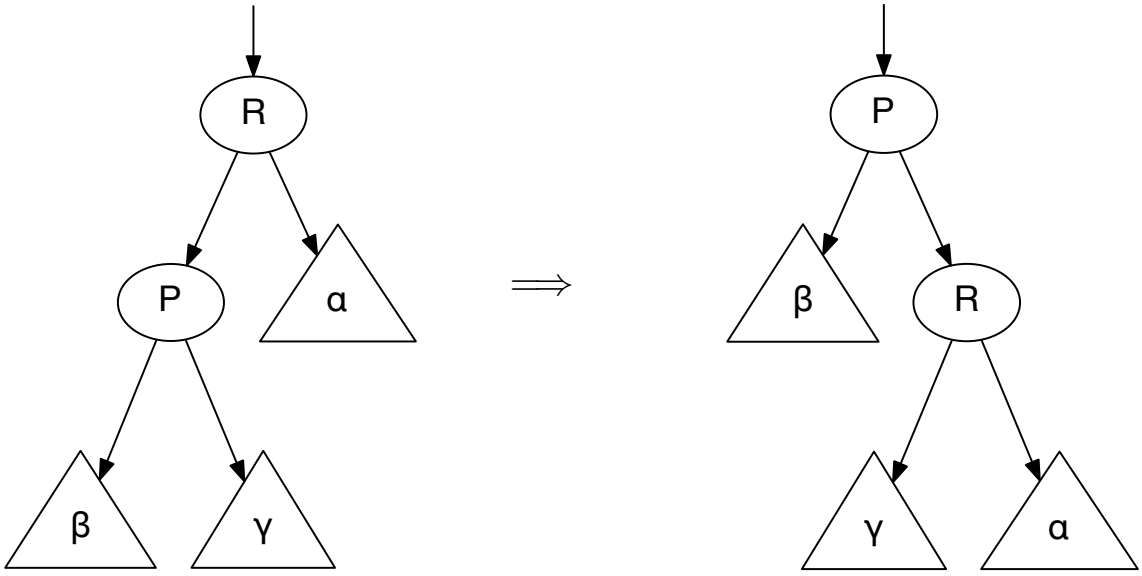
- Wie komme ich von einem unbalancierten Baum zu einem balancierten Baum?
- . . . unter Erhalt der Suchbaumeigenschaft
- . . . billig

Rotationen

- Rotationen machen einen Nachbarknoten der Wurzel zur neuen Wurzel
  - Dabei: Suchbaumkriterium bleibt erhalten
  - Höhe der Teilbäume ändert sich um 1
  - Terminologie: Der zur Wurzel beförderte Knoten heißt Pivot
- Rechts-Rotation:
  - Der linke Nachfolger der Wurzel ist der Pivot und wird neue Wurzel
  - Der rechte Nachfolger des Pivot wird linker Nachfolger der Wurzel
  - Die alte Wurzel wird rechter Nachfolger des Pivot
- Links-Rotation:
  - Alles andersrum

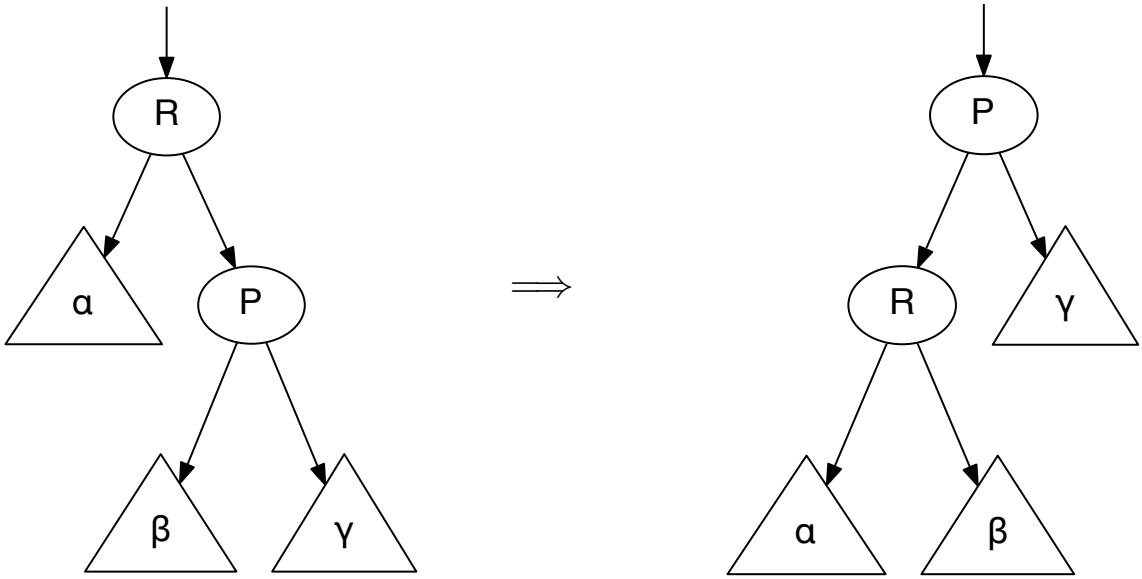
Rechts-Rotation

Anmerkung: α, β, γ sind beliebige Teilbäume



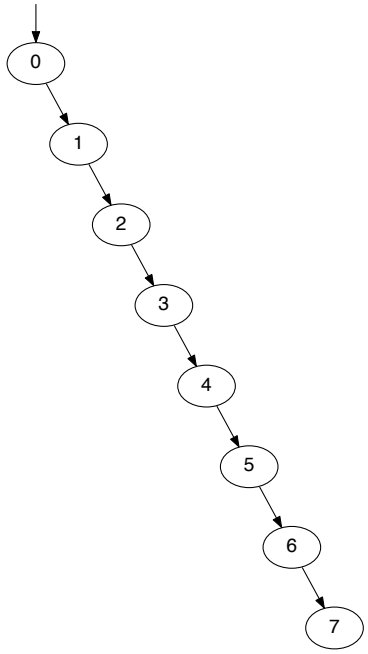
Links-Rotation

Anmerkung: α, β, γ sind beliebige Teilbäume



Übung: Rebalancieren

Rebalancieren Sie den folgenden Baum nur mit Rechts- und Linksrotationen. Wie gut werden Sie? Wie viele Rotationen brauchen Sie?



Rotationen Implementiert

```
def rotate_r(tree):
    pivot = tree.lchild
    tree.lchild = pivot.rchild
    pivot.rchild = tree
    return pivot

def rotate_l(tree):
    pivot = tree.rchild
    tree.rchild = pivot.lchild
    pivot.lchild = tree
    return pivot
```

AVL-Bäume

- Georgy **A**delson-**V**elsky and E. M. **L**andis: “An algorithm for the organization of information” (1962)
- Automatisch balancierenden Binäre Suchbäume
  - Höhenbalanciert
  - Maximaler erlaubter Höhenunterschied für die Kinder einer Wurzel: $+/-1$
- Höhenunterschied wird in jedem Knoten gespeichert
- Anpassen bei Ein- oder Ausfügen
- Wird der Unterschied größer als 1: Rebalancieren mit Rotationen
  - Maximal zwei Rotationen notwendig ($\leadsto \mathcal{O}(1)$)

AVL-Bäume

- Georgy **A**delson-**V**elsky and E. M. **L**andis: “An algorithm for the organization of information” (1962)
- Automatisch balancierenden Binäre Suchbäume
  - Höhenbalanciert
  - Maximaler erlaubter Höhenunterschied für die Kinder einer Wurzel: $+/-1$
- Höhenunterschied wird in jedem Knoten gespeichert
- Anpassen bei Ein- oder Ausfügen
- Wird der Unterschied größer als 1: Rebalancieren mit Rotationen
  - Maximal zwei Rotationen notwendig ($\leadsto \mathcal{O}(1)$)

Ende Vorlesung 17

Graphalgorithmen

Graphen

Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Ein ungerichteter Graph V ist ein gerichteter Graph (V, E), bei dem die Relation E symmetrisch ist.
$(a, b) \in E \Leftrightarrow (b, a) \in E$

(Default: Ungerichtete Graphen)

Graphen

Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Ein ungerichteter Graph V ist ein gerichteter Graph (V, E), bei dem die Relation E symmetrisch ist.
$(a, b) \in E \Leftrightarrow (b, a) \in E$

(Default: Ungerichtete Graphen)

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1), \\
&\quad (1,5),(5,1),(3,4),(4,3), \\
&\quad (4,5),(5,4)\}
\end{aligned}
$$

Graphen

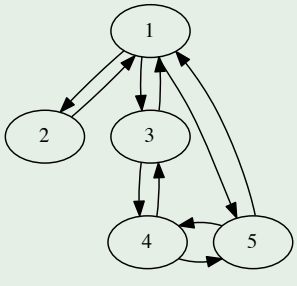
Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Ein ungerichteter Graph V ist ein gerichteter Graph (V, E), bei dem die Relation E symmetrisch ist.
$(a, b) \in E \Leftrightarrow (b, a) \in E$

(Default: Ungerichtete Graphen)

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1), \\
&\quad (1,5),(5,1),(3,4),(4,3), \\
&\quad (4,5),(5,4)\}
\end{aligned}
$$



Graphen

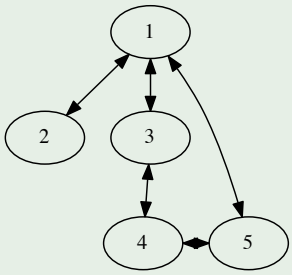
Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Ein ungerichteter Graph V ist ein gerichteter Graph (V, E), bei dem die Relation E symmetrisch ist.
$(a, b) \in E \Leftrightarrow (b, a) \in E$

(Default: Ungerichtete Graphen)

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),\\
&\quad (1,5),(5,1),(3,4),(4,3),\\
&\quad (4,5),(5,4)\}
\end{aligned}
$$



Graphen

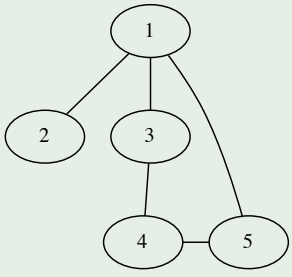
Graph

Ein gerichteter Graph (Digraph) besteht aus einer Knotenmenge V und einer Kantenmenge $E \subseteq V \times V$.

Ein ungerichteter Graph V ist ein gerichteter Graph (V, E), bei dem die Relation E symmetrisch ist.
$(a, b) \in E \Leftrightarrow (b, a) \in E$

(Default: Ungerichtete Graphen)

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),\\
&\quad (1,5),(5,1),(3,4),(4,3),\\
&\quad (4,5),(5,4)\}
\end{aligned}
$$



205

Beschriftete Graphen

Beschrifteter Graph

Ein knoten-/kantenbeschrifteter Graph $G = (V, E)$: Graph mit Beschriftungsfunktion(en) $v : V \to L_V$ bzw. $e : E \to L_E$ für Mengen L_V, L_E.
Ist L eine Menge von Zahlen, spricht man auch von einem gewichteten Graphen.

Beschriftete Graphen

Beschrifteter Graph

Ein knoten-/kantenbeschrifteter Graph $G = (V, E)$: Graph mit Beschriftungsfunktion(en) $v : V \to L_V$ bzw. $e : E \to L_E$ für Mengen L_V, L_E.
Ist L eine Menge von Zahlen, spricht man auch von einem gewichteten Graphen.

Beschriftungen

$$
\begin{aligned}
G &= (V, E) = (\{1,2,3,4,5\}, \{(1,2),(2,1),(1,3),\ldots\}) \\
L_1 &= \{\text{Köln, Hamburg, Bremen, Stuttgart, Frankfurt}\} \\
L_2 &= \mathbb{N} \\
v &= \{1 \mapsto \text{Köln}, 2 \mapsto \text{Hamburg}, \ldots\} \\
e &= \{(1,2) \mapsto 430, (1,3) \mapsto 321, (3,4) \mapsto 626, \ldots\}
\end{aligned}
$$

Anwendung von Graphen

- Netzwerke
  - Straßen, Wasser-, Stromversorgung
  - Computernetzwerke
  - soziale Netzwerke
- Technik
  - Schaltkreise
  - Flussdiagramme
  - Links im WWW
- Hierarchien
  - Vererbung in OO-Sprachen
  - Taxonomien

Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix

Die Adjazenzmatrix A für einen Graphen mit n Knoten hat die Dimension $n \times n$ und die Werte 0 und 1. $A(x, y) = 1$ bedeutet, dass es eine Kante von x nach y gibt.

Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix

Die Adjazenzmatrix A für einen Graphen mit n Knoten hat die Dimension $n \times n$ und die Werte 0 und 1. $A(x, y) = 1$ bedeutet, dass es eine Kante von x nach y gibt.

Die Adjazenzmatrix kann als zweidimensionales Array repräsentiert werden.

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),\\
&\quad (1,5),(5,1),(3,4),(4,3),\\
&\quad (4,5),(5,4)\}
\end{aligned}
$$

Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix

Die Adjazenzmatrix A für einen Graphen mit n Knoten hat die Dimension $n \times n$ und die Werte 0 und 1. $A(x, y) = 1$ bedeutet, dass es eine Kante von x nach y gibt.

Die Adjazenzmatrix kann als zweidimensionales Array repräsentiert werden.

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),\\
&\quad (1,5),(5,1),(3,4),(4,3),\\
&\quad (4,5),(5,4)\}
\end{aligned}
$$

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	0	0	0
3	1	0	0	1	0
4	0	0	1	0	1
5	1	0	0	1	0

Repräsentation von Graphen: Adjazenzliste

Adjazenzliste

Die Adjazenzliste L für einen Knoten x in einem Graphen G enthält alle Knoten y, zu denen es eine von x ausgehende Kante gibt.

Repräsentation von Graphen: Adjazenzliste

Adjazenzliste

Die Adjazenzliste L für einen Knoten x in einem Graphen G enthält alle Knoten y, zu denen es eine von x ausgehende Kante gibt.

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1), \\
&\quad (1,5),(5,1),(3,4),(4,3), \\
&\quad (4,5),(5,4)\}
\end{aligned}
$$

Repräsentation von Graphen: Adjazenzliste

Adjazenzliste

Die Adjazenzliste L für einen Knoten x in einem Graphen G enthält alle Knoten y, zu denen es eine von x ausgehende Kante gibt.

$$
\begin{aligned}
V &= \{1,2,3,4,5\} \\
E &= \{(1,2),(2,1),(1,3),(3,1),\\
&\quad (1,5),(5,1),(3,4),(4,3),\\
&\quad (4,5),(5,4)\}
\end{aligned}
\qquad
\begin{aligned}
1 &\mapsto (2,3,5) \\
2 &\mapsto (1) \\
3 &\mapsto (1,4) \\
4 &\mapsto (3,5) \\
5 &\mapsto (1,4)
\end{aligned}
$$

Vorteil gegenüber Matrix:

- Platz $\mathcal{O}(|E|)$ statt $\mathcal{O}(|V|^2)$
- vor allem bei mageren (sparse) Graphen ($|E| \sim |V|$)

Nachteil gegenüber Matrix:

- Entscheidung, ob $(x,y) \in E$ gilt, ist $\mathcal{O}(|N|)$ statt $\mathcal{O}(1)$
- vor allem bei dichten (dense) Graphen ($|E| \sim |V|^2$)

Repräsentation von Graphen-Beschriftungen: Matrix

Bei beschrifteten Graphen können Knotenbeschriftungen in einem Array und Kantenbeschriftungen in der Adjazenzmatrix repräsentiert werden.

n	$v(n)$
1	Köln
2	Hamburg
3	Bremen
4	Stuttgart
5	Frankfurt

$e(m,n)$	1	2	3	4	5
1	∞	430	321	∞	190
2	430	∞	∞	∞	∞
3	321	∞	∞	626	∞
4	∞	∞	626	∞	205
5	190	∞	∞	205	∞

Repräsentation von Graphen-Beschriftungen: Liste

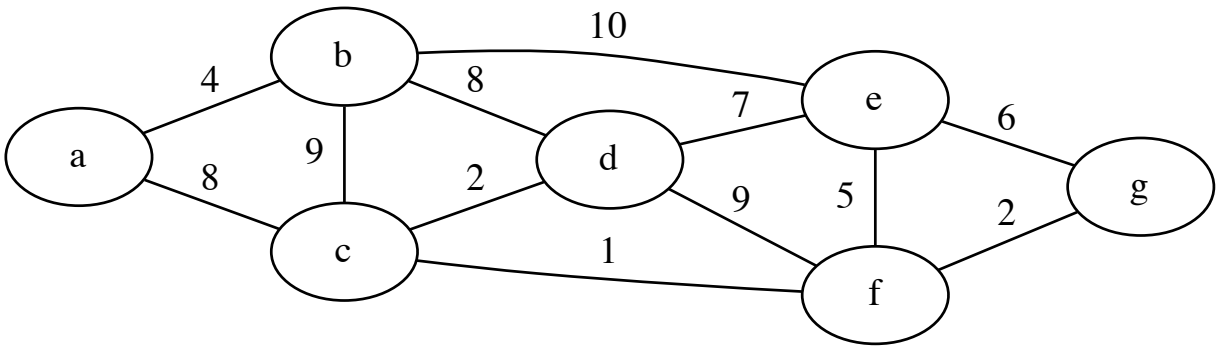
In einer Adjazenzliste können die Kantengewichte zusammen mit der Kante gespeichert werden.

n	$v(n)$
1	Köln
2	Hamburg
3	Bremen
4	Stuttgart
5	Frankfurt

$$
\begin{aligned}
1 &\mapsto ((2,430),(3,321),(5,190)) \\
2 &\mapsto ((1,430)) \\
3 &\mapsto ((1,321),(4,626)) \\
4 &\mapsto ((3,626),(5,205)) \\
5 &\mapsto ((1,190),(4,205))
\end{aligned}
$$

Übung: Adjazenzmatrix und -liste

Geben Sie für den Graphen die Adjazenzmatrix sowie die Adjazenzlisten an.



Graphen: Definitionen

Pfad, Zyklus, Baum

Für einen Graphen G ist ein Pfad eine Folge von Knoten $(v_1, v_2, \ldots, v_k)$, so dass gilt: $\forall i < k : (v_1, v_{i+1} \in E)$.

Graphen: Definitionen

Pfad, Zyklus, Baum

Für einen Graphen G ist ein Pfad eine Folge von Knoten $(v_1, v_2, \ldots, v_k)$, so dass gilt: $\forall i < k : (v_1, v_{i+1} \in E)$.

Ein Knoten y ist von einem Knoten x erreichbar, wenn es einen Pfad von x nach y gibt.

Graphen: Definitionen

Pfad, Zyklus, Baum

Für einen Graphen G ist ein Pfad eine Folge von Knoten $(v_1, v_2, \ldots, v_k)$, so dass gilt: $\forall i < k : (v_1, v_{i+1} \in E)$.

Ein Knoten y ist von einem Knoten x erreichbar, wenn es einen Pfad von x nach y gibt.

Ein Graph $G = (V, E)$ heißt verbunden, wenn wenn jeder Knoten in V von jedem anderen Knoten erreichbar ist.

Graphen: Definitionen

Pfad, Zyklus, Baum

Für einen Graphen G ist ein Pfad eine Folge von Knoten $(v_1, v_2, \ldots, v_k)$, so dass gilt: $\forall i < k : (v_1, v_{i+1} \in E)$.

Ein Knoten y ist von einem Knoten x erreichbar, wenn es einen Pfad von x nach y gibt.

Ein Graph $G = (V, E)$ heißt verbunden, wenn wenn jeder Knoten in V von jedem anderen Knoten erreichbar ist.

Ein Zyklus ist ein Pfad mit $v_1 = v_k$.
(Bei ungerichteten Graphen: Jede Kante darf nur in einer Richtung benutzt werden.)

213

Graphen: Definitionen

Pfad, Zyklus, Baum

Für einen Graphen G ist ein Pfad eine Folge von Knoten $(v_1, v_2, \ldots, v_k)$, so dass gilt: $\forall i < k : (v_1, v_{i+1} \in E)$.

Ein Knoten y ist von einem Knoten x erreichbar, wenn es einen Pfad von x nach y gibt.

Ein Graph $G = (V, E)$ heißt verbunden, wenn wenn jeder Knoten in V von jedem anderen Knoten erreichbar ist.

Ein Zyklus ist ein Pfad mit $v_1 = v_k$.
(Bei ungerichteten Graphen: Jede Kante darf nur in einer Richtung benutzt werden.)

Ein Baum ist ein verbundener azyklischer Graph.

Problemstellungen für Graphen

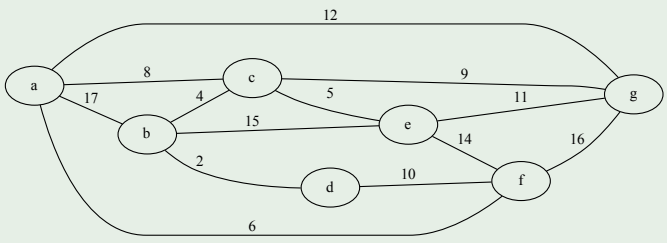
- Minimaler Spannbaum
  - Gegeben: Ungerichteter verbundener kantengewichteter Graph G
  - Gesucht: verbundener Teilgraph G_{min} von G mit minimaler Summe der Kantengewichte
  - Beobachtung: G_{min} muss ein Baum sein
  - Anwendung: Versorgungsnetze
- Kürzeste Pfade
  - Gegeben: Verbundener kantengewichteter Graph G
  - Gesucht: Kürzeste Pfade von x nach y für alle Knoten x, y
  - Anwendung: Routing

Minimaler Spannbaum

	a	b	c	d	e	f	g
a	∞	17	8	∞	∞	6	12
b		∞	4	2	15	7	∞
c			∞	∞	5	∞	9
d				∞	∞	10	∞
e					∞	14	11
f						∞	16
g							∞

Minimaler Spannbaum

	a	b	c	d	e	f	g
a	∞	17	8	∞	∞	6	12
b		∞	4	2	15	7	∞
c			∞	∞	5	∞	9
d				∞	∞	10	∞
e					∞	14	11
f						∞	16
g							∞



Übung: Versuchen Sie, den minimalen Spannbaum zu finden.

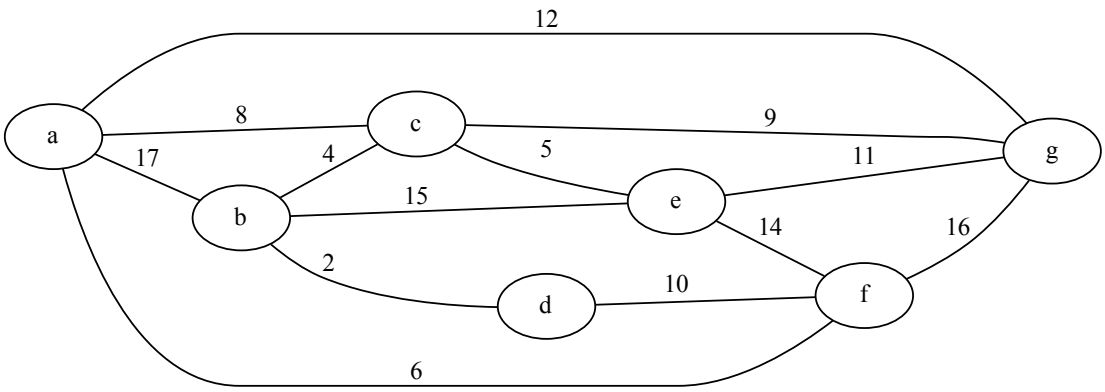
Prim-Algorithmus

(wieder-) entdeckt 1957 von Robert C. Prim, geb. 1921, amerikanischer Mathematiker

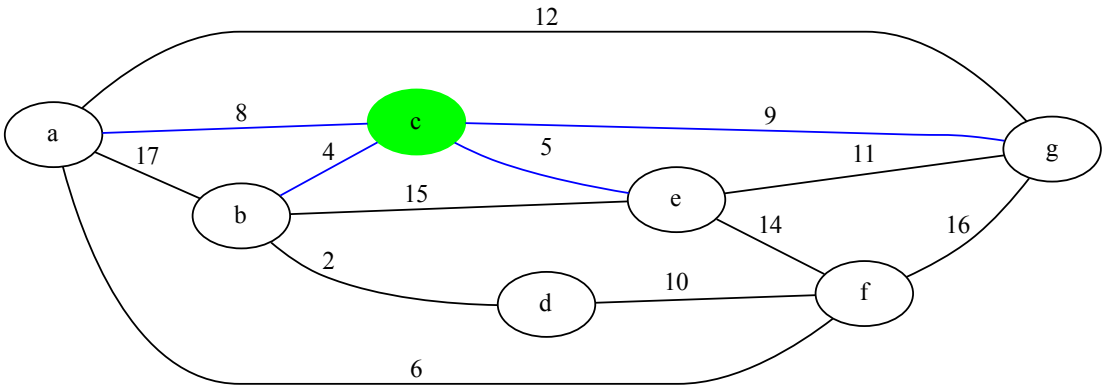
Eingabe: Graph $G = (V, E)$
Ausgabe: MST $T = (V, E_T)$

1. $E_T = \emptyset$
2. wähle v_{start}; $V_b = \{v_{start}\}$; $V_n = V \setminus \{v_{start}\}$
3. solange V_n Knoten enthält
   1. $e_n = (v_b, v_n)$ sei billigste Kante zwischen Knoten aus V_b und V_n
   2. nimm e_n zu E_T hinzu
   3. entferne v_n aus V_n
   4. nimm v_n zu V_b hinzu
4. gib (V, E_T) zurück

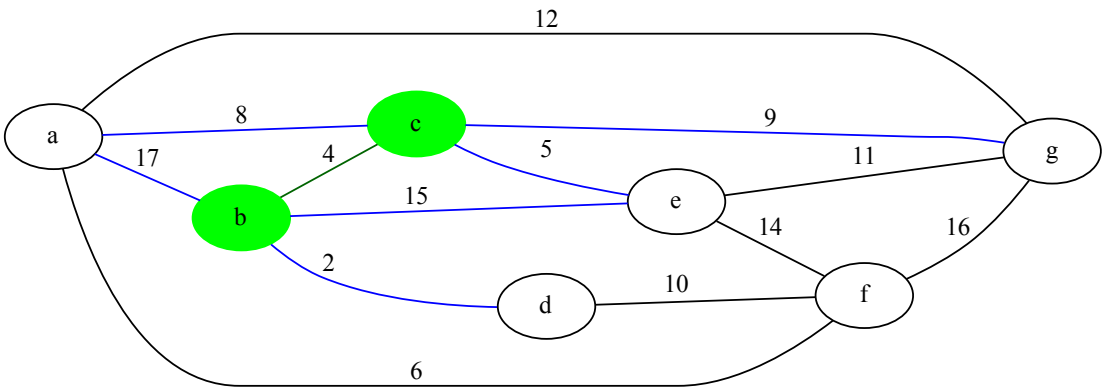
Beispiel: Prim-Algorithmus



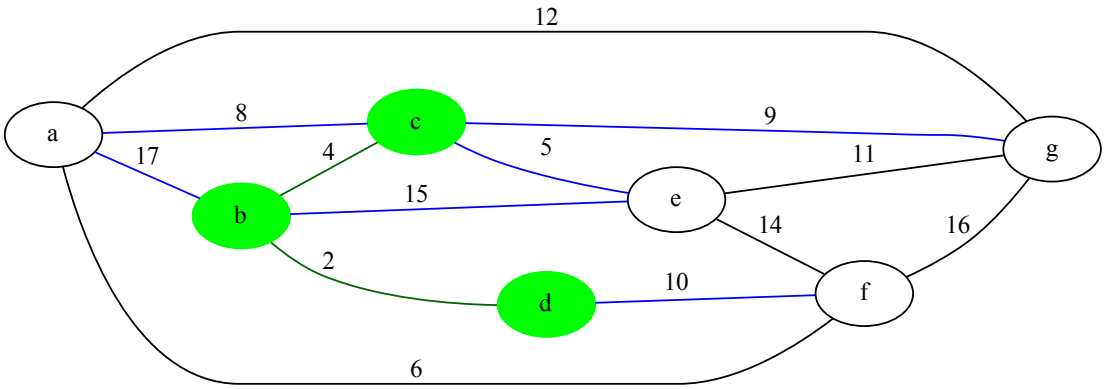
Beispiel: Prim-Algorithmus



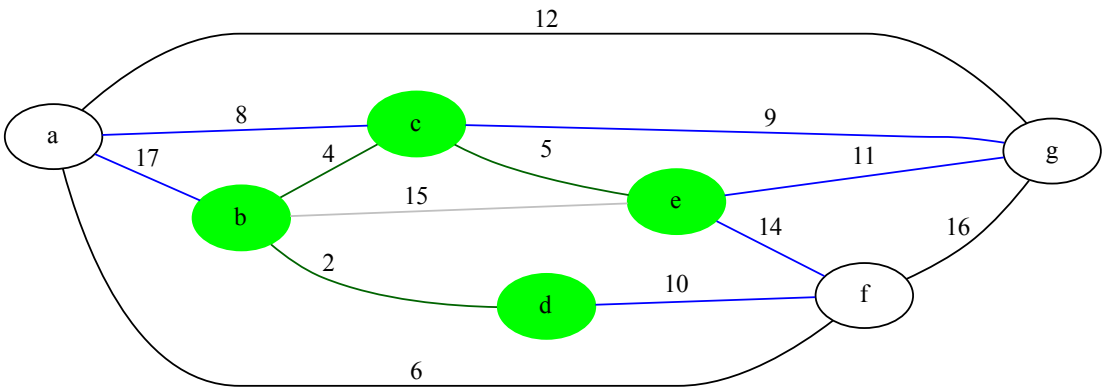
Beispiel: Prim-Algorithmus



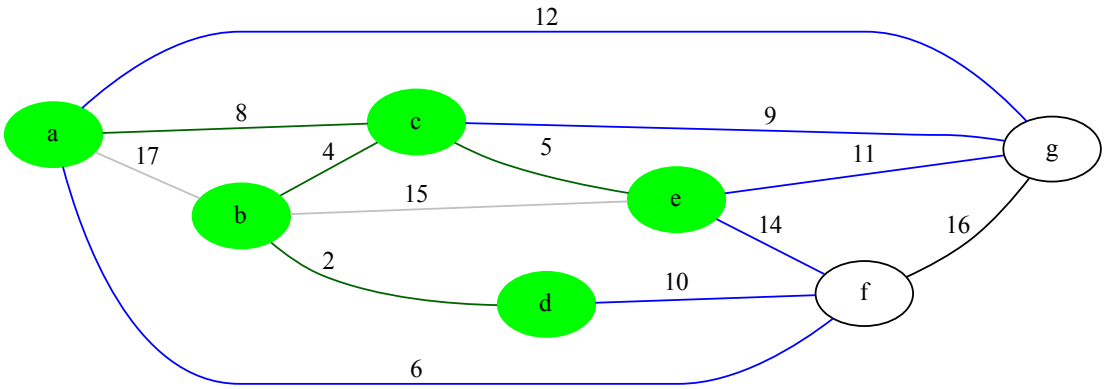
Beispiel: Prim-Algorithmus



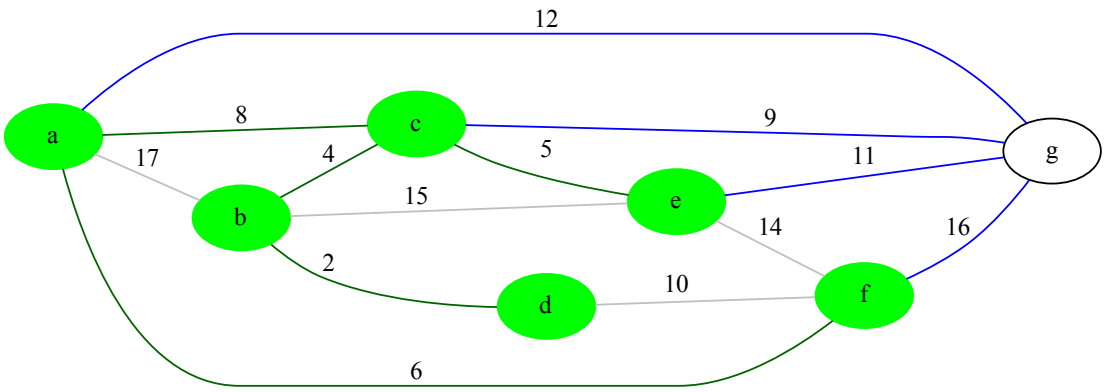
Beispiel: Prim-Algorithmus



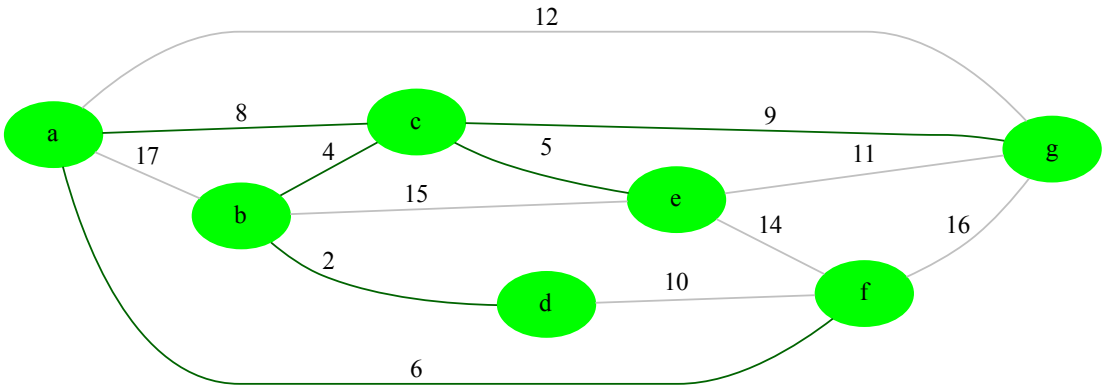
Beispiel: Prim-Algorithmus



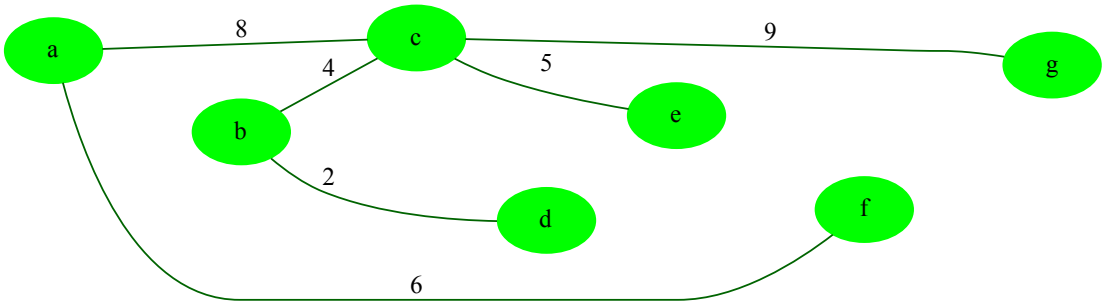
Beispiel: Prim-Algorithmus



Beispiel: Prim-Algorithmus

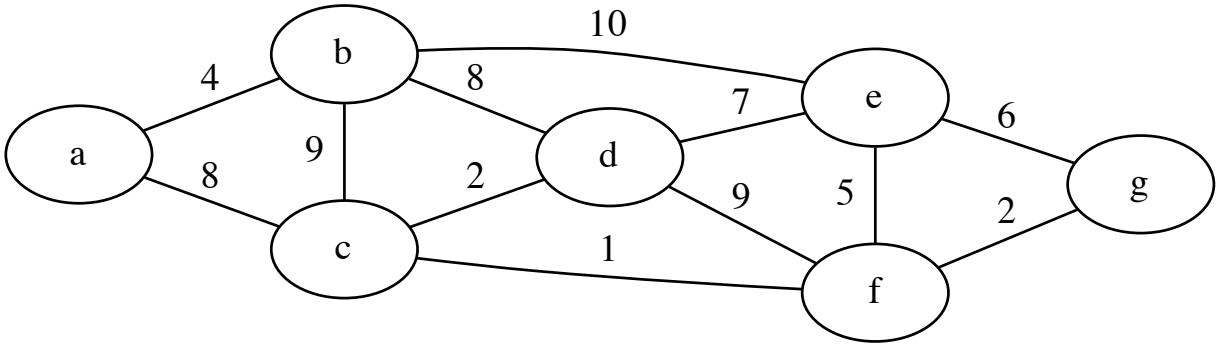


Beispiel: Prim-Algorithmus



Übung: Prim-Algorithmus

Bestimmen Sie einen minimalen Spannbaum für folgenden Graphen. Beginnen Sie mit einem zufälligen Knoten. Falls Sie genug Zeit haben, wiederholen Sie die Übung mit einen zweiten Startknoten. Was können Sie beobachten?



Gruppenübung: Komplexität des Prim-Algorithmus

Welche Komplexität hat der naive Prim-Algorithmus?

Eingabe: Graph $G = (V, E)$
Ausgabe: MST $T = (V, E_T)$

1. $E_T = \emptyset$
2. wähle v_{start}; $V_b = \{v_{start}\}$; $V_n = V \setminus \{v_{start}\}$
3. solange V_n Knoten enthält
   1. $e_n = (v_b, v_n)$ sei billigste Kante zwischen Knoten aus V_b und V_n
   2. nimm e_n zu E_T hinzu
   3. entferne v_n aus V_n
   4. nimm v_n zu V_b hinzu
4. gib (V, E_T) zurück

Gruppenübung: Komplexität des Prim-Algorithmus

Welche Komplexität hat der naive Prim-Algorithmus?

Eingabe: Graph $G = (V, E)$
Ausgabe: MST $T = (V, E_T)$

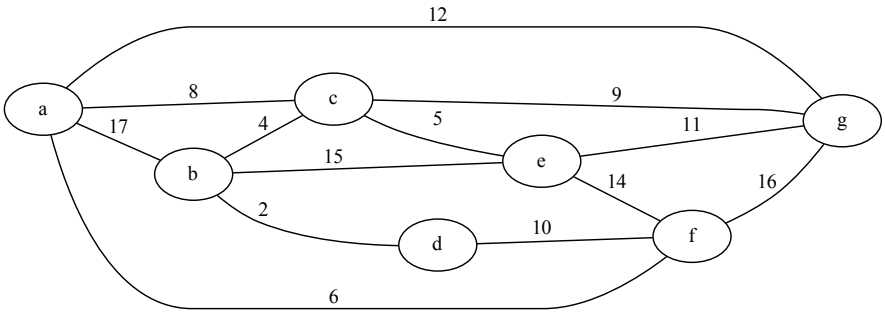
1. $E_T = \emptyset$
2. wähle v_{start}; $V_b = \{v_{start}\}$; $V_n = V \setminus \{v_{start}\}$
3. solange V_n Knoten enthält
   1. $e_n = (v_b, v_n)$ sei billigste Kante zwischen Knoten aus V_b und V_n
   2. nimm e_n zu E_T hinzu
   3. entferne v_n aus V_n
   4. nimm v_n zu V_b hinzu
4. gib (V, E_T) zurück

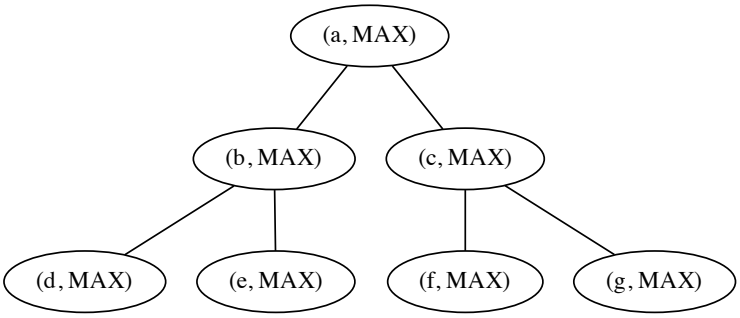
Ende Vorlesung 18

Optimierung des Prim-Algorithmus: Idee

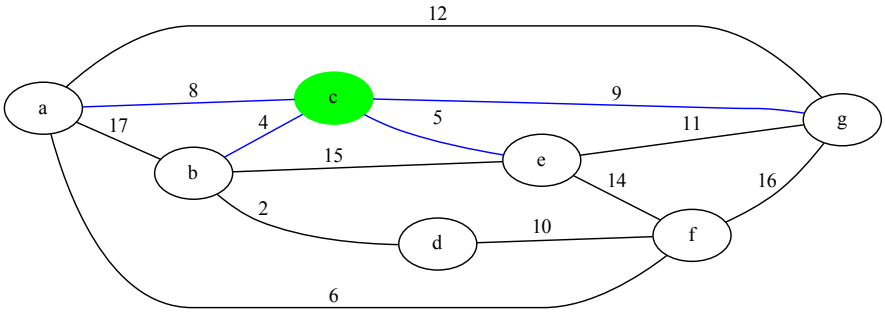
- E wird durch Adjazenzliste repräsentiert
- organisiere V_n als Min-Heap
  - Elemente: Knoten
  - Gewicht: Kosten der billigsten Kante zu einem Knoten aus V_b
- entferne der Reihe nach Knoten mit minimalem Gewicht
  - anschließend: Bubble-down der neuen Wurzel
- nach jedem neuen Knoten v_n: Update der Gewichte der mit v_n direkt verbundenen Knoten
  - ggf. bubble-up der Knoten

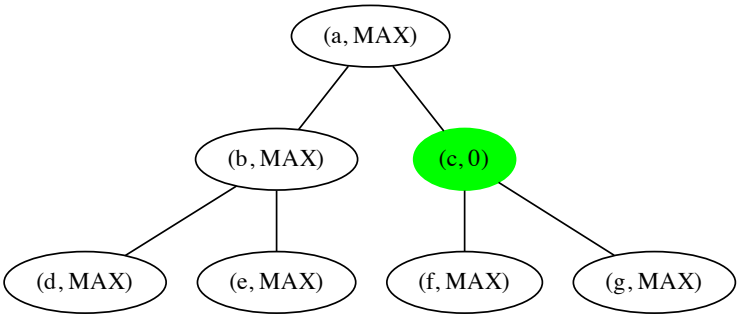
Beispiel: Prim-Algorithmus mit Heap



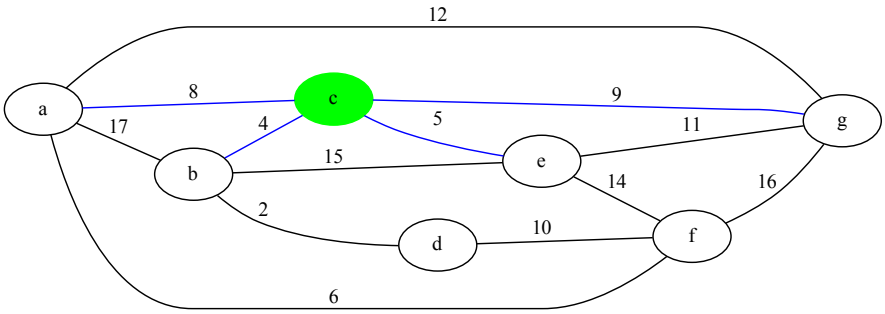


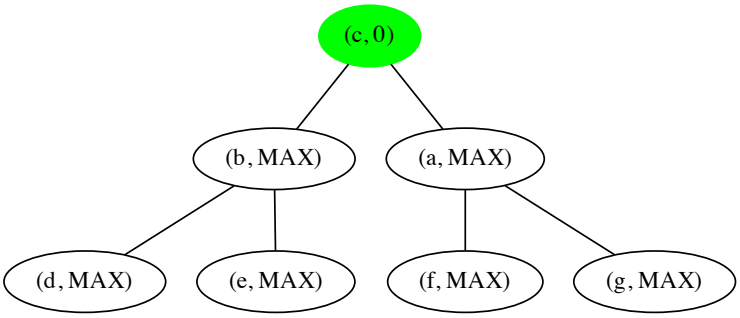
Beispiel: Prim-Algorithmus mit Heap



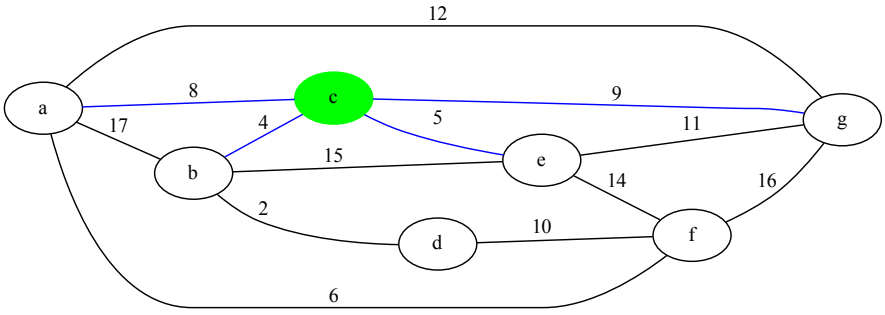


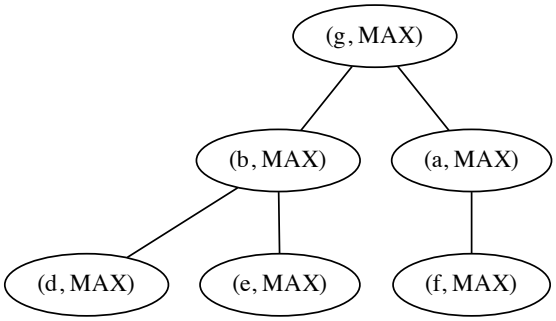
Beispiel: Prim-Algorithmus mit Heap



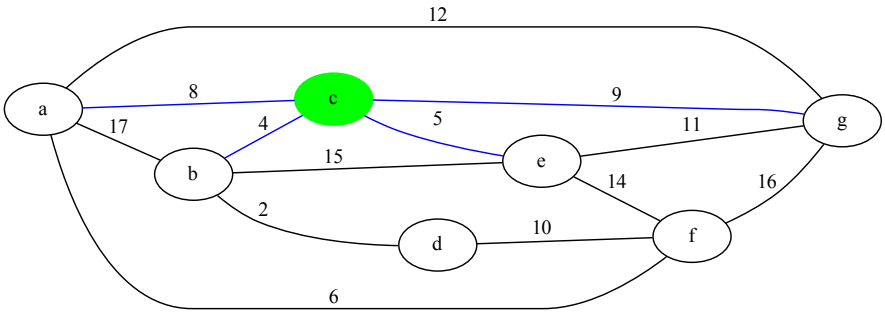


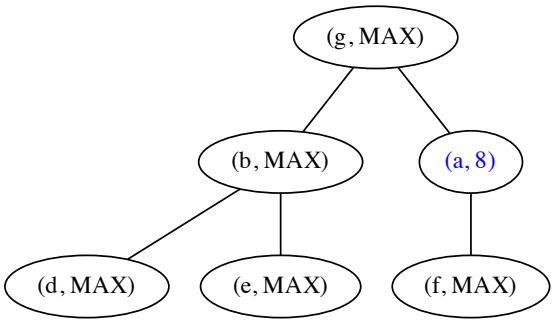
Beispiel: Prim-Algorithmus mit Heap



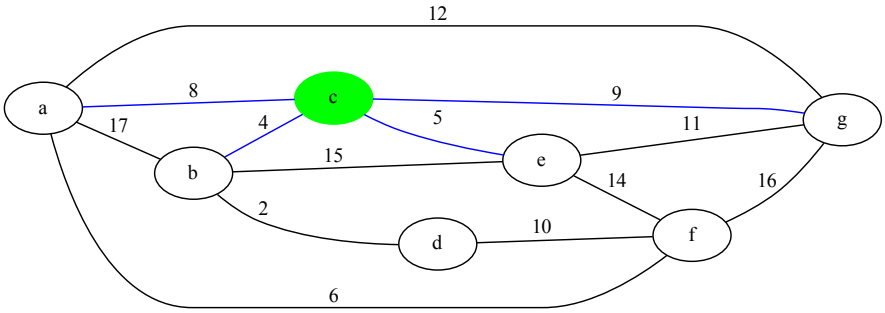


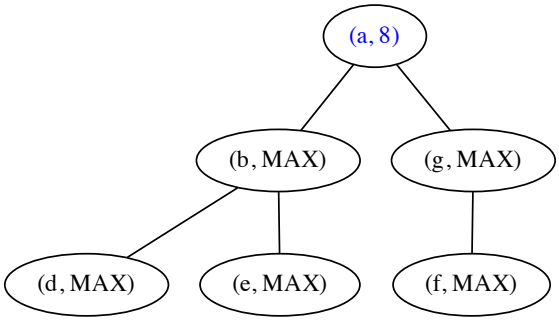
Beispiel: Prim-Algorithmus mit Heap



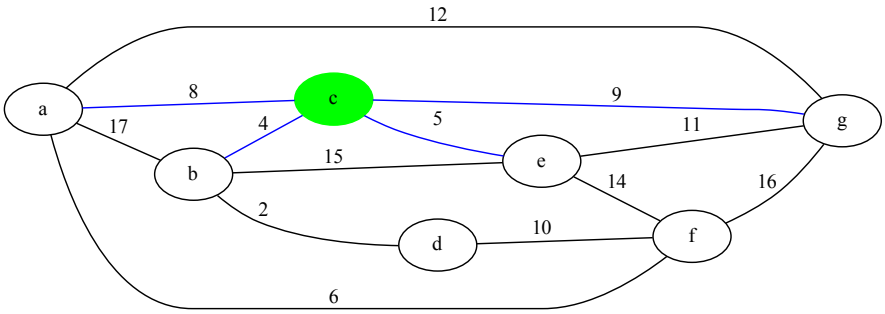


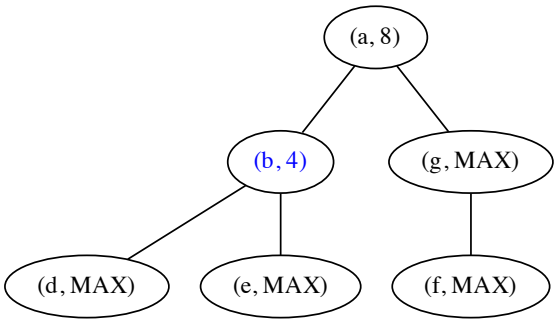
Beispiel: Prim-Algorithmus mit Heap



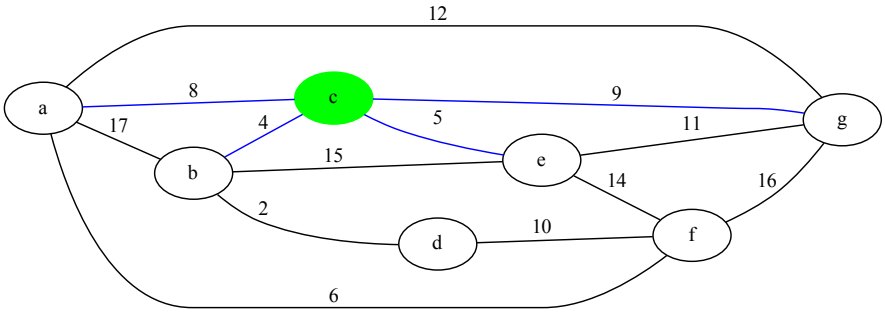


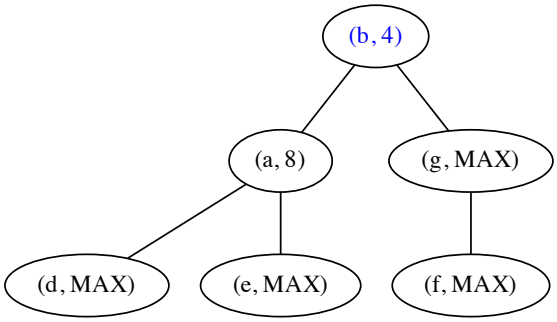
Beispiel: Prim-Algorithmus mit Heap



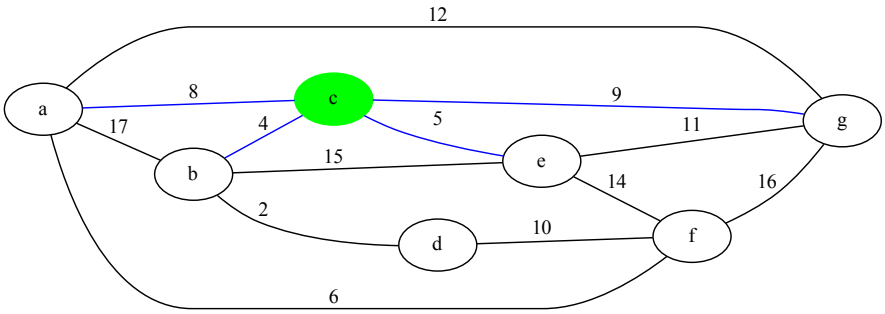


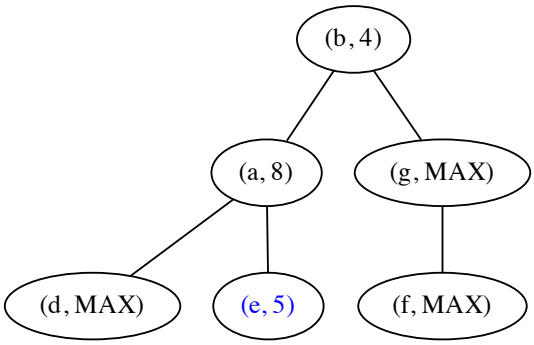
Beispiel: Prim-Algorithmus mit Heap



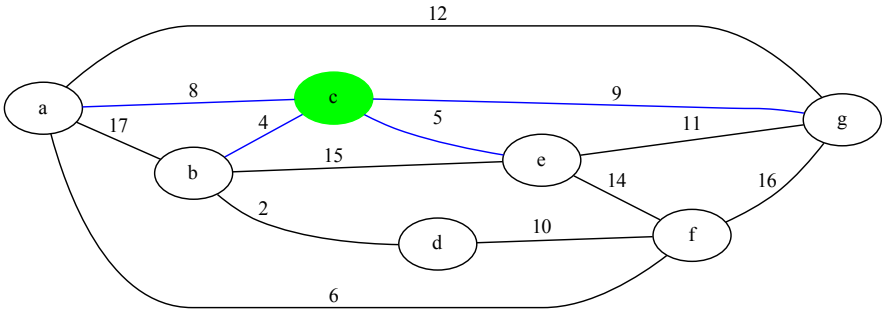


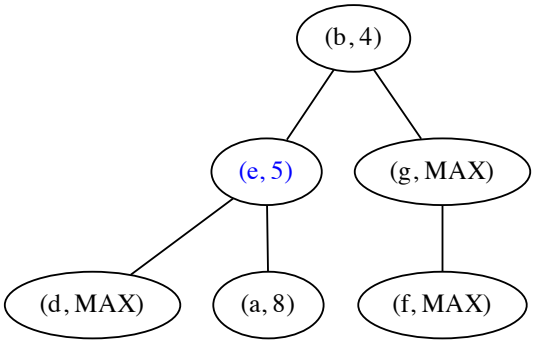
Beispiel: Prim-Algorithmus mit Heap



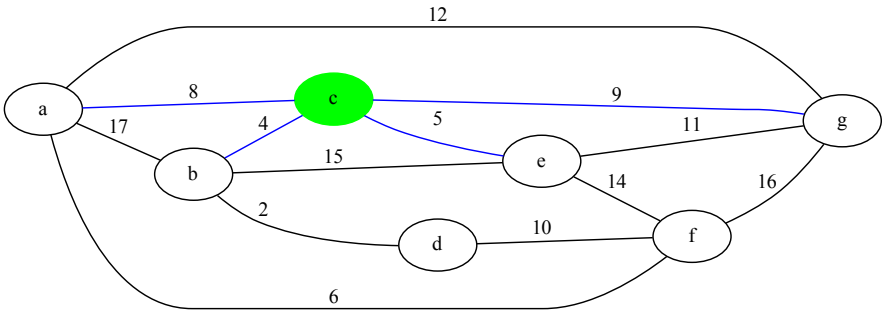


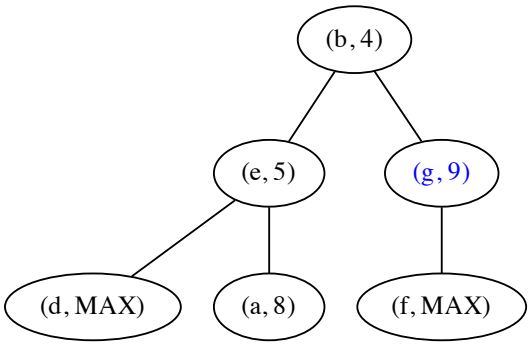
Beispiel: Prim-Algorithmus mit Heap



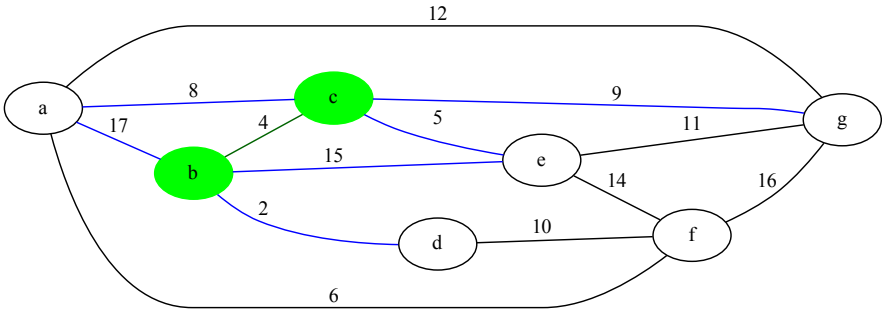


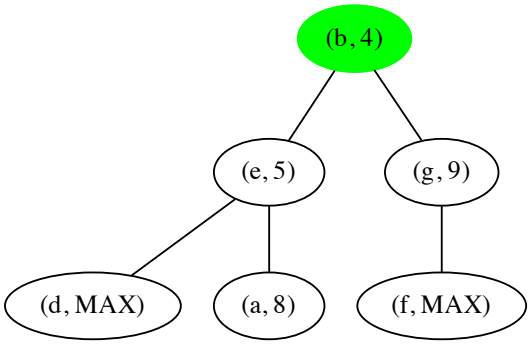
Beispiel: Prim-Algorithmus mit Heap



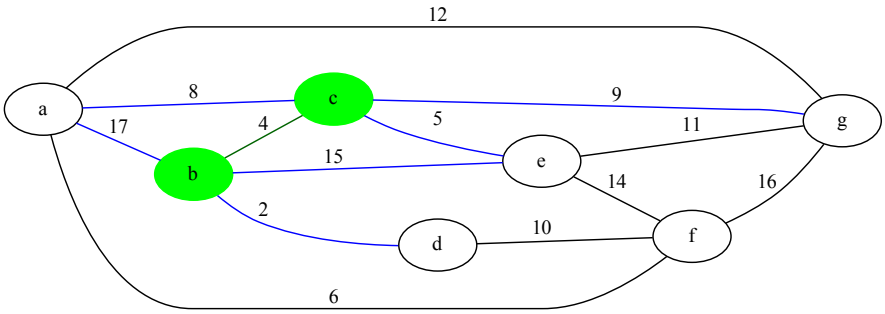


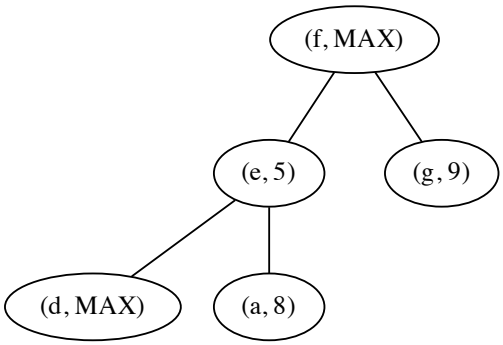
Beispiel: Prim-Algorithmus mit Heap



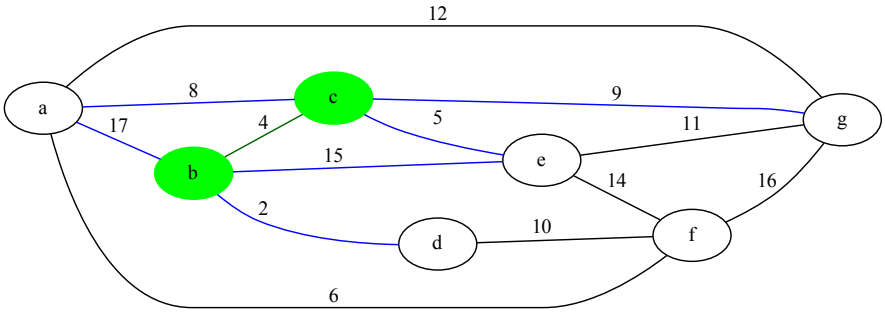


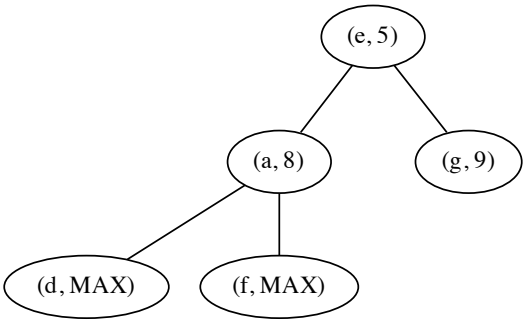
Beispiel: Prim-Algorithmus mit Heap



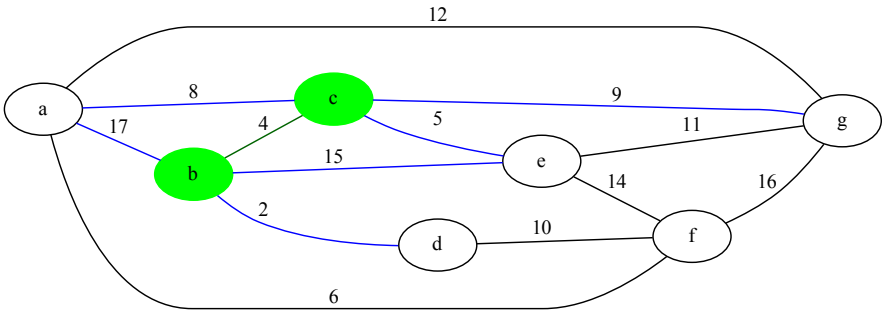


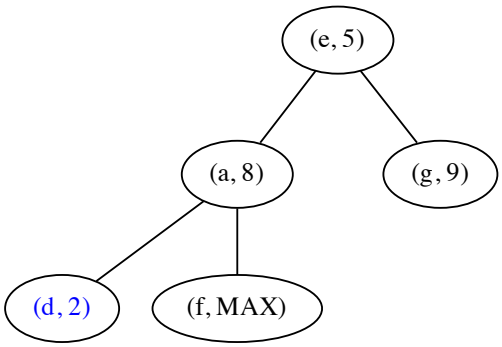
Beispiel: Prim-Algorithmus mit Heap



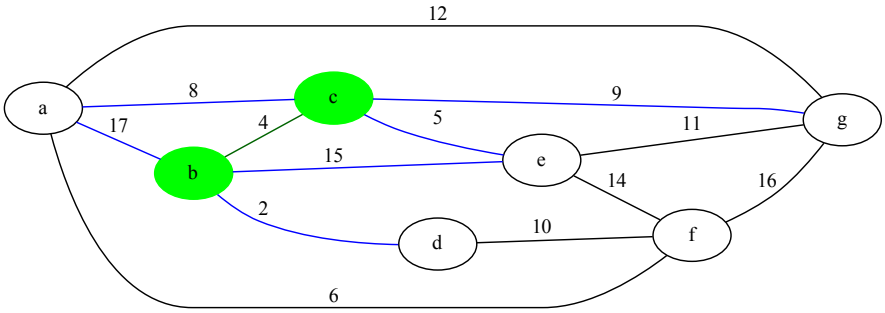


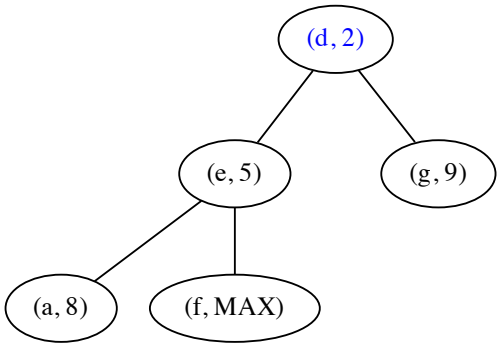
Beispiel: Prim-Algorithmus mit Heap



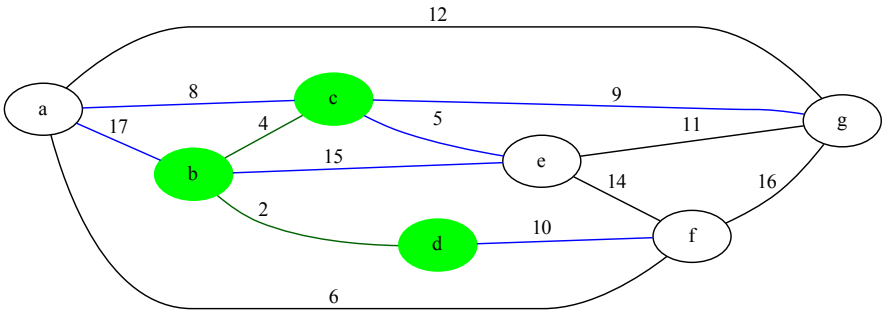


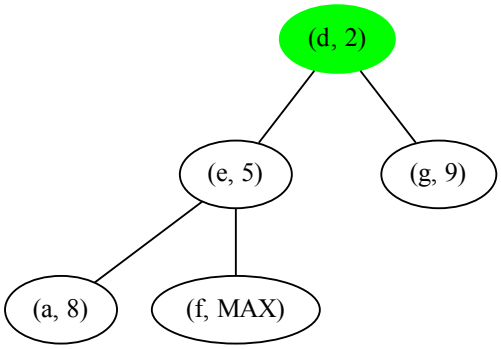
Beispiel: Prim-Algorithmus mit Heap



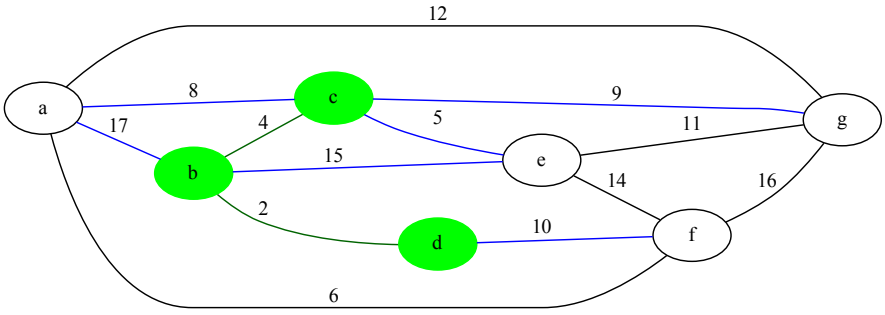


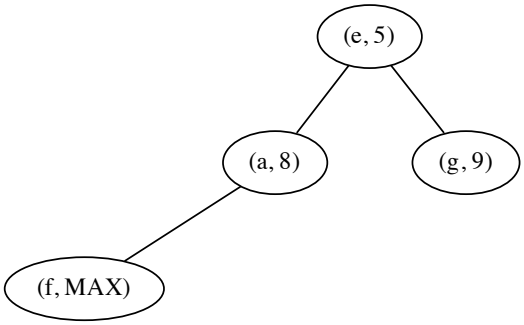
Beispiel: Prim-Algorithmus mit Heap



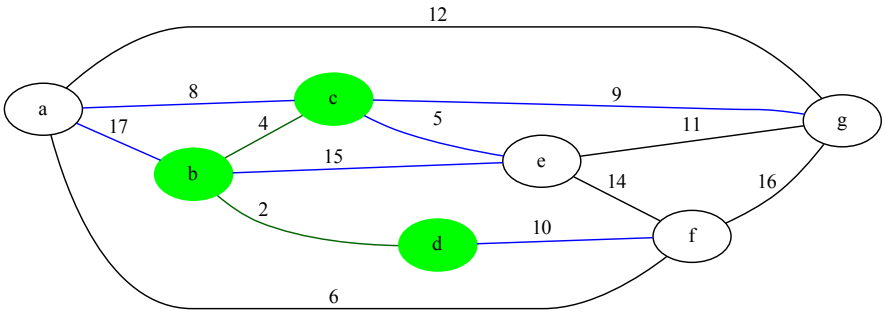


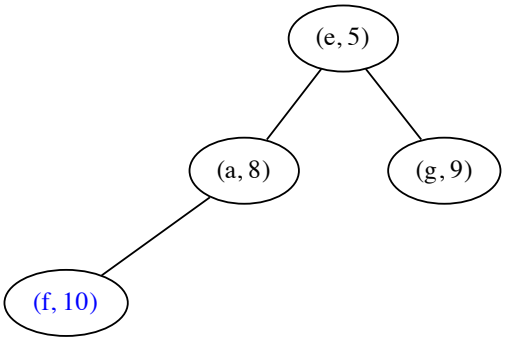
Beispiel: Prim-Algorithmus mit Heap



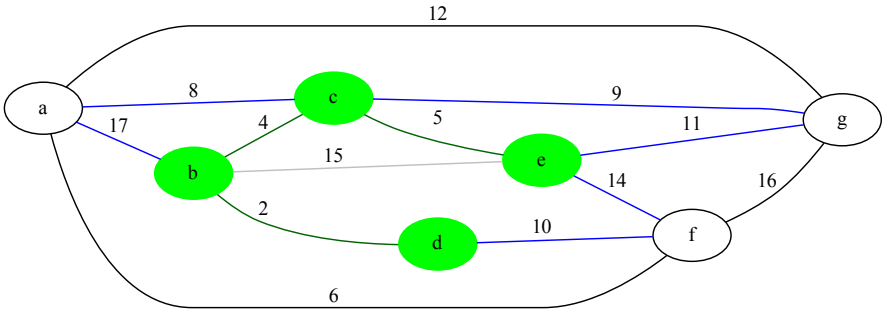


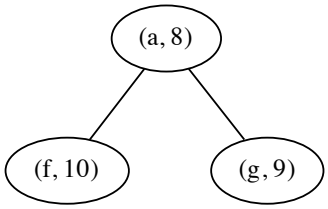
Beispiel: Prim-Algorithmus mit Heap



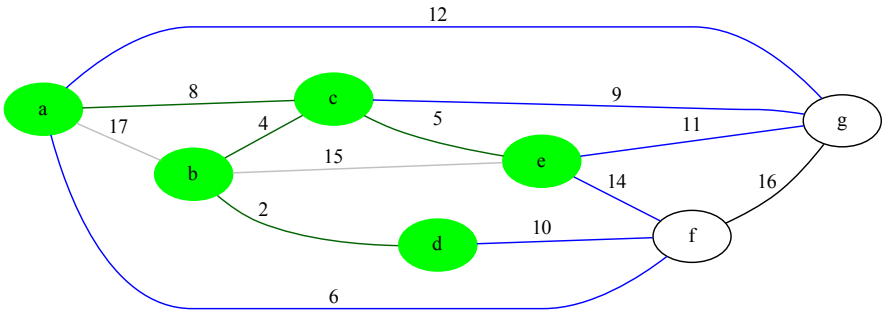


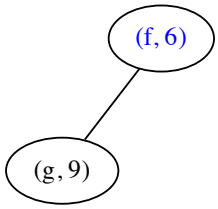
Beispiel: Prim-Algorithmus mit Heap



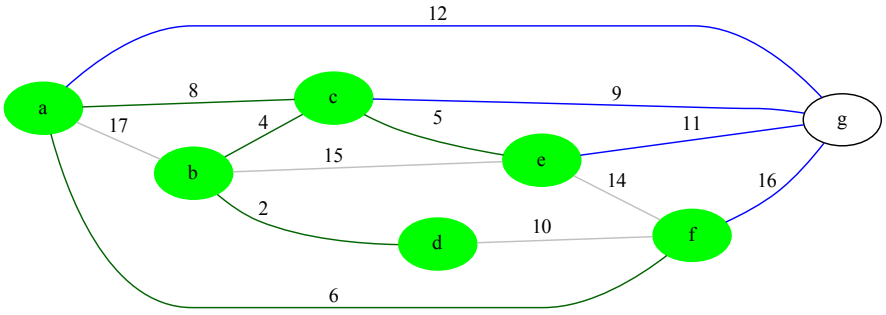


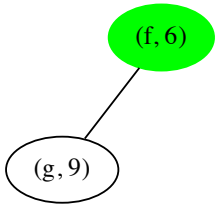
Beispiel: Prim-Algorithmus mit Heap



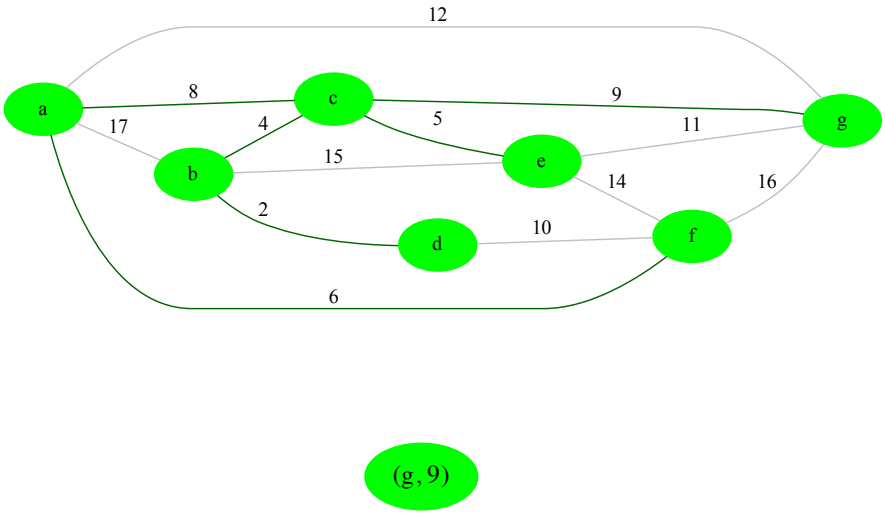


Beispiel: Prim-Algorithmus mit Heap





Beispiel: Prim-Algorithmus mit Heap



Komplexität des optimierten Prim-Algorithmus

- für jeden Knoten einmal bubble-down
  - $\mathcal{O}(|V| \log |V|)$
- für jede Kante einmal bubble-up
  - $\mathcal{O}(|E| \log |V|)$
  - jede Kante wird nur einmal betrachtet
  - Kanten möglicherweise ungleich verteilt über Knoten
  - wichtig: Adjazenzliste, bei Matrix: $|V|^2 \log |V|$
- verbundener Graph $\rightsquigarrow |V| \leq |E| - 1$
  - Gesamtkomplexität: $\mathcal{O}(|E| \log |V|)$ statt $\mathcal{O}(|E| \cdot |V|)$

Routing: Der Dijkstra-Algorithmus

- Problem: Finde (garantiert) den kürzesten/günstigsten Weg von A nach B
- Anwendungen:
  - Luftverkehr
  - Straßenverkehr
  - Routing von Paketen (UPS, DHL)
  - Routing von Paketen (Cisco, DE-CIX)
  - ...

Routing: Der Dijkstra-Algorithmus

- Problem: Finde (garantiert) den kürzesten/günstigsten Weg von A nach B
- Anwendungen:
  - Luftverkehr
  - Straßenverkehr
  - Routing von Paketen (UPS, DHL)
  - Routing von Paketen (Cisco, DE-CIX)
  - ...
- Grundlage: Gewichteter Graph (V, E)
  - V sind die besuchbaren/passierbaren Orte
  - E sind die Verbindungen
  - $e : E \to \mathbb{N}$ sind die Kosten einer Einzelverbindung

Routing: Der Dijkstra-Algorithmus

- Problem: Finde (garantiert) den kürzesten/günstigsten Weg von A nach B
- Anwendungen:
  - Luftverkehr
  - Straßenverkehr
  - Routing von Paketen (UPS, DHL)
  - Routing von Paketen (Cisco, DE-CIX)
  - ...
- Grundlage: Gewichteter Graph (V, E)
  - V sind die besuchbaren/passierbaren Orte
  - E sind die Verbindungen
  - $e : E \to \mathbb{N}$ sind die Kosten einer Einzelverbindung
- Besonderheit: Wir finden den kürzesten Weg von A zu jedem anderen Ziel (das ist im Worst-Case nicht schwieriger)

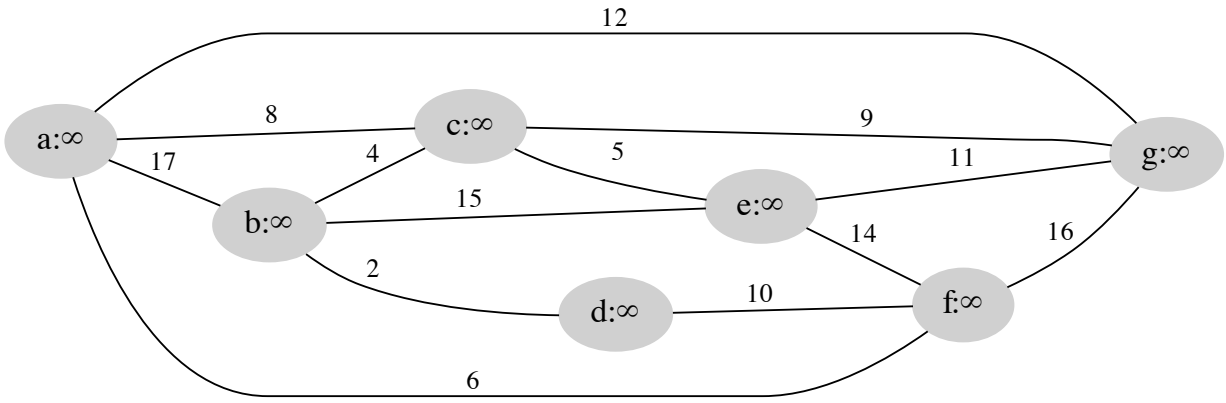
Edsger W. Dijkstra (1930-2002)
Turing-Award 1972
"Goto Considered Harmful", 1968

Dijkstra: Idee

Eingabe: Graph (V, E), Kantengewichtsfunktion e, Startknoten v_s
Ausgabe: Funktion $d : V \to \mathbb{N}$ mit Entfernungen von v_s
Variablen: Liste der besuchten Knoten B, aktueller Knoten v_c

1. Setze $d(v_s) = 0$, $d(v_i) = \infty$ für alle $v_i \neq v_s$, $B = \emptyset$
2. Setze $v_c = v_s$
3. Für alle Nachbarn v von v_c:
   1. Berechne $d_{tmp} = d(v_c) + e(v_c, v)$
   2. Wenn $d_{tmp} < d(v)$ gilt, setze $d(v) = d_{tmp}$
4. Füge v_c zu B hinzu
5. Wenn $B = V$ gilt: fertig
   (oder wenn für alle Knoten $v \in V \setminus B$ gilt: $d(v) = \infty$)
6. Sonst: Wähle als neuen v_c den Knoten aus $V \setminus B$ mit geringster Entfernung
7. Fahre bei 3 fort

Dijkstra: Beispiel

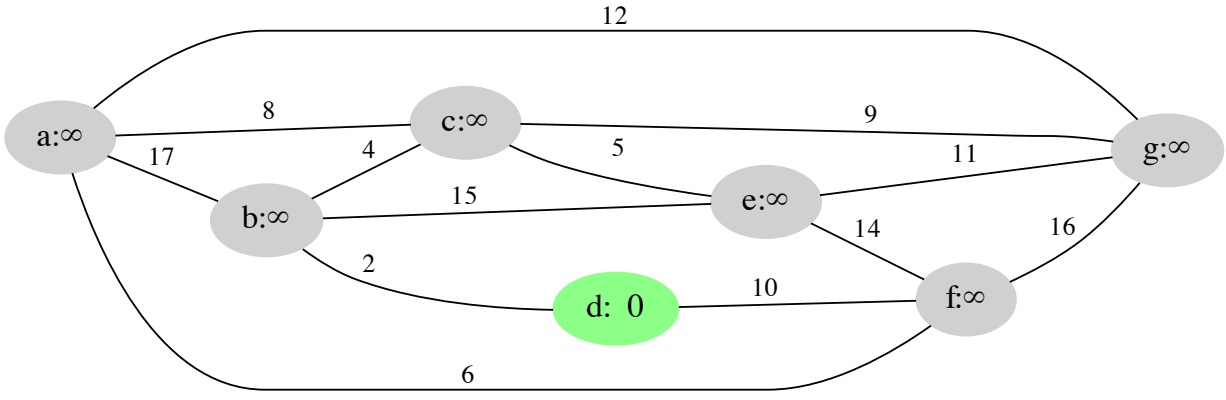


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

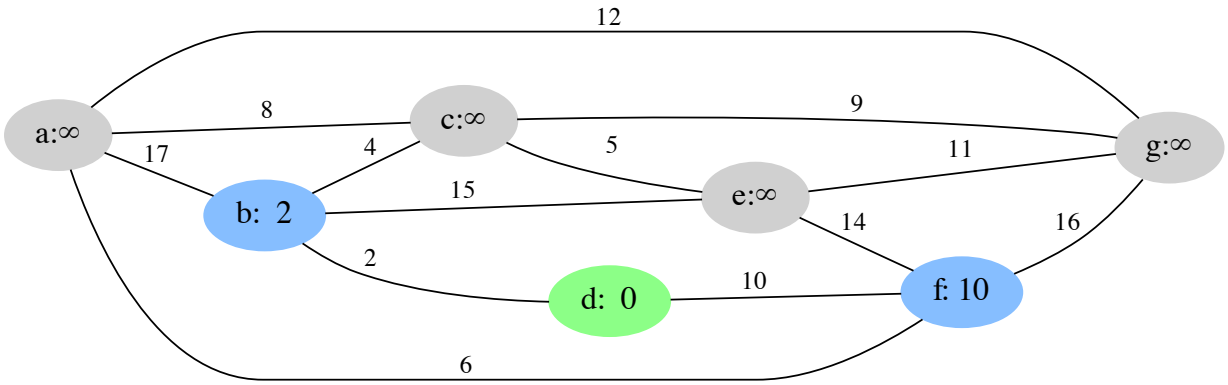


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

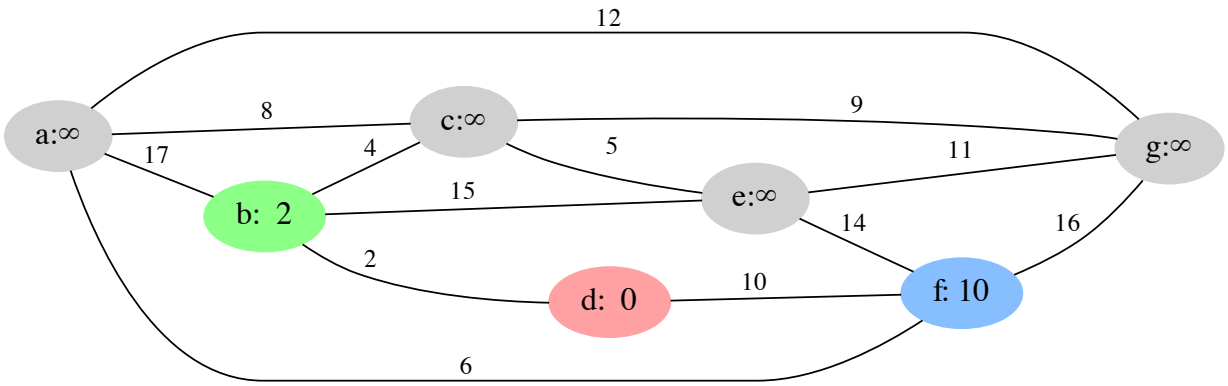


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

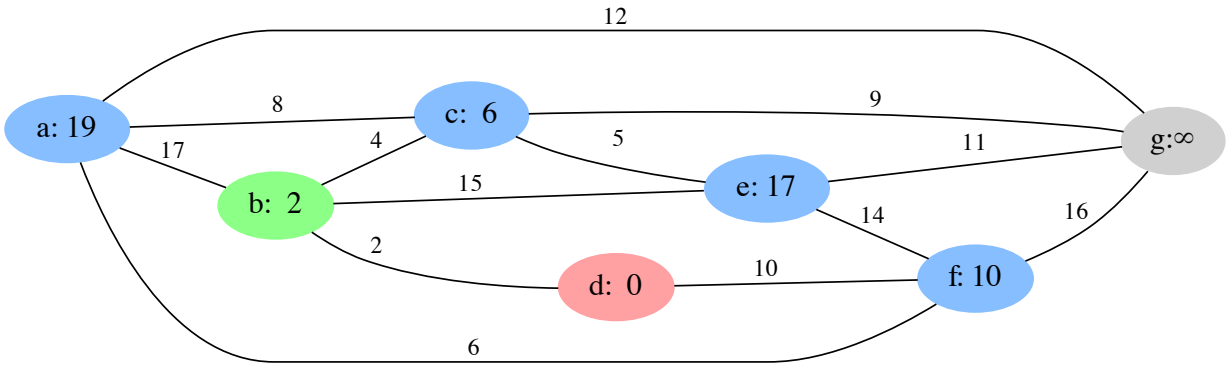


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

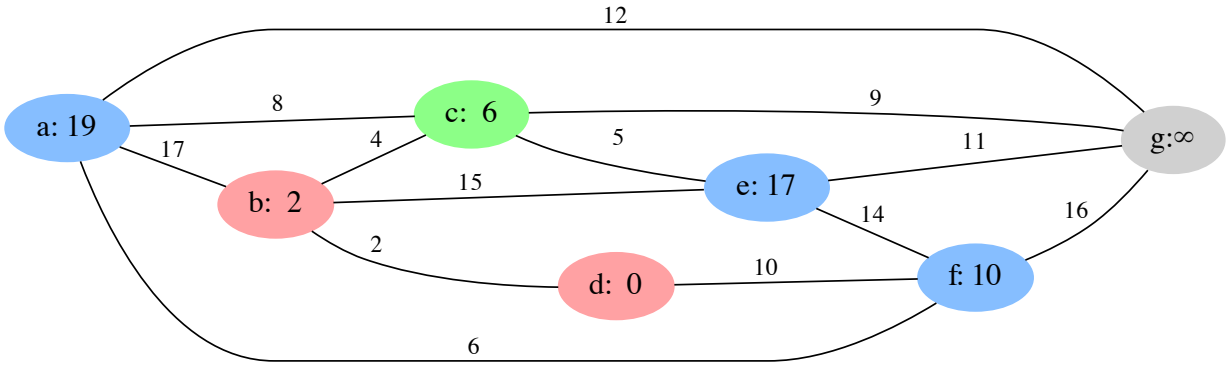


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

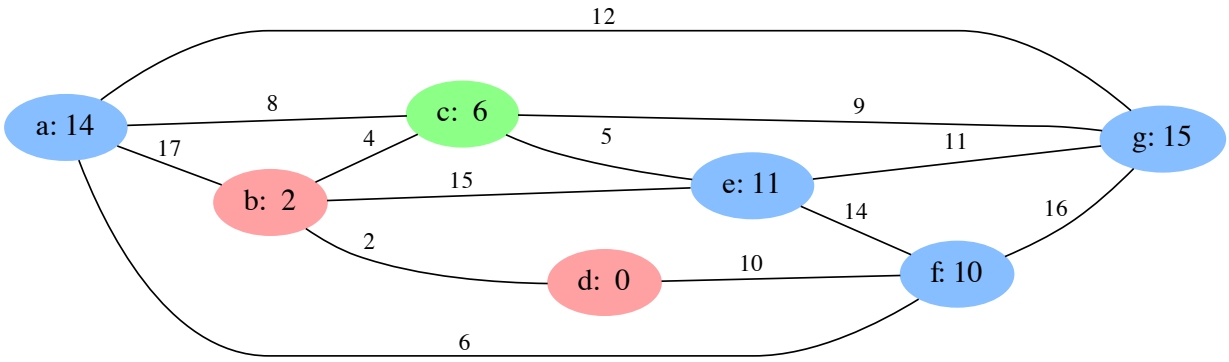


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

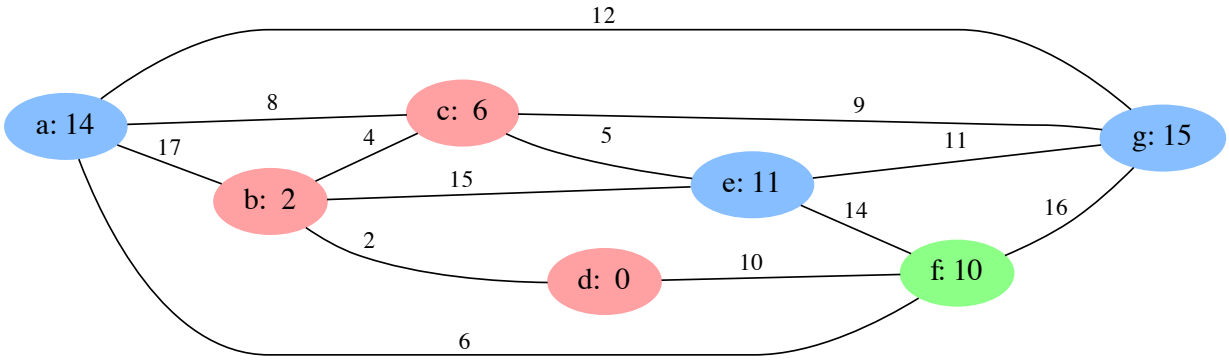


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

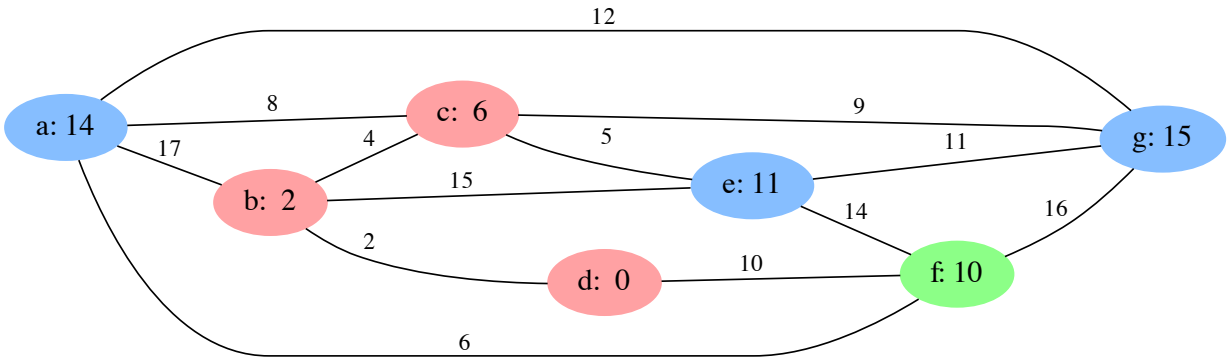


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

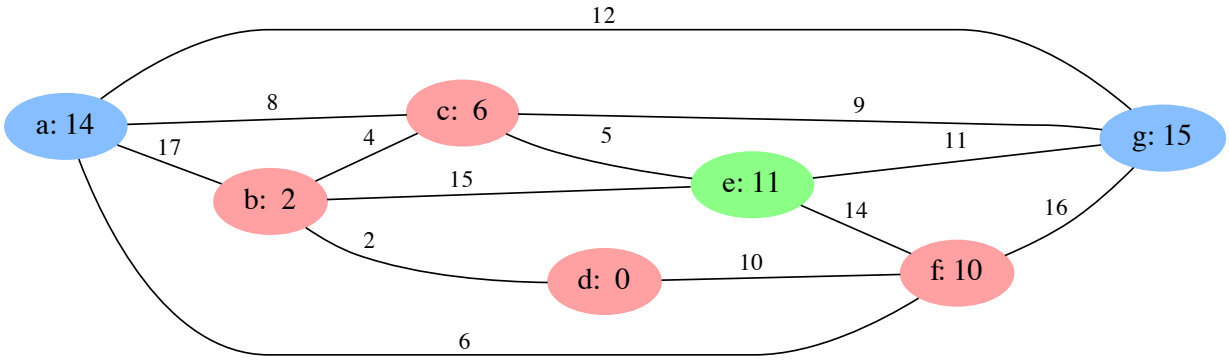


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

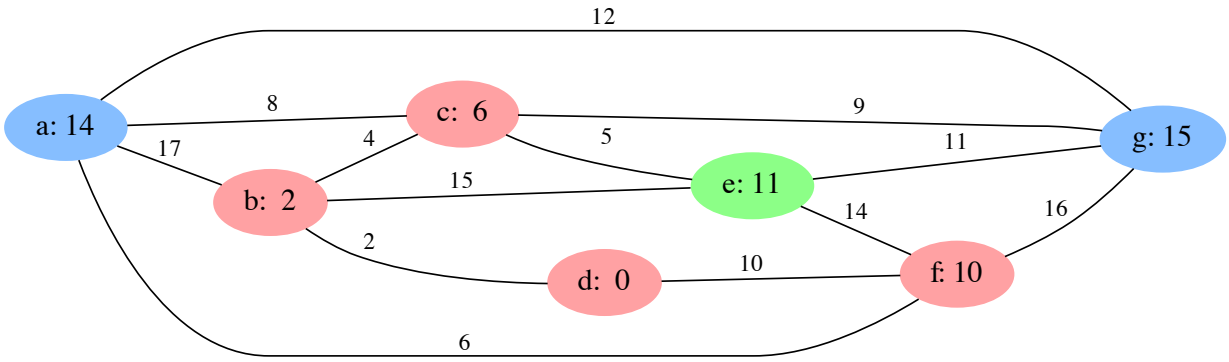


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

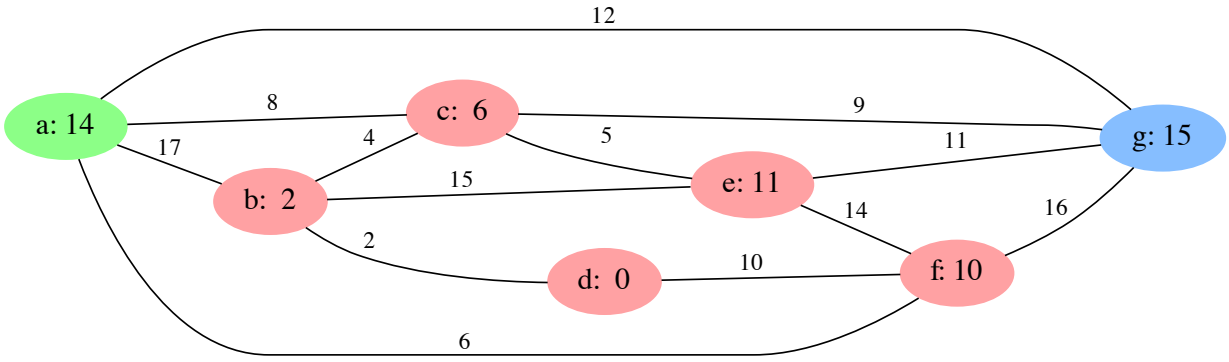


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

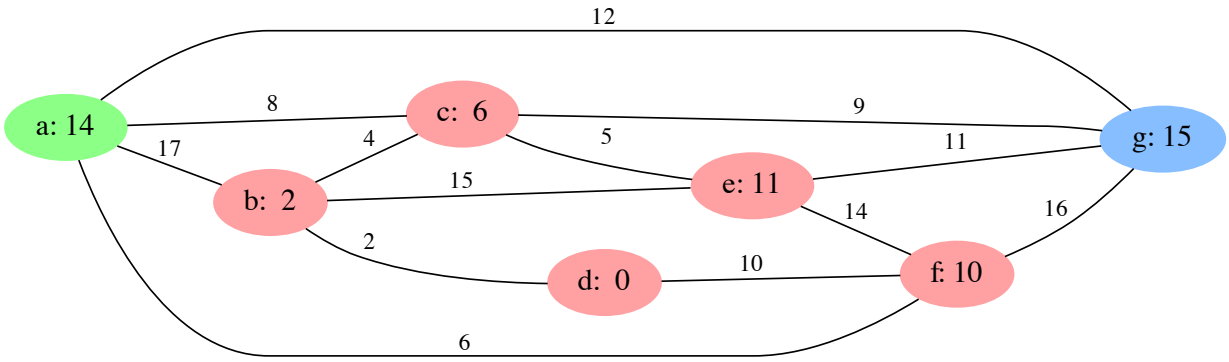


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

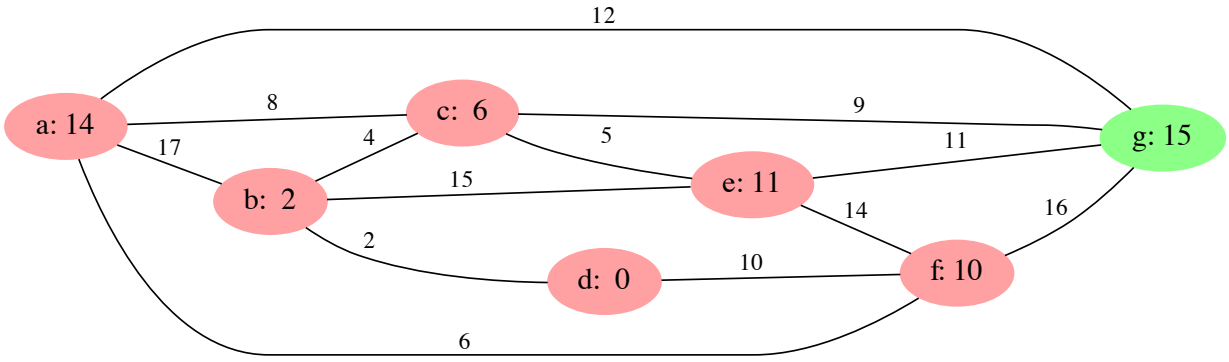


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

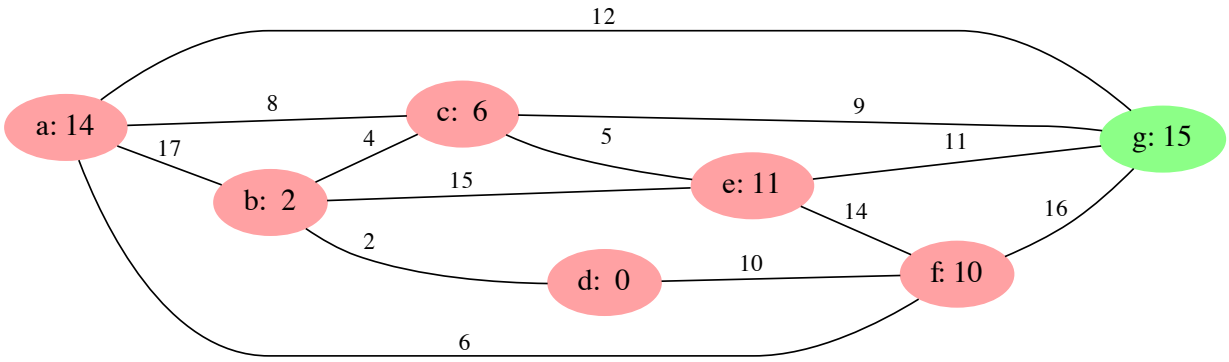


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel

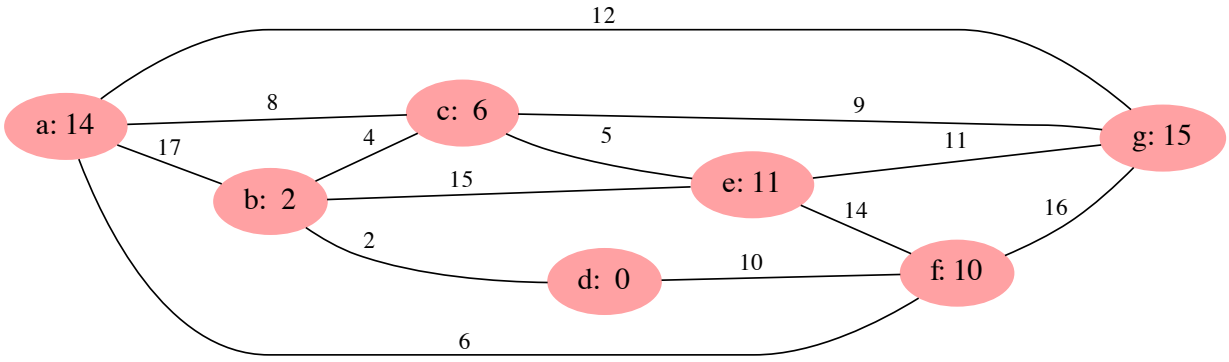


Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

Dijkstra: Beispiel



Farbschema

Normaler Knoten	Aktueller Knoten v_c
Gewichtet	Besucht/Fertig

Ende Vorlesung 19

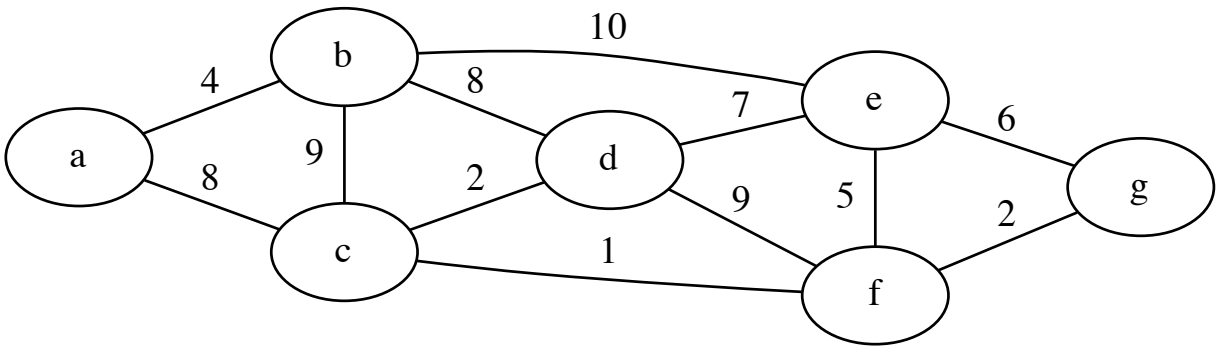
Dijkstra Implementiert

```
def dijkstra(graph, start):
    vc = graph.node(start)
    vc.updateDist(0)
    while vc:
        for n,w in vc.adj_list:
            node = graph.node(n)
            node.updateDist(vc.dist+w)
        vc.setVisited()
        vc = None
        for n in graph.nodes.values():
            if not n.visited:
                if n.dist!=Infty:
                    if not vc or n.dist < vc.dist:
                        vc = n
```

- `vc`: Aktueller Knoten
- `n`: Knotenname,
- `w`: Gewicht der aktuellen Kante
- `node`: Knotendatenstruktur
  - `updateDist()`
  - `dist` - geschätzte Distance
  - `adj_list`
  - `setVisited()`
  - `visited`

Übung: Dijkstra-Algorithmus

Bestimmen Sie die kürzesten Entfernungen vom Knoten *c* im folgenden Graphen.



Dijkstra: Komplexität

- jeden Knoten besuchen: $\mathcal{O}(|V|)$
  - alle Nachbarn updaten: maximal $\mathcal{O}(|V|)$ (insgesamt maximal $\mathcal{O}(|E|)$)
  - nächsten Knoten finden: $\mathcal{O}(|V|)$
- Naiv: insgesamt: $\mathcal{O}(|V|^2)$
- Mit Heap und Adjazenzlisten geht es besser!

Gruppenübung: Dijkstra optimaler

- Bei naiver Implementierung: $\mathcal{O}(|V|^2)$
- Erreichbar (mit Mitteln dieser Vorlesung): $\mathcal{O}(|E| \log |V|)$
- Frage: Wie?

Greedy-Algorithmen

- Prim und Dijkstra sind Greedy-Algorithmen
- Greedy: Wähle lokal beste Lösung
  - billigsten neuen Knoten
  - kürzesten Weg zum Nachbarknoten
- liefert globales Optimum, weil dieses sich aus lokalen Optima zusammensetzt
- Gegenbeispiele
  - Rucksack-Problem: 5 Liter Platz im Rucksack; Wasserbehälter mit 2,3,4 Liter vorhanden
  - Problem des Handlungsreisenden: alle Städte (Knoten des Graphen) besuchen

Greedy-Algorithmen

- Prim und Dijkstra sind Greedy-Algorithmen
- Greedy: Wähle lokal beste Lösung
  - billigsten neuen Knoten
  - kürzesten Weg zum Nachbarknoten
- liefert globales Optimum, weil dieses sich aus lokalen Optima zusammensetzt
- Gegenbeispiele
  - Rucksack-Problem: 5 Liter Platz im Rucksack; Wasserbehälter mit 2,3,4 Liter vorhanden
  - Problem des Handlungsreisenden: alle Städte (Knoten des Graphen) besuchen

Ende Vorlesung 20

Ende Material / Anfang Einzelvorlesungen

Ziele Vorlesung 1

- Kennenlernen (oder Wiedererkennen)
- Übersicht über die Vorlesung
- Was ist ein Algorithmus?
- Beispiel: Euklid

Vorstellung

- Stephan Schulz
  - Dipl.-Inform., U. Kaiserslautern, 1995
  - Dr. rer. nat., TU München, 2000
  - Visiting professor, U. Miami, 2002
  - Visiting professor, U. West Indies, 2005
  - Gastdozent (Hildesheim, Offenburg, . . .) seit 2009
  - Praktische Erfahrung: Entwicklung von Flugsicherungssystemen
    - System engineer, 2005
    - Project manager, 2007
    - Product Manager, 2013
  - Professor, DHBW Stuttgart, 2014

Vorstellung

- Stephan Schulz
  - Dipl.-Inform., U. Kaiserslautern, 1995
  - Dr. rer. nat., TU München, 2000
  - Visiting professor, U. Miami, 2002
  - Visiting professor, U. West Indies, 2005
  - Gastdozent (Hildesheim, Offenburg, . . .) seit 2009
  - Praktische Erfahrung: Entwicklung von Flugsicherungssystemen
    - System engineer, 2005
    - Project manager, 2007
    - Product Manager, 2013
  - Professor, DHBW Stuttgart, 2014

Research: Logik & Deduktion

233

Introduction

- Jan Hladik
  - Dipl.-Inform.: RWTH Aachen, 2001
  - Dr. rer. nat.: TU Dresden, 2007
  - Industrieerfahrung: SAP Research
    - Öffentlich geförderte Forschungsprojekte
    - Zusammenarbeit mit SAP-Produktgruppen
    - Betreuung von Bachelor- und Master-Studenten, Doktoranden
  - Professor: DHBW Stuttgart, 2014

Introduction

- Jan Hladik
  - Dipl.-Inform.: RWTH Aachen, 2001
  - Dr. rer. nat.: TU Dresden, 2007
  - Industrieerfahrung: SAP Research
    - Öffentlich geförderte Forschungsprojekte
    - Zusammenarbeit mit SAP-Produktgruppen
    - Betreuung von Bachelor- und Master-Studenten, Doktoranden
  - Professor: DHBW Stuttgart, 2014

Forschung: Semantic Web, Semantische Technologien, Schlussfolgerungsverfahren

Zusammenfassung

- Kennenlernen (oder Wiedererkennen)
- Übersicht über die Vorlesung
- Was ist ein Algorithmus?
- Beispiel: Euklid

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 2

- Rückblick/Wiederholung
- Kurz: Ganzzahldivision und Modulus (Divisionsrest)
- Euklid (zweite Runde)
- Algorithmen und Datenstrukturen
- Effizienz und Komplexität von Algorithmen

Rückblick/Wiederholung

- Algorithmenbegriff
- Beispiel von Algorithmenklassen
  - Suchen/Sortieren
  - Optimieren
  - Kompression
  - ...
- Spezifikation von Algorithmem
  - Informal
  - Semi-Formal
  - (Pseudo-)Code
  - Flussdiagramme
  - ...
- Der GGT-Algorithmus von Euklid

Zur Vorlesung 2

Zusammenfassung

- Rückblick/Wiederholung
- Kurz: Ganzzahldivision und Modulus (Divisionsrest)
- Euklid (zweite Runde)
- Algorithmen und Datenstrukturen
- Effizienz und Komplexität von Algorithmen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 3

- Komplexität konkret
  - Was zählen wir?
  - Wovon abstrahieren wir?
- *Big-$\mathcal{O}$ Notation*
  - Definition

Rückblick/Wiederholung

- Kurz: Ganzzahldivision und Modulus (Divisionsrest)
- Euklid (zweite Runde)
- Algorithmen und Datenstrukturen
  - Namen unsortiert, sortiert, ...
- Effizienz und Komplexität von Algorithmen
  - Zeit und Platz
  - *Average case* und *Worst case*

Zur Vorlesung 3

Zusammenfassung

- Komplexität konkret
  - Was zählen wir?
  - Wovon abstrahieren wir?
- *Big-$\mathcal{O}$ Notation*
  - Definition

Feedback

- Was war der beste Teil der heutigen Vorlesung? v
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 4

- *Big-$\mathcal{O}$ Notation*
  - Definition
  - Rechenregeln
  - Beispiele

Rückblick/Wiederholung

- Komplexität konkret
  - Was zählen wir?
  - Wovon abstrahieren wir?
- *Big-$\mathcal{O}$ Notation*
  - Definition

Rückblick/Wiederholung

- Komplexität konkret
  - Was zählen wir?
  - Wovon abstrahieren wir?
- *Big-$\mathcal{O}$ Notation*
  - Definition

$\mathcal{O}$-Notation

Für eine Funktion f bezeichnet $\mathcal{O}(f)$ die Menge aller Funktionen g mit

$$\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$$

Zur Vorlesung 4

Zusammenfassung

- *Big-$\mathcal{O}$ Notation*
  - Definition
  - Rechenregeln
  - Beispiele

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 5

- Rückblick und offene Fragen
- Beispiele für Komplexitäten
- Algorithmenansätze und Komplexität
  - Iteration
  - Rekursion

Rückblick/Wiederholung

- *Big-$\mathcal{O}$ Notation*
  - Für eine Funktion f bezeichnet $\mathcal{O}(f)$ die Menge aller Funktionen g mit $\exists k \in \mathbb{N} \quad \exists c \in \mathbb{R}^{\geq 0} \quad \forall n > k : g(n) \leq c \cdot f(n)$
- Rechenregeln
  - $f \in \mathcal{O}(f)$
  - Konstanter Faktor
  - Summe
  - Transitivität
  - Grenzwert!
- Regel von l'Hôpital
- Beispiele
  - Offen: n^n vs. $n \cdot 2^n$

Zur Vorlesung 5

Zusammenfassung

- Rückblick und offene Fragen
- Beispiele für Komplexitäten
- Algorithmenansätze und Komplexität
  - Iteration
  - Rekursion

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 6

- Rückblick
- Elegante und schnelle Algorithmen: Dynamische Programmierung
  - Beispiele
  - Voraussetzungen
  - Grenzen

Rückblick/Wiederholung

- Analyse: n^n gegen $n \cdot 2^n$
- Stirlingsche Formel, Folgerung: $n!$ steigt schneller als $e^{c \cdot n}$ für beliebiges c
- Komplexitätsklassen
  - $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), \ldots, O(2^n) \ldots$
- Übung/Hausaufgabe: Fibonacci-Zahlen

Rückblick/Wiederholung

- Analyse: n^n gegen $n \cdot 2^n$
- Stirlingsche Formel, Folgerung: $n!$ steigt schneller als $e^{c \cdot n}$ für beliebiges c
- Komplexitätsklassen
  - $O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), \ldots, O(2^n) \ldots$
- Übung/Hausaufgabe: Fibonacci-Zahlen
  - Naiv rekursiv: Einfach, offensichtlich korrekt, $O(2^n)$
  - Iterativ: Komlizierter, aber $O(n)$

Zur Vorlesung 6

Zusammenfassung

- Rückblick
- Elegante und schnelle Algorithmen: Dynamische Programmierung
  - Beispiele
  - Voraussetzungen
  - Grenzen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 7

- Rückblick
- Komplexität rekursiver Programme
  - Rekurrenzrelationen
  - Das Master-Theorem

Rückblick/Wiederholung

- Dynamische Programmierung
  - Fibonacci naive rekursiv/iterative/DP
  - Eigenschaften von DP
  - Beispiel: Optimales Verkleinern nach Regeln

Zur Vorlesung 7

Zusammenfassung

- Rückblick
- Komplexität rekursiver Programme
  - Rekurrenzrelationen
  - Das Master-Theorem

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 8

- Rückblick
- Master-Theorem und Übungen
- Die Geschwister von $\mathcal{O}$: Ω, Θ und $\sim$
- Konkrete Datentypen: Arrays

Rückblick/Wiederholung

- Komplexität rekursiver Programme
  - Beispiel: m^n naiv $O(n)$, clever $O(\log n)$
  - Kosten mit Rekurrenzrelationen abschätzen
- Divide and Conquer
  - Beispiel: Binäre Suche
- Das Master-Theorem
  - Rekurenzrelationen für Divide-and-Conquer abschätzen
  - Beispiele: m^n, Binäre Suche

Erinnerung: Master-Theorem

Master-Theorem

$$f(n) = \underbrace{a \cdot f(\left\lfloor \frac{n}{b} \right\rfloor)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Kombination zur Gesamtlösung}} \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

wobei $a \in \mathbb{N}^{\geq 1}, b \in \mathbb{N}^{\geq 2}, d \in \mathbb{R}^{\geq 0}$. Dann gilt:

1. $a < b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(n^d)$
2. $a = b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(\log_b n \cdot n^d)$
3. $a > b^d \quad \Rightarrow \quad f(n) \in \mathcal{O}(n^{\log_b a})$

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$
  - $a = 4, b = 2, d = 1$, also $a > b^d$ (Fall 3)

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$
  - $a = 4, b = 2, d = 1$, also $a > b^d$ (Fall 3)
  - $f \in \mathcal{O}(n^{\log_2 4}) =$

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$
  - $a = 4, b = 2, d = 1$, also $a > b^d$ (Fall 3)
  - $f \in \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$
  - $a = 4, b = 2, d = 1$, also $a > b^d$ (Fall 3)
  - $f \in \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$
- $f(n) = 4 \cdot f(\frac{n}{2}) + n^2$
  - $a = 4, b = 2, d = 2$, also $a = b^d$ (Fall 2)
  - $f \in \mathcal{O}(\log_2 n \cdot n^2)$

Lösungen: Master-Theorem

- $f(n) = 4 \cdot f(\frac{n}{2}) + n$
  - $a = 4, b = 2, d = 1$, also $a > b^d$ (Fall 3)
  - $f \in \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$
- $f(n) = 4 \cdot f(\frac{n}{2}) + n^2$
  - $a = 4, b = 2, d = 2$, also $a = b^d$ (Fall 2)
  - $f \in \mathcal{O}(\log_2 n \cdot n^2)$
- $f(n) = 4 \cdot f(\frac{n}{2}) + n^3$
  - $a = 4, b = 2, d = 3$, also $a < b^d$ (Fall 1)
  - $f \in \mathcal{O}(n^3)$

Zur Vorlesung 8

265

Zusammenfassung

- Rückblick
- Master-Theorem und Übungen
- Die Geschwister von $\mathcal{O}$: Ω, Θ und $\sim$
- Konkrete Datentypen: Arrays

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 9

- Rückblick
- Arrays Teil 2
- Listen

Rückblick/Wiederholung

- $\mathcal{O}, \Omega, \Theta, \sim$
  - Irgendwann im wesentlichen nicht schneller - $\mathcal{O}$
  - Irgendwann im wesentlichen nicht langsamer - Ω
  - Irgendwann im wesentlichen genau so schnell - Θ
  - Irgendwann genau so schnell - $\sim$
- Arrays
  - Definition
  - Organisation im Speicher
  - Verwendung/Datenhaltung in Arrays
  - Standard-Operationen

Rückblick: Komplexitätsrelationen

- Betrachten Sie folgende Funktionen:
  - $h_1(x) = x^2 + 100x + 3$
  - $h_2(x) = x^2$
  - $h_3(x) = \frac{1}{3}x^2 + x$
  - $h_4(x) = x^3 + x$

$$g \in \mathcal{O}(f)\text{: } \lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$$
$$g \in \Omega(f)\text{: } \lim_{x\to\infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$$
$$g \in \Theta(f)\text{: } \lim_{x\to\infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$$
$$g \sim f\text{: } \lim_{x\to\infty} \frac{g(x)}{f(x)} = 1$$

Zeile steht in Relation . . . zu Spalte:

	h_1	h_2	h_3	h_4
h_1	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_2	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}$
h_3	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta$	$\mathcal{O}, \Omega, \Theta, \sim$	$\mathcal{O}$
h_4	Ω	Ω	Ω	$\mathcal{O}, \Omega, \Theta, \sim$

Zur Vorlesung 9

Zusammenfassung

- Rückblick
- Arrays Teil 2
- Listen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 10

- Rückblick
- Grundlagen des Sortierens
  - Klassifikation
- Einfache Sortierverfahren
  - Selection Sort
  - Insertion Sort

Rückblick/Wiederholung

- Einfach verkettete Listen
  - Struktur
  - Zyklenerkennung mit Hase und Igel
- Doppelt verkettete Listen
  - Struktur
  - Einfügen/Ausfügen
- Eigenschaften
  - Listen und Arrays
  - Operationen mit Komplexitäten

Zur Vorlesung 10

Zusammenfassung

- Rückblick
- Grundlagen des Sortierens
  - Klassifikation
- Einfache Sortierverfahren
  - Selection Sort
  - Insertion Sort

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 11

- Rückblick
- Einfache Sortierverfahren (Fortsetzung)
  - Bubble Sort
  - Analyse der einfachen Verfahren
- Sortieren mit Divide-and-Conquer (1)
  - Quicksort

Rückblick/Wiederholung

- Klassifikationskriterien
  - Verwendete Datenstrukturen
  - Verhältnis der benötigten Operationen
  - Benötigter zusätzlicher Speicher (in-place vs. out-of-place)
  - Stabilität: Auswirkung auf Elemente mit gleichem Schlüssel
- Einfache Sortierverfahren
  - Selection Sort
  - Insertion Sort
  - `https://www.youtube.com/watch?v=kPRA0W1kECg`

Zur Vorlesung 11

Zusammenfassung

- Rückblick
- Einfache Sortierverfahren (Fortsetzung)
  - Bubble Sort
  - Analyse der einfachen Verfahren
- Sortieren mit Divide-and-Conquer (1)
  - Quicksort

Hausaufgabe

Bestimmen Sie mit dem Master-Theorem Abschätzungen für die folgenden Rekurrenzen, oder geben Sie an, warum das Theorem nicht anwendbar ist. Finden Sie in diesen Fällen eine andere Lösung?

- $T(n) = 3T(\frac{n}{2}) + n^2$
- $T(n) = 7T(\frac{n}{2}) + n^2$
- $T(n) = 4T(\frac{n}{2}) + n \log n$
- $T(n) = 4T(\frac{n}{2}) + \log n$
- $T(n) = T(n-1) + n$
- $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n^2$
- $T(n) = 2T(\frac{n}{4}) + \log n$

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 12

- Terminplanung
- Rückblick
- Sortieren mit Divide-and-Conquer (2)
  - Quicksort - Komplexitätsanalyse
  - Übung: Master-Theorem

Vorlesungsverlegungen

- 1.7. 2015: Berufungsgespräche (14:00-18:00)
  - Vorschlag: Vorlesung 14C 10:00-11:30
  - Vorschlag: Vorlesung 14B:12:30-14:00
- 13.7.: Lehrkolleg 2
  - Vorschlag: Vorlesung 14C 15.7. 14:20-15:50
  - Labor?

Rückblick/Wiederholung

- Bubble Sort
- Analyse der einfachen Verfahren
- Sortieren mit Divide-and-Conquer (1)
  - Quicksort
    - Rate Pivot p
    - Teile Array in Teil $\leq$ p, p, Teil $\geq p$
    - Sortiere die Teile rekursiv

Zur Vorlesung 12

Zusammenfassung

- Rückblick
- Sortieren mit Divide-and-Conquer (2)
  - Quicksort - Komplexitätsanalyse
  - Übung: Master-Theorem

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 13

- Rückblick
- Sortieren mit Divide-and-Conquer (3)
  - Mergesort - Algorithmus
  - Mergesort - Komplexität
- Einführung Heaps

Rückblick/Wiederholung

- Quicksort - Analyse
  - Best case: Pivot zerlegt in Hälften
    - Master-Theorem: $q(n) = 2q(\lfloor \frac{n}{2} \rfloor) + kn$
    - $a = 2, b = 2, d = 1$: Fall 2: $q(n) \in \mathcal{O}(\log_2 n \cdot n^1) = \mathcal{O}(n \log n)$
  - Worst-case: Pivot ist kleinstes (analog größtes)
    - $q(n) = kn + q(n-1)$ (ohne Master-Theorem)
    - Also: $q(n) \in \mathcal{O}(n^2)$
  - Hausaufgabe: Master-Theorem
    - Trick: Wenn $c(n)$ kein Polynom ist, dann mit Polynom nach oben abschätzen ($\mathit{log} n \in O(n)$ und $n \log n \in O(n^2)$

Zur Vorlesung 13

Zusammenfassung

- Rückblick
- Sortieren mit Divide-and-Conquer (3)
  - Mergesort - Algorithmus
  - Mergesort - Komplexität
- Einführung Heaps

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 14

- Rückblick
- Heaps
  - Operationen auf Heaps
  - Komplexität
  - Heapsort
- Sortieren – Abschluss

Rückblick/Wiederholung

- Mergesort - Algorithmus
- Mergesort - Komplexität
  - Master-Theorem: $q(n) = 2q(\lfloor \frac{n}{2} \rfloor) + kn$
  - $a = 2, b = 2, d = 1$: Fall 2: $q(n) \in \mathcal{O}(\log_2 n \cdot n^1) = \mathcal{O}(n \log n)$
- Bottom-Up Mergesort
- Heaps
  - “Shape”: Fast vollständige Binärbäume
  - “Heap”: Eltern $\geq$ Kinder (Max-Heap)

Zur Vorlesung 14

Zusammenfassung

- Rückblick
- Heaps
  - Operationen auf Heaps
  - Komplexität
  - Heapsort
- Sortieren – Abschluss
  - Besser als $\mathcal{O}(n \log n)$ wird es nicht
  - Mergesort ist nahezu optimal

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 15

- Rückblick
- Logarithmen
- Einführung: Suchen und Finden

Rückblick/Wiederholung

- Heap Eigenschaften
  - Shape
  - Heap
  - Heaps als Arrays
- Operationen auf Heaps
  - Einfügen und bubble-up
  - Ausfügen (der Wurzel) und bubble-down
  - Heapify
- Heapsort
  - Heapify Array
  - Tausche Größtes gegen Letztes
  - Verkleinere Heap um ein Element
  - Bubble-Down der Wurzel
  - . . . bis das Array sortiert ist
- Sortieren allgemein: $\mathcal{O}(n \log n)$ ist gut

Zur Vorlesung 15

Zusammenfassung

- Rückblick
- Logarithmen
  - Ideen
  - Rechenregeln
  - Anwendungen
- Einführung: Suchen und Finden
  - Schlüssel und Werte
  - Abstrakte Operationen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 16

- Rückblick
- Binäre Suchbäume
  - Definition
  - Suche
  - Einfügen
  - Analyse
    - Best case
    - Worst case
  - Löschen

Rückblick/Wiederholung

- Logarithmen
  - Umkehr der Exponentialfunktionen
  - Langsames Wachstum (“fast konstant”)
  - $\log_b(x \cdot y) = \log_b x + \log_b y$
  - $\log_a x = \frac{\log_b x}{\log_b a} \rightsquigarrow \mathcal{O}(\log_a x) = \mathcal{O}(\log_b x)$
  - “Wie oft kann ich eine Menge halbieren, bis die Teile nur noch einzelne Elemente enthalten?”
  - “Wie hoch ist ein vollständiger Binärbaum mit n Knoten?”
- Dictionaries
  - Verwalten Schlüssel/Wert-Paare
  - Uns interessiert meist nur der Schlüssel - den Wert denken wir uns ;-)
  - Operationen:
    - Anlegen, Einfügen, Suchen, Löschen, Ausgabe, ...

Zur Vorlesung 16

Zusammenfassung

- Rückblick
- Binäre Suchbäume
  - Definition
  - Suche
  - Einfügen
  - Analyse
    - Best case
    - Worst case
  - Löschen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 17

- Rückblick
- Binäre Suchbäume
  - Löschen
  - Scorecard
- Balancierte Binärbäume
  - Problem: Verlust der Balance
  - Rotationen
  - Einführung: AVL-Bäume

Rückblick/Wiederholung

- Binäre Suchbäume
  - Definition
  - Suche
  - Einfügen
  - Analyse
    - Best case $\rightsquigarrow \mathcal{O}(\log n)$ für Einfügen/Suchen/(Löschen)
    - Worst case $\rightsquigarrow \mathcal{O}(n)$ für Einfügen/Suchen/(Löschen)
  - Löschen - erst angedacht

Zur Vorlesung 17

Zusammenfassung

- Rückblick
- Binäre Suchbäume
  - Löschen
  - Scorecard
- Balancierte Binärbäume
  - Problem: Verlust der Balance
  - Rotationen
  - Einführung: AVL-Bäume

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 18

- Rückblick: Binärbäume
- Graphalgorithmen
  - Graphen
    - Definitionen
    - Anwendungen
    - Repräsentation
  - Problem: Minimaler Spannbaum
    - Prim's Algorithmus

Rückblick/Wiederholung

- Löschen in binären Suchbäumen
  - Fall 1: Abschneiden von Blättern
  - Fall 2: “Kurzschließen” von Knoten mit einem Nachfolger
  - Fall 3: Tausche Knoten mit kleinstem Knoten aus rechtem Teilbaum, dann Fall 1/Fall 2
- Binäre Suchbäume: Scorecard
  - “Alles log n”
- Balancierte Binärbäume
  - Größenbalanciert
  - Höhenbalanciert
  - Rotationen
  - Konzept AVL-Baum

Zur Vorlesung 18

Zusammenfassung

- Rückblick: Binärbäume
- Graphalgorithmen
  - Graphen
    - Definitionen
    - Anwendungen
    - Repräsentation

  v
  - Problem: Minimaler Spannbaum
    - Prim’s Algorithmus

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 19

- Rückblick: Graphen, Prim
- Komplexität:
  - Prim naiv vs. Prim optimiert
- Routing: Dijkstra
  - Algorithmus
  - Komplexität

Rückblick/Wiederholung

- Graphen: V, E (Knoten und Kanten)
  - Gerichtet, ungerichtet
  - Pfade, Zyklen, Verbundenheit, Bäume
- Gewichtete Graphen:
  - Beschrifungsfunktion (z.B.: $e : E \to \mathbb{N}$)
- Anwendungen
- Repräsentationen
  - Adjazenzmatrix
  - Adjazenzlisten
- Minimaler Spannbaum
  - Algorithmus von Prim (Greedy)
  - Naiv: $\mathcal{O}(|V| \cdot |E|)$
  - Geht es besser?

Zur Vorlesung 19

Zusammenfassung

- Rückblick: Graphen, Prim
- Komplexität:
  - Prim naive vs. Prim optimiert
- Routing: Dijkstra
  - Algorithmus
  - Komplexität

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Ziele Vorlesung 20

- Rückblick
- Dijkstra Teil 2
  - Implementierung
  - Komplexität
  - Optimierungen
- Greedy-Algorithmen und ihre Grenzen

Rückblick/Wiederholung

- Komplexität Prim
  - Prim naive: $\mathcal{O}(|V| \cdot |E|)$
  - Prim optimiert: $\mathcal{O}(|E| \cdot \log |V|)$
- Routing: Dijkstra
  - Algorithmus
  - Komplexität

Zur Vorlesung 20

Zusammenfassung

- Rückblick
- Dijkstra Teil 2
  - Implementierung
  - Komplexität
  - Optimierungen
- Greedy-Algorithmen und ihre Grenzen

Feedback

- Was war der beste Teil der heutigen Vorlesung?
- Was kann verbessert werden?
  - Optional: Wie?

Klausurvorbereitung: Übungsaufgaben

Feedback

- Was war die Vorlesung insgesamt?
- Was kann verbessert werden?
  - Optional: Wie?
- Feedback auch gerne über das Feedback-System
  `https://feedback.dhbw-stuttgart.de/`
  - Wird noch freigeschaltet!

Klausurvorbereitung: Übungsklausur

Lösungen: Master-Theorem

Lösungen zu Seite 629

- $T(n) = 3T(\frac{n}{2}) + n^2$
  - $a = 3, b = 2, d = 2 \rightsquigarrow a = 3 < b^d = 4$: Fall 1, $T(n) \in \mathcal{O}(n^d) = \mathcal{O}(n^2)$
- $T(n) = 7T(\frac{n}{2}) + n^2$
  - $a = 7, b = 2, d = 2 \rightsquigarrow a = 7 > b^d = 4$: Fall 3, $T(n) \in \mathcal{O}(n^{\log_b a}) \approx \mathcal{O}(n^{2.807})$
- $T(n) = 4T(\frac{n}{2}) + n \log n$
  - $n \log n$ kann (schlecht) nach oben durch $f(n) = n^2$ abgeschätzt werden (oder auch: $n \log n \in \mathcal{O}(n^2)$). Damit: $a = 4, b = 2, d = 2$, also $a = b^d$ (Fall 2) und $T(n) \in \mathcal{O}(\log_b n \cdot n^d) = \mathcal{O}(n^2 \log n)$
- $T(n) = 4T(\frac{n}{2}) + \log n$
  - Siehe oben. Das $\log n < n$ für alle größeren n gilt $\log n \in \mathcal{O}(n)$. Damit: $a = 4, b = 2, d = 1$, also $a > b^d$ und $T(n) \in \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$

Lösungen: Master-Theorem

- $T(n) = T(n-1) + n$
  - Master-Theorem ist nicht anwendbar (Problem wird nicht ge-n-telt)
  - Nachdenken ergibt: Es sind n Schritte bis zur 0. Jedes mal kommt die aktuelle Größe dazu. Also: $T(n) \approx \sum_{i=0}^{n} i \approx \frac{1}{2}n^2 \in \mathcal{O}(n^2)$
- $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n^2$
  - Master-Theorem nicht anwendbar - das Akra-Bazzi-Theorem wäre anwendbar (haben wir aber nicht gemacht). Mann kann die Lösung (sehr schlecht) nach oben anschätzen, wenn man für die zweite Rekursion auch $T(\frac{n}{2})$ annimmt (dann: $a = 2, b = 2, d = 2 \rightsquigarrow$ Fall1, $T(n) \in \mathcal{O}(n^2)$)
- $T(n) = 2T(\frac{n}{4}) + \log n$
  - Siehe vorige Seite. $\log n \in \mathcal{O}(n)$, also $T(n) \in \mathcal{O}(n)$ nach Fall 1.