Algorithmen und Datenstrukturen

Bachelor of Science - Informatik

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

SoSe 2024

Übersicht

In den *Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen* (Stand Juli 2016) der Gesellschaft für Informatik[1] werden für die einzelnen kognitiven Prozessdimensionen des Inhaltsbereiches *Algorithmen und Datenstrukturen* vorgeschlagen:

- Verstehen
  - Asymptotisches Verhalten von Funktionen (Landau-Symbole) sowie die wichtigsten Komplexitätsklassen erläutern.
  - Korrektheitsbeweise auf der Basis von Schleifeninvarianten erklären.
  - Kernidee der Entwurfsparadigmen Backtracking, Greedy, Divide-and-Conquer und Dynamisches Programmieren sowie Beispielalgorithmen erklären.
  - Grundlegende Datenstrukturen (Feld, verkettete Liste, binäre Bäume, Hash-Tabellen, balancierte Bäume wie z.B. AVL-Bäume oder B-Bäume) erklären.
  - Die Datenstrukturen Stack, Warteschlange und Prioritätswarteschlange einschließlich ihrer Implementierung erklären.
  - Einfache Beispiele für nebenläufige Algorithmen erläutern.

[1] `https://gi.de/service/publikationen/empfehlungen`

- Anwenden und Übertragen
  - Algorithmen wie z.B. Breiten-/Tiefensuche, Dijkstra- und Floyd-Warshall-Algorithmus auf Beispieleingaben anwenden.
  - Typische Operationen (Suchen, Einfügen, Löschen) beispielhaft für Datenstrukturen durchführen.
  - Laufzeit rekursiver Algorithmen mit einer Rekursionsgleichung beschreiben und in eine geschlossene Form überführen.
  - Für kleine Aufgaben wie z.B. das rekursive Umdrehen einer Liste, eigene Algorithmen entwickeln. Einen Algorithmus für ein Anwendungsszenario implementieren.
- Analysieren und Bewerten
  - Grundlegende Algorithmen und Datenstrukturen bzgl. der Laufzeit und des Speicherbedarfs analysieren und ihre Komplexität bestimmen.
  - Die prinzipielle algorithmische Schwierigkeit einfacher Probleme im Sinne der Komplexitätstheorie einschätzen.
  - Algorithmen und Datenstrukturen anhand der asymptotischen Laufzeiten vergleichen.
  - Implementierung und Laufzeitmessung von einfachen Anwendungsfällen durchführen und die gemessenen Laufzeiten den theoretischen Ergebnissen gegenüberstellen.
  - Unter gegebenen Randbedingungen passenden Algorithmus/Datenstruktur wählen.

Formale, algorithmische und mathematische Kompetenzen:

Informatikerinnen und Informatiker müssen Probleme und Anforderungen exakt beschreiben, um diese in geeigneten Datenstrukturen und effizienten Algorithmen umzusetzen.

Für die Modellierung von Problemen und Sachverhalten werden logische und algebraische Kalküle, graphentheoretische Notationen, formale Sprachen und Automaten sowie spezielle Kalküle wie Petri-Netze oder [. . .] eingesetzt.

Zur Bewältigung einer Problemstellung kommen Verfahrensweisen zum Einsatz, um den algorithmischen Kern des Problems zu identifizieren - darauf basierend werden Algorithmen entworfen, verifiziert und bzgl. ihres Ressourcenbedarfs bewertet. Die in den Algorithmen enthaltenen Kalküle werden zur angemessenen fachlichen Kommunikation und Bewertung von Problemlösungen [. . .] genutzt.

Für spezielle Bereiche der Informationsverarbeitung, wie Signal- und Bildverarbeitung, Kryptographie oder Mustererkennung, werden fortgeschrittene Kenntnisse der Analysis, der Algebra, der Kombinatorik und der Statistik benötigt.

Inhaltsbereiche für formale, algorithmische und mathematische Grundkompetenzen des Bachelor-Studiums:

- Diskrete Strukturen, Logik und Algebra
- Analysis und Numerik
- Wahrscheinlichkeitstheorie und Statistik
- Formale Sprachen und Automaten
- Modellierung
- Algorithmen und Datenstrukturen

nur damit Sie wissen, wo Sie hier gelandet sind ...

Schulgesetz NRW, §42, Abs. 3:

> Schülerinnen und Schüler haben die Pflicht daran mitzuarbeiten, dass die Aufgabe der Schule erfüllt und das Bildungsziel erreicht werden kann. Sie sind insbesondere verpflichtet, sich auf den Unterricht vorzubereiten, sich aktiv daran zu beteiligen, die erforderlichen Arbeiten anzufertigen und *die Hausaufgaben zu erledigen*. Sie haben die Schulordnung einzuhalten und ...

Bereinigte Amtliche Sammlung der Schulvorschriften NRW 12-63 Nr. 3

> Hausaufgaben sollen die individuelle Förderung unterstützen. Sie können dazu dienen, *das im Unterricht Erarbeitete einzuprägen, einzuüben und anzuwenden*. Sie müssen aus dem Unterricht erwachsen und wieder zu ihm führen, in ihrem Schwierigkeitsgrad und Umfang die Leistungsfähigkeit, Belastbarkeit und Neigungen der Schülerinnen und Schüler berücksichtigen und von diesen selbstständig ohne fremde Hilfe [...] erledigt werden können.

Im Studium nehmen Übungen den Platz der Hausaufgaben ein. Sie sind wichtig zum Verstehen der Modulinhalte.

Heute an der Hochschule: Übungen

Das Modul ALD ist mit sechs Leistungspunkten (CP) in der Prüfungsordnung angegeben. Laut der *Verordnung zur Regelung des Näheren der Studienakkreditierung* in Nordrhein-Westfalen vom 25.01.2018, Paragraph 8 Leistungspunktesystem, Satz (1) entspricht ein Leistungspunkt einer Gesamtarbeitsleistung von 25 bis höchstens 30 Zeitstunden.

Insgesamt ergibt sich also ein Arbeitsaufwand von etwa 150 Stunden im Semester. Diese Stunden verteilen sich bei 15 Wochen Vorlesungszeit im Semester auf zehn Stunden pro Woche. Darin ist die Zeit für die Klausurvorbereitung enthalten.

Wenn Sie nicht an den Übungen teilnehmen oder die Übungen nicht bearbeiten, haben Sie zwar in der Woche mehr Zeit für andere Dinge, aber die Vorbereitungszeit für die Klausur wird entsprechend länger.

Eine gute Mitarbeit in und an den Übungen ist hilfreich zum Verstehen der Vorlesungsinhalte und damit zum Bestehen der Klausur. Eine aktive Teilnahme an den Übungen verkürzt außerdem die Vorbereitungszeit für die Prüfung und erhöht die Chance, die Klausur zu bestehen.

Inhalt

Grundlagen

- Analyse von Algorithmen
- Entwurfsmethoden

Sortieren

- Selectionsort
- Insertionsort
- Quicksort
- Mergesort
- Heapsort
- Untere Schranke
- Counting-/Radixsort
- Bucketsort

Inhalt

Suchen

- Exponentielle Suche
- Interpolationssuche
- Auswahlproblem (Median-Bestimmung)

Datenstrukturen

- abstrakte Datentypen
- Array (Feld)
- Linked List (verkettete Liste)
- Tree (Baum)
- Heap (Halde)
- Hashtable (Hash-Tabelle)

Inhalt

Graphalgorithmen

- Tiefen- und Breitensuche und deren Anwendungen
- Minimale Spannbäume
- Kürzeste Wege

Lösen schwerer Probleme

- Greedy Heuristiken
- Backtracking
- Branch-and-Bound
- Lokale Suche

Aktuelle Informationen, Sprechzeiten, Folien stehen im Moodle-Kurs und unter

`https://lionel.kr.hsnr.de/~rethmann/`

Anmerkungen, Korrekturen oder Verbesserungsvorschläge sind immer willkommen! Sprechen Sie mich an oder schicken Sie mir eine E-Mail.

Büro: F 202 E-Mail: `mailto:jochen.rethmann@hs-niederrhein.de`

Stellen Sie Fragen! Nur so kann ich beurteilen, ob Sie etwas verstanden haben oder noch im Trüben fischen.

Konfuzius:

> Wer fragt, ist ein Narr für eine Minute.
> Wer nicht fragt, ist ein Narr sein Leben lang.

- Entspannen Sie sich. Richten Sie Ihre volle Aufmerksamkeit auf die Veranstaltung.
- Setzen Sie sich Ziele. Was wollen Sie in dieser Veranstaltung lernen?
- Hören Sie aktiv zu. Denken Sie mit und sorgen Sie dafür, dass alle Unklarheiten ausgeräumt werden.
- Notieren Sie Wichtiges. Machen Sie sich Notizen zur Veranstaltung und markieren Sie die wichtigsten Aspekte.
- Formulieren Sie Fragen. Notieren Sie Fragen und bringen Sie diese ein.
- Beteiligen Sie sich. Bringen Sie Ihre Anliegen und Ideen ein.
- Haben Sie Geduld. Lernen Sie, andere Ansichten zu akzeptieren. Helfen Sie, andere besser zu verstehen.
- Denken Sie positiv. Werden Sie sich darüber klar, wie die Veranstaltung zu Ihrem Lernerfolg beiträgt.
- Handeln Sie schnell. Setzen Sie Ihre Ziele bald um. Verzögerung ist der erste Schritt zum Vergessen.

Literatur

- T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* Springer Spektrum
  als e-book in der Digi-Bib erhältlich
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms.*
  Oldenbourg Wissenschaftsverlag
  als e-book in der Digi-Bib erhältlich
- Robert Sedgewick: *Algorithms.* Pearson Studium
- Uwe Schöning: *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag
- Volker Heun: *Grundlegende Algorithmen.* Vieweg Verlag
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Datastructures and Algorithms.*
  Addison-Wesley
- St.J. Goebbels, J. Rethmann: *Eine Einführung in die Mathematik an Beispielen aus der Informatik – Logik, Zahlen, Graphen, Analysis und Lineare Algebra.*
  Springer Spektrum Berlin, Heidelberg 2023

Internet

Im Internet stehen gute Animationen der hier präsentierten Inhalte sowohl zu Algorithmen als auch zu Datenstrukturen zur Verfügung, z.B. auf der folgenden Seite:

`https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`

Im Detail können die dort präsentierten Algorithmen und Datenstrukturen von den in der Vorlesung vorgestellten abweichen. So sind bspw. die Splay-Bäume anders implementiert als in der Vorlesung. Unsere Definition der Splay-Bäume kommt aus dem Buch von Ottmann und Widmayer.

Übersicht

Übersicht

Was ist ein Algorithmus?

- Der Begriff Algorithmus ist vom Namen des arabischen Mathematikers Muhammed al-Chwarizmi, etwa 783-850 abgeleitet.
- Verfahren zur Lösung eines Problems, unabhängig von Implementierung in konkreter Programmiersprache.
- Vorteil: Konzentrieren auf das Problem, nicht auf die Syntax, Grammatik und Eigenarten einer Programmiersprache.

Programmiersprachen haben gemeinsame Konzepte

Konzepte	C/C++
Wertzuweisungen/Ausdrücke	`y = 2*x + c`
Bedingte Anweisung	`if ... else`
Iterative Anweisung	`for, while`
Ein- und Ausgabeanweisung	`printf/cout`
Prozedur-/Funktionsaufrufe	`sqrt(4.789)`

Welche Eigenschaften interessieren uns?

- *Korrektheit* (E. Dijkstra: Testen kann die Anwesenheit, aber nicht die Abwesenheit von Fehlern zeigen.)
- *Laufzeit* (Wie schnell wird das Problem gelöst?)
- *Speicherplatz* (Wie viel Speicherplatz wird benötigt?)

Weitere interessante Eigenschaften:

- *Kommunikationszeit* (bei parallelen/verteilten Algorithmen)
- *Güte* (oft nur approximative Lösung in akzeptabler Zeit berechenbar)

Wichtiger als Performance?

- Wartbarkeit/Erweiterbarkeit
- Entwicklungszeit/Einfachheit
- Zuverlässigkeit/Ausfallsicherheit
- Bedienbarkeit
- IT-Sicherheit

$\Rightarrow$ siehe Vorlesungen PE1, PE2, IAS, SWE, ITS usw.

Wenn wir eine Aufgabe mit verschiedenen Algorithmen lösen können, müssen wir Algorithmen bewerten bzw. vergleichen:

- Wann ist ein Algorithmus besser als ein anderer?
- Was sind gute/schlechte Algorithmen?
- Gibt es einen optimalen Algorithmus?

$\Rightarrow$ Laufzeit messen und vergleichen?

Probleme beim Messen und Vergleichen der Laufzeit:

- unterschiedlich schnelle Hardware
- unterschiedlich gute Compiler bzw. unterschiedliche Compiler-Optionen
- unterschiedliche Betriebssysteme bzw. Laufzeitumgebungen
- unterschiedliche Eingabedarstellungen bzw. Datenstrukturen

Systemumgebung 1:

Hardware:	Pentium III mobile, 1GHz, 256 MB, Q1 2000
Linux:	Kernel 2.4.10, gcc 2.95.3
Windows:	XP Home (SP1), Borland C++ 5.02

Systemumgebung 2:

Hardware:	Pentium M (Centrino), 1,5GHz, 512 MB, Q1 2004
Linux:	Kernel 2.6.4, gcc 3.3.3
Windows:	XP Home (SP2), Borland C++ 5.5.1

Systemumgebung 3:

Hardware:	Intel Core) Intel i7-4800, 2.7 GHz, 16 GByte, Q2 2013
Linux:	Kernel 3.19.0, gcc 4.6.3
Windows:	Windows 8, Visual Studio 2013

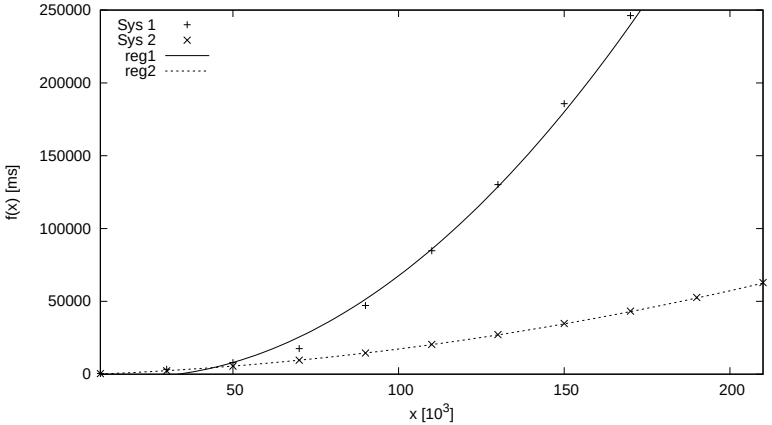
Messergebnisse für das Sortieren von Zahlen:

input size	System 1		System 2		System 3	
	Linux	XP [s]	Linux	XP [s]	Linux	W8 [s]
8192	1	0	1	0	0	0.1
16384	3	2	2	1	0.1	0.4
32768	9	4	6	3	0.6	1.5
65536	34	17	21	9	2	5.6
131072	221	137	72	29	9	17

Problem: Bewertung der Messergebnisse

- Unterschiede aufgrund Compiler, Betriebssystem, Taktfrequenz, Prozessor, RAM?
- Skalierung: doppelte Eingabegröße, vierfache Laufzeit?

Laufzeit beschreibbar durch Polynom $ax^2 + bx + c$ **???**



Reg.1:	$10.6 \cdot x^2$	-	$399.4 \cdot x$	+	1507.3
Reg.2:	$1.1 \cdot x^2$	+	$69.9 \cdot x$	-	648.1

Die Koeffizienten des Polynoms werden bestimmt durch die Minimierung der Summe der Fehlerquadrate:

$$F(x) := \sum_i (y_i - (ax_i^2 + bx_i + c))^2 \longrightarrow \min$$

Koeffizienten bestimmen durch quadratische Regression:

$$\frac{\partial F}{\partial a} = 0 \qquad \frac{\partial F}{\partial b} = 0 \qquad \frac{\partial F}{\partial c} = 0$$

$\Rightarrow$ es ist ein Gleichungssystem mit drei Gleichungen und drei Unbekannten zu lösen

Problem: Koeffizienten sind abhängig von Hard-/Software

Lösung: Korrekturfaktoren für jede Hard- und Softwareumgebung einführen.
$\rightarrow$ Wäre sehr aufwändig und ist nur sehr eingeschränkt möglich.

Messen der Laufzeit:

- Implementieren in einer konkreten Sprache/Compiler.
- Festgelegt bei Ausführung: Rechner/Eingabemenge.

Probleme:

- Festlegen auf Norm nur schwer möglich.
- Ergebnisse lassen sich nur schwer übertragen.
- Aussage über Skalierung basiert auf Vermutung.
- Speicherbegrenzung: Paging/Swaping, Cache-Effekte

$\Rightarrow$ Messen und Vergleichen der Laufzeiten ist oft nicht praktikabel oder sinnvoll!

Bewertung von Algorithmen[2]

Auswege:

- *idealisiertes Modell* (RAM: Random Access Machine)
  - Festgelegter Befehlssatz (Assembler-ähnlich)
  - abzählbar unendlich viele Speicherzellen

  $\Rightarrow$ Laufzeit: Anzahl ausgeführter RAM-Befehle

  $\Rightarrow$ Speicherbedarf: Anzahl benötigter Speicherzellen
- *charakteristische Parameter ermitteln*
  - Sortieren: Schlüsselvergleiche, Vertauschungen
  - Arithmetik: Additionen, Multiplikationen

Mehr zu Maschinenmodellen und Komplexitätstheorie in der Vorlesung THI, *Theoretische Informatik*, im vierten Semester.

[2] siehe z.B. *The Art of Computer Programming*, D.E.Knuth

Laufzeit und Speicherbedarf sind in der Regel abhängig von der Größe der Eingabe.

Warum ist die Komplexität der Algorithmen interessant?

Sind unsere heutigen Rechner schnell genug, um jede Aufgabe in einer vertretbaren Zeit zu erledigen?

Beispiel: Traveling Salesperson Problem (TSP)

- Gegeben: Eine Menge von Orten, die untereinander durch Wege verbunden sind. Die Wege sind unterschiedlich lang.
- Gesucht: Eine Rundreise, die durch alle Orte genau einmal führt und unter allen Rundreisen minimale Länge hat. (Tourenplanung)

Das Problem ist NP-vollständig[(3)], der beste bekannte Algorithmus hat eine Laufzeit, die proportional zu 2^n ist, wobei n die Anzahl der Orte ist.

Frage: Wie groß darf n werden, wenn die Lösung innerhalb eines Tages berechnet werden soll?

[(3)] Die Bedeutung von NP-vollständig wird in THI erklärt.

Annahme: Der Prozessor führt 10^9 charakteristische Operationen pro Sekunde aus.

- $1.000.000.000 = 1 \cdot 10^9$ Schritte pro Sekunde
- $3.600.000.000.000 = 3.600 \cdot 10^9$ Schritte pro Stunde
- $86.400.000.000.000 = 24 \cdot 3.600 \cdot 10^9$ Schritte pro Tag

$\Rightarrow$ *Lösbare Problemgröße*

- am Tag: 46 Städte
- im Jahr: 55 Städte
- in 100 Jahren: 61 Städte

In Deutschland gibt es ungefähr 5000 Städte!

Frage: Löst ein schnellerer Rechner das Problem???
Antwort: Nein!

Beobachtung: Geschwindigkeit verdoppelte sich alle 1,5 Jahre.
Annahme: Dieser Trend setzt sich fort.

- in 10 Jahren: Lösbare Problemgröße am Tag: 52 Orte
- in 100 Jahren: Lösbare Problemgröße am Tag: 112 Orte
- Fortschritt nach 1,5 Jahren (Computer B):

Computer	Dauer	Anzahl Schritte
A	1 Tag	2^n
B	1 Tag	$2 \cdot 2^n = 2^{n+1}$

Problem: Die Touren sollen jetzt geplant werden, nicht in 100 Jahren!

Wir lassen das Problem von mehreren Prozessoren, Kernen bzw. Computern gleichzeitig bearbeiten. Jede Recheneinheit bearbeitet ein Teilproblem.

Frage: Lösen parallele oder verteilte Systeme das Problem?
Antwort: Nein!

Lösbare Problemgröße am Tag bei linearem Speedup, also Rechenzeit halbiert sich bei doppelter Anzahl von Rechenkernen.

#Rechnenkerne	TSP $\approx 2^n$	Algo. mit $\approx n^3$ (Matrixmultiplikation)
1	46	44.208
100	53	205.197
1.000	56	442.083
10.000	59	952.440

Der schnellste Rechner hat (Stand Nov. 2019) 2.414.592 Kerne (Platz 3 hat 10.649.600 Kerne)[4]

[4] `https://www.top500.org`

Lösung: bessere Algorithmen!

Laufzeit	Dauer für 5000 Städte		#Städte (Zeit: 1 Tag)	
$\approx n^6$	$5000^6 \cdot 10^{-9}$ sek	$\approx$ 495.465 Jahre	$\sqrt[6]{86,4 \cdot 10^{12}}$	≈ 210
$\approx n^5$	$5000^5 \cdot 10^{-9}$ sek	$\approx$ 99 Jahre	$\sqrt[5]{86,4 \cdot 10^{12}}$	≈ 612
$\approx n^4$	$5000^4 \cdot 10^{-9}$ sek	$\approx$ 7 Tage	$\sqrt[4]{86,4 \cdot 10^{12}}$	≈ 3.048
$\approx n^3$	$5000^3 \cdot 10^{-9}$ sek	$\approx$ 2 Minuten	$\sqrt[3]{86,4 \cdot 10^{12}}$	≈ 44.208
$\approx n^2$	$5000^2 \cdot 10^{-9}$ sek	$<$1 Sekunde	$\sqrt{86,4 \cdot 10^{12}}$	$\approx 9.295.160$

Speed is fun!

Frage: Gibt es bessere Algorithmen für das Problem???

Es gibt ein Preisgeld von 1.000.000\$[(5)] für einen Algorithmus, dessen Laufzeit für dieses Problem ein Polynom in der Anzahl der Städte ist. Es gibt ständig neue Fast-Beweise.

[(5)] `https://www.claymath.org/millennium-problems/`

Dichtestes Zahlenpaar: Finde aus n reellen Zahlen $x_1, \ldots, x_n \in \mathbb{R}$ das Zahlenpaar x_i, x_j, $i \neq j$, das unter allen Zahlenpaaren den kleinsten Abstand $d(x_i, x_j) = |x_i - x_j|$ hat.

Naiver Algorithmus: Betrachte alle $\binom{n}{2}$ Paare und bestimme das dichteste Paar.

$min := \infty$, $a := 0$, $b := 0$
for $i := 1, \ldots, n-1$ **do**
  for $j := i+1, \ldots, n$ **do**
    $diff := |x_i - x_j|$
    if $diff < min$ **then**
      $min := diff$, $a := i$, $b := j$
output x_a and x_b

Laufzeit: $\binom{n}{2} = \frac{n \cdot (n-1)}{2} = \frac{n^2 - n}{2} \approx n^2$

Übung 1. Entwickeln Sie einen effizienten Algorithmus für das Problem. Implementieren Sie Ihren und obigen Algorithmus und vergleichen Sie deren Laufzeiten für verschiedene Eingabegrößen.

Genaue Angabe der Komplexitätsfunktion ist oft schwierig oder unmöglich.

Unscharfe Aussagen: Abstrahieren von

- additiven Konstanten,
- konstanten Faktoren und
- Termen niedrigerer Ordnung.

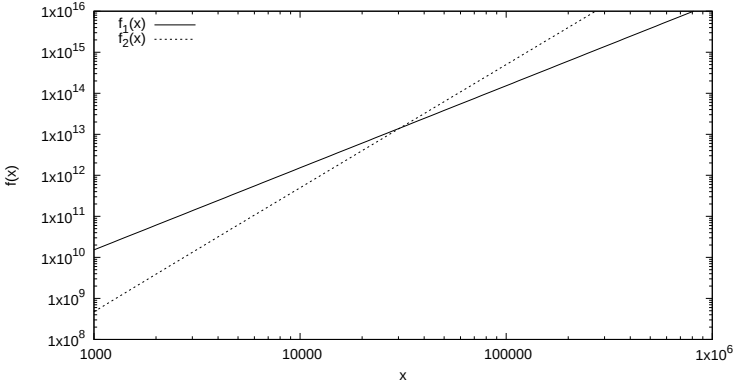
Beispiel:

$$\begin{aligned} 7 \cdot \log_2(n) + 5 \cdot n^3 + 3 \cdot n^4 &\leq 7 \cdot n^4 + 5 \cdot n^4 + 3 \cdot n^4 \\ &\leq 15 \cdot n^4 \\ &\approx n^4 \end{aligned}$$

Frage: Dürfen wir additive Konstanten, konstante Faktoren und Terme niedrigerer Ordnung vernachlässigen?

Betrachten wir dazu die folgenden Funktionen:

$$f_1(x) = 15243 \cdot x^2 + 17842 \cdot x$$
$$f_2(x) = 1/2 \cdot x^3 - 27 \cdot x^2$$



Anmerkungen:

- Bei kleinen Eingabegrößen sind die konstanten Faktoren und Terme niedriger Ordnung entscheidend.
- Große multiplikative Konstanten führen zwar oft zu theoretisch guten Lösungen, die in der Praxis aber manchmal nicht akzeptabel sind.
- *Hard- und Software-abhängige Konstanten fallen nicht mehr ins Gewicht*

Auf den folgenden Seiten betrachten wir die Landau-Symbole $\mathcal{O}$, Ω und Θ, allerdings etwas anders, als es in der Mathematik üblich ist. In der Mathematik betrachtet man Funktionen $f : \mathbb{N} \to \mathbb{R}$, wir betrachten Funktionen $f : \mathbb{N} \to \mathbb{N}$, wie es in der Komplexitätstheorie[6] üblich ist.

Außerdem gehen wir oft davon aus, dass die Funktionen monoton wachsen, also $f(n+1) \geq f(n)$ gilt. Denn wenn die Eingabe größer wird, steigt typischerweise auch die Laufzeit bzw. der Speicherbedarf.

[6] `https://de.wikipedia.org/wiki/Komplexit%C3%A4tstheorie#Landau-Notation`

Definition: Sei $g : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann bezeichnet $\mathcal{O}(g)$ die Menge der Funktionen, die asymptotisch höchstens so stark wachsen wie g.

$$\mathcal{O}(g) = \left\{ f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^+ : \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c \right\}$$

$\Rightarrow g$ ist proportional zu einer oberen Schranke für große n!

Anmerkung: Wir betrachten nur Funktionen $f : \mathbb{N} \to \mathbb{N}$, da sowohl die Größe der Eingabe als auch die Anzahl der ausgeführten Operationen eines Algorithmus immer ganzzahlig und nicht-negativ sind.

- Es werden $n \in \mathbb{N}$ Zahlen sortiert, dafür werden $f_1(n) \in \mathbb{N}$ Vergleiche und $f_2(n) \in \mathbb{N}$ Vertauschungen benötigt.
- Es wird eine Zahl in einer Folge von $n \in \mathbb{N}$ Zahlen gesucht, dafür werden $f(n) \in \mathbb{N}$ Vergleiche benötigt.

Wir schreiben abkürzend:

$\mathcal{O}(n)$	für	$\mathcal{O}(g)$	falls $g(n) = n$
$\mathcal{O}(n^k)$	für	$\mathcal{O}(g)$	falls $g(n) = n^k$
$\mathcal{O}(\log(n))$	für	$\mathcal{O}(g)$	falls $g(n) = \log(n)$
$\mathcal{O}(\sqrt{n})$	für	$\mathcal{O}(g)$	falls $g(n) = \sqrt{n}$
$\mathcal{O}(2^n)$	für	$\mathcal{O}(g)$	falls $g(n) = 2^n$

Da wir nur Funktionen $f, g : \mathbb{N} \to \mathbb{N}$ betrachten, sind einige der obigen Schreibweisen nicht ganz korrekt, zum Beispiel $g(n) = \log(n)$ oder $g(n) = \sqrt{n}$.

Trotzdem verwenden wir die Landau-Symbole $\mathcal{O}$ oder Ω auch für solche Funktionen. In diesen Fällen sind die entsprechenden nicht-negativen, ganzzahligen Funktionen $\hat{f}(n) = \max\{\lceil f(n) \rceil, 0\}$ gemeint.

Dabei bezeichnet $\lceil x \rceil$ die kleinste ganze Zahl, die nicht kleiner als x ist.
Beispiel: $\lceil 3,1 \rceil = 4$.

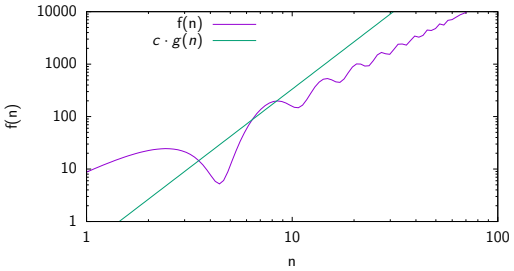
In den meisten Büchern zu Algorithmen und Datenstrukturen finden Sie eine andere Definition, die ausführlich in THI besprochen wird.

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$$

Die Menge aller Funktionen $f : \mathbb{N} \to \mathbb{N}$, für die ein $c \in \mathbb{N}$ und ein $n_0 \in \mathbb{N}$ existiert, sodass für alle $n \geq n_0$ gilt: $0 \leq f(n) \leq c \cdot g(n)$

Für alle Funktionen f und g, für die der Grenzwert $\lim_{n\to\infty} {}^{f(n)}/_{g(n)}$

- existiert, sind beide obigen Definitionen von $\mathcal{O}$ äquivalent.
- nicht existiert, kann nur die Definition von dieser Seite betrachtet werden.



Beispiel: $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^3)$

Zu betrachten ist

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2 \cdot n^2 + 37 \cdot n^3}{n^3} \leq c$$

Die Brüche müssen so umgeformt werden, dass im Zähler oder im Nenner beim Grenzwert $\lim_{n\to\infty}$ kein Ausdruck mehr steht, der gegen ∞ geht, damit der Grenzwert berechnet werden kann.

$$\frac{2 \cdot n^2 + 37 \cdot n^3}{n^3} \leq \frac{2 \cdot n^3 + 37 \cdot n^3}{n^3} = \frac{39 \cdot n^3}{n^3} = 39 < \infty \text{ oder}$$

$$\lim_{n\to\infty} \frac{2 \cdot n^2 + 37 \cdot n^3}{n^3} = \lim_{n\to\infty} \left(\frac{2}{n} + 37\right) = 37 < \infty$$

Die Aussage $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^3)$ ist also korrekt, da z.B. für $c = 39$ oder $c = 42$ die Definiton von $\mathcal{O}$ erfüllt ist.

Betrachten wir nun die zweite Definition.

Beispiel: $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^3)$

Fast wie gehabt:

$$2 \cdot n^2 + 37 \cdot n^3 \ \leq \ 2 \cdot n^3 + 37 \cdot n^3 \ = \ 39 \cdot n^3 \ \leq \ c \cdot n^3$$

Die Aussage $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^3)$ ist also korrekt, da ein $c \in \mathbb{N}$ und ein $n_0 \in \mathbb{N}$ existiert, hier z.B. $c = 42$ und $n_0 = 12$ oder $c = 39$ und $n_0 = 1$, sodass für alle $n \geq n_0$ gilt: $0 \leq 2 \cdot n^2 + 37 \cdot n^3 \leq c \cdot n^3$

In dieser Vorlesung betrachten wir in den Übungen ausführlich die Definition von $\mathcal{O}$ über Grenzwerte.

Weitere Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in \mathcal{O}(n^4)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin \mathcal{O}(n^2)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \mathcal{O}(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \mathcal{O}(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \notin \mathcal{O}(\log(n))$
- $17 \cdot \sqrt{n} + 139 \cdot n \notin \mathcal{O}(\sqrt{n})$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in \mathcal{O}(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \mathcal{O}(n^k)$ für $c_1, \ldots, c_k$ konstant, $c_k > 0$

Wir suchen natürlich immer die kleinste, obere Schranke!

Definition: Sei $g : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann bezeichnet $\Omega(g)$ die Menge der Funktionen, die asymptotisch mindestens so stark wachsen wie g.

$$\Omega(g) = \left\{ f : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^+ : \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c \right\} \quad \text{oder} \quad \Omega(g) = \left\{ f \mid g \in \mathcal{O}(f) \right\}$$

$\Rightarrow$ g ist proportional zu einer unteren Schranke für große n!

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^3)$
- $2 \cdot n^2 + 37 \cdot n^3 \in \Omega(n^2)$
- $2 \cdot n^2 + 37 \cdot n^3 \notin \Omega(n^4)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Omega(n)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Omega(n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Omega(n^k)$ für $c_1, \ldots, c_k$ konstant, $c_k > 0$

Wir suchen natürlich immer die größte, untere Schranke!

Definition: Sei $g : \mathbb{N} \to \mathbb{N}$ eine Funktion. Dann bezeichnet $\Theta(g)$ die Menge der Funktionen, die asymptotisch genauso stark wie g wachsen.

$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{N} \mid f \in \mathcal{O}(g) \wedge f \in \Omega(g)\}$$

Beispiele:

- $2 \cdot n^2 + 37 \cdot n^3 \in \Theta(n^3)$
- $42 \cdot n \cdot \log(n) + 3 \cdot n^2 \in \Theta(n^2)$
- $17 \cdot \sqrt{n} + 139 \cdot n \in \Theta(n)$
- $928 \cdot n^4 + 0.7 \cdot 2^n \in \Theta(2^n)$
- $c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_k \cdot n^k \in \Theta(n^k)$ für $c_1, \ldots, c_k$ konstant, $c_k > 0$

Wichtige Aufwandsklassen: Sei k konstant.

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

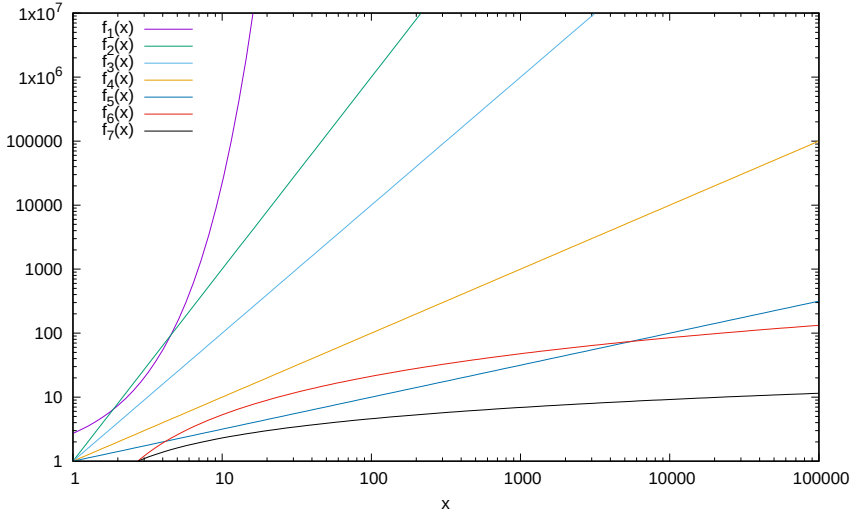
Inklusionen der wichtigsten Aufwandsklassen:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\log^2(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \cdot \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n)$$

Übung 2. Warum geben wir bei der logarithmischen Laufzeit keine Basis des Logarithmus an?

Vergleich der Aufwandsklassen

$f_1 : e^x$
$f_2 : x^3$
$f_3 : x^2$
$f_4 : x$
$f_5 : \sqrt{x}$
$f_6 : \log^2(x)$
$f_7 : \log(x)$



Frage: Welchen Vorteil hat ein 10-mal so schneller Rechner?

GOp/sec	n^2	n^3	n^4	e^n
1	31.622	1.000	177	20
10	100.000	2.154	316	23
100	316.227	4.641	562	25
1000	1.000.000	10.000	1.000	27
$\Delta =$ $\cdot 10$	$\cdot 3,16$	$\cdot 2,15$	$\cdot 1,78$	$+2,3$

$$
\begin{array}{lll}
n^2: & n_1 = \sqrt{1e9} = 31.622 \\
& n_2 = \sqrt{1e10} = \sqrt{1e9} \cdot \sqrt{10} = n_1 \cdot \sqrt{10} = n_1 \cdot 3,16 \\
n^3: & n_1 = \sqrt[3]{1e9} = 1.000 \\
& n_2 = \sqrt[3]{1e10} = \sqrt[3]{1e9} \cdot \sqrt[3]{10} = n_1 \cdot \sqrt[3]{10} = n_1 \cdot 2,15 \\
e^n: & n_1 = \ln(1e9) = 20.723\ldots \\
& n_2 = \ln(1e10) = \ln(1e9 \cdot 10) = \ln(1e9) + \ln(10) = n_1 + \ln(10) = n_1 + 2,3
\end{array}
$$

Vergleich der Aufwandsklassen

Annahme: Der Prozessor führt 10^9 charakteristische Operationen pro Sekunde aus.

Lösbare Problemgröße bei verschiedenen Zeitvorgaben:

Aufwand	1 Sek	1 Min	1 Std	1 Tag
n^2	31.622	244.948	1.897.366	9.295.160
n^3	1.000	3.914	15.326	44.208
n^4	177	494	1.377	3.048
n^5	63	143	324	612
e^n	20	24	28	32

In der Regel bestehen Algorithmen aus verschiedenen Teilen, die unterschiedliche Laufzeiten haben. Um die Laufzeit des gesamten Algorithmus angeben zu können, benötigen wir Rechenregeln.

Übung 3. Beweisen Sie die folgende Aussage.

$\forall g : \mathbb{N} \to \mathbb{N}\ \forall f \in \mathcal{O}(g)\ \forall c \in \mathbb{N}$ gilt:

- $f + c \in \mathcal{O}(g)$: Führe eine Prozedur und eine konstante Anzahl von Operationen aus.
- $f \cdot c \in \mathcal{O}(g)$: Führe eine Prozedur c-mal aus.

Dabei soll $f + c$ die Funktion $f' : \mathbb{N} \to \mathbb{N}$ mit $f'(n) := f(n) + c$ bezeichnen. Analog sei $f \cdot c$ als $f''(n) := f(n) \cdot c$ definiert.

Wir gehen davon aus, dass die Funktionen nicht monoton fallend sind, sondern Laufzeiten von Algorithmen beschreiben.

Übung 4. Beweisen Sie die folgende Aussage.

$\forall g_1, g_2 : \mathbb{N} \to \mathbb{N}\ \forall f_1 \in \mathcal{O}(g_1)\ \forall f_2 \in \mathcal{O}(g_2)$ gilt:

- $f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$: Führe zwei verschiedene Prozeduren hintereinander aus.
- $f_1 \cdot f_2 \in \mathcal{O}(g_1 \cdot g_2)$: Eine Prozedur mit Laufzeit f_2 wird jeweils innerhalb einer Schleife aufgerufen, wobei der Schleifenrumpf die Laufzeit f_1 hat.

Dabei bezeichnet

- $f_1 + f_2$ die Funktion $f' : \mathbb{N} \to \mathbb{N}$ mit $f'(n) := f_1(n) + f_2(n)$
- und $f_1 \cdot f_2$ die Funktion $f''(n) := f_1(n) \cdot f_2(n)$.

Weitere Rechenregeln:

- $f \in \mathcal{O}(g) \ \Rightarrow\ f + g \in \mathcal{O}(g)$
- $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \ \Rightarrow\ f \in \mathcal{O}(h)$
- für festes $k \in \mathbb{N}$ gilt $\Theta(\log(n^k)) = \Theta(\log(n))$.

Oft benötigen wir asymptotische Aufwandsabschätzungen für mehrstellige Funktionen:

- naive Textsuche: Verschiebe ein Muster der Länge m über einen Text der Länge n und prüfe stellenweise auf Gleichheit. Dabei ergibt sich als Laufzeit $(n - (m - 1)) \cdot m$.
- Bei der Tiefensuche auf einem Graphen mit n Knoten und m Kanten ergibt sich eine Laufzeit, die von n und m abhängt.

Wir erweitern daher die Groß-O-Notation auf zweistellige Funktionen $g : \mathbb{N}^2 \to \mathbb{N}$.

$$\mathcal{O}(g) := \left\{ f : \mathbb{N}^2 \to \mathbb{N} \;\middle|\; \begin{array}{l} \exists c \in \mathbb{N}\ \exists n_0 \in \mathbb{N}\ \forall x, y \in \mathbb{N} : \\ x, y \geq n_0 \;\Rightarrow\; f(x, y) \leq c \cdot g(x, y) \end{array} \right\}$$

gegeben: N natürliche Zahlen $a_1, \ldots, a_N$ und ein Wert k
gesucht: Position des Schlüssels k in der Folge
Annahme: Die Zahlen seien in einem Array A gespeichert: $A[1], A[2], \ldots, A[N]$

Zahlen sind nicht sortiert
$\rightarrow$ sequentielle Suche

```
function SeqSearch(k)
    for i := 1 to N do
        if A[i] = k then
            return i
    return −1
```

Zahlen sind aufsteigend sortiert
$\rightarrow$ binäre Suche

```
function BinSearch(k, ℓ, r)
    while ℓ ≤ r do
        p := (ℓ+r)/2
        if A[p] = k then
            return p
        if k < A[p] then
            r := p − 1
        else ℓ := p + 1
    return −1
```

Übung 5. Formulieren Sie die binäre Suche als rekursiven Algorithmus.

Bei einigen Problemen hängt die Laufzeit nicht nur von der Menge der Eingabewerte ab, sondern auch von der Reihenfolge der Werte. Man unterscheidet die Laufzeit

- im besten Fall (best case)
- im Mittel (average case)
- im schlechtesten Fall (worst case)

Vergleich: Lineare Suche vs. binäre Suche

Algorithmus	best case	average case	worst case
lineare Suche	1	$N/2$	N
binäre Suche	1	$\log_2(N)$	$\log_2(N)$

Probleme bei Average-Case-Analyse:

- Worüber bildet man den Durchschnitt?
- Sind alle Eingaben der Länge n gleich wahrscheinlich? (Wir betrachten in dieser Vorlesung nur gleichverteilte Daten.)
- Technisch oft sehr viel schwieriger durchzuführen als worst-case-Analyse.

Murphys Gesetz: Alles was schief gehen kann, wird auch schief gehen.
anders ausgedrückt: Immer wenn ich das Programm ausführe, warte ich ewig.

Ungeeignet für kritische Anwendungen, bei denen maximale Reaktionszeiten garantiert werden müssen. → *Echtzeitsysteme*

Definition: (Worst-Case-Komplexität)

- W_n: Menge der zulässigen Eingaben der Länge n.
- $A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

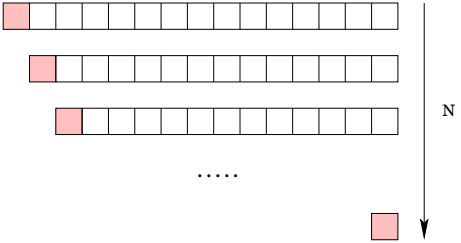
Worst-Case-Komplexität (schlechtester Fall):

$$T_A(n) = \sup\{A(w) \mid w \in W_n\}$$

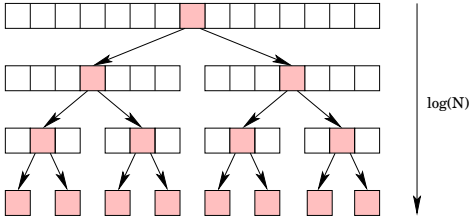
ist eine obere Schranke für die maximale Anzahl der Schritte, die Algorithmus A benötigt, um Eingaben der Größe n zu bearbeiten.

Worst-Case-Komplexität: Beispiel

lineare Suche:



binäre Suche:



Zum Vergleich:

N	$\log(N)$
1.000.000	20
1.000.000.000	30
1.000.000.000.000	40

Average-Case-Komplexität

Definition: (Average-Case Komplexität)

- W_n: Menge der zulässigen Eingaben der Länge n.
- $A(w)$: Anzahl Schritte von Algorithmus A für Eingabe w.

Average-Case-Komplexität (erwarteter Aufwand):

$$\overline{T}_A(n) = \frac{1}{|W_n|} \cdot \sum_{w \in W_n} A(w)$$

ist die mittlere Anzahl von Schritten, die Algorithmus A benötigt, um eine Eingabe der Größe n zu bearbeiten.

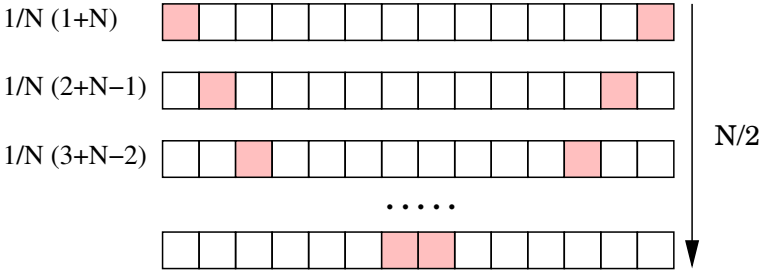
Wir setzen hier eine Gleichverteilung voraus. $\rightarrow$ arithmetischer Mittelwert

Average-Case-Komplexität: Beispiel

lineare Suche:

Kosten: $1, \ldots, N$ Vergleiche

erwartete Kosten: $\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N)$

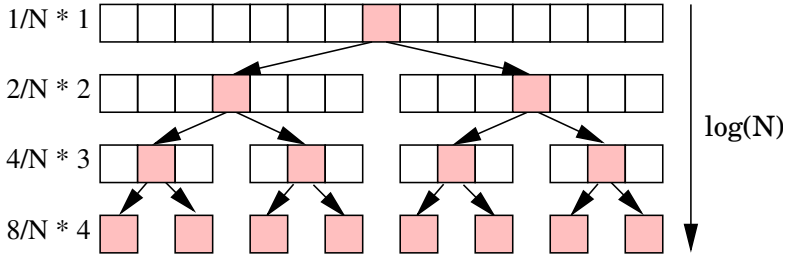


$$\frac{1}{N} \cdot (1 + 2 + 3 + \ldots + N) = \frac{1}{N} \cdot \frac{N(N+1)}{2} = \frac{N+1}{2}$$

Average-Case-Komplexität: Beispiel

binäre Suche:

Kosten: $1, \ldots, \log(N)$
zur Vereinfachung: $N = 2^x - 1$
erwartete Kosten: $\frac{1}{N} \cdot (1 + 2 \cdot 2 + 4 \cdot 3 + \ldots + 2^{x-1} \cdot x)$



Behauptung (Induktionsvermutung I.V.):

$$\sum_{i=1}^{x} i \cdot 2^{i-1} = (x-1) \cdot 2^x + 1$$

Beweis mittels vollständiger Induktion:

Induktionsanfang: $x = 1$

$$1 \cdot 2^0 = 1 \cdot 1 = 1 \stackrel{!}{=} 0 \cdot 2^1 + 1 = 1$$

Induktionsschritt: $x \to x + 1$:

$$\begin{aligned} \sum_{i=1}^{x+1} i \cdot 2^{i-1} &= \sum_{i=1}^{x} i \cdot 2^{i-1} \; + \; (x+1) \cdot 2^x \\ &\stackrel{I.V.}{=} (x-1) \cdot 2^x + 1 \; + \; (x+1) \cdot 2^x \\ &= 2x \cdot 2^x + 1 = x \cdot 2^{x+1} + 1 \end{aligned}$$

Average-Case-Komplexität: Beispiel

Aus der Annahme $N = 2^x - 1$ folgt: $\log_2(N+1) = x$

Somit ergibt sich:

$$\begin{aligned}
\frac{1}{N} \cdot \sum_{i=1}^{x} i \cdot 2^{i-1} &= \frac{1}{N} \cdot [(x-1) \cdot 2^x + 1] \\
&= \frac{1}{N} \cdot [(\log_2(N+1) - 1) \cdot (N+1) + 1] \\
&= \frac{1}{N} \cdot [(N+1) \cdot \log_2(N+1) - N] \\
&\approx \log_2(N+1) - 1 \text{ für große } N
\end{aligned}$$

Im Mittel verursacht binäres Suchen also nur etwa eine Kosteneinheit weniger als im schlechtesten Fall.

Übersicht

Entwurfsmethoden

Neben vielen Einzellösungen gibt es Entwurfsprinzipien, die den Entwurf von Algorithmen erleichtern können. Hier hauptsächlich nur

- Divide-and-Conquer-Algorithmen (Teile und Herrsche), Top-Down
- Greedy-Algorithmen (gierig)
- Dynamisches Programmieren, Bottom-Up
- Backtracking
- Branch-and-Bound
- Lokale Suche

Es gibt viele weitere Entwurfsmethoden wie z.B.

- ganzzahlige Programmierung
- randomisierte Algorithmen
- Fest-Parameter-Algorithmen
- usw.

Divide-and-Conquer

Entwurfsprinzip:

- *Divide* the problem into subproblems.
- *Conquer* the subproblems by solving them recursively.
- *Combine* subproblem solutions.

Beispiele:

- Binäre Suche
- Potenzieren einer Zahl
- Matrix-Multiplikation
- Quicksort (der Sortieralgorithmus der C-Standard-Bibliothek)

Problem: Berechne x^n für ein $n \in \mathbb{N}$.

Einfacher Algorithmus:

```
erg := 1
for i := 1 to n do
    erg := erg * x
```

$\rightarrow$ Laufzeit: $\Theta(n)$ Multiplikationen

Divide-and-Conquer:

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{für } n \text{ gerade und } n > 1 \\ x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x & \text{für } n \text{ ungerade und } n > 1 \\ x & \text{für } n = 1 \end{cases}$$

function POWER(x, n)
  if $n = 1$ **then**
    return x
  if n is even **then**
    $t :=$ POWER(x, $n/2$)
    return $t \cdot t$
  else
    $t :=$ POWER(x, $(n-1)/2$)
    return $t \cdot t \cdot x$

Laufzeit?

Der Exponent wird in jedem Schritt mindestens halbiert.

Eine obere Schranke für die Laufzeit ergibt sich, wenn der Exponent auf die nächst höhere 2er-Potenz erhöht wird. Wir nehmen also an, dass $n = 2^k$ für ein $k \in \mathbb{N}$ gilt.

Außerdem sei $T(1) = c_1$ und der Aufwand für Divide und Combine sei höchstens c_2. Dann gilt wegen $n = 2^k \iff k = \log_2(n)$:

$$\begin{aligned}
T(n) &\leq T(n/2) + c_2 \\
&\leq T(n/4) + c_2 + c_2 = T(n/4) + 2 \cdot c_2 \\
&\leq T(n/8) + c_2 + 2 \cdot c_2 = T(n/8) + 3 \cdot c_2 \\
&\vdots \\
&\leq T(n/2^k) + k \cdot c_2 = T(1) + \log_2(n) \cdot c_2 = c_1 + \log_2(n) \cdot c_2 \\
\Rightarrow T &\in \Theta(\log(n))
\end{aligned}$$

Wir zählen hier die Anzahl der Multiplikationen und gehen davon aus, dass alle Multiplikationen gleich lange dauern.

Laufzeit bei der Unterteilung in k Teile der Größe $n_1, n_2, \ldots, n_k$:

$$T(n) = \begin{cases} c_1 & \text{falls } n \leq n_0 \\ T(n_1) + \ldots + T(n_k) + c_2 & \text{sonst} \end{cases}$$

Für a viele Teile jeweils der Größe n/b und einem Aufwand von $\Theta(n^k)$ für Divide- und Combine gilt also folgende Rekursionsgleichung:

$$T(n) = a \cdot T(n/b) + \Theta(n^k)$$

Das *Master-Theorem* gibt die Lösung dieser Gleichung an, wobei wir $T(1) = c$ für eine Konstante c annehmen und drei Fälle unterscheiden:

- Für $a < b^k$ gilt: $T \in \Theta(n^k)$
- Für $a = b^k$ gilt: $T \in \Theta(n^k \cdot \log(n))$
- Für $a > b^k$ gilt: $T \in \Theta(n^{\log_b(a)})$

Induktive Einsetzungsmethode

- Rate eine Lösung und bestätige diese durch vollständige Induktion.
- *Beispiel:* Für $T(n) = 2 \cdot T(n/2) + b$ "raten" wir $T(n) \leq c \cdot n - a$ als Lösung und rechnen nach:

$$\begin{aligned} T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot (c \cdot \frac{n}{2} - a) + b \\ &\leq c \cdot n - 2a + b \\ &= c \cdot n - a + (b - a) \\ &\leq c \cdot n - a \text{ für } b \leq a \end{aligned}$$

- Problem ist falsches Raten. Nehmen wir bei obigem Beispiel $T \in \mathcal{O}(n)$ an, also $T(n) \leq c \cdot n$, dann erhalten wir:

$$\begin{aligned} T(n) = 2 \cdot T(n/2) + b &\leq 2 \cdot (c \cdot \frac{n}{2}) + b \\ &\leq c \cdot n + b \; \lightning \end{aligned}$$

Iterative Methode

- Setze Rekursionsgleichung fort bis zu einer geschlossenen Form.
- *Beispiel:* $T(n) = 2 \cdot T(n/2) + b$

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + b \\ &= 2 \cdot (2 \cdot T(n/4) + b) + b \;=\; 4 \cdot T(n/4) + 3b \\ &= 4 \cdot (2 \cdot T(n/8) + b) + 3b \;=\; 8 \cdot T(n/8) + 7b \\ &\vdots \\ &= 2^k \cdot T(n/2^k) + (2^k - 1) \cdot b \end{aligned}$$

Für $n = 2^k \iff \log_2(n) = k$ und $T(1) = const$ erhalten wir:

$$T(n) = n \cdot T(1) + (n-1) \cdot b \qquad \rightarrow \qquad T \in \mathcal{O}(n)$$

Variablensubstitution

- Ersetze n durch einen Ausdruck, sodass die Rekursionsgleichung eine bekannte Form bekommt.
- *Beispiel:* Für $T(n) = 2 \cdot T(\sqrt{n}) + \log_2(n)$ erhalten wir mit der Substitution $m = \log_2(n) \iff 2^m = n$ die folgende Rekursionsgleichung:

$$\begin{aligned} T(2^m) &= 2 \cdot T(\sqrt{2^m}) + m \\ &= 2 \cdot T(2^{m/2}) + m \end{aligned}$$

- Löse dann die Gleichung mit einer der ersten beiden Methoden.

Wir lösen die Gleichung aus dem Beispiel der vorigen Seite mit der iterativen Methode.

$$\begin{aligned} T(2^m) &= 2 \cdot T(2^{m/2}) + m \\ &= 2 \cdot [2 \cdot T(2^{m/2^2}) + m/2] + m \;=\; 2^2 \cdot T(2^{m/2^2}) + 2m \\ &= 2^2 \cdot [2 \cdot T(2^{m/2^3}) + m/2^2] + 2m \;=\; 2^3 \cdot T(2^{m/2^3}) + 3m \\ &\vdots \\ &= 2^k \cdot T(2^{m/2^k}) + k \cdot m \end{aligned}$$

Mit $m = 2^k \iff k = \log_2(m)$ erhalten wir $T(2^m) = m \cdot T(2) + \log_2(m) \cdot m$ und daher $T \in \Theta(m + m \cdot \log(m))$.

Weil wir als Substitution $m = \log_2(n)$ gewählt hatten, erhalten wir schließlich $T \in \Theta(\log(n) + \log(n) \cdot \log(\log(n)))$ als Laufzeit.

Wir wollen nun das *Master-Theorem* herleiten. Sei $T(n) = a \cdot T(n/b) + \Theta(n^k)$ und $T(1) \in \mathcal{O}(1)$.

- Für $a < b^k$ gilt: $T \in \Theta(n^k)$
- Für $a = b^k$ gilt: $T \in \Theta(n^k \cdot \log(n))$
- Für $a > b^k$ gilt: $T \in \Theta(n^{\log_b(a)})$

Auf den nächsten Folien wenden wir einiges an, was aus der Mathematik bekannt ist, z.B. die Potenzgesetze:

$$(a^n)^m = a^{n \cdot m} = a^{m \cdot n} = (a^m)^n \qquad \left(\frac{a}{b}\right)^n = \frac{a^n}{b^n} \qquad a^0 = 1 \text{ für } a \neq 0$$

Außerdem gilt für die n-te Partialsumme s_n der geometrischen Reihe:

$$s_n = \sum_{k=0}^{n} q^k = \frac{q^{n+1} - 1}{q - 1} = \frac{1 - q^{n+1}}{1 - q}$$

Wir nutzen die iterative Methode und erhalten:

$$\begin{aligned} T(n) &= a \cdot T(n/b) + cn^k \\ &= a \cdot [a \cdot T(n/b^2) + c(n/b)^k] + cn^k \\ &= a^2 \cdot T(n/b^2) + cn^k \cdot (1 + a/b^k) \\ &= a^2 \cdot [a \cdot T(n/b^3) + c(n/b^2)^k] + cn^k \cdot (1 + a/b^k) \\ &= a^3 \cdot T(n/b^3) + cn^k \cdot (1 + a/b^k + (a/b^k)^2) \\ &\vdots \\ &= a^\ell \cdot T(n/b^\ell) + cn^k \cdot \sum_{i=0}^{\ell-1} (a/b^k)^i \end{aligned}$$

mit $n/b^\ell = 1 \iff n = b^\ell$ und somit $\ell = \log_b(n)$ gilt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

Fall 1: $a = b^k$

$$\begin{aligned} T(n) &= a^{\log_b(n)} + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} 1^i \\ &= (b^k)^{\log_b(n)} + cn^k \cdot \log_b(n) \\ &= (b^{\log_b(n)})^k + cn^k \cdot \log_b(n) \\ &= n^k + cn^k \cdot \log_b(n) \\ \Rightarrow T &\in \Theta(n^k \cdot \log(n)) \end{aligned}$$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

Fall 2: $a < b^k$

$$T(n) = a^{\log_b(n)} + cn^k \cdot \frac{1 - (a/b^k)^{\log_b(n)}}{1 - a/b^k}$$

Es gilt:

- $a^{\log_b(n)} = \left(b^{\log_b(a)}\right)^{\log_b(n)} = \left(b^{\log_b(n)}\right)^{\log_b(a)} = n^{\log_b(a)}$ und wegen $a < b^k$ gilt: $n^{\log_b(a)} < n^{\log_b(b^k)} = n^{k \cdot \log_b(b)} = n^k$
- $\lim_{n\to\infty} \frac{1-(a/b^k)^{\log_b(n)}}{1-a/b^k} = \frac{1}{1-a/b^k} = \textit{konstant}$

und daher gilt: $T \in \Theta(n^k)$

Wir hatten festgestellt:

$$T(n) = a^{\log_b(n)} \cdot T(1) + cn^k \cdot \sum_{i=0}^{\log_b(n)-1} (a/b^k)^i$$

Fall 3: $a > b^k$

$$T(n) = a^{\log_b(n)} + cn^k \cdot \frac{(a/b^k)^{\log_b(n)} - 1}{a/b^k - 1}$$

Da $a/b^k - 1$ konstant ist, können wir eine neue Konstante $\hat{c}$ einführen und es gilt:
$T(n) = n^{\log_b(a)} + \hat{c}n^k \cdot ((a/b^k)^{\log_b(n)} - 1)$

Außerdem gilt:

$$\begin{aligned}(a/b^k)^{\log_b(n)} &= a^{\log_b(n)}/(b^k)^{\log_b(n)} \\ &= n^{\log_b(a)}/(b^{\log_b(n)})^k \\ &= n^{\log_b(a)}/n^k\end{aligned}$$

Damit erhalten wir:

$$\begin{aligned} T(n) &= n^{\log_b(a)} + \hat{c}n^k \cdot ((a/b^k)^{\log_b(n)} - 1) \\ &= n^{\log_b(a)} + \hat{c}n^k \cdot ((n^{\log_b(a)}/n^k) - 1) \\ &= n^{\log_b(a)} + \hat{c} \cdot n^{\log_b(a)} - \hat{c}n^k \end{aligned}$$

Wegen $a > b^k$ gilt $n^{\log_b(a)} > n^{\log_b(b^k)} = n^k$, und wir erhalten schließlich:

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

Beispiele:

- Binäre Suche: $T(n) = 1 \cdot T(n/2) + \Theta(1)$, also $a = 1$, $b = 2$, $k = 0$

$$a = b^k \;\rightarrow\; T \in \Theta(n^k \cdot \log(n)) = \Theta(\log(n))$$

- Merge-Sort (Kapitel 3.4): $T(n) = 2 \cdot T(n/2) + \Theta(n)$, also $a = 2$, $b = 2$, $k = 1$

$$a = b^k \;\rightarrow\; T \in \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

Leider lässt sich das Master-Theorem nicht immer anwenden. Dann muss die Rekursionsgleichung „per Hand“ berechnet werden. Beispiel Quicksort, Kapitel 3.3:

$$T(n) = \frac{1}{n} \sum_{p=1}^{n} (c \cdot n + T(p-1) + T(n-p))$$

Matrix-Multiplikation

- *Eingabe:* zwei $n \times n$-Matrizen A und B
- *Ausgabe:* $C = A \cdot B$

Es gilt:

$$\begin{pmatrix} c_{11} & \cdots & c_{1n} \\ c_{21} & \cdots & c_{2n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ b_{21} & \cdots & b_{2n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix}$$

mit

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

Einfacher Algorithmus: (kein Divide-and-Conquer)

```
for i := 1 to n do
    for j := 1 to n do
        c[i][j] := 0
        for k := 1 to n do
            c[i][j] := c[i][j] + a[i][k] * b[k][j]
```

$\rightarrow$ Laufzeit: $\Theta(n^3)$ Additionen/Multiplikationen

Die Laufzeit ergibt sich aus der Tabellengröße mal Aufwand pro Eintrag: Die Tabelle hat n^2 Einträge, für jeden Eintrag wird eine Summe über n viele Produkte gebildet.

Aufteilen der $n \times n$-Matrizen in jeweils vier $\frac{n}{2} \times \frac{n}{2}$-Matrizen:

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

$$C = A \cdot B$$

mit

$$\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned} \quad \Rightarrow \quad \begin{array}{l} 8 \text{ Multiplikationen von } n/2 \times n/2\text{-Matrizen} \\ 4 \text{ Additionen von } n/2 \times n/2\text{-Matrizen} \end{array}$$

Laufzeit laut Master-Theorem: $T(n) = 8 \cdot T(n/2) + 4 \cdot (n/2)^2 \quad \rightarrow \quad T \in \Theta(n^3)$

Strassens Idee

$$\left(\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right) = \left(\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right) \cdot \left(\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right)$$

mit

$$\begin{aligned} P_1 &= a \cdot (f - h) \\ P_2 &= (a + b) \cdot h \\ P_3 &= (c + d) \cdot e \\ P_4 &= d \cdot (g - e) \\ P_5 &= (a + d) \cdot (e + h) \\ P_6 &= (b - d) \cdot (g + h) \\ P_7 &= (a - c) \cdot (e + f) \end{aligned}$$

und

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

Laufzeit: $T(n) = 7T(n/2) + 18(n/2)^2 \quad \rightarrow \quad T \in \Theta(n^{\log_2(7)}) \approx \Theta(n^{2,807})$

Der zur Zeit schnellste Algorithmus[7] hat eine Laufzeit von $\mathcal{O}(n^{2,397})$.

Vergleich der Laufzeiten unter der Annahme, dass ein Rechner $20 \cdot 10^9$ charakteristische Operationen pro Sekunde ausführen kann:

n	n^3	time	$n^{2,807}$	time	$n^{2,397}$	time
1e3	1e9	<1 s	2,64e8	<1 s	1,55e7	<1 s
10e3	1e12	50 s	1,69e11	9 s	3,87e9	<1 s
100e3	1e15	14 h	1,08e14	90 m	9,66e11	49 s
1.000e3	1e18	2 j	6,95e16	40 t	2,41e14	201 m
10.000e3	1e21	1.585 j	4,46e19	71 j	6,01e16	35 t

s: Sekunden, m: Minuten, h: Stunden, t: Tage, j: Jahre

[7] Don Coppersmith, Shmuel Winograd: Matrix Multiplication via Arithmetic Progressions. Journal of Symbolic Computation 9(3): 251-280 (1990)

Characteristische Größen

Obige Laufzeitabschätzungen zur Matrixmultiplikation zählen nur die Anzahl der Multiplikationen und Additionen. Wir berücksichtigen dabei nicht die Größe der Zahlen!

- Addition zweier Zahlen der Länge ℓ nach Schulmethode: $\mathcal{O}(\ell)$
- Multiplikation zweier Zahlen der Länge ℓ nach
  - Schulmethode: $\mathcal{O}(\ell^2)$
  - Karazuba: $\mathcal{O}(\ell^{\log_2(3)}) = \mathcal{O}(\ell^{1,5849})$
  - Schönhage/Strassen: $\mathcal{O}(\ell \cdot \log(\ell) \cdot \log(\log(\ell)))$

Die Laufzeit der einfachen Matrix-Multiplikation unter Berücksichtigung der Größe der Zahlen ergäbe nach

- Schulmethode: $\mathcal{O}(n^3 \cdot \ell^2)$
- Karazuba: $\mathcal{O}(n^3 \cdot \ell^{\log_2(3)})$
- Schönhage/Strassen: $\mathcal{O}(n^3 \cdot \ell \cdot \log(\ell) \cdot \log(\log(\ell)))$

Karazuba-Algorithmus

Die Zifferntupel seien $X = (x_{2n-1} \dots x_0)_b$ und $Y = (y_{2n-1} \dots y_0)_b$.

Jedes Zifferntupel aufspalten in zwei Tupel der Länge n:

$$X_h = (x_{2n-1} \dots x_n)_b \quad \textit{und} \quad X_l = (x_{n-1} \dots x_0)_b$$
$$Y_h = (y_{2n-1} \dots y_n)_b \quad \textit{und} \quad Y_l = (y_{n-1} \dots y_0)_b.$$

Damit ist $X = X_h b^n + X_l$ und $Y = Y_h b^n + Y_l$ und wir erhalten

$$XY = X_h Y_h b^{2n} + (X_h Y_l + X_l Y_h)\, b^n + X_l Y_l.$$

Den Term $X_h Y_l + X_l Y_h$ in andere Form bringen:

$$\begin{aligned} X_h Y_l + X_l Y_h &= (X_h Y_h + X_h Y_l + X_l Y_h + X_l Y_l) - (X_h Y_h + X_l Y_l) \\ &= (X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l). \end{aligned}$$

Dann sind im Produkt nur noch drei Produkte enthalten:

$$XY = X_h Y_h b^{2n} + ((X_h + X_l)(Y_h + Y_l) - (X_h Y_h + X_l Y_l))\, b^n + X_l Y_l$$

Greedy- (gierige) Algorithmen sind geeignet, um Optimierungsprobleme zu lösen oder zu approximieren.

Es wird zwischen *exakten Greedy-Algorithmen* und *Greedy-Heuristiken* unterschieden.

Beispiele

- Wechselgeldproblem (exakt oder approximativ)
- Rucksackproblem (exakt oder approximativ)
- kürzeste Wege (exakt)
- minimaler Spannbaum (exakt)
- Strategien für das metrische TSP-Problem (approximativ)

Damit eine optimale Lösung gefunden wird, muss das *Optimalitätsprinzip von Bellman* gelten: „Eine optimale Lösung setzt sich aus optimalen Teillösungen zusammen."

Dies ist allerdings nur eine notwendige Bedingung, keine hinreichende. Die Optimalität muss jeweils explizit bewiesen werden.

Vorgehen bzw. Idee:

- Jeder Schritt wird nur aufgrund der „lokal verfügbaren“ Information durchgeführt.
- Es wird aus allen möglichen „Fortsetzungen einer Teillösung“ diejenige ausgewählt, die momentan den besten Erfolg bringt.

→ Es werden also nicht verschiedene Kombinationen von Teillösungen getestet, sondern nur eine einzige ausgewählt. Diese Lösung ist ggf. nicht optimal.

Wechselgeldproblem (coin changing problem, change making problem)

gegeben: beliebig viele Münzen der Werte $(d_1, \ldots, d_k) \in \mathbb{N}^k$ sowie ein Betrag v
gesucht: Vektor $c = (c_1, \ldots, c_k) \in \mathbb{N}_0^k$ mit $\sum_{i=1}^{k} c_i \cdot d_i = v$ und $\sum_{i=1}^{k} c_i$ minimal.
anders ausgedrückt: Herausgabe von Wechselgeld mit möglichst wenigen Münzen

Beispiel: Verfügbare Münzen mit Werten zu 25, 10, 5, und 1 Cent
Ziel: Gebe den Betrag $v = 88$ Cent zurück mit so wenigen Münzen wie möglich

$$88c = 3 \cdot 25c + 1 \cdot 10c + 0 \cdot 5c + 3 \cdot 1c \rightarrow 7 \text{ Münzen}$$

mögliches Vorgehen zum Bestimmen einer Lösung:

i	d_i	v_i	$c_i := v_i \text{ div } d_i$	$v_{i+1} := v_i \bmod d_i$
1	25	88	3	13
2	10	13	1	3
3	5	3	0	3
4	1	3	3	0

Übung 6. Formulieren Sie einen iterativen und einen rekursiven Algorithmus für das Wechselgeldproblem. Implementieren Sie die beiden Algorithmen und testen Sie die Programme für einige Eingaben.

Welche Laufzeit haben die Algorithmen?

Ohne Beweis: Für unser europäisches Münzsystem sowie für das US-amerikanische Münzsystem liefert der Greedy-Algorithmus optimale Lösungen.

Für allgemeine Münzsysteme wird nicht notwendigerweise ein Optimum gefunden:

- Münzen: 11, 5 und 1 Cent
- Zielwert $v = 15$ Cent
- Greedy-Lösung: $15c = 1 \cdot 11c + 4 \cdot 1c \rightarrow 5$ Münzen
- optimale Lösung: $15c = 3 \cdot 5c \rightarrow 3$ Münzen

Problem der Handlungsreisenden: (Traveling Salesperson Problem - TSP)

- gegeben: n Orte sowie Kosten $c(i, j)$, um vom Ort i nach Ort j zu reisen
- gesucht: Eine preiswerteste Rundreise, die alle Orte genau einmal besucht und wieder am Start endet, also eine Permutation p, sodass die Summe der Kosten $\sum_{i=1}^{n-1} c(p(i), p(i+1)) + c(p(n), p(1))$ minimal ist.

Greedy-Algorithmus: Beginne mit dem ersten Ort. Wähle als nächsten Ort einen noch nicht besuchten Ort, der möglichst nahe am zuletzt besuchten Ort liegt.

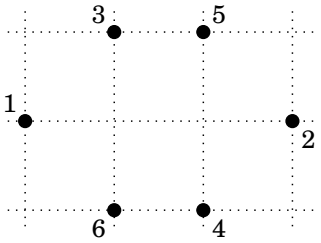
$p[1] := 1$
for $i := 2, \ldots, n$ **do**
    Sei j der nächstgelegene, unbesuchte Knoten zu $p[i-1]$
    $p[i] = j$

Laufzeit?

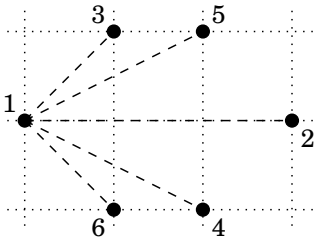
Um die Laufzeit zu bestimmen, ist der obige Algorithmus nicht präzise genug formuliert.

for $i := 1, \ldots, n$ **do**
    $m[i] := 0$ ▷ initial: alle Knoten als unbesucht markieren
$p[1] := 1$
for $i := 2, \ldots, n$ **do**
    $min := \infty$
    for $j := 2, \ldots, n$ **do**
        if $m[j] = 0$ and $c(p[i-1], j) < min$ **then**
            $p[i] = j$
            $min := c(p[i-1], j)$
    $m[p[i]] = 1$

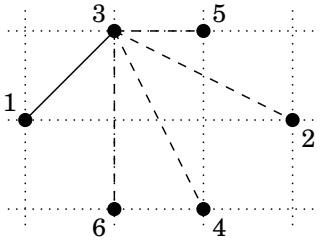
Laufzeit: $\mathcal{O}(n^2)$



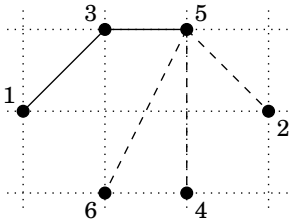
initial



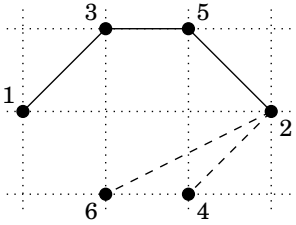
p[1] = 1



p[1] = 1, p[2] = 3



p[1] = 1, p[2] = 3, p[3] = 5



p[1] = 1, p[2] = 3, p[3] = 5, p[4] = 2

und so weiter

Übung 7. Für das Problem des Handlungsreisenden seien folgende Entfernungen gegeben:

- Für $i = 1, \ldots, n-1$ gelte $c(i, i+1) = 1$.
- Sei $c(n, 1) = M$ für eine große Zahl M.
- Sonst gelte $c(i, j) = 2$.

Fragen:

- Wie sieht eine optimale Tour aus?
- Welche Tour berechnet der Greedy-Algorithmus?
- Wie groß ist der absolute Fehler?

Präfixcode nach Huffman

- *Gegeben:* Ein Alphabet $\Sigma = \{a_1, a_2, \ldots, a_n\}$ und die Wahrscheinlichkeit p_i für das Zeichen a_i.
- *Gesucht:* Optimaler Präfixcode für die Wahrscheinlichkeitsverteilung.

Bei einem *Präfixcode* ist kein Codewort $c(a_i)$ Anfangsstück (Präfix) eines anderen Codeworts $c(a_j)$. Dadurch ist das Dekodieren einfach.

Ziel: Minimiere die mittlere Codewortlänge.

$$\sum_{i=1}^{n} p_i \cdot |c(a_i)| \quad \longrightarrow \quad \min$$

Mit $|c(a_i)|$ bezeichnen wir die Länge des Codewortes von a_i.

Leider werden die Begriffe *Codieren* und *Chiffrieren* meist nicht sauber unterschieden und umgangssprachlich oft synonym verwendet.

Codierung: Abbildung $f : A \to B$ zwischen zwei Alphabeten A und B.

Morse-Code:

'a'	$\cdot -$	'd'	$- \cdot \cdot$	'g'	$- - \cdot$	'j'	$\cdot - - -$
'b'	$- \cdot \cdot \cdot$	'e'	$\cdot$	'h'	$\cdot \cdot \cdot \cdot$	'k'	$- \cdot -$
'c'	$- \cdot - \cdot$	'f'	$\cdot \cdot - \cdot$	'i'	$\cdot \cdot$	'l'	$\cdot - \cdot \cdot$

Problem: Der Morse-Code ist kein Präfixcode. So ist etwa $f(\text{e})$ ein Präfix von $f(\text{'i'})$ und $f(\text{'h'})$, außerdem ist $f(\text{'a'})$ ein Präfix von $f(\text{'l'})$ usw.

$$f(\text{'h'}) = f(\text{'ii'}) = f(\text{'eeee'}) = f(\text{'eie'})$$

Daher wird beim Morse-Code ein zusätzliches Zeichen eingeführt, die Pause. Dadurch ist die Umkehrfunktion eindeutig, also das Dekodieren möglich:

$$f^{-1}(\cdot - \ \lrcorner \ \cdot \cdot) = \text{'ai'} \qquad \text{oder} \qquad f^{-1}(\cdot - \cdot \ \lrcorner \ \cdot) = \text{'re'}$$

Problem hier sind die unterschiedlich langen Code-Wörter.

Blockcode: Jedes Code-Wort besteht aus einer festen Anzahl n von Zeichen.

American Standard Code for Information Interchange (ASCII): 8-Bit

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	‘	j	t	~
7:	%	/	9	C	M	W	a	k	u	DEL
8:	&	0	:	D	N	X	b	l	v	
9:	’	1	;	E	O	Y	c	m	w	

Werden Blockcodes[8] zur Kanalkodierung eingesetzt, dann nutzt man in der Regel fehlerkorrigierende Codes:

Zeichen	Codewort
0	000
1	111

Fehlerhaftes Codewort	Korrigiertes Codewort	Zugeordnetes Zeichen
00$\underline{1}$	000	0
0$\underline{1}$0	000	0
$\underline{1}$00	000	0
$\underline{0}$11	111	1
1$\underline{0}$1	111	1
11$\underline{0}$	111	1

Durch Hinzufügen zusätzlicher Symbole in die Codewörter entsteht Redundanz und die Informationsrate sinkt, allerdings kann der Empfänger die redundanten Informationen nutzen, um Übertragungsfehler zu erkennen und ggf. zu korrigieren. (Quelle: wikipedia)

[8] Mehr zu linearen Codes und was diese mit linearer Algebra zu tun haben, können Sie in dem Buch *Goebbels, Rethmann, Einführung in die Mathematik an Beispielen aus der Informatik, Springer Spektrum, 2023,* nachlesen.

Verschlüsselung (Chiffrierung): Dient zur Geheimhaltung von Informationen. Dazu wird die Information (der Klartext) in einen Geheimtext (auch Chiffrat) umgewandelt. Außerdem soll erkannt werden, ob Dritte die Information verändert haben.

Minimalziele laut dem Bundesamt für Sicherheit in der Informationstechnik (BSI):

- Die Verschlüsselung und Entschlüsselung von Texten muss einfach sein, wenn der Schlüssel bekannt ist.
- Ohne Kenntnis des Schlüssels soll für einen Angreifer eine Entschlüsselung von Nachrichten auch dann nicht praktisch möglich sein, wenn er über beträchtliche Mittel verfügt und das Verfahren kennt.

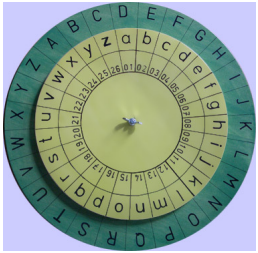
Zum Ver- und Entschlüsseln wird ein Schlüssel benötigt.

- symmetrisch: Zum Ver- und Entschlüsseln wird derselbe Schlüssel genutzt, der zuvor auf einem sicheren Weg ausgetauscht werden muss. Beispiel: AES
- asymmetrisch: Zum Verschlüsseln wird ein anderer Schlüssel genutzt als zum Entschlüsseln. Beispiel: RSA

Caesar-Code: Ersetzte jeden Buchstaben des Textes durch den Buchstaben, der im Alphabet n Stellen danach kommt.

Beispiel: Für $n = 2$ wird aus dem Klartext `ELEFANTEN` der verschlüsslete Text `GNGHCPVGP`.

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
CDEFGHIJKLMNOPQRSTUVWXYZAB
```



Codebrechen ist leicht mittels Häufigkeitstabelle:

Buchstabe	Häufigkeit
E	17,40%
N	9,78%
I	7,55%
S	7,27%
R	7,00%

Paare	Häufigkeit
ER	4,09%
EN	4,00%
CH	2,42%
DE	1,93%
EI	1,87%

Für die Vigenère-Chiffre[9] wird eine quadratische Matrix genutzt, die aus allen möglichen „Caesar-verschobenen“ Alphabeten besteht:

```
  ABCDEFGHIJKLMNOPQRSTUVWXYZ
---------------------------
A ABCDEFGHIJKLMNOPQRSTUVWXYZ
B BCDEFGHIJKLMNOPQRSTUVWXYZA
C CDEFGHIJKLMNOPQRSTUVWXYZAB
D DEFGHIJKLMNOPQRSTUVWXYZABC
E EFGHIJKLMNOPQRSTUVWXYZABCD
F FGHIJKLMNOPQRSTUVWXYZABCDE
G GHIJKLMNOPQRSTUVWXYZABCDEF
  ...
X XYZABCDEFGHIJKLMNOPQRSTUVW
Y YZABCDEFGHIJKLMNOPQRSTUVWX
Z ZABCDEFGHIJKLMNOPQRSTUVWXY
```

Verschlüsseln:
Klartext: Spalte Buchstabe K
Schlüssel: Zeile Buchstabe E
Der Kreuzungspunkt in der Matrix liefert dann die Chiffre: Buchstabe O

Entschlüsseln:
Chiffre: Buchstabe O
Schlüssel: Zeile Buchstabe E
Die Spalte, wo Buchstabe O gefunden wird, liefert den Klartext: Buchstabe K

[9] https://de.wikipedia.org/wiki/Vigen%C3%A8re-Chiffre

Soll die Nachricht „TreffenUmAchtzehnUhrUnterDerAltenBruecke“ mit dem Schlüssel „totalgeheim“ verschlüsselt werden, dann muss der Schlüssel durch Wiederholungen verlängert werden:

```
totalgeheimtotalgeheimtotalgeheimtotalge
TreffenUmAchtzehnUhrUnterDerAltenBruecke
----------------------------------------
MFXFQKRBQIOAHSESTYOVCZMSKDPXESXMZUFNENQI
```

Verschlüsseln und entschlüsseln erfolgt zeichenweise, wie oben erklärt.

Wird der Schlüssel zufällig und so lang wie der zu verschlüsselnde Text gewählt, dann entsteht aus dem obigen Verfahren das One-Time-Pad[10], ein unknackbares Verfahren.

Kryptographie wird im Wahlfach „Sicherheit und Zugriffskontrolle“ behandelt.

[10] `https://de.wikipedia.org/wiki/One-Time-Pad`

Beispiel: Sei $\Sigma = \{a, b, c, d, e, f\}$ und

$$p(a) = 0,45 \qquad p(b) = 0,13 \qquad p(c) = 0,12$$
$$p(d) = 0,16 \qquad p(e) = 0,09 \qquad p(f) = 0,05$$

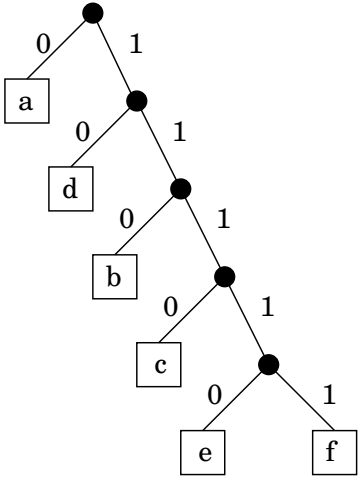
Präfixcode 1: $a \mapsto 0$ $\quad b \mapsto 110$ $\quad c \mapsto 1110$
$d \mapsto 10$ $\quad e \mapsto 11110$ $\quad f \mapsto 11111$

0 11110 10 0 0 11111 10 0 1110 110 = a e d a a f d a c b

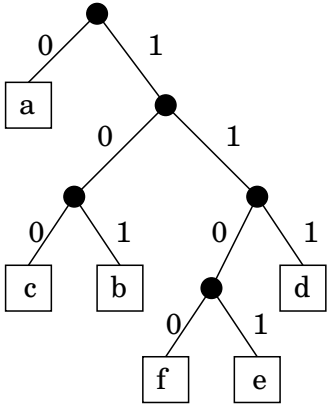
Präfixcode 2: $a \mapsto 0$ $\quad b \mapsto 101$ $\quad c \mapsto 100$
$d \mapsto 111$ $\quad e \mapsto 1101$ $\quad f \mapsto 1100$

0 1101 111 0 0 1100 111 0 100 101 = a e d a a f d a c b

Jeder Code lässt sich in Form eines Codebaums darstellen:



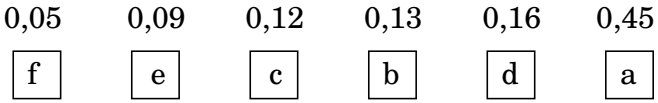
Mittlere Codewortlänge: 2,34



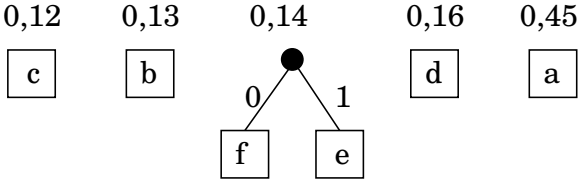
Mittlere Codewortlänge: 2,24

Greedy: Fasse jeweils zwei Bäume zusammen, indem man diejenigen mit den kleinsten Werten auswählt und eine gemeinsame Wurzel erstellt.

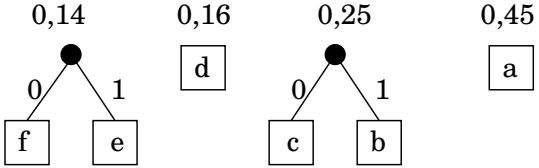
Initial: Jedes Symbol ist ein eigener Baum.



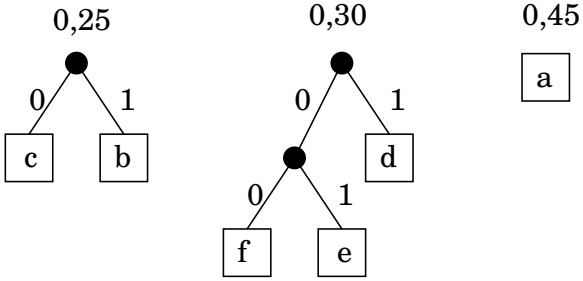
1. Schritt:



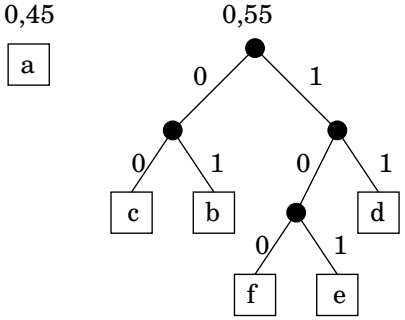
2. Schritt:



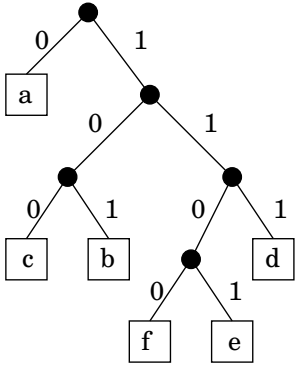
3. Schritt:



4. Schritt:



5. Schritt:



Korrektheit? $\rightarrow$ Uwe Schöning: Algorithmen – kurz gefasst.

Problem der Auswahl von Aktivitäten:

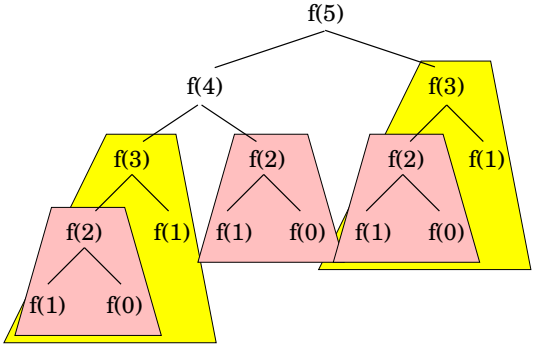
- Gegeben: Eine Menge $S = \{a_1, \ldots, a_n\}$ von n Aktivitäten, die alle die gleiche Ressource benötigen. Eine Aktivität a_i hat stets einen Beginn b_i und ein Ende e_i.
- Gesucht: Eine möglichst große Menge paarweise kompatibler Aktivitäten. Zwei Aktivitäten a_i und a_j heißen kompatibel, wenn sich die Zeitintervalle nicht überschneiden, also $[b_i, e_i) \cap [b_j, e_j) = \emptyset$ gilt.
- Annahme: Die Aktivitäten sind anhand der Endzeitpunkte aufsteigend sortiert, also $e_1 \leq e_2 \leq \ldots \leq e_n$.

Übung 8. Formulieren Sie einen Algorithmus nach folgender Idee: Wähle stets diejenige Aktivität mit frühestem Endzeitpunkt, die legal eingeplant werden kann!

Welche Laufzeit hat Ihr Algorithmus? Liefert das Greedy-Verfahren eine optimale Lösung?

Motivation: Berechne die Fibonacci-Zahl F_n für ein $n \in \mathbb{N}_0$ rekursiv (top down) mittels des Rekursionsschrittes $F_n = F_{n-1} + F_{n-2}$ sowie den Rekursionsanfängen $F_0 = 0$ und $F_1 = 1$.

```
long fibo(unsigned int n) {
    long f1, f2;

    if (n <= 1)
        return n;

    f1 = fibo(n-1);
    f2 = fibo(n-2);
    return f1 + f2;
}
```



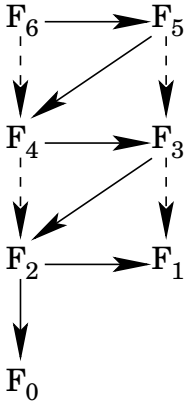
Problem: Zwischenlösungen werden mehrfach berechnet! Die Laufzeit ist daher sehr hoch!

Lösung: Speichere bereits berechnete Zwischenlösungen in einer Tabelle. Diese Technik nennt man *Memorieren*. Es ist immer noch ein Top-Down-Ansatz.

```
long fibo(unsigned n) {
   long f1, f2;

   if (fib[n-1] >= 0)
      f1 = fib[n-1];
   else {
      f1 = fibo(n-1);
      fib[n-1] = f1;
   }
   if (fib[n-2] >= 0)
      f2 = fib[n-2];
   else {
      f2 = fibo(n-2);
      fib[n-2] = f2;
   }

   return f1 + f2;
}
```

Frage: Wie kann das Programm vereinfacht werden?

Die zweite Fallunterscheidung kann entfallen: Nachdem `fibo(n-1)` berechnet wurde, ist in `fib[n-2]` der korrekte Wert gespeichert. Dadurch ergibt sich folgende Struktur der rekursiven Aufrufe:



Gestrichelte Linien stellen den Zugriff auf bereits gespeicherte, memorierte Werte dar.

Bottom-Up-Ansatz: Aus Rekursion wird Iteration, indem aus kleinen Teillösungen größere Lösungen berechnet werden.

Die Berechnung beginnt beim Rekursionsende, da nur diese Werte initial bekannt sind. Während der Berechnung muss sichergestellt werden, dass die in diesem Schritt benötigten Teillösungen bereits berechnet wurden.

```
long fibo(unsigned int n) {
    fib[0] = 0;
    fib[1] = 1;
    for (int i = 2; i <= n; i++)
        fib[i] = fib[i-1] + fib[i-2];

    return fib[n];
}
```

```
fib[2] = fib[1] + fib[0]
fib[3] = fib[2] + fib[1]
fib[4] = fib[3] + fib[2]
fib[5] = fib[4] + fib[3]
fib[6] = fib[5] + fib[4]
fib[7] = fib[6] + fib[5]
...
```

Allerdings ist dies ein untypisches Beispiel für dynamische Programmierung: Es fehlt die Optimierung!

Im Gegensatz zur Greedy-Methode speichert die Methode der dynamischen Programmierung alle bereits berechneten Teillösungen in einer Tabelle (Bottom-Up).

- Aus den Teillösungen wird die Gesamtlösung zusammengesetzt, indem alle Kombinationen von Teillösungen betrachtet werden und die optimale Lösung ausgewählt wird.

$\rightarrow$ Die Laufzeit ist in der Regel höher als bei einem Greedy-Ansatz.

Auch bei der dynamischen Programmierung wird nur dann eine optimale Lösung berechnet, wenn das *Optimalitätsprinzip nach Bellman* gilt: „Die optimale Lösung eines Teilproblems setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.“

Beispiel: Rekursiver Ansatz für das Wechselgeldproblem, wenn es mit Greedy nicht optimal zu lösen ist. Gegeben seien die Münzen $d_1 = 11$, $d_2 = 5$ und $d_3 = 1$ sowie der Betrag $p = 15$.

In einem Schritt werden alle möglichen Münzen gewählt und die optimale Darstellung des Restbetrags (rekursiv) berechnet:

- Wähle d_1 $\rightarrow$ Restbetrag $15 - 11 = 4$ benötigt 4 Münzen: $4 = 4 \cdot 1$
  - $\rightarrow$ Kombination benötigt 5 Münzen, einmal d_1 sowie die Münzen für den Restbetrag.
- Wähle d_2 $\rightarrow$ Restbetrag $15 - 5 = 10$ benötigt 2 Münzen: $10 = 2 \cdot 5$
  - $\rightarrow$ Kombination benötigt 3 Münzen, einmal d_2 sowie die Münzen für den Restbetrag.
- Wähle d_3 $\rightarrow$ Restbetrag $15 - 1 = 14$ benötigt 4 Münzen: $14 = 1 \cdot 11 + 3 \cdot 1$
  - $\rightarrow$ Bei dieser Kombination werden 5 Münzen benötigt.

In jedem Schritt (auch bei der Rekursion) wird immer der minimale Wert gewählt.

Betrachten wir ein Münzsystem mit den Werten $d_1, \ldots, d_k$. Sei $C(p)$ die minimale Anzahl an Münzen, um den Betrag p auszuzahlen. Rekursiver Ansatz:

$$C(p) = \begin{cases} 0 & \text{if } p = 0 \\ \min\limits_{i:d_i \leq p} \{1 + C(p - d_i)\} & \text{if } p > 0 \end{cases}$$

Beim Buttom-Up-Ansatz für das Wechselgeldproblem beginnt die Berechnung beim Rekursionsende, also $p = 1$:

$C[0] := 0$
for $p := 1$ to n **do**
  $min := \infty$
  for $i := 1$ to k **do**
    if $p - d[i] \geq 0$ **and** $1 + C[p - d[i]] < min$ **then**
      $min := 1 + C[p - d[i]]$
  $C[p] := min$

Frage: Gilt das Optimalitätsprinzip von Bellman?

Antwort: Das Optimalitätsprinzip von Bellman gilt. Beispiel:

- $(20, 10, 5, 2, 1)$ ist optimal für das deutsche Münzsystem und den Betrag 38 Cent.
- Dann muss $(10, 5, 2, 1)$ optimal sein für $38 - 20$ Cent, denn gäbe es eine Lösung (b_1, b_2, b_3) mit weniger Münzen und $b_1 + b_2 + b_3 = 18$ Cent, dann wäre $(20, b_1, b_2, b_3)$ eine bessere Lösung für 38 Cent.

Wir führen also einen Beweis durch Widerspruch:

- Sei $(a_1, \ldots, a_m)$ mit $a_i \in \{d_1, \ldots, d_k\}$ eine optimale Lösung für den Wert n, wobei $\sum_{i=1}^{m} a_i = n$ gilt, also alle Münzen zusammen den Wert n ergeben, und die Münzen aus dem gegebenen Wertebereich sind.
- Wenn wir die Münze a_1 wegnehmen, dann ist $(a_2, \ldots, a_m)$ eine optimale Lösung für den Wert $n - a_1$.

  Andernfalls gäbe es eine Lösung $(b_1, \ldots, b_k)$ mit $k < m - 1$ für den Wert $n - a_1$. Dann wäre aber $(a_1, b_1, b_2, \ldots, b_k)$ eine bessere Lösung für den Wert n. $\lightning\lightning$

Das Rucksackproblem

Packe eine Teilmenge von n Objekten der Größen $g_1, \ldots, g_n$ und den Werten $w_1, \ldots, w_n$ so in einen Rucksack der Größe G, dass der Gesamtwert maximal ist.

Wir unterscheiden zwei Arten:

- Beim 0/1-Rucksackproblem dürfen nur ganze Objekte verpackt werden,
- während beim Bruchteil-Rucksackproblem auch Teile eines Objektes verpackt werden dürfen.

Gesucht wird in diesem Problem also ein Vektor $(a_1, \ldots, a_n)$

- mit $a_i \in \{0, 1\}$ (beim 0/1-Rucksackproblem) bzw.
- mit $0 \leq a_i \leq 1$ (beim Bruchteil-Rucksackproblem)

und

$$\sum_{i=1}^{n} a_i g_i \leq G \quad \text{und} \quad \sum_{i=1}^{n} a_i w_i \to \max$$

Das Bruchteil-Rucksackproblem kann mit einem Greedy-Algorithmus optimal gelöst werden. Sortiere zuerst die Objekte, sodass gilt

$$\frac{w_1}{g_1} \geq \frac{w_2}{g_2} \geq \cdots \geq \frac{w_n}{g_n}$$

Optimale Lösung, falls höchstens die ersten k Objekte ganz in den Rucksack passen: $(1, \ldots, 1, b, 0 \ldots 0)$ von Objekten mit $b = (G - \sum_{i=1}^{k} g_i)/g_{k+1}$

Beispiel:

$$g_1 = 1, \quad g_2 = 2, \quad g_3 = 3, \quad G = 5, \quad w_1 = 3, \quad w_2 = 5, \quad w_3 = 6,$$

Greedy:

$$w_1/g_1 = 3 \qquad w_2/g_2 = 2.5 \qquad w_3/g_3 = 2$$

d.h. $\vec{a} = (1, 1, 2/3)$ und $w = 3 + 5 + 6 \cdot 2/3 = 12$.

Falls keine Bruchstücke zugelassen werden, würde der Greedy-Algorithmus auch zuerst Objekt 1, dann Objekt 2 einpacken und käme auf einen Gesamtwert von 8.

Die optimale Wahl beim 0/1-Rucksackproblem sind die Objekte 2 und 3 mit einem Wert von $w = 11$. Dies kann mit Hilfe der Dynamischen Programmierung gezeigt werden, indem nacheinander alle Kombinationen von Objekten getestet werden, bei einem Objekt beginnend, dann zwei Objekte usw.

Rekursiver Algorithmus: Sei $knap(h, i)$ der maximale Wert, der mit den Objekten $i, \ldots, n$ und Rucksackgröße h erreicht werden kann. Dann gilt für $i < n$ und $h \geq g_i$:

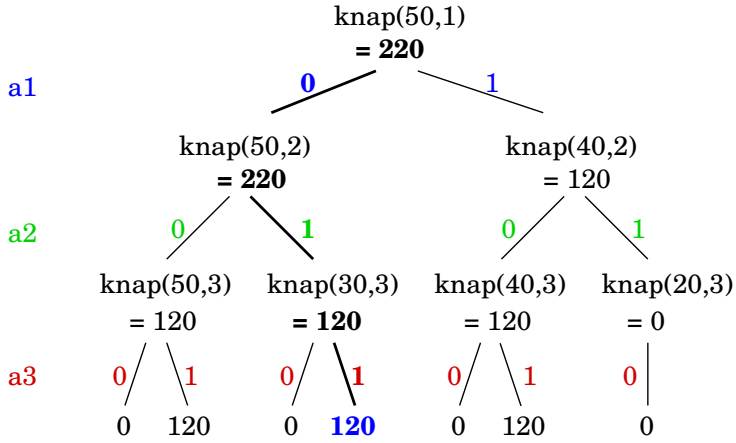
$$knap(h, i) = \max\{knap(h, i+1),\ w_i + knap(h - g_i, i+1)\}$$

Sonst gilt:

$$knap(h, i) = \begin{cases} knap(h, i+1), & i < n, h < g_i \\ 0, & i = n, h < g_n \\ w_n, & i = n, h \geq g_n \end{cases}$$

Initialer Auruf: $knap(G, 1)$

Beispiel: $g = (10, 20, 30)$, $w = (60, 100, 120)$, $G = 50$



Es werden (rekursiv) alle möglichen Kombinationen von Teillösungen getestet.

Bottom-Up-Strategie: Zuerst werden die Werte berechnet, für die keine Teillösungen benötigt werden, also die Werte des Rekursionsendes, Start also bei $i = n$.

```
1: function DynamicKP(w⃗, g⃗, G)
2:     for i := n, ..., 1 do
3:         for h := 0, ..., G do
4:             berechne knap(h, i) nach obiger Formel (ohne Funktionsaufrufe)
```

In Zeile 4 wird keine Funktion aufgerufen, sondern der Algorithmus greift auf die Werte eines zweidimensionalen Arrays zu, die natürlich zuvor berechnet werden mussten.

→ Laufzeit in $\mathcal{O}(n \cdot G)$, aber Achtung: Die Laufzeit ist abhängig vom Zahlenwert G, nicht nur abhängig von der Anzahl n der Elemente!

Beispiel: Betrachten Sie zunächst die Laufzeit zur obigen Eingabe $\vec{g} = (1, 2, 3)$, $\vec{w} = (6, 10, 12)$ und $G = 5$.

Wie ändert sich die Laufzeit für $\vec{g'} = (100, 200, 300)$ und $G' = 500$ bei gleichen Werten $\vec{w}$? Wie viel Speicherplatz wird benötigt?

Details des Beispiels: $\vec{g} = (1, 2, 3)$, $\vec{w} = (6, 10, 12)$, $G = 5$

$i \backslash h$	0	1	2	3	4	5
3	0	0	0	12	12	12
2	-	-	-	-	-	-
1	-	-	-	-	-	-

$i = 3$, $g_3 = 3$, $w_3 = 12$

$$k[h,3] = \begin{cases} 0, & h < g_3 \\ w_3, & h \geq g_3 \end{cases}$$

$i \backslash h$	0	1	2	3	4	5
3	0	0	0	12	12	12
2	0	0	10	12	12	22
1	-	-	-	-	-	-

$i = 2$, $g_2 = 2$, $w_2 = 10$

$$k[h,2] = \begin{cases} k[h,3], & \text{falls } h < g_2 \\ \max\{k[h,3],\ w_2 + k[h-g_2,3]\} \end{cases}$$

$i \backslash h$	0	1	2	3	4	5
3	0	0	0	12	12	12
2	0	0	10	12	12	22
1	0	6	10	16	18	22

$i = 1$, $g_1 = 1$, $w_1 = 6$

$$k[h,1] = \begin{cases} k[h,2], & \text{falls } h < g_1 \\ \max\{k[h,2],\ w_1 + k[h-g_1,2]\} \end{cases}$$

Das Optimalitätsprinzip von Bellman gilt auch beim 0/1-Rucksackproblem. Wieder einmal zeigen wir diese Aussage durch einen Widerspruchsbeweis.

- Sei $(\vec{w}, \vec{g}, G)$ eine Eingabe des Rucksackproblems mit n Objekten und sei $\vec{a} = (a_1, \ldots, a_n)$ ein 0-1-Vektor, also $a_i \in \{0, 1\}$, der eine *optimale* Lösung beschreibt.
- Dann gilt für jedes Objekt i mit $a_i = 1$, dass auch $\vec{a'}$ eine optimale Lösung für $(\vec{w'}, \vec{g'}, G - g_i)$ ist. Dabei entstehen die neuen Vektoren aus den alten, indem das i-te Element gestrichen wird.
- Denn gäbe es eine bessere Lösung für $(\vec{w'}, \vec{g'}, G - g_i)$ als die Lösung $\vec{a'}$, dann würden wir diese nutzen, das i-te Element hinzunehmen und hätten eine *bessere* Lösung als $\vec{a}$ für das gegebene Problem. $\lightning\lightning$

Levenshtein-Distanz: Bezeichnet ein Maß für den Unterschied zwischen zwei Zeichenketten A und B bezüglich der minimalen Anzahl der Operationen Einfügen, Löschen und Ersetzen, um die eine Zeichenkette in die andere zu überführen.

Die Levenshtein-Distanz $D(A, B)$ zweier Wörter A und B wird auch als Edit-Distanz, Editierdistanz oder Editierabstand bezeichnet.

Beispiel: Die Edit-Distanz von `Tier` und `Tor` ist zwei.

`Tier` $\rightarrow$ `Toer` (ersetze i durch o) $\rightarrow$ `Tor` (Lösche e)

In der Praxis: Bestimmen der Ähnlichkeit von Zeichenketten zur Rechtschreibprüfung oder bei einer Duplikaterkennung.

Eigenschaften der Editierdistanz: Bezeichne ϵ das leere Wort, also ein Wort der Länge null, nicht das Leerzeichen. Außerdem seien a und b zwei verschiedene Zeichen, also unterschiedliche Wörter jeweils der Länge eins. Dann gilt für die Levenshtein-Distanz:

$$D(\epsilon, a) = 1 \quad \text{und} \quad D(a, \epsilon) = 1 \quad \text{und} \quad D(a, b) = 1$$

Wir nehmen an, dass die Kosten jeder einzelnen Operation gleich eins ist. Manche setzen für die Ersetzung eines Zeichens höhere Kosten an als für das Einfügen oder Löschen eines Zeichens.

Es gelten zwei triviale Schranken. Sei $|A| = n$ bzw. $|B| = m$ die Länge der Wörter.

- $D(A, B) \leq n + m$: Entferne die Zeichen von A und füge die Zeichen von B ein.
- $D(A, B) \geq |n - m|$: Ist A länger als B, dann müssen mindestens $n - m$ Zeichen entfernt werden, sonst müssen mindestens $m - n$ Zeichen hinzugefügt werden.

Weiterhin gilt die Dreiecksungleichung: Die Editierdistanz zwischen zwei Zeichenketten A und B kann nicht größer sein als die Editierdistanz zwischen A und C plus der Editierdistanz zwischen C und B, wobei C eine beliebige Zeichenkette ist.

$$D(A, B) \leq D(A, C) + D(C, B)$$

Warum? Weil die Editierdistanz die minimalen Kosten angibt.

Rekursiver Algorithmus: Seien $A = (a_1, \ldots, a_n)$ und $B = (b_1, \ldots, b_m)$ zwei Wörter. Sei $D_{i,j}(A, B)$ der minimale Editierabstand zwischen den Teilwörtern bis Position i bzw. j. Dann gilt initial:

$$D_{0,0}(A, B) = 0, \qquad D_{0,j}(A, B) = j, \qquad D_{i,0}(A, B) = i$$

Außerdem gilt:

$$D_{i,j}(A, B) = \min \left\{ \begin{array}{ll} D_{i,j-1}(A, B) + 1 & \text{für } b_j \text{ einfügen} \\ D_{i-1,j}(A, B) + 1 & \text{für } a_i \text{ löschen} \\ D_{i-1,j-1}(A, B) + c & \text{sonst} \end{array} \right\}$$

Dabei ist $c = 0$, falls $a_i = b_j$ gilt, sonst ist $c = 1$ (ersetze a_i durch b_j).

Frage: Gilt das Optimalitätsprinzip? Dürfen wir dynamische Programmierung für dieses Problem nutzen?

Antwort: Für das Problem der Levenshtein-Distanz gilt das Optimalitätsprinzip von Bellman. Ein Editierpfad von x nach y ist eine Folge

$$x = x_0, x_1, x_2, \ldots, x_\ell = y,$$

so dass $x_{i+1} = \omega(x_i)$ für eine geeignete Operation ω gilt.

Optimalitätsprinzip: Die optimale Lösung eines Teilproblems (der Größe n) setzt sich aus optimalen Lösungen kleinerer Teilprobleme zusammen.

Wenn der Editierpfad P von x_0 nach x_ℓ

$$x_0, x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_j, x_{j+1}, \ldots, x_\ell$$

optimal ist, dann ist auch der Editierpfad P' von x_i nach x_j optimal, denn sonst könnte P' ersetzt werden durch einen besseren Pfad und damit würde auch P verbessert werden. $\lightning$

Der Bottom-Up-Ansatz startet mit den Werten des Rekursionsendes. Die Tabelle wird zeilenweise, innerhalb einer Zeile von links nach rechts gefüllt. So ist sichergestellt, dass in jedem Schritt die benötigten Teillösungen bereits berechnet wurden.

function EDITDISTANCE(A, B : String)
  for $i := 0, \ldots, n$ **do** $D[i][0] := i$
  for $j := 0, \ldots, m$ **do** $D[0][j] := j$
  for $i := 1, \ldots, n$ **do**
    for $j := 1, \ldots, m$ **do**
      $min := D[i-1][j-1]$
      if $a_i \neq b_j$ **then**
        $min := min + 1$
      if $D[i-1][j] + 1 < min$ **then**
        $min := D[i-1][j] + 1$
      if $D[i][j-1] + 1 < min$ **then**
        $min := D[i][j-1] + 1$
  return $D[n][m]$

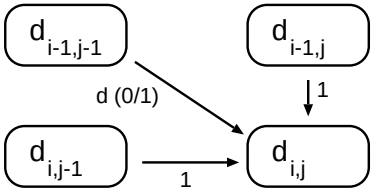
-	ϵ	T	o	r
ϵ	0	1	2	3
T	1			
i	2			
e	3			
r	4			

-	ϵ	T	o	r
ϵ	0	1	2	3
T	1	0	1	2
i	2			
e	3			
r	4			

-	ϵ	T	o	r
ϵ	0	1	2	3
T	1	0	1	2
i	2	1	1	2
e	3			
r	4			

-	ϵ	T	o	r
ϵ	0	1	2	3
T	1	0	1	2
i	2	1	1	2
e	3	2	2	2
r	4			

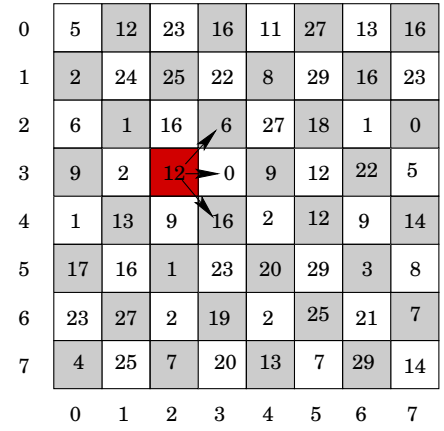
-	ϵ	T	o	r
ϵ	0	1	2	3
T	1	0	1	2
i	2	1	1	2
e	3	2	2	2
r	4	3	3	2



Da das Optimalitätsprinzip von Bellman gilt und alle möglichen Kombinationen von Teilproblemen untersucht werden, erhalten wir auch hier ein optimales Ergebnis.

Betrachten wir folgendes Optimierungsproblem:

- Gegeben: $n \times m$ Spielbrett, eine Spielfigur
- Geldbetrag pro Feld, wird beim Passieren des Feldes gutgeschrieben
- Spiel beginnt im Feld links oben
- Spielfigur darf nur um eine Position nach rechts, nach rechts oben oder nach rechts unten gezogen werden.
- Ist die Spielfigur in der rechten Spalte angekommen, ist das Spiel zu Ende. Danach ist der Gegner an der Reihe.
- Wer das meiste Geld eingesammelt hat, hat gewonnen.



Wie können wir sicherstellen, dass wir das Spiel gewinnen?

Wie sieht die Rekursionsformel aus? Betrachten wir dazu die Situation, in der die Spielfigur auf einem beliebigen Feld auf Position (x, y) steht.

- Sei $w(x, y)$ der optimale Wert, der von Position (x, y) aus erzielt werden kann.
- Sei $val(x, y)$ der Geldbetrag auf Feld (x, y).
- allgemein: Spielfigur kann auf drei Nachbarfelder gezogen werden.

$$w(x, y) = \max\{w(x + 1, y - 1), w(x + 1, y), w(x + 1, y + 1)\} + val(x, y)$$

  Da wir nicht wissen, welcher der Wege den höchsten Gewinn erzielt, müssen alle drei Wege ausprobiert werden.
- Bei einem Feld am oberen Rand ($y = 0$) entfällt der erste Term; am unteren Rand ($y = m$) entfällt der letzte Term innerhalb der Maximum-Berechnung.

Wie können wir diesen rekursiven Ansatz in einem Bottom-Up-Ansatz umwandeln?

Um stets auf bereits berechnete Teillösungen zugreifen zu können, bauen wir die 2-dimensionale Tabelle W so auf, dass zunächst die Werte in der rechten Spalte ($x = n$) initialisiert werden.

Anschließend gehen wir immer eine Spalte weiter nach links und berechnen alle Zeilenwerte.

Übung 9. Schreiben Sie einen Algorithmus zum Lösen dieses Problems und schätzen Sie seine Laufzeit ab. Implementieren Sie den Algorithmus. Im Moodle-Kurs steht ein Rahmenprogramm zur Verfügung. Überlegen Sie sich, wie der Algorithmus erweitert werden kann, sodass auch der optimale Weg bestimmt wird.

Betrachten Sie eine Variante des Spiels: In welchem Feld der ersten Spalte muss man starten, um möglichst viel Geld zu sammeln? Schreiben Sie auch hierfür einen Algorithmus und bestimmen Sie seine Laufzeit.

Übung 10. Gilt das Optimalitätsprinzip von Bellman für das Traveling-Salesperson-Problem?

Übersicht

Gegeben ist eine Folge von Objekten $(a_1, ..., a_n)$, jedes Objekt a_i besitzt einen Schlüssel k_i. Zwischen den Schlüsseln sei eine Ordnungsrelation $\leq$ definiert.

Gesucht ist eine Permutation π sodass $k_{\pi(1)} \leq k_{\pi(2)} \leq k_{\pi(3)} \leq \ldots \leq k_{\pi(n)}$ gilt.

Wir gehen im weiteren davon aus, dass die Objekte z.B. als Struktur (in C) oder als Klassen (in C++) definiert sind

```c
typedef struct {  // in C
    char *name, *vorname;
    long int matrikelnr;
    datum_t geburtsdatum;
} student_t;
```

```cpp
class Buch {  // in C++
    string autor, titel;
    int jahr;
    string isbn;
};
```

und in einem globalen Array a mit den Elementen $a[1], \ldots, a[n]$ zur Verfügung stehen.

In der Praxis werden Daten tatsächlich sehr häufig anhand von Schlüsseln sortiert, aber mittels Callback-Funktionen (in C) oder Funktoren (in C++) können Daten natürlich auch anhand anderer Attribute sortiert werden.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int tag, monat, jahr;
} datum_t;

typedef struct {
    char *name, *vorname;
    long int matrikelnr;
    datum_t gebdatum;
} student_t;

student_t a[] = {
    {"Meier", "Max", 123456789, {2, 3, 1990}},
    {"Schulze", "Clara", 987654321, {1, 2, 1989}},
    {"Papadopoulos", "Demetrios", 456987123, {3, 1, 1989}},
    {"Yilmaz", "Defne", 789456123, {8, 4, 1989}}
};
```

```
int cmpDatum(const void *a, const void *b) {
    datum_t d1 = ((student_t *) a)->gebdatum;
    datum_t d2 = ((student_t *) b)->gebdatum;

    int x = d1.jahr * 10000 + d1.monat * 100 + d1.tag;
    int y = d2.jahr * 10000 + d2.monat * 100 + d2.tag;

    if (x < y)
        return -1;
    if (x > y)
        return +1;
    return 0;
}
```

```
void output(void) {
    for (int i = 0; i < 4; i++)
        printf("%14s %12s %14ld  (%02d,%02d,%4d)\n",
                a[i].name, a[i].vorname, a[i].matrikelnr,
                a[i].gebdatum.tag, a[i].gebdatum.monat,
                a[i].gebdatum.jahr);
}

int main(void) {
    printf("vorher:\n");
    output();

    qsort(a, 4, sizeof(student_t), cmpDatum);

    printf("\nnachher:\n");
    output();
    return 0;
}
```

Ausgabe:

```
vorher:
         Meier          Max      123456789  (02,03,1990)
       Schulze         Clara      987654321  (01,02,1989)
  Papadopoulos    Demetrios      456987123  (03,01,1989)
        Yilmaz         Defne      789456123  (08,04,1989)

nachher:
  Papadopoulos    Demetrios      456987123  (03,01,1989)
       Schulze         Clara      987654321  (01,02,1989)
        Yilmaz         Defne      789456123  (08,04,1989)
         Meier          Max      123456789  (02,03,1990)
```

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
using namespace std;

class Buch {
    friend class BuchVerglJahr;
    friend ostream& operator<<(ostream&, const Buch&);

    string _autor, _titel;
    int _jahr;
    string _isbn;
public:
    Buch(string autor, string titel, int jahr, string isbn)
            : _autor(autor), _titel(titel), _jahr(jahr),
              _isbn(isbn) {}
};
```

```
ostream& operator<<(ostream& os, const Buch& b) {
    os << setw(25) << b._titel << ", "
       << setw(14) << b._autor << ", "
       << b._jahr << ", " << b._isbn;
    return os;
}

Buch a[] = {
    {"Tolkien", "Der Herr der Ringe", 1969, "123-456789-1"},
    {"Orwell", "1984", 1949, "456-123789-2"},
    {"Rowling", "H. Potter: Feuerkelch", 2000, "321-456789-4"},
    {"Saint-Exupery", "Der kleine Prinz", 1974, "987-654321-3"}
};
```

```
class BuchVerglJahr {  // Funktor
public:
    bool operator()(const Buch& a, const Buch& b) {
        if (a._jahr < b._jahr)
            return true;
        return false;
    }
};

void output(void) {
    for (int i = 0; i < 4; i++)
        cout << a[i] << endl;
}
```

```
int main(void) {
    cout << "vorher:" << endl;
    output();

    sort(begin(a), end(a), BuchVerglJahr());

    cout << "\nnachher:" << endl;
    output();
}
```

Ausgabe:

```
vorher:
       Der Herr der Ringe,        Tolkien, 1969, 123-456789-1
                     1984,         Orwell, 1949, 456-123789-2
    H. Potter: Feuerkelch,        Rowling, 2000, 321-456789-4
         Der kleine Prinz,  Saint-Exupery, 1974, 987-654321-3

nachher:
                     1984,         Orwell, 1949, 456-123789-2
       Der Herr der Ringe,        Tolkien, 1969, 123-456789-1
         Der kleine Prinz,  Saint-Exupery, 1974, 987-654321-3
    H. Potter: Feuerkelch,        Rowling, 2000, 321-456789-4
```

Übersicht

Selectionsort (Sortieren durch Auswahl)

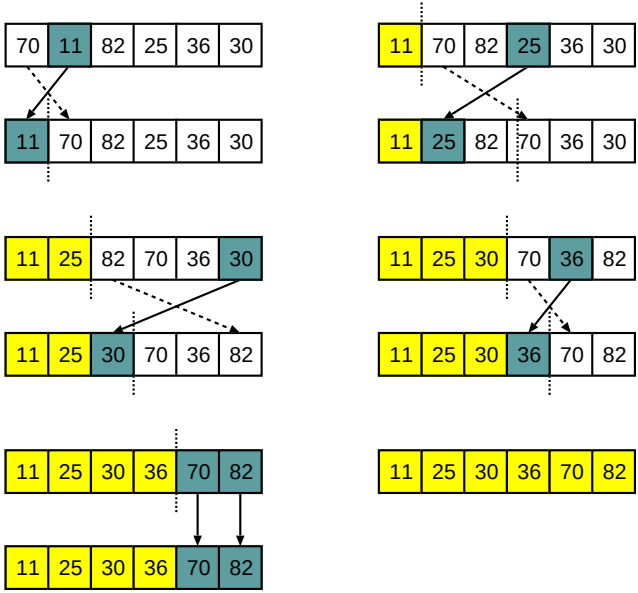
Wir definieren zunächst eine Funktion, um zwei Elemente im Array zu tauschen:

```
function SwapAt(a : sequence; i, j : Integer)
    t := a[i]
    a[i] := a[j]
    a[j] := t
```

Dann definieren wir die Funktion zum Sortieren durch Auswahl:

```
function SelectionSort(a : sequence)
    for i := 1, ..., n − 1 do
        min := i                ▷ bestimme Position des kleinsten Elements der Restfolge
        for j := i + 1, ..., n do
            if a[j].key < a[min].key then
                min := j
        SwapAt(a, i, min)                ▷ swap elements at position i and min
```

Selectionsort



Anzahl Vergleiche (Comparisons):

$$\begin{aligned} C_{min} = C_{max} = C_{avg} &= (N-1) + (N-2) + (N-3) + \ldots + 1 \\ &= \sum_{i=1}^{N-1} i \;=\; \frac{N \cdot (N-1)}{2} \;\in\; \Theta(N^2) \end{aligned}$$

Anzahl Zuweisungen (Moves):

$$M_{min} = M_{max} = M_{avg} = 3 \cdot (N-1)$$

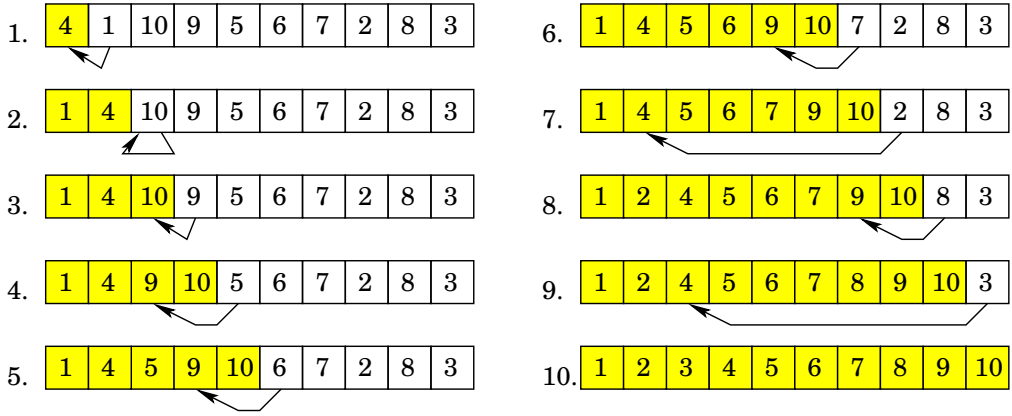
Übersicht

Insertionsort (Sortieren durch Einfügen)

Beim Sortieren durch Einfügen werden Elemente im Array verschoben, um Platz an einer bestimmten Position zu schaffen.

```
function InsertionSort(a : sequence)
    for i := 2, ..., n do
        j := i
        t := a[i]
        while a[j − 1] > t.key do                ▷ verschieben der Elemente
            a[j] := a[j − 1]
            j := j − 1
        a[j] := t
```

Beispiel:



Anzahl Verschiebungen: 23

Anzahl Vergleiche (Comparisons):

$$C_{min} = N - 1 \qquad C_{max} = \sum_{i=2}^{N} i \in \Theta(N^2)$$

Anzahl Zuweisungen (Moves):

$$M_{min} = 2(N-1) \qquad M_{max} = \sum_{i=2}^{N} (i+1) \in \Theta(N^2)$$

Mittlere Laufzeit: Hängt ab von der erwarteten Anzahl von Elementen, die im Anfangsstück $a[1], \ldots, a[i-1]$ in der falschen Reihenfolge bezüglich des i-ten Elements stehen. $\rightarrow$ Anzahl von Inversionen (auch ein Maß für „Vorsortiertheit“)

Im Mittel kann man erwarten, dass die Hälfte der dem i-ten Element k_i vorangehenden Elemente größer als k_i ist.

$$M_{avg} = C_{avg} = \sum_{i=2}^{N} \frac{i}{2} = \frac{1}{2} \sum_{i=2}^{N} i \in \Theta(N^2)$$

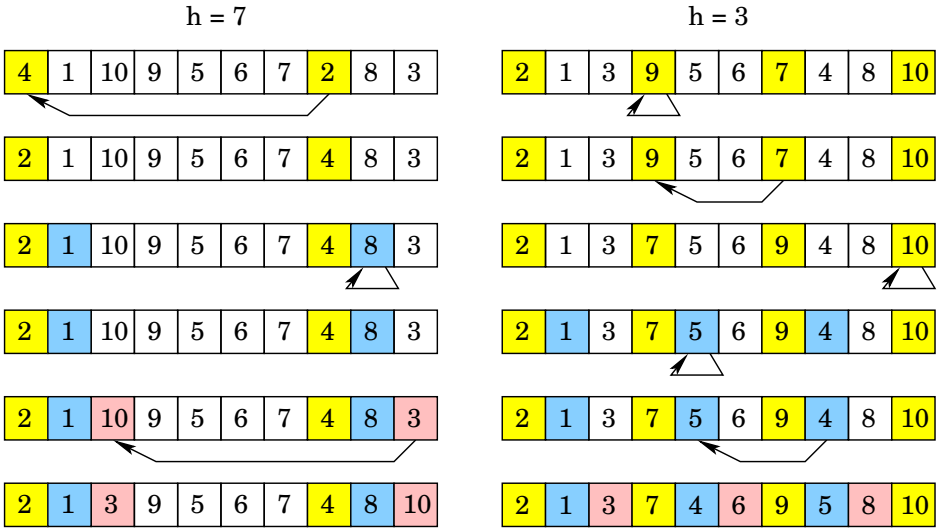
Shellsort

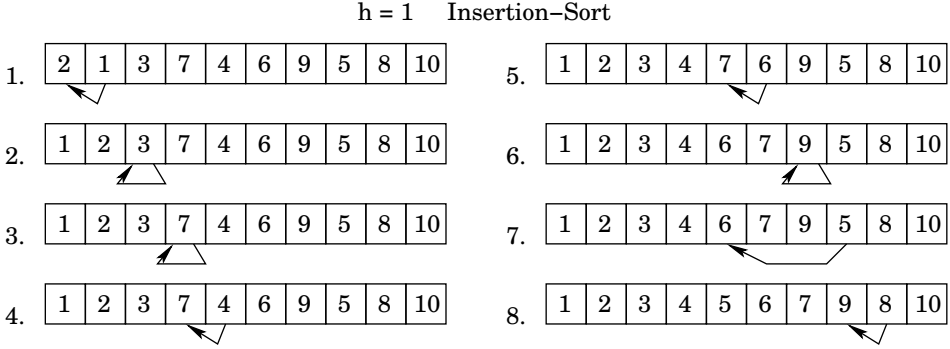
Problem bei Insertionsort: Es dauert ggf. sehr lange, eine Zahl an die richtige Position zu bringen.

Idee von Shellsort: Erlaube zunächst große Sprünge.

- Wähle eine abnehmende und mit 1 endende Folge von Inkrementen $h_t, h_{t-1}, \ldots, h_1$, z.B. die Folge 11,7,3,1.
- Betrachte der Reihe nach für jedes h_i alle Folgen aus Elementen mit Abstand h_i zueinander:
  - Für $j = 1, \ldots, h_i$ sei $F_{i,j}$ die Folge mit den Elementen $j,\ j + h_i,\ j + 2 \cdot h_i,\ j + 3 \cdot h_i, \ldots$
  - Sortiere jede Folge $F_{i,j}$ mittels Insertion-Sort.

Die Korrektheit folgt aus der Sortierung von $F_{1,1}$.


h = 7
4 1 10 9 5 6 7 2 8 3
2 1 10 9 5 6 7 4 8 3
2 1 10 9 5 6 7 4 8 3
2 1 10 9 5 6 7 4 8 3
2 1 10 9 5 6 7 4 8 3
2 1 3 9 5 6 7 4 8 10
h = 3
2 1 3 9 5 6 7 4 8 10
2 1 3 9 5 6 7 4 8 10
2 1 3 7 5 6 9 4 8 10
2 1 3 7 5 6 9 4 8 10
2 1 3 7 5 6 9 4 8 10
2 1 3 7 4 6 9 5 8 10



Anzahl Verschiebungen: 11

Für welche Folgen von Inkrementen benötigt das Verfahren möglichst wenige Vertauschungen?

Es gibt eine Reihe überraschender, jedoch unvollständiger Antworten:

- Laufzeit $\mathcal{O}(n^{1,5})$ für die Folge $1, 3, 7, 15, 31, 63, \ldots, 2^k - 1$.
- Laufzeit $\mathcal{O}(n \cdot \log^2(n))$, für die Folge $1, 2, 3, 4, 6, 8, 9, 12, 16, \ldots, 2^p 3^q$.

Aus Donald E. Knuth. The Art of Computer Programming. Part 3: Sorting and Searching. Addison-Wesley.

Übersicht

Quicksort

Einige Fakten:

- 1962 von C.A.R. Hoare veröffentlicht
- Divide-and-Conquer-Algorithmus
- eines der schnellsten allgemeinen Sortierverfahren
- in-situ: kein zusätzlicher Speicherplatz zur Speicherung von Datensätzen erforderlich (außer einer konstanten Anzahl von Hilfsspeicherplätzen für Tauschoperationen)
- praxistauglich
- implementiert in der C-Lib (stdlib.h)

Algorithmus:

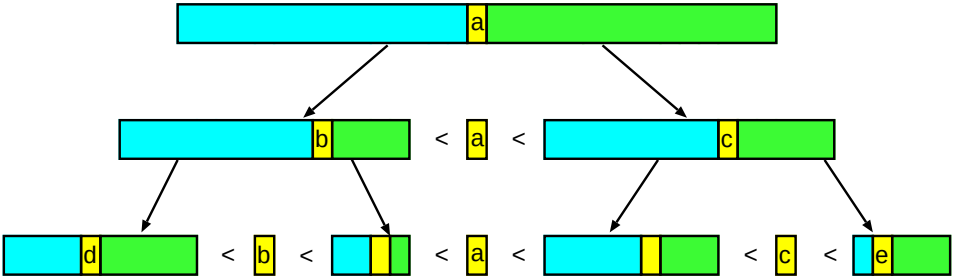
1. *Divide:* Wähle aus allen Werten einen beliebigen Wert p, das Pivotelement aus und teile die Folge in zwei Teilfolgen K und G auf:
   - K enthält Werte die kleiner oder gleich p sind,
   - G enthält Werte die größer oder gleich p sind.

2. *Conquer:* Sortiere K und G rekursiv
3. *Combine:* trivial (entfällt)

Noch offene Punkte:

- aufteilen der Folge in zwei Teilfolgen
- wählen des Pivot-Elements

schematische Darstellung:



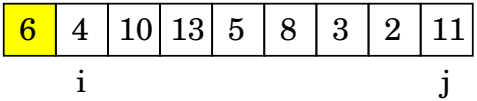
Mögliche Wahl des Pivot-Elements: erstes Element der Teilfolge

Aufteilen der Folge in zwei Teilfolgen:

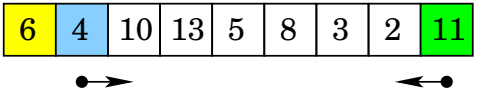
function PARTITION(a : sequence; ℓ, r: Integer)
  $p := a[\ell], \quad i := \ell + 1, \quad j := r$
  repeat
    while $i < r$ and $a[i] \leq p$ **do**
      $i := i + 1$
    while $j > \ell$ and $a[j] \geq p$ **do**
      $j := j - 1$
    if $i < j$ **then**
      SWAPAT(a, i, j)
  until $j \leq i$
  SWAPAT(a, ℓ, j)
  return j

$\rightarrow$ Die Laufzeit ist in $\Theta(n)$ für n Elemente.

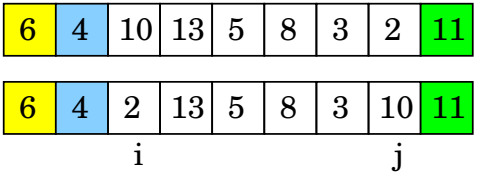
Beispiel:



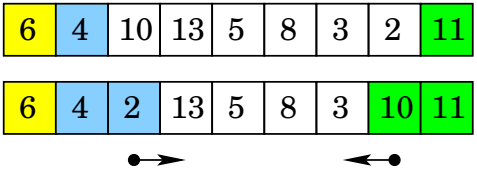
Beispiel:



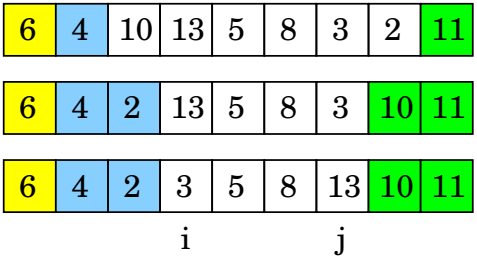
Beispiel:



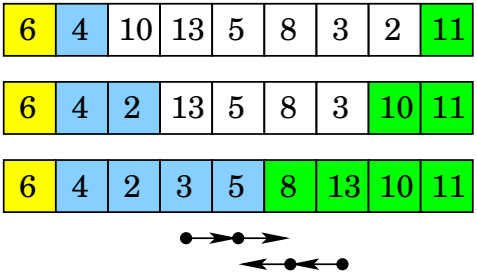
Beispiel:



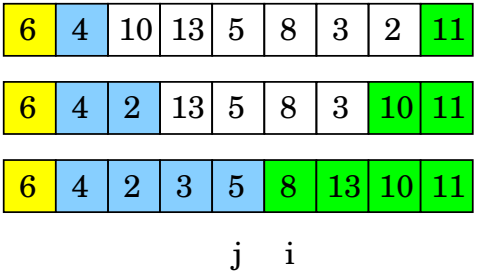
Beispiel:



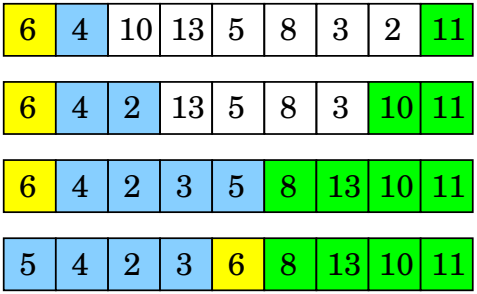
Beispiel:



Beispiel:



Beispiel:



Pseudo-Code:

function QuickSort(a : sequence; ℓ, r: Integer)
  if $\ell < r$ **then**
    $m :=$ Partition(a, ℓ, r)
    QuickSort(a, ℓ, $m-1$)
    QuickSort(a, $m+1$, r)

Initialer Aufruf: QuickSort(1, n)

Worst-Case-Analyse:

Betrachte sortierte Folge. Bei jedem rekursiven Aufruf ist die Teilfolge K leer und Teilfolge G ist um ein Element (dem Pivot-Element) kürzer geworden.

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= T(n-2) + \Theta(n-1) + \Theta(n) \\ &= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\ &\vdots \\ &= T(1) + \Theta(2) + \ldots + \Theta(n-2) + \Theta(n-1) + \Theta(n) \\ \rightarrow T &\in \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2) \end{aligned}$$

Anmerkung: Wenn wir etwas in der Art $T(n) = T(n-1) + \Theta(n)$ schreiben, dann meinen wir eigentlich $T(n) \leq T(n-1) + c \cdot n$ für eine geeignete Konstante c.

Best-Case-Analyse:

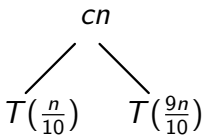
- die Folge wird bei jeder Aufteilung halbiert
- $T(n) = 2 \cdot T(n/2) + \Theta(n)$

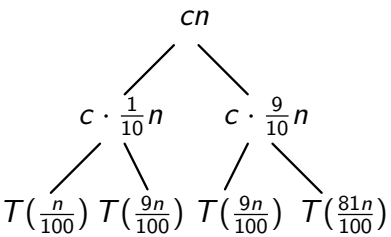
Das Master-Theorem liefert für $a = 2$, $b = 2$ und $k = 1$:

$$T \in \Theta(n^k \cdot \log(n)) = \Theta(n \cdot \log(n))$$

Frage: Welche Laufzeit hat Quicksort, wenn die Aufteilung immer in einem festen Verhältnis, z.B. im Verhältnis $\frac{1}{10} : \frac{9}{10}$ erfolgt?

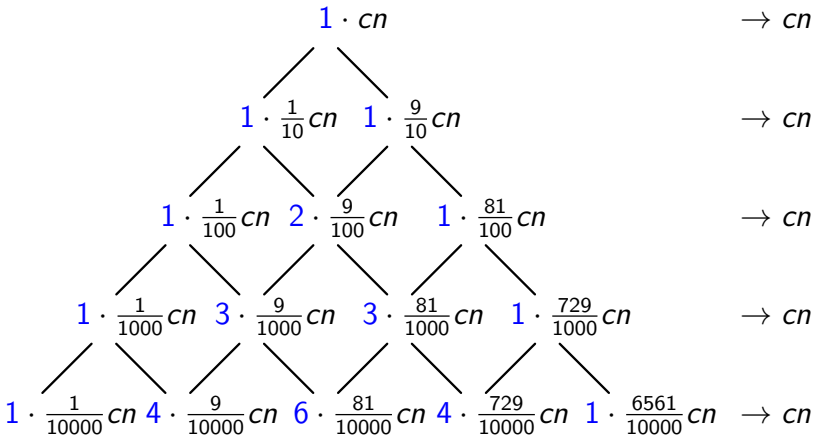
Wenn die Aufteilung im Verhältnis $\frac{1}{10}$ zu $\frac{9}{10}$ erfolgt, erhalten wir folgenden Rekursionsbaum:





$\rightarrow cn$

Wir setzen die Aufteilung fort, wobei wir gleich große Teile zusammen fassen:



Auf jeder vollständigen Ebene k des Rekursionsbaumes ergibt sich eine Laufzeit von cn, denn:

$$\begin{aligned}\sum_{i=0}^{k}\binom{k}{i}\frac{9^i}{10^k}cn &= \frac{cn}{10^k}\sum_{i=0}^{k}\binom{k}{i}9^i \\ &= \frac{cn}{10^k}\sum_{i=0}^{k}\binom{k}{i}9^i\cdot 1^{k-i} \\ &= \frac{cn}{10^k}(9+1)^k = cn\end{aligned}$$

Wir erhalten einen Baum, der rechts wesentlich tiefer ist als links. Das Rekursionsende im rechten Zweig ist erreicht bei:

$$\left(\frac{9}{10}\right)^k \cdot n = 1 \iff n = \left(\frac{10}{9}\right)^k \iff \log_{10/9}(n) = k$$

Der Baum hat also nur eine logarithmische Tiefe.

Die Wahrscheinlichkeit, dass die Folge an Position p aufgeteilt wird, ist $1/n$. Das Aufteilen in zwei Teilfolgen erfolgt in Zeit $\mathcal{O}(n)$.

$$T(n) \;=\; c \cdot n \;+\; \frac{1}{n} \cdot \sum_{p=1}^{n} (T(p-1) + T(n-p))$$

Es gilt:

$$T(0) + \ldots + T(n-1) \;=\; T(n-1) + \ldots + T(0)$$

Damit erhalten wir:

$$T(n) \;=\; c \cdot n \;+\; \frac{2}{n} \cdot \sum_{p=1}^{n} T(p-1)$$

Multiplikation mit n ergibt:

$$n \cdot T(n) \;=\; c \cdot n^2 \;+\; 2 \cdot \sum_{p=1}^{n} T(p-1)$$

Wir hatten auf der letzten Folie festgestellt:

$$n \cdot T(n) \;=\; c \cdot n^2 \;+\; 2 \cdot \sum_{p=1}^{n} T(p-1)$$

Subtraktion der gleichen Formel für $n-1$ ergibt:

$$\begin{aligned} & n \cdot T(n) \;-\; (n-1) \cdot T(n-1) \\ &= \; cn^2 + 2 \cdot \sum_{p=1}^{n} T(p-1) \;-\; c(n-1)^2 - 2 \cdot \sum_{p=1}^{n-1} T(p-1) \end{aligned}$$

Die Terme der beiden Summen heben sich gegenseitig auf, bis auf den Term $2 \cdot T(n-1)$. Außerdem gilt $(n-1)^2 = n^2 - 2n + 1$, daher erhalten wir:

$$\begin{aligned} n \cdot T(n) - (n-1) \cdot T(n-1) &= 2 \cdot T(n-1) + c \cdot (2n-1) \\ n \cdot T(n) &\leq (n+1) \cdot T(n-1) + 2cn \end{aligned}$$

Wir hatten bereits festgestellt:

$$n \cdot T(n) \leq (n+1) \cdot T(n-1) + 2cn$$

Division durch $n \cdot (n+1)$ liefert:

$$\frac{T(n)}{n+1} \leq \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Dies lässt sich fortsetzen:

$$\begin{aligned} \frac{T(n)}{n+1} &\leq \frac{T(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\ &\leq \frac{T(n-3)}{n-2} + \frac{2c}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\ &\leq \frac{T(1)}{2} + \sum_{p=2}^{n} \frac{2c}{p+1} \end{aligned}$$

Wir formen das Ergebnis der letzten Folie ein wenig um:

$$\begin{aligned}\frac{T(n)}{n+1} &\leq \frac{T(1)}{2} + \sum_{p=2}^{n} \frac{2c}{p+1} = \frac{T(1)}{2} + 2c \cdot \sum_{p=3}^{n+1} \frac{1}{p} \\ &= \frac{T(1)}{2} + 2c \cdot \sum_{p=2}^{n} \frac{1}{p} + 2c\frac{1}{n+1} - c\end{aligned}$$

Wir wissen, dass für die harmonische Reihe gilt:

$$\sum_{p=2}^{n} \frac{1}{p} \leq \ln(n)$$

Also erhalten wir:

$$\frac{T(n)}{n+1} \leq \frac{T(1)}{2} + 2c \cdot \ln(n) + 2c\frac{1}{n+1} - c$$

Wir hatten bereits festgestellt:

$$\frac{T(n)}{n+1} \leq \frac{T(1)}{2} + 2c \cdot \ln(n) + 2c\frac{1}{n+1} - c$$

Multiplikation mit $n+1$ liefert:

$$T(n) \leq (n+1) \cdot \frac{T(1)}{2} + 2c(n+1) \cdot \ln(n) + 2c - c(n+1)$$

Lassen wir Konstanten und Terme niedriger Ordnung weg, so erhalten wir schließlich:

$$T(n) \in \Theta(n \cdot \log(n))$$

Fazit:

- Worst-Case: $T \in \Theta(n^2)$
- Average-Case: $T \in \Theta(n \cdot \log(n))$
- Best-Case: $T \in \Theta(n \cdot \log(n))$

Problem: Laufzeit $\mathcal{O}(n^2)$ bei stark vorsortierten Folgen

Lösung:

- *Zufallsstrategie:* Wähle als Pivot-Element ein zufälliges Element aus $a[\ell], \ldots, a[r]$ und vertausche es mit $a[\ell]$.

$\rightarrow$ Laufzeit ist (fast) unabhängig von der zu sortierenden Folge

$\rightarrow$ mittlere bzw. erwartete Laufzeit: $\Theta(n \cdot \log(n))$

Man nennt ein Sortierverfahren *glatt* (smooth), wenn es im Mittel N verschiedene Schlüssel in $\mathcal{O}(N \cdot \log(N))$ und N gleiche Schlüssel in $\mathcal{O}(N)$ Schritten zu sortieren vermag mit einem weichen Übergang zwischen diesen Werten.

3-Wege-Split Quicksort:

Teile die Folge $a[\ell], \ldots, a[r]$ in drei Folgen F_ℓ, F_m, F_r auf.

1. F_ℓ enthält die Elemente mit Schlüssel $< k$.
2. F_m enthält die Elemente mit Schlüssel $= k$.
3. F_r enthält die Elemente mit Schlüssel $> k$.

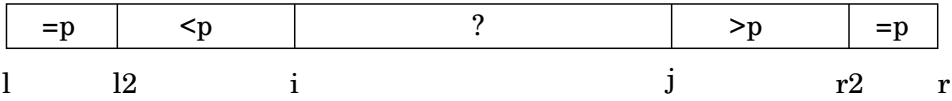
Sortiere F_ℓ und F_r auf dieselbe Weise.

Anmerkungen:

- keine zwei gleichen Schlüssel $\rightarrow$ keine Ersparnis
- alle Schlüssel identisch $\rightarrow$ kein rekursiver Aufruf

Laufzeit: Im Mittel werden $\mathcal{O}(N \cdot \log(n) + N)$ Schritte benötigt, wobei n die Anzahl der verschiedenen Schlüssel unter den N Schlüsseln der Eingabefolge ist.

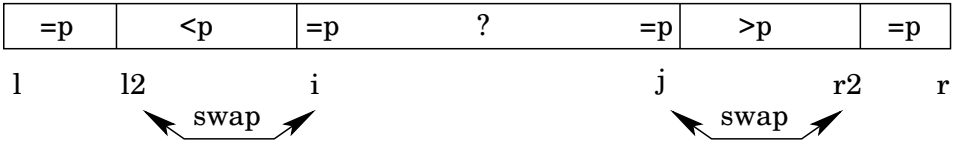
Idee: Die Pivotelemente werden zuerst am Rand zwischen `l` und `l2` sowie zwischen `r2` und `r` gesammelt und vor dem rekursiven Aufruf in die Mitte transportiert.



Initial setzen wir `l2 := l` und `r2 := r`, da noch keine Elemente gefunden wurden, die gleich dem Pivot-Element sind.

Während der Aufteilung, also der Partitionierung, sind vier Fälle zu unterscheiden.

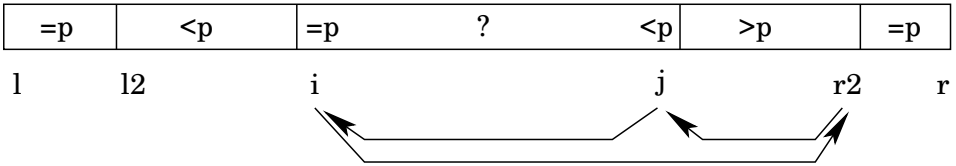
Fall 1: `a[i] == p` und `a[j] == p`



Nach dem Tauschen setze `l2 := l2 + 1` und `r2 := r2 - 1`, da jeweils am linken und rechten Rand Elemente positioniert wurden, die gleich dem Pivot-Element sind.

In allen vier Fällen setzen wir nach dem Tausch der Werte schließlich `i := i + 1` und `j := j - 1`.

Fall 2: `a[i] == p` und `a[j] < p`

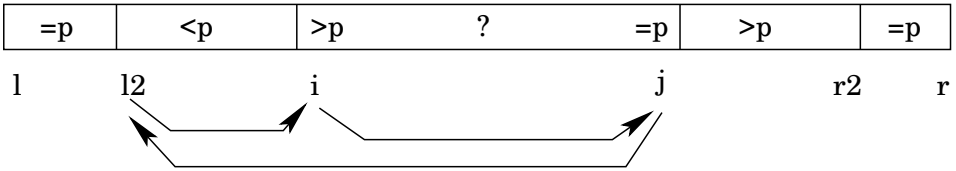



```
r2 := r2 - 1
```

Tausch von drei Elementen:

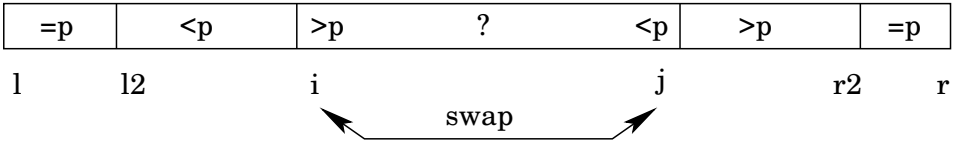
```
void swap3(int i, int j, int r2) {
    tmp = a[i];
    a[i] = a[j];
    a[j] = a[r2];
    a[r2] = tmp;
}
```

Fall 3: `a[i] > p und a[j] == p`



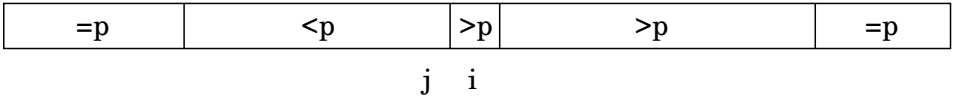

```
l2 := l2 + 1
```

Der vierte Fall wird wie im ursprünglichen Quicksort behandelt:

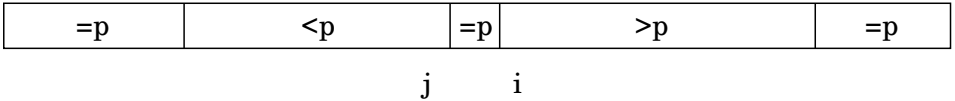


Unterscheide drei Fälle nach der Aufteilung, falls `i = j`:

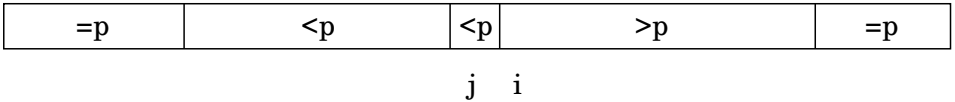
`if a[i] > p then j := j - 1`



`if a[i] = p then j := j - 1; i := i + 1`



`if a[i] < p then i := i + 1`



Anschließend wird vom linken Rand nach `j` und vom rechten Rand nach `i` kopiert

```
for k := l to l2 do
    swapAt(k, j);
    j := j - 1;
```

```
for k := r downto r2 do
    swapAt(k, i);
    i := i + 1;
```

Übung 11.

- Implementieren Sie Quicksort und vergleichen Sie Ihre Implementierung mit der `qsort`-Implementierung aus der Standardbibliothek.

  Vergleichen Sie dazu die Laufzeiten für zufällige Zahlenfolgen der Länge $2^{15}, 2^{16}, 2^{17}, \ldots, 2^{25}$ als Eingabe.
- Ändert sich das Laufzeitverhalten, wenn die Zufallszahlen nur aus dem Bereich von 0 bis 99.999 gewählt werden?
- Implementieren Sie den 3-Wege-Split Quicksort und vergleichen Sie die Laufzeit mit der ursprünglichen Version.

Quicksort mit beschränkter Rekursionstiefe

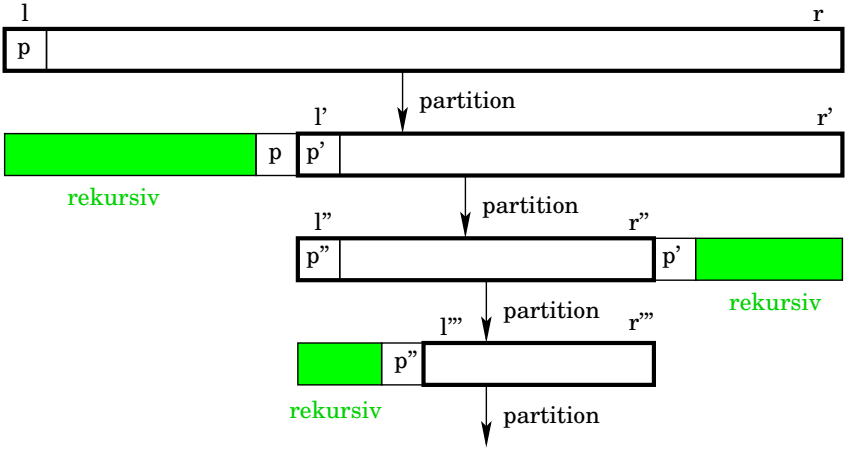
Problem: Im ungünstigen Fall ist die Rekursionstiefe $\mathcal{O}(n)$.

Verringern der Rekursionstiefe auf $\mathcal{O}(\log(n))$: Löse das kleinere Teilproblem rekursiv und löse das größere Teilproblem iterativ direkt.

```
quicksort(int ℓ, int r)
    while ℓ < r
        m := partition(ℓ, r)
        if (m − ℓ) < (r − m) then
            quicksort(ℓ, m − 1)
            ℓ := m + 1
        else
            quicksort(m + 1, r)
            r := m − 1
```

- $(m - \ell)$: Länge der Teilfolge K mit Elementen kleiner gleich p.
- $(r - m)$: Länge der Teilfolge G mit Elementen größer gleich p.

Quicksort mit beschränkter Rekursionstiefe



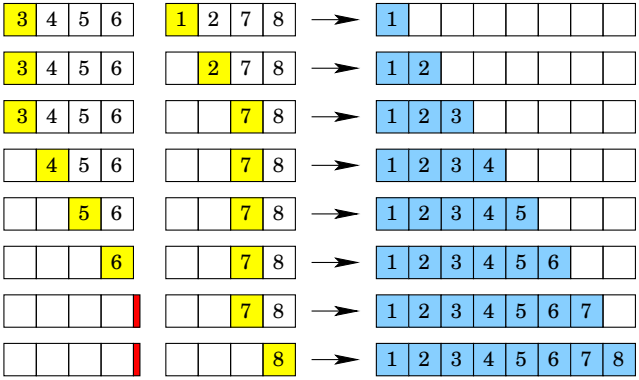
Die hier dargestellten Aufrufe der Funktion *partition* erfolgen iterativ innerhalb der While-Schleife. Der rekursive Aufruf der Funktion *quicksort* erfolgt nur für die kürzere Teilfolge, die höchstens $n/2$ Elemente besitzt.

Übersicht

Mergesort arbeitet auch nach dem Divide-and-Conquer-Prinzip.

1. Divide: Teile die Daten in 2 etwa gleich große Hälften auf.
2. Conquer: Sortiere die beiden Hälften durch rekursive Anwendung von Mergesort.
3. Merge: Mische die sortierten Hälften in eine einzige sortierte Folge.

Merge: Übernehme in die Resultatfolge das jeweils kleinere Element. $\rightarrow \mathcal{O}(n)$



Merge (angelehnt an Volker Heun)

```
void merge(int a[], int b[], int l, int m, int r) {
    int i = l;
    int j = m + 1;
    for (int k = l; k <= r; k++) {
        if ((j > r) or ((i <= m) and (a[i] <= a[j])))
            b[k] = a[i++];
        else b[k] = a[j++];
    }
}
```

Rekursiver Mergesort

Die Folge wird sortiert, indem zunächst die linke und rechte Teilfolge (rekursiv) sortiert wird und anschließend beide Teilfolgen im Merge-Schritt zusammengefasst werden.

```
void mergesort(int a[], int b[], int l, int r) {
    if (l < r) {
        int m = (l+r) / 2;
        mergesort(a, b, l, m);
        mergesort(a, b, m+1, r);
        merge(a, b, l, m, r);
        copy(b, a, l, r);
    }
}
```

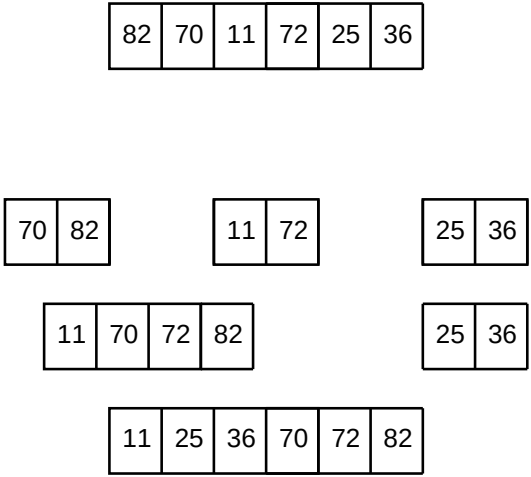
Alternative: Verwende keine *copy*-Funktion für jedes zusammengefügte Teilarray, sondern speichere vor dem Merge-Schritt den rechten Teil in umgekehrter Reihenfolge ab (siehe z.B. Robert Sedgewick).

Iterativer Mergesort: Der rekursive Mergesort ist einfach und verständlich, kostet jedoch Speicher und Zeit. Effizienter ist der iterative Mergesort: Mische von links nach rechts Folgen, deren Längen Zweierpotenzen sind.

```
void mergesort(int a[], int b[], int len) {
    for (int w = 1; w < len; w *= 2) {
        for (int i = 0; i < len; i += 2*w) {
            int l = i;
            int m = min(len, i + w) - 1;
            int r = min(len, i + 2*w) - 1;
            merge(a, b, l, m, r);
        }
        copy(b, a, 0, len);
    }
}
```

Code nach Volker Heun.

Iterativer Mergesort



Laufzeit:

- $\mathcal{O}(\log(n))$ Divide- und Merge-Schritte
- $\mathcal{O}(n)$ Schritte für ein Merge
- oder wie im Best-Case von Quicksort:

$$T(n) = 2 \cdot T(n/2) + \Theta(n) \;\rightarrow\; T \in \Theta(n \cdot \log(n))$$

  Auch hier meinen wir wieder $T(n) \leq 2 \cdot T(n/2) + c \cdot n$ für ein passendes $c \in \mathbb{R}^+$.
- Laufzeit unabhängig von der Sortierung der Eingabedaten.

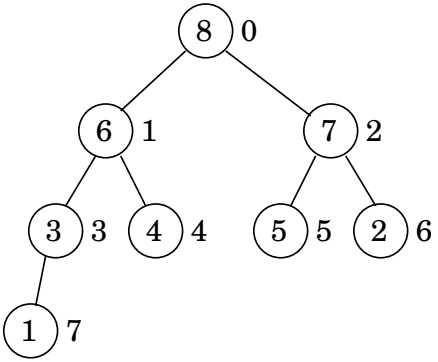
Vor- und Nachteile zu Quicksort:

- Die Laufzeit ist auch im schlechtesten Fall in $\mathcal{O}(n \cdot \log(n))$.
- Merge-Schritt benötigt zusätzlichen Speicher in der Größe der Datenmenge.
- Die Kopieroperationen sind zeitaufwändig.
- Kann zu einem externen Sortierverfahren erweitert werden.

Übersicht

Heap: Eine Folge $F = k_0, \ldots, k_n$ von Schlüsseln, sodass $k_i \geq k_{2i+1}$ und $k_i \geq k_{2(i+1)}$ gilt, sofern $2i + 1 \leq n$ bzw. $2(i + 1) \leq n$. (Hier nutzen wir einen Max-Heap.)

Beispiel: $F = 8, 6, 7, 3, 4, 5, 2, 1$



In diesem Beispiel ist die Heap-Eigenschaft erfüllt:

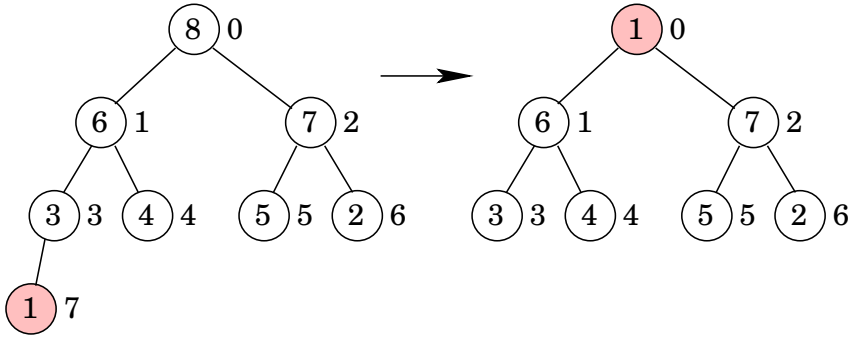
- $k_0 = 8 \geq k_1 = 6$ und $k_0 \geq k_2 = 7$
- $k_1 = 6 \geq k_3 = 3$ und $k_1 \geq k_4 = 4$
- $k_2 = 7 \geq k_5 = 5$ und $k_2 \geq k_6 = 2$
- $k_3 = 3 \geq k_7 = 1$

Achtung: Wir zeichnen den Heap zwar als Baum, aber die Werte sind in einem Array gespeichert!

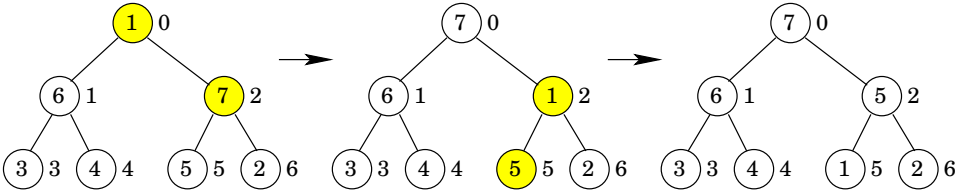
Bestimmung des Maximums ist leicht: k_0 ist das Maximum.

Das nächst kleinere Element wird bestimmt, indem das Maximum aus F entfernt wird und die Restfolge wieder zu einem Heap transformiert wird:

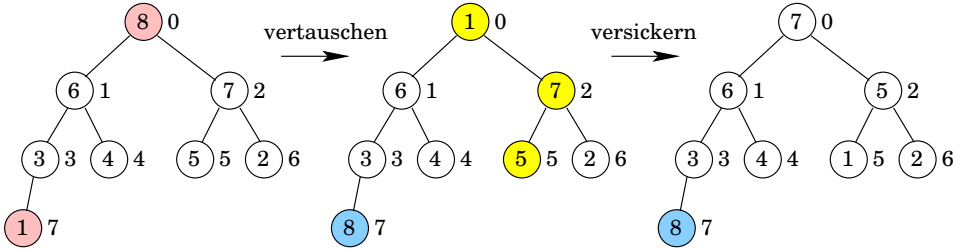
1. Setze den Schlüssel mit dem größten Index an die erste Position.
   $\Rightarrow$ Heap-Eigenschaft verletzt!



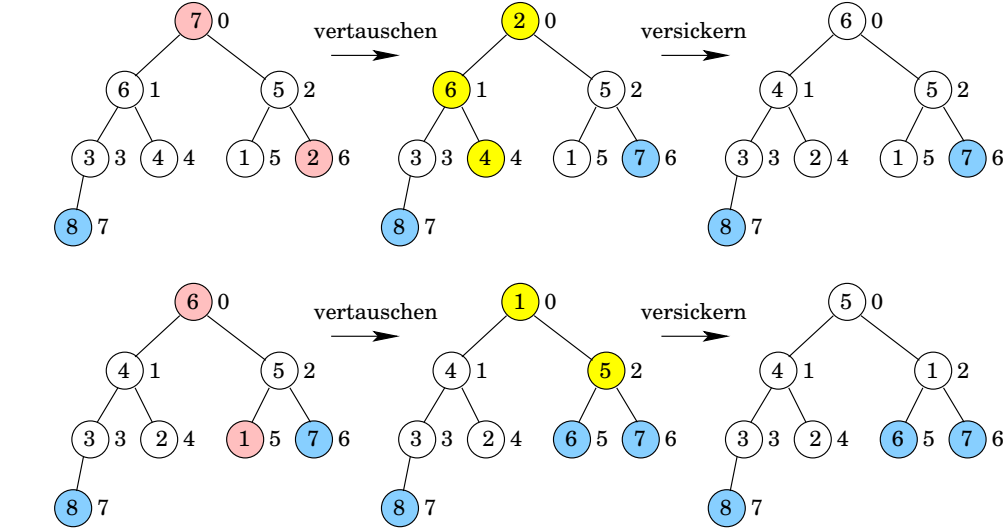
2. Schlüssel versickern lassen, indem er immer mit dem größten seiner Nachfolger getauscht wird, bis entweder beide Nachfolger kleiner sind oder der Schlüssel unten angekommen ist.



Die Datensätze können sortiert werden, indem das jeweils aus dem Heap entfernte Maximum an die Stelle desjenigen Schlüssels geschrieben wird, der nach dem Entfernen des Maximums nach k_0 übertragen wird.



Die blau markierten Knoten werden nicht weiter betrachtet, denn diese Werte stehen bereits an der korrekten Stelle.


7 0
6 1
5 2
3 3
4 4
1 5
2 6
8 7
vertauschen
2 0
6 1
5 2
3 3
4 4
1 5
7 6
8 7
versickern
6 0
4 1
5 2
3 3
2 4
1 5
7 6
8 7
6 0
4 1
5 2
3 3
2 4
1 5
7 6
8 7
vertauschen
1 0
4 1
5 2
3 3
2 4
6 5
7 6
8 7
versickern
5 0
4 1
1 2
3 3
2 4
6 5
7 6
8 7

usw.

Analyse:

- Es erfolgen $n-1$ Vertauschungen außerhalb der Funktion `versickern`.
- Innerhalb von `versickern` wird ein Schlüssel wiederholt mit einem seiner Nachfolger vertauscht, wobei der Datensatz jeweils eine Stufe tiefer wandert.
- Ein Heap mit n Datensätzen hat $\lceil \log_2(n+1) \rceil$ viele Ebenen.

$\Rightarrow$ Wir erhalten $\mathcal{O}(n \cdot \log(n))$ als obere Schranke für die Anzahl der Vertauschungen.

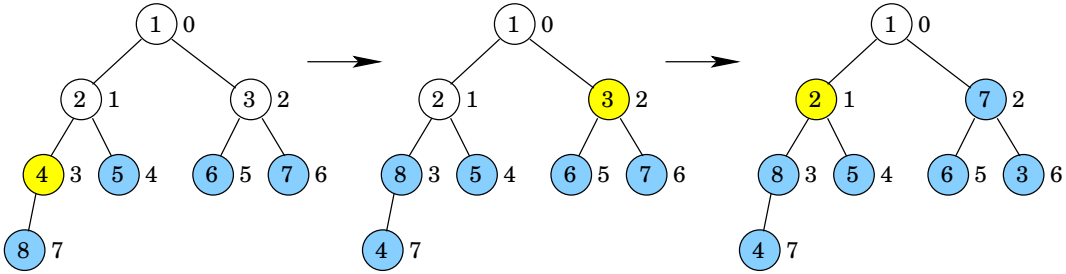
Achtung: Bisher haben wir die Heap-Eigenschaft genutzt, um die Zahlen zu sortieren. Wir sind davon ausgegangen, dass die Zahlen im Array bereits die Heap-Eigenschaft erfüllen.

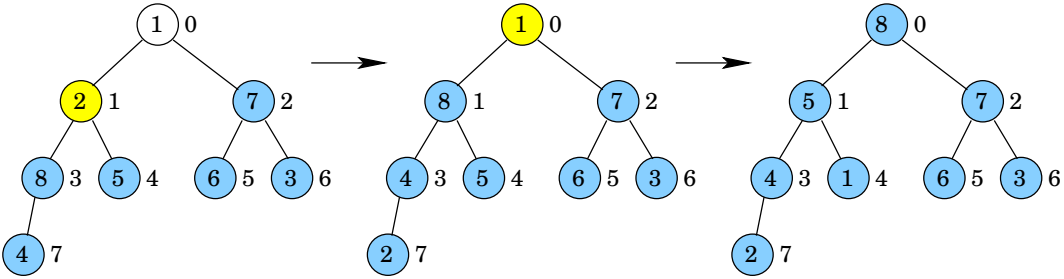
Das ist in der Regel natürlich nicht so. Wir müssen also die Werte im Array zunächst so umordnen, dass die Heap-Eigenschaft erfüllt ist.

Wie wird die Anfangsfolge in einen Heap transformiert?

Idee: Lasse die Schlüssel $k_{n/2-1}, \ldots, k_0$ versickern. Dadurch werden schrittweise immer größere Heaps aufgebaut, bis letztlich ein Heap übrig bleibt.

Beispiel: $F = 1, 2, 3, 4, 5, 6, 7, 8$





Analyse: Aufbauen des initialen Heaps ist in linearer Zeit möglich, weil ...

Betrachten wir einen vollständigen Binärbaum mit $\ell + 1$ Ebenen, also mit $\approx 2^{\ell+1}$ vielen Knoten.

- Auf jeder Ebene i, $0 \leq i \leq \ell$, befinden sich 2^i viele Knoten.
- Wenn ein Element von Ebene i versickert wird, werden höchstens $\ell - i$ viele Vertauschungen durchgeführt. Auf Ebene ℓ werden keine Elemente versickert.
- Die Anzahl Vertauschungen insgesamt beträgt also:

$$\begin{aligned} & 2^0 \cdot \ell + 2^1 \cdot (\ell - 1) + 2^2 \cdot (\ell - 2) + \ldots + 2^{\ell-1} \cdot 1 \\ = \;& 2^{\ell-1} \cdot 1 + 2^{\ell-2} \cdot 2 + 2^{\ell-3} \cdot 3 + \ldots + 2^0 \cdot \ell \\ = \;& 2^\ell \cdot (1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \ldots + \ell/2^\ell) \end{aligned}$$

- Um $1/2^1 + 2/2^2 + 3/2^3 + 4/2^4 + \ldots + \ell/2^\ell$ zu berechnen, sei $s_m := 1/2^1 + 2/2^2 + 3/2^3 + \ldots + m/2^m$

$$
\begin{array}{rcll}
s_1 & = & \frac{1}{2} & = 2 - \frac{3}{2} \\
s_2 & = & s_1 + \frac{2}{2^2} = \frac{1}{2} + \frac{2}{4} = \frac{2}{4} + \frac{2}{4} = \frac{4}{4} & = 2 - \frac{4}{4} \\
s_3 & = & s_2 + \frac{3}{2^3} = \frac{4}{4} + \frac{3}{8} = \frac{8}{8} + \frac{3}{8} = \frac{11}{8} & = 2 - \frac{5}{8} \\
s_4 & = & s_3 + \frac{4}{2^4} = \frac{11}{8} + \frac{4}{16} = \frac{22}{16} + \frac{4}{16} = \frac{26}{16} & = 2 - \frac{6}{16} \\
s_5 & = & s_4 + \frac{5}{2^5} = \frac{26}{16} + \frac{5}{32} = \frac{52}{32} + \frac{5}{32} = \frac{57}{32} & = 2 - \frac{7}{32} \\
s_6 & = & s_5 + \frac{6}{2^6} = \frac{57}{32} + \frac{6}{64} = \frac{114}{64} + \frac{6}{64} = \frac{120}{64} & = 2 - \frac{8}{64}
\end{array}
$$

Vermutung $s_m = 2 - \frac{m+2}{2^m}$ mittels vollständiger Induktion beweisen!

Für die Anzahl der Vertauschungen gilt also:

$$2^\ell \cdot (1/2^1 + 2/2^2 + 3/2^3 + \ldots + \ell/2^\ell) < 2^\ell \cdot 2 = 2^{\ell+1} \in \mathcal{O}(n)$$

Damit ergibt sich insgesamt eine Laufzeit von $\mathcal{O}(n \cdot \log(n))$.

Anmerkungen:

- Eine Vorsortierung der Eingabefolge schadet und nützt der Sortierung nichts.
- Heapsort benötigt nur konstant viel zusätzlichen Speicherplatz, ist also ein in-situ Sortierverfahren.
- Heapsort ist nicht cache-effizient, es treten sehr viele Cache-Misses auf.

Sortieren mittels C-Standardbibliothek

- glibc: modifizierter Quicksort[11]
  - chose pivot by median-of-three decision
  - use insertion sort for small partitions
  - bounded depth of stack
- bionic (android)[12]
  - chose pivot by median-of-three decision
  - use insertion sort for small partitions
  - bounded depth of stack
- musl: smoothsort (variant of heapsort)[13]
- uclibc-ng: Shellsort[14]
- dietlibc: Quicksort with 3-way partitioning[15]

[11] https://sourceware.org/git/?p=glibc.git;a=blob;f=stdlib/qsort.c
[12] https://android.googlesource.com/platform/bionic.git/+/master/libc/upstream-freebsd/lib
[13] http://git.musl-libc.org/cgit/musl/tree/src/stdlib/qsort.c
[14] https://cgit.uclibc-ng.org/cgi/cgit/uclibc-ng.git/tree/libc/stdlib/stdlib.c
[15] https://github.com/ensc/dietlibc/blob/master/lib/qsort.c

Übersicht

Allgemeine Sortierverfahren: Algorithmen, die nur Vergleichsoperationen zwischen Schlüsseln verwenden.

Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel mindestens $\Omega(n \cdot \log(n))$ Schlüsselvergleiche.

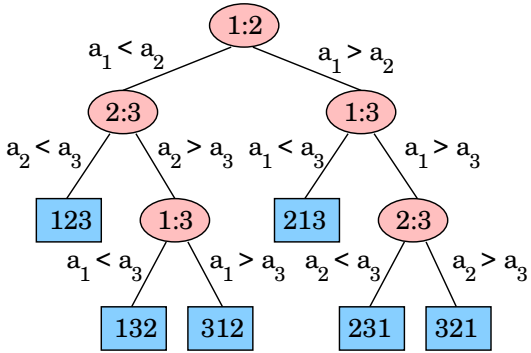
Dobosiewicz[16] hat erstmals ein Sortierverfahren vorgestellt, das arithmetische Operationen und die Floor-Funktion benutzt, und das im Mittel eine Laufzeit von $\mathcal{O}(n)$ hat. Darauf beruht z.B. der Bucketsort-Algorithmus.

Wir werden im Folgenden eine Informationstheoretische untere Schranke zeigen: Die Anzahl der notwendigen Ja/Nein-Entscheidungen, um unterschiedliche Permutationen zu unterscheiden.

Wir betrachten dabei nur die Entscheidungen, keinerlei Verschiebungen oder Vertauschungen von Elementen. Daher ist die Anzahl der Entscheidungen eine untere Schranke für die Laufzeit der Algorithmen.

[16] W. Dobosiewicz: Sorting by distributive partitioning. Information Processing Letters, 7(1), 1978

Entscheidungsbaum:



- Innere Knoten sind mit $i : j$ beschriftet, wobei $i, j \in \{1, 2, \ldots, n\}$ gilt.
- Linker Teilbaum enthält alle nachfolgenden Vergleiche, falls $a_i < a_j$.
- Rechter Teilbaum enthält alle nachfolgenden Vergleiche, falls $a_i > a_j$.

- Jedes Blatt stellt eine Permutation $(\pi(1), \pi(2), \ldots, \pi(n))$ dar und bezeichnet die Sortierung $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.
- hier bedeutet z.B. 132: $a_1 \leq a_3 \leq a_2$

Ein Entscheidungsbaum kann jedes allgemeine, also vergleichsbasierte Sortierverfahren modellieren:

- Es gibt jeweils einen Baum für jede Eingabegröße n.
- Jedes Blatt enthält eine Permutation[17] $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, und bezeichnet eine mögliche Reihenfolge $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Laufzeit des Algorithmus = Länge des gewählten Pfades.

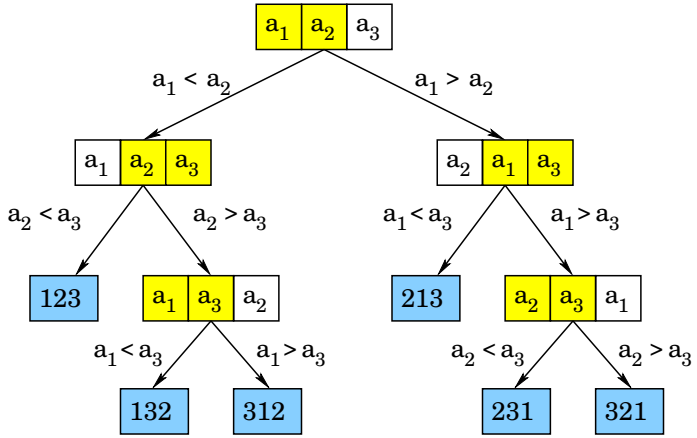
- Worst-Case: längster Weg zu einem Blatt, also Tiefe des Baums
- Average-Case: mittlere Tiefe des Baums
- Best-Case: kürzester Weg zu einem Blatt des Baums

Die mittlere Tiefe eines Baums T mit n Blättern:

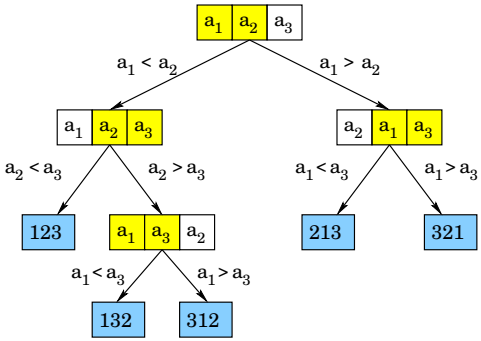
$$\mathrm{mT}(T) = \frac{1}{n} \cdot \sum_{i=1}^{n} \text{Tiefe von Blatt } i$$

[17] Umordnung der Elemente, also eine bijektive Abbildung von $\{1, \ldots, n\}$ nach $\{1, \ldots, n\}$

Im zuvor dargestellten Entscheidungsbaum ist der Insertionsort-Algorithmus für $n = 3$ dargestellt:



Jede Permutation muss in einem der Blätter vorkommen. Machen wir uns das an einem Beispiel klar. Nehmen wir an, dass die Permutation $\pi = (2, 3, 1)$ nicht im Baum vorhanden wäre.



Dann fehlt natürlich auch die entsprechende Verzweigung im Baum, also ein Vergleich. Sehen wir uns an, wie bspw. die Eingabe $(a_1 = 8, a_2 = 3, a_3 = 5)$ verarbeitet wird:

- $a_1 > a_2$, also wird von der Wurzel aus nach rechts verzweigt.
- $a_1 > a_3$, also wird wieder nach rechts verzweigt.

$\rightarrow$ Permutation $(3, 2, 1)$

Die Eingabe wird also nicht sortiert.

Worst-Case-Analyse:

- es gibt $n!$ verschiedene Permutationen über n Zahlen
- der Entscheidungsbaum hat mindestens $n!$ Blätter
- ein Binärbaum der Tiefe t (Anzahl Ebenen - 1) hat maximal 2^t Blätter

$\Rightarrow 2^t \geq n!$

gesucht wird eine untere Schranke für t

$$
\begin{array}{rcll}
t & \geq & \log_2(n!) & \text{log ist monoton steigend} \\
 & \geq & \log_2((n/e)^n) & \text{Stirling-Formel} \\
 & = & \log_2(n^n) - \log_2(e^n) & \\
 & = & n \cdot \log_2(n) - n \cdot \log_2(e) \ \in \ \Omega(n \cdot \log(n)) &
\end{array}
$$

Stirling-Formel: $n! \sim \frac{n^n}{e^n} \cdot \sqrt{2\pi n}$. Dabei bezeichnet $\sim$ eine Äquivalenzrelation[18], also eine reflexive, symmetrische und transitive Relation.

[18] Seien $f, g : \mathbb{N} \to \mathbb{R}$ zwei reelwertige Funktionen. Dann gilt $f \sim g$ genau dann wenn der relative Fehler gegen null strebt, also $\lim_{n\to\infty} {}^{f(n)}/_{g(n)} = 1$ gilt.

Wir können die untere Schranke auch ohne die Stirling-Formel herleiten:

$$\begin{aligned} n! &= 1 \cdot 2 \cdot 3 \cdot \ldots \cdot \frac{n}{2} \cdot \left(\frac{n}{2}+1\right) \cdot \left(\frac{n}{2}+2\right) \cdot \ldots \cdot n \\ &\geq \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \ldots \cdot \frac{n}{2}}_{n/2\text{-mal}} = \left(\frac{n}{2}\right)^{n/2} \end{aligned}$$

Nun nutzen wir den gleichen Ansatz wie oben. Aus $2^t \geq n!$ folgt:

$$\begin{aligned} t &\geq \log_2(n!) && \text{log ist monoton steigend} \\ &\geq \log_2((n/2)^{n/2}) && \text{siehe oben} \\ &= n/2 \cdot \log_2(n/2) \\ &= n/2 \cdot (\log_2(n) - 1) \in \Omega(n \cdot \log(n)) \end{aligned}$$

Average-Case-Analyse: Beweis durch Widerspruch!

- *Behauptung:* Die mittlere Tiefe eines Entscheidungsbaums mit k Blättern ist wenigstens $\log_2(k)$.
- *Annahme:* Es existiert ein Entscheidungsbaum mit k Blättern dessen mittlere Tiefe kleiner als $\log_2(k)$ ist.

Sei T ein solcher Baum, der unter all diesen Bäumen der Baum mit den wenigsten Blättern ist, für den die Annahme gilt. T habe k Blätter. Dann gilt:

- T hat einen linken Teilbaum T_1 mit k_1 Blättern
- T hat einen rechten Teilbaum T_2 mit k_2 Blättern
- es gilt $k_1 < k$, $k_2 < k$ und $k_1 + k_2 = k$

Da T der kleinste Baum ist, für den die Annahme gilt, und T_1 und T_2 kleiner sind als T, muss für die Teilbäume gelten:

$$\mathrm{mT}(T_1) := \text{mittlere Tiefe}(T_1) \geq \log_2(k_1)$$
$$\mathrm{mT}(T_2) := \text{mittlere Tiefe}(T_2) \geq \log_2(k_2)$$

Jedes Blatt in T_1 bzw. T_2 auf Tiefe t hat in T die Tiefe $t+1$. Also gilt insgesamt:

$$\begin{aligned}
\mathrm{mT}(T) &= \frac{1}{k}\left[k_1(\mathrm{mT}(T_1)+1) + k_2(\mathrm{mT}(T_2)+1)\right] \\
&\geq \frac{1}{k}\left[k_1(\log_2(k_1)+1) + k_2(\log_2(k_2)+1)\right] \\
&= \frac{1}{k}[k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2)]
\end{aligned}$$

Unter der Nebenbedingung $k_1 + k_2 = k$ hat die Funktion

$$\mathrm{mT}(T) \;=\; \frac{1}{k}(k_1 \cdot \log_2(2k_1) + k_2 \cdot \log_2(2k_2))$$

ein Minimum bei $k_1 = k_2 = k/2$. Damit gilt

$$\mathrm{mT}(T) \;\geq\; \frac{1}{k}\left(\frac{k}{2}\log_2(k) + \frac{k}{2}\log_2(k)\right) \;=\; \log_2(k)$$

im Widerspruch zur Annahme.

Jeder Entscheidungsbaum zur Sortierung von n Zahlen hat $n!$ Blätter, daher gilt:

$$\mathrm{mT}(T) \;\geq\; \log_2(n!) \;\geq\; \log_2((n/e)^n) \;\in\; \Omega(n\log(n))$$

Übersicht

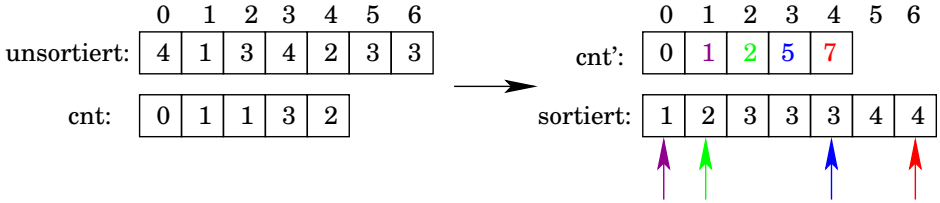
Sortierverfahren ohne Schlüsselvergleiche!

Eingabe: $a[0, 1, ..., n-1]$ mit $a[i] \in \{0, \ldots, k-1\}$

Ausgabe: $b[0, 1, ..., n-1]$ sortiert

Hilfsspeicher: $cnt[0, 1, ..., k-1]$ zum Zählen

Idee: Zähle die Häufigkeiten der Zahlen $0, \ldots, k-1$ in der zu sortierenden Liste und berechne daraus die Position der jeweiligen Zahl in der sortierten Liste.



Pseudo-Code:

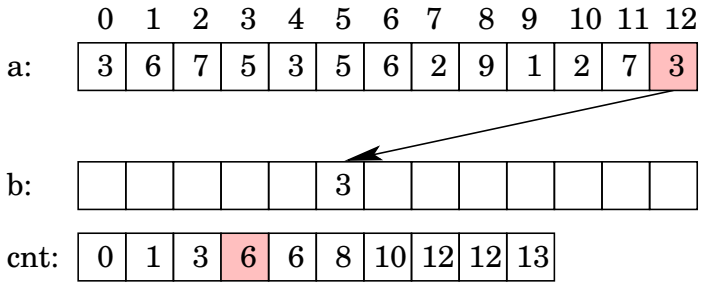
```
for i := 0 to k-1 do                                    /* init */
    cnt[i] := 0

for j := 0 to n-1 do                                    /* count */
    cnt[a[j]] := cnt[a[j]] + 1

for i := 1 to k-1 do                                    /* collect */
    cnt[i] := cnt[i] + cnt[i-1]

for j := n-1 downto 0 do                                /* rearrange */
    b[cnt[a[j]] - 1] := a[j]
    cnt[a[j]] := cnt[a[j]] - 1
```

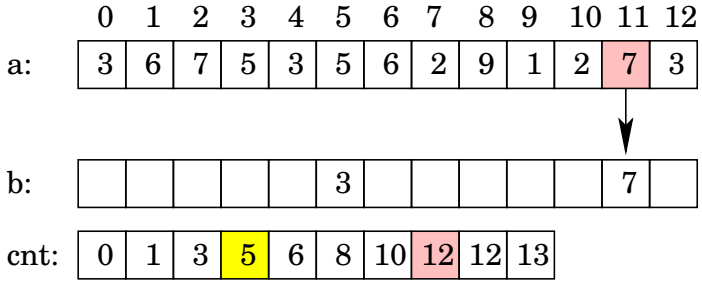
	0	1	2	3	4	5	6	7	8	9	10	11	12
a:	3	6	7	5	3	5	6	2	9	1	2	7	3

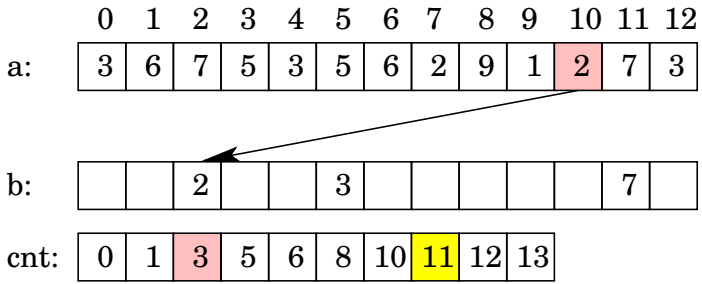
cnt:	0	1	2	3	0	2	2	2	0	1

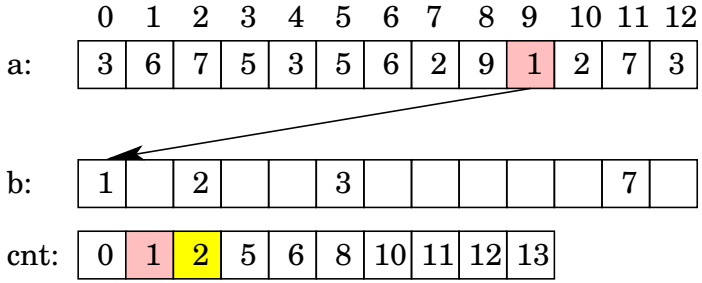
cnt:	0	1	3	6	6	8	10	12	12	13

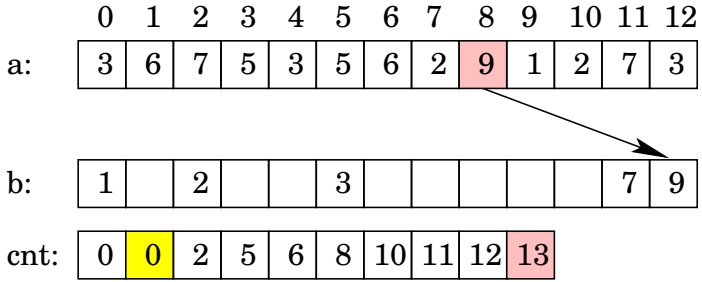
Countingsort



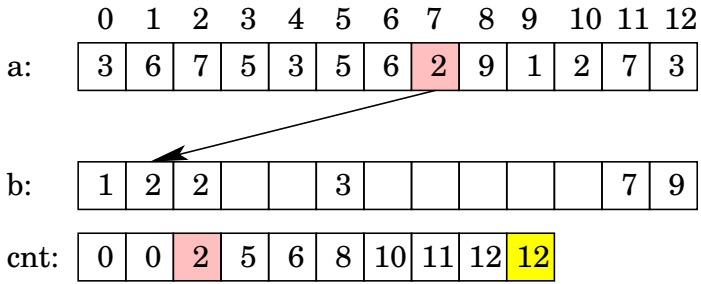

0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 3 7
cnt: 0 1 3 5 6 8 10 12 12 13

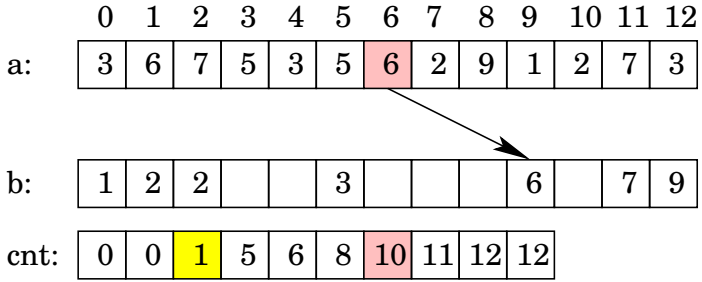

0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 2 3 7
cnt: 0 1 3 5 6 8 10 11 12 13


0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 1 2 3 7
cnt: 0 1 2 5 6 8 10 11 12 13

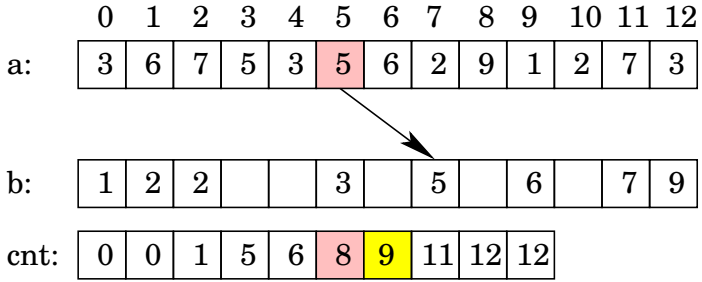

0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 1 2 3 7 9
cnt: 0 0 2 5 6 8 10 11 12 13

Countingsort

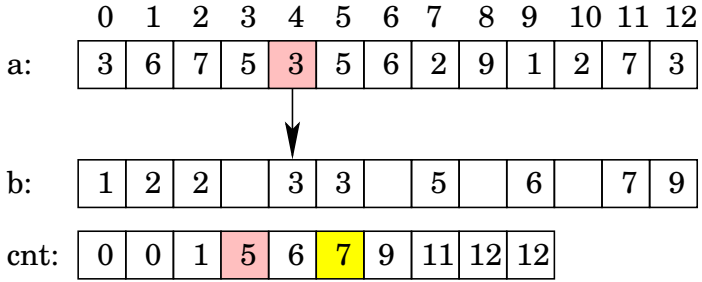


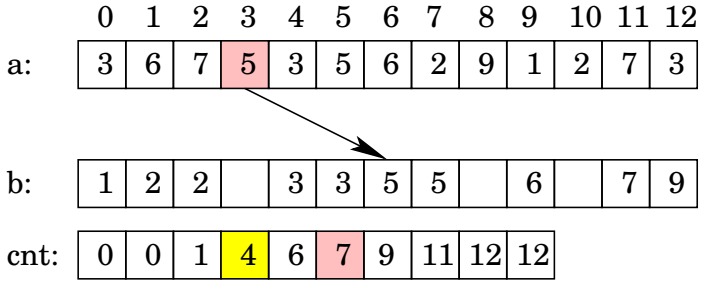

0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 1 2 2 3 6 7 9
cnt: 0 0 1 5 6 8 10 11 12 12

Countingsort

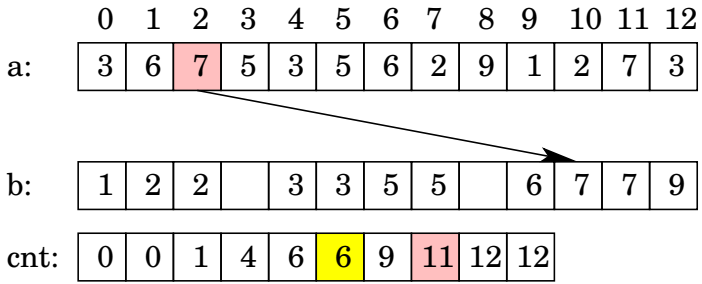


Countingsort

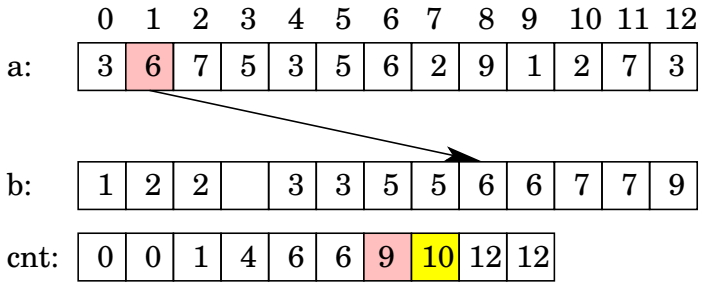



0 1 2 3 4 5 6 7 8 9 10 11 12
a: 3 6 7 5 3 5 6 2 9 1 2 7 3
b: 1 2 2 3 3 5 5 6 7 9
cnt: 0 0 1 4 6 7 9 11 12 12

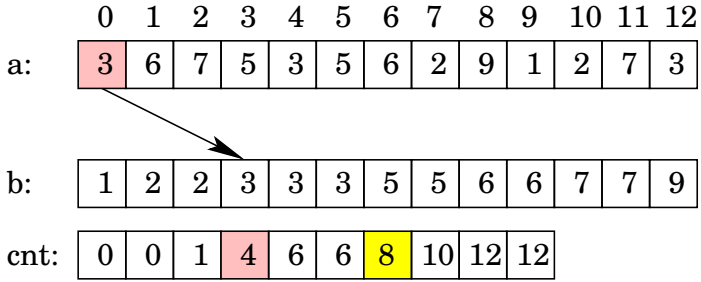
Countingsort



Countingsort



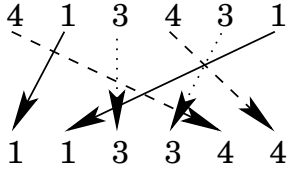
Countingsort



Laufzeit:

init	$\Theta(k)$
count	$\Theta(n)$
collect	$\Theta(k)$
rearrange	$\Theta(n)$
	$\Theta(n+k)$

Countingsort ist ein *stabiles Sortierverfahren*: Gleiche Elemente stehen nach dem Sortieren in der gleichen Reihenfolge wie vor dem Sortieren.



Übersicht

Problem: Countingsort ist ineffizient bzgl. Laufzeit und Speicherplatz, wenn der Wertebereich groß ist im Vergleich zur Anzahl der Zahlen: $\Theta(n + k) = \Theta(n^2)$ für $k = n^2$

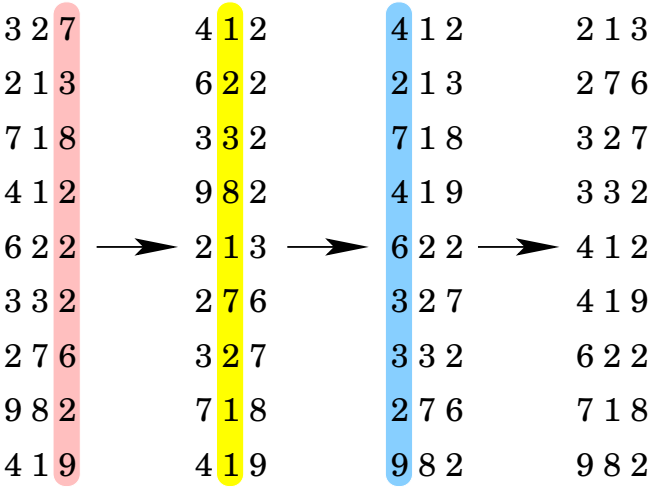
Beispiel: Es sollen 100.000 Datensätze mit einem 32-Bit-Schlüssel sortiert werden, also $k = 2^{32} = 4.294.967.296$

- $n + k \approx 4.300.000.000$
- $n \cdot \log(n) = 100.000 \cdot \log_2(100.000) \approx 1.660.000$

Lösung: Sortiere die Zahlen anhand der einzelnen Ziffern, beginnend mit der niederwertigsten Stelle. Verwende ein stabiles Sortierverfahren wie Countingsort. Falls nötig, müssen führende Nullen eingefügt werden.

Anmerkung: Dieser Ansatz ist für ganze Zahlen geeignet, aber nicht für reelle Zahlen, da es überabzählbar viele reelle Zahlen gibt. Daher sind Counting- und Radixsort nicht für den Einsatz in allgemeinen Bibliotheken geeignet.

Beispiel:


3 2 7
2 1 3
7 1 8
4 1 2
6 2 2
3 3 2
2 7 6
9 8 2
4 1 9
4 1 2
6 2 2
2 1 3
2 7 6
3 2 7
7 1 8
4 1 9
4 1 2
2 1 3
7 1 8
4 1 9
6 2 2
3 2 7
3 3 2
2 7 6
9 8 2
2 1 3
2 7 6
3 2 7
3 3 2
4 1 2
4 1 9
6 2 2
7 1 8
9 8 2

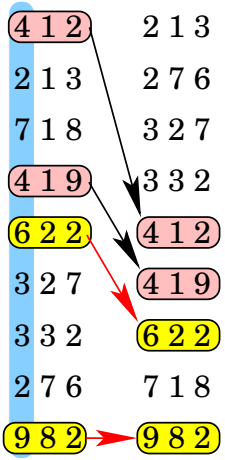
Korrektheit: Induktion über die betrachtete Position.

I.A. Nach der ersten Sortierphase sind die Zahlen bezüglich der Position 0 sortiert.

I.V. Die Zahlen sind bezüglich der $t-1$ niederwertigsten Ziffern sortiert.

I.S. Sortiere nach Ziffer t:

- Zwei Zahlen, die sich an Position t unterscheiden, sind richtig sortiert.
- Zwei Zahlen, die an Position t dieselbe Ziffer haben, sind nach I.V. richtig sortiert und bleiben aufgrund des stabilen Sortierverfahrens sortiert.



Laufzeit:

- fasse jeweils r Bits zu einer Ziffer zusammen
- bei einer Wortlänge von b Bits müssen b/r Phasen durchlaufen werden

Frage: Wie groß muss r im Verhältnis zu b und n sein, um eine gute Laufzeit zu erzielen?

- Countingsort hat Laufzeit $\Theta(n+k)$. Hier: $\Theta(n+2^r)$
- Bei b/r Phasen erhalten wir: $\Theta(b/r \cdot (n+2^r))$
- Extremwert bestimmen: Bei welchem Wert von r wird die Laufzeit minimal?

Praxis: 32-Bit-Wörter in Gruppen von 8 Bit $\Rightarrow$ 4 Phasen

Übersicht

Das Verfahren wird auch *Bin-Sort* oder *Sortieren durch Fachverteilung* genannt.

Wie bei Countingsort treffen wir Annahmen über die zu sortierenden Folgen.

- Countingsort: ganze Zahlen aus kleinem Wertebereich
- Bucketsort: Zahlen aus $[0, 1)$ sind gleichverteilt

Die Werte des Arrays werden in verschiedene Fächer einsortiert.

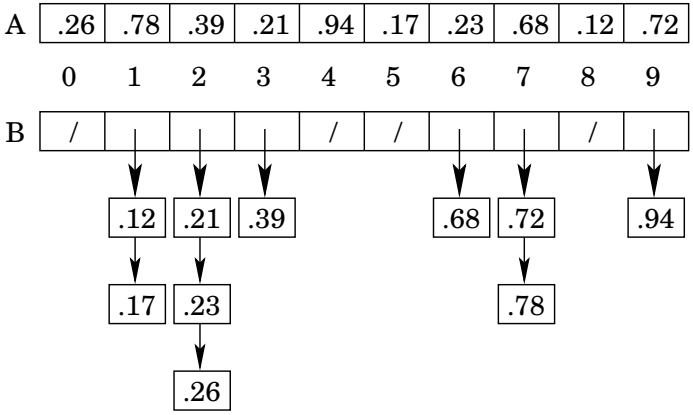
function BucketSort(A)
  $n := length(A)$
  for $i := 1$ to n **do**
    insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
  put values back to A in order of lists $B[0], \ldots, B[n-1]$

Das Einfügen eines Elements in eine sortierte Liste der Länge n kostet $\mathcal{O}(n)$ Schritte.

Anmerkung: Auch Bucketsort ist nicht für den Einsatz in allgemeinen Bibliotheken geeignet, da nicht beliebige Schlüssel sortiert werden können.

[19] gilt bei Ottmann/Widmayer als eine Radixsort-Variante

Beispiel:



Laufzeit:

- worst-case: alle Werte müssen in ein Fach einsortiert werden $\rightarrow \mathcal{O}(n^2)$
- best-case: genau ein Wert pro Fach $\rightarrow \mathcal{O}(n)$

Average-Case-Laufzeit: $\mathcal{O}(n)$

Sei X_i eine Zufallsvariable, die die Anzahl der Elemente im Fach $B[i]$ beschreibt. Für jedes Element $A[j]$ gilt:

$$Pr(A[j] \text{ fällt in Fach } i) = 1/n$$

Dies gilt unabhängig für alle j, also liegt ein Bernoulli-Experiment vor:

> Bei einem solchen Experiment gibt es stets nur zwei Ausgänge, Treffer oder Niete. Zudem muss die Wahrscheinlichkeit p für einen Treffer, und somit auch die für eine Niete $1 - p$, bei jedem der Experimente dieselbe sein. (wikipedia)

$B(k \mid p, n)$ bezeichnet die Wahrscheinlichkeit, genau k Erfolge zu erzielen.

$$B(k \mid p, n) = \begin{cases} \binom{n}{k} p^k (1-p)^{n-k} & \text{falls } k \in \{0, 1, \ldots, n\} \\ 0 & \text{sonst} \end{cases}$$

Somit ist X_i binomialverteilt mit den Parametern n und $p = 1/n$. Also gilt:

$$E[X_i] = n \cdot p = 1 \qquad \text{und} \qquad Var[X_i] = n \cdot p \cdot (1-p) = 1 - 1/n$$

Außerdem wissen wir:

$$Var[X_i] = E[X_i^2] - E^2[X_i] \iff E[X_i^2] = Var[X_i] + E^2[X_i]$$

Die erwartete Zeit zum Einfügen der X_i vielen Elemente in Fach i dauert $E[\mathcal{O}(X_i^2)] = \mathcal{O}(E[X_i^2])$ Zeit. (denn: $E[a \cdot X_i] = a \cdot E[X_i]$)

Die erwartete Zeit zum Einsortieren aller Zahlen in die Fächer beträgt also:

$$\begin{aligned}\sum_{i=0}^{n-1} \mathcal{O}(E[X_i^2]) = \mathcal{O}\left(\sum_{i=0}^{n-1} E[X_i^2]\right) &= \mathcal{O}\left(\sum_{i=0}^{n-1} Var[X_i] + E^2[X_i]\right) \\ &= \mathcal{O}\left(\sum_{i=0}^{n-1}(2 - 1/n)\right) = \mathcal{O}(2n-1)\end{aligned}$$

Übung 12. Wie kann die Worst-Case-Laufzeit von Bucketsort auf $\mathcal{O}(n \cdot \log(n))$ verbessert werden, ohne die Average-Case-Laufzeit zu verschlechtern?

Übersicht

Lineare Suche und binäre Suche

In einem früheren Kapitel haben wir ausführlich die

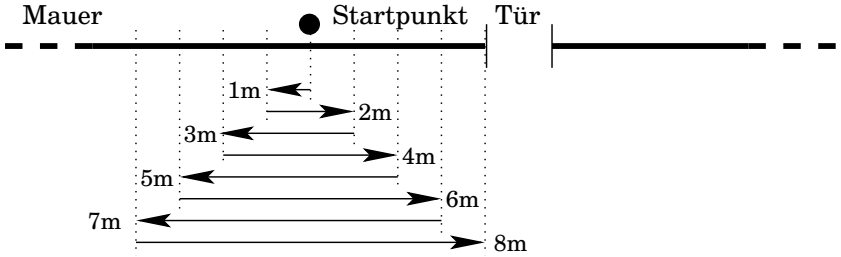
- lineare (Elemente nacheinander lesen) und die
- binäre (rekursive, immer mit dem mittleren Element vergleichen)

Suche besprochen und die Komplexitäten im besten, mittleren und schlechtesten Fall berechnet. Es ergaben sich folgende Laufzeiten:

	best-case	average-case	worst-case
lineare Suche	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
binäre Suche	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$

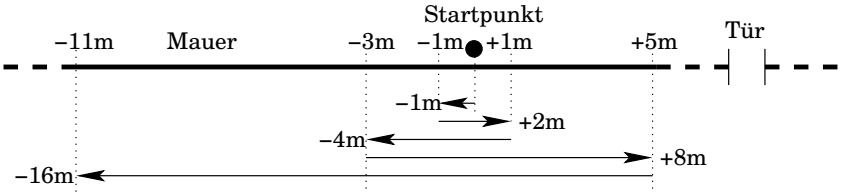
Übersicht

Annahme: Sie stehen vor einer unendlich langen Mauer, von der Sie wissen, dass links oder rechts eine Tür existiert, durch die Sie auf die andere Seite der Mauer gelangen können. Leider wissen Sie nicht, ob sich die Tür links oder rechts von Ihnen befindet. Wie finden Sie in endlicher Zeit diese Tür? → immer wieder Richtungswechsel machen



Wenn die Tür ℓ Meter entfernt ist und Sie zunächst in die falsche Richtung gehen, dann legen Sie insgesamt eine Strecke von $\sum_{i=1}^{2\ell} i = {2\ell \cdot (2\ell+1)}/{2} \approx 2 \cdot \ell^2$ Metern zurück.

Gibt es eine effizientere Methode?



Gehe nach jedem Richtungswechsel die doppelte Strecke als vorher. Bei insgesamt n Richtungswechseln legen wir also folgende Stecke zurück:

$$\text{geometrische Summe:} \quad \sum_{i=0}^{n} 2^i = \frac{2^{n+1}-1}{2-1} \approx 2^{n+1}$$

Es werden die folgenden Wendepunkte erreicht:

$$0 \xrightarrow{+2^0} -1 \xrightarrow{+2^1} 1 \xrightarrow{+2^2} -3 \xrightarrow{+2^3} 5 \xrightarrow{+2^4} -11 \xrightarrow{+2^5} 21 \xrightarrow{+2^6} -43 \xrightarrow{+2^7} 85 \xrightarrow{+2^8} -171$$

$$\xrightarrow{+2^9} 341 \xrightarrow{+2^{10}} -683 \xrightarrow{+2^{11}} 1365 \xrightarrow{+2^{12}} -2731 \xrightarrow{+2^{13}} 5461 \xrightarrow{+2^{14}} -10923 \longrightarrow \ldots$$

Wenn die Tür also ℓ Meter entfernt ist und wir zunächst in die falsche Richtung (im obigen Bild also nach links) gehen, legen wir höchstens $12 \cdot \ell$ Meter zurück.

Richtungswechsel	zurückgelegte Strecke	Wendepunkt
0	1	-1
1	$1 + 2 = 3$	$0 \ldots 1$
2	$3 + 4 = 7$	-3
3	$7 + 5 \ldots 8 = 12 \ldots 15$	$2 \ldots 5$
4	$15 + 16 = 31$	-11
5	$31 + 17 \ldots 32 = 48 \ldots 63$	$6 \ldots 21$
6	$63 + 64 = 127$	-43
7	$127 + 65 \ldots 128 = 192 \ldots 255$	$22 \ldots 85$
8	$255 + 256 = 511$	-171
9	$511 + 257 \ldots 512 = 768 \ldots 1023$	$86 \ldots 341$
10	$1023 + 1024 = 2047$	-683
11	$2047 + 1025 \ldots 2048 = 3072 \ldots 4095$	$342 \ldots 1365$

Exponentielle Suche

Gesucht wird ein Element mit Schlüssel k. Die exponentielle Suche eignet sich zum Suchen in nach den Schlüsselwerten sortierten Feldern, deren Länge nicht bekannt ist bzw. wenn $k \lll n$ ist.

Bestimme in exponentiell wachsenden Schritten einen Bereich, in dem der Schlüssel liegen muss (erster Index ist 1):

```
int i = 1;
while (k > a[i].key)
    i = 2*i;
```

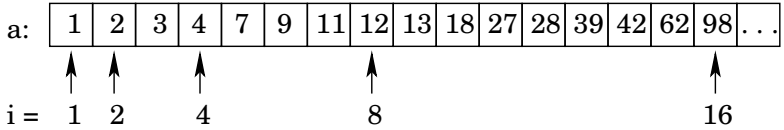
Danach gilt: `a[i/2].key` $<$ `k` $\leq$ `a[i].key`
Dieser Bereich wird mittels binärer Suche durchsucht.

Exponentielle Suche: Beispiel

Wir suchen den Schlüssel $k = 42$ im folgenden Feld:

a:	1	2	3	4	7	9	11	12	13	18	27	28	39	42	62	98	...

(1) bestimme den Bereich, in dem k liegen muss:



(2) binäre Suche im Bereich `a[i/2+1] ... a[i]`:

m = (9 + 16) / 2 = 12

...	13	18	27	28	39	42	62	98	...

m = (13 + 16) / 2 = 14

...	13	18	27	28	39	42	62	98	...

Laufzeit:

- *Voraussetzung:* Werte sind paarweise verschieden. (Dies ist z.B. dann gegeben, wenn Schlüsselwerte gesucht werden: Telefonnummer, Matrikelnummer, Personalnummer, usw.)

$\rightarrow$ Schlüssel wachsen mindestens so stark wie die Indices.

$\rightarrow$ Die Bereichsbestimmung erfolgt in $\log_2(k)$ Schritten.

$\rightarrow$ Der zu durchsuchende Bereich hat höchstens k Zahlen und kann in Zeit $\log_2(k)$ mittels binärer Suche durchsucht werden.

Insgesamt ergibt sich also eine Laufzeit von $\Theta(\log(k))$ im Gegensatz zur binären Suche, die eine Laufzeit von $\mathcal{O}(\log(n))$ hätte.

Falls die Folge auch gleiche Werte enthalten darf, ist keine Laufzeitabschätzung möglich. Das Verfahren funktioniert aber natürlich auch dann.

Übersicht

Idee: Schätze die Position des Elements mit Schlüssel k.

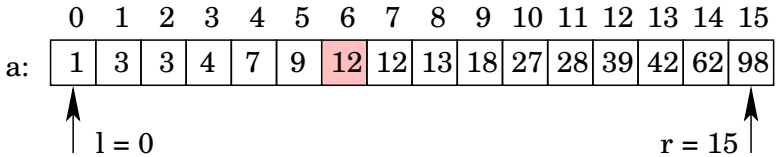
Beispiel: Im Telefonbuch stehen Namen wie `Yilmaz` oder `Zimmermann` weit hinten, wohingegen `Alexopoulos` oder `Brockmann` eher am Anfang zu finden sind.

Sei ℓ die linke Grenze, r die rechte Grenze des Suchbereichs:

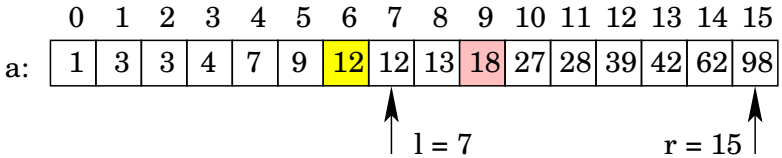
- Bei der binären Suche wurde der Index des nächsten zu inspizierenden Elements bestimmt als
$$m = \left\lfloor \frac{\ell + r}{2} \right\rfloor = \ell + \left\lfloor \frac{1}{2} \cdot (r - \ell) \right\rfloor .$$
- Bei der Interpolationssuche inspiziere als nächstes das Element auf Position
$$m = \ell + \left\lfloor \frac{k - a[\ell].key}{a[r].key - a[\ell].key} \cdot (r - \ell) \right\rfloor .$$

Beispiel: In der folgenden Folge suchen wir den Schlüssel $k = 42$.

Interpolations-Suche

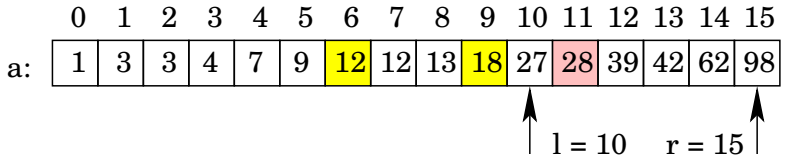


$$m := 0 + \left\lfloor \frac{42-1}{98-1} \cdot (15-0) \right\rfloor = 6$$



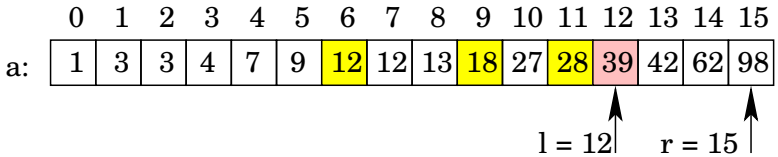
$$m := 7 + \left\lfloor \frac{42 - 12}{98 - 12} \cdot (15 - 7) \right\rfloor = 9$$

Interpolations-Suche

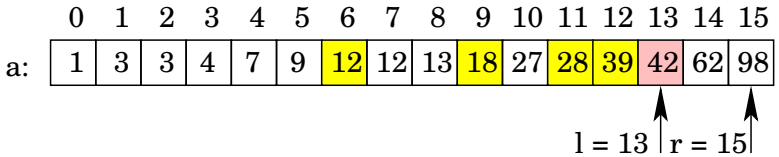


$$m := 10 + \left\lfloor \frac{42 - 27}{98 - 27} \cdot (15 - 10) \right\rfloor = 11$$

Interpolations-Suche



$$m := 12 + \left\lfloor \frac{42 - 39}{98 - 39} \cdot (15 - 12) \right\rfloor = 12$$



$$m := 13 + \left\lfloor \frac{42 - 42}{98 - 42} \cdot (15 - 13) \right\rfloor = 13$$

Laufzeit: Im Worst-Case hat das Verfahren eine lineare Laufzeit, es benötigt also $\Theta(n)$ viele Vergleiche bei n Schlüsselwerten.

Beispiel: $k = 10$ und $F = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1000$

Bei gleichverteilten Daten im Intervall $[1, 1000]$ müsste der Wert 10 relativ weit vorne liegen. Da die Daten in der Folge F aber nicht gleichmäßig verteilt sind, liegt der Wert 10 sehr weit hinten in der Folge F und wird daher erst nach vielen Schritten gefunden.

Laufzeit: Im Mittel[20] werden $\mathcal{O}(\log(\log(n)))$ Schlüsselvergleiche ausgeführt, im Gegensatz zu $\mathcal{O}(\log(n))$ Vergleichen bei der binären Suche.

n	2^{2^1}	2^{2^2}	2^{2^3}	2^{2^4}	2^{2^5}	2^{2^6}
$\log_2(n)$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$
$\log_2(\log_2(n))$	1	2	3	4	5	6

In der Praxis geht dieser Vorteil aber oft durch die auszuführenden arithmetischen Operationen verloren. (Betrachte bspw. Laufzeit bei $2^{2^5} = 4.294.967.296$ Elementen.)

Übung 13. Implementieren Sie die Interpolationssuche und messen Sie die Laufzeiten für verschiedene Eingabegrößen und verschiedene Verteilungen der Zahlen.

Informieren Sie sich über die Anzahl der Taktzyklen, die verschiedene Operationen der CPU wie Addition, Multiplikation, Division oder der Zugriff auf L1- oder L2-Cache benötigen. Wie lässt sich der Unterschied zwischen Theorie und Praxis erklären?

[20] A.C. Yao and F.F. Yao: The complexity of searching an ordered random table. In: Proceedings of the Symposium on Foundations of Computer Science, 1976

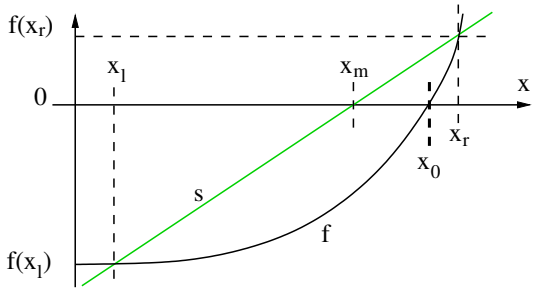
Laufzeit: Im Best-Case wird jedes Element sofort gefunden, es benötigt also $\Theta(1)$ viele Vergleiche bei n Schlüsselwerten.

Beispiel: Betrachten wir die Folge $F = 10, 12, 14, 16, 18, 20, 22, 24, 26$ mit den neun gleichverteilten Elementen $f_0, \ldots, f_8$:

$$m \;=\; 0 + \frac{k - a[0].key}{a[8].key - a[0].key} \cdot (8 - 0) \;=\; \frac{k - 10}{26 - 10} \cdot 8 \;=\; \frac{k - 10}{2}$$

Jedes Element $f_i \in F$ wird direkt gefunden.

Die Interpolationssuche ist bereits aus der Mathematik bekannt, sie wird dort Sekantennäherungsverfahren oder Regula Falsi genannt und wird genutzt, um Nullstellen von Polynomen zu berechnen.



Zunächst benötigen wir zwei Werte x_ℓ und x_r, für die $f(x_\ell) \cdot f(x_r) < 0$ ist, also der eine Funktionswert unter, der andere über der x-Achse liegt. Wenn die Funktion $f(x)$ stetig ist, dann muss zwischen x_ℓ und x_r mindestens eine Nullstelle liegen. Wir wollen uns dieser Nullstelle annähern, indem wir die Nullstelle der Sekante s berechnen.

Die Steigung der Sekante s ergibt sich als

$$\frac{dy}{dx} = \frac{y_r - y_\ell}{x_r - x_\ell}.$$

Die Geradengleichung für s erhalten wir aus der Zweipunktegleichung:

$$s(x) = y_\ell + \frac{y_r - y_\ell}{x_r - x_\ell} \cdot (x - x_\ell)$$

Wir setzen $s(x) = 0$, da wir ja eine Nullstelle suchen, und lösen nach x auf und nennen den resultierenden Wert x_m. Dann erhalten wir

$$\begin{aligned} y_\ell + \frac{y_r - y_\ell}{x_r - x_\ell} \cdot (x_m - x_\ell) = 0 \quad &\iff \quad \frac{y_r - y_\ell}{x_r - x_\ell} \cdot (x_m - x_\ell) = -y_\ell \\ &\iff \quad (x_m - x_\ell) = -y_\ell \cdot \frac{x_r - x_\ell}{y_r - y_\ell} \\ &\iff \quad x_m = x_\ell - y_\ell \cdot \frac{x_r - x_\ell}{y_r - y_\ell} \end{aligned}$$

Nun wollen wir dieses Verfahren auf die Suche in einem sortierten Array anwenden. Wir nehmen hier zur Vereinfachung an, dass keine Werte mehrfach vorhanden sind.

Nehmen wir an, dass ein Wert k gesucht wird, und dieser Wert k an Position j im Array a enthalten ist, also $a[j] = k$ gilt.

- Aus den obigen x-Werten werden Positionen im Array, also Indizes. (Dazu müssen wir die berechneten x-Werte natürlich auf ganze Zahlen abbilden, z.B. abrunden.)
- Aus den y-Werten werden die Inhalte des Arrays an der entsprechenden Position.
- Wenn wir von allen Werten im Array den Wert k subtrahieren, dann gilt
  - $a[i] - k < 0$ für alle Positionen $0 \leq i < j$, und es gilt
  - $a[i] - k > 0$ für alle Positionen $j < i < n$,

  weil das Array sortiert ist.

→ Damit ist die Voraussetzung erfüllt, dass ein Wert unter und ein Wert über der x-Achse liegen muss.

Für das Sekantennäherungsverfahren hatten wir festgestellt:

$$x_m = x_\ell - y_\ell \cdot \frac{x_r - x_\ell}{y_r - y_\ell}$$

Für unseren Algorithmus heißt das mit obigen Bemerkungen:

$$\begin{aligned} m &= \left\lfloor \ell - (a[\ell] - k) \cdot \frac{r - \ell}{(a[r] - k) - (a[\ell] - k)} \right\rfloor = \left\lfloor \ell - \frac{a[\ell] - k}{a[r] - a[\ell]} \cdot (r - \ell) \right\rfloor \\ &= \left\lfloor \ell + \frac{k - a[\ell]}{a[r] - a[\ell]} \cdot (r - \ell) \right\rfloor = \ell + \left\lfloor \frac{k - a[\ell]}{a[r] - a[\ell]} \cdot (r - \ell) \right\rfloor \end{aligned}$$

Genauso ist es im obigen Algorithmus der Interpolationssuche implementiert.

Natürlich müssen wir dieses Verfahren iterativ immer wieder anwenden, um uns der Nullstelle zu nähern. Wenn $a[m]$ größer als der gesuchte Wert ist, müssen wir den rechten Rand r für die nächste Iteration auf $m - 1$ setzen, da dann die Nullstelle links von m liegt. Andernfalls setzen wir den linken Rand ℓ auf $m + 1$.

Übersicht

Einfache Selektionsaufgabe:

- Bestimme das Minimum einer Zahlenfolge.
- Bestimme das Maximum einer Zahlenfolge.

Unterscheide:

- Unsortierte Werte $\Rightarrow$ Alle Werte müssen mindestens einmal betrachtet und verglichen werden.
- Sortierte Werte $\Rightarrow$ Bei wahlfreiem Zugriff kann in einem einzigen Schritt das Minimum bzw. Maximum bestimmt werden.

Hier: Finde zu einer gegebenen Zahlenfolge das i-kleinste Element. Wir gehen davon aus, dass die Zahlen in einem Array `A` gespeichert sind.

speziell: Suche das mittlere Element, das auch Median[21] bzw. Zentralwert genannt wird.

[21] `https://de.wikipedia.org/wiki/Median`

Laut wikipedia gilt: Der Median teilt eine Liste von Werten in zwei Teile. Er kann auf folgende Weise bestimmt werden:

- Alle Werte werden (aufsteigend) geordnet.
- Wenn die Anzahl der Werte ungerade ist, ist die mittlere Zahl der Median.
- Wenn die Anzahl der Werte gerade ist, wird der Median meist als arithmetisches Mittel der beiden mittleren Zahlen definiert, die dann Unter- und Obermedian heißen.

Abweichend von der Mathematik bezeichnen wir mit Median den Untermedian. Das liegt einfach daran, dass wir ein Element der Folge finden wollen. Bei einer sortierten Folge $(x_1, x_2, \ldots, x_n)$ ist der Wert

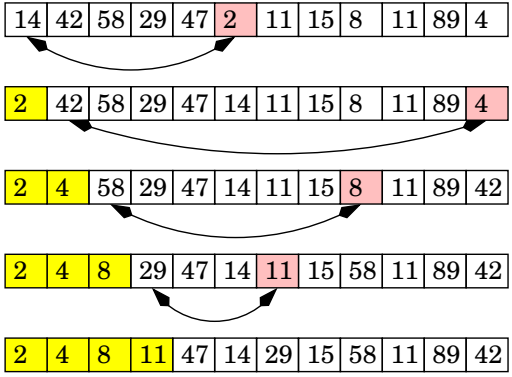
$$\tilde{x}_u = x_{\lfloor \frac{n+1}{2} \rfloor}$$

der Untermedian.

Versuch 1: *Iterative Selektion*

- Sortiere die Werte im Array `A` mittels Selectionsort.
- Brich den Sortiervorgang ab, wenn i Elemente sortiert sind.

Beispiel: Ermitteln des viertkleinsten Elements einer Folge:



Laufzeit: $\mathcal{O}(i \cdot n)$ $\longrightarrow$ nur für kleine Werte von i geeignet

Versuch 2: *Sortieren und Selektieren*

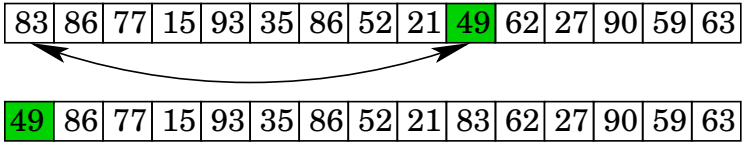
Zunächst werden die Elemente des Arrays `A` mit einem effizienten Algorithmus, wie z.B. Mergesort oder Heapsort sortiert. Anschließend steht das gesuchte Element auf Position i des Arrays.

Laufzeit: $\mathcal{O}(n \cdot \log(n))$

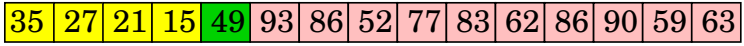
Es geht aber noch schneller, sogar ohne die Werte zu sortieren.

`randomSelect(A, l, r, i)` basiert auf der Idee von Quicksort:

- Wähle aus allen Werten ein Pivot-Element zufällig aus und tausche es mit dem Element am linken Rand.



- `k := partition(A, l, r)`: Teile die Folge in zwei Teilfolgen L und R auf.



- $i < k \Rightarrow$ Gib `randomSelect(A, l, k-1, i)` zurück.
- $i = k \Rightarrow$ Gib das Pivot-Element `A[k]` zurück.
- $i > k \Rightarrow$ Gib `randomSelect(A, k+1, r, i)` zurück.

Die Partitionierung erfolgt in Zeit $\Theta(n)$.

Laufzeit:

- Im günstigen Fall wird die Folge bei jedem rekursiven Aufruf mindestens um einen konstanten Faktor kleiner. Bei der folgenden Abschätzung werden bspw. immer $\frac{1}{10}$ der Zahlen ausgeschlossen. Dann erhalten wir:

$$T(n) = T(9/10 \cdot n) + \Theta(n) \in \Theta(n)$$

- Im ungünstigen Fall wird die rekursiv zu durchsuchende Folge immer nur um ein Element kleiner. Dann erhalten wir:

$$T(n) = T(n-1) + \Theta(n) \in \Theta(n^2)$$

Average-case Laufzeit: Alle Werte im Array der Länge n seien verschieden.

- Für jedes $j \in \{1, \ldots, n\}$ gilt: Das Pivot-Element wird mit Wahrscheinlichkeit $1/n$ das j-kleinste Element des Arrays.
- Nach der Partitionierung enthält die Teilfolge L dann $j-1$ Elemente, R enthält $n-j$ Elemente.

$$\begin{aligned} \overline{T}(n) &\leq \Theta(n) + \frac{1}{n} \cdot \sum_{j=1}^{n} \overline{T}(\max(j-1, n-j)) \\ &\leq \Theta(n) + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} \overline{T}(j) \end{aligned}$$

Wir zeigen auf der folgenden Seite:

$$\sum_{j=\lfloor n/2 \rfloor}^{n-1} j \leq \frac{3}{8} n^2$$

$$
\begin{aligned}
\sum_{j=\lfloor n/2 \rfloor}^{n-1} j &= \sum_{j=1}^{n-1} j - \sum_{j=1}^{\lfloor n/2 \rfloor - 1} j \\
&= \frac{(n-1)\cdot n}{2} - \frac{(\lfloor n/2 \rfloor - 1)\cdot \lfloor n/2 \rfloor}{2} \\
&\leq \frac{(n-1)\cdot n}{2} - \frac{(n/2 - 1)\cdot n/2}{2} \\
&= \frac{n^2 - n}{2} - \frac{n^2/4 - n/2}{2} \\
&= \frac{n^2 - n - n^2/4 + n/2}{2} \\
&= \frac{3/4 \cdot n^2 - 1/2 \cdot n}{2} = \frac{3}{8}n^2 - \frac{1}{4}n \leq \frac{3}{8}n^2
\end{aligned}
$$

Behauptung: $\overline{T}(n) \leq c \cdot n$ für ein geeignetes $c \in \mathbb{R}^+$

Wir überprüfen dies mittels der Einsetzungsmethode:

$$\begin{aligned}
\overline{T}(n) &\leq \Theta(n) + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} \overline{T}(j) \\
&\leq c_1 \cdot n + \frac{2}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} c \cdot j \\
&= c_1 \cdot n + \frac{2c}{n} \cdot \sum_{j=\lfloor n/2 \rfloor}^{n-1} j \\
&\leq c_1 \cdot n + \frac{2c}{n} \cdot \frac{3}{8} n^2 = \left(c_1 + \frac{3}{4} c\right) \cdot n
\end{aligned}$$

Für $c \geq 4c_1$ ist damit obige Aussage gezeigt.

Median-Strategie

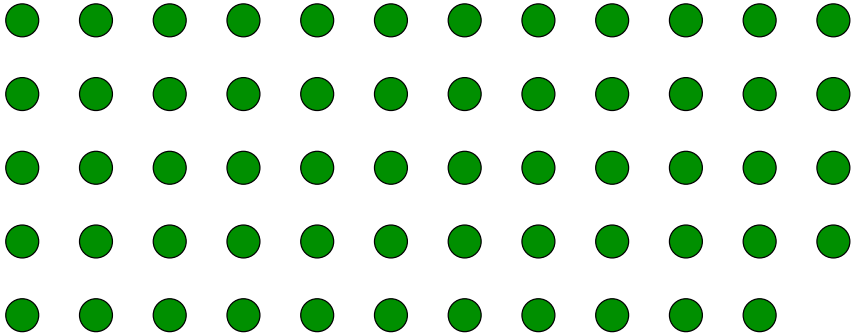
Aus dem Buch *Introduction to Algorithms* von Cormen, Leiserson, Rivest und Stein:

- generate a good pivot recursively, numbers in array A
- due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973]

`select(A, l, r, i)` liefert das `i`-kleinste Element

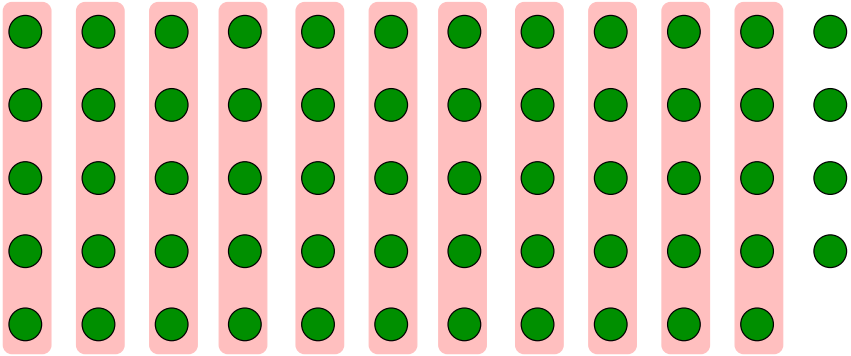
1. divide the $n = r - l + 1$ elements into groups of 5
2. find median of each 5-element group and put it into array `B`
3. recursively select the median `x` of the $\lfloor n/5 \rfloor$ group medians to be the pivot
   $\rightarrow$ `x := select(B, 0, n/5, n/10)`
4. partition around the pivot `x` and let `k := partition(A, l, r)`
5. if $i < k$ return `select(A, l, k-1, i)`
6. if $i = k$ return pivot `x`
7. if $i > k$ return `select(A, k+1, r, i)`

Median-Strategie



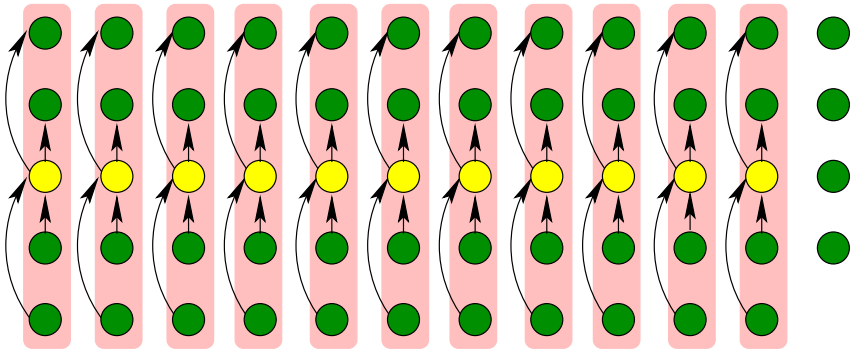
initial: $n = r - l + 1$ elements

Median-Strategie



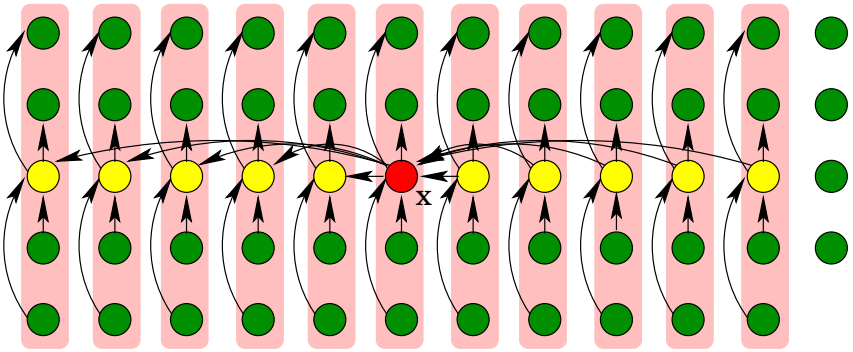
divide the $n = r - l + 1$ elements into groups of 5

Median-Strategie



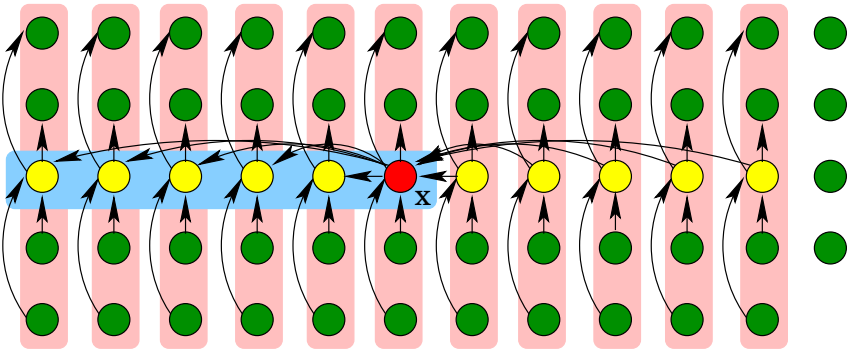
find the median of each 5-element group

Median-Strategie



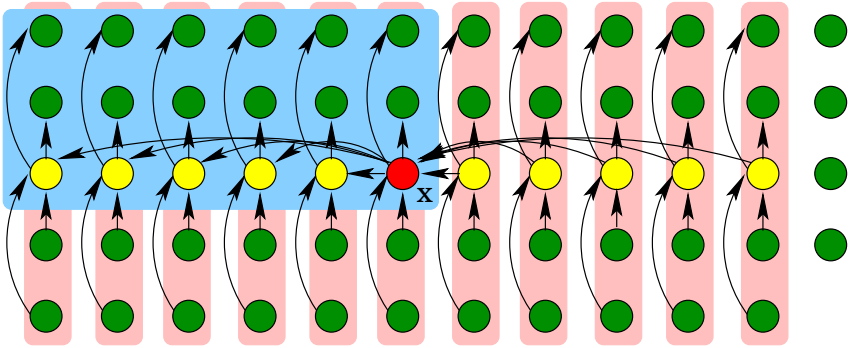
recursively `select` the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot

Median-Strategie



at least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians

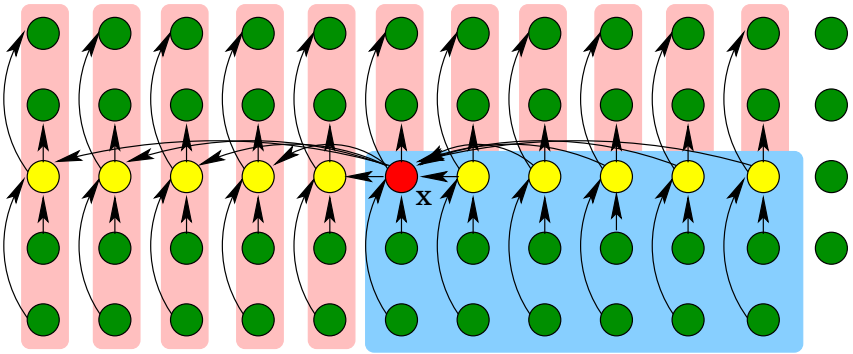
Median-Strategie



at least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians

- therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$

Median-Strategie



at least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians

- therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$
- similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$

Laufzeit:

- Aufwand zum Bestimmen des Medians der Mediane: $T(n/5)$
- Aufwand zum Bestimmen des k-kleinsten Elements: $T(7n/10)$

$\Rightarrow$ Aufwand insgesamt: $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

Vermutung: $T(n) \leq cn$

$$T(n) \leq \frac{1}{5}cn + \frac{7}{10}cn + kn = \frac{9}{10}cn + kn \overset{!}{\leq} cn$$

Wählen wir c groß genug, so ist die Ungleichung erfüllt:

$$\begin{aligned} \frac{9}{10}cn + kn \leq cn \quad &\iff \quad kn \leq cn - \frac{9}{10}cn = \frac{1}{10}cn \\ &\iff \quad 10k \leq c \end{aligned}$$

Übersicht

Übersicht

abstrakte Datentypen vs. Datenstrukturen

Ein *abstrakter Datentyp* wird definiert durch einen Wertebereich, also einer Menge von Objekten, und darauf definierten Operationen. Die Menge der Operationen nennt man die *Schnittstelle* des Datentyps.

Beispiele:

- Integer:
  - Wertebereich: ganze Zahlen
  - Operationen: Addition, Subtraktion, Multiplikation, Division, usw.
- Boolean:
  - Wertebereich: die Zahlen 0 und 1 oder die Werte `false` und `true`
  - Operationen: AND, OR, NOT, usw.
- Stack:
  - Wertebereich sind Objekte vom Typ `T`, in C++ mittels Templates
  - Operationen: push, pop, top, create, destroy usw.

Eine *Datenstruktur* ist eine Realisierung, also eine konkrete Implementierung eines abstrakten Datentyps.

abstrakte Datentypen vs. Datenstrukturen

Wir werden uns in diesem Kapitel sechs abstrakte Datentypen ansehen:

- Stack (Keller oder Stapel)
- Queue (Warteschlange)
- Sequence (Folge oder Sequenz)
- Dictionary (Wörterbuch)
- Priority-Queue (Vorrangwarteschlange)
- Set (Menge)

Die Operationen können rein mathematisch, also unabhängig von einer konkreten Implementierung definiert werden. Die Namen der Operationen können in Büchern, auf Internet-Seiten oder in Bibliotheken (API) von den hier gewählten Namen abweichen.

Im Folgenden bezeichne `D` jeweils den abstrakten Datentyp, die Objekte seien jeweils vom Typ `T`, und `Bool` bezeichne die Menge der booleschen Werte.

Zunächst geben wir Definitions- und Wertebereich der Operationen an, die Signatur:

- `create`: $\rightarrow$ `D`
- `push`: `D` $\times$ `T` $\rightarrow$ `D`
- `pop`: `D` $\rightarrow$ `D` $\cup$ {'error'}
- `top`: `D` $\rightarrow$ `T` $\cup$ {'error'}
- `empty`: `D` $\rightarrow$ `Bool`

in C++:

```
std::stack<int> s;
s.push(42);
s.pop();
int x = s.top();
if (s.empty())
```

Anschließend definieren wir die Semantik, also das Verhalten der Operationen:

- $create := ()$
- $push((a_1, \ldots, a_n), b) := (b, a_1, \ldots, a_n)$
- $pop((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n) & \text{falls } n \geq 1 \\ \text{undefiniert} & \text{sonst} \end{cases}$
- $top((a_1, \ldots, a_n)) := \begin{cases} a_1 & \text{falls } n \geq 1 \\ \text{undefiniert} & \text{sonst} \end{cases}$
- $empty((a_1, \ldots, a_n) := \begin{cases} 0 & \text{falls } n \geq 1 \\ 1 & \text{sonst} \end{cases}$

Stack

Wer es gerne weniger mathematisch mag:

- `create()` erzeugt einen neuen, leeren Stack (Last-In, First-Out)
  $\rightarrow$ in C++ der Konstruktor
- `push(D, x)` fügt Element `x` in den Stack `D` ein
- `top(D)` liefert das zuletzt eingefügte Element des Stacks `D`
- `pop(D)` entfernt das zuletzt eingefügte Element aus dem Stack `D`
- `empty(D)` gibt an, ob der Stack `D` leer ist

Für eine konkrete Implementierung definiert man oft weitere Operationen:

- `destroy(D)` zerstört den Stack `D` und gibt belegten Speicher frei
  $\rightarrow$ in C++ der Destruktor
- `size(D)` liefert die Größe des Stacks `D` als Anzahl gespeicherter Objekte

Stack

Ein Stack (Stapel, Kellerspeicher) ist also ein abstrakter Datentyp, bei dem Elemente in der sogenannten *LIFO = Last-In-First-Out*-Reihenfolge abgelegt werden und wieder genutzt werden können.

- Die Daten werden in der umgekehrten Reihenfolge gelesen wie geschrieben.
- Anwendung: Auf diese Art werden die Daten bei Funktionsaufrufen auf den *Stack-Speicher* gelegt.

Im folgenden Beispiel sehen wir, wie wir ohne Kenntnis der konkreten Implementierung eines Stacks ein Programm zur Auswertung arithmetischer Ausdrücke schreiben können.

- Die Datei `stack.h` enthält nur die Beschreibung des abstrakten Datentyps `stack_t` sowie die Funktionen, die bereitgestellt werden.
- Die eigentliche Implementierung findet sich typischerweise in einer Datei `stack.c`, die bereits kompiliert ist und durch den Linker dem ausführbaren Programm hinzugefügt wird.

Beispiel: Ausdruck $5,1 \cdot ((91 + 28) \cdot 4,3)$ in UPN: `5.1 91 28 + 4.3 * *`

```
#include <stdio.h>
#include "stack.h"
int main(void) {
    stack_t *S = create();
    push(S, 5.1); push(S, 91); push(S, 28);
    double y = top(S); pop(S);
    double x = top(S); pop(S);
    push(S, x + y);
    push(S, 4.3);
    y = top(S); pop(S);
    x = top(S); pop(S);
    push(S, x * y);
    y = top(S); pop(S);
    x = top(S); pop(S);
    push(S, x * y);
    printf("%lf\n", top());
}
```

Wir beschreiben zunächst wieder die Definitions- und Wertebereiche der Operationen:

- `create`: $\to$ `D`
- `enter`: `D` $\times$ `T` $\to$ `D`
- `leave`: `D` $\to$ `D` $\cup$ {'error'}
- `front`: `D` $\to$ `T` $\cup$ {'error'}
- `empty`: `D` $\to$ `Bool`

in C++:

```
std::queue<int> q;
q.push(42);
q.pop();
int x = q.front();
if (q.empty())
```

Dann beschreiben wir die Semantik, also das Verhalten der Operationen:

- $create() := ()$
- $enter((a_1, \ldots, a_n), b) := (a_1, \ldots, a_n, b)$
- $leave((a_1, \ldots, a_n)) := \begin{cases} (a_2, \ldots, a_n) & \text{falls } n \geq 1 \\ \text{undefiniert} & \text{sonst} \end{cases}$
- $front((a_1, \ldots, a_n)) := \begin{cases} a_1 & \text{falls } n \geq 1 \\ \text{undefiniert} & \text{sonst} \end{cases}$
- $empty((a_1, \ldots, a_n)) := \begin{cases} 0 & \text{falls } n \geq 1 \\ 1 & \text{sonst} \end{cases}$

Die Semantik der Operationen können wir auch ohne Mathematik beschreiben:

- `create()` erzeugt eine leere Queue (First-In, First-Out)
- `enter(D, x)` fügt Element `x` in die Queue `D` ein
- `front(D)` liefert zuerst eingefügtes Element von `D`
- `leave(D)` entfernt zuerst eingefügtes Element aus `D`
- `empty(D)` gibt an, ob die Queue `D` leer ist
- `destroy(D)` zerstört die Queue `D` und gibt den belegten Speicher frei
- `size(D)` liefert die Größe der Queue `D` als Anzahl gespeicherter Objekte

Anwendungen: • Pufferung zwischen unterschiedlich leistungsfähigen Datenquellen wie z.B. Druckerwarteschlangen • *pipes* in Unix • Ereignisverarbeitung

Anmerkungen: • Die Operation `enter` wird auch als `enqueue`, `put` oder `push` bezeichnet. • Die Operation `leave` wird auch `dequeue`, `get` oder `pop` genannt. • Die Operation `front` heißt auch `first`, `head` oder `top`.

Wir beschreiben nur die Semantik der Operationen, die Definitions- und Wertebereiche der Operationen ergeben sich daraus:

- `create()` erzeugt leere Sequenz bzw. Folge
- `put(D, x, p)` fügt das Element `x` vor dem Element an Position `p` in die Folge `D` ein; für `p = NIL` wird `x` als letztes Element eingefügt
- `get(D, p)` liefert das Element an Position `p` der Folge `D`
- `erase(D, p)` entfernt das Element an Position `p` aus der Folge `D`
- `empty(D)` gibt an, ob die Folge leer ist
- `conc(D1, D2)` hängt die Folge `D2` hinten an die Folge `D1` an (concatenate)
- `destroy(D)` zerstört die Folge und gibt Speicher frei
- `size(D)` liefert die Größe der Folge als Anzahl gespeicherter Objekte

Ein Wörterbuch stellt eine rechtseindeutige Relation $R \subseteq K \times V$ dar, also eine partielle Funktion $K \to V$, bei der jedem Schlüssel $k \in K$ höchstens ein Wert $v \in V$ zugeordnet ist. K bezeichnet die Menge der Schlüssel (key) und V die Menge der Werte (value).

Auch hier beschreiben wir nur die Operationen:

- `create()` erzeugt ein leeres Wörterbuch
- `insert(D, k, v)` Falls der Schlüssel `k` noch nicht im Wörterbuch `D` enthalten ist, wird Wert `v` mit Schlüssel `k` in `D` eingefügt. Sonst wird der zum Schlüssel `k` zugeordnete Wert auf `v` geändert.
- `erase(k)` entfernt Schlüssel `k` inklusive seinem Wert aus dem Wörterbuch `D`
- `search(k)` sucht den Schlüssel `k` im Wörterbuch `D` und gibt den dazu gespeicherten Wert zurück, falls vorhanden, sonst NIL
- `empty(D)` gibt an, ob das Wörterbuch leer ist
- `destroy(D)` zerstört das Wörterbuch und gibt belegten Speicher frei
- `size(D)` liefert die Größe des Wörterbuchs als Anzahl gespeicherter Schlüssel-Wert-Paare

Schlüssel

In der Informatik ist der Begriff *Schlüssel* oft nicht klar definiert bzw. der Begriff wird nicht einheitlich verwendet.

- Bei Datenbanksystemen wird ein Datensatz eindeutig durch einen Schlüssel identifiziert. Das kann ein künstlicher Schlüssel wie eine ID sein, oder ein fachlicher Schlüssel wie eine IBAN oder eine Telefonnummer.
- Bei Algorithmen und Datenstrukturen bezeichnet ein Schlüssel oft nur etwas, anhand dessen Werte sortiert werden, z.B. den Namen einer Person. Solche „Schlüssel“ können dann mehrfach in einer Datenstruktur vorkommen.

Bei der Vorrangwarteschlange (Priority-Queue) bezeichnet „Schlüssel“ sogar etwas ganz anderes: eine Priorität. Dort sind die Werte entsprechend einer Priorität angeordnet, wobei verschiedene Datensätze (Werte) gleiche Priorität (Schlüssel) haben können.

Auch hier wird eine Relation $R \subseteq K \times V$ dargestellt, aber R muss nicht rechtseindeutig sein. K ist die Menge der „Schlüssel“ (Priorität) und V die Menge der Werte (Daten).

Folgende Operationen muss eine Priority-Queue unterstützen:

- `create()` erzeugt eine neue Priority-Queue ohne Elemente.
- `insert(D, k, v)` fügt Wert `v` mit „Schlüssel“ bzw. Priorität `k` in `D` ein.
- `minimum(D)` liefert den Wert mit dem minimalen Schlüssel/Priorität in `D`.
- `extractMin(D)` entfernt den Wert/Datensatz mit dem minimalen Schlüssel bzw. Priorität aus `D`.
- `decreaseKey(D, v, k)` weist dem Wert `v` der Priority-Queue `D` einen neuen, kleineren „Schlüssel“ bzw. Prioritätswert `k` zu.
- `erase(D, v)` entfernt den Wert `v` aus Priority-Queue `D`.
- `empty(D)` gibt an, ob die Priority-Queue `D` leer ist.
- `destroy(D)` zerstört Priority-Queue `D` und gibt den belegten Speicher frei.
- `size(D)` liefert die Anzahl der in `D` gespeicherten Schlüssel-Wert-Paare.

Menge

Folgende Operationen muss der abstrakte Datentyp Menge unterstützen:

- `create()` erzeugt eine neue Menge ohne Elemente.
- `insert(D, k)` fügt Wert `k` in Menge `D` ein.
- `erase(D, k)` entfernt den Wert `k` aus der Menge `D`.
- `member(D, k)` gibt an, ob die Menge `D` das Element `k` enthält.
- `empty(D)` gibt an, ob die Menge `D` leer ist.
- `destroy(D)` zerstört Menge `D` und gibt den belegten Speicher frei.
- `size(D)` gibt die Anzahl der Elemente der Menge `D` an.

Oft sollen auch folgende Operationen unterstützt werden:

- `subset(D1, D2)` gibt an, ob `D1` eine Teilmenge von `D2` ist. $D_1 \subseteq D_2$?
- `union(D1, D2)` liefert Vereinigungsmenge aus `D1` und `D2`. $D := D_1 \cup D_2$
- `cut(D1, D2)` liefert Schnittmenge von `D1` und `D2`. $D := D_1 \cap D_2$
- `diff(D1, D2)` liefert Differenzmenge von `D1` und `D2`. $D := D_1 - D_2$

abstrakte Datentypen vs. Datenstrukturen

Eine *Datenstruktur* ist eine Realisierung, also eine konkrete Implementierung eines abstrakten Datentyps.

Objektorientierte Programmiersprachen unterstützen dieses Konzept sehr gut.

- Die Klassendeklaration oder ein Interface (Java) oder eine abstrakte Klasse (C++), die nur rein virtuelle Methoden enthält, definiert die Objekte sowie die darauf definierten Operationen.
  In Java kann mittels Generics und in C++ kann mittels Templates der Wertebereich auch von außen festgelegt werden.
- Die konkreten oder abgeleiteten Klassen implementieren dann den Datentyp auf eine spezielle Art und Weise, und diese Klassen sind dann die Datenstrukturen.
- In C++ können die Operatoren wie + oder ∗ für selbstdefinierte, abstrakte Datentypen überladen werden.

Auch in C kann man die Schnittstelle von der Implementierung trennen: Header- und Code-Dateien.

In der STL von C++ können verschiedene Datenstrukturen genutzt werden, um einen Datentyp wie `std::stack` oder `std::queue` zu realisieren.

```
int main(void) {
    cout << "     n      |  deque |   list" << endl;
    cout << "------------+--------+-------" << endl;
    for (int n = MIN; n <= MAX; n *= 2) {
        queue<int, deque<int> > q1; // !!!!!
                //  ^^^^^^^^^^^^  !!!!!!!!!!!!
        clock_t t1 = clock();
        for (int i = 0; i < n; i++)
            q1.push(i);
        for (int i = 0; i < n; i++)
            q1.pop();
        clock_t t2 = clock();
        long t = (t2 - t1) * 1000;  // Zeit in Millisekunden
        cout << setw(10) << n << " | "
             << setw(6) << t / CLOCKS_PER_SEC << flush;
```

```
        queue<int, list<int> > q2; // !!!!!
                // ^^^^^^^^^ !!!!!!!!!!!!!!

        t1 = clock();
        for (int i = 0; i < n; i++)
            q2.push(i);
        for (int i = 0; i < n; i++)
            q2.pop();
        t2 = clock();
        t = t2 - t1;
        t *= 1000;  // Zeit in Millisekunden
        cout << " | " << setw(6) << t / CLOCKS_PER_SEC << endl;
    }
}
```

Übersetzt mittels `g++ -Wall -Wextra -O3`

Ausgabe auf einem Intel Core i7-4900MQ, Linux Ubuntu 22.04 LTS, Kernel 5.15.0-70, 16 GB RAM, gcc 11.3.0

```
     n      |  deque |   list
------------+--------+-------
   1048576  |      3 |     43
   2097152  |     11 |     72
   4194304  |      8 |    143
   8388608  |     42 |    283
  16777216  |     34 |    567
  33554432  |    160 |   1131
  67108864  |    132 |   2267
```

Offensichtlich hat die Art der Implementierung einen großen Einfluss auf die Laufzeit der Operationen.

abstrakte Datentypen vs. Datenstrukturen

In der Praxis wird die Unterscheidung zwischen dem (abstrakten) Datentyp und der (konkreten) Datenstruktur, also der konkreten Implementierung eines Datentyps nicht immer konsequent eingehalten:

`std::set` und `std::unordered_set` sind beides Datenstrukturen, also konkrete Implementierungen für den abstrakten Datentyp Menge (Set).

- `std::set` ist eine Realisierung, die auf Suchbäumen basiert,
- `std::unordered_set` basiert auf Hash-Tabellen,
- in C++ 2023 ist der Wrapper `std::flat_set` hinzugekommen.

Was das genau bedeutet, sehen wir auf den folgenden Seiten.

Gleichartige Daten (ob vom elementaren Datentyp oder von Strukturen) werden meist bei abschätzbarer Datenmenge in einem Array gehalten, ansonsten wird die Größe der Datenstruktur dynamisch verwaltet.

Statische Datenobjekte

- Speicherbedarf muss zum Zeitpunkt der Programmerstellung bekannt sein
- automatische Speicherverwaltung
- Speicherbelegung bei Programmstart
- fortlaufender Speicherbereich

Dynamische Datenobjekte

- Größe des Speicherbedarfs passt sich zur Laufzeit an den tatsächlichen Bedarf an
- programmlaufabhängige Speicherverwaltung
- Speicherbelegung nach Bedarf zur Laufzeit
- oft verketteter Speicherbereich

Je nachdem, wie ein (abstrakter) Datentyp implementiert wird, also welche konkrete Datenstruktur verwendet wird, ergeben sich unterschiedliche Laufzeiten für die Operationen.

Übersicht

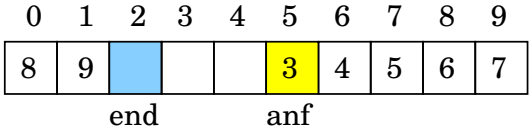
Da Arrays eine feste Größe haben, können die abstrakten Datentypen nur dann mittels Arrays implementiert werden, wenn die maximale Anzahl der Objekte beschränkt ist und abgeschätzt werden kann.

Implementieren wir unsere sechs abstrakten Datentypen mittels eines Arrays, dann ergeben sich folgende Worst-Case-Laufzeiten.

- *Stack:* hinten anhängen; hinten entfernen
  `push` $\in \mathcal{O}(1)$, `pop` $\in \mathcal{O}(1)$, `top` $\in \mathcal{O}(1)$
- *Queue:* hinten anhängen; vorne entfernen, dabei aufschieben
  `enter` $\in \mathcal{O}(1)$, `front` $\in \mathcal{O}(1)$, `leave` $\in \mathcal{O}(n)$
- *Sequence:* Objekte müssen verschoben bzw. kopiert werden
  `put` $\in \mathcal{O}(n)$, `get` $\in \mathcal{O}(1)$, `conc` $\in \mathcal{O}(n)$

Die Operationen `create`, `empty` und `size` haben jeweils konstante Laufzeit, also $\mathcal{O}(1)$.

Alternativ bei *Queue:* Nutze Indizes `anf` und `end`. $\rightarrow$ zirkuläres Array



Sei `N` die Größe des Arrays. (ohne Fehlerbehandlung)

- `create()`: $\in \mathcal{O}(1)$

```
size = anf = end = 0;
```

- `enter(D, x)`: $\in \mathcal{O}(1)$

```
D[end] = x;
end = (end + 1) % N;
size += 1;
```

- `front(D)`: $\in \mathcal{O}(1)$

```
return D[anf];
```

- `leave(D)`: $\in \mathcal{O}(1)$

```
anf = (anf + 1) % N;
size -= 1;
```

- `empty(D)`: $\in \mathcal{O}(1)$

```
return size == 0;
```

Unterscheide bei den letzten drei Datentypen, ob die Werte sortiert oder unsortiert gespeichert sind. Bei sortierter Ablage müssen die Daten ggf. verschoben werden.

- *Dictionary:* Binäre Suche bei sortierten Daten.
  - sortiert nach Schlüsseln: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `search` $\in \mathcal{O}(\log(n))$
  - unsortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `search` $\in \mathcal{O}(n)$
- *Priority-Queue:* Indizes `anf` und `end` nutzen $\rightarrow$ zirkuläres Array
  - sortiert nach Schlüsseln: `insert` $\in \mathcal{O}(n)$, `minimum` $\in \mathcal{O}(1)$, `extractMin` $\in \mathcal{O}(1)$, `decreaseKey` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$
  - unsortiert: `insert` $\in \mathcal{O}(1)$, `minimum` $\in \mathcal{O}(n)$, `extractMin` $\in \mathcal{O}(n)$, `decreaseKey` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$
- *Menge:* Binäre Suche bei sortierten Daten.
  - sortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `member` $\in \mathcal{O}(\log(n))$
  - unsortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `member` $\in \mathcal{O}(n)$

Übung 14. Bei den Operationen `erase` und `decreaseKey` nützt die Sortierung anhand der Schlüssel nichts, da die eigentlichen Werte gesucht werden müssen. Wie würden sich die Laufzeiten der Operationen ändern, wenn sie als weiteren Parameter den Index hätten, wo sich das entsprechende Objekt befindet?

Bei den Laufzeiten in unsortierten Arrays muss folgendes berücksichtigt werden.

- Wörterbuch (Dictionary): Ist der Schlüssel `k` bereits vorhanden, so wird der alte Wert durch `v` ersetzt, wenn ein `insert(D,k,v)` ausgeführt wird. Es muss also in jedem Fall zuerst der Schlüssel `k` gesucht werden. $\rightarrow$ `insert` $\in \mathcal{O}(n)$
- Menge (Set): In einer Menge ist jedes Element höchstens einmal enthalten. Somit muss beim Einfügen `insert(D,k)` geprüft werden, ob das Element `k` bereits in `D` enthalten ist. $\rightarrow$ `insert` $\in \mathcal{O}(n)$

$\rightarrow$ Obwohl das Einfügen in ein unsortiertes Array sehr schnell in Zeit $\mathcal{O}(1)$ erfolgen könnte, indem das neue Element einfach hinten angehängt wird, bringt es hier keine Vorteile.

Bei sortierten Daten kann ein Schlüssel zwar sehr schnell gefunden werden, aber damit nach einem Einfügen oder Löschen die Sortierung erhalten bleibt, müssen die anderen Elemente verschoben werden. Erfolgt die Operation in der Mitte, so müssen $n/2 \in \mathcal{O}(n)$ viele Elemente verschoben werden. Dadurch geht auch hier der Vorteil wieder verloren.

Soll ein Element eingefügt oder angehängt werden, obwohl kein Platz mehr vorhanden ist, muss das Array vergrößert werden:

- Speicherbereich für $n+1$ Elemente allokieren.
- Verschiebe n Elemente in den größeren Speicherbereich.
- Gib alten Speicherbereich frei.
- Füge neues Element in das Array ein.

$\rightarrow$ Worst-Case-Laufzeit für `insert`: $\mathcal{O}(n)$

N Ausführungen von `insert` dauern also:

$$\sum_{i=1}^{N} T_{insert}(i) \leq \sum_{i=1}^{N} c \cdot i = c \cdot \sum_{i=1}^{N} i = c \cdot \frac{N \cdot (N+1)}{2} \in \mathcal{O}(N^2)$$

Besser: Vergrößere Speicherbereich nicht nur um ein einziges Element sondern von n auf $2n$ Elemente. $\rightarrow$ Ein `insert` ist nur sehr selten teuer!

N Ausführungen von `insert` dauern jetzt:

$$\sum_{i=1}^{N} T_{insert}(i) \;=\; \sum_{i=1}^{N} 1 + \sum_{i=1}^{\log_2(N)} 2^i \;\leq\; N + 2^{\log_2(N)+1} \;=\; 3 \cdot N \;\in\; \mathcal{O}(N)$$

Amortisierte[22] Worst-Case-Laufzeit von `insert`:

$$\frac{1}{N} \cdot \sum_{i=1}^{N} T_{insert}(i) \;\leq\; \frac{3 \cdot N}{N} \;=\; 3 \quad\rightarrow\quad \widetilde{T}_{insert} \in \mathcal{O}(1)$$

Die Datenstruktur `std::vector`[23] in C++ ist so implementiert: The storage of the vector is handled automatically, being expanded as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted.

[22] amortisieren: ausgleichen, aufwiegen

[23] `https://en.cppreference.com/w/cpp/container/vector`

dynamisches Array

```
#include <iostream>
#include <vector>
using namespace std;

int main(void) {
    vector<int> v;
    size_t c = v.capacity();

    cout << "0: " << c << endl;
    for (int i = 1; i <= 512; i++) {
        v.push_back(i);
        if (v.capacity() != c) {
            c = v.capacity();
            cout << i << ": " << c << endl;
        }
    }
}
```

Ausgabe:

```
0: 0
1: 1
2: 2
3: 4
5: 8
9: 16
17: 32
33: 64
65: 128
129: 256
257: 512
```

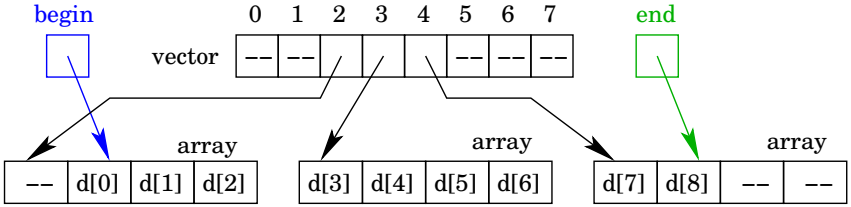
Die Datenstruktur `std::deque`[24] in C++ ist auch mittels dynamischer Arrays implementiert, allerdings anders als bei `std::vector`, damit Elemente am Anfang und am Ende in konstanter, amortisierter Zeit eingefügt und entnommen werden können:

> As opposed to `std::vector`, the elements of a `std::deque` are not stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays, with additional bookkeeping, which means indexed access to deque must perform two pointer dereferences, compared to vector's indexed access which performs only one.
>
> The storage of a `std::deque` is automatically expanded and contracted as needed. Expansion of a `std::deque` is cheaper than the expansion of a `std::vector` because it does not involve copying of the existing elements to a new memory location.

[24] `https://en.cppreference.com/w/cpp/container/deque`

Prinzip / grundsätzlicher Aufbau von `std::deque`:

- Ein `std::vector` verwaltet eine Menge von gleich großen Arrays.
- Die Arrays speichern konsekutiv die eigentlichen Daten.
- Beim Vergrößern des Vektors wird vorne und hinten freier Platz eingefügt.



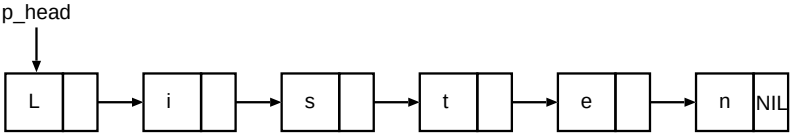
Laufzeiten:

- Einfügen vorne/hinten oder Entfernen vorne/hinten: $\mathcal{O}(1)$ amortisiert
- Einfügen oder Entfernen sonst: $\mathcal{O}(n)$

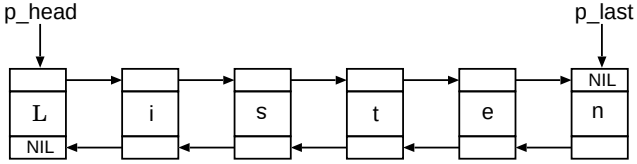
Übersicht

Durch Zeiger verkettete Folge von Elementen eines gegebenen Typs T.

- *einfach* verkettet[25]: Jedes Listenelement enthält neben den Nutzdaten vom Typ T auch *einen Verweis* (Zeiger) auf seinen unmittelbaren Nachfolger.



- *doppelt* verkettet: Jedes Listenelement enthält *zwei Verweise*, einen auf seinen direkten Nachfolger und einen auf seinen direkten Vorgänger.



[25] Kann auch einen Zeiger `p_last` besitzen, falls z.B. Elemente am Ende angehängt werden sollen.

Die Datenstruktur `std::list`[26] in C++ ist als doppelt verkettete Liste implementiert:

> `std::list` is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as a doubly-linked list.

Die Datenstruktur `std::forward_list`[27] in C++ ist als einfach verkettete Liste implementiert:

> `std::forward_list` is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list. Compared to `std::list` this container provides more space efficient storage when bidirectional iteration is not needed.

[26] `https://en.cppreference.com/w/cpp/container/list`

[27] `https://en.cppreference.com/w/cpp/container/forward_list`

In C++ werden Iteratoren bereitgestellt, die ein Traversieren, also ein Durchlaufen der Elemente ermöglichen.

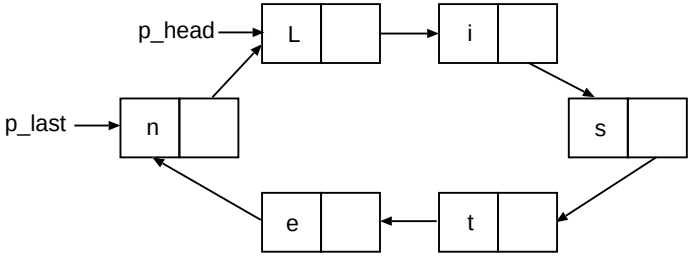
- Die einfach verkettete Liste `std::forward_list` stellt nur den Iterator `begin` bereit, damit die Liste von vorne nach hinten durchlaufen werden kann.
- Die doppelt verkettete Liste `std::list` stellt zusätzlich auch den Iterator `rbegin` bereit, um die Liste von hinten nach vorne zu durchlaufen. (r für reverse)

Außerdem besitzt die einfach verkettete Liste offenbar keinen Zeiger auf das Ende der Liste, denn

- die einfach verkettete Liste `std::forward_list` besitzt nur die Operationen `push_front`, `front` und `pop_front`,
- wohingegen die doppelt verkettete Liste `std::list` zusätzlich die Operationen `push_back`, `back` und `pop_back` bereitstellt.

Zirkulare Liste oder Ringliste: Jedes Listenelement hat einen Nachfolger, der Zeiger des letzten Elements zeigt wieder auf das erste Element.

einfach verkettet:

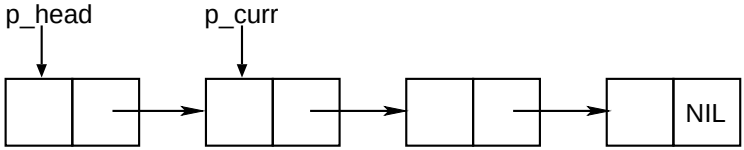


Einige Anwendungen:

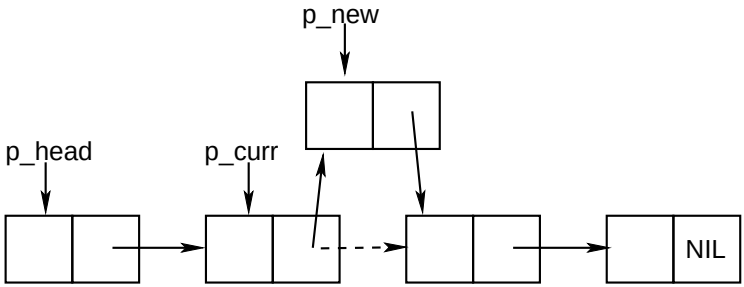
- geordnete Ablage von Daten,
- bei der Speicherverwaltung zur Verkettung der freien Speicherbereiche,
- schwach besetzte Matrizen.

ein Element hinter `p_curr` in eine Liste (sortiert) einfügen

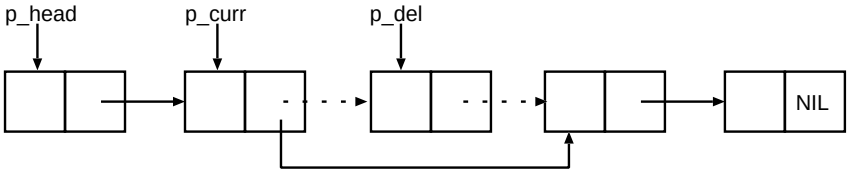
vorher:



nachher:



das Element hinter `p_curr` aus einer Liste entfernen



Vor- und Nachteile bei einer einfach verketteten Liste:

- Vorteil: Weniger Platzbedarf als doppelt verkettete Liste.
- Nachteil: Die Operation `erase(element_t *p)` dauert länger, da der Vorgänger des Elements, auf das `p` zeigt, nicht bekannt ist. Das Element muss gesucht werden, indem die Liste vom Anfang durchlaufen wird.

→ Die einfach verkettete Liste `std::forward_list` hat daher die Operation `erase_after` anstelle von `erase`.

In unserer Implementierung sind jeweils vorne und hinten weitere (Dummy-) Elemente eingefügt, die keine Werte speichern. Sie sollen nur die Implementierung vereinfachen, indem weniger Fälle zu unterscheiden sind.

liste.h

```
#ifndef _LISTE_H
#define _LISTE_H

// some error flags ----------------------------------------
extern char OKAY;
extern char NOT_FOUND;
extern char EMPTY_LIST;

// incomplete data types (forward declaration) -------------
typedef struct element element_t;
typedef struct list list_t;
```

```
// interface ------------------------------------------------
list_t *create();
void destroy(list_t *l);
element_t *find(list_t *l, int v);

void push_back(list_t *l, int v);  // insert as last element
int back(list_t *l);
void pop_back(list_t *l);
void push_front(list_t *l, int v);  // insert as first element
int front(list_t *l);
void pop_front(list_t *l);
void insert(list_t *l, element_t *p, int v); // insert after p
void erase(list_t *l, element_t *p);
size_t size(list_t *l);    // additional functions
char empty(list_t *l);
char getError(list_t *l);

#endif
```

liste.c

```
#include <malloc.h>
#include "liste.h"

char OKAY = 0;
char NOT_FOUND = 1;
char EMPTY_LIST = 2;

struct element {   // doubly linked: successor and predecessor
    int value;
    element_t *succ, *pred;
};

struct list {      // header: information about the list
    size_t len;
    char error;
    element_t *head, *last;
};
```

```
list_t *create() {
    list_t *l = (list_t *) malloc(sizeof(list_t));
    element_t *head = (element_t *) malloc(sizeof(element_t));
    element_t *last = (element_t *) malloc(sizeof(element_t));

    head->value = 0;  // dummy element
    head->succ = last;
    head->pred = NULL;

    last->value = 0;  // dummy element
    last->succ = NULL;
    last->pred = head;

    l->head = head;
    l->last = last;
    l->len = 0;

    return l;
}
```

```
void destroy(list_t *l) {
    element_t *curr = l->head;
    element_t *next = curr->succ;

    while (next != NULL) {
        free(curr);
        curr = next;
        next = curr->succ;
    }
    free(curr);
}
```

```
element_t *find(list_t *l, int v) {
    element_t *curr = l->head->succ;

    while (curr != l->last && curr->value != v)
        curr = curr->succ;

    if (curr == l->last) {
        l->error = NOT_FOUND;
        return NULL;
    }
    return curr;
}
```

```
void push_back(list_t *l, int v) {
    insert(l, l->last->pred, v);
}

int back(list_t *l) {
    if (l->len == 0) {
        l->error = EMPTY_LIST;
        return 0;
    }
    return l->last->pred->value;
}

void pop_back(list_t *l) {
    if (l->len == 0) {
        l->error = EMPTY_LIST;
        return;
    }
    erase(l, l->last->pred);
}
```

```
void push_front(list_t *l, int v) {
    insert(l, l->head, v);
}

int front(list_t *l) {
    if (l->len == 0) {
        l->error = EMPTY_LIST;
        return 0;
    }
    return l->head->succ->value;
}

void pop_front(list_t *l) {
    if (l->len == 0) {
        l->error = EMPTY_LIST;
        return;
    }
    erase(l, l->head->succ);
}
```

```
void insert(list_t *l, element_t *p, int v) {
    element_t *n = (element_t *) malloc(sizeof(element_t));

    n->value = v;
    n->succ = p->succ;
    n->pred = p;

    p->succ = n;
    n->succ->pred = n;
    l->len += 1;
}
```

```
void erase(list_t *l, element_t *p) {
    p->succ->pred = p->pred;
    p->pred->succ = p->succ;
    free(p);
    l->len -= 1;
}

size_t size(list_t *l) {
    return l->len;
}

char empty(list_t *l) {
    return l->len == 0;
}
```

main.c

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
    printf("Stack:  "); // nutze verkettete Liste für Stack
    list_t *stack = create();
    for (int i = 0; i < 10; i++)
        push_back(stack, i);

    while (! empty(stack)) {
        printf("%2d  ", back(stack));
        pop_back(stack);
    }
    printf("\nsize of stack: %lu\n\n", size(stack));
    destroy(stack);
```

```
printf("Folge:  "); // nutze verkettete Liste für Sequence
int values[10] =       {3,2,1,6,5,4,9,8,7,0};
int predecessors[10] = {0,0,0,3,3,3,6,6,6,0};
list_t *seq = create();
for (int i = 0; i < 10; i++) {
    element_t *c = find(seq, predecessors[i]);
    if (c != NULL)
        insert(seq, c, values[i]);
    else push_front(seq, values[i]);
}

while (! empty(seq)) {
    printf("%2d  ", front(seq));
    pop_front(seq);
}
printf("\nsize of seq: %lu\n\n", size(seq));
destroy(seq);
```

```
    printf("Queue:  "); // nutze verkettete Liste für Queue
    list_t *queue = create();
    for (int i = 0; i < 10; i++)
        push_back(queue, i);

    while (! empty(queue)) {
        printf("%2d  ", front(queue));
        pop_front(queue);
    }
    printf("\nsize of queue: %lu\n", size(queue));
    destroy(queue);

    return 0;
}
```

Ausgabe des Programms:

```
Stack:   9   8   7   6   5   4   3   2   1   0
size of stack: 0

Folge:   0   1   2   3   4   5   6   7   8   9
size of seq: 0

Queue:   0   1   2   3   4   5   6   7   8   9
size of queue: 0
```

Übung 15. Implementieren Sie eine (a) einfach (b) doppelt verkettete Liste mit einem Zeiger auf den Anfang und einem Zeiger auf das Ende der Liste, aber ohne zusätzliche Dummy-Elemente.

Welche Fallunterscheidungen sind nötig, die bei der obigen Implementierung mit zusätzlichen Dummy-Elementen nicht nötig waren?

Implementieren wir unsere sechs abstrakten Datentypen mittels einer doppelt verketteten Liste mit jeweils einem Zeiger auf den Anfang und das Ende der Liste, dann ergeben sich folgende Worst-Case-Laufzeiten.

- *Stack:* hinten anhängen, hinten entfernen
  `push` $\in \mathcal{O}(1)$, `pop` $\in \mathcal{O}(1)$, `top` $\in \mathcal{O}(1)$
- *Queue:* hinten anhängen, vorne entfernen
  `enter` $\in \mathcal{O}(1)$, `leave` $\in \mathcal{O}(1)$, `front` $\in \mathcal{O}(1)$
- *Sequence:*
  - Ein Zeiger[28] zum Einfügen und Entfernen ist jeweils als Parameter gegeben, daher gilt `put` $\in \mathcal{O}(1)$ und `get` $\in \mathcal{O}(1)$.
  - Bei `conc(r,s)` wird der Zeiger `succ` vom letzten Element aus `r` auf das erste Element von `s` gesetzt, daher gilt `conc` $\in \mathcal{O}(1)$.

Die Operationen `create`, `empty` und `size` haben jeweils konstante Laufzeit, also $\mathcal{O}(1)$.

[28] Ist nur die Position gegeben, also als wievieltes Element der neue Wert einzufügen ist bzw. das wievielte Element zu löschen ist, dann erhalten wir als Laufzeiten `put` $\in \mathcal{O}(n)$ und `get` $\in \mathcal{O}(n)$ durch das Iterieren bis zur gewünschten Stelle.

Übung 16. Wir betrachten als Datenstruktur eine (I) einfach bzw. eine (II) doppelt verkettete Liste, die (a) nur einen Zeiger auf das erste Element bzw. (b) jeweils einen Zeiger auf das erste und letzte Element besitzt.

Geben Sie für alle Kombinationen I-a, I-b, II-a und II-b die Laufzeiten für die Operationen `get`, `put`, `erase` und `conc` des Datentyps Sequenz an.

Wir unterscheiden wieder, ob die Werte sortiert oder unsortiert gespeichert sind.

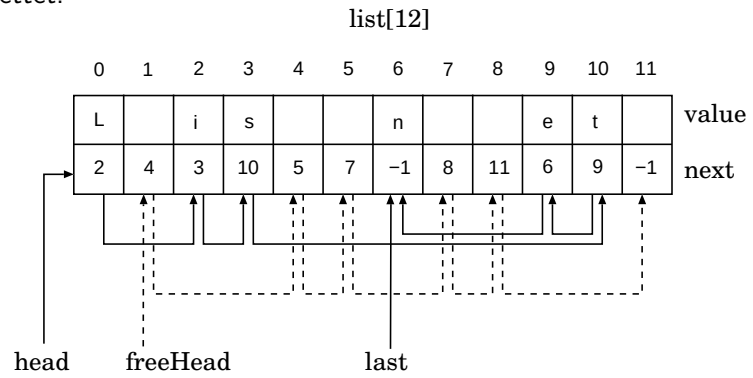
- *Dictionary:* keine binäre Suche möglich, da kein wahlfreier Zugriff
  - sortiert nach Schlüsseln: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `search` $\in \mathcal{O}(n)$
  - unsortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `search` $\in \mathcal{O}(n)$
- *Priority-Queue:*
  - sortiert nach Schlüsseln: `insert` $\in \mathcal{O}(n)$, `minimum` $\in \mathcal{O}(1)$, `extractMin` $\in \mathcal{O}(1)$, `decreaseKey` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$
  - unsortiert: `insert` $\in \mathcal{O}(1)$, `minimum` $\in \mathcal{O}(n)$, `extractMin` $\in \mathcal{O}(n)$, `decreaseKey` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$
- *Menge:* keine binäre Suche möglich, da kein wahlfreier Zugriff
  - sortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `member` $\in \mathcal{O}(n)$
  - unsortiert: `insert` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$, `member` $\in \mathcal{O}(n)$

Übung 17. Bei den Operationen `erase` und `decreaseKey` nützt die Sortierung anhand der Schlüssel nichts, da die eigentlichen Werte gesucht werden müssen. Wie würden sich die Laufzeiten der Operationen `erase` und `decreaseKey` ändern, wenn die Operationen als weiteren Parameter einen Zeiger auf das entsprechende Objekt hätten?

Array-basierte verkettete Listen

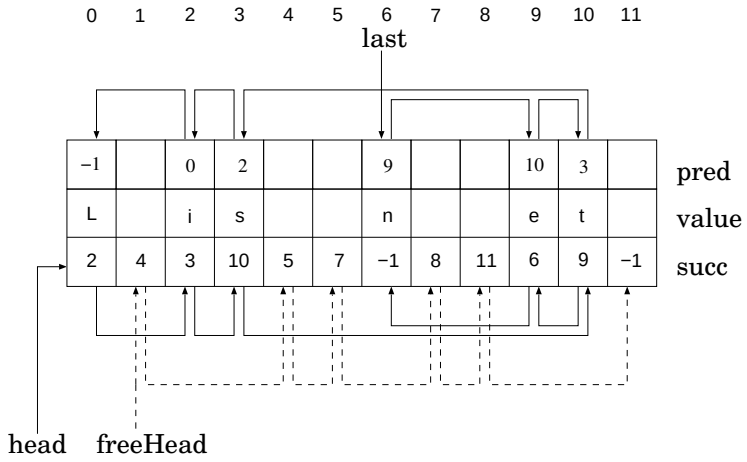
Liegt die maximale Größe der Liste fest, kann eine statische Allokierung vorteilhaft sein. Anstatt Zeiger auf das nächste Datenelement zu verwalten, werden die belegten und freien Elemente in einer Index-Liste verwaltet: `head` und `freeHead` sind keine Zeiger sondern Indizes, also Werte vom Typ `size_t` bzw. `int`.

einfach verkettet:



Auch die freien, nicht belegten Plätze werden als Liste verwaltet.

doppelt verkettet



Auch die freien, nicht belegten Plätze werden als Liste verwaltet.

Array-basierte verkettete Listen

Array-basierte verkettete Listen vs. „normale“ Arrays:

- höherer Speicherbedarf durch Verweise
- sortiertes Einfügen oder Löschen ist effizient möglich

Array-basierte verkettete Listen vs. „normale“ verkettete Listen:

- Begrenzte Anzahl der Elemente
- vermeidet Speicherfragmentierung
- geringere Laufzeit, falls oft Änderungen an den Daten vorgenommen werden, weil die Zeit für das Allokieren (`malloc`) und Freigeben (`free`) von Speicher entfällt

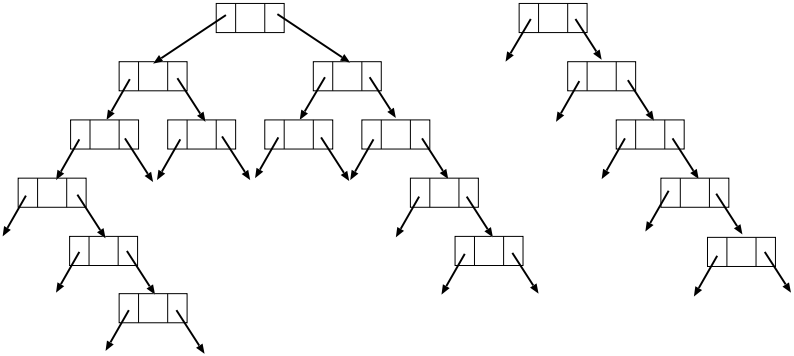
Eine Array-basierte verkettete Liste wird auch Bounded-Linked-List genannt, wohingegen eine „normale“ verkettete Liste auch Unbounded-Linked-List heißt.

Übung 18. Implementieren Sie eine Array-basierte verkettete Liste. Die Größe der Liste soll als Parameter der Funktion `create` übergeben werden.

Übersicht

Bäume sind *verzweigte Datenstrukturen*: Jedes Element kann mehr als einen Nachfolger haben. Im Gegensatz dazu nennt man verkettete Listen auch lineare Datenstruktur.

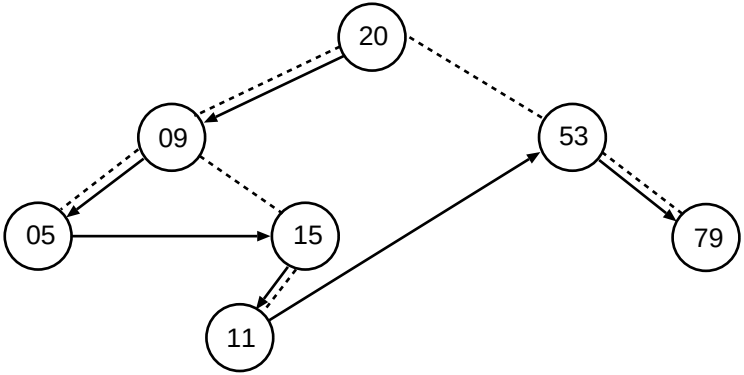
Binärbaum:



Das Element ohne Vorgänger nennt man Wurzel des Baums, die Elemente ohne Nachfolger nennt man Blätter. In der Informatik werden Bäume in der Regel so dargestellt, dass die Wurzel oben ist.

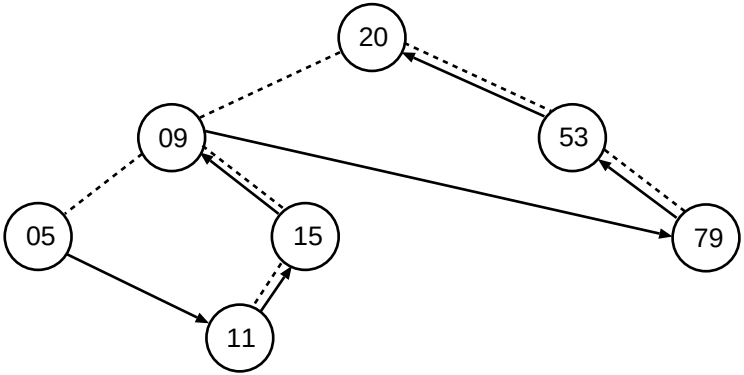
Preorder-Traversierung (auch Hauptreihenfolge genannt)

1. Besuchen des Wurzelknotens
2. Preorder-Traversierung des linken Teilbaums
3. Preorder-Traversierung des rechten Teilbaums



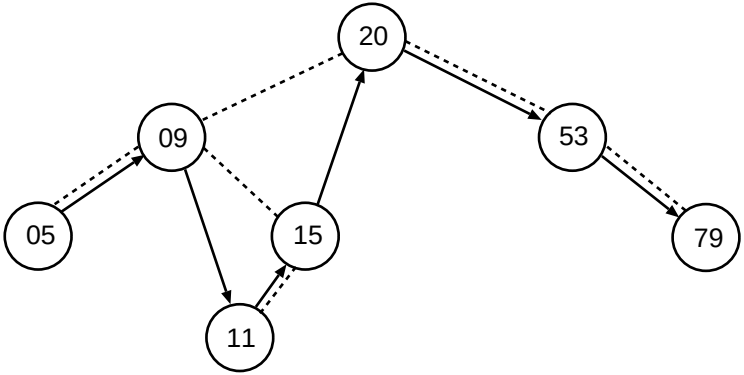
Postorder-Traversierung (auch Nebenreihenfolge genannt)

1. Postorder-Traversierung des linken Teilbaums
2. Postorder-Traversierung des rechten Teilbaums
3. Besuchen des Wurzelknotens



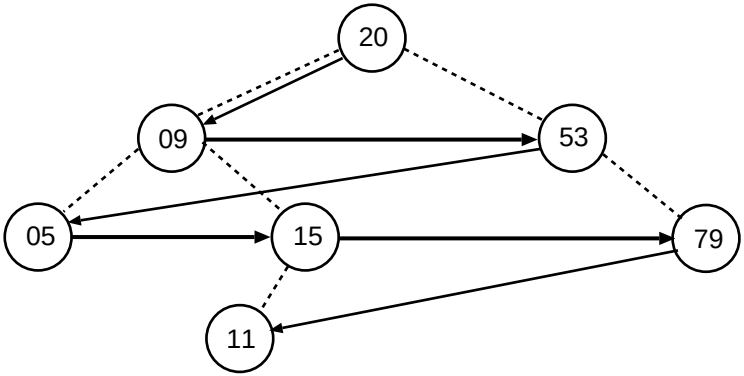
Inorder-Traversierung (auch symmetrische Reihenfolge genannt)

1. Inorder-Traversierung des linken Teilbaums
2. Besuchen des Wurzelknotens
3. Inorder-Traversierung des rechten Teilbaums



Level-Order-Traversierung

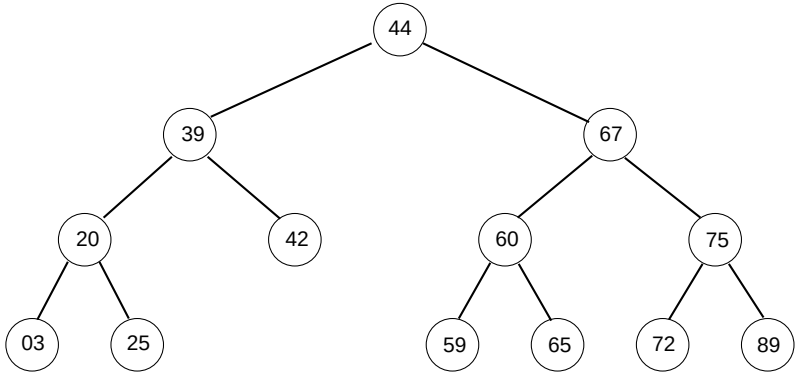
1. Besuchen des Wurzelknotens
2. Durchlaufe jede weitere Ebene von links nach rechts



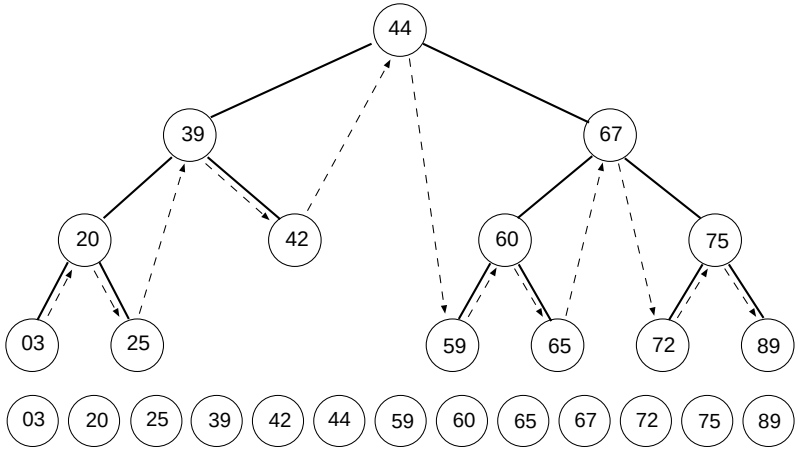
Die Knoten eines Baumes sind oft nach einer bestimmten Ordnungsregel angeordnet.

- Wert des linken Nachfolgers ist kleiner als der Wert des Knotens.
- Wert des rechten Nachfolgers ist größer als oder gleich dem Wert des Knotens.

Solche Bäume nennen wir *Suchbäume*.



Inorder-Traversierung bei Suchbäumen liefert eine sortierte Folge:



Binäre Suchbäume sind links-rechts-geordnete binäre Bäume, d.h. der linke und der rechte Nachfolger sind unterscheidbar.

Wir implementieren die Knoten eines Binärbaums wie folgt:

- Das Attribut `parent` ist ein Verweis auf den Eltern-Knoten.
- Das Attribut `left` zeigt auf das linke Kind, `right` auf das rechte.
- Der Wert `key` enthält den Schlüssel, der in der Regel eine natürliche Zahl ist.
- Das Attribut `value` speichert den eigentlichen Datensatz.

Diese „doppelte Verkettung“ nach „oben“ und „unten“ vereinfacht das Entfernen von Knoten, weil man dadurch immer Zugriff auf den Vorgänger hat.

Suchbäume unterstützen folgende Operationen:

- Einfügen (Insert)
- Entfernen (Delete)
- Suchen (Search)
- Minimum bestimmen: Das Minimum befindet sich links unten im Baum.
- Maximum bestimmen: Das Maximum befindet sich rechts unten im Baum.
- direkten Vorgänger (Predecessor) eines Elements bestimmen
- direkten Nachfolger (Successor) eines Elements bestimmen

Die Laufzeit aller Operationen ist proportional zur Höhe des Baums, also der maximalen Anzahl der Schritte von der Wurzel zu einem Blatt. Ein Baum der Höhe 0 hat also genau einen Knoten, die Wurzel.

Hinweis: Die Höhe bzw. Tiefe eines Baum wird in der Literatur manchmal auch als Anzahl der Ebenen eines Baum definiert. Ein Baum der Höhe t hat nach unserer Definition $t + 1$ Ebenen.

Suchen

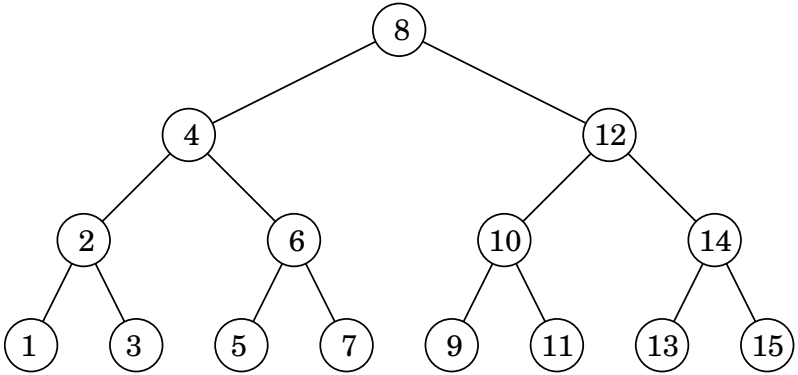
- *Gegeben:* ein Suchbaum T und ein Schlüssel k
- *Gesucht:* ein Knoten x mit Schlüssel k

Algorithmus:

```
function TreeSearch(x: Knoten, k: int) : Knoten
    if (x = NIL) or (x.key = k) then
        return x
    if k < x.key then
        return TreeSearch(x.left, k)
    return TreeSearch(x.right, k)
```

Initial wird TreeSearch mit der Wurzel des Baums aufgerufen.

Falls kein Knoten mit Schlüssel k enthalten ist, soll das Ergebnis der Suche NIL sein. Gibt es mehrere Knoten mit Schlüssel k, so wird einer dieser Knoten berichtet.


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

Direkten Vorgänger bestimmen:

- *Gegeben:* ein Suchbaum T und ein Schlüssel k
- *Gesucht:* der Knoten mit nächst kleinerem Schlüssel

Wir gehen hier davon aus, dass kein Schlüssel mehrfach vorkommt.

Algorithmus:

- Zunächst müssen wir einen Knoten x mit Schlüssel k mittels TreeSearch finden.
- Wenn der gefundene Knoten x einen linken Teilbaum besitzt, dann ist der Vorgänger der maximale Wert des linken Teilbaums.
- Sonst muss der Weg zur Wurzel solange durchlaufen werden, bis der erste Schlüssel gefunden wird, der kleiner als k ist.

  Wird kein solcher Schlüssel gefunden, dann ist k der kleinste Schlüssel im Baum und es gibt keinen Vorgänger.

Einfügen eines Elements

Beim Einfügen eines Wertes `val` mit Schlüssel `key` in den Baum muss die Suchbaum-Eigenschaft erhalten bleiben. Wir gehen daher wie folgt rekursiv vor:

- Ist der Teilbaum leer, so wird ein neuer Knoten mit Wert `val` und Schlüssel `key` angelegt und als Wurzel des Teilbaums gespeichert.
- Andernfalls muss der neue Knoten entweder im linken oder im rechten Teilbaum der Wurzel x eingefügt werden.
  - Falls `key` kleiner ist als x.key, füge den neuen Knoten im linken Teilbaum x.left ein.
  - Sonst füge den neuen Knoten im rechten Teilbaum x.right ein.

Löschen eines Elements

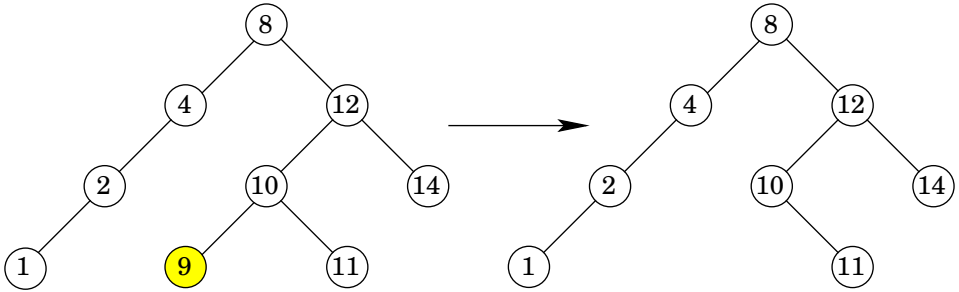
Wir unterscheiden drei Fälle, die auf den nächsten Seiten visualisiert sind: Wenn der zu entfernende Knoten

- ein Blatt ist, dann kann das Blatt einfach gelöscht werden.

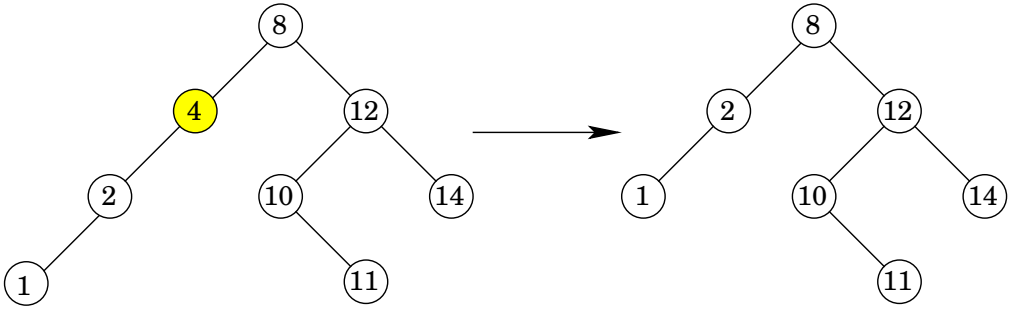
  Achtung: Beim Elternteil e das Löschen des entsprechenden Nachfolgers `e.left` bzw. `e.right` nicht vergessen!
- nur ein Kind besitzt, dann kann der Knoten durch dieses Kind ersetzt werden.

  Achtung: Anpassen des `parent`-Eintrags beim Kind nicht vergessen!
- zwei Kinder besitzt, suchen wir zunächst den direkten Nachfolger des Knotens. Der direkte Nachfolger besitzt kein linkes Kind. Wir ersetzen den Knoten durch seinen direkten Nachfolger und entfernen den direkten Nachfolger wie oben aus dem Baum.

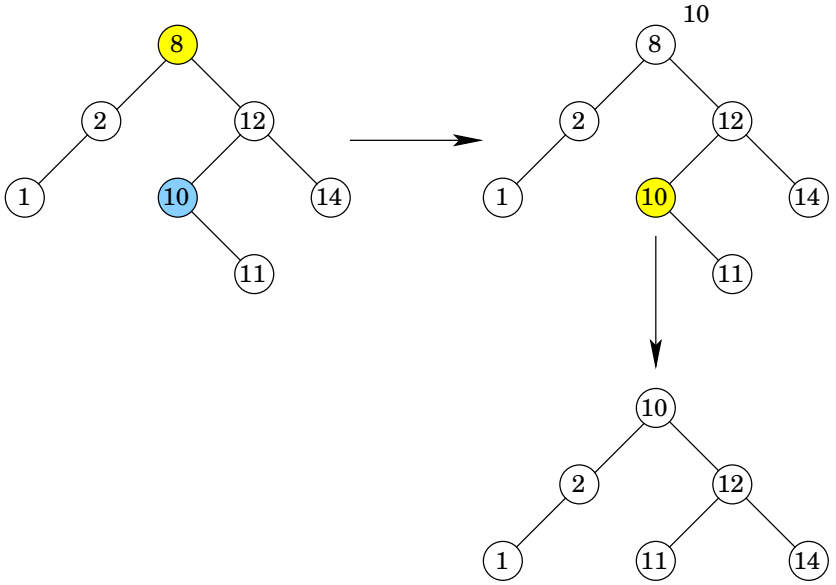
Fall 1: Der zu löschende Knoten ist ein Blatt



Fall 2: Der zu löschende Knoten hat nur ein Kind.



Fall 3: Der zu löschende Knoten hat zwei Kinder.



tree.h

```
#ifndef _TREE_H
#define _TREE_H

// some error flags ----------------------------------------
extern char OKAY;
extern char NOT_FOUND;
extern char EMPTY_TREE;

// incomplete data types (forward declaration) -------------
typedef struct node node_t;
typedef struct tree tree_t;

// interface -----------------------------------------------
int getKey(node_t *n);
int getVal(node_t *n);
char getError(tree_t *t);

tree_t * createTree(void);
void destroyTree(tree_t *t);
```

```
void preOrder(tree_t *);
void postOrder(tree_t *);
void inOrder(tree_t *);
void levelOrder(tree_t *);

node_t * findTree(tree_t *t, int key);
node_t * min(tree_t *t);
node_t * max(tree_t *t);
node_t * pred(tree_t *t, int key);
node_t * succ(tree_t *t, int key);
void insertTree(tree_t *t, int k, int v);
void eraseTree(tree_t *t, int k);

char emptyTree(tree_t *t);
size_t sizeTree(tree_t *t);

#endif
```

tree.c

```
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

char OKAY = 0;
char NOT_FOUND = 1;
char EMPTY_TREE = 2;

struct node {
    int key;
    int value;
    node_t *left, *right, *parent;
};

struct tree {
    node_t *root;
    size_t cnt;
    char error;
};
```

```
int getKey(node_t *n) {
    return n->key;
}

int getVal(node_t *n) {
    return n->value;
}

char getError(tree_t *t) {
    return t->error;
}

char emptyTree(tree_t *t) {
    return t->cnt == 0;
}

size_t sizeTree(tree_t *t) {
    return t->cnt;
}
```

```
tree_t * createTree(void) {
    tree_t *t = (tree_t *) malloc(sizeof(tree_t));

    t->root = NULL;
    t->cnt = 0;
    return t;
}

// static here means private, not available outside
static node_t * createNode(node_t *parent, int k, int v) {
    node_t *n = (node_t *) malloc(sizeof(node_t));

    n->key = k;
    n->value = v;
    n->left = NULL;
    n->right = NULL;
    n->parent = parent;
    return n;
}
```

```
// static here means private, not available outside
// destroyNode is *not* given in header file
static void destroyNode(node_t *n) {
    if (n == NULL)
        return;

    destroyNode(n->left);
    destroyNode(n->right);
    free(n);
}

void destroyTree(tree_t *t) {
    if (t == NULL)
        return;

    destroyNode(t->root);
    free(t);
}
```

```
// static here means private, not available outside
// preOrderNode is *not* given in header file
static void preOrderNode(node_t *n) {
    if (n == NULL)
        return;

    printf("(%d, %d)  ", n->key, n->value);
    preOrderNode(n->left);
    preOrderNode(n->right);
}

void preOrder(tree_t *t) {
    if (t == NULL) {
        printf("--- leer ---\n");
        return;
    }
    preOrderNode(t->root);
    printf("\n");
}
```

```
// static here means private, not available outside
static void postOrderNode(node_t *n) {
    if (n == NULL)
        return;

    postOrderNode(n->left);
    postOrderNode(n->right);
    printf("(%d, %d)  ", n->key, n->value);
}

void postOrder(tree_t *t) {
    if (t == NULL) {
        printf("--- leer ---\n");
        return;
    }
    postOrderNode(t->root);
    printf("\n");
}
```

```
// static here means private, not available outside
static void inOrderNode(node_t *n) {
    if (n == NULL)
        return;

    inOrderNode(n->left);
    printf("(%d, %d)  ", n->key, n->value);
    inOrderNode(n->right);
}

void inOrder(tree_t *t) {
    if (t == NULL) {
        printf("--- leer ---\n");
        return;
    }
    inOrderNode(t->root);
    printf("\n");
}
```

Übung 19. Wie kann eine Level-Order-Traversierung implementiert werden?

```
node_t * findTree(tree_t *t, int key) {
    if (t == NULL || t->root == NULL) {
        t->error = NOT_FOUND;
        return NULL;
    }

    node_t *n = t->root;
    while (n != NULL && n->key != key)
        if (key < n->key)
            n = n->left;
        else n = n->right;

    if (n == NULL)
        t->error = NOT_FOUND;
    return n;
}
```

```
// static here means private, not available outside
static node_t * getMin(node_t *n) {
    while (n->left != NULL)
        n = n->left;
    return n;
}

node_t * min(tree_t *t) {
    if (t == NULL || t->root == NULL) {
        t->error = NOT_FOUND;
        return NULL;
    }

    node_t *n = t->root;
    return getMin(n);
}
```

```
static node_t * getMax(node_t *n) {  // private
    while (n->right != NULL)
        n = n->right;
    return n;
}

node_t * max(tree_t *t) {
    if (t == NULL || t->root == NULL) {
        t->error = NOT_FOUND;
        return NULL;
    }

    node_t *n = t->root;
    return getMax(n);
}
```

```
node_t * pred(tree_t *t, int key) {
    node_t *n = findTree(t, key);

    if (n == NULL) {
        t->error = NOT_FOUND;
        return NULL;
    }

    if (n->left == NULL) {
        // walk up the tree
        n = n->parent;
        while (n != NULL && n->key >= key)
            n = n->parent;
        return n;
    }
    return getMax(n->left);
}
```

```
node_t * succ(tree_t *t, int key) {
    node_t *n = findTree(t, key);

    if (n == NULL) {
        t->error = NOT_FOUND;
        return NULL;
    }

    if (n->right == NULL) {
        // walk up the tree
        n = n->parent;
        while (n != NULL && n->key <= key)
            n = n->parent;
        return n;
    }
    return getMin(n->right);
}
```

```
// static here means private, not available outside
static void insertNode(node_t *n, int k, int v) {
    if (k < n->key) {
        if (n->left == NULL)
            n->left = createNode(n, k, v);
        else insertNode(n->left, k, v);
    } else {
        if (n->right == NULL)
            n->right = createNode(n, k, v);
        else insertNode(n->right, k, v);
    }
}
```

```
void insertTree(tree_t *t, int k, int v) {
    if (t->root == NULL) {
        t->root = createNode(NULL, k, v);
        t->cnt = 1;
        return;
    }

    node_t *n = findTree(t, k);

    if (n != NULL) {
        n->value = v;  // update !!!
        return;
    }

    insertNode(t->root, k, v);
    t->cnt += 1;
}
```

Übung 20. Implementieren Sie die Funktion `eraseTree`.

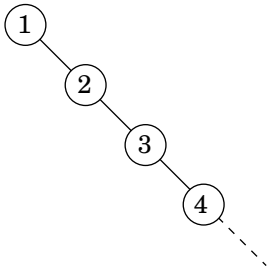
main.c

```c
#include <stdio.h>
#include "tree.h"

int main(void) {
    int key[10] = {4,2,6,1,3,5,7,8,9,0};
    int val[10] = {1,2,3,4,5,6,7,8,9,0};

    tree_t *t = createTree();
    for (int i = 0; i < 10; i++)
        insertTree(t, key[i], val[i]);

    printf("Pre-Order-Traversierung:\n");
    preOrder(t);

    printf("Post-Order-Traversierung:\n");
    postOrder(t);

    printf("In-Order-Traversierung:\n");
    inOrder(t);
```

```
printf("minKey: %d\n", getKey(min(t)));
printf("maxKey: %d\n", getKey(max(t)));

printf("succ(4): %d\n", getKey(succ(t, 4)));
printf("pred(4): %d\n", getKey(pred(t, 4)));
printf("succ(5): %d\n", getKey(succ(t, 5)));
printf("pred(3): %d\n", getKey(pred(t, 3)));

node_t *n = succ(t, 9);
if (n == NULL || getError(t) != OKAY)
    printf("succ(9): not found\n");
else printf("succ(9): %d\n", getKey(n));
```

```
    n = pred(t, 0);
    if (n == NULL || getError(t) != OKAY)
        printf("pred(0): not found\n");
    else printf("pred(0): %d\n", getKey(n));
    printf("size of tree: %lu\n", sizeTree(t));

    printf("erase(3)\n");
    eraseTree(t, 3);
    printf("size of tree: %lu\n", sizeTree(t));
    inOrder(t);

    printf("erase(4)\n");
    eraseTree(t, 4);
    printf("size of tree: %lu\n", sizeTree(t));
    inOrder(t);

    destroyTree(t);
    return 0;
}
```

Durch das Einfügen von aufsteigend sortierten Werten entsteht ein zu einer Liste degenerierter Baum.



Beim Einfügen eines Wertes muss also immer bis ans Ende der Liste gelaufen werden. Als Laufzeit für das Einfügen von n Werten erhalten wir daher:

$$T(n) = \sum_{i=1}^{n} i \quad \rightarrow \quad T \in \Theta(n^2)$$

Das Einfügen von zufälligen Werten geht deutlich schneller. Beim erfolgreichen Suchen eines Wertes werden so viele Vergleiche benötigt, wie der gesuchte Knoten von der Wurzel entfernt ist.

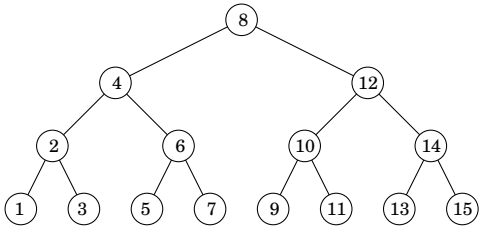
Die interne Pfadlänge (internal path length) eines Baums t ist:

$$I(t) = \sum_{v \in t} \text{depth}(v) = \sum_{v \in t} \text{ Distanz von Knoten } v \text{ zur Wurzel}$$

Beispiel: Interne Pfadlänge eines balancierten, vollständigen Baums.

Für den rechts dargestellten Baum gilt:

- Knoten 8 hat Tiefe 0.
- Knoten 4 und 12 haben Tiefe 1.
- Knoten 2, 6, 10 und 14 haben Tiefe 2.
- Die restlichen Knoten haben Tiefe 3.



Allgemein gilt für einen balancierten, vollständigen Baum t der Tiefe k, also einen Baum mit $k+1$ Ebenen:

$$I(t) \;=\; 1\cdot 0 + 2\cdot 1 + 4\cdot 2 + 8\cdot 3 + \ldots \;=\; \sum_{i=1}^{k} 2^i \cdot i \;=\; (k-1)\cdot 2^{k+1} + 2$$

Das der letzte Teil auch wirklich gilt, zeigen wir auf der nächsten Seite mittels vollständiger Induktion.

Induktionsanfang $k = 1$:

$$\sum_{i=1}^{k} 2^i \cdot i \;=\; 2^1 \cdot 1 \;=\; 2 \;\stackrel{!}{=}\; (k-1) \cdot 2^{k+1} + 2 \;=\; 0 \cdot 2^2 + 2 \;=\; 2 \;\checkmark$$

Induktionsschluss $k \rightsquigarrow k+1$:

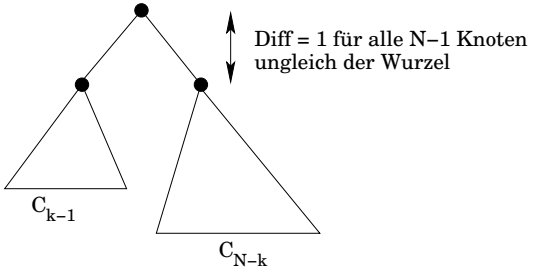
$$\begin{aligned} \sum_{i=1}^{k+1} 2^i \cdot i &= \left(\sum_{i=1}^{k} 2^i \cdot i \right) + 2^{k+1} \cdot (k+1) \\ &\stackrel{I.V.}{=} \left((k-1) \cdot 2^{k+1} + 2 \right) + 2^{k+1} \cdot (k+1) \\ &= 2k \cdot 2^{k+1} + 2 \;=\; k \cdot 2^{k+2} + 2 \;\checkmark \end{aligned}$$

Bei $n = 2^{k+1} - 1 \iff \log_2(n+1) = k+1$ gilt: interne Pfadlänge $I(t) \in \mathcal{O}(n \cdot \log(n))$

durchschnittliche Anzahl benötigter Vergleiche pro Insert-Operation: interne Pfadlänge dividiert durch die Anzahl der Knoten n im Baum. Hier: $I(t)/n \in \mathcal{O}(\log(n))$

Sei C_N die durchschnittliche interne Pfadlänge eines binären Suchbaums mit N Knoten. Dann gilt:

$$\begin{aligned} C_1 &= 1 \\ C_N &= N-1 \\ &+ \frac{1}{N} \sum_{1 \le k \le N} (C_{k-1} + C_{N-k}) \end{aligned}$$



Analog zu der Rekursionsformel von Quicksort (Kapitel 3.3) gilt:

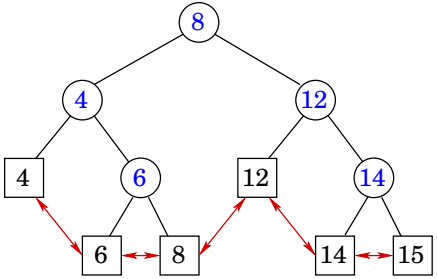
$$C_N \in \Theta(N \log N)$$

Dividieren wir durch N so erhalten wir die durchschnittliche Anzahl Vergleiche bei einer erfolgreichen Suche: $T_{search} \in \Theta(\log N)$

Die Analyse einer erfolglosen Suche ist komplizierter, liefert aber dieselbe asymptotische Laufzeitabschätzung.

Blattsuchbäume: Die Werte werden nur in den Blättern des Baums gespeichert. Innere Knoten enthalten nur „Wegweiser“ für die Suche, und zwar den maximalen Wert des linken Teilbaums.

Werden alle Blätter doppelt verkettet, dann können Bereichsanfragen der Form „Welche Werte liegen zwischen k_1 und k_2?“ schnell beantwortet werden.

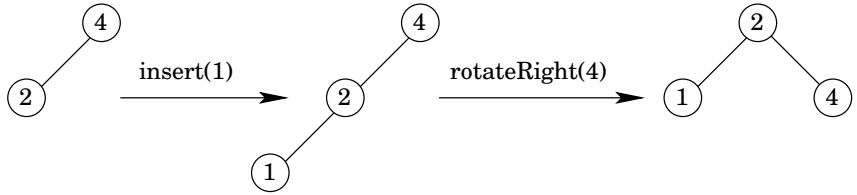


Beispiel: $k_1 = 5, k_2 = 13$

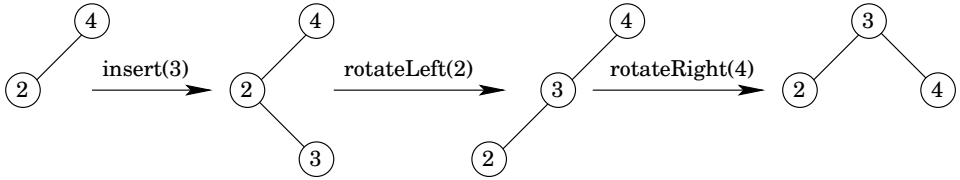
- Suche den Wert k_1: Die Suche endet im Knoten mit dem Wert 6.
- Laufe die verkettete Liste entlang, bis ein Wert $\geq k_2$ erreicht wird, und gib alle Werte auf diesem Weg aus.

Übung 21. Beschreiben Sie, wie Knoten in einen Blattsuchbaum eingefügt bzw. aus einem solchen Baum gelöscht werden können. Wie kann bei einem Suchbaum eine Bereichsanfrage effizient implementiert werden?

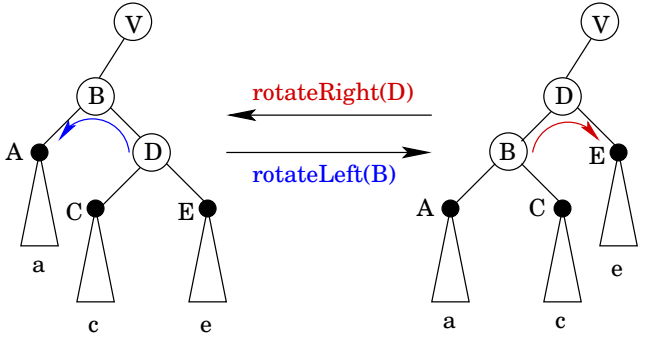
Um zu vermeiden, dass die Höhe eines Baums zu groß wird, werden beim Einfügen und Löschen ggf. sogenannte Rotationen ausgeführt.



Je nachdem, ob der neue Knoten als linker oder rechter Nachfolger eingefügt wird, ist zuvor noch eine gegensätzliche Rotation notwendig:



Rotationen können in konstanter Zeit ausgeführt werden, da nur einige Zeiger „verbogen“ werden müssen. *Beispiel:* rotateLeft(B)




```
V->left = D;
B->parent = D;
D->parent = V;
B->right = D->left;
D->left = B;
C->parent = B;
```

Falls `B` der rechte Nachfolger von `V` war, muss entsprechend `V->right = D` gesetzt werden.

Der Baum „oberhalb“ von `B` (im linken Teil) bleibt unverändert.

Balancierte Suchbäume wie AVL- oder Rot-Schwarz-Bäume nutzen Rotationen. Sie garantieren eine Worst-Case-Laufzeit von $\Theta(\log(n))$ für jede Einfüge-, Such- und Lösche-Operation.

Die Datenstrukturen `std::set`[29] und `std::map`[30] in C++ sind mittels Rot-Schwarz-Bäumen implementiert:

- `std::set` is an associative container that contains a sorted set of unique objects of type Key. [...] Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees.
- `std::map` is a sorted associative container that contains key-value pairs with unique keys. [...] Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees

[29] `https://en.cppreference.com/w/cpp/container/set`

[30] `https://en.cppreference.com/w/cpp/container/map`

Ausgleichen eines Baums

Naiver, rekursiver Algorithmus zum Ausgleichen eines Baums:

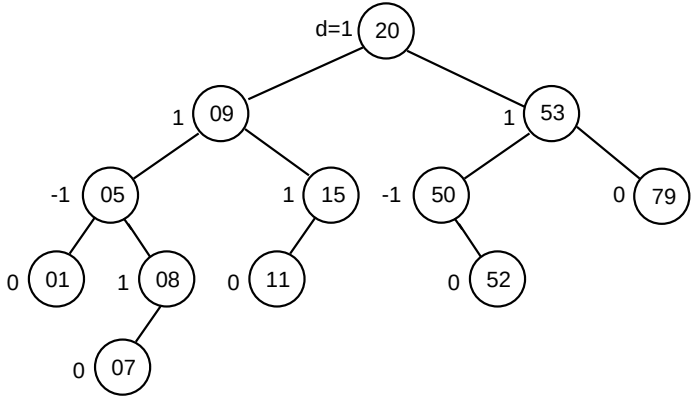
- Finde das in der Ordnung mittlere Element eines Baums.
- Speichere das Element als Wurzel.
- Verfahre auf gleiche Art mit den Teilbäumen.

Sehr aufwendiges Verfahren, dass nach jedem Löschen bzw. Hinzufügen eines Knotens wiederholt werden müsste.

Schwächere Definition von Ausgeglichenheit nach Adelson-Velski und Landis (1962)
$\Rightarrow$ AVL-Bäume.

In einem AVL-Baum unterscheiden sich die Höhen d des linken und des rechten Teilbaums für jeden Knoten höchstens um eins. $\rightarrow d \in \{-1, 0, 1\}$

Es ist günstig, die Differenz der Höhen der Teilbäume "links - rechts" bei jedem Knoten zu speichern.



Einfügen und Löschen von Knoten erfolgt zunächst wie in unbalancierten Suchbäumen.

Minimale Anzahl $n(t)$ von Knoten in einem Baum der Höhe t:

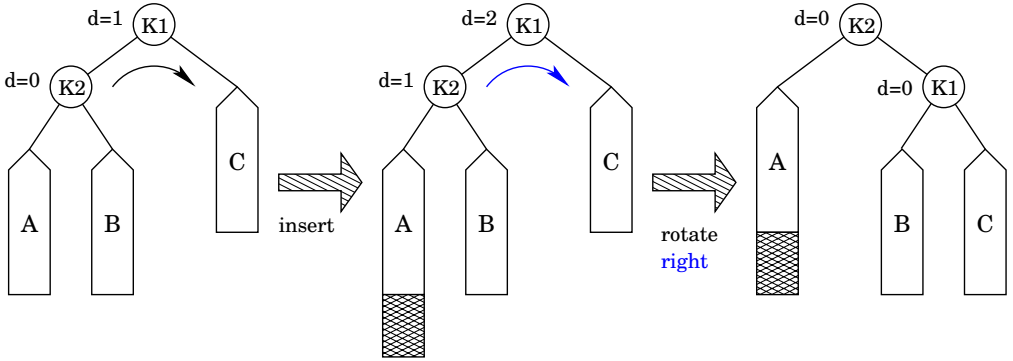
- Ein minimaler AVL-Baum der Höhe $t > 1$ setzt sich aus einer Wurzel und einem Teilbaum der Höhe $t-1$ sowie einem Teilbaum der Höhe $t-2$ zusammen.
  - $\rightarrow$ $n(t) = 1 + n(t-1) + n(t-2) = fib(t+3) - 1 \approx 1.61803^{t+4}$
  - $\rightarrow$ Die Höhe eines AVL-Baumes mit n Knoten ist $\mathcal{O}(\log(n))$
- Ein Binärbaum der Höhe t hat höchstens $2^{t+1} - 1$ viele Knoten

t	0	1	2	3	4	5	6	7	8	9	10
n(t)	1	2	4	7	12	20	33	54	87	141	228
$2^{t+1} - 1$	1	3	7	15	31	63	127	255	511	1023	2047

Durch Einfügen oder Löschen können Knoten mit $d = 2$ oder $d = -2$ entstehen und somit die AVL-Eigenschaft zerstören. $\rightarrow$ Durch Rotation kann in diesen Fällen die AVL-Bedingung wieder hergestellt werden.

Im folgenden sei K1 ein Knoten im Baum und K2 das jeweilige Kind von K1.

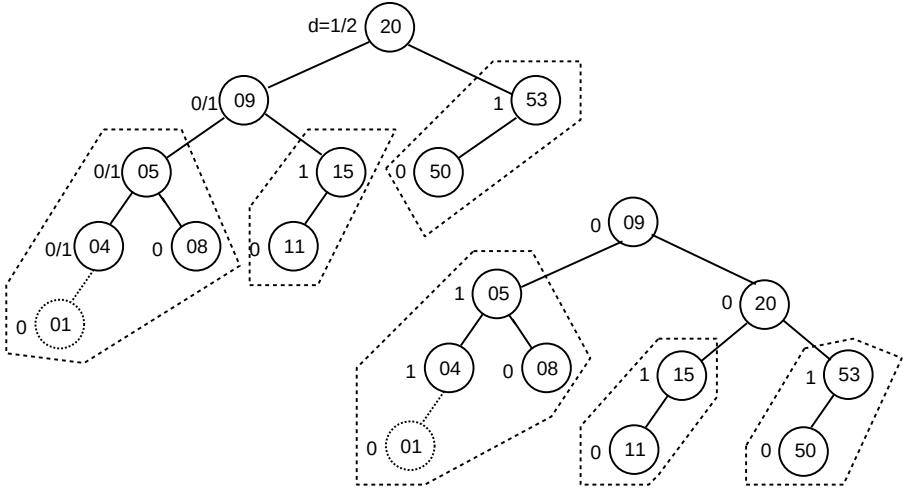
LL-Rotation: Ein Knoten wird links von K1 und links von K2 eingefügt, wobei K1 der Knoten ist, bei dem nach dem Einfügen $d = 2$ auftritt.



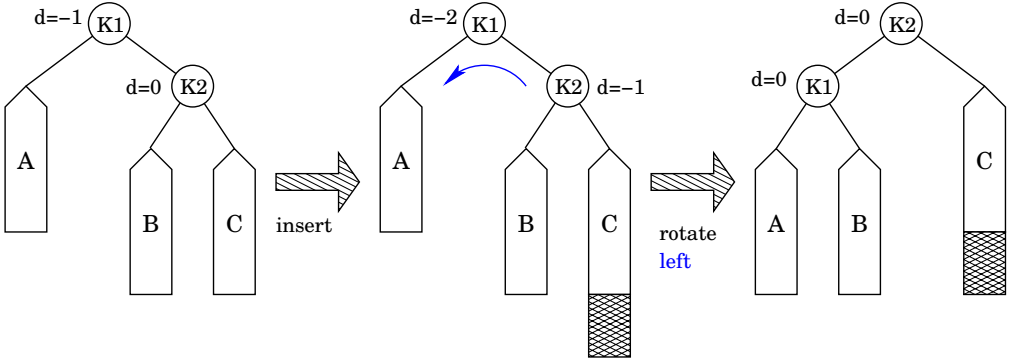

```
K2->parent = K1->parent;
K1->left = K2->right;
```

```
K2->right = K1;
K1->parent = K2;
```

Es müssen noch zwei weitere Zeiger geändert werden. Welche?


d=1/2
20
0/1
09
1
53
0/1
05
1
15
0
50
0/1
04
0
08
0
11
0
01
0
09
1
05
0
20
1
04
0
08
1
15
1
53
0
01
0
11
0
50

RR-Rotation: Wird ein Knoten rechts von K1 und rechts von K2 eingefügt, wobei K1 der Knoten ist, bei dem anschließend $d = -2$ auftritt, so wird eine einfache Rotation analog zur LL-Rotation durchgeführt.

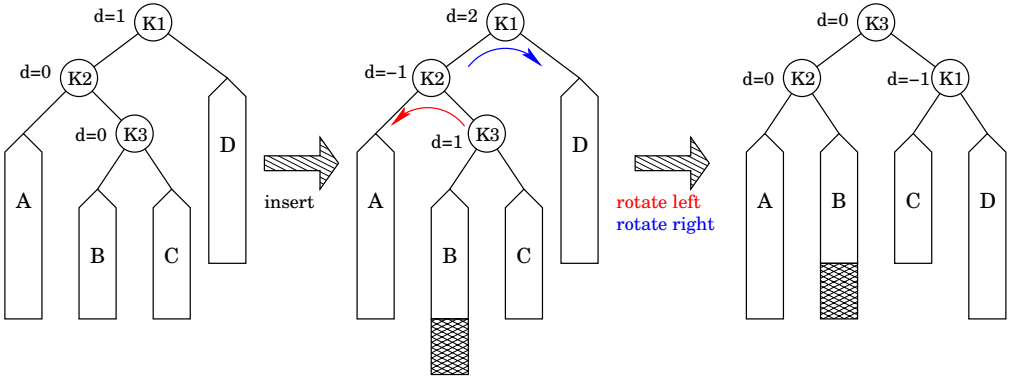



```
K2->parent = K1->parent;
K1->right = K2->left;
```

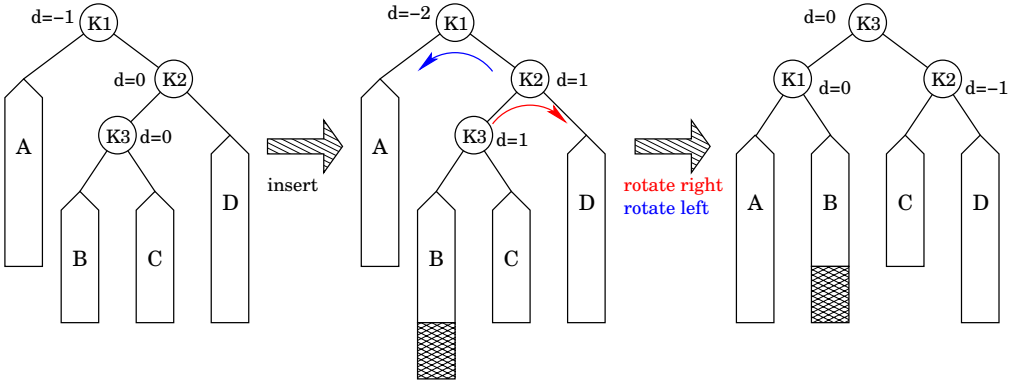
```
K2->left = K1;
K1->parent = K2;
```

Es müssen noch zwei weitere Zeiger geändert werden. Welche?

LR-Rotation: Wird ein Knoten links von K1 und rechts von K2 eingefügt, wobei K1 der Knoten ist, bei dem anschließend $d = 2$ auftritt, so muss eine Doppelrotation durchgeführt werden.



RL-Rotation: Wird ein Knoten rechts von K1 und links von K2 eingefügt, wobei K1 der Knoten ist, bei dem anschließend $d = -2$ auftritt, so muss eine Doppelrotation durchgeführt werden.



Es muss nach dem Einfügen eines Werts maximal eine Rotation bzw. eine Doppelrotation ausgeführt werden, danach ist die AVL-Eigenschaft wieder hergestellt.

Aufwand des Rebalancierens:

- Werte der Höhendifferenz aktualisieren: $\mathcal{O}(\log(n))$, weil sich das Aktualisieren bis zur Wurzel fortsetzen kann.
  Die Balance-Werte werden beim Abbau der Rekursion aktualisiert. Je nachdem, ob der Knoten links oder rechts von einem Knoten n eingefügt wurde, wird der Balance-Wert von n aktualisiert: Links $\rightarrow$ plus 1, rechts $\rightarrow$ minus 1
- Umhängen der Zeiger: $\mathcal{O}(1)$

$\rightarrow$ Der Gesamtaufwand zum Einfügen eines Knotens beträgt $\mathcal{O}(\log(n))$.

Übung 22. Geben Sie einen AVL-Baum an, der mindestens die Tiefe drei hat, also aus mindestens vier Ebenen besteht, und bei dem durch das Einfügen eines Elements das Ausgleichen der Balance-Werte bis zur Wurzel fortgesetzt wird.

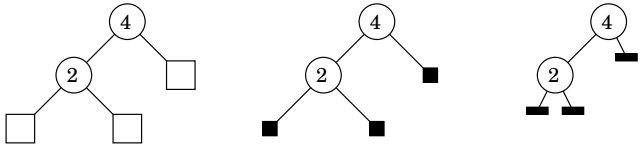
Das Löschen eines Knotens geschieht analog zum Einfügen. Dementsprechend beträgt der Gesamtaufwand zum Löschen eines Knotens ebenfalls nur $\mathcal{O}(\log(n))$.

Wir schauen uns jetzt eine etwas andere Art von Bäumen an, die ähnlich gute Zugriffszeiten erreicht. Splay-Bäume sind *selbstanordnende Suchbäume*: Jeder Schlüssel x, auf den zugegriffen wurde, wird mittels Umstrukturierungen zur Wurzel bewegt.

Strukturanpassung an unterschiedliche Zugriffshäufigkeiten, wobei die Zugriffshäufigkeiten vorher nicht bekannt sind.

- Oft angefragte Schlüssel werden in Richtung Wurzel bewegt.
- Selten angefragte Schlüssel wandern zu den Blättern hinab.

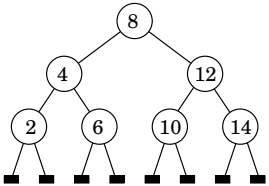
Bei den Splay-Bäumen gehen wir davon aus, dass die Blätter keine Werte speichern. Oder anders gesagt, ist der linke oder rechte Nachfolger NULL, so bezeichnet NULL ein Blatt.



Die Splay-Operation: Sei `T` ein Suchbaum und `x` ein Schlüssel. Dann ist `splay(T,x)` der Suchbaum, den man wie folgt erhält:

- Schritt 1: Suche nach `x` in `T`. Sei `p` der Knoten, bei dem die erfolgreiche Suche endet, falls `x` in `T` vorkommt.

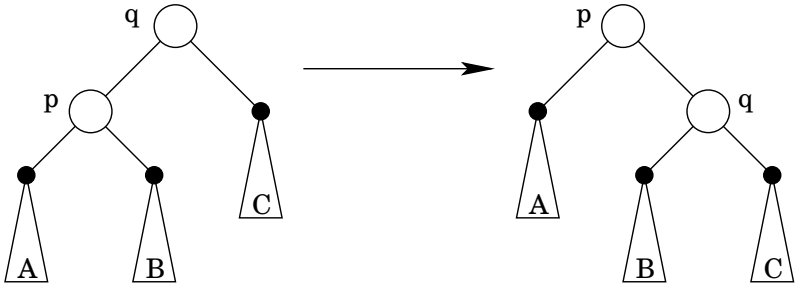
  Ansonsten sei `p` der Vorgänger des Blattes, an dem die Suche nach `x` endet, falls `x` nicht in `T` vorkommt.
- Schritt 2: Wiederhole die folgenden Operationen `zig`, `zig-zig` und `zig-zag` beginnend bei `p` solange, bis sie nicht mehr ausführbar sind, weil `p` Wurzel geworden ist.



Beispiel zu Schritt 1: Wird der Wert 5 gesucht, dann endet die Suche im Blatt links von 6, also ist p der Knoten mit Wert 6. Wird der Wert 11 gesucht, dann endet die Suche rechts von 10, also ist p der Knoten mit Wert 10. Wird der Wert 12 gesucht, dann ist p der Knoten mit dem Wert 12.

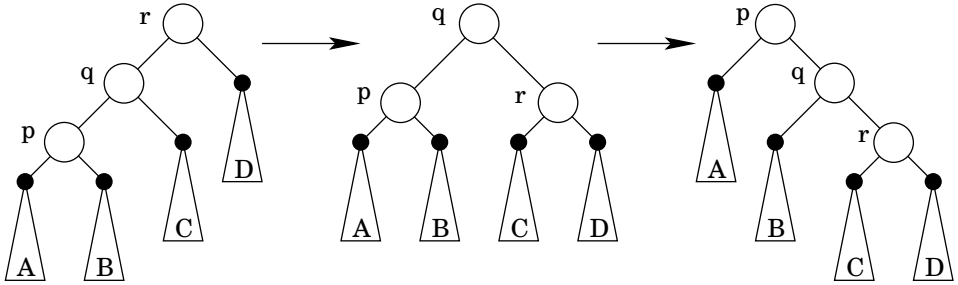
Fall 1: p hat den Vorgänger q und q ist die Wurzel.

Dann führe die Operation zig aus: eine Rotation nach links oder rechts, die p zur Wurzel macht.



Fall 2: p hat den Vorgänger q und den Vor-Vorgänger r. Außerdem sind p und q beides rechte oder beides linke Nachfolger.

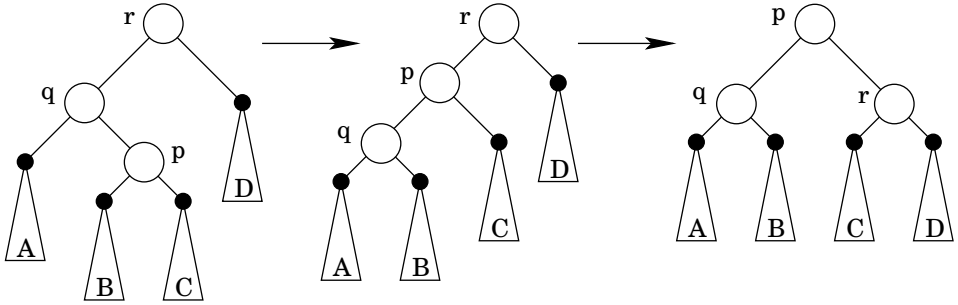
Dann führe die Operation zig-zig aus: zwei aufeinanderfolgende Rotationen in dieselbe Richtung, die p zwei Ebenen hinaufbewegen.



hier: Zunächst eine Rechts-Rotation bei r, dann eine Rechts-Rotation bei q.

Fall 3: p hat Vorgänger q und Vor-Vorgänger r, wobei p linker Nachfolger von q und q rechter Nachfolger von r ist, bzw. p ist rechter Nachfolger von q und q ist linker Nachfolger von r.

Dann führe die Operation zig-zag aus: zwei Rotationen in entgegengesetzter Richtung, die p zwei Ebenen hinaufbewegen.



hier: Zunächst eine Links-Rotation bei q, dann eine Rechts-Rotation bei r.

Anmerkungen:

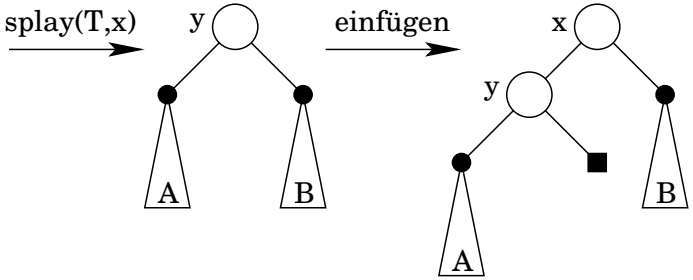
- Kommt `x` in `T` vor, so erzeugt `splay(T,x)` einen Suchbaum, der den Schlüssel `x` in der Wurzel speichert.
- Kommt `x` nicht in `T` vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel `x` unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

Suchen: Um nach `x` in `T` zu suchen wird `splay(T,x)` ausgeführt und danach geprüft, ob `x` jetzt in der Wurzel steht.

Einfügen: Um `x` in `T` einzufügen, rufe zunächst `splay(T,x)` auf. Der Wert `x` wird also *noch nicht* in den Baum eingefügt, das eigentliche Einfügen des Werts erfolgt erst *nach* der Operation `splay(T,x)`, also jetzt!

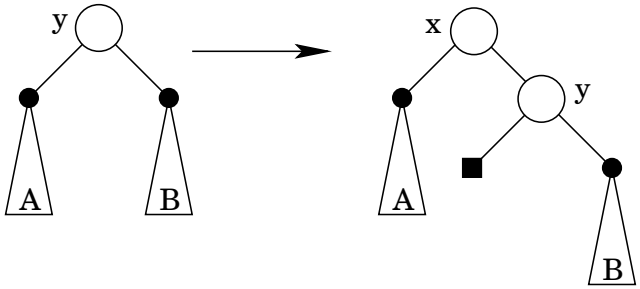
Einfügen: (Fortsetzung) Ist x nicht in der Wurzel, so füge wie folgt eine neue Wurzel mit x ein.

- Falls der Schlüssel der Wurzel von T kleiner als x ist:



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung: Kommt x nicht in T vor, so wird der in der symmetrischen Reihenfolge dem Schlüssel x unmittelbar vorangehende oder unmittelbar folgende Schlüssel zum Schlüssel der Wurzel.

- Falls der Schlüssel der Wurzel von `T` größer als `x` ist:



Dieses Vorgehen ist korrekt aufgrund obiger zweiter Anmerkung.

amortisierte Laufzeit: Das Ausführen einer beliebigen Folge von m Such-, Einfüge- und Entferne-Operationen, in der höchstens n-mal eingefügt wird und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $\mathcal{O}(m \cdot \log(n))$ Schritte.

$\rightarrow$ Jede der m Operationen benötigt amortisiert (also im Schnitt) die Zeit $\mathcal{O}(\log(n))$.

Entfernen: Um `x` aus `T` zu entfernen, rufe `splay(T,x)` auf.

Ist `x` in der Wurzel, dann sei `L` der linke und `R` der rechte Teilbaum unter der Wurzel.

Rufe `splay(L,` ∞`)` auf. Dadurch entsteht ein Teilbaum mit dem größten Schlüssel von `L` an der Wurzel und einem leeren rechten Teilbaum. Ersetze den rechten leeren Teilbaum durch `R`.

Hinweis: Die Operation `splay(T,x)` schließt immer die Suche nach `x` ein.

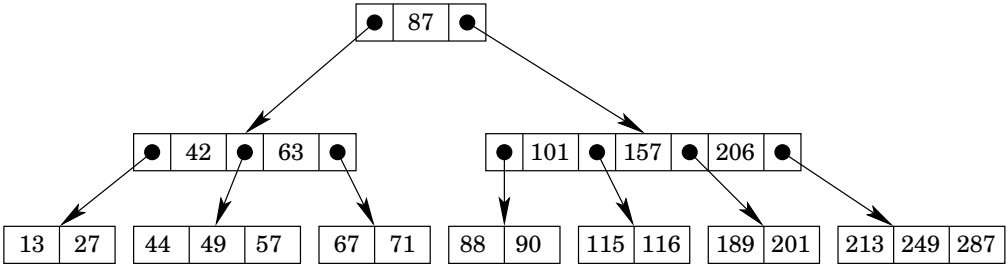
AVL- und Splay-Bäume[31] eignen sich zur Implementierung von Wörterbüchern und Vorrangwarteschlangen:

- Dictionary: `insert` $\in \mathcal{O}(\log(n))$, `erase` $\in \mathcal{O}(\log(n))$, `search` $\in \mathcal{O}(\log(n))$
- Priority-Queue: `insert` $\in \mathcal{O}(\log(n))$, `minimum` $\in \mathcal{O}(\log(n))$, `extractMin` $\in \mathcal{O}(\log(n))$, `decreaseKey` $\in \mathcal{O}(n)$, `erase` $\in \mathcal{O}(n)$

Wie können die Laufzeiten von `decreaseKey` und `erase` verbessert werden?

[31] Bei Splay-Bäumen sind die angegebenen Zeiten amortisierte Laufzeiten.

Wir beschränken uns nicht länger auf binäre Suchbäume.



Jeder innere Knoten eines 2-3-4-Baums hat mindestens 2 und höchstens 4 Kinder.

Der B-Baum wurde 1972 von Rudolf Bayer und Edward M. McCreight entwickelt.
(Quelle: wikipedia)

Ein B-Baum der Ordnung m hat folgende Eigenschaften:

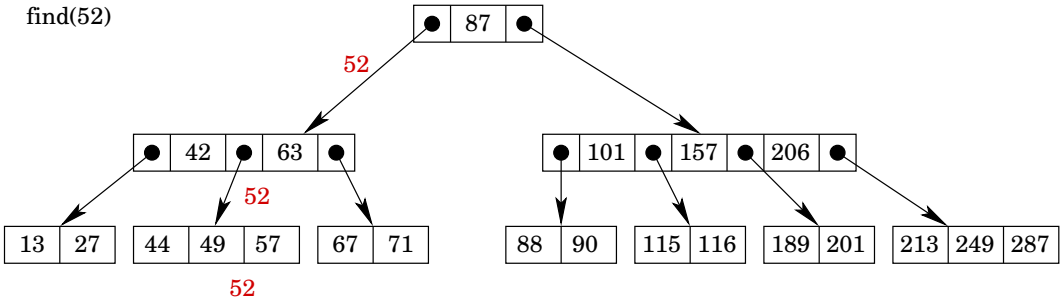
- Alle Blätter befinden sich in gleicher Tiefe.
- Alle inneren Knoten außer der Wurzel haben mindestens $\lceil m/2 \rceil$ Kinder. In einem nicht leeren Baum hat die Wurzel mindestens 2 Kinder.
- Jeder innere Knoten hat höchstens m Kinder. Ein Knoten mit k Kindern speichert $k-1$ Schlüsselwerte.
- Alle Schlüsselwerte eines Knotens sind aufsteigend sortiert.
- Seien $k_1, \ldots, k_s$ die Schlüssel eines inneren Knotens. Dann gibt es die Zeiger $z_0, z_1, \ldots, z_s$ auf die Kinder und es gilt:
  - z_0 zeigt auf einen Teilbaum mit Werten kleiner als k_1.
  - z_i für $i = 1, \ldots, s-1$ zeigt auf einen Teilbaum mit Werten größer als k_i und kleiner als k_{i+1}.
  - z_s zeigt auf einen Teilbaum mit Werten größer als k_s.

Suchen eines Elements:

Die Suche nach einem Schlüssel k in einem B-Baum ist eine natürliche Verallgemeinerung der Suche in einem Suchbaum:

- Man beginnt mit der Wurzel, die die Schlüssel $k_1, \ldots, k_s$ speichert, und bestimmt den kleinsten Index i mit $x \leq k_i$, falls es ein solches i gibt. Ansonsten ist x größer als der größte Schlüssel k_s.
  - Ist $x = k_i$, dann wurde x gefunden.
  - Ist $x < k_i$, so wird die Suche bei z_{i-1} fortgesetzt.
  - Ist $x > k_i$, dann wird die Suche bei z_i fortgesetzt.

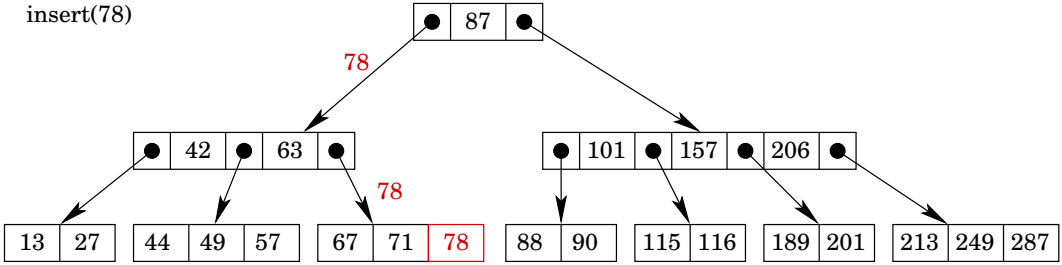
  Dies wird solange wiederholt, bis man x gefunden hat, oder an einem Blatt erfolglos endet.
- Die Suche nach dem kleinsten Index i mit $x \leq k_i$ kann mit linearer oder binärer Suche erfolgen.


find(52)
87
52
42
63
101
157
206
52
13
27
44
49
57
67
71
88
90
115
116
189
201
213
249
287
52

Einfügen eines Wertes:

Bestimme zunächst mittels der Suche das Blatt, in dem der Wert abgelegt werden muss. Wir unterscheiden zwei Fälle:

- Fall 1: Das Blatt hat noch nicht die maximale Anzahl $m-1$ von Schlüsseln gespeichert.
  Dann fügen wir den Wert entsprechend der Sortierung ein.

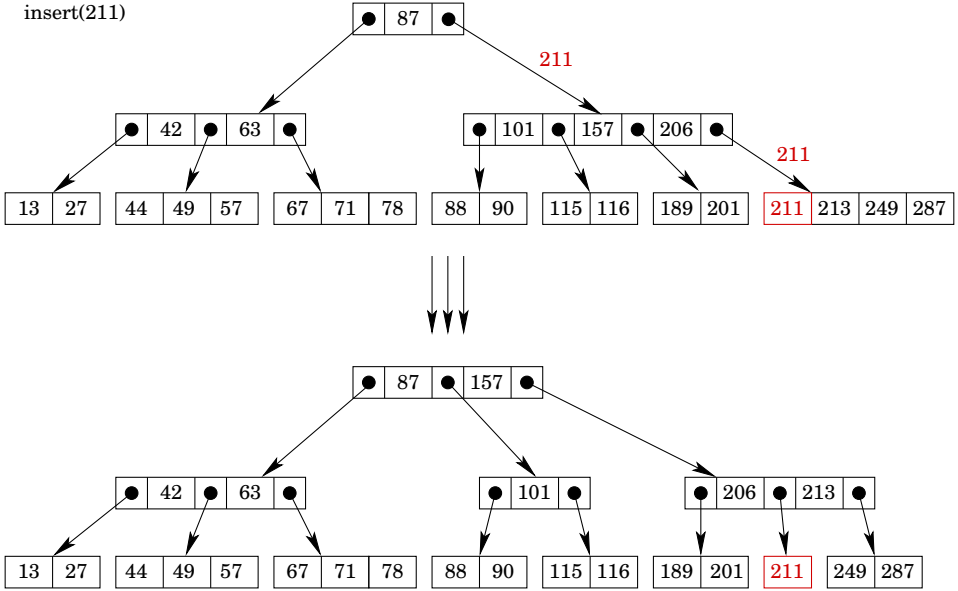


- Fall 2: Das Blatt hat bereits die maximale Anzahl $m - 1$ von Schlüsseln gespeichert.

  In diesem Fall ordnen wir den Wert wie im Fall 1 entsprechend seiner Größe ein und teilen anschließend den zu groß gewordenen Knoten in der Mitte auf. Das mittlere Element wird in den Vorgänger-Knoten eingefügt.

  Dieses Teilen wird solange längs des Suchpfades bis zur Wurzel fortgesetzt, bis ein Knoten erreicht ist, der noch nicht die maximale Anzahl von Schlüsseln speichert, oder bis die Wurzel erreicht wird.

  Muss die Wurzel geteilt werden, so schafft man eine neue Wurzel, die den mittleren Schlüssel als einzigen Schlüssel speichert.


insert(211)
87
211
42
63
101
157
206
211
13
27
44
49
57
67
71
78
88
90
115
116
189
201
211
213
249
287
87
157
42
63
101
206
213
13
27
44
49
57
67
71
78
88
90
115
116
189
201
211
249
287

Ein B-Baum der Ordnung m mit Höhe h hat die minimale Blattanzahl, wenn seine Wurzel nur 2 und jeder innere Knoten nur $\lceil m/2 \rceil$ viele Kinder hat. Daher ist die minimale Blattanzahl

$$N_{\text{min}} = 2 \cdot \lceil m/2 \rceil^{h-1}.$$

Die Blattanzahl wird maximal, wenn jeder innere Knoten die maximal mögliche Anzahl von Kindern hat. Somit ist die maximale Blattanzahl

$$N_{\text{max}} = m^h.$$

Also gilt:

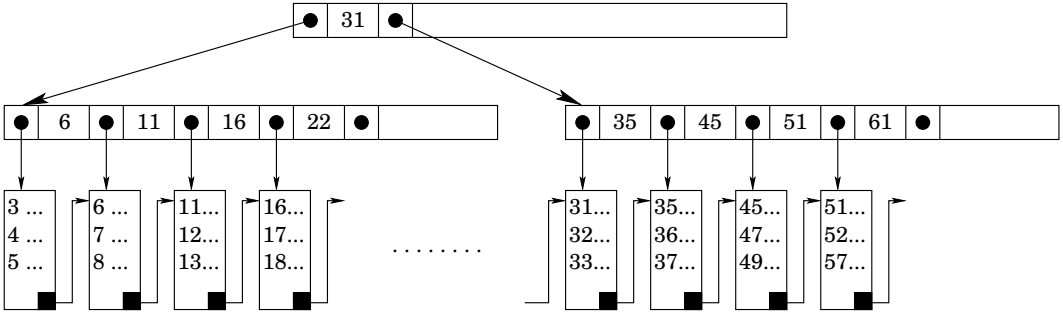
$$N_{\text{min}} = 2 \cdot \lceil m/2 \rceil^{h-1} \;\leq\; n+1 \;\leq\; m^h = N_{\text{max}}$$

und somit $h \leq 1 + \log_{\lceil m/2 \rceil}(\frac{n+1}{2})$ und $\log_m(n+1) \leq h$.

B-Bäume sind also balanciert.

Für den Einsatz in Datenbanksystemen werden die B-Bäume geringfügig modifiziert:

- Die Blätter werden zu einer linearen Liste verkettet.
- Werte werden nur in den Blättern gespeichert. $\rightarrow$ Blattsuchbaum
- Innere Knoten enthalten nur Hinweise: kleinster Schlüsselwert im rechten Teilbaum



B^+-Bäume

B- und B^+-Bäume sind externe Datenstrukturen: Spezielle Datenstrukturen, um große Datenmengen mit wenigen I/O-Zugriffen verwalten zu können.

Große Datenmengen können nicht im Hauptspeicher verwaltet werden, daher müssen Teile der Daten auf einen sekundären oder externen Speicher wie HDD, SSD oder Flash-Medien ausgelagert werden.

Die Seitenersetzungsverfahren wie LRU der Betriebssysteme sowie das Caching und Prefetching sind allgemeine Verfahren, die nicht speziell auf die jeweiligen Probleme abgestimmt sind. Daher werden spezielle Datenstrukturen entwickelt, um auch große Datenmengen effizient verwalten zu können.

Sei b die Anzahl der Schlüssel, die in einem Knoten abgelegt werden können. Bei einer Blockgröße von 4kB lassen sich inklusive der zugehörigen Zeiger etwa $b = 256$ Elemente vom Typ `long int` oder $b = 340$ Elemente vom Typ `int` speichern.

unter Linux:

- `sudo tune2fs -l /dev/sda1` liefert Blockgröße des Dateisystems
- `sudo fdisk -l /dev/sda` liefert Größe der Sektoren

ermitteln der Größen von Datentypen in C++ mittels `sizeof`:

- `sizeof(uintptr_t)` → typisch: 8 Byte
- `sizeof(size_t)` → typisch: 8 Byte
- `sizeof(int)` → typisch: 4 Byte
- `sizeof(long int)` → typisch: 8 Byte

Bei obigen Werten b und einer Blockgröße von 4kB können in einem B-Baum der Höhe h also etwa 256^{h+1} viele `long int`-Werte abgespeichert werden, wobei maximal h viele I/Os nötig sind.

Höhe	ungefähre Anzahl Elemente	
3	$16.384 \cdot 2^{10}$	(16.384 Kilo oder 16 Mega)
4	$4.096 \cdot 2^{20}$	(4.096 Mega oder 4 Giga)
5	$1.024 \cdot 2^{30}$	(1.024 Giga oder 1 Tera)
6	$256 \cdot 2^{40}$	(256 Tera)
7	$65.536 \cdot 2^{40}$	(65.536 Tera oder 64 Peta)
8	$16.384 \cdot 2^{50}$	(16.384 Peta oder 16 Exa)

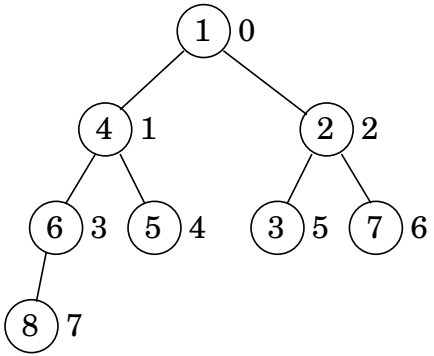
Suchen, Einfügen und Löschen benötigen $\mathcal{O}(\log_b(N))$ viele I/Os.

Bereichsanfragen lassen sich mit B$^+$-Bäumen aufgrund der verketteten Blätter sehr effizient realisieren.

Übersicht

Heap: Eine Folge $F = k_0, \ldots, k_n$ von Schlüsseln, sodass $k_i \leq k_{2i+1}$ und $k_i \leq k_{2(i+1)}$ gilt, sofern $2i + 1 \leq n$ bzw. $2(i + 1) \leq n$. (Wir nutzen hier einen Min-Heap.)

Beispiel: $F = 1, 4, 2, 6, 5, 3, 7, 8$



Heap-Eigenschaft ist erfüllt:

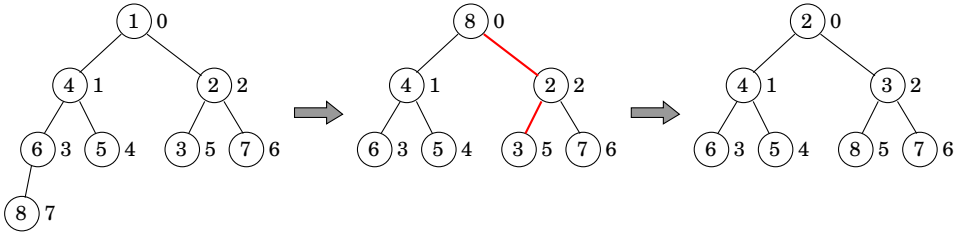
- $k_0 = 1 \leq k_1 = 4$ und $k_0 \leq k_2 = 2$
- $k_1 = 4 \leq k_3 = 6$ und $k_1 \leq k_4 = 5$
- $k_2 = 2 \leq k_5 = 3$ und $k_2 \leq k_6 = 7$
- $k_3 = 6 \leq k_7 = 8$

Der Vorgänger des Elements an Position i befindet sich auf Position $\lfloor (i-1)/2 \rfloor$ im Array.

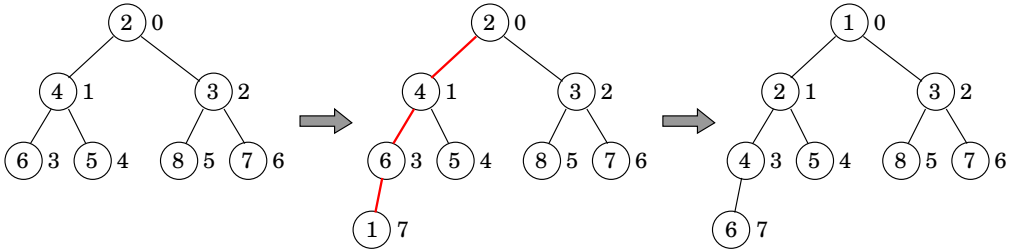
Heaps werden mittels balancierten Binärbäumen oder Arrays implementiert. Wir betrachten nur die Implementierung mittels Arrays.

Ein Heap kann genutzt werden, um den abstrakten Datentyp Vorrangwarteschlange (priority queue) zu implementieren. Diese werden z.B. beim Algorithmus von Prim (minimaler Spannbaum, Kapitel 6.2) und dem Algorithmus von Dijkstra (kürzeste Wege, Kapitel 6.3) benötigt.

- `minimum(H)`: Das Minimum eines Heaps `H` ist das erste Element: k_0
- `extractMin(H)`: Ersetze k_0 durch k_n. Dadurch wird in der Regel die Heap-Eigenschaft verletzt. $\rightarrow$ Lasse k_0 versickern: Tausche das Element mit dem kleineren seiner beiden Nachfolger, solange bis entweder beide Nachfolger größer sind oder das Element unten angekommen ist.



- `insert(H, k, v)`: Füge `v` als letztes Element im Heap `H` ein. Dadurch wird in der Regel die Heap-Eigenschaft verletzt. → Lasse k_n aufsteigen: Tausche das Element mit seinem Vorgänger, solange bis der Vorgänger kleiner ist oder der Anfang erreicht ist.



- `decreaseKey(H, v, k)`: Verkleinere den Schlüssel von `v` auf den Wert `k`. Dadurch wird in der Regel die Heap-Eigenschaft verletzt. → Lasse das Element aufsteigen.
- `erase(H, v)`: Verkleinere den Schlüssel von `v` auf einen sehr kleinen Wert mittels `decreaseKey` und entferne den Wert dann mittels `extractMin`.

Damit die Operationen `decreaseKey` und `erase` nur logarithmische Zeit benötigen, müssen die Positionen der Werte bekannt sein.

Dann eignen sich Heaps zur Implementierung von Vorrangwarteschlangen:

- `insert` $\in \mathcal{O}(\log(n))$
- `minimum` $\in \mathcal{O}(1)$
- `extractMin` $\in \mathcal{O}(\log(n))$
- `decreaseKey` $\in \mathcal{O}(\log(n))$
- `erase` $\in \mathcal{O}(\log(n))$

Übung 23. Implementieren Sie einen Heap in C++ oder C. Das Programm soll zehn zufällige Werte in den Heap einfügen, anschließend mittels einer `while`-Schleife jeweils das kleinste Element ausgeben und entfernen, solange bis der Heap leer ist.

Übersicht

Hashing = Schlüsselsuche durch Berechnung der Array-Indizes!

Idee:

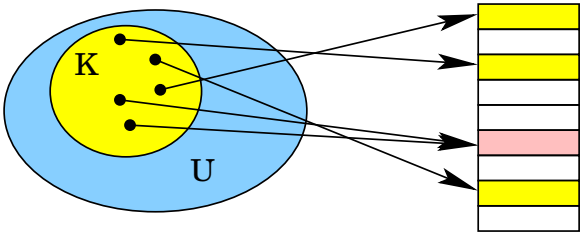
- Notation: Für $n \in \mathbb{N}$ sei $[n] := \{0, 1, \ldots, n-1\}$.
- Bei einer Menge K von Werten aus einem Universum $U = [u]$, also $K \subseteq U$,
- verwende ein Array $A[0 \ldots u-1]$ und speichere die Schlüssel wie folgt:

$$A[k] = \begin{cases} x & \text{falls } x.key \in K \text{ und } x.key = k \\ nil & \text{sonst} \end{cases}$$

$\rightarrow$ suchen, einfügen und löschen in $\mathcal{O}(1)$ Schritten

Probleme: Der Wertebereich $[u]$ kann sehr groß sein. Für 8-stellige Namen ergeben sich ungefähr $26^8 \approx 208 \cdot 10^9$ viele mögliche Werte. Außerdem wird viel Speicherplatz verschwendet, weil in der Regel nur wenige Plätze im Array genutzt werden.

Lösung: Verwende Hash-Funktion h, um das Universum U und insbesondere die Menge der Schlüsselwerte $K \subseteq U$ auf Zahlen in $[m]$, $m \lll u$, abzubilden, also $h : U \to [m]$.



Anmerkungen:

- Die Hash-Funktion ist nicht injektiv, also nicht linkseindeutig, d.h. verschiedene Schlüssel werden auf dieselbe Hashadresse abgebildet. → Adress-Kollision
  - Schlüssel, die auf die gleiche Adresse abgebildet werden, heißen *Synonyme*.
  - Die Menge der Synonyme bezüglich einer Adresse heißt *Kollisionsklasse*.
- Da die Hash-Funktion zum Platzieren und Suchen verwendet wird, muss sie einfach bzw. effizient zu berechnen sein.

Schlüssel, die nicht als Zahlen interpretiert werden können, müssen vorher geeignet umgerechnet werden, z.B. Zeichenketten.

- ASCII-Darstellung: `"i"` $\rightarrow$ 105, `"2"` $\rightarrow$ 50, `"_"` $\rightarrow$ 95, `" "` $\rightarrow$ 32, usw.
- Interpretiere die Zahlen als Ziffern einer Zahl zur Basis 256.
- *Beispiel:* Für die Eingabe `"i2"` $\rightarrow$ 0 105 50 erhalten wir $k = 0 \cdot 256^2 + 105 \cdot 256^1 + 50 \cdot 256^0 = 26930$

Input	k_1	k_2	k_3	k	$h(k) = k \bmod 11$	$h(k) = k \bmod 13$
`" i"`	32	32	105	2105449	5	8
`"i"`	0	0	105	105	6	1
`"i2"`	0	105	50	26930	2	7
`"ii"`	0	105	105	26985	2	10
`"iii"`	105	105	105	6908265	1	0
`"i_2"`	105	95	50	6905650	4	11

Ein Hashverfahren muss zwei Anforderungen genügen:

1. es sollen möglichst wenige Adress-Kollisionen auftreten
   - Hash-Funktion soll die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen, um Adress-Kollisionen zu vermeiden.
   - Häufungen in der Schlüsselverteilung sollen sich nicht auf die Verteilung der Adressen auswirken.
   - Es gilt: Wenn eine Hash-Funktion $\sqrt{\pi n/2}$ Schlüssel auf eine Tabelle der Größe n abbildet, dann gibt es fast sicher eine Kollision. (für $n = 365$ ist $\sqrt{\pi n/2} \approx 23$)
2. die Adress-Kollisionen müssen effizient aufgelöst werden

Division-Rest-Methode: Der Schlüssel wird ganzzahlig durch die Länge der Hashtabelle dividiert. Der Rest wird als Index verwendet: $h(k) = k \bmod m$

- Der Wert m soll keinen kleinen Teiler haben und soll keine Potenz der Basis des Zahlensystems sein!
  - Beispiel: für $m = 2^r$ hängt der Hash-Wert nur von den letzten r Bit ab

$\rightarrow$ Wähle m als Primzahl, die nicht nah an Potenz der Basis des Zahlensystems liegt.

- Beispiel: $m = 761$, aber nicht $m = 509$ (nah an 2^9) oder $m = 997$ (nah an 10^3)

Multiplikative Methode: Sei $m = 2^r$ eine Zweierpotenz. Bei einer Wortgröße w wähle eine Zahl a so, dass $2^{w-1} < a < 2^w$ ist.

$$h(k) = (k \cdot a \bmod 2^w) \textbf{ div } 2^{w-r}$$

Anmerkungen:

- Wähle a nicht zu nah an 2^w
- Modulo-Operation und Rechts-Shift (Integer-Division) ist schnell
- gute Ergebnisse für $a \approx \frac{\sqrt{5}-1}{2} \cdot 2^w$ (goldener Schnitt)

Beispiel: Für $w = 8$, $r = 3$, $a = 191$ und $k = 23$ erhalten wir $h(k) = 1$.

$$\begin{array}{rcl} & 10111111 & 191 \\ * & 00010111 & 23 \\ \hline 10001001 & 01001 & 4393 \end{array}$$

Bewertung: Güte der Hash-Funktion h hängt von der gewählten Schlüsselmenge K ab.

- Güte ist nur unzureichend analysierbar
- zu h lässt sich immer ein K finden mit besonders vielen Kollisionen
- keine Hash-Funktion ist immer besser als alle anderen

Probleme treten auf

- beim Einfügen, wenn die berechnete Hashadresse nicht leer ist
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält

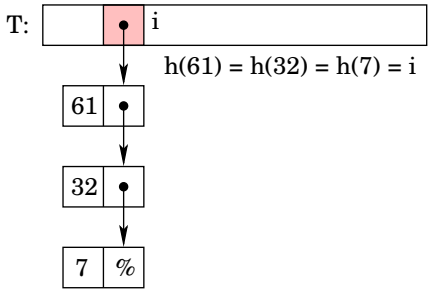
Kollisionsbehandlung für k_p falls $h(k_q) = h(k_p)$ für ein bereits gespeichertes k_q gilt:

- füge k_p in einem separaten Überlaufbereich außerhalb der Hashtabelle ein
  $\rightarrow$ *Verkettung der Überläufer*
- füge k_p an einem freien Platz innerhalb der Hashtabelle ein
  $\rightarrow$ *offenes Hashing*

Hashing: Verkettung der Überläufer

Kollisionsauflösung:

- Überläufer werden in linearer Liste verkettet
- Liste wird an Hashtabelleneintrag angehängt
- Verkettung der Synonyme (Überläufer) pro Kollisionsklasse
- Suchen, Einfügen und Löschen sind auf eine Kollisionsklasse beschränkt



Durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge durch die Hash-Funktion gleichmäßig auf M Listen verteilt werden. *Belegungsfaktor:* $\alpha = N/M$

Kosten von Zugriffen sind abhängig vom Belegungsfaktor.

- $S(\alpha)$: Kosten für erfolgreiche Suche und Löschen (finde Schlüssel: Search)
- $U(\alpha)$: Kosten für erfolglose Suche (entspricht Einfügekosten: Update)

Mittlere Laufzeit bei erfolgreicher Suche:

- beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge $j-1/M$
- spätere Suche nach j-tem Schlüssel betrachtet im Schnitt $1 + j-1/M$ Einträge, wenn stets am Listenende eingefügt wird und keine Elemente gelöscht wurden

$$S(\alpha) = \frac{1}{N}\sum_{j=1}^{N}\left(1+\frac{j-1}{M}\right) = 1 + \frac{1}{NM}\cdot\frac{(N-1)N}{2} = 1 + \frac{N-1}{2M} \approx 1 + \frac{N}{2M} = 1 + \frac{\alpha}{2}$$

Mittlere Laufzeit bei erfolgloser Suche: $U(\alpha) = 1 + N/M = 1 + \alpha$

In der Regel gilt $N \in \mathcal{O}(M)$: Die Größe der Hashtabelle ist ungefähr so groß wie die Anzahl der Datensätze. $\rightarrow \alpha = N/M = \mathcal{O}(M)/M \in \mathcal{O}(1)$

Zum Vergleich: Binäre Suche hat Laufzeit $\mathcal{O}(\log(N))$ plus Aufwand $\mathcal{O}(N \cdot \log(N))$ zum Sortieren der Datensätze.

Problem: Die dynamischen Speicheranforderungen für die Elemente der verketteten Listen sind teuer.

Idee: Speichere Überläufer in der Hashtabelle, nicht in zusätzlichen Listen.

- Ist Hashadresse $h(k)$ belegt, suche systematisch eine Ausweichposition.
- *Sondierungsfolge:* Folge der zu betrachtenden Speicherplätze für einen Schlüssel.
- Die Hash-Funktion hängt nun vom Schlüssel und von der Anzahl durchgeführter Platzierungsversuche ab:

$$h : U \times [m] \to [m]$$

- *wichtig:* Die Sondierungsfolge muss eine Permutation der Zahlen $0, \ldots, m-1$ sein, damit alle Einträge der Hashtabelle genutzt werden können.

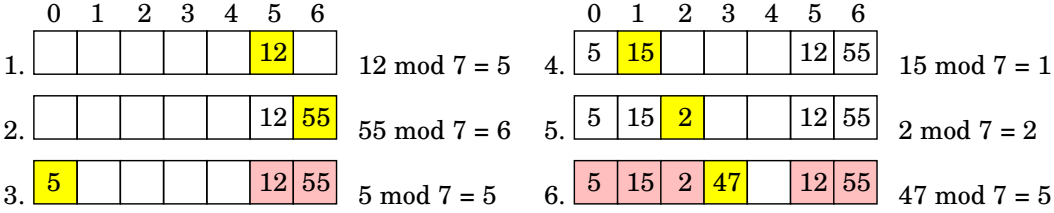
Anmerkungen:

- Beim Einfügen und Suchen wird dieselbe Sondierungsfolge durchlaufen.
- Beim Löschen wird der Datensatz nicht gelöscht, sondern nur als gelöscht markiert: Der Wert wird ggf. bei einem späteren Einfügen überschrieben.
- Je voller die Tabelle wird, umso schwieriger wird das Einfügen neuer Schlüssel.

Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k, i) = (h'(k) + i) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$ und $h'(k) = k \bmod 7$. (Sondierung: $+1, +2, +3, +4, +5, \ldots \bmod 7$)

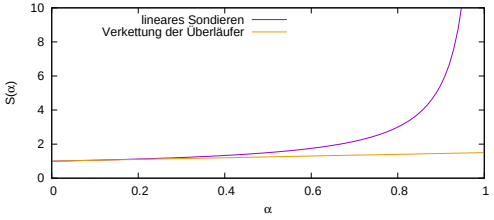


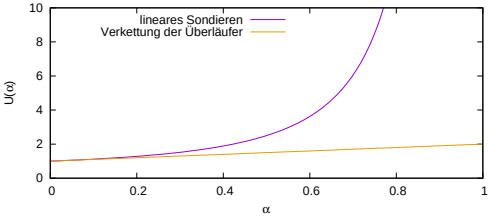
Die Sondierungsfolge $1, 2, 3, \ldots, m$ (modulo m) ist offensichtlich eine Permutation der Hash-Adressen, aber führt zu *primärer Häufung*: Behandlung einer Kollision erhöht Wahrscheinlichkeit einer Kollision in benachbarten Tabelleneinträgen.

Analyse nach Knuth[32] für $0 \leq \alpha < 1$:

$$S(\alpha) \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad U(\alpha) \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Vergleich *Verkettung der Überläufer* mit *linearem Sondieren*:



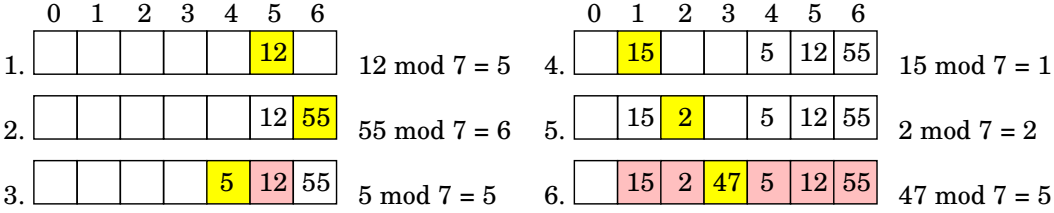


[32] Knuth, Donald E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, 1973.

Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k,i) = (h'(k) + (-1)^i \cdot \lceil i/2 \rceil^2) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$ und $h'(k) = k \bmod 7$. (Sondierung: $\mp 1^2, \mp 2^2, \mp 3^2, \mp 4^2, \ldots \bmod 7$)

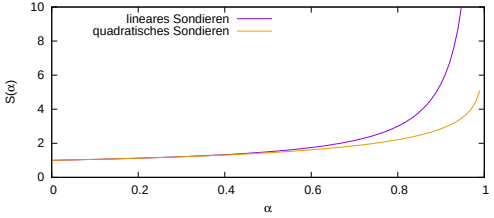


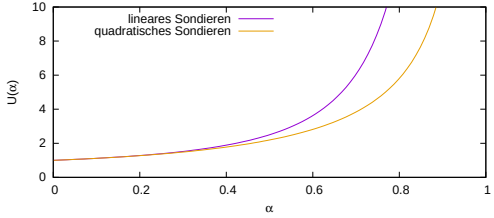
Für Primzahl $M = 4 \cdot j + 3 = 7, 11, 19, 23, 31, 43, \ldots$ ist die Sondierungsfolge eine Permutation der Hash-Adressen; keine lokale Häufung mehr, aber *sekundäre Häufung*: Schlüssel durchlaufen bei gleichem Hash-Wert immer die gleiche Ausweichsequenz!

Analyse nach Knuth für $0 \leq \alpha < 1$:

$$S(\alpha) \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \qquad U(\alpha) \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

Vergleich *lineares Sondieren* mit *quadratischem Sondieren*:





Motivation: Funktion $h(k, i)$ berücksichtigt beim linearen und quadratischen Sondieren lediglich den Schritt i. Die Sondierungsfolge ist vom Schlüssel k unabhängig und damit für alle Synonyme die gleiche.

Schlüssel behindern sich weniger, wenn die Sondierungsfolge auch für Synonyme variiert. Uniformes Sondieren berechnet daher die Folge $h(k, 1), \ldots, h(k, M)$ als Permutation aller möglichen Hash-Werte in Abhängigkeit vom Schlüssel k.

- Vorteil: Häufung wird vermieden, da unterschiedliche Schlüssel mit gleichem Hash-Wert zu unterschiedlichen Sondierungsfolgen führen.
- Nachteil: schwierige praktische Realisierung

Approximation des uniformen Sondierens:

- zufälliges Sondieren: Wähle zufällige Hashadresse für $h(k, i)$, die abhängig von k ist. Ein belegter Platz wird evtl. noch einmal betrachtet, da es keine Permutation aller Hashadressen ist, aber dies ist selten.
- double hashing: später

Annahme: Sondierungsfolgen zu verschiedenen Schlüsseln sind zufällig und jede der $M!$ möglichen Permutationen wird mit gleicher Wahrscheinlichkeit $1/M!$ gewählt.

- *Ereignis* A_i: Beim i-ten Sondierungsversuch erfolgt Zugriff auf belegten Slot.
- *Zufallsvariable* X: Anzahl der sondierten Slots bei nicht erfolgreicher Suche.
- Dann gilt für $i \geq 2$: $\quad P(X \geq i) = P(A_1 \cap \ldots \cap A_{i-1})$
- verallgemeinerter Multiplikationssatz der Wahrscheinlichkeitsrechnung:

$$\begin{aligned} & P(A_1 \cap \ldots \cap A_{i-1}) \\ = \;& P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \cdot \ldots \cdot P(A_i \mid A_1 \cap \ldots \cap A_{i-2}) \end{aligned}$$

Aufgrund unserer Annahme gilt:

$$P(A_1) = \frac{N}{M} \quad \text{und} \quad P(A_j \mid A_1 \cap \ldots \cap A_{j-1}) = \frac{N-(j-1)}{M-(j-1)}$$

Suche in $M-(j-1)$ Plätzen, von denen $N-(j-1)$ belegt sind.

Kumulierte Wahrscheinlichkeit, dass *erfolglose Suche* mindestens i Schritte braucht:

$$\begin{aligned} P(X \geq i) &= P(A_1 \cap \ldots \cap A_{i-1}) \\ &= P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \cdot \ldots \cdot P(A_{i-1} \mid A_1 \cap \ldots \cap A_{i-2}) \\ &= \frac{N}{M} \cdot \frac{N-1}{M-1} \cdot \ldots \cdot \frac{N-i+2}{M-i+2} \leq \left(\frac{N}{M}\right)^{i-1} = \alpha^{i-1} \end{aligned}$$

Der Erwartungswert ist die Summe der kumulierten Wahrscheinlichkeiten:

$$U(\alpha) = \sum_{i=1}^{N} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Bei einer *erfolgreichen Suche* wird dieselbe Sondierungsfolge wie beim Einfügen des Schlüssels durchlaufen. Für den als $(i+1)$-tes eingefügten Schlüssel entspricht dies einer erfolglosen Suche in einer Hashtabelle der Länge M mit i Schlüsseln:

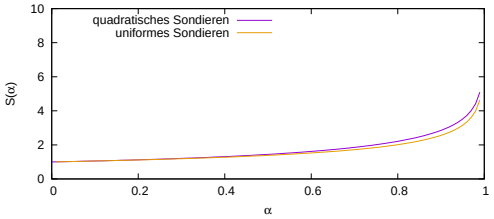
$$q_i \;=\; \frac{1}{1-i/M} \;=\; \frac{M}{M-i}$$

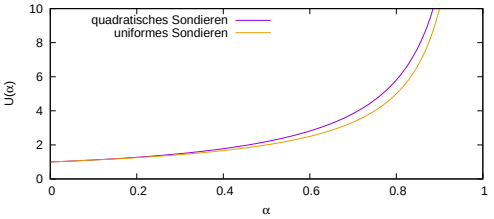
Mittelwert über alle N Schlüssel:

$$\begin{aligned} S(\alpha) \;&=\; \frac{1}{N}\cdot\sum_{i=0}^{N-1} q_i \;=\; \frac{M}{N}\cdot\sum_{i=0}^{N-1}\frac{1}{M-i} \;=\; \frac{1}{\alpha}\cdot\sum_{i=M-N+1}^{M}\frac{1}{i} \\ &\leq\; \frac{1}{\alpha}\cdot\int_{M-N}^{M}\frac{1}{x}dx \;=\; \frac{1}{\alpha}\big[\ln(M)-\ln(M-N)\big] \;=\; \frac{1}{\alpha}\ln\left(\frac{M}{M-N}\right) \\ &=\; \frac{1}{\alpha}\ln\left(\frac{1}{1-\frac{N}{M}}\right) \;=\; \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Für jede monoton fallende Funktion f gilt: $\sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)dx$

Vergleich *quadratisches Sondieren* mit *uniformes Sondieren*:





Aber: Verkettung der Überläufer hat zwei große Vorteile:

- Die asymptotische Laufzeit ist am besten.
- Die Hash-Tabelle kann mehr Elemente speichern, als das Array groß ist.

offenes Hashing vs. Verkettung der Überläufer

Hash-Tabellen mit Verkettung der Überläufer eignen sich als externe Datenstruktur: Spezielle Datenstrukturen, um große Datenmengen mit wenigen I/O-Zugriffen verwalten zu können.

Große Datenmengen können nicht im Hauptspeicher verwaltet werden, daher müssen Teile der Daten auf einen sekundären oder externen Speicher wie HDD, SSD oder Flash-Medien ausgelagert werden.

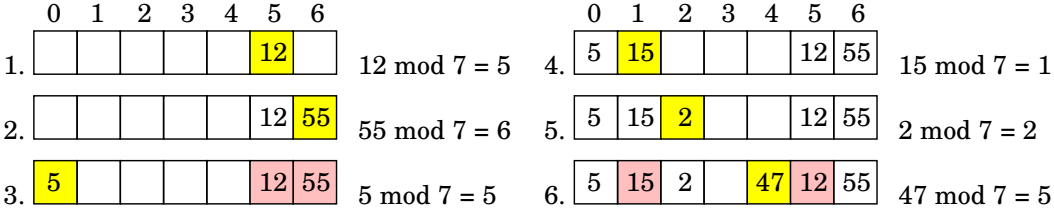
Die Seitenersetzungsverfahren wie LRU der Betriebssysteme sowie das Caching und Prefetching sind allgemeine Verfahren, die nicht speziell auf die jeweiligen Probleme abgestimmt sind.

Externe Datenstrukturen werden in der Vorlesung EAL im Master-Studiengang behandelt.

Zu zwei gegebenen Hash-Funktionen $h_1(k)$ und $h_2(k)$ sei

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$, $h_1(k) = k \bmod 7$ und $h_2 = 1 + k \bmod 5$.



Wörterbücher und Mengen werden mittels Hash-Tabellen effizient implementiert.

Die Container `std::unordered_map`[33] und `std::unordered_set`[34] in C++ sind mittels Hash-Tabellen implementiert.

> Unordered set is an associative container that contains a set of unique objects of type Key. Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal have average constant-time complexity.
>
> Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its value. This allows fast access to individual elements, since once a hash is computed, it refers to the exact bucket the element is placed into.
>
> Container elements may not be modified (even by non const iterators) since modification could change an element's hash and corrupt the container.

[33] `https://en.cppreference.com/w/cpp/container/unordered_map`

[34] `https://en.cppreference.com/w/cpp/container/unordered_set`

`std::pair<iterator,bool> insert( const value_type& value )` inserts `value` into the container, if the container doesn't already contain an element with an equivalent key.

If rehashing occurs due to the insertion, all iterators are invalidated. Otherwise iterators are not affected. References are not invalidated. Rehashing occurs only if the new number of elements is greater than `max_load_factor() * bucket_count()`. If the insertion is successful, pointers and references to the element obtained while it is held in the node handle are invalidated, and pointers and references obtained to that element before it was extracted become valid.

Achtung: Die beim Hashing angegebenen Laufzeiten sind durchschnittliche Laufzeiten. Wenn Systeme harte Echtzeitanforderungen erfüllen müssen, eignen sich Suchbäume ggf. besser als Implementierung von Wörterbüchern und Mengen, siehe `std::set`[35] und `std::map`[36]. Diese können ggf. ein Überschreiten der Antwortzeiten verhindern.

[35] `https://en.cppreference.com/w/cpp/container/set`

[36] `https://en.cppreference.com/w/cpp/container/map`

Hashtabellen werden in C++ automatisch vergrößert:

```
#include .....
const int N = 1 << 18;

int main(void) {
   unordered_set<int> s1;
   size_t c = s1.bucket_count();

   cout << setw(7) << 0 << ": " << setw(7) << c << endl;
   clock_t t1 = clock();
   for (int i = 1; i <= N; i++) {
      s1.insert(i);
      if (s1.bucket_count() != c) {
         c = s1.bucket_count();
         cout << setw(7) << i << ": " << setw(7) << c << endl;
      }
   }
   clock_t t2 = clock();
   cout << "time: " << t2 - t1 << " in Clock-Tics" << endl;
```

```
    unordered_set<int> s2;
    c = s2.bucket_count();

    s2.max_load_factor(0.8);  // !!!!!
    cout << setw(7) << 0 << ": " << setw(7) << c << endl;
    t1 = clock();
    for (int i = 1; i <= N; i++) {
        s2.insert(i);
        if (s2.bucket_count() != c) {
            c = s2.bucket_count();
            cout << setw(7) << i << ": " << setw(7) << c << endl;
        }
    }
    t2 = clock();
    cout << "time: " << t2 - t1 << " in Clock-Tics" << endl;
}
```

```
     0:       1
     1:      13
    14:      29
    30:      59
    60:     127
   128:     257
   258:     541
   542:    1109
  1110:    2357
  2358:    5087
  5088:   10273
 10274:   20753
 20754:   42043
 42044:   85229
 85230:  172933
172934:  351061
time: 22415 in Clock-Tics
```

```
     0:       1
     1:      17
    14:      37
    30:      79
    64:     167
   134:     337
   270:     709
   568:    1493
  1195:    3209
  2568:    6427
  5142:   12983
 10387:   26267
 21014:   53201
 42561:  107897
 86318:  218971
175177:  444487
time: 18836 in Clock-Tics
```

Datenstrukturen:

- `std::array` (static contiguous array)
  Is a container that encapsulates fixed size arrays.
- `std::vector` (dynamic contiguous array)
  Is a sequence container that encapsulates dynamic size arrays.
- `std::deque` is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end.
- `std::forward_list` (singly-linked list)
  Is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list. (kein erase, sondern erase-after)
- `std::list` (doubly-linked list)
  Is a container that supports constant time insertion and removal of elements from anywhere in the container.

[37] `https://en.cppreference.com/w/cpp/container`

abstrakte Datentypen vs. Datenstrukturen in der STL

Mischmasch:

- `std::set` collection of unique keys, sorted by keys
- `std::map` collection of key-value pairs, sorted by keys, keys are unique
- `std::unordered_set` collection of unique keys, hashed by keys
- `std::unordered_map` collection of key-value pairs, hashed by keys, keys are unique
- `std::multiset` collection of keys, sorted by keys
- `std::multimap` collection of key-value pairs, sorted by keys
- `std::unordered_multiset` collection of keys, hashed by keys
- `std::unordered_multimap` collection of key-value pairs, hashed by keys

weiterer Mischmasch in Vorbereitung für C++ 2023:

- `std::flat_set` adapts a container to provide a collection of unique keys, sorted by keys
- `std::flat_map` adapts a container to provide a collection of key-value pairs, sorted by unique keys
- `std::flat_multiset` adapts a container to provide a collection of keys, sorted by keys
- `std::flat_multimap` adapts a container to provide a collection of key-value pairs, sorted by keys

(abstrakte[38]) Datentypen:

- `std::stack` is a container adaptor that gives the programmer the functionality of a stack - specifically, a LIFO (last-in, first-out) data structure.

```
template<class T,
    class Container = std::deque<T>
> class stack;
```

  Operationen: push, pop, top, empty, size

- `std::queue` is a container adaptor that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

```
template<class T,
    class Container = std::deque<T>
> class queue;
```

  Operationen: front, back, push, pop, top, empty, size

[38] Da immer ein Default-Container angegeben ist, wird hier aus dem abstrakten Datentypen eine konkrete Datenstruktur.

`std::priority_queue` is a container adaptor that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.

```
template<class T,
    class Container = std::vector<T>,
    class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

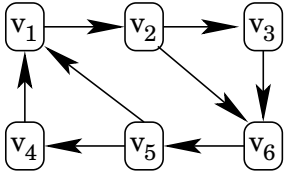
Operationen: push, pop, top, empty, size; aber kein decrease-key und kein erase

Übersicht

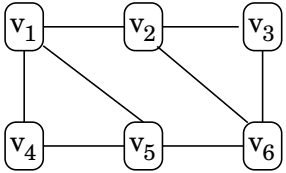
Ein *gerichteter Graph* $G = (V, E)$ besteht aus

- einer endlichen Menge von *Knoten* (engl. vertices) $V = \{v_1, \ldots, v_n\}$ und
- einer Menge von gerichteten *Kanten* (engl. edges) $E \subseteq V \times V$. [39]

Bei einem *ungerichteten Graphen* $G = (V, E)$ sind die Kanten ungeordnete Paare: $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$



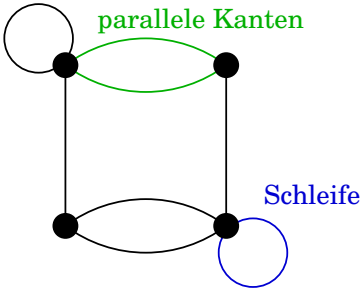
gerichteter Graph



ungerichteter Graph

[39] Zur Erinnerung: Das kartesische Produkt $V \times V$ ist die Menge aller geordneten Paare (a, b) mit $a \in V$ und $b \in V$ und beschreibt in diesem Fall die Menge aller möglichen Kanten mit Start- und Endknoten in V.

Nach dieser Definition gilt für ungerichtete Graphen:



Parallele Kanten sind nicht möglich, da die Kanten über eine Teilmenge definiert sind und in Mengen keine Elemente mehrfach vorkommen:

$$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$$

Da in der Definition $u \neq v$ gefordert wird, sind auch keine Schleifen möglich.

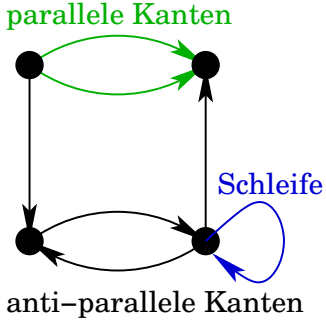
Möchte man parallele Kanten oder Schleifen zulassen, kann man die Definition mittels Multimengen formulieren.

Wir bezeichnen im Folgenden $|V| = \mathcal{V}$ und $|E| = \mathcal{E}$. Wenn es keine parallelen Kanten und keine Schleifen gibt, dann kann es maximal

$$\frac{\mathcal{V} \cdot (\mathcal{V} - 1)}{2}$$

viele Kanten geben, also gilt: $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$

Nach dieser Definition gilt für gerichtete Graphen:



Parallele Kanten sind nicht möglich, da die Kanten über eine Teilmenge definiert sind und in Mengen keine Elemente mehrfach vorkommen:

$$E \subseteq V \times V$$

Schleifen sind möglich, ebenso sind anti-parallele Kanten möglich.

Möchte man parallele Kanten zulassen, kann man die Definition mittels Multimengen formulieren.

Wenn es keine parallelen Kanten aber Schleifen gibt, dann kann es maximal $\mathcal{V} \cdot \mathcal{V}$ viele Kanten geben, also gilt auch für gerichtete Graphen: $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$

Die Kanten eines Graphen können gewichtet sein, um z.B. Längen oder Zeiten beschreiben zu können.

Gewichtete Graphen G werden wir als 3-Tupel $G = (V, E, c)$ angeben, wobei

- V die Knotenmenge,
- E die Kantenmenge und
- $c : E \to \mathbb{R}$ eine totale Funktion[40] bezeichnet, die jeder Kante e eine Zahl $c(e) \in \mathbb{R}$ zuordnet.

Graphen verwenden wir überall dort, wo ein Sachverhalt darstellbar ist durch

- eine Menge von Objekten (Entitäten)
- und Beziehungen zwischen den Objekten.

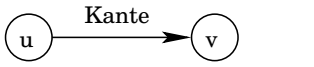
[40] Im Einzelfall kann c auch in die natürlichen, ganzen oder positiven reellen Zahlen abbilden.

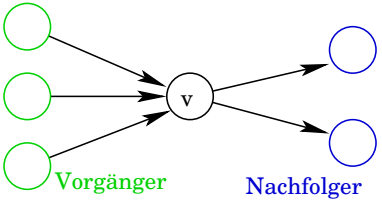
Beispiele:

- *Routenplanung:* Kreuzungen sind durch Straßenabschnitte verbunden; die Straßenabschnitte haben eine gewisse Länge, die als Kantengewicht angegeben werden kann.
- *Kursplanung:* Kurse setzen andere Kurse voraus.
- *Produktionsplanung:* Produkte werden aus Einzelteilen und Teilprodukten zusammengesetzt.
- *Schaltkreisanalyse:* Bauteile sind durch elektrische Leitungen verbunden; der Querschnitt oder die Strombelastbarkeit der Leitungen kann als Kantengewicht angegeben werden.
- *Spiele:* Der Zustand/Status eines Spiels wird durch einen Spielzug geändert.

Sei $G = (V, E)$ ein gerichteter Graph und seien $u, v \in V$.

- Sei $e = (u, v)$ eine Kante. Knoten u ist der *Startknoten*, v der *Endknoten* von e. Knoten v ist *adjazent* zu Knoten u, man nennt v auch *Nachbar* von u. Knoten u bzw. v und Kante e sind *inzident*.





- Die Vorgängermenge $N^{in}(v)$ und Nachfolgermenge $N^{out}(v)$ eines Knoten v sind definiert als

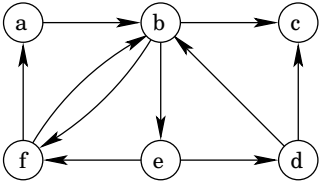
$$N^{in}(v) = \{u \in V \mid (u, v) \in E\} \qquad N^{out}(v) = \{w \in V \mid (v, w) \in E\}.$$

- Der *Eingangsgrad* $indeg(u)$ und der *Ausgangsgrad* eines Knotens u sind definiert als die Anzahl der in u einlaufenden bzw. aus u auslaufenden Kanten:

$$indeg(u) = |N^{in}(u)| \qquad outdeg(u) = |N^{out}(u)|$$

Sei $G = (V, E)$ ein gerichteter Graph und seien $u, v \in V$.

- $p = (v_0, v_1, \ldots, v_k)$ ist ein *gerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $(v_{i-1}, v_i) \in E$ für $1 \leq i \leq k$.
- Der gerichteter Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein gerichteter Weg $p = (v_0, v_1, \ldots, v_k, v_0)$ heißt *Kreis*, falls alle Kanten (v_{i-1}, v_i) und (v_k, v_0) paarweise verschieden sind. (In der Literatur entfällt manchmal der Konditionalsatz.)

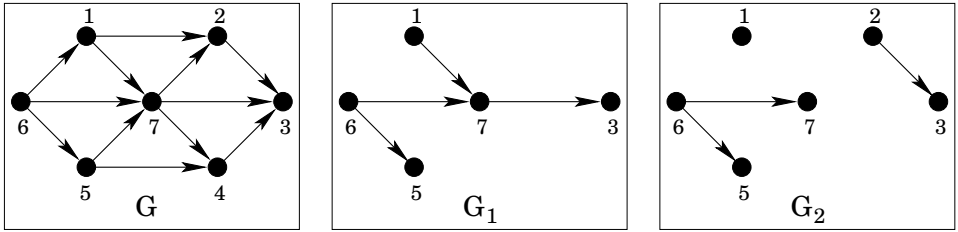


Beispiele: (a, b, e, f, a) sowie (b, e, d, b) und (a, b, e, d, b, f, a) sind Kreise, aber (a, b, e, d, b, e, f, a) ist kein Kreis, weil Kante (b, e) zweimal „durchlaufen“ wird.

- Der Graph heißt *stark zusammenhängend*, wenn für je zwei Knoten $u, v \in V$ gilt: Es existiert ein gerichteter Weg von u nach v.

Sei $G = (V, E)$ ein gerichteter Graph. Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$ gilt.

Beispiel: Die Graphen G_1 und G_2 sind Teilgraphen von G.



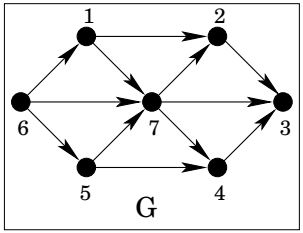
Nächste Seite: Für einen induzierten Teilgraphen G' von G fordern wir zusätzlich, dass eine Kante $e = (u, v) \in E$ mit $u, v \in V'$, bei der also Start- und Endknoten in V' liegen, auch im Teilgraphen G' vorhanden ist.

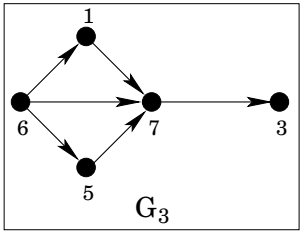
Sei $G = (V, E)$ ein gerichteter Graph. Ein Graph $G' = (V', E')$ heißt *induzierter Teilgraph* von G, falls $V' \subseteq V$ und $E' = E \cap (V' \times V')$ gilt. Dabei ist $V' \times V'$ die Menge aller möglichen Kanten mit Start- und Endknoten in V'.

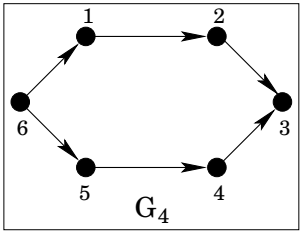
Der Graph $G|_{V'}$ bezeichnet den durch V' induzierten Teilgraphen von G. Wird in der Literatur auch als $G[V']$ geschrieben.

Oben: G_1 und G_2 sind keine induzierten Teilgraphen von G, da in G_1 die Kanten $(6, 1)$ und $(5, 7)$ fehlen und in G_2 bspw. die Kanten $(1, 7)$ und $(7, 2)$ fehlen.

Unten: G_3 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G und G_4 ist der durch die Knotenmenge $\{1, 2, 3, 4, 5, 6\}$ induzierte Teilgraph von G.







Begriffe: ungerichtete Graphen

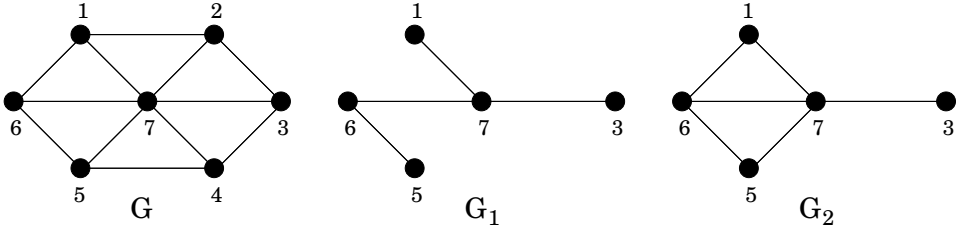
Sei $G = (V, E)$ ein ungerichteter Graph und seien $u, v \in V$.

- Sei $e = \{u, v\}$ eine Kante. Die Knoten u und v sind *adjazent*, Knoten u bzw. v und Kante e sind *inzident*. Knoten u und v sind die *Endknoten* der Kante e.
- Der *Knotengrad* von u, geschrieben $deg(u)$, ist die Anzahl der zu u inzidenten Kanten. Es gilt $2 \cdot \mathcal{E} = \sum_{v \in V} deg(v)$.
- $p = (v_0, v_1, \ldots, v_k)$ ist ein *ungerichteter Weg* in G der Länge k von Knoten u nach Knoten w, falls gilt: $v_0 = u$, $v_k = w$ und $\{v_{i-1}, v_i\} \in E$ für $1 \leq i \leq k$.
- Der ungerichtete Weg p ist *einfach*, wenn kein Knoten mehrfach vorkommt.
- Ein ungerichteter Weg $p = (v_0, v_1, \ldots, v_k, v_0)$ heißt *Kreis*, falls alle Kanten, also $\{v_{i-1}, v_i\}$ und $\{v_k, v_0\}$, paarweise verschieden sind. Hier ist der Konditionalsatz wichtig, da sonst eine einzelne Kante bereits einen Kreis darstellen würde!
- Ein ungerichteter Graph heißt *azyklisch*, wenn er keinen Kreis enthält (kreisfrei).
- Ein ungerichteter Graph heißt *zusammenhängend*, wenn je zwei Knoten durch einen Weg verbunden sind (siehe auch Kapitel Spannbäume).

Sei $G = (V, E)$ ein ungerichteter Graph.

- Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G, geschrieben $G' \subseteq G$, falls $V' \subseteq V$ und $E' \subseteq E$.
- Ein Graph $G' = (V', E')$ heißt *induzierter Teilgraph* von G, falls $V' \subseteq V$ und $E' = E \cap \{\{u, v\} \mid u, v \in V', u \neq v\}$ gilt.

Unten: G_1 ist ein Teilgraph, aber kein induzierter Teilgraph von G, da die Kanten $\{1, 6\}$ und $\{5, 7\}$ fehlen. G_2 ist der durch die Knotenmenge $\{1, 3, 5, 6, 7\}$ induzierte Teilgraph von G und daher insbesondere auch ein Teilgraph von G.

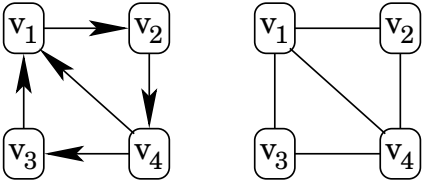


Speicherung von Graphen

Im einfachsten Fall wird ein Graph durch eine Liste der Kanten gespeichert.

- Speicherplatz: $\mathcal{O}(\mathcal{E})$
- Hinzufügen einer Kante: $\mathcal{O}(1)$
- Test, ob u adjazent zu v ist: $\mathcal{O}(\mathcal{E})$
- Zähle alle zu Knoten u benachbarten (adjazenten) Knoten auf: $\mathcal{O}(\mathcal{E})$

Beispiel:



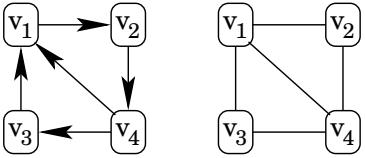
gerichtet: $\big((v_1, v_2), (v_2, v_4), (v_4, v_1), (v_4, v_3), (v_3, v_1)\big)$

ungerichtet: $\big(\{v_1, v_2\}, \{v_2, v_4\}, \{v_4, v_1\}, \{v_4, v_3\}, \{v_3, v_1\}\big)$

Die *Adjazenz-Matrix* für G ist eine $\mathcal{V} \times \mathcal{V}$-Matrix $A_G = (a_{ij})$:

$$a_{ij} = \begin{cases} 0, & \text{falls } (v_i, v_j) \notin E \\ 1, & \text{falls } (v_i, v_j) \in E \end{cases}$$

Beispiel:



gerichtet:

A	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	1	0	1	0

ungerichtet:

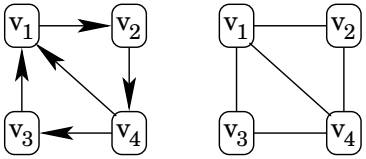
A	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

- Speicherplatz: $\mathcal{O}(\mathcal{V}^2)$
- Test, ob u adjazent zu v ist: $\mathcal{O}(1)$
- Zähle alle zu Knoten u benachbarten (adjazenten) Knoten auf: $\mathcal{O}(\mathcal{V})$

Hinzufügen und Löschen von Knoten und Kanten ist möglich.

Bei einer *Adjazenz-Liste* werden für jeden Knoten v eines Graphen $G = (V, E)$ in einer doppelt verketteten Liste $Adj[v]$ alle von v ausgehenden Kanten gespeichert.

Beispiel:



gerichtet:

$Adj[v_1] = v_2$
$Adj[v_2] = v_4$
$Adj[v_3] = v_1$
$Adj[v_4] = v_1 \leftrightarrow v_3$

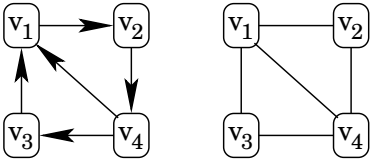
ungerichtet:

$Adj[v_1] = v_2 \leftrightarrow v_3 \leftrightarrow v_4$
$Adj[v_2] = v_1 \leftrightarrow v_4$
$Adj[v_3] = v_1 \leftrightarrow v_4$
$Adj[v_4] = v_1 \leftrightarrow v_2 \leftrightarrow v_3$

- Speicherplatz: $\mathcal{O}(\mathcal{V} + \mathcal{E})$
- Test, ob u adjazent zu v ist: $\mathcal{O}(deg(u))$
- Zähle alle zu Knoten u benachbarten (adjazenten) Knoten auf: $\mathcal{O}(deg(u))$

Hinzufügen und Löschen von Knoten und Kanten ist einfach möglich.

Bei einem *Adjazenz-Array* werden alle Kanten $\{u, v\}$ (ungerichtet) bzw. (u, v) (gerichtet) in einem Array abgelegt. Alle zu einem Knoten inzidenten Kanten liegen hintereinander im Array.



gerichtet:

u	1	2	3	4	5
ID	1	2	3	4	6

ID	1	2	3	4	5	6
v	2	4	1	1	3	-

ungerichtet:

u	1	2	3	4	5
ID	1	4	6	8	11

ID	1	2	3	4	5	6	7	8	9	10	11
v	2	3	4	1	4	1	4	1	2	3	-

Im zweiten Array kann zusätzlich zum Zielknoten v auch das Gewicht der Kante gespeichert werden. Hinzufügen und Löschen von Knoten oder Kanten wird nicht unterstützt; nur geeignet für statische Graphen.

Vergleich:

Daten-struktur	Speicher-platz	Kante hinzufügen	ist Knoten v adjazent zu u?	iteriere über die Nachbarn von u
Kantenliste	$\mathcal{O}(\mathcal{E})$	$\mathcal{O}(1)$	$\mathcal{O}(\mathcal{E})$	$\mathcal{O}(\mathcal{E})$
Adjazenz-Matrix	$\mathcal{O}(\mathcal{V}^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\mathcal{V})$
Adjazenz-Liste	$\mathcal{O}(\mathcal{V}+\mathcal{E})$	$\mathcal{O}(1)$	$\mathcal{O}(deg(u))$	$\mathcal{O}(deg(u))$
Adjazenz-Array	$\mathcal{O}(\mathcal{V}+\mathcal{E})$	–	$\mathcal{O}(deg(u))$	$\mathcal{O}(deg(u))$

Die Adjazenz-Matrix ist geeignet für *dichte* Graphen (dense graphs), die anderen Datenstrukturen sind auch geeignet für *dünn besetzte* Graphen (sparse graphs).

Anmerkung: In der numerischen Mathematik müssen oft dünn besetzte Matrizen abgespeichert werden, also Matrizen, bei denen sehr viele Einträge gleich Null sind. Dort wird das „compressed row storage“[(41)] (CRS) oder auch „compressed sparse row“ (CSR) genannte Verfahren genutzt, das den Adjazenz-Arrays sehr ähnlich ist.

[(41)] `https://de.wikipedia.org/wiki/Compressed_Row_Storage`

Um die Leistungsfähigkeit von verschiedenen Rechnersystemen zu vergleichen, gibt es unter anderem den Graph500-Benchmark[42]. Unter „Benchmark Specification“ ist dort angegeben:

- step 3: generating the edge list

  Es wird eine Kantenliste genutzt, da das Einfügen von Knoten und Kanten dort am besten unterstützt wird.
- step 4: compute sparse adjacency matrix representation

  Damit die anschließende Breitensuche und das Bestimmen der kürzesten Wege effizient durchgeführt werden können, wird aus der Kantenliste ein Adjazenz-Array erzeugt.
- step 6: breadth first search
- step 7: compute single source shortest path

[42] `https://graph500.org`

Übersicht

Aufgabe: Die Suche soll alle von einem gegebenen Startknoten s aus erreichbaren Knoten finden.

Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

```
markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle aus s auslaufenden Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante (u, v) aus D
    falls der Knoten v als „unbesucht“ markiert ist:
        markiere Knoten v als „besucht“
        füge alle aus v auslaufenden Kanten zu D hinzu
```

Anmerkungen:

- In der Datenstruktur D speichern wir diejenigen Kanten, von denen vielleicht noch unbesuchte Knoten erreicht werden können.
- Es kann auch eine Datenstruktur genutzt werden, in der die noch unbesuchten Knoten anstatt der Kanten gespeichert werden.

Laufzeit:

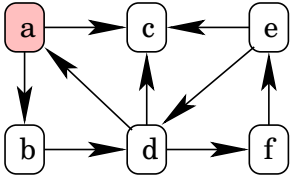
- Jede Kante wird höchstens einmal in D eingefügt.
- Jeder Knoten wird höchstens einmal inspiziert.

$\Rightarrow$ Die Laufzeit ist proportional zur Anzahl der vom Startknoten aus erreichbaren Knoten und Kanten, also $\mathcal{O}(\mathcal{V} + \mathcal{E})$.

Typ der Datenstruktur bestimmt die *Durchlaufordnung:*

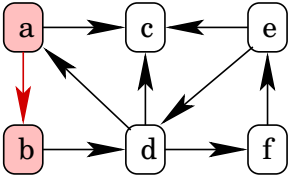
- Stack (last in, first out): *Tiefensuche*
- Queue (first in, first out): *Breitensuche*

(1)



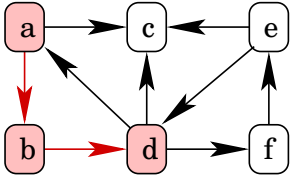
D = {(a,b), (a,c)}

(2)



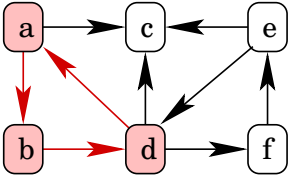
D = {(b,d), (a,c)}

(3)



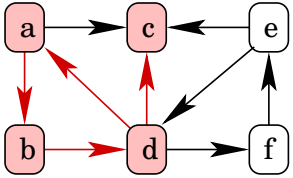
D = {(d,a), (d,c), (d,f), (a,c)}

(4)



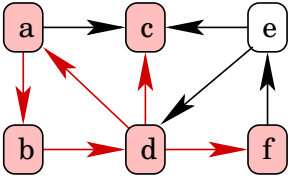
D = {(d,c), (d,f), (a,c)}

(5)



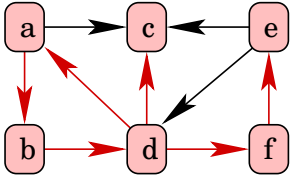
D = {(d,f), (a,c)}

(6)



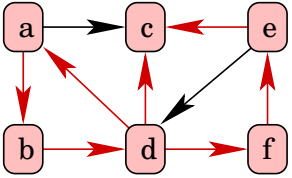
D = {(f,e), (a,c)}

(7)



D = {(e,c), (e,d), (a,c)}

(8)



D = {(e,d), (a,c)} . . .

Wir unterscheiden die Kanten bei einem gerichteten Graphen nach der Rolle, die sie bei der Tiefensuche spielen.

- *Baumkante:* Kante $(u, v) \in E$, der die Tiefensuche folgt, wo also während der Abarbeitung von `dfs(`u`)` ein Aufruf `dfs(`v`)` erfolgt.
- *Vorwärtskante:* Kante $(u, v) \in E$ mit dfb[v] $>$ dfb[u], die keine Baumkante ist.
- *Querkante:* Eine Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $<$ dfe[u].
- *Rückwärtskante:* Kante $(u, v) \in E$ mit dfb[v] $<$ dfb[u] und dfe[v] $>$ dfe[u].

Rekursive Tiefensuche: Sei $s \in V$ ein beliebiger Knoten.

```
markiere alle Knoten als „unbesucht“
dfbZähler := 0
dfeZähler := 0
dfs(s)
```

Diese Art der Tiefensuche, wo die Suche bei einem gegebenen Startknoten beginnt, bezeichnen wir als *einfache Tiefensuche*.

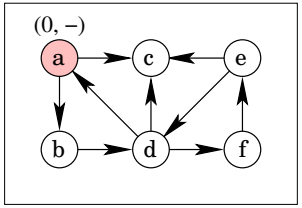
`dfs(`u`: Knoten)`

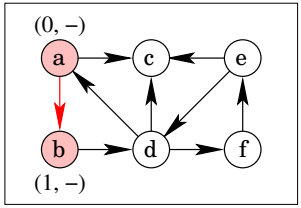
```
markiere u als „besucht“
dfb[u] := dfbZähler
dfbZähler := dfbZähler + 1
betrachte alle Kanten (u, v) ∈ E:
    falls Knoten v als „unbesucht“ markiert ist:
        dfs(v)
dfe[u] := dfeZähler
dfeZähler := dfeZähler + 1
```

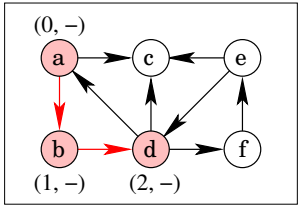
Auf den folgenden Seiten ist eine einfache Tiefensuche dargestellt, die beim Knoten a startet und die jeweils benachbarte Knoten in lexikographischer Reihenfolge besucht.

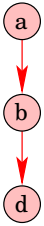
- Besuchte Knoten sind pink markiert,
- abgearbeitete Knoten sind blau markiert.

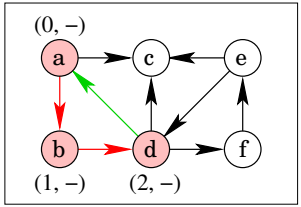
Tiefensuche

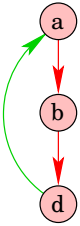


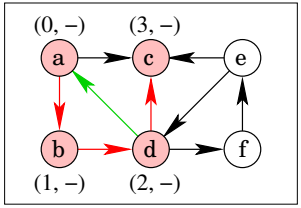


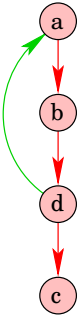


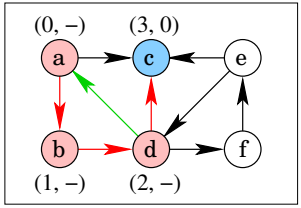


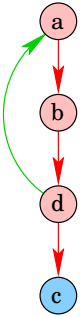


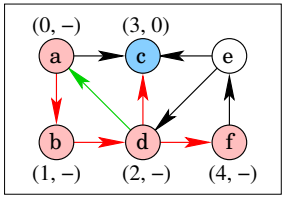


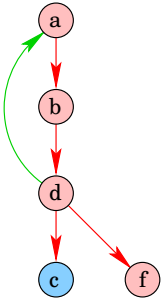

(0, –)
(3, –)
a
c
e
b
d
f
(1, –)
(2, –)

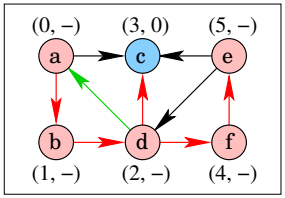

a
b
d
c

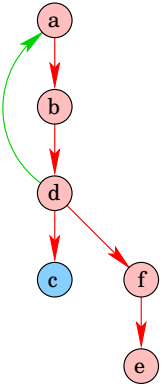

(0, –)
(3, 0)
a
c
e
b
d
f
(1, –)
(2, –)

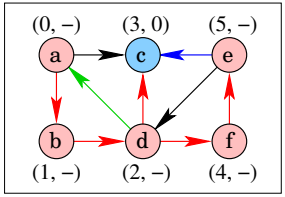

a
b
d
c

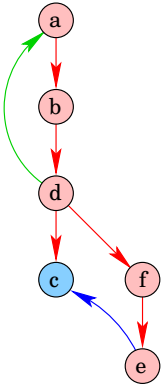

(0, –)
(3, 0)
a
c
e
b
d
f
(1, –)
(2, –)
(4, –)

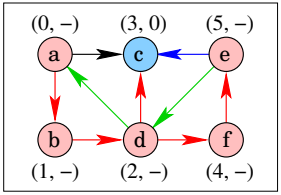

a
b
d
c
f

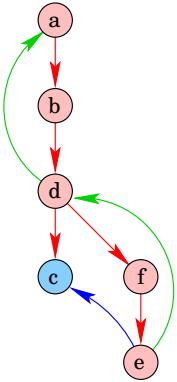

(0, –)
(3, 0)
(5, –)
a
c
e
b
d
f
(1, –)
(2, –)
(4, –)

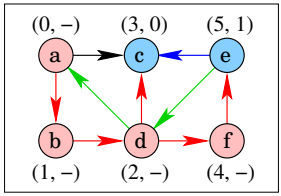

a
b
d
c
f
e

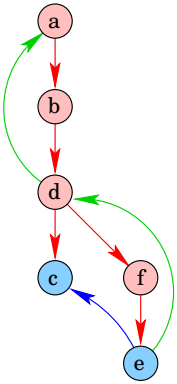

(0, –)
(3, 0)
(5, –)
a
c
e
b
d
f
(1, –)
(2, –)
(4, –)

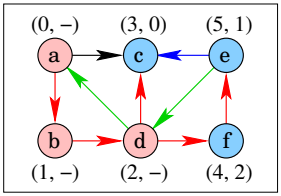

a
b
d
c
f
e

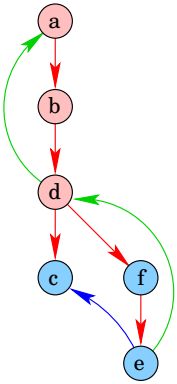

(0, –)
(3, 0)
(5, –)
a
c
e
b
d
f
(1, –)
(2, –)
(4, –)

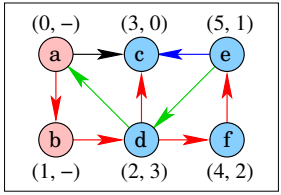

a
b
d
c
f
e

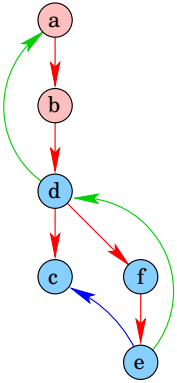

(0, –)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, –)
(2, –)
(4, –)

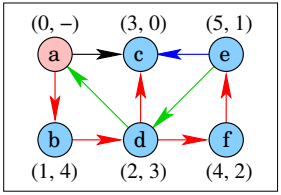

a
b
d
c
f
e

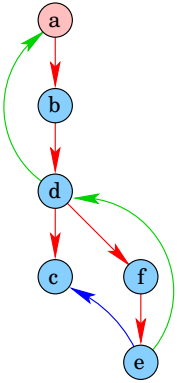

(0, –)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, –)
(2, –)
(4, 2)

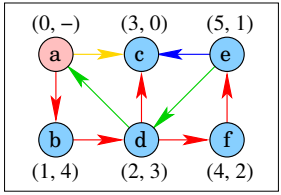

a
b
d
c
f
e

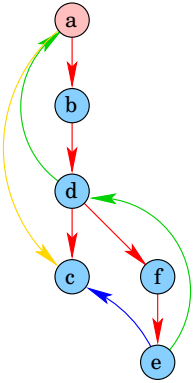

(0, –)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, –)
(2, 3)
(4, 2)

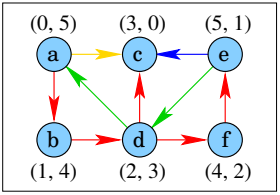

a
b
d
c
f
e

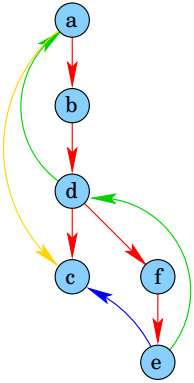

(0, –)
(3, 0)
(5, 1)
a
c
e
b
d
f
(1, 4)
(2, 3)
(4, 2)


a
b
d
c
f
e



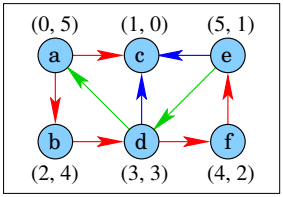


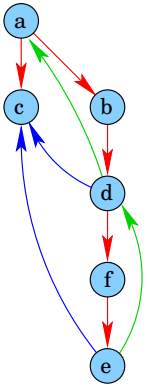


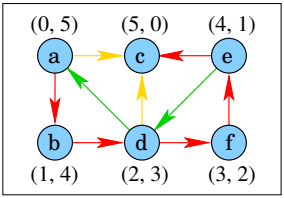


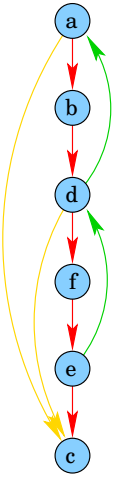
- Baumkanten (u, v) sind rot markiert: $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Rückwärtskanten (u, v) sind grün markiert: $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] < \text{dfe}[v]$
- Querkanten (u, v) sind blau markiert: $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Vorwärtskanten (u, v) sind gelb markiert: $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$

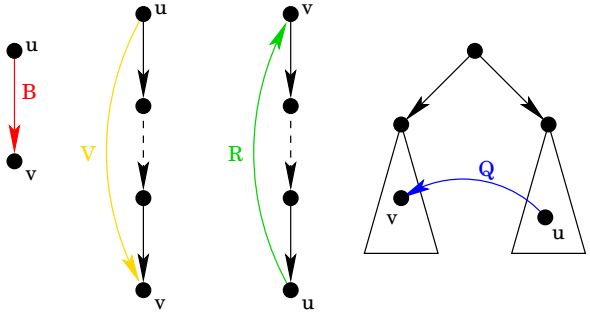
Werden die Knoten des Graphen in einer anderen Reihenfolge besucht, so ergibt sich eine andere Aufteilung der Kantenarten:











- Baumkante (u, v): $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Vorwärtskante (u, v): $\text{dfb}[u] < \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$
- Rückwärtskante (u, v): $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] < \text{dfe}[v]$
- Querkante (u, v): $\text{dfb}[u] > \text{dfb}[v]$ und $\text{dfe}[u] > \text{dfe}[v]$

Übung 24. Zeigen Sie, dass die Tiefensuche korrekt ist, dass also tatsächlich alle vom Startknoten s aus erreichbaren Knoten berichtet bzw. ausgegeben werden.

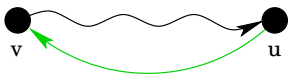
Tiefensuche: (Suche beginnt *nicht* bei einem ausgezeichneten Startknoten.)

```
markiere alle Knoten als unbesucht
solange ein unbesuchter Knoten v existiert:
    dfs(v)
```

Satz: Der gerichtete Graph G enthält einen Kreis $\iff$ die Tiefensuche auf G liefert eine Rückwärtskante.

Beweis:

"$\Leftarrow$" Für eine Rückwärtskante (u, v) gilt: $dfb(u) > dfb(v)$ und $dfe(u) < dfe(v)$. Außerdem existiert ein Weg von v nach u.



Gäbe es keinen Weg von v nach u, dann wäre `dfs(v)` beendet, bevor `dfs(u)` aufgerufen wird, also $dfe(v) < dfe(u)$. ↯↯

Beweis: (Fortsetzung)

"$\Rightarrow$" Sei $C = (v_1, v_2, \ldots, v_k, v_1)$ ein Kreis in G.

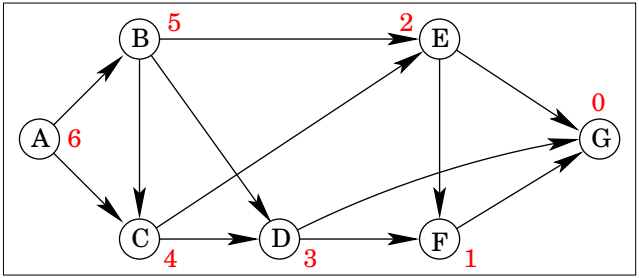
O.B.d.A. sei v_1 der Knoten aus C, der von der Tiefensuche zuerst besucht wird, also $dfb(v_1) < dfb(v_k)$.

`dfs`(v_1) wird erst beendet, wenn alle von v_1 aus erreichbaren Knoten, also insbesondere v_k, besucht und abgearbeitet wurden. Daher gilt $dfe(v_1) > dfe(v_k)$.

Also ist (v_k, v_1) eine Rückwärtskante.

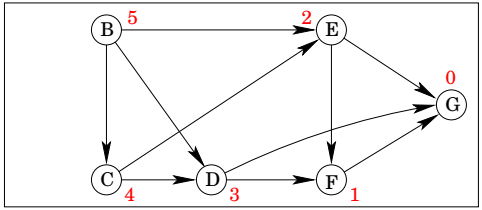
Gegeben: Ein gerichteter Graph $G = (V, E)$.

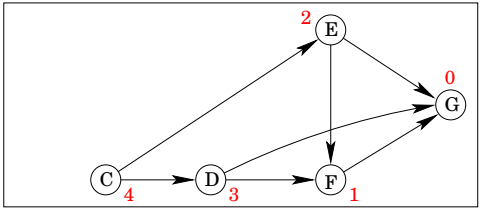
Gesucht: Eine Nummerierung $\pi(v_1), \ldots, \pi(v_n)$ der Knoten, sodass gilt:
$(u, v) \in E \;\Rightarrow\; \pi(u) > \pi(v)$

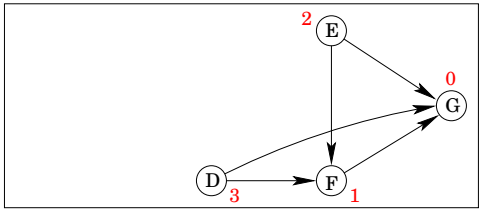


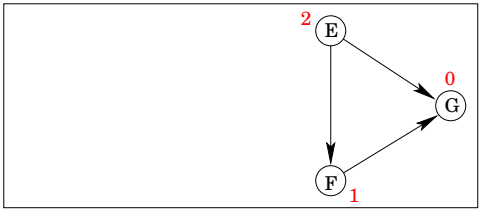
Der Knoten mit der größten Nummer hat keine einlaufenden Kanten!

Entfernt man den am höchstens nummerierten Knoten aus dem Graphen, dann hat der Knoten mit der nun größten Nummer keine einlaufenden Kanten:









Algorithmus: G ist kreisfrei $\Rightarrow$ die dfe-Nummern einer Tiefensuche sind eine topologische Sortierung!

Korrektheit: Für alle Kanten $(u, v) \in E$ muss $dfe(u) > dfe(v)$ gelten.

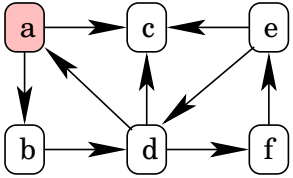
- Wenn G kreisfrei ist, treten keine Rückwärtskanten auf.
- (u, v) ist eine Baumkante
  $\rightarrow dfe(u) > dfe(v)$ ✓
  Denn: `dfs(u)` wird erst beendet, wenn `dfs(v)` bereits abgeschlossen ist.

- (u, v) ist eine Vorwärtskante
  $\rightarrow dfe(u) > dfe(v)$ ✓
  Denn: `dfs(v)` ist bereits beendet, während bei `dfs(u)` die Vorwärtskante entdeckt wird.

- (u, v) ist eine Querkante
  $\rightarrow dfe(u) > dfe(v)$ nach Definition ✓

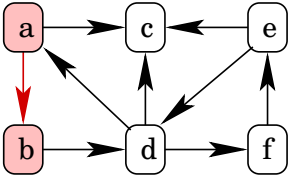
Anmerkung: Wenn G nicht kreisfrei ist, also einen Kreis enthält, dann existiert keine topologische Sortierung der Knoten.

(1)



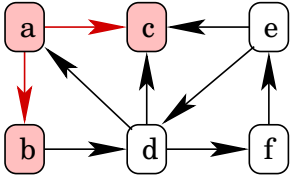
D = {(a,b), (a,c)}

(2)



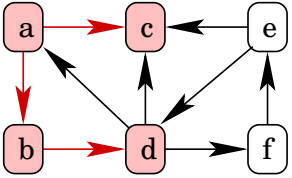
D = {(a,c), (b,d)}

(3)



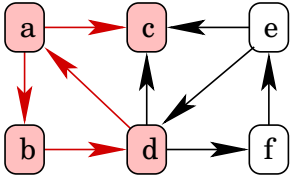
D = {(b,d)}

(4)



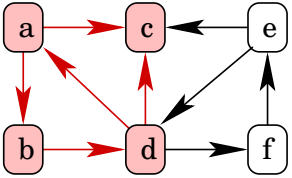
D = {(d,a), (d,c), (d,f)}

(5)



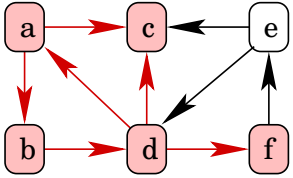
D = {(d,c), (d,f)}

(6)



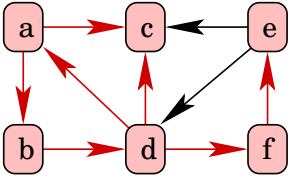
D = {(d,f)}

(7)



D = {(f,e)}

(8)



D = {(e,c), (e,d)} . . .

Obige Suche funktioniert mit leichter Modifikation auch für ungerichtete Graphen.

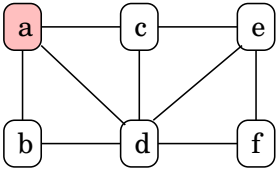
Sei D eine Datenstruktur zum Speichern von Kanten.

Algorithmus:

```
markiere alle Knoten als „unbesucht“
markiere den Startknoten s als „besucht“
füge alle mit s inzidenten Kanten zu D hinzu
solange D nicht leer ist:
    entnehme eine Kante {u, v} aus D
    falls der Knoten u/v als „unbesucht“ markiert ist:
        markiere Knoten u/v als „besucht“
        füge alle zu u/v inzidenten Kanten zu D hinzu
```

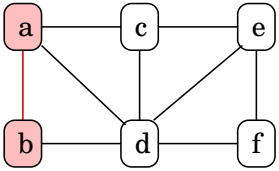
Übung 25. Welche Laufzeit hat die obige Tiefen-/Breitensuche auf ungerichteten Graphen? Ändert sich die Laufzeit, wenn keine Kanten doppelt eingefügt werden, indem vorm Einfügen einer Kante getestet wird, ob die Kante bereits enthalten ist?

(1)



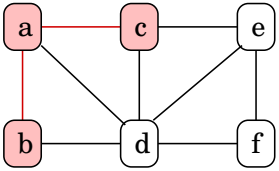
D = {{a,b}, {a,c}, {a,d}}

(2)



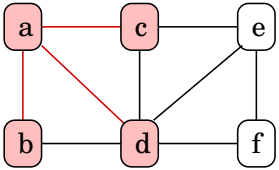
D = {{a,c}, {a,d}, {b,d}}

(3)



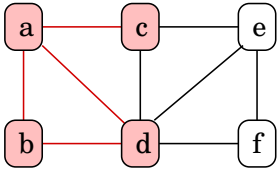
D = {{a,d}, {b,d}, {c,d}, {c,e}}

(4)



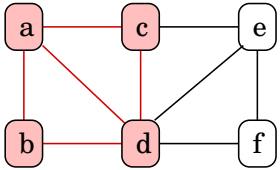
D = {{b,d}, {c,d}, {c,e}, {d,e}, {d,f}}

(5)



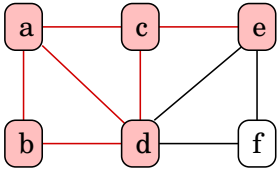
D = {{c,d}, {c,e}, {d,e}, {d,f}}

(6)



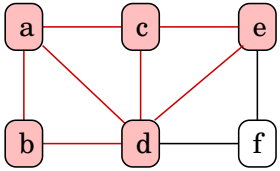
D = {{c,e}, {d,e}, {d,f}}

(7)



D = {{d,e}, {d,f}, {e,f}}

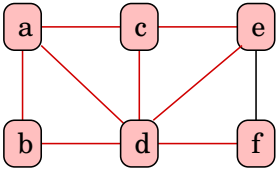
(8)



D = {{d,f}, {e,f}}

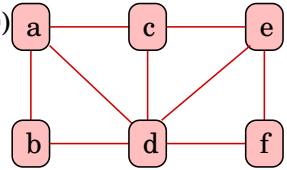
Breitensuche: ungerichtet

(9)



D = {{e,f}}

(10)



D = {}

Übersicht

Definition:

- Ein ungerichteter Graph $G = (V, E)$ heißt *zusammenhängend*, wenn gilt: Für alle $u, v \in V$ existiert ein Weg von u nach v.
- Ein *Spannbaum* T von $G = (V, E)$ ist ein zusammenhängender Teilgraph $T = (V, E')$ von G mit $\mathcal{V} - 1$ Kanten

gegeben: Ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit Kostenfunktion $c : E \to \mathbb{R}^+$.

gesucht: Ein Spannbaum $T = (V, E')$ von G mit minimalen Kosten

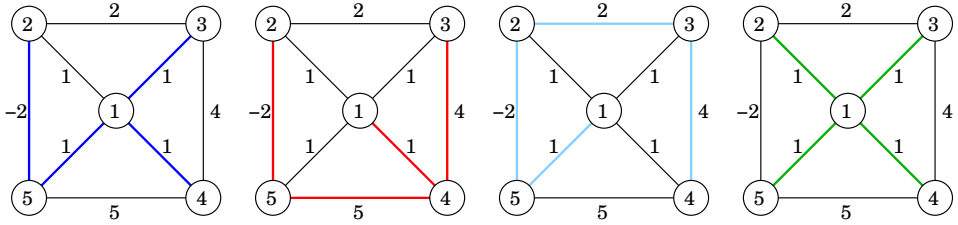
$$c(T) = \sum_{e \in E'} c(e).$$

Übung 26. Zeigen Sie, dass ein zusammenhängender Graph $G = (V, E)$ genau dann kreisfrei ist, wenn $|E| = |V| - 1$ gilt.

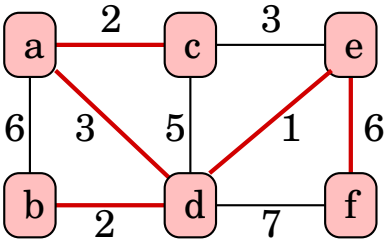
Anmerkung: Man könnte die Kosten natürlich auch anders definieren, z.B. als

- $c(T) = \max_{e \in E'}\{c(e)\}$ maximales Kantengewicht des Baums, als
- $c(T) = \prod_{e \in E'} c(e)$ Produkt der Kantengewichte oder als
- $c(T) = \Delta(T)$, wobei $\Delta(T) = \max_{v \in V}\{deg_T(v)\}$ den maximalen Knotengrad in T bezeichnet.

Dann ergäben sich evtl. jeweils andere minimale Spannbäume. Nur wenn man exakt eine Kostenfunktion festlegt, kann ein entsprechender Algorithmus zur Lösung des Problems angegeben werden.



Wir betrachten hier als Kostenfunktion die Summe der Kantengewichte.



zur Zeit bester Algorithmus:

- Karger, Klein, Tarjan: A randomized linear-time algorithm to find minimum spanning trees. Journal of the ACM, 1995.

$\rightarrow$ randomisierter Algorithmus mit erwarteter Laufzeit in $\mathcal{O}(\mathcal{E} + \mathcal{V})$

Motivation:

- *Verkabelung:* Alle Häuser sollen ans Telefonnetz angeschlossen werden und aus Kostengründen sollen dazu möglichst wenige Kabel verlegt werden.
- *Stromversorgung* von elektrischen Bauteilen auf einer Platine.
- *Routing:*
  - CISCO IP Multicast
  - Spanning Tree Protocol

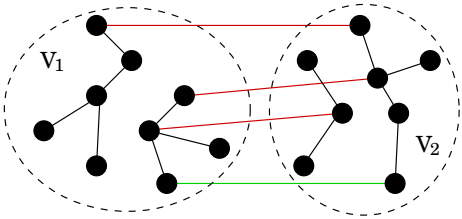
☺ Es werden nur die Straßen repariert, sodass nach wie vor alle Häuser erreichbar sind.

Beobachtung: Sei (V_1, V_2) eine disjunkte Zerlegung der Knotenmenge V, also $V_1 \cap V_2 = \emptyset$ und $V_1 \cup V_2 = V$. Dann gilt: Es gibt einen minimalen Spannbaum, der die billigste Kante $e = \{u, v\} \in E$ mit $u \in V_1$ und $v \in V_2$ enthält.

Beweis durch Widerspruch:

Wir nehmen an, kein minimaler Spannbaum enthält die billigste Kante zwischen V_1 und V_2.

Betrachte Kante eines minimalen Spannbaums, die einen Knoten aus V_1 mit einem Knoten aus V_2 verbindet.



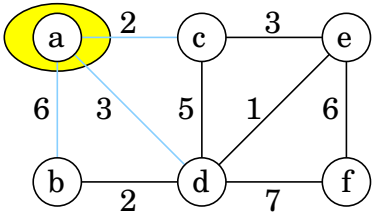
- Durch Hinzunahme der billigsten Kante zwischen V_1 und V_2 entsteht ein Kreis.
- Durch Streichen der teueren Kante zwischen V_1 und V_2 entsteht ein Spannbaum, dessen Gewicht sogar geringer ist als der ursprüngliche Spannbaum. ↯↯

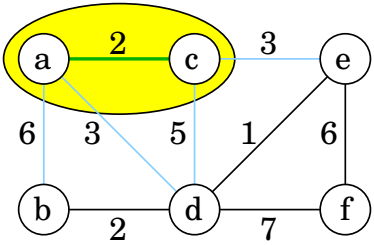
Prims Algorithmus – Idee

Starte mit nur einem einzigen Knoten in der Menge V_1.

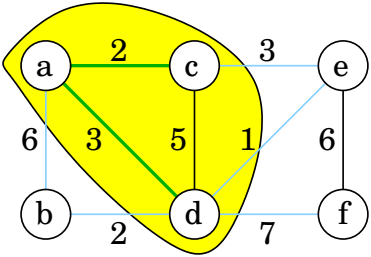
Wiederhole jeweils die folgenden Schritte, bis alle Knoten in V_1 enthalten sind:

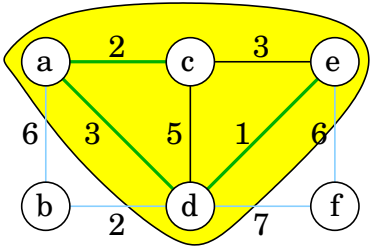
- Wähle eine preiswerteste Kante $\{u, v\}$ mit $u \in V_1$ und $v \in V_2$ aus
- und füge v der Menge V_1 hinzu.

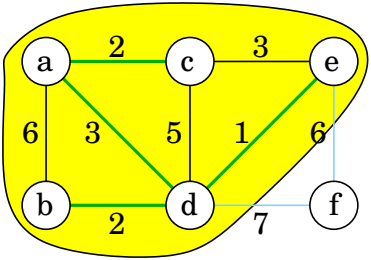


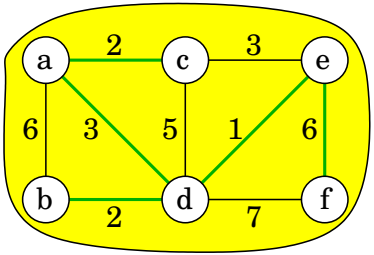


Prims Algorithmus – Idee





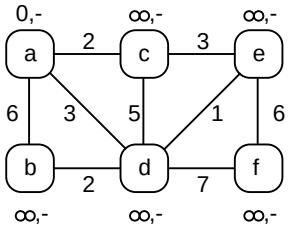




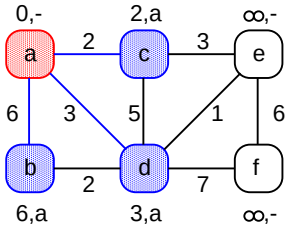
Prims Algorithmus – Umsetzung

Sei Q eine Vorrangwarteschlange zum Speichern von Knoten. Jeder Knoten $v \in V$ ist mit einem Kantenwert $key[v]$ gewichtet.

```
key[v] := ∞ for all v ∈ V
key[s] := 0 for some arbitrary s ∈ V
for all v ∈ V do
    insert(Q, key[v], v)

while not empty(Q) do
    u := minimum(Q)
    extractMin(Q)
    for all v ∈ Adj(u) do          (alle adjazenten Knoten zu u)
        if v ∈ Q and c((u, v)) < key[v]
        then key[v] := c((u, v))
             prev[v] := u
             decreaseKey(Q, v, key[v])
```



Q={(a,0);(b,∞);(c,∞);(d,∞);(e,∞);(f,∞)}

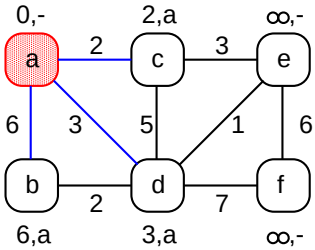


Q={(c, 2);(d,3);(b,6);(e,∞);(f,∞)}

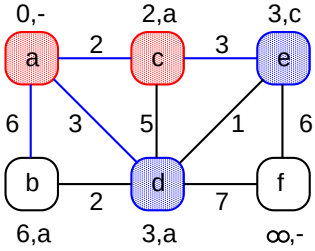
In DNE oder dem Wahlfach DNM werden Sie die Algorithmen, notiert in Tabellenform wiederfinden:

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
0,–	6,a	2,a	3,a	∞,–	∞,–

Prims Algorithmus



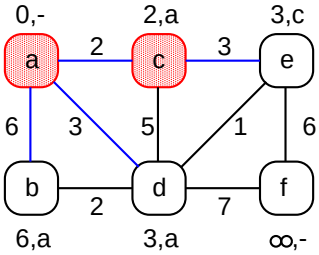
Q={(c, 2);(d,3);(b,6);(e,∞);(f,∞)}



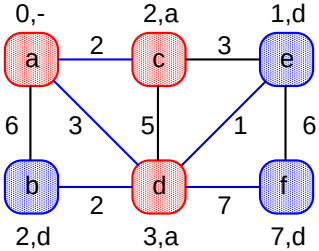
Q={(d,3);(e,3);(b,6);(f,∞)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	3,c	∞,–

Prims Algorithmus



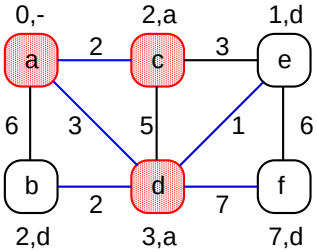
Q={(d,3);(e,3);(b,6);(f,∞)}



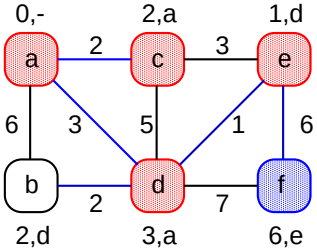
Q={(e,1);(b,2);(f,7)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	3,c	∞,–
	2,d			1,d	7,d

Prims Algorithmus



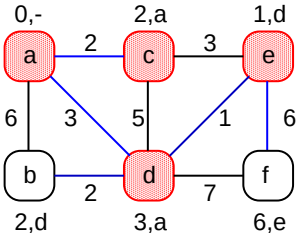
Q={(e,1);(b,2);(f,7)}



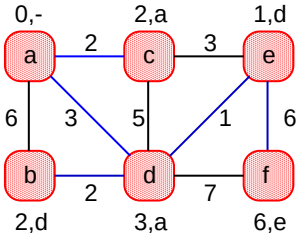
Q={(b,2);(f,6)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	3,c	∞,–
	2,d			1,d	7,d
	2,d				6,e

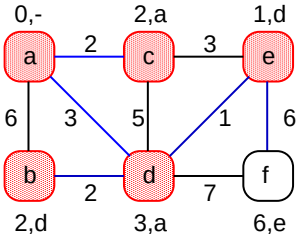
Prims Algorithmus



Q={(b,2);(f,6)}



Q={}



Q={(f,6)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	3,c	∞,–
	2,d			1,d	7,d
	2,d				6,e
					6,e
0,–	2,d	2,a	3,a	1,d	6,e

Laufzeit:

$$T(V,E) \;=\; \Theta(\mathcal{V}) \cdot T_{minimum} \;+\; \Theta(\mathcal{V}) \cdot T_{extractMin} \;+\; \Theta(\mathcal{E}) \cdot T_{decreaseKey}$$

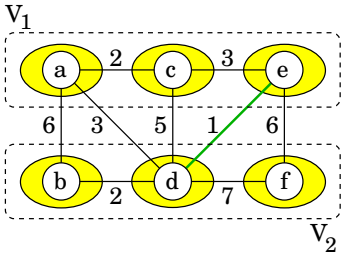
Je nach verwendeter Datenstruktur ergeben sich unterschiedliche Laufzeiten:

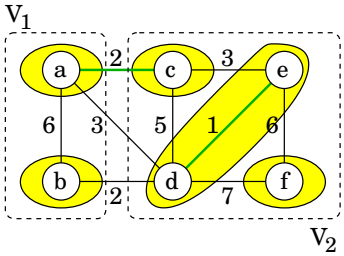
- Q als unsortiertes Array $\rightarrow$ Laufzeit $\mathcal{O}(\mathcal{V}^2 + \mathcal{E}) \subseteq \mathcal{O}(\mathcal{V}^2)$
  - $T_{minimum} \in \mathcal{O}(\mathcal{V})$
  - $T_{extractMin} \in \mathcal{O}(\mathcal{V})$
  - $T_{decreaseKey} \in \mathcal{O}(1)$
- Q als Binär-Heap $\rightarrow$ Laufzeit $\mathcal{O}(\mathcal{V} + (\mathcal{V} + \mathcal{E}) \cdot \log \mathcal{V}) \subseteq \mathcal{O}(\mathcal{E} \cdot \log \mathcal{V})$
  - $T_{minimum} \in \mathcal{O}(1)$
  - $T_{extractMin} \in \mathcal{O}(\log \mathcal{V})$
  - $T_{decreaseKey} \in \mathcal{O}(\log \mathcal{V})$
- Q als Fibonacci-Heap $\rightarrow$ amortisierte Laufzeit $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log \mathcal{V})$
  - $T_{minimum} \in \mathcal{O}(1)$
  - $T_{extractMin} \in \mathcal{O}(\log \mathcal{V})$
  - $T_{decreaseKey} \in \mathcal{O}(1)$

Initial speichern wir die Knoten in disjunkten Mengen: $M_i := \{v_i\}$

Betrachte die Kanten in der Reihenfolge aufsteigender Gewichte: $c(e_1) \leq \ldots \leq c(e_m)$

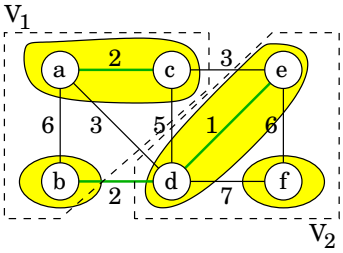
Liegen die Endknoten in verschiedenen Mengen, dann vereinige die Mengen.

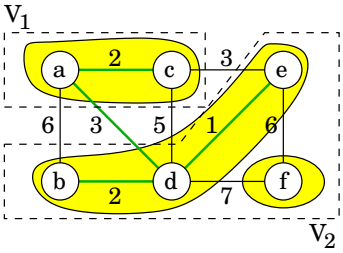




Warum ist das korrekt? Fasse die Mengen M_i zu einer Partitionierung des Graphen in zwei Knotenmengen V_1 und V_2 zusammen, sodass die aktuell betrachtete Kante $\{u, v\}$ mit $u \in V_1$ und $v \in V_2$ liegt. Dann gilt die Aussage des obigen Satzes von Seite 509: Es gibt einen minimalen Spannbaum, der die Kante $\{u, v\}$ enthält.

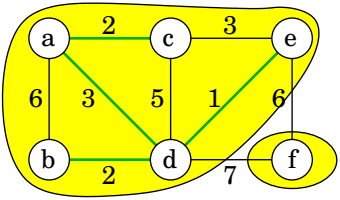
Kruskals Algorithmus – Idee

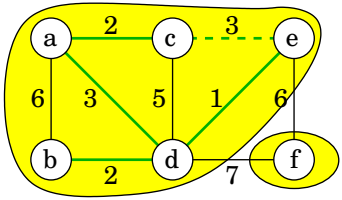




Solange die Mengen M_i zu einer Partitionierung des Graphen in zwei Knotenmengen V_1 und V_2 zusammengefasst werden können, sodass die aktuell betrachtete Kante $\{u, v\}$ mit $u \in V_1$ und $v \in V_2$ liegt, lässt sich das Verfahren fortsetzen und der obige Satz anwenden.

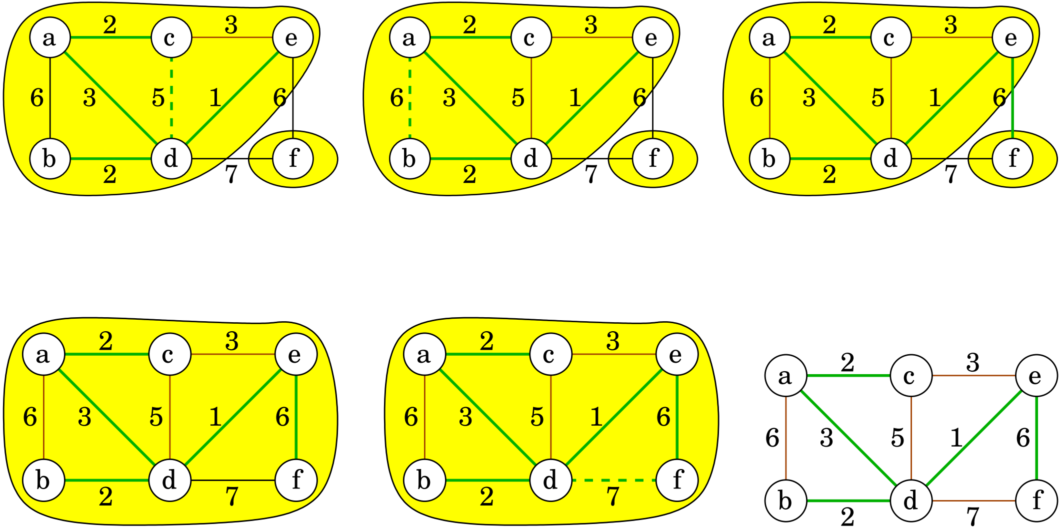
Eine solche Partitionierung lässt sich nicht finden, wenn beide Endknoten der aktuell betrachteten Kante innerhalb einer Menge M_i liegen. (rechtes Bild)





Jede Menge M_i repräsentiert einen Spannbaum des induzierten Teilgraphen $G|_{M_i}$. Würden wir diesem Spannbaum eine Kante hinzufügen, entstünde ein Kreis und die Menge M_i würde keinen Spannbaum repräsentieren.

Die Kante wird also einfach ignoriert und nicht in den Spannbaum aufgenommen.


a 2 c 3 e
6 3 5 1 6
b 2 d 7 f
a 2 c 3 e
6 3 5 1 6
b 2 d 7 f
a 2 c 3 e
6 3 5 1 6
b 2 d 7 f
a 2 c 3 e
6 3 5 1 6
b 2 d 7 f
a 2 c 3 e
6 3 5 1 6
b 2 d 7 f
a 2 c 3 e
6 3 5 1 6
b 2 d 7 f

Kruskals Algorithmus – Umsetzung

Der Algorithmus nutzt eine Datenstruktur `S`, um die Knoten des Graphen in disjunkten Mengen zu speichern. Die Operation `findSet` liefert den Repräsentanten einer solchen Menge, mittels `union` werden zwei Mengen (eigentlich zwei Bäume) verschmolzen.

$A := \emptyset$
for each vertex $v \in V$ do
    `makeSet(S, v)`

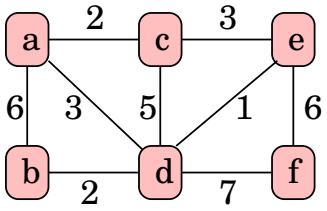
sort the edges of E by non-decreasing weight

for each edge $(u, v) \in E$ do
    $r_u :=$ `findSet(S, u)`
    $r_v :=$ `findSet(S, v)`
    if $r_u \neq r_v$ then
        $A := A \cup \{\{u, v\}\}$
        `union(S, r_u, r_v)`

Am Ende enthält die Menge A die Kanten eines minimalen Spannbaums.

Zunächst werden die Kanten anhand ihrer Gewichtung sortiert. Anschließend wird in jedem Schritt eine Kante $\{u, v\}$ aus der sortierten Liste entfernt und geprüft, ob die Knoten u und v in verschiedenen Mengen liegen. Falls ja, dann werden diese Mengen verschmolzen. Jede Menge stellt einen Spannbaum für einen Teil des Graphen dar.

Beispiel:



sorted(E)	Knotenmengen
initial	$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$
$\{d, e\}$	$\{a\}, \{b\}, \{c\}, \{d, e\}, \{f\}$
$\{a, c\}$	$\{a, c\}, \{b\}, \{d, e\}, \{f\}$
$\{b, d\}$	$\{a, c\}, \{b, d, e\}, \{f\}$
$\{a, d\}$	$\{a, c, b, d, e\}, \{f\}$
$\{c, e\}$	unverändert
$\{c, d\}$	unverändert
$\{a, b\}$	unverändert
$\{e, f\}$	$\{a, c, b, d, e, f\}$
$\{d, f\}$	unverändert

Wir benötigen eine Datenstruktur, mit der wir effizient Mengen verwalten können. Insbesondere sollen die Funktionen `findSet` und `union` unterstützt werden.

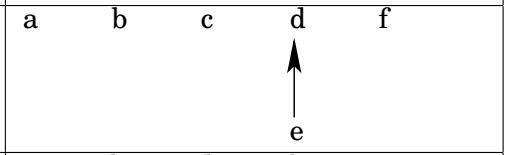
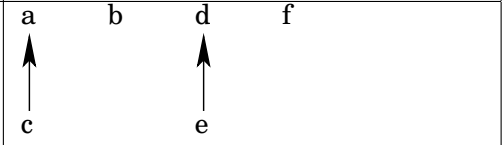
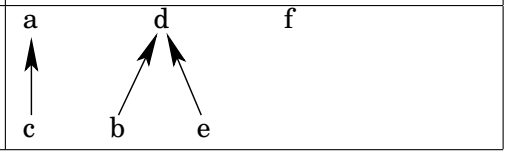
Union-Find-Datenstruktur:

- Zur Speicherung einer disjunkten Zerlegung einer Menge

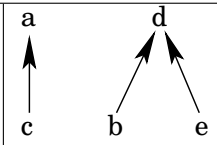
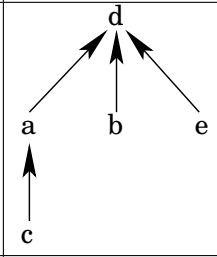
$$S = X_1 \cup X_2 \cup \ldots \cup X_k \text{ mit } X_i \cap X_j = \emptyset \text{ für } i \neq j.$$

- Speichere jede Klasse X_i in einem Baum.
- Der Repräsentant einer Klasse X_i ist die Wurzel des Baums.
- Die Funktion `findSet(`v`)` liefert den Repräsentanten des Baums, in dem Knoten v gespeichert ist.
- Damit ein Knoten schnell gefunden werden kann, werden Zeiger auf alle Elemente $v \in S$ gespeichert.
- Die Funktion `union` hängt den kleineren/flacheren Baum an die Wurzel des größeren/tieferen an.

In unserem Beispiel:

sorted(E)	Mengenrepräsentation durch Bäume
initial	
$\{d, e\}$	
$\{a, c\}$	
$\{b, d\}$	

Fortsetzung:

sorted(E)	Mengenrepräsentation durch Bäume
$\{b, d\}$	
$\{a, d\}$	
...	...

Die Funktion `makeSet(`v`)` initialisiert den Vorgänger und den Rang[43] der Wurzel:

$$\begin{array}{|l} \text{pred}[v] := 0 \\ \text{rang}[v] := 0 \end{array}$$

Laufzeit: $\mathcal{O}(1)$

Wir speichern zu jedem Knoten dessen Rang, um eine Vereinigung zweier Bäume nach ihrer Höhe durchführen zu können. Aktualisiert wird nur der Rang der Wurzel bei `union(`r_u`,`r_v`)`:

$$\begin{array}{|ll} \text{wenn} & \text{rang}[r_u] < \text{rang}[r_v] \\ & \quad \text{dann pred}[r_u] := r_v \\ \text{sonst} & \text{pred}[r_v] := r_u \\ & \quad \text{wenn} \quad \text{rang}[r_u] = \text{rang}[r_v] \\ & \qquad \quad \text{dann rang}[r_u] := \text{rang}[r_u] + 1 \end{array}$$

Laufzeit: $\mathcal{O}(1)$

Dabei sind r_u und r_v die Repräsentanten der Mengen, die u bzw. v enthalten.

[43] In unserem Fall ist der Rang die Höhe des Baums.

Wir zeigen mittels vollständiger Induktion über die Höhe h eines Baums, dass die Höhe der so entstehenden Bäume nur logarithmisch in der Anzahl der Knoten ist. Also: Ein Baum der Höhe h hat mindestens 2^h viele Knoten.

Induktion über die Höhe h des Baums.

- Induktionsanfang $h = 0$: Der Baum T besteht nur aus dem Wurzelknoten, also gilt $\mathit{size}(T) = 1 \geq 2^0 = 1$
- Induktionsschritt $h \rightsquigarrow h + 1$: Dieser Fall kann nur auftreten, wenn zwei gleich hohe Bäume T_1 und T_2 aneinander gehängt werden.

$$\begin{aligned} \mathit{size}(T) &= \mathit{size}(T_1) + \mathit{size}(T_2) \\ &\geq 2^h + 2^h = 2 \cdot 2^h = 2^{h+1} \checkmark \end{aligned}$$

Folgerung: Die Laufzeit einer `findSet`-Operation ist logarithmisch in der Anzahl der Knoten beschränkt.

Laufzeit:

- Initialisierung: $\mathcal{O}(\mathcal{V})$
- Sortieren der Kanten: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{E}))$
- Schleifendurchläufe mal Aufwand `findSet` und `union`: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$
- Gesamtlaufzeit: $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$

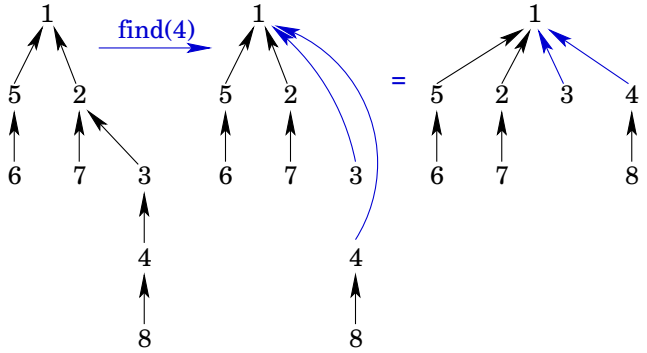
Wir hatten bereits festgestellt, dass $\mathcal{E} \in \mathcal{O}(\mathcal{V}^2)$ gilt. Also gilt:

$$\mathcal{O}(\log(\mathcal{E})) \;=\; \mathcal{O}(\log(\mathcal{V}^2)) \;=\; \mathcal{O}(2 \cdot \log(\mathcal{V})) \;=\; \mathcal{O}(\log(\mathcal{V}))$$

Eine bessere Laufzeit erhält man, wenn man während der `findSet`-Operation die Pfadlängen verkürzt.

`findSet(`x`)` mit Pfadkomprimierung:

```
res := x
z := x
while pred[res] ≠ 0 do
        res = pred[res]
while pred[z] ≠ res do
        tmp := z
        z := pred[z]
        pred[tmp] := res
return res
```



- Die Pfadkomprimierung macht die `findSet`-Methode ungefähr doppelt so teuer wie vorher, die asymptotische Laufzeit wird nicht größer.
- Eine amortisierte Laufzeitanalyse liefert: Kruskals Algorithmus hat für alle praktischen Eingaben eine lineare Laufzeit.

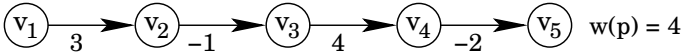
Übersicht

Gegeben sei ein gerichteter, stark zusammenhängender Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \to \mathbb{R}$.

- Die *Länge eines Weges* $p = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} v_3 \xrightarrow{e_3} \cdots \xrightarrow{e_{k-1}} v_k \xrightarrow{e_k} v_{k+1}$ ist definiert als Summe der Kantengewichte:

$$w(p) = \sum_{i=1}^{k} c(e_i)$$

Beispiel:



- Ein *kürzester Weg* von u nach v ist ein Weg minimaler Länge von u nach v.

[44] Wir betrachten in diesem Kapitel meist gerichtete Graphen, um bspw. für die Routenplanung auch Einbahnstraßen modellieren zu können. Die Algorithmen funktionieren aber auch für ungerichtete Graphen.

Gegeben: Ein (un-)gerichteter, (stark) zusammenhängender Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \to \mathbb{R}$.

Varianten:

1. *Single pair shortest paths:* (one-to-one) Suche von einem gegebenen Knoten $s \in V$ (source) aus den kürzesten Weg zu einem gegebenen Knoten $t \in V$ (target).
2. *Single source shortest paths:* (one-to-all) Suche von einem gegebenen Knoten $s \in V$ aus die kürzesten Wege zu allen anderen Knoten.

$\to$ Algorithmen von Dijkstra (nur für $c : E \to \mathbb{R}_0^+$) oder Bellman-Ford
für Typ 1 wird nur die Abbruchbedingung modifiziert

3. *All pairs shortest paths:* (all-to-all) Suche für jedes Paar $(u, v) \in V \times V$ den kürzesten Weg von u nach v.

$\to$ Algorithmus von Floyd

Motivation:

- *Reiseplanung:* Finde den kürzesten bzw. schnellsten Weg zum Urlaubsort.
- *Kostenminimierung:* Finde Zugverbindung
  - mit möglichst kurzer Reisezeit,
  - bei der man möglichst wenig umsteigen muss[45], oder
  - die möglichst preiswert ist.
- *Routing im Internet:* OSPF (Open Shortest Path First)

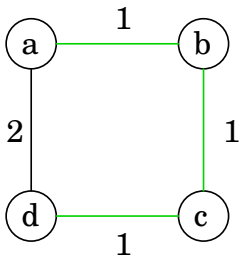
Anmerkung: Das Problem, einen *einfachen längsten Weg* zu finden, ist sehr schwer, genauer, es ist NP-vollständig[46]. Man zeigt so etwas durch eine Reduktion von einem Problem, das bekanntermaßen schwer zu lösen ist: Hamilton Path $\leq_p$ Longest Path

> Graph $G = (V, E)$ contains a hamilton path (see page 571) if and only if there is a simple longest path of length $\mathcal{V} - 1$.

[45] Die Algorithmen unterscheiden sich grundsätzlich von den hier vorgestellten, siehe bspw. die Übersicht „Route Planning in Transportation Networks“ unter `https://arxiv.org/abs/1504.05140`

[46] Die Bedeutung von NP-vollständig wird in THI erklärt.

Warum benötigen wir einen neuen Algorithmus zur Berechnung kürzester Wege? Sind die kürzesten Wege nicht bereits durch einen minimalen Spannbaum gegeben?

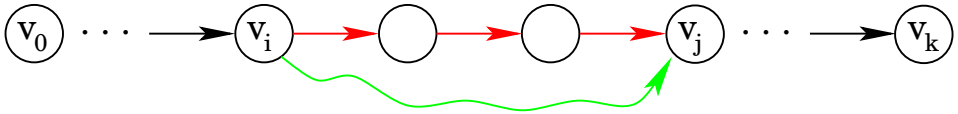


- Der minimale Spannbaum im neben stehenden Graphen ist eindeutig und durch die grün markierten Kanten gekennzeichnet.
- Der kürzeste Weg von a nach d ist allerdings durch die Kante $\{a, d\}$ gegeben, die nicht zum minimalen Spannbaum gehört.

Um dynamische Programmierung oder Greedy nutzen zu können, müssen wir zeigen, dass die *Optimale Sub-Struktur* gilt: Ein Teilweg eines kürzesten Weges ist ebenfalls ein kürzester Weg.

Theorem: Sei $(v_0, v_1, v_2, \ldots, v_k)$ ein kürzester Weg von v_0 nach v_k. Dann ist jeder Teilweg von v_i nach v_j für $0 \leq i < j \leq k$ auch ein kürzester Weg von v_i nach v_j.

Beweis: Cut and paste.



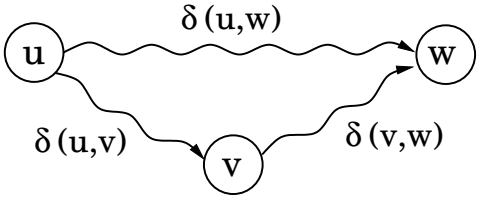
Gäbe es einen kürzeren Weg von v_i nach v_j, dann könnte auch der Weg von v_0 nach v_k verkürzt werden. ↯↯

Bezeichne $\delta(u, v)$ die *Länge des kürzesten Wegs* von u nach v. Falls kein Weg von u nach v existiert, dann gelte $\delta(u, v) = \infty$.

Dreiecksungleichung: Für alle Knoten $u, v, w \in V$ gilt:

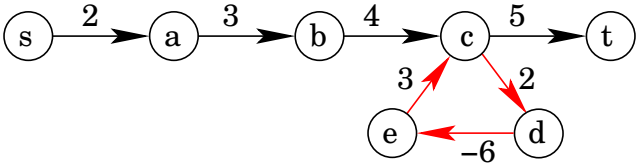
$$\delta(u, w) \leq \delta(u, v) + \delta(v, w)$$

Beweis:



Wäre $\delta(u, v) + \delta(v, w) < \delta(u, w)$, dann wäre $\delta(u, w)$ nicht die Länge des kürzesten Weges von u nach w.

Enthält ein Graph einen Kreis negativer Länge, dann existieren einige kürzeste Wege nicht, da der Kreis bei jedem Durchlaufen die Weglänge verkürzt. Im folgenden Beispiel existieren die kürzesten Wege von s nach c, d, e und t nicht.

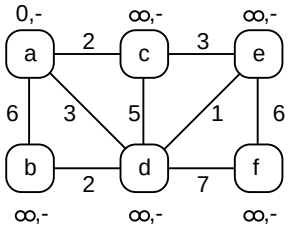


Bezeichnungen hier wie bisher:

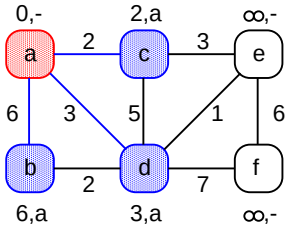
- Eine endliche Folge von Knoten $p = (v_1, v_2 \ldots, v_m)$ heißt Weg, falls $(v_{i-1}, v_i) \in E$ für $i \in \{2, \ldots, m\}$.
- Ein Weg heißt einfach, wenn alle v_i paarweise verschieden sind, wenn also kein Knoten mehrfach vorkommt.
- Suchen wir *einfache* Wege, dann sind diese auch bei Kreisen negativer Länge wohldefiniert. Aber: Kürzeste (einfache) Wege ist NP-vollständig!

Single source shortest paths: Sei Q eine Vorrangwarteschlange zum Speichern von Knoten. Jeder Knoten $v \in V$ ist mit einem Wert $d[v]$ (distance) gewichtet und besitzt einen Vorgängerknoten $pred[v]$ (predecessor). Es gelte $c : E \to \mathbb{R}_0^+$.

```
d[v] := ∞ for all v ∈ V
d[s] := 0 for some arbitrary s ∈ V
for all v ∈ V do
    insert(Q, d[v], v)

while not empty(Q) do
    u := minimum(Q)
    extractMin(Q)
    S := S ∪ {u}          ▷ nur notwendig für den Beweis
    for all v ∈ Adj(u) do
        if v ∈ Q and d[v] > d[u] + c((u, v))
        then d[v] := d[u] + c((u, v))
             pred[v] := u
             decreaseKey(Q, v, d[v])
```



Q={(a,0);(b,∞);(c,∞);(d,∞);(e,∞);(f,∞)}

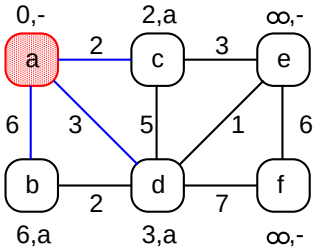


Q={(c, 2);(d,3);(b,6);(e,∞);(f,∞)}

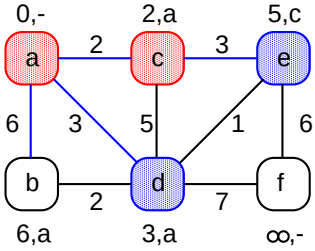
Tabellennotation nach DNE und DNM:

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
0,–	6,a	2,a	3,a	∞,–	∞,–

Dijkstras Algorithmus



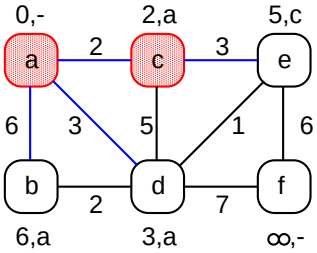
Q={(c, 2);(d,3);(b,6);(e,∞);(f,∞)}



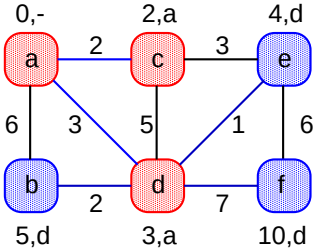
Q={(d,3);(e,5);(b,6);(f,∞)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	5,c	∞,–

Dijkstras Algorithmus

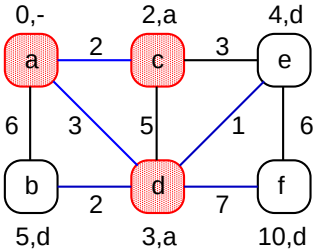


Q={(d,3);(e,5);(b,6);(f,∞)}

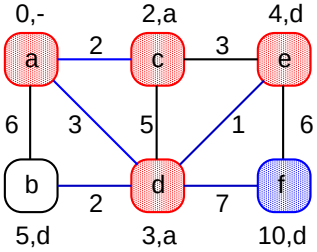


Q={(e,4);(b,5);(f,10)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	5,c	∞,–
	5,d			4,d	10,d



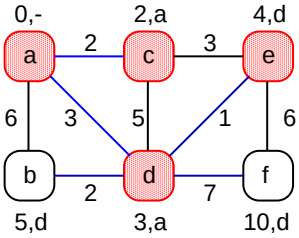
Q={(e,4);(b,5);(f,10)}



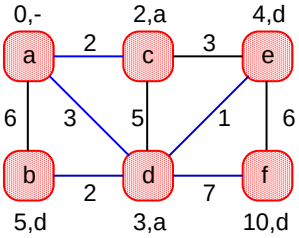
Q={(b,5);(f,10)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	5,c	∞,–
	5,d			4,d	10,d
	5,d				10,d

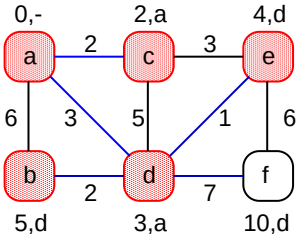
Dijkstras Algorithmus



Q={(b,5);(f,10)}



Q={}



Q={(f,10)}

a	b	c	d	e	f
0,–	∞,–	∞,–	∞,–	∞,–	∞,–
	6,a	2,a	3,a	∞,–	∞,–
	6,a		3,a	5,c	∞,–
	5,d			4,d	10,d
	5,d				10,d
					10,d
0,–	5,d	2,a	3,a	4,d	10,d

Laufzeit: Gleiche Laufzeit wie Prims Algorithmus!

$$T(V, E) \;=\; \Theta(\mathcal{V}) \cdot T_{minimum} \;+\; \Theta(\mathcal{V}) \cdot T_{extractMin} \;+\; \Theta(\mathcal{E}) \cdot T_{decreaseKey}$$

Implementiere Q mittels

- Array: Laufzeit $\mathcal{O}(\mathcal{V}^2)$
- Binär-Heap: Laufzeit $\mathcal{O}(\mathcal{E} \cdot \log(\mathcal{V}))$
- Fibonacci-Heap: amortisierte Laufzeit $\mathcal{O}(\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))$

Ungewichtete Graphen: Wird die Länge eines Weges durch die Anzahl der Kanten bestimmt, reicht eine modifizierte Breitensuche zur Bestimmung der kürzesten Wege.

Die Modifikation besteht darin, dass die Vorgänger abgespeichert werden, um die Wege zu markieren.

Lemma: Während der Ausführung des Algorithmus gilt stets $d[v] \geq \delta(s, v)$ für alle Knoten $v \in V$ des Graphen.

Beweis durch Widerspruch:

- Sei v der Knoten, für den während der Ausführung des Algorithmus zum ersten Mal $d[v] < \delta(s, v)$ gilt.
- Außerdem sei u der Knoten, durch den $d[v]$ verkleinert wurde: $d[v] := d[u] + c((u, v))$

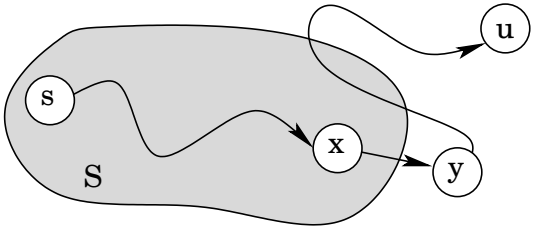
Dann gilt:

$$
\begin{array}{lcll}
d[v] & < & \delta(s, v) & \text{aufgrund der Annahme} \\
 & \leq & \delta(s, u) + \delta(u, v) & \text{wegen der Dreiecksungleichung} \\
 & \leq & \delta(s, u) + c((u, v)) & \text{kürzester Weg} \leq \text{konkreter Weg} \\
 & \leq & d[u] + c((u, v)) \; \lightning\lightning & v \text{ verletzt Invariante zum 1. Mal}
\end{array}
$$

Theorem: Dijkstras Algorithmus liefert $d[v] = \delta(s, v)$ für alle Knoten $v \in V$.

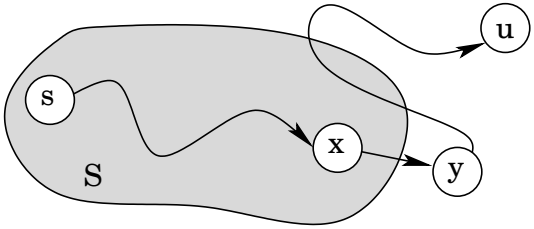
Beweis: Wir zeigen, dass $d[v] = \delta(s, v)$ gilt, wenn v zur Menge S hinzu genommen wird. Da die Werte von $d[v]$ nur kleiner werden können, ist damit die Aussage gezeigt.

Angenommen, u ist der erste Knoten, der zur Menge S hinzu genommen wird, für den $d[u] \neq \delta(s, u)$ gilt.



Sei y der erste Knoten aus $V - S$ auf einem kürzesten Weg von s nach u, und sei x der Vorgänger von y auf diesem Weg.

(Fortsetzung)

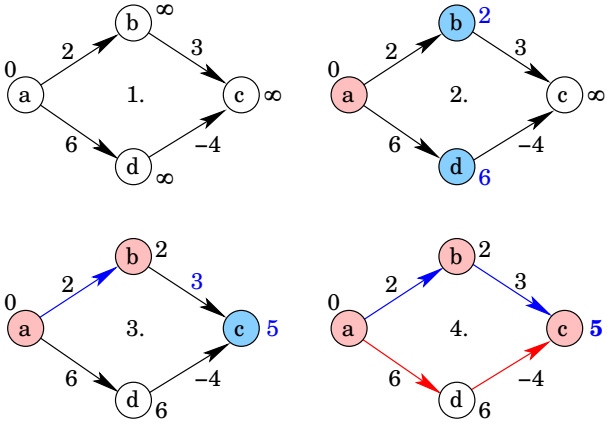


- Es gilt $d[x] = \delta(s, x)$, da u der erste Knoten ist, der die Invariante verletzt.
- Da Teilwege von kürzesten Wegen auch kürzeste Wege sind, wurde $d[y]$ auf $\delta(s, x) + c((x, y)) = \delta(s, y)$ gesetzt, als die Kante (x, y) nach Hinzunahme von x zu S untersucht wurde.

  Also gilt $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$, denn $c : E \to \mathbb{R}_0^+$.
- Aber es gilt $d[u] \leq d[y]$, weil der Algorithmus u wählt.
- Also: $d[y] = \delta(s, y) = \delta(s, u) = d[u]$ ↯↯

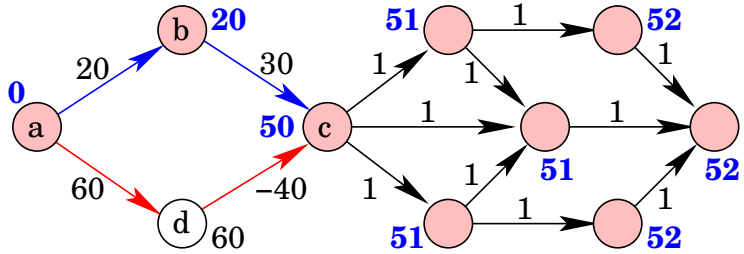
Anmerkung: Dijkstras Algorithmus arbeitet nur korrekt, wenn im Graphen keine negativen Kantengewichte existieren.

Beispiel:



Frage:
Warum erweitern wir Dijkstra nicht so, dass die Distanz bei c geändert wird, wenn ein kürzerer Weg gefunden wird?

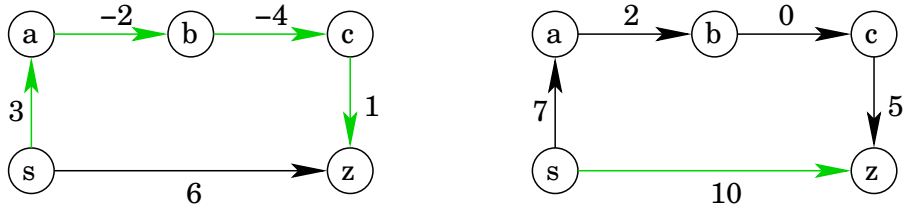
Antwort: Alle von c aus erreichbaren und schon berechneten kürzesten Wege müssen erneut berechnet werden. $\rightarrow$ Laufzeit steigt enorm



Weitere Frage: Wie werden Endlosschleifen bei Kreisen negativer Länge verhindert?

Frage: Warum addieren wir nicht einfach den Betrag des kleinsten Kantengewichts auf alle Kantengewichte auf und berechnen dann mittels Dijkstras Algorithmus die kürzesten Wege?

Antwort: Weil es nicht funktioniert, da die kürzesten Wege nicht erhalten bleiben, wie folgendes Beispiel zeigt.



- Der kürzeste Weg zwischen Startknoten s und Zielknoten z ist jeweils grün markiert.
- Links ist der Originalgraph zu sehen, im rechten Graphen wurde auf jedes Kantengewicht der Wert 4 addiert.

Lösung des Single-Source-Shortest-Paths-Problems:

- Im Gegensatz zu Dijkstras Algorithmus sind negative Kantengewichte erlaubt.
- gegeben: Graph $G = (V, E, c)$ mit Kostenfunktion $c : E \to \mathbb{R}$ und ein Startknoten $s \in V$.

$d[v] := \infty$ for all $v \in V$
$d[s] := 0$ for some arbitrary $s \in V$
for $i := 1$ to $\mathcal{V} - 1$ do
    for each edge $(u, v) \in E$ do
        if $d[v] > d[u] + c((u, v))$
        then $d[v] := d[u] + c((u, v))$
for each edge $(u, v) \in E$ do
    if $d[v] > d[u] + c((u, v))$
    then report: „negative-weight cycle exists“

Anmerkung:

- Ein kürzester Weg, der keine negativen Kreise enthält, besucht keinen Knoten zweimal.
- Daher besteht ein solcher Weg aus höchstens $\mathcal{V} - 1$ Kanten.

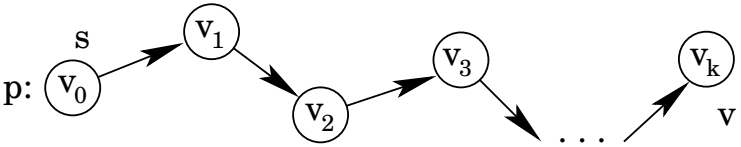
Laufzeit:

- Die äußere Schleife wird $(\mathcal{V} - 1)$-mal durchlaufen.
- Die innere Schleife wird $\mathcal{E}$-mal durchlaufen.
- Alle Operationen der inneren Schleife kosten Zeit $\mathcal{O}(1)$.

$\rightarrow$ Gesamte Laufzeit in $\Theta(\mathcal{V} \cdot \mathcal{E})$.

Korrektheit: Der Graph G enthalte keine negativen Kreise.

- Sei $v \in V$ ein beliebiger Knoten, und betrachte einen kürzesten Weg p von s nach v mit minimaler Anzahl Kanten.

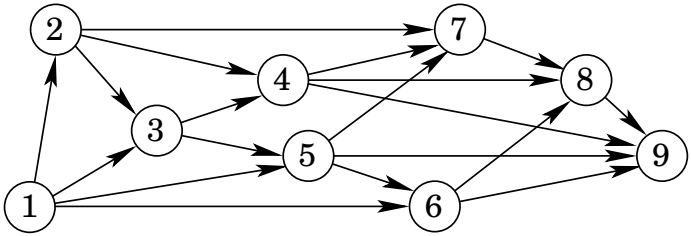


  Da p kürzester Weg ist: $\delta(s, v_i) = \delta(s, v_{i-1}) + c((v_{i-1}, v_i))$.
- Initial gilt: $d[v_0] = 0 = \delta(s, v_0)$
- Wir betrachten die Durchläufe der äußeren Schleife:
  - nach 1. Durchlauf durch E gilt: $d[v_1] = \delta(s, v_1)$.
  - nach 2. Durchlauf durch E gilt: $d[v_2] = \delta(s, v_2)$.
  - $\vdots$
  - nach k. Durchlauf durch E gilt: $d[v_k] = \delta(s, v_k)$.
- Ein kürzester Weg besteht aus höchstens $\mathcal{V} - 1$ Kanten.

Anmerkung: Wenn ein Wert $d[v]$ nach $\mathcal{V} - 1$ Durchläufen der äußeren Schleife nicht konvergiert, dann enthält Graph G einen negativen Kreis, der von s aus erreichbar ist.

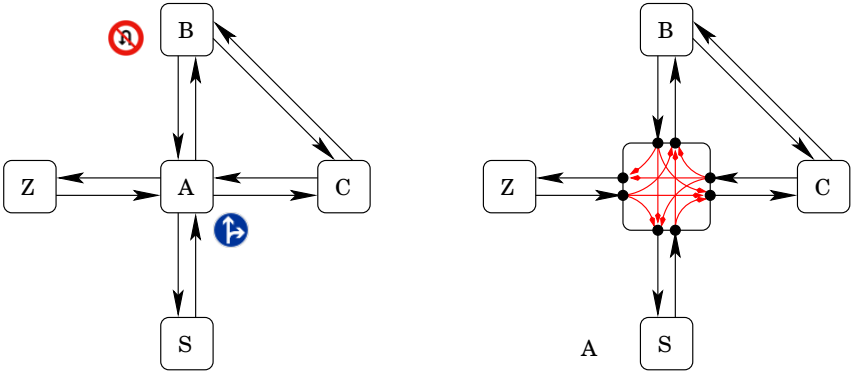
Berechnung kürzester Wege in gerichteten, azyklischen Graphen in Zeit $\Theta(\mathcal{V} + \mathcal{E})$:

- Berechne topologische Sortierung der Knoten.



- Durchlaufe die Knoten $u \in V$ entsprechend dieser Reihenfolge und relaxiere die Kanten von $v \in Adj[u]$: falls $d[v] > d[u] + c(u, v)$ dann $d[v] := d[u] + c(u, v)$

Wie können Zusatzinformationen wie Abbiegeinformationen bei der Suche nach kürzesten Wegen berücksichtigt werden? Problem: Wenn Abbiegeverbote einzuhalten sind, wie bei der Routenplanung, müssen ggf. Knoten mehrfach besucht werden:
$S \to A \to B \to C \to A \to Z$



mögliche Lösung: Knoten aufteilen und Kreuzung modellieren
Nachteil: Anzahl Knoten und Kanten steigt und damit die Laufzeit

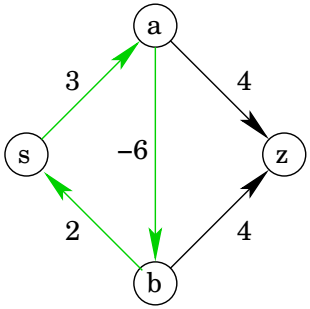
All pairs shortest paths:

Gegeben: Gewichteter Graph $G = (V, E, c)$ mit $c : E \to \mathbb{R}$.

Achtung: Negative Kantengewichte sind erlaubt!

Gesucht: $\forall u, v \in V$ der kürzeste Weg von u nach v.

Voraussetzung: G enthält keine negativen Kreise!



Im nebenstehenden Graphen hat der grün markierte Kreis (s, a, b) negative Länge.

Daher kann der kürzeste Weg von s nach z immer weiter verkleinert werden, indem der Kreis ein weiteres Mal durchlaufen wird.

In ungerichteten Graphen stellt bereits eine einzige negative Kante einen Kreis negativer Länge dar.

Idee: Sei d_{ij}^k die Länge eines kürzesten Weges von i nach j, der nur über Knoten mit Nummern kleiner gleich k läuft, wobei wir $V = \{1, 2, \ldots, n\}$ voraussetzen. Dann gilt:

- $d_{ij}^0 = c(i,j)$
- $d_{ij}^k = \min\left\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\right\}$

Algorithmus: dynamische Programmierung

$d[i,j] := \infty$ for each $i,j \in V, i \neq j$
$d[i,i] := 0$ for each $i \in V$
$d[i,j] := c(i,j)$ for each $(i,j) \in E$

for $k := 1$ to n do
  for $i := 1$ to n do
    for $j := 1$ to n do
      if $d[i,j] > d[i,k] + d[k,j]$
        $d[i,j] := d[i,k] + d[k,j]$

Laufzeit von Floyds Algorithmus: $\mathcal{O}(\mathcal{V}^3)$

Wie wird ein Kreis negativer Länge erkannt?

→ Ein negativer Wert d_{ii} auf der Diagonalen bedeutet, dass mindestens ein Kreis negativer Länge vorhanden ist.

Warum wird kein 3D-Array d benötigt?

→ Weil in einer Runde k immer nur auf die Werte der letzten Runde $k-1$ zugegriffen wird. Bei dynamischer Programmierung werden oft auch weiter zurück liegende Teilergebnisse benötigt, hier nicht.

Bei Graphen ohne negative Kanten könnte auch für jede Quelle ein Aufruf von Dijkstras Algorithmus durchgeführt werden. Das hätte die Laufzeit

$$\mathcal{O}(\mathcal{V} \cdot (\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))) = \mathcal{O}(\mathcal{V} \cdot \mathcal{E} + \mathcal{V}^2 \cdot \log(\mathcal{V}))$$

und wäre damit für dünne Graphen besser als die obige Laufzeit und für dichte Graphen nicht schlechter.

Transformiere Graph in einen Distanzgraphen[47]

Wie können wir die Kantengewichte modifizieren, sodass die relative Ordnung der Weglängen erhalten bleibt, aber keine negativen Kantengewichte existieren?

Sei $G = (V, E, c)$ ein Graph ohne Kreise negativer Länge. Mit Hilfe eines Graphen G' berechnen wir eine Funktion $k : E \to \mathbb{R}_0^+$, sodass $G_D = (V, E, k)$ ein Distanzgraph ist.

$V' := V \cup \{s\}$ ▷ füge neuen Knoten s hinzu
$E' = E \cup \{(s, v) \mid v \in V\}$ ▷ verbinde s mit allen Knoten
$c'(s, v) := 0$ for each $v \in V$ ▷ neue Kanten bekommen Gewicht 0
$c'(u, v) := c(u, v)$ for each $(u, v) \in E$ ▷ übernehme alte Kantengewichte

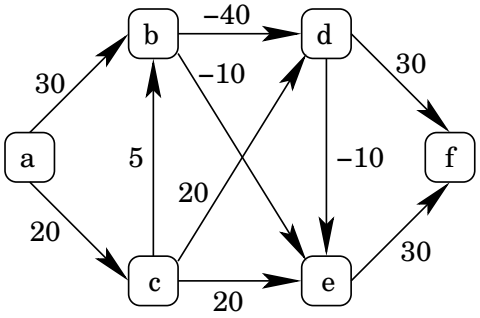
calculate single source shortest paths on G' from s by Bellman-Ford's algorithm

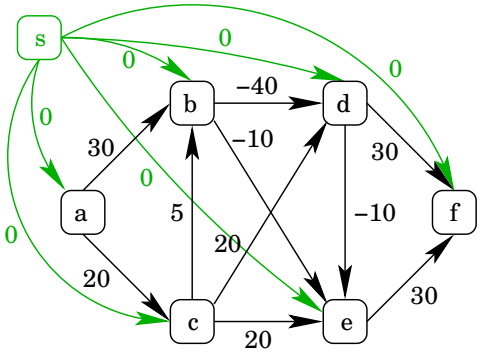
for each $(u, v) \in E$ do
    $k(u, v) := c(u, v) + \delta(s, u) - \delta(s, v)$

Schauen wir uns zunächst ein Beispiel an.

[47] Wir bezeichnen einen gewichteten Graphen $G = (V, E, c)$ mit $c : E \to \mathbb{R}_0^+$ als Distanzgraphen. Die Kanten haben also keine negativen Gewichte. Keine einheitliche Bezeichnung in der Literatur.

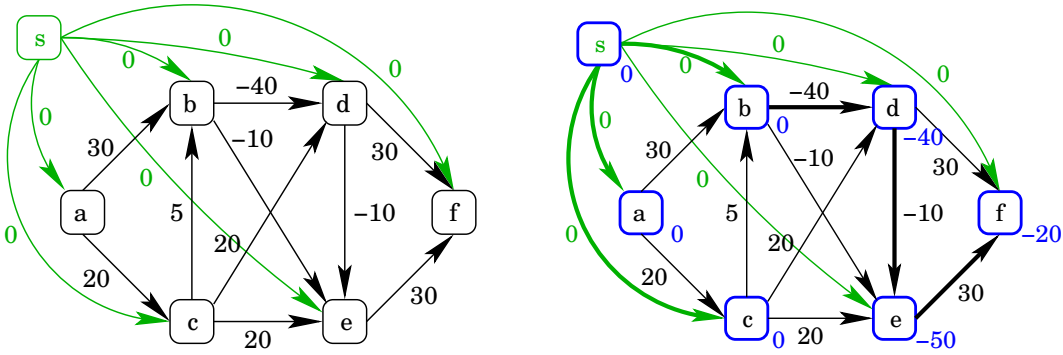
Transformiere Graph in einen Distanzgraphen





Füge neuen Knoten s hinzu und füge für jeden Knoten $v \in V$ eine Kante (s, v) mit Gewicht 0 hinzu.

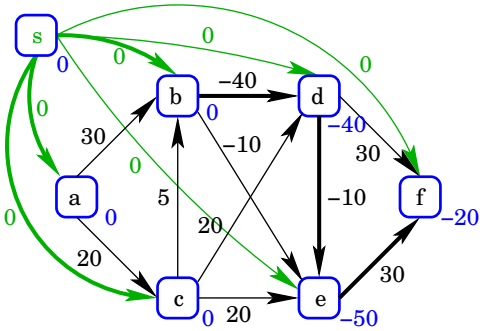
Transformiere Graph in einen Distanzgraphen

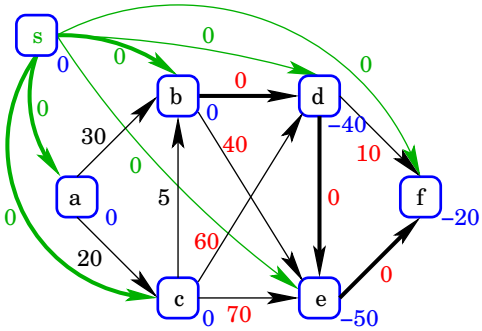


Berechne für den modifizierten Graphen mittels Algorithmus von Bellman-Ford die Länge der kürzesten Wege von s zu allen anderen Knoten.

Hier geht die Voraussetzung ein, dass G keine Kreise negativer Länge enthalten darf. Andernfalls liefert der Algorithmus von Bellman-Ford kein Ergebnis.

Transformiere Graph in einen Distanzgraphen





Ersetze für alle Kanten $e = (u, v) \in E$ das alte Kantengewicht durch $k(u, v) := c(u, v) + \delta(s, u) - \delta(s, v)$.

Der so modifizierte Graph[48] $G_D = (V, E, k)$ ist ein Distanzgraph.

Nun kann für jeden Knoten der Algorithmus von Dijkstra für G_D aufgerufen werden und so das All-Pairs-Shortest-Path-Problem für G gelöst werden.

[48] ohne den Knoten s und die dazu inzidenten Kanten

Korrektheit:

- Es gilt die Dreiecksungleichung:
  $\delta(s,v) \leq \delta(s,u) + c(u,v) \iff 0 \leq c(u,v) + \delta(s,u) - \delta(s,v)$
- Daher gilt für die modifizierten Kantengewichte
  $k(u,v) = c(u,v) + \delta(s,u) - \delta(s,v) \geq 0$ und daher ist G_D ein Distanzgraph.
- Für zwei Knoten $u, w \in V$ gilt: Die relative Ordnung der Längen aller Wege von u nach w bleibt erhalten. Sei $P = (v_1, v_2, \ldots, v_k)$ ein Weg von $u = v_1$ nach $w = v_k$, dann gilt $k(P) = c(P) + \delta(s,u) - \delta(s,w)$.

$$
\begin{aligned}
k(P) &= c(v_1,v_2) + \delta(s,v_1) - \delta(s,v_2) \\
&\quad + c(v_2,v_3) + \delta(s,v_2) - \delta(s,v_3) \\
&\quad + c(v_3,v_4) + \delta(s,v_3) - \delta(s,v_4) \\
&\ldots \quad + c(v_{k-1},v_k) + \delta(s,v_{k-1}) - \delta(s,v_k) \\
&= c(P) + \delta(s,u) - \delta(s,w)
\end{aligned}
$$

Transformiere Graph in einen Distanzgraphen

Laufzeit für das Erstellen des Distanzgraphen:

- Initialisierung: $\mathcal{O}(\mathcal{V} + \mathcal{E})$
- Bellman-Ford: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$
- Anpassen der Gewichte: $\mathcal{O}(\mathcal{E})$

$\Rightarrow$ insgesamt: $\mathcal{O}(\mathcal{V} \cdot \mathcal{E})$

Laufzeit für All-Pairs-Shortest-Path:

- $\mathcal{V}$ viele Aufrufe von Dijkstras Algorithmus

$\Rightarrow$ insgesamt: $\mathcal{O}(\mathcal{V} \cdot (\mathcal{E} + \mathcal{V} \cdot \log(\mathcal{V}))) = \mathcal{O}(\mathcal{V} \cdot \mathcal{E} + \mathcal{V}^2 \cdot \log(\mathcal{V}))$

Gegeben: Ein gerichteter Graph $G = (V, E)$.

Gesucht: Reflexiver, transitiver Abschluss $G^* = (V, E^*)$ von G.

In G^* existiert eine Kante (v_i, v_j) genau dann, wenn in G ein Weg zwischen v_i und v_j existiert. Sei auch hier $V = \{1, 2, \ldots, n\}$.

Algorithmus von Warshall:

$t_{ii} := 1$
$t_{ij} := 1$ for each $(v_i, v_j) \in E$

for $k := 1$ to n do
    for $i := 1$ to n do
        for $j := 1$ to n do
            $t_{ij} := t_{ij} \vee (t_{ik} \wedge t_{kj})$

$\rightarrow$ Laufzeit in $\mathcal{O}(\mathcal{V}^3)$ und Speicherplatz in $\mathcal{O}(\mathcal{V}^2)$.

Diese Laufzeit kann verbessert werden indem die innere Schleife nur dann ausgeführt wird, wenn es einen Weg von i nach k gibt:

```
t_ii := 1
t_ij := 1 for each (v_i, v_j) ∈ E
for k := 1 to n do
    for i := 1 to n do
        if (t_ik = 1) then
            for j := 1 to n do
                t_ij := t_ij ∨ (t_ik ∧ t_kj)
```

Übung 27. Zeigen Sie, dass die Laufzeit in $\mathcal{O}(\mathcal{V}^2 + \mathcal{V} \cdot |E^*|)$ liegt.

Für Graphen, die eine kleine transitive Hülle haben, ist diese Laufzeit besser als die obige. Sei G der folgende Graph: $a \to b \leftarrow c \to d \leftarrow e \to f \leftarrow g$

Dann enthält G^* gegenüber G nur zusätzlich die Schleifen $v \to v$ für alle $v \in V$.

Übung 28. Laut Definition existiert in G^* eine Kante (u, v) genau dann, wenn in G ein Weg zwischen u und v existiert.

Warum führen wir dann nicht einfach von jedem Knoten $u \in V$ eine Tiefensuche durch und fügen Kanten (u, v) in G^* ein für alle erreichbaren Knoten v?

Wäre die Laufzeit dann nicht nur $\mathcal{O}(\mathcal{V} \cdot (\mathcal{V} + \mathcal{E})) = \mathcal{O}(\mathcal{V}^2 + \mathcal{V} \cdot \mathcal{E})$ und der Speicherplatz in $\mathcal{O}(\mathcal{V} + \mathcal{E})$, weil Adjazenzlisten genutzt werden können?

Übersicht

Schwere und leichte Probleme

schwer: Hamilton-Kreis/Pfad

- Hamilton-Kreis: Eine Rundreise (also ein Kreis) in einem gegebenen Graphen $G = (V, E)$, die genau einmal durch jeden Knoten in V läuft und wieder am Startknoten endet.
- Hamilton-Pfad: Eine Route (also ein Weg) von einem gegebenen Knoten $s \in V$ zu einem gegebenen Knoten $t \in V$, $s \neq t$, die genau einmal durch jeden Knoten in $V - \{s, t\}$ läuft.

$\rightarrow$ kein Algorithmus mit polynomieller Laufzeit bekannt

leicht: Euler-Kreis/Pfad

- Euler-Kreis: Eine Rundreise durch einen gegebenen Graphen $G = (V, E)$, bei der jede Kante aus E genau einmal benutzt wird. $\rightarrow$ Königsberger Brückenproblem
- Euler-Pfad: Eine Route vom gegebenen Knoten $s \in V$ zum gegebenen Knoten $t \in V$, $s \neq t$, die genau einmal jede Kante aus E nutzt. $\rightarrow$ Haus vom Nikolaus

Das Euler-Kreis-Problem kann sehr effizient gelöst werden:

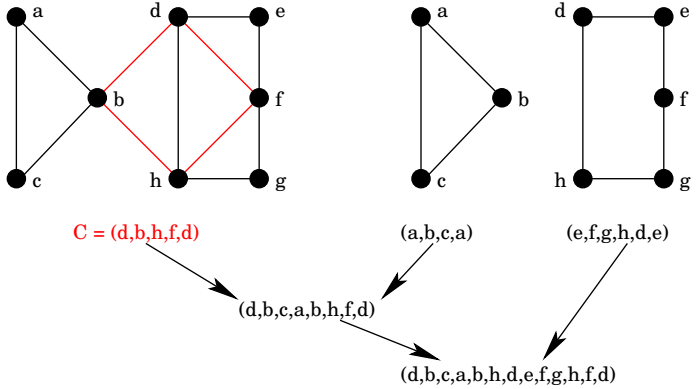
> Ein zusammenhängender Graph $G = (V, E)$ mit $|V| \geq 3$ besitzt genau dann einen Euler-Kreis, wenn der Knotengrad eines jeden Knotens $v \in V$ gerade ist.

Wenn ein Knoten bei einer Euler-Tour über eine Kante besucht wird, dann muss der Knoten über eine andere Kante wieder verlassen werden. Es ist also notwendig, dass der Knotengrad gerade ist.

Dass die Bedingung auch hinreichend ist, kann mittels vollständiger Induktion gezeigt werden. (nächste Folie)

Bestimme iterativ einen Euler-Kreis:

- Finde einfachen Kreis C in G und entferne alle Kanten des Kreises C aus G.
- Zerfällt der Graph in mehrere Zusammenhangskomponenten, dann bestimme in jeder ZSHK einen Euler-Kreis.
- Setze die Kreise zu einem Euler-Kreis zusammen.



schwer: Traveling-Salesperson-Problem (TSP)

- Gegeben: gewichteter, (un-) gerichteter, vollständiger[49] Graph $G = (V, E, c)$ und eine Zahl $k \in \mathbb{N}$.
- Gibt ist es einen Hamilton-Kreis in G, sodass die Summe der Kantengewichte auf diesem Kreis höchstens k ist?

$\rightarrow$ kein Algorithmus mit polynomieller Laufzeit bekannt

Unterscheide verschiedene Varianten bei TSP:

- symmetrisch $c(u, v) = c(v, u)$ vs. asymmetrisch: $c(u, v) \neq c(v, u)$
- metrisch: symmetrisch und Dreiecksungleichung $c(u, w) \leq c(u, v) + c(v, w)$ gilt

leicht: minimaler Spannbaum

- Entferne eine Kante aus Hamilton-Kreis $\rightarrow$ Spannbaum
- Kosten minimaler Spannbaum $\leq$ Kosten minimale Rundreise

[49] jeder Knoten ist mit jedem anderen Knoten durch eine Kante verbunden

Vehicle Routing Problem (VRP)

In vielen praktischen Anwendungen zur Tourenplanung müssen mehrere Touren für mehrere Fahrzeuge geplant werden.

- alle Start- und Zielpunkte liegen in einem Depot
- identische Fahrzeuge mit beschränkter Kapazität

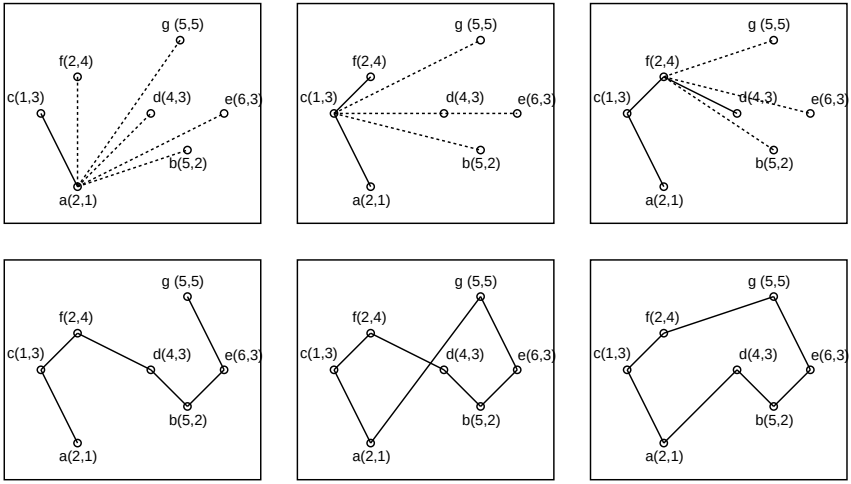
In der Praxis gibt es oft weitere Varianten mit zusätzlichen oder anderen Restriktionen:

- Waren werden von mehreren Depots ausgeliefert
- unterschiedliche Fahrzeugtypen mit unterschiedlicher Größenbeschränkung
- Ziele dürfen nur zu bestimmten Zeiten beliefert werden (Zeitfenster)
- Ziele müssen mehrfach angefahren werden (Leerung einiger Briefkästen mehrmals am Tag, Nachfüllen von Automaten)
- dynamische Kantengewichte (100 km/h von 6 bis 19 Uhr)

Für VRP sind bisher keine Algorithmen mit polynomieller Laufzeit bekannt.

Übersicht

Näherungsverfahren: Wähle vom letzten Punkt der Route den unbesuchten Punkt mit geringstem Abstand aus. → Nächste-Nachbar-Heuristik (nearest neighbor)



Vertex Cover[50]

Problem: Finde zu einem Graphen $G = (V, E)$ und einer Zahl $k \in \mathbb{N}$ eine Menge von Knoten $V' \subseteq V$, $|V'| \leq k$, sodass jede Kante $e \in E$ inzident zu einem Knoten in V' ist.

Brute-Force-Ansatz: Sei $G = (V, E)$ ein ungerichteter Graph mit n Knoten. Suche ein Vertex-Cover, indem alle $\binom{n}{k}$ vielen Teilmengen von V der Größe k betrachtet werden.

Laufzeit: $\mathcal{O}(\binom{n}{k} \cdot |G|) \subseteq \mathcal{O}(n^k \cdot |G|)$

$$\begin{aligned} \binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} \\ &\leq \frac{n \cdot n \cdot \ldots \cdot n}{\cancel{k!}} \leq n^k \end{aligned}$$

Die Laufzeit ist polynomiell, falls k konstant ist, also nicht zur Eingabe des Problems gehört. Für beliebiges k, also bspw. wenn k in Abhängigkeit der Größe des Graphen als $k = n/10$ gewählt wird, ist die Laufzeit nicht polynomiell.

[50] Vertex Cover ist NP-vollständig, siehe Richard M. Karp: Reducibility Among Combinatorial Problems. Complexity of Computer Computations, 1972.

Greedy-Heuristik:

$C := \emptyset$

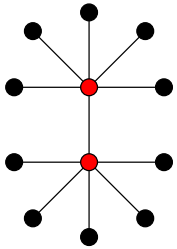
while es gibt noch Kanten in G **do**

sei v ein Knoten mit größter Anzahl Nachbarn

nimm v in C auf

entferne v und alle inzidenten Kanten aus G

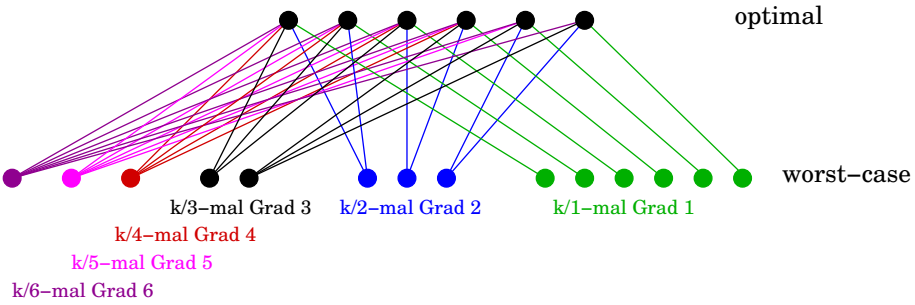
Laufzeit: $\mathcal{O}(n^2)$



Anmerkung: Es gibt Graphen G, für die die Greedy-Heuristik ein Vertex-Cover der Größe $\ln(k) \cdot k$ liefert, wobei k die Größe des kleinsten Vertex-Cover ist.

Der Fehler kann also $\ln(k)$ groß werden und ist nicht durch eine Konstante beschränkt.

Worst-Case-Graph mit $k = 6$ für die Greedy-Heuristik:



- Obiger bipartiter Graph hat ein Vertex-Cover der Größe $k = 6$.
- Im schlechtesten Fall werden die unteren Knoten ins VC aufgenommen, das die Größe $6 + 3 + 2 + 1 + 1 + 1 = 14$ hat.
- Fehler im Beispiel: $\frac{14}{6} \approx 2,33$
- harmonische Reihe: $\lfloor \frac{k}{1} \rfloor + \lfloor \frac{k}{2} \rfloor + \lfloor \frac{k}{3} \rfloor + ... + \lfloor \frac{k}{k} \rfloor = k \cdot \sum_{i=1}^{k} \lfloor \frac{1}{k} \rfloor \approx k \cdot \ln(k)$

Approximationsalgorithmus:

$C := \emptyset$
while es gibt noch Kanten in G **do**
    wähle irgendeine Kante $\{u, v\}$ von G zufällig aus
    nimm u und v beide in C auf
    entferne u und v und alle dazu inzidenten Kanten aus G

Anmerkung: Der Algorithmus liefert für alle Graphen ein Vertex-Cover, dass höchstens doppelt so viele Knoten enthält wie ein minimales Vertex-Cover:

- Sei F die Menge der ausgewählten Kanten, und sei $C = \{u, v \mid \{u, v\} \in F\}$ das berechnete Vertex-Cover.
- Jedes Vertex-Cover C', also insbesondere auch ein minimales, muss u oder v enthalten, da sonst die Kante $\{u, v\}$ nicht abgedeckt würde. Also gilt:

$$|C'| \geq 1/2|C| \iff |C| \leq 2|C'|$$

Übersicht

Backtracking

Idee: Erweitere schrittweise eine Teillösung zu einer Gesamtlösung, ähnlich wie bei Greedy, aber getroffene Entscheidungen werden ggf. wieder rückgängig gemacht.

Voraussetzungen:

- Lösung ist als Folge $a_1, a_2, \ldots, a_m$ darstellbar; m muss nicht bekannt sein.
- Jedes Element a_i wird aus einer endlichen Menge A_i gewählt.
- Es gibt einen effizienten Test zum Erkennen inkonsistenter Teillösungen.

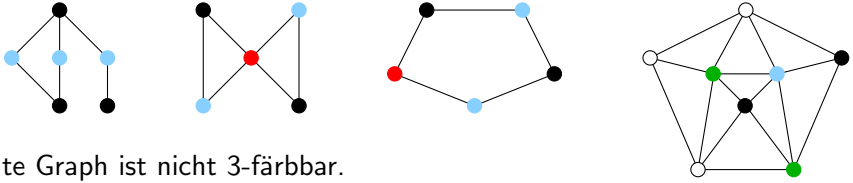
Anwendung bei Entscheidungs- bzw. Suchproblemen:

- Ist der Graph färbbar mit 4 Farben?
- Finde eine Lösung des n-Damen-Problems.
- Berechne eine Lösung für ein Sudoku.

Anwendung, wenn keine Berechnungsvorschrift bekannt ist:

- Finde einen Weg aus einem Labyrinth/Irrgarten.
- Finde eine Zugfolge bei Brett-Solitaire.

Sei $G = (V, E)$ ein ungerichteter Graph und sei $f : V \to \{1, 2, \ldots, k\}$ eine Abbildung der Knoten auf die ersten k natürlichen Zahlen. Dann nennt man f eine *k-Färbung* (k-coloring) des Graphen G, wenn $f(u) \neq f(v)$ für alle Kanten $\{u, v\} \in E$ gilt.



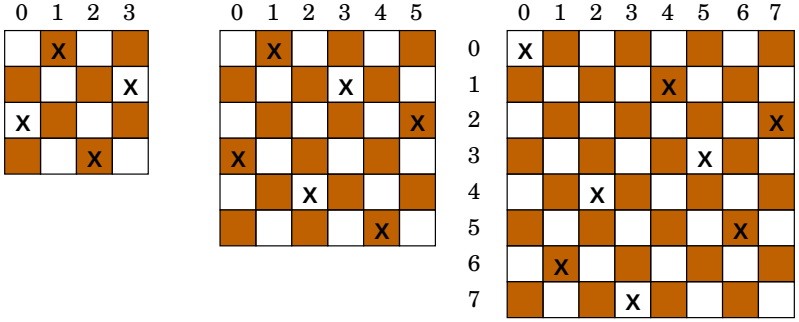
Der rechte Graph ist nicht 3-färbbar.

Varianten des Färbungsproblems:

- Entscheidungsvariante: Gibt es für einen gegebenen Graphen $G = (V, E)$ eine Färbung der Knoten mit höchstens k Farben? Dabei ist k Teil der Eingabe.
- Optimierungsversion: Wie viele Farben reichen aus, um den Graphen zu färben?
- *Suchvariante:* Bestimme eine Abbildung $f : V \to [k]$, sodass f eine k-Färbung von G ist und keine Färbung mit weniger Farben existiert.

[51] `https://en.wikipedia.org/wiki/Graph_coloring`

Gegeben ist ein Schachbrett mit $n \times n$ Feldern. Die Aufgabe besteht darin, n Damen so auf dem Schachbrett zu positionieren, dass sich keine der Damen gegenseitig schlagen.



Viele bekannte Spiele sind schwer zu lösen, siehe Folie 588. Das n-Damenproblem gehört nicht dazu, es ist leicht zu lösen, eignet sich aber sehr gut, um die Algorithmen für schwere Probleme zu implementieren.

[52] `https://de.wikipedia.org/wiki/Damenproblem`

wikipedia: In der üblichen Version ist es das Ziel, ein 9×9-Gitter mit den Ziffern 1 bis 9 so zu füllen, dass jede Ziffer in jeder Einheit (Spalte, Zeile, Block) genau einmal vorkommt – und in jedem der 81 Felder exakt eine Ziffer vorkommt.

Ausgangspunkt ist ein Gitter, in dem bereits mehrere Ziffern vorgegeben sind.

Das Spiel lässt sich sehr einfach auf größere Quadrate verallgemeinern: 16×16, 25×25, 36×36 usw.

Grundsätzlich sind auch Sudoku-Rätsel mit rechteckigen Blöcken möglich.

	8	1	9		7			6
	7	3		2				
	5			3		1		
	4			5				1
			6			8		
8			3					9
	9	2		8			5	4
7	3				1	9		
5		8						

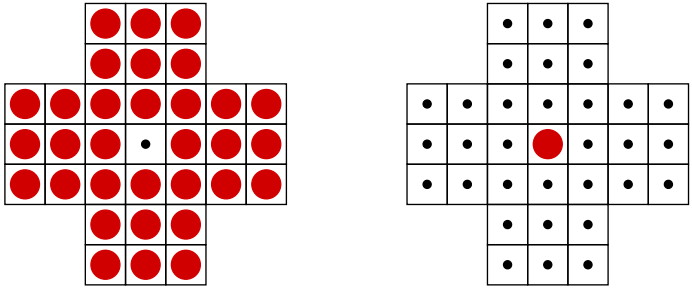
1	3				4
		6		2	
5			1		
	4				
		3	6		5
2					

[53] https://de.wikipedia.org/wiki/Sudoku

Brett-Solitaire[54]

Gegeben ist ein (englisches) Spielbrett in Kreuzform mit 33 Löchern, in denen 32 Stäbchen stecken. Das Loch in der Mitte ist leer. Man muss nacheinander die Stäbchen entfernen, indem man sie in waagerechter oder senkrechter Richtung überspringt.

Am Ende muss ein Stäbchen in der Mitte übrigbleiben.



[54] https://en.wikipedia.org/wiki/Peg_solitaire

- Sudoku ist NP-vollständig.[55]
- Brett-Solitaire ist NP-vollständig.[56]
- Mastermind[57] und Minesweeper[58] sind NP-vollständig.
- Sokoban[59] und Rush Hour[60] sind PSPACE-vollständig.

Eine schöne Übersicht über die Komplexität von Spielen gibt das Papier[61] von G. Kendall, A.J. Parkes, K. Spoerer: A Survey of NP-Complete puzzles. ICGA Journal. 2008 (31). 13-34.

[55] T. Yato, T. Seta: Complexity and completeness of findig another solution and its application to puzzles. IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences, 2003.

[56] R. Uehara, S. Iwata: Generalized Hi-Q is NP-complete. Trans. IEICE, 1990.

[57] J. Stuckman, G.-Q. Zhang: Mastermind is NP-Complete. CoRR abs/cs/0512049, 2005

[58] R. Kaye: Minesweeper is NP-complete. The Mathematical Intelligencer, 2000

[59] D. Dor, U. Zwick: SOKOBAN and Other Motion Planning Problems. Computational Geometry: Theory and Applications, 1999.

[60] G.W. Flake, E.B. Baum: Rush Hour is PSPACE-complete, or Why you should generously tip parking lot attendants. Theoretical Computer Science, 2002.

[61] `https://www.researchgate.net/publication/220174445_A_Survey_of_NP-Complete_puzzles`

Beispiele:

- Sudoku: Seien $p_1, \ldots, p_m$ die Felder, die nicht belegt sind.
  - Wähle $a_i \in \{1, \ldots, 9\}$ aus und teste, ob sich in der Zeile, Spalte oder inneres Quadrat, in dem sich Position p_i befindet, ein anderes Feld befindet, dass auch den Wert a_i hat. $\rightarrow$ inkonsistent
  - Gesamtlösung ist gefunden, wenn alle Felder $p_1, \ldots, p_m$ mit Werten belegt sind.
- n-Damen-Problem: Wir platzieren Dame i in Zeile i.
  - Wähle für Dame i eine Spalte $j \in \{1, \ldots, n\}$ aus und teste, ob sich eine bereits platzierte Dame in der gleichen Spalte oder Diagonalen befindet. $\rightarrow$ inkonsistent
  - Gesamtlösung ist gefunden, wenn alle Damen $1, \ldots, n$ platziert wurden.
- Labyrinth: Seien $k_1, \ldots, k_m$ die Kreuzungen bzw. Weggabelungen.
  - Wähle für die Weggabelung k_i eine Richtung a_i aus der dort möglichen Menge von Richtungen aus.
  - Teillösung ist inkonsistent, falls Weggabelung k_i bereits besucht wurde.
  - Gesamtlösung ist gefunden, wenn der Ausgang erreicht wurde.

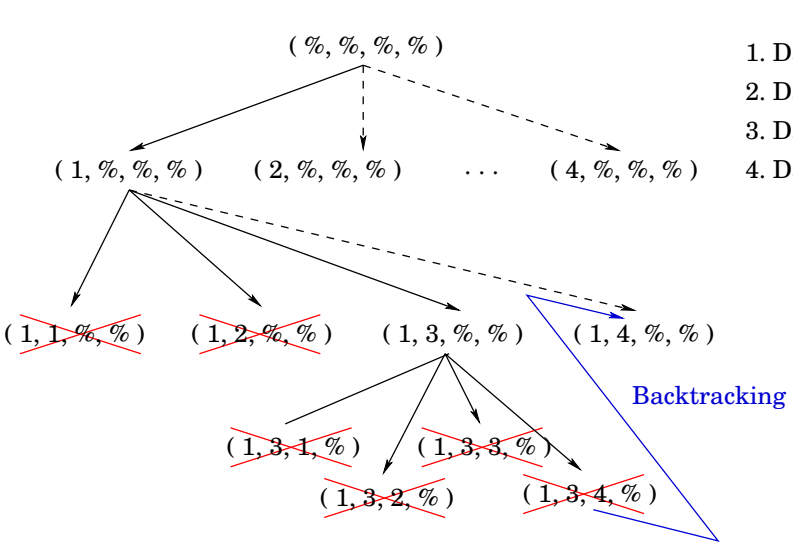
allgemeiner Rahmen:

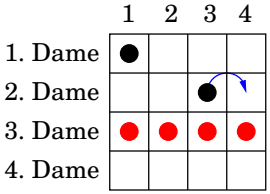
- Wähle initial ein Element $a_1 \in A_1$ als mögliche Teillösung.
- Solange $a_1, \ldots, a_i$ noch keine Gesamtlösung ist:
  - Wähle ein Element $a_{i+1} \in A_{i+1}$ aus.
  - Falls $a_1, a_2, \ldots, a_{i+1}$ konsistent ist, füge a_{i+1} der Teillösung hinzu, sonst wähle ein anderes Element aus A_{i+1} aus.
  - Falls kein Element a_{i+1} gefunden wird, das zu einer Gesamtlösung führt, gehe zurück (backtrack) und wähle a_i neu. (Falls kein a_i mehr gewählt werden kann, gehe zurück zu a_{i-1} usw.)
- Gib die Lösung $a_1, a_2, \ldots, a_m$ aus.

Vorteile:

- Es kann frühzeitig festgestellt werden, ob eine Teillösung noch zu einer Gesamtlösung erweitert werden kann. Ist das nicht der Fall, wird ein unnötiges Testen von Kombinationen verhindert.
- Ist universell einsetzbar.

4-Damenproblem:





Kann oft sehr einfach mittels rekursiver Funktion implementiert werden:

```
function BackTrack(i: Integer)
    if a_1, ..., a_i ist Gesamtlösung then
        gib Lösung aus
        return true
    gefunden := false;
    while nicht alle Elemente aus A_i sind untersucht do
        wähle nächstes Element a_i ∈ A_i aus
        gefunden := BackTrack(i + 1)
        if gefunden = true then
            return true
    mache Auswahl von a_i rückgängig
    return false
```

Wir gehen davon aus, dass sowohl das Array a als auch die Mengen A_i global zur Verfügung stehen.

Backtracking: *n*-Damenproblem

```
class Queens {
private:
    int _n, *_queens;

    bool check(int q1, int q2) const;
    int test(int q) const;

public:
    Queens(int n);
    ~Queens(void);
    bool solve(int);
    string toString(void);
};

Queens::~Queens(void) {
    delete[] _queens;
}
```

Backtracking: *n*-Damenproblem

```
Queens::Queens(int n) : _n(n), _queens(new int[n]) {
    for (int i = 0; i < n; i++)
        _queens[i] = 0;
}

string Queens::toString() {
    ostringstream os;

    for (int i = 0; i < _n; i++) {
        for (int j = 0; j < _n; j++)
            if (_queens[i] == j)
                os << "|Q";
            else os << "|-";
        os << "|" << endl;
    }
    return os.str();
}
```

Backtracking: *n*-Damenproblem

```
bool Queens::check(int a, int b) const {
    if (_queens[a] == _queens[b])
        return true;

    int dx = a - b;
    int dy = _queens[a] - _queens[b];

    if ((dx == dy) || (dx == -dy))
        return true;
    return false;
}
int Queens::test(int q) const {
    int res = 0;

    for (int i = 0; i < q; i++)
        if (check(i, q))
            res += 1;
    return res;
}
```

Backtracking: *n*-Damenproblem

```
// backtracking procedure (recursive)
bool Queens::solve(int q) {
    if (q == _n)
        return true;

    for (int i = 0; i < _n; i++) {
        _queens[q] = i;  // test each possible column i

        if (test(q) == 0) {
            if (solve(q+1))
                return true;
        }
    }
    return false; // backtrack
}
```

Backtracking: *n*-Damenproblem

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        cerr << "usage: " << argv[0]
             << " number-of-queens" << endl;
        return 1;
    }

    int n = atoi(argv[1]);
    cout << "try to solve " << n << "-queens problem" << endl;

    Queens q(n);

    if (q.solve(0))
        cout << q.toString() << endl;
    else cout << "could not solve the problem!" << endl;
}
```

Übung 29. Schreiben Sie ein Programm, das eine Lösung für ein Sudoku findet. Im Moodle-Kurs steht ein Rahmen-Programm zur Verfügung, das ein Sudoku-Rätsel aus einer Datei einliest.

Übung 30. Schreiben Sie ein Programm, das einen Weg aus einem Labyrinth findet. Im Moodle-Kurs steht ein Rahmen-Programm zur Verfügung, das ein Labyrinth aus einer Datei einliest.

Übung 31. Schreiben Sie ein Programm, das eine Lösung für das folgende Zahlenrätsel findet: `MEHR + MATHE + MACHT = FREUDE`

Dabei bedeuten gleiche Buchstaben gleiche Ziffern, verschiedene Buchstaben stehen für unterschiedliche Ziffern.

Übersicht

Branch-and-Bound

Findet eine Lösung wie Backtracking, gibt sich aber nicht mit der ersten gefundenen Lösung zufrieden, sondern setzt die Suche fort, um ggf. eine bessere Lösung zu finden.
$\rightarrow$ löse Optimierungsprobleme

- Berechne eine Färbung des Graphen mit möglichst wenigen Farben.
- Finde einen möglichst kurzen Weg aus einem Labyrinth/Irrgarten.
- Finde eine kürzeste Rundreise. (TSP: Traveling Salesperson Problem)

Idee: Branch-and-Bound besteht aus zwei Teilen.

- Branch: Verzweige wie Backtracking im Lösungsraum.
- Bound: Schneide bestimmte Zweige des Baumes ab, um den Rechenaufwand zu begrenzen.

$\rightarrow$ Falls eine Teillösung nicht mehr zu einer optimalen Lösung erweitert werden kann, beende die Suche im aktuellen Zweig und gehe zurück zu einer früheren Teillösung.

Für den Bound-Schritt werden in der Regel eine obere und eine untere Schranke für den Wert einer optimalen Lösung benötigt:

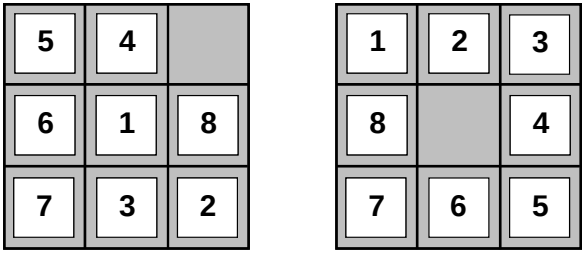
- obere Schranke: Wert der bisher gefundenen besten Lösung.
- untere Schranke: Abschätzung der noch mindestens erforderlichen Kosten für den restlichen Teil der Lösung.

Wir gehen im Folgenden von einem Optimierungsproblem aus, bei dem ein Minimum gesucht ist, z.B. eine möglichst kurze Rundreise.

- Sei $c(\mathcal{I})$ der Wert der bisher besten Lösung für die Eingabe $\mathcal{I}$.
- Sei $a_1, \ldots, a_i$ die aktuelle Teillösung.
  - Sei $g(a_1, \ldots, a_i)$ die Kosten der aktuellen Teillösung.
  - Sei $h(a_1, \ldots, a_i)$ die Schätzung der Restkosten, um das Ziel zu erreichen, wobei die tatsächlichen Kosten nicht überschätzt werden dürfen.

$\rightarrow$ Wenn $g(a_1, \ldots, a_i) + h(a_1, \ldots, a_i) \geq c(\mathcal{I})$ gilt, dann kann die Suche abgebrochen werden.

Beispiel 8-Puzzle[62]:

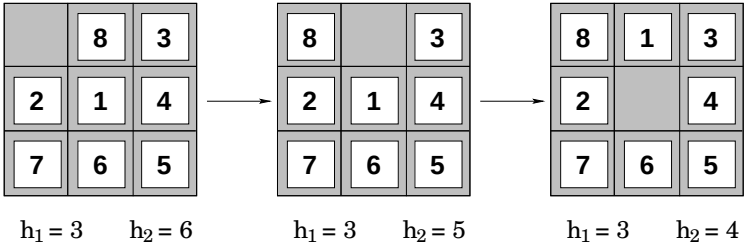


Schätzungen für die Restkosten bis zum Ziel vom Zustand s aus:

- $h_1(s)$: Anzahl Plättchen, die an falscher Stelle liegen. Hier: $h_1(s) = 7$
- $h_2(s)$: Summe der Entfernungen aller Plättchen zu dessen Zielposition, also die Summe der Manhatten-Distanzen. Hier: $h_2(s) = 2+3+3+2+4+2+0+2 = 18$

[62] aus S. Russell, P. Norvig: Artificial Intelligence – A Modern Approach. Prentice Hall, 2002.

Je besser die Restkosten abgeschätzt werden können, umso früher kann der Bound erfolgen und umso stärker kann die Laufzeit beschränkt werden.

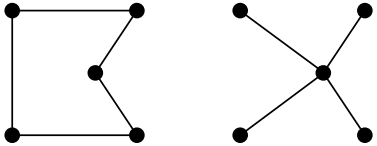


- Die heuristische Funktion h_2 differenziert stärker als h_1, d.h. h_2 kann Zustände unterscheiden, die von h_1 gleich bewertet werden.
- Eine heuristische Funktion ist um so brauchbarer, je mehr Zustände sie unterschiedlich bewertet.
- Eine heuristische Funktion, die alle Zustände gleich bewertet, ist unbrauchbar.

Ist keine Abschätzung der Restkosten möglich oder erfolgt der Bound immer erst sehr tief im Baum, dann ist die Laufzeit sehr hoch.

Abschätzung der Restkosten beim Traveling-Salesperson-Problem:

- Jede der noch nicht besuchten Städte muss besucht und auch wieder verlassen werden.
  - Addiere jeweils die Werte der zwei kostengünstigsten Kanten, die inzident zu einem noch unbesuchten Knoten sind.
  - Die Summe muss noch durch 2 dividiert werden, da jede Kante zweimal genutzt wird, um einen Knoten zu verlassen und um den nächsten zu besuchen.
- oder: Berechne für die noch nicht besuchten Knoten einen minimalen Spannbaum.



  - Lässt man auf einer Rundreise eine Kante weg, erhält man einen Spannbaum.
  - Die Kosten eines minimalen Spannbaums sind also höchstens so groß wie die Kosten einer minimalen Rundreise.

Übung 32. Schreiben Sie ein Programm, dass eine kürzeste Rundreise findet. Im Moodle-Kurs steht ein Rahmenprogramm zur Verfügung, außerdem einige Beispiele.

Übung 33. Schreiben Sie ein Programm, dass eine optimale Lösung des 0/1-Rucksackproblems findet. Achtung: Beim Rucksackproblem soll ein Maximum gefunden werden.

Eine obere Schranke für den maximal noch zu erzielenden Gewinn kann wie beim Bruchteil-Rucksackproblem (Seite 117) ermittelt werden.

Das Bruchteil-Rucksackproblem ist eine Relaxation des 0/1-Rucksackproblems: Lasse die Ganzzahligkeitsbedingung weg.

Übersicht

In der Klausur sind keine Hilfsmittel außer einem Kugelschreiber erlaubt, also keine Bücher, keine handgeschriebenen Zettel, kein Ausdruck der Vorlesungsunterlagen, kein Taschenrechner oder ähnliches!

Auf den folgenden Folien sind einige Aufgaben aus Klausuren[63] von Professor Dr. Kuhn von der Universität Freiburg aufgeführt. Die Fragen decken nicht den gesamten Stoff der Vorlesung ab und sind daher weder vollständig noch repräsentativ für die Klausur.

Zur Vorbereitung auf die Klausur ist es notwendig, die Inhalte der Vorlesung nachzuarbeiten und zu verstehen. Außerdem sollten die Übungsaufgaben von den Übungsblättern als auch die Aufgaben aus der Vorlesung bearbeitet werden.

[63] `https://ac.informatik.uni-freiburg.de/teaching/ss_22/ad-lecture.php`

Übung 34. Sei $f : \mathbb{R} \to \mathbb{R}$ eine stetige Funktion mit $f(0) = -1$ und $f(1) = 1$.

Beschreiben Sie einen Algorithmus, welcher in Zeit $\mathcal{O}(\log(n))$ eine Nullstelle von f bis auf $1/n$ genau findet. (Der Wert n ist also gegeben.) Ihr Algorithmus soll also eine Zahl x ausgeben, sodass f im Intervall $(x - 1/n, x + 1/n)$ eine Nullstelle hat.

Wir nehmen dabei an, dass das Auswerten von f an einer Stelle x (d.h. die Berechnung von $f(x)$) konstante Zeit beansprucht.

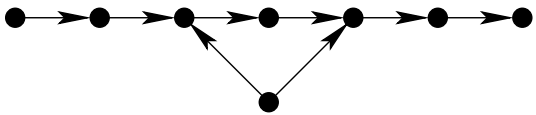
Begründen Sie die Korrektheit Ihres Algorithmus.

Übung 35. Nehmen wir an, es gäbe eine Vorrangwarteschlange H, deren Operationen folgende Laufzeit hätten: Sowohl `create` wie auch `insert`, `minimum` und `decreaseKey` haben konstante Laufzeit $\mathcal{O}(1)$, während ein Aufruf von `extractMin` eine Laufzeit von $\mathcal{O}(\log(n))$ hat (wenn n Elemente in der Datenstruktur gespeichert sind).

Welche asymptotische Laufzeit hätte Dijkstras Algorithmus (in Abhängigkeit der Kantenzahl m und der Knotenzahl n), wenn man H als zugrundeliegende Vorrangwarteschlange benutzt. Mit Begründung!

Übung 36.

(a) Wie viele topologische Sortierungen gibt es für den folgenden Graphen?



(b) Gegeben seien zwei gerichtete, kreisfreie Graphen $G = (V, E)$ und $G' = (V, E')$ mit gleicher Knotenmenge V, sodass G ein Teilgraph von G' ist, d.h. $E \subseteq E'$.

Auf welchem Graphen gibt es mehr Möglichkeiten, die Knotenmenge topologisch zu sortieren? Begründen Sie Ihre Antwort.

Übung 37. Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, der durch Adjazenzlisten gegeben ist.

Angenommen wir wissen, dass G genau einen Kreis enthält.

Beschreiben Sie einen Algorithmus mit Laufzeit $\mathcal{O}(|V|)$, welcher den Kreis von G ausgibt. Die Ausgabe sollte also die Form $v_1, v_2, \ldots, v_k, v_1$ haben, wobei $v_1, v_2, \ldots, v_k$ paarweise verschieden sind mit $\{v_i, v_{i+1}\} \in E$ für $i = 1, \ldots, k-1$ und $\{v_1, v_k\} \in E$.

Begründen Sie die Laufzeit.

Übung 38. Gegeben seien zwei Arrays A und B mit $|A| = m$ und $|B| = n$ und $m \leq n$. Die Einträge der Arrays seien natürliche Zahlen. Man möchte herausfinden, ob es eine Zahl gibt, die sowohl in A als auch in B vorkommt.

- Geben Sie einen möglichst effizienten Algorithmus für dieses Problem an, der Hashing nutzt. Sie können dabei annehmen, dass das Finden und das Einfügen eines Werts in einer Hashtabelle $\mathcal{O}(1)$ Zeit benötigt, solange der Load der Hashtabelle $\mathcal{O}(1)$ ist.
- Geben Sie einen möglichst effizienten Algorithmus für dieses Problem ohne Nutzung von Hashing an.

Analysieren Sie jeweils die Laufzeit als Funktion von m und n.

Übung 39. Gegeben sei ein ungerichteter, ungewichteter (aber nicht notwendigerweise zusammenhängender) Graph $G = (V, E)$. Wir möchten testen, ob dieser Graph *fast* ein Spannbaum ist.

Darunter verstehen wir, dass G aus einem Spannbaum und höchstens c vielen zusätzlichen Kanten besteht, wobei $c \in \mathcal{O}(1)$ eine gegebene Konstante ist.

Beschreiben Sie einen Algorithmus, der diesen Test in Zeit $\mathcal{O}(|V|)$ durchführt und begründen Sie die Laufzeit.

Übung 40. Gegeben seien k sortierte Arrays $A_1, \ldots, A_k$ mit insgesamt n Elementen. Wir fassen diese Arrays zu einem einzelnen sortierten Array A der Länge n zusammen.

(a) Eine Lösungsmöglichkeit ist folgender Algorithmus, wobei Merge die Merge-Operation wie im Mergesort-Algorithmus ist.

```
function SequentialMerge(A_1, ..., A_k)
    A := A_1
    for i = 2, ..., k do
        A := Merge(A, A_i)
    return A
```

Sei k ein Teiler von n und alle Arrays haben die Länge n/k. Geben Sie die Laufzeit von SequentialMerge als Funktion von n und k an. Begründen Sie Ihre Antwort.

(b) Ein Studierender schlägt stattdessen vor, alle Elemente auf beliebige Weise in ein Array der Länge n zu schreiben und dieses mit dem Mergesort-Algorithmus aus der Vorlesung zu sortieren. Ist dieses Vorgehen schneller oder langsamer als SequentialMerge? Begründen Sie Ihre Antwort.

Übung 41. Gegeben sei ein gerichteter Graph $G = (V, E)$, abgespeichert via Adjazenzlisten, und ein Knoten $v \in V$.

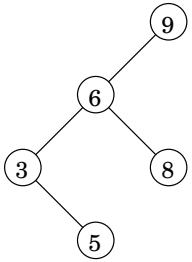
Geben Sie einen Algorithmus an, mit welchem Sie in möglichst kurzer Zeit die Menge $U = \{u \in V \mid \exists \text{ Weg von } u \text{ nach } v\}$ finden, also die Menge aller Knoten $u \in V$, von welchen aus ein Weg nach v existiert.

Übung 42. Gegeben seien k bereits sortierte Arrays $A_1, \ldots, A_k$ mit je m Sortierschlüsseln.

Beschreiben Sie einen Algorithmus, der ein sortiertes Array A der Größe $k \cdot m$ aus allen Werten der Arrays $A_1, \ldots, A_k$ berechnet und dabei $\mathcal{O}(k \cdot m \cdot \log(k))$ Vergleiche benötigt.

Sie dürfen voraussetzen, dass k eine Zweierpotenz ist, also $k = 2^\ell$ für ein $\ell \in \mathbb{N}$ gilt.

Übung 43. Gegeben sei der folgende binäre Suchbaum mit Wurzel 9:



(a) Geben Sie alle Folgen von Einfüge-Operationen an, welche diesen Baum erzeugen.

(b) Geben Sie jeweils die Besuchsreihenfolge der Knoten für die In-Order-, Pre-Order-, Post-Order- und Level-Order-Traversierung für den obigen Baum an.

Übung 44.

(a) Nennen Sie einen Vorteil von einfach verketteten Listen gegenüber doppelt verketteten Listen.

(b) Geben Sie für beide Datenstrukturen die Laufzeit der Lösche-Operation an, falls ein Zeiger auf das zu löschende Element gegeben ist. Begründen Sie Ihre Antworten.

Übung 45. Angenommen wir wissen, dass ein gegebener Algorithmus A für eine Eingabe der Größe n eine Laufzeit von $\Omega(n \cdot \log(n))$ hat.

Wir wissen außerdem, dass ein Algorithmus B für die selbe Eingabe eine Laufzeit von $\Theta(n^2)$ hat.

Können wir folgern, dass A eine bessere asymptotische Laufzeit hat als B? Begründen Sie Ihre Antwort.

Übung 46. Betrachten Sie die folgenden Funktionen auf natürlichen Zahlen:

$$f_1(n) = n - 1 \qquad f_2(n) = \begin{cases} \frac{n}{2} & \text{falls } 2 \mid n \\ n & \text{sonst} \end{cases} \qquad f_3(n) = \begin{cases} \frac{n}{3} & \text{falls } 3 \mid n \\ n & \text{sonst} \end{cases}$$

Dabei bedeutet $m \mid n$, dass die Zahl m den Wert n teilt, also dass ein $k \in \mathbb{N}$ existiert, sodass $k \cdot m = n$ ist.

Man betrachtet folgendes Problem: Für ein gegebenes $n \geq 1$ finde die minimale Anzahl an Anwendungen von f_1, f_2, f_3, die man benötigt, um den Wert 1 zu erhalten. Formal: Finde das minimale k, sodass es $i_1, \ldots, i_k \in \{1, 2, 3\}$ gibt mit
$f_{i_1}(f_{i_2}(\ldots(f_{i_k}(n))\ldots) = 1$.

Ein Studierender schlägt dazu folgenden Algorithmus vor:

```
function StepsToOne(n)
    s := 0
    while n > 1 do
        if 3 | n then
            n := n/3
        else if 2 | n then
            n := n/2
        else   n := n − 1
        s := s + 1
    return s
```

(a) Welches Algorithmendesign-Prinzip benutzt der obige Algorithmus?

(b) Zeigen Sie, dass der Algorithmus z.B. für $n = 10$ kein optimales Ergebnis liefert.

(c) Geben Sie einen Algorithmus in Pseudocode an, der das Prinzip der dynamischen Programmierung anwendet, um das obige Problem zu lösen. Analysieren Sie die (asymptotische) Laufzeit Ihres Algorithmus.