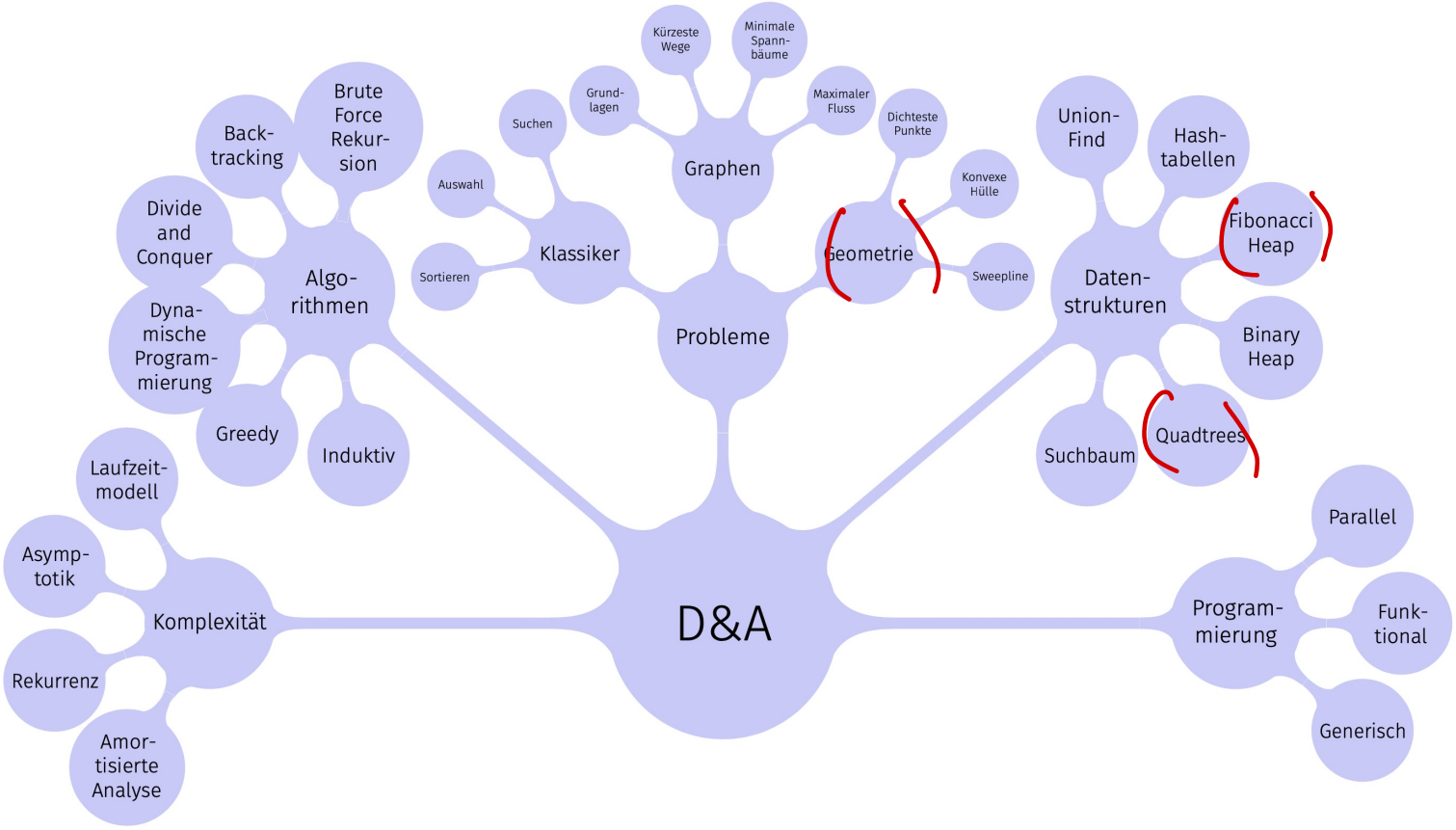
PVK

Prüfungsvorbereitungskurs
Datenstrukturen & Algorithmen
Frühlingssemester 2023


D&A
Algo-
rithmen
Brute
Force
Rekur-
sion
Back-
tracking
Divide
and
Conquer
Dyna-
mische
Program-
mierung
Greedy
Induktiv
Komplexität
Laufzeit-
modell
Asymp-
totik
Rekurrenz
Amor-
tisierte
Analyse
Probleme
Klassiker
Suchen
Auswahl
Sortieren
Graphen
Kürzeste
Wege
Minimale
Spann-
bäume
Grund-
lagen
Maximaler
Fluss
Geometrie
Dichteste
Punkte
Konvexe
Hülle
Sweepline
Daten-
strukturen
Union-
Find
Hash-
tabellen
Fibonacci
Heap
Binary
Heap
Quadtrees
Suchbaum
Program-
mierung
Parallel
Funk-
tional
Generisch

Inhaltsverzeichnis

1 Prüfungsanalyse **1**

Prüfungsthemata . . . 1

Prüfungsschema . . . 1

Aufgabe 1: Verschiedenes . . . 1

Aufgabe 2: Asymptotik . . . 2

Aufgaben 3, 4: Theorieaufgaben . . . 2

Aufgaben 5, 6, 7: Programmieraufgaben . . . 2

2 Mathematische Grundlagen **3**

Asymptotisches Verhalten . . . 3

Grenzwerte . . . 3

Dominanz von Funktionen . . . 4

Induktionsbeweise . . . 4

Rekursionsgleichungen . . . 4

Expansion und Vereinfachung . . . 4

Mastertheorem . . . 6

Laufzeitanalyse von Code-Snippets . . . 6

Amortisierte Analyse . . . 7

Aggregatsanalyse . . . 8

Kontomethode . . . 8

Potentialmethode . . . 8

3 Grundlagen des Programmierens **9**

Token `auto` . . . 9

Tokens `template` und `typename` . . . 10

Funktionen, Funktoren und Lambdas . . . 11

Funktionen . . . 11

Funktoren . . . 11

Lambda-Funktionen . . . 12

4 Abstrakte Datentypen und Datenstrukturen **14**

Abstrakte Datentypen . . . 14

Stack . . . 14

Queue . . . 15

Wörterbücher . . . 15

Mengen . . . 15

Datenstrukturen . . . 15

Bäume . . . 15

Union-Find . . . 20
Hashtabelle . . . 20

5 Grundlegende Algorithmen **22**
Suchalgorithmen . . . 22
Lineare Suche . . . 22
Binäre Suche . . . 23
Auswahlalgorithmen . . . 23
Sortieralgorithmen . . . 23
Bubblesort . . . 23
Selectionsort . . . 24
Insertionsort . . . 24
Mergesort . . . 25
Quicksort . . . 25
Heapsort . . . 26
Radixsort . . . 27
Bucketsort . . . 27

6 Dynamische Programmierung vs. Gierige Algorithmen **28**
Memoisation . . . 28
Top-Down vs. Bottom-Up . . . 28
Top-Down . . . 28
Bottom-Up . . . 28
Dynamische Programmierung . . . 29
Lösungsschema . . . 29
Gierige Algorithmen (*Greedy Algorithms*) . . . 29
Huffman-Codierung . . . 30
Kruskals Algorithmus . . . 30

7 Paralleles Programmieren **32**
Amdahl vs. Gustafson . . . 32
Amdahls Gesetz . . . 32
Gustafsons Gesetz . . . 33
Paralleles Programmieren . . . 33
Multithreading mit `std::threads` . . . 33
Zugänge verwalten mit `std::mutex` . . . 34
Race Conditions . . . 35
Locks . . . 35
Threads wecken mit `std::condition_variable` . . . 36

Vorwort

Dieses Skript basiert auf den Handouts der Vorlesung Datenstrukturen und Algorithmen aus den Frühlingssemestern 2020 und 2023, sowie Teilen des Skripts für den Prüfungsvorbereitungsworkshop des VIS (Simone Guggiari, Leonardo Del Giudice, François Hublet). Es wurde von Christoph Grötzbach angefangen und durch Adel Gavranović ergänzt, verschönert und auf das Frühlingssemester 2023 ausgelegt.

Massgeblich beigetragen haben

- Christoph Grötzbach
- Adel Gavranović

Kapitel 1

Prüfungsanalyse

Warnung

Dieser Abschnitt beruht auf Spekulation und Extrapolation von Prüfungen aus den vergangenen Jahre. **An dieser Stelle sei ausdrücklich bemerkt, dass dies *keine* Garantie ist, dass die folgenden Themata und Aufgabentypen in zukünftigen Prüfungen vorkommen werden.** Alte Prüfungen zu lösen ist eine sehr hilfreiche Strategie um sich für kommende Prüfungen vorzubereiten, bietet aber *keine* Garantie, dass kommende Prüfungen nicht doch *komplett* anders ausfallen werden.

Prüfungsthemata

Analysiert man die Prüfungen der letzten Jahre, findet man die folgen Aufgabenthemata sehr häufig vor. Jedes Thema kann in jedem Aufgabentypen vorkommen.

- **Asymptotik**
- **Graphen**
  - **allgemeine Algorithmen**
  - **kürzeste Wege**
  - **minimale Spannbäume**
- **Flussnetzwerke**
- **Dynamic Programming**
- **Parallele Programmierung**
  - **Amdahl & Gustafson**
  - **Multithreading in C++**

Prüfungsschema

Analysiert man die Prüfungen der letzten Jahre, findet man das folgede Aufgabenschema sehr häufig vor. Jedes Thema kann in allen Aufgabentypen vorkommen.

Aufgabe 1: Verschiedenes

Die erste Aufgabe besteht meistens aus sehr vielen kleineren Aufgaben, die Themata aus dem gesamten Semester abfragen. Von Hashing über Sortieralgorithmen bishin zu selbstbalancierenden Bäumen könnte hier alles vorkommen. Für die Theoriefragen hilft eine Zusammenfassung, tiefergehendes Wissen bleibt aber unabdigbar. Bei den praktischeren Fragen hilft viel Übung.

Aufgabe 2: Asymptotik

In diesem Abschnitt werden oft folgende drei Aufgabentypen gestellt:

- Rangliste von Laufzeiten in der Landau-Notation bestimmen
- Rekurrenzgleichungen lösen
- Laufzeiten von Code-Snippets bestimmen

Hier helfen eine stützende Zusammenfassung, viel Übung und gute Analysis-Skills.

Aufgaben 3, 4: Theorieaufgaben

Oft sind zwei Aufgaben eher theoretischer Natur und prüfen die Fähigkeit, Konzepte auf konkrete Situation anzuwenden oder Algorithmen stumpf von Hand auszuführen. Oft wird hier ein (idealisiertes) Problem geschildert und man hat die Aufgabe dieses auf ein Model (Graph, Flussnetzwerk, Baum) abzubilden und dann mit einem der erlernten Konzepte (Dynamic Programming, Algorithmen) zu lösen, oder zumindest ausführlicher zu beschreiben, wie ein Konzept aus dem Kurs zur Lösung dieses Problems angewandt werden kann.

Aufgaben 5, 6, 7: Programmieraufgaben

Hier werden sowohl theoretisches Verständnis der Konzepte als auch praktisches Programmieren abgefragt. Hier hilft es enorm, die wöchentlichen Programmieraufgaben auf `[code]`expertzu lösen und die jeweiligen Musterlösungen gut zu studieren.

Kapitel 2

Mathematische Grundlagen

Relevante Prüfungsaufgaben

- ☐ Prüfung 13.02.2023, Aufgabe 2
- ☐ Prüfung 25.08.2022, Aufgabe 2
- ☐ Prüfung 02.02.2022, Aufgabe 2
- ☐ Prüfung 27.08.2021, Aufgabe 2
- ☐ Prüfung 03.02.2021, Aufgabe 2
- ☐ Prüfung 21.08.2020, Aufgabe 2
- ☐ Prüfung 29.01.2020, Aufgabe 2

Asymptotisches Verhalten

Asymptotisches Verhalten beschreibt, wie ein Algorithmus sich mit wachsender Problemgrösse $n \to \infty$ verhält. Dabei werden zwischen drei Kategorien unterschieden.

- **Obere Grenze**: Ab einer gewissen Datengrösse n_0 liegt f unterhalb von g.

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

- **Untere Grenze**: Ab einer gewissen Datengrösse n_0 liegt f oberhalb von g.

$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

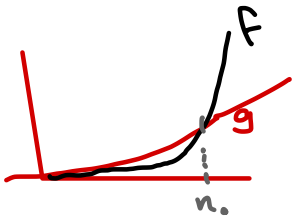
- **Enge Grenze**: Ab einer gewissen Datengrösse n_0 liegt f zwischen $\mathcal{O}(g)$ und $\Omega(g)$.

$$\Theta(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

Grenzwerte

Um das asymptotische Verhalten zu berechenen, ist es hilfreich, sich mit Grenzwerten auszukennen. Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ so dass der Grenzwert $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existiert. Dann gilt (in kompakter in pseudomathematischer Notation):

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \begin{cases} 0 & \implies f \in \mathcal{O}(g) \\ C \in \mathbb{R}^+ & \implies f \in \Theta(g) \\ \infty & \implies f \in \Omega(g) \end{cases}$$



Dominanz von Funktionen

Muss man Laufzeiten der Wachstumsgeschwindigkeit nach ordnen, hilft es folgendes auf der Zusammenfassung zu haben. Hier gilt $n \to \infty$ und $c \in \mathbb{R}^+$:

$$a < \log\log n < \log^b n < \sqrt{n} < n < n\log n < n^c < c^n < n! < n^n$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \in \Theta(n^k)$$

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

$$n! \in \mathcal{O}(n^n)$$

Ausserdem hilfreich sind foldenge Summen und ihr asymptotisches Verhalten

$$\sum_{i=0}^{n} i = \frac{n \cdot (n+1)}{2} \qquad \in \mathcal{O}(n^2)$$

$$\sum_{i=0}^{n} i = \sum_{i=0}^{n} (n-i) \qquad \in \mathcal{O}(n^2)$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+2)}{6} \in \mathcal{O}(n^3)$$

$$\sum_{i=0}^{n^2} i \qquad \in \mathcal{O}(n^4)$$

$$\sum_{i=0}^{n} i^k \qquad \in \mathcal{O}(n^{k+1})$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \qquad \in \mathcal{O}(n^k)$$

Induktionsbeweise

Diese Art von Beweis kennt man bereits aus der Vorlesungen Diskrete Mathematik und Analysis. In dieser Vorlesung wurden diese Beweise hauptsächlich zur Herleitung der Laufzeit eines Algorithmus anhand einer Rekursionsformeln verwendet. Oft hängt hier der Induktionsschritt von Potenzen ab, weshalb im folgenden das Schema für solch einen Schritt gewählt wurde.

1. Induktionshypothese: definiere $T(n)$.
2. Basisfall: Beweise $T(1)$
3. Induktionsschritt: Beweise $T(2^k) \implies T(2^{k+1})$.

Oft muss man an der Prüfung argumentieren, dass der Basisfall erreicht wird. Dies macht man am besten, indem man argumentiert, dass eine gewisse Grösse *monoton fällt/steigt*, bis der Basisfall erreicht ist.

Rekursionsgleichungen

Expansion und Vereinfachung

Eine häufige Prüfungsaufgabe ist das Auflösen von Rekursionsgleichungen. Sie kamen bei praktisch allen Prüfungen der letzten Semester vor und lassen sich mit Übung gut meistern.

Beispiel "Rekursionsgleichung mittels Expansion und Vereinfachung"

Die Rekursionsgleichung sehen meistens etwa wie folgt aus:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \frac{n}{2} + 1, & n > 1 \\ 3, & n = 1 \end{cases}$$

Die Grundstrategie ist, zu expandieren und umzustellen bis man ein Muster erkennt und das dann bis zum Basisfall weiterzieht, wo man dann mittels Analysis vereinfacht.

$n = 2^k$

$$\begin{aligned}
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + \frac{n}{2} + 1 && | \; T\left(\frac{n}{2}\right) \text{ expandieren} \\
&= 2 \cdot \left[2 \cdot T\left(\frac{\frac{n}{2}}{2}\right) + \frac{\frac{n}{2}}{2} + 1\right] + \frac{n}{2} + 1 && | \; \text{umstellen} \\
&= 2 \cdot \left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{4} + 1\right] + \frac{n}{2} + 1 && | \; T \text{ isolieren} \\
&= 2 \cdot 2 \cdot \left[T\left(\frac{n}{4}\right)\right] + \frac{n}{2} + 2 + \frac{n}{2} + 1 && | \; \text{umstellen} \\
&= 2^2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot \frac{n}{2} + 2 + 1 && | \; \text{umstellen} \\
&= 2^2 \cdot T\left(\frac{n}{4}\right) + 2 \cdot \frac{n}{2} + 2 + 1 && | \; T\left(\frac{n}{4}\right) \text{ expandieren} \\
&= 2^2 \cdot \left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n}{2^2} + 1\right] + 2 \cdot \frac{n}{2} + 2 + 1 && | \; \text{umstellen \& isolieren} \\
&= 2^3 \cdot T\left(\frac{n}{16}\right) + 3 \cdot \frac{n}{2} + 4 + 2 + 1 && | \; \text{Muster erkennen} \\
&\vdots && | \; \text{höchstens } \log_2(n) \text{ Mal} \\
&= 2^{\log_2(n)} \cdot T(1) + \log_2(n) \cdot \frac{n}{2} + \sum_{i=0}^{\log_2(n)} 2^i && | \; T(1) = 3 \\
&= n \cdot 3 + \frac{n \log_2(n)}{2} + \sum_{i=1}^{\log_2(n)} 2^i && | \; \sum_{i=1}^{\log_2(n)} 2^i = 2n - 1 \\
&= n \cdot 3 + \frac{n \log_2(n)}{2} + n - 1 && | \; \text{vereinfachen} \\
&= 4 \cdot n + \frac{n \log_2(n)}{2} - 1
\end{aligned}$$

Beispiel "Rekursionsgleichung mit Annahme $n = 2^k$"

Es kann helfen, eine neue (hoffentlich einfachere) Rekursionsgleichung zu definieren, oft mit der Annahme, dass $n = 2^k$. Hat man folgende Rekursionsgleichung gegeben

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + a \cdot n, & n > 1 \\ c, & n = 1 \end{cases}$$

kann man mit der Annahme, dass $n = 2^k$ folgende neue Rekursionsgleichung definieren

$$\tilde{T}(k) := T(2^k) = \begin{cases} 2\tilde{T}(k-1) + a \cdot 2^k, & k > 0 \\ c, & k = 0 \end{cases}$$

und die dann mit den oben beschriebenen Mitteln auflösen.

Mastertheorem

Einige Rekursionsgleichungen lassen sich mit dem Mastermethode angehen, die besagt, dass wenn eine Rekursionsgleichung einer folgender Form entspricht

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n), \quad (a \geq 1, b > 1)$$

a : Anzahl Unterprobleme

$1/b$: Aufteilungsquotient

$f(n)$: Divisions- und Summationskosten

und gewissen Anforderungen genügt, man das asymptotische Verhalten dieser Rekursionsgleichung leicht bestimmen kann. Man verwendet folgendes Schema:

1. Forme Rekursionsgleichung in oben gezeigte Form
2. Berechne Fallunterscheidungszahl $K := \log_b a$
3. Wende Fallunterscheidung an ($\varepsilon > 0$):

Die Fallunterscheidung ist hier erneut in pseudomathematischer Form gegeben:

$$f(n) \in \begin{cases} \mathcal{O}\left(n^{K-\varepsilon}\right) & \Longrightarrow T(n) \in \Theta\left(n^K\right) \\ \Theta\left(n^K\right) & \Longrightarrow T(n) \in \Theta\left(n^K \log(n)\right) \\ \Omega\left(n^{K+\varepsilon}\right) \wedge af(\frac{n}{b}) \leq cf(n), 0 < c < 1 & \Longrightarrow T(n) \in \Theta(f(n)) \end{cases}$$

Das heisst, wenn $f(n)$ in der jeweiligen Menge enthaltn ist (und im Fall von Ω noch eine weitere Bedingung erfüllt), gilt für die ursprüngliche Rekursionsgleichung das, was jeweils rechts steht.

Laufzeitanalyse von Code-Snippets

Eine weitere sehr beliebte Aufgabenart an den Prüfungen ist die Laufzeitanalyse von Code-Snippets. Die Strategie hierbei ist es, aus dem Code-Snippet eine Rekursionsgleichung herzuleiten und diese dann

aufzulösen. Nach ein paar gelösten Aufgaben ist das sogar nicht mehr nötig und man schaut sich nur noch an, wie viele Interationen die `for`-Schleifen erzeugen und leitet sich dann direkt die Laufzeit mittels Multiplikation her.

Beispiel "Anzahl Funktionsaufrufe"

Bestimmme die Anzahl der Funktionsaufrufe von `f()` in den Folgenden Code-Snippets. Man kann davon ausgehen, dass $n = 2^k$ für ein $k \in \mathbb{N}$.

```
void g(int n){
    if(n>0){
        f();
        g(n/2);
        f();
        g(n/2);
    }
}
```

Hier sieht man schnell, dass die $M(n)$, also die Anzahl der Aufrufe von `f(n)` in Abhängigkeit von n, die folgende Form hat

$$\underbrace{2 \cdot M\Big(\overbrace{\frac{n}{2}}^{\text{weil g jeweils mit n/2 aufgerufen wird}}\Big)}_{\text{weil g 2 Mal aufgerufen wird}} + \underbrace{2}_{\text{weil f() jeweils 2 mal pro g() aufgerufen wird}}$$

und sich als Rekursionsgleichung recht leicht auflösen lässt

$$\begin{aligned} M(n) &= 2 \cdot M\left(\frac{n}{2}\right) + 2 \\ &= 4 \cdot M\left(\frac{n}{4}\right) + 4 + 2 \\ &= 8 \cdot M\left(\frac{n}{8}\right) + 8 + 4 + 2 \\ &= \dots \\ &= n + \frac{n}{2} + \frac{n}{4} + \dots + 2 \\ &= \sum_{i=1}^{n} \\ \implies M(n) &\in \Theta(n) \end{aligned}$$

und damit liegt die Laufzeit des Code-Snippets in Abhängigkeit von n in $\Theta(n)$

Amortisierte Analyse

Die Amortisierte Analyse dient der Laufzeitberechnung. Genauer gesagt wird die Durchschnittliche Laufzeit jeder Operation im schlechtesten Fall betrachtet. In der Vorlesung wurde dies anhand eines Stacks erklärt.

Aggregatsanalyse

Hier berechnet man eine Obere Schranke für die Gesamtzahl der Elementaroperationen und teilt durch die Anzahl der Operationen.
Bei n Operationen können maximal n Elemente auf den Stack gelegt und dadurch maximal n Elemente davon entfernt werden. Die Gesamtkosten betragen dann

$$\sum_{i=1}^{n} \text{Kosten}(op_i) \leq 2n$$

und somit sind die Kosten pro Operation, die **amortisierten Kosten** $\leq 2 \in \mathcal{O}(1)$

Kontomethode

Bei der Kontomethode werden Münzen auf ein Konto eingezahlt. Jede Elementaroperation kosten eine Münze, welche irgendwann eingezahlt werden muss.
Für jede Operation op_k werden a_k Münzen auf Konto A eingezahlt: $A_k = A_{k-1} + a_k$. Das Guthaben wird verwendet, um die realen Kosten t_k bezahlen zu können. A muss zu jeder Zeit genügend Münzen haben, um die Kosten der Operation op_k bezahlen zu können $A_k - t - K > 0$.
a_k sind schliesslich die amortisierten Kosten der Operation op_k.
Beim Stack würde jede *push* und *pull* Operation 1Fr. kosten. Für jedes *push* zahlt man nun 2Fr. ein, einer davon für die kommenden *pops*. Damit hat man $a_k = 2$ für *push* und $a_k = 0$ für *pop*. Folglich ist $a_k \leq 2 \forall k$, also hat man konstante amortisierte Kosten.

Potentialmethode

Das Potential Φ_i ist der Zustand der Datenstruktur zur Zeit i. Es soll zum Ausgleichen teurer Operationen verwendet werden, also muss es bei günstigen, häufigen Operationen erhöht werden während teure, seltene Operationen es senken.
Die amortisierten Kosten sind definiert als $a_i = t_i + \Phi_i - \Phi_{i-1}$, also den realen kosten plus die Potentialdifferenz.
Beim Stack ist die Potentialfunktion Φ_i = Anzahl Elemente auf denm Stack.

- **push** hat $t_i = 1$ und steigert das Potential um 1, somit gilt $a_i = 2$.
- **pop** hat ebenfalls $t_i = 1$, entzieht dem Stapel aber ein Element und reduziert somit das Potential um $1 \Rightarrow a_i = 0$.

Aufgabe "Amortisierte Analyse"

Betrachte ein Array das bei Bedarf dynamisch wachsen kann. Wir nehmen an, dass Werte immer einzeln eingefügt werden. Sind mehr als n Elemente im Array gespeichert, wird ein neues Array der Länge $k > n$ in k Schritten erstellt, in das die Werte kopiert werden. Die Kopie eines Eintrags passiert in konstanter Zeit.

Beschreibe, wie man k wählen sollte, damit jede Einfügeoperation konstante amortisierte Kosten hat und somit n Elemente in $\Omega(n)$ eingefügt werden können.

Kapitel 3

Grundlagen des Programmierens

Relevante Prüfungsaufgaben

- ☐ Prüfung 13.02.2023, Aufgaben 5, 6, 7
- ☐ Prüfung 25.08.2022, Aufgaben 5, 6, 7
- ☐ Prüfung 02.02.2022, Aufgaben 5, 6, 7
- ☐ Prüfung 27.08.2021, Aufgaben 5, 6, 7
- ☐ Prüfung 03.02.2021, Aufgabe 7
- ☐ Prüfung 21.08.2020, Aufgabe 7
- ☐ Prüfung 29.01.2020, Aufgabe 7

Als erstes werden wir die grundlegendsten Erweiterungen euer C++ -Kenntnisse repetieren. Das meiste davon braucht ihr auch in den nächsten Semestern, also empfehle ich euch hier sehr gut aufzupassen.

Token `auto`

Das Keyword `auto` wird anstelle von Typen (wie `int`, `bool`, etc.) benutzt, insbesondere dann, wenn der Typ einer Variable nur sehr umständlich oder gar nicht ermittelbar ist, wie zum Beispiel beim Rückgabetypen einer komplizierten Funktion oder wenn man mit Containern (wie `std::vector`) arbeitet. Man überlässt es quasi dem Compiler festzustellen, welcher Typ erwartet wird. Meistens trifft man `auto` in bereichsbasierten `for`-Schleifen oder Lambdas an. Anstatt

```
for(std::vector<int>::iterator it = v.begin(); it != v.end(); ++it){
    // ...
}
```

benutzt man dann entweder

```
for(auto it = v.begin(); it != v.end(); ++it)
```

oder sogar eine bereichsbasierte-`for`-Schleife

```
for(const auto x : v){
    // do stuff but keep entries const
}
for(auto& y : v){
```

```
    // do stuff and maybe change entries
}
```

je nach dem ob man die Elemente nur lesen will oder auch ändern möchte.

Tokens `template` und `typename`

Templates sind ein wichtiges Element der generischen Programmierung. Funktionen die mit Templates implementiert wurden, haben die Eigenschaft, dass der Compiler selbstständig die Varianten für verschiedene Typen erstellen kann. Dies passiert noch bevor die Funktion richtig aufgerufen wird. So kann man es sich beispielsweise sparen, je eine Summierfunktion `sum()` für verschiedene Typen (`int`, `unsigned int`, `float`, `double`) zu definieren und stattdessen einfach eine Funktion mit Templates zu implementieren. Aus den Funktionen

```
int sum(int a, int b){
    return a + b;
}

unsigned int sum(unsigned int a, unsigned int b){
    return a + b;
}

float sum(float a, float b){
    return a + b;
}

double sum(double a, double b){
    return a + b;
}
```

wird also

```
template<typename T>
T sum(T a, T b){
    return a + b;
}
```

und wird folgendermassen aufgerufen

```
int alpha = 13;
int beta  = 12;

int result = sum<int>(alpha, beta);
```

Ein Beispiel für eine Funktion die mit Templates implementiert wurde und zwei verschiedene Typen benutzt, wobei `R` für den Rückgabetypen der Funktion gebraucht wird und `T` für den Eingabetypen. Hierbei muss man teils noch auf implizite Konvertierungen achten[1]

[1] https://en.cppreference.com/w/cpp/language/implicit_conversion

```cpp
template<typename T, typename R>
R square(T number){
    return number * number;
}

// ...

auto w = square<double, double>(1.5);      // w = 2.25 of type double
auto x = square<double, int>(1.5);         // x = 2    of type int
auto y = square<int, int>(1.5);            // y = 1    of type int
auto z = square<int, double>(1.5);         // z = 1.   of type double
```

Funktionen, Funktoren und Lambdas

Funktionen, Funktoren und Lambda-Funktionen können alle in sehr ähnlichen Fällen verwendet werden. Hier schauen wir uns alle drei anhand eines Beispiels an, um die Unterschiede hervorzuheben. Möchte man beispielsweise jede Zahl, die kleiner als 5 ist, aus einem Vektor aus Zahlen herausfiltern, muss man etwas haben, dass einen Input annimmt und sagen kann, ob der Input kleiner als 5 ist oder nicht. Dazu hat man drei Optionen:

Funktionen

Eine Funktion wie `bool less5(T x)` schreiben, die `true` zurückgibt, wenn $x < 5$. Will man dann aber auch noch alle Werte unter 3 filtern, muss man eine weitere Funktion `bool less3(T x)` implementieren.

```cpp
template<typename T>
bool lessthan5(T x){
    return x < 5;
}
```

Funktoren

Die nächste Option ist ein Funktor, der in etwa so aussehen könnte:

```cpp
template<typename T>
class Lessthan_n{
    T value;
public:
    // Constructor
    Lessthan_n(T x): value(x) {}

    // the useful part
    bool operator() (T par) const{
        return par < value;
    }
};
```

Funktoren sind Klassen[2] mit überladenem `operator()`, von denen man mehrere Instanzen erstellen kann, die jeweils andere Werte beinhalten können. Beispielsweise folgendermassen:

```
Lessthan_n<double> lessthan5(5);    // instantiate with 5
Lessthan_n<double> lessthan3(3);    // instantiate with 3
// ...
if(lessthan5(x)){...}               // use like a function
```

Konzeptionell ist ein Funktor also eine Funktion mit Gedächtnis. Diesen Funktor kann man dann einfach mit weiteren Vergleichswerten instanzieren (so wie im Code-Snippet mit 3), um den Vektor mit anderen Werten zu filtern. Das praktische an Funktoren ist, dass sie die gleiche Syntax wie Funktionen erlauben, solange man sie mit einem Wert x aufrufen kann. `Funktion(x)` und `Funktor(x)` sehen gleich aus und funktionieren sehr ähnlich.

Lambda-Funktionen

Am elegantesten und kompaktesten sind für diese Aufgabe allerdings Lambda-Funktionen. Im Code-Snippet weisen wir der Variable `f` eine Lambda-Funktion zu. Hier sieht man auch einen guten Einsatz von `auto`.

```
auto f = [value](int x){return x < value;};
```

(Templates kann man für Lambda Funktionen erst mit dem kommenden C++20 erstellen.)
Lambdas sind wie folgt aufgebaut:

`[value]` (Capture) `(int x)` (Parameters) `->bool` (Return type) `{return x < value;}` (Instruction)

Capture

Variablen, die die Funktion aus dem Kontext des übergeordneten Kontextes entnimmt. Diese können beispielsweise folgendermassen aussehen:

- `[x]` : Zugriff auf kopierten Wert von `x`
- `[&x]` : Zugriff auf Referenz von `x`
- `[&]` : Referenz auf alle Objekte des Kontextes
- `[=]` : Kopie aller Objekte im Kontext
- `[=, &x]` : `x` als Referenz, den Rest als Kopie

Parameters

Parameter, die der Funktion beim Aufruf übergeben werden. Kann weggelassen werden, wenn die Funktion keine annimmt. Die Klammern `()` müssen aber bleiben.

[2] `struct` oder `class` spielt keine Rolle, solang der `operator()` unter `public` ist

Return type

Spezifiziert welcher Rückgabetyp zurückgegeben werden soll. Meistens kann er weggelassen werden, da der Compiler anhand des `return` selbst darauf schliessen kann.

Instruction

Wie ein Funktionsrumpf. Hier stehen die Anweisungen, die die Lambda-Funktion ausführen soll und was sie zurückgibt. Man kann auf alle Werte im Capture und auf alle Parameter zugreifen.

Kapitel 4

Abstrakte Datentypen und Datenstrukturen

Oft werden die Unterschiede zwischen abstrakten Datentypen und Datenstrukturen nicht genügent hervorgehoben. Ein abstrakter Datentyp wird durch eine Datenstruktur repräsentiert. Hier ein kurzer Überblick.

Abstrakter Datentyp (ADT) Abstrakte Datentypen zeichnen sich durch den Fokus auf Interface (API) und Spezifikation (Anforderung an die Operationen), also konkrete Funktionen die einen Effekt auf die Daten haben, aus. Diese beantworten die Frage *welche* Probleme gelöst werden.

Datenstruktur Bei Datenstrukturen liegt der Fokus auf der Implementation und Repräsentation der Daten und die dazugehörigen (auf die Daten ausgelegten) Operationen und Algorithmen. Datenstrukturen sind eine Antwort auf die Frage *wie* Probleme gelöst werden.

Ein abstrakter Datentyp kann in der Regel von mehreren Datenstrukturen implementiert werden, so kann beispielsweise der abstrakte Datentyp *Wörterbuch* durch die Datenstruktur *Hashtable*, die Datenstruktur *Red-Black Tree* oder durch die Datenstruktur *Array* implementiert werden. Siehe Vorlesungsslides[1] für weitere konkrete Beispiele.

Abstrakte Datentypen

Stack

Beim Stack gilt *last in, first out* (LIFO). Ein Stack unterstützt die Operationen:

- `push(x,S)`: legt Element x auf Stapel S
- `pop(S)`: entfernt oberstes Element von S und gibt es zurück (oder *nullptr*)
- `top(S)`: bibt oberstes Element (oder *nullptr*) zurück, ohne es zu entfernen
- `isEmpty(S)`: gibt `true` zurück wenn der Stack leer ist, und sonst `false`
- `emptyStack()`: gibt leeren Stack zurück

[1] Slide 303 ff. https://lec.inf.ethz.ch/DA/2023/slides/daLecture7.handout.pdf

Queue

Bei einer Queue gilt *first in, first out* (FIFO). Eine Queue unterstützt die Operationen:

- `enqueue(x,Q)`: fügt x am Ende der Schlange an
- `dequeue(Q)`: entfernt das vorderste Element der Schlange und gibt es zurück
- `head(Q)`: gibt das Element am vorderen Ende der Schlange zurück, ohne es zu entfernen
- `isEmpty(Q)`: `true` wenn Queue leer ist, sonst `false`
- `emptyQueue()`: gibt leere Queue zurück

Wörterbücher

Ein Wörterbuch ist eine abstrakter Datentyp zur Verwaltung von Schlüsseln k aus $\mathcal{K}$ mit Operationen

- `insert(k,D)`: fügt $k \in \mathcal{K}$ ins Wörterbuch D ein. Elemente können nicht doppelt vorhanden sein
- `delete(k,D)`: entfernt k aus D
- `search(k,D)`: gibt `true` zurück wenn k im Wörterbuch aufgenommen wurde, und sonst `false`

Mengen

Viele komplexe Algorithmen brauchen Menge wie man sie aus der Mathematik kennt. Mengen enthalten keine doppelten Werte und kennen keine Reihenfolge, aber kennen unter anderem folgende Operationen:

- `add(v,S)`: fügt v in die Menge S ein. Elemente können nicht doppelt vorhanden sein
- `remove(v,S)`: entfernt v aus S, falls $v \in S$
- `union(Z,S)`: vereint die Mengen S und Z
- `find(v)`: gibt an, in welcher Menge der Wert v liegt

Datenstrukturen

Bäume

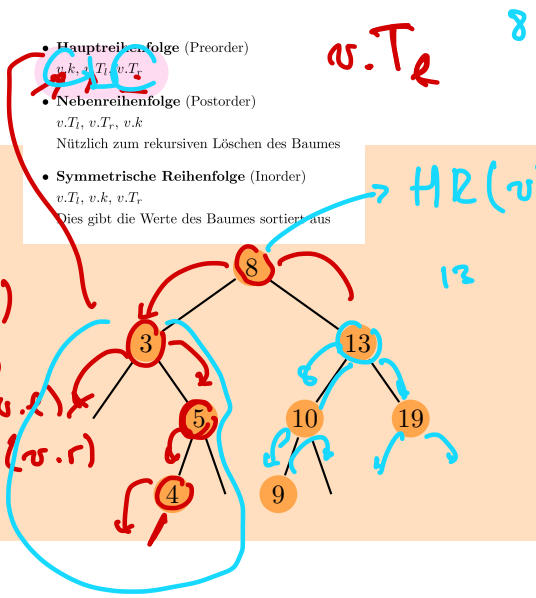
Traversierungen

Bei den Traversierungsarten in Bäumen unterscheidet man zwischen

- **Hauptreihenfolge** (Preorder)
  $v.k$, $v.T_l$, $v.T_r$
- **Nebenreihenfolge** (Postorder)
  $v.T_l$, $v.T_r$, $v.k$
  Nützlich zum rekursiven Löschen des Baumes
- **Symmetrische Reihenfolge** (Inorder)
  $v.T_l$, $v.k$, $v.T_r$
  Dies gibt die Werte des Baumes sortiert aus

Beispiel "Traversierung eines Suchbaums"

- **Hauptreihenfolge** (Preorder)
  $8, 3, 5, 4, 13, 10, 9, 19$
- **Nebenreihenfolge** (Postorder)
  $4, 5, 3, 9, 10, 19, 13, 8$
- **Symmetrische Reihenfolge** (Inorder)
  $3, 4, 5, 8, 9, 10, 13, 19$



Heaps

Ein häufiger Fehler bei Aufgaben mit Bäumen ist die falsche Annahme, dass alle Bäume Suchbäume sind. Insbesondere Heaps sind keine Suchbäume, auch wenn sie auf den ersten Blick so aussehen. Man unterscheidet Min- und Max-Heaps. Diese Datenstruktur ist optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. Die folgenden Eigenschaften beschreiben einen Heap:

- Vollständig bis auf die letzte Ebene
- Lücken in der letzten Ebene des Baumes höchstens rechts.
- Min-Heap: Schlüssel der Kinder sind grösser als der des Elternknotens.

Im Array (beginnend bei 1) sind die Kinder des Elternknotes i an den Stellen $\{2i, 2i+1\}$. Der Elternknoten eines Kindes i ist bei $\lfloor i/2 \rfloor$.

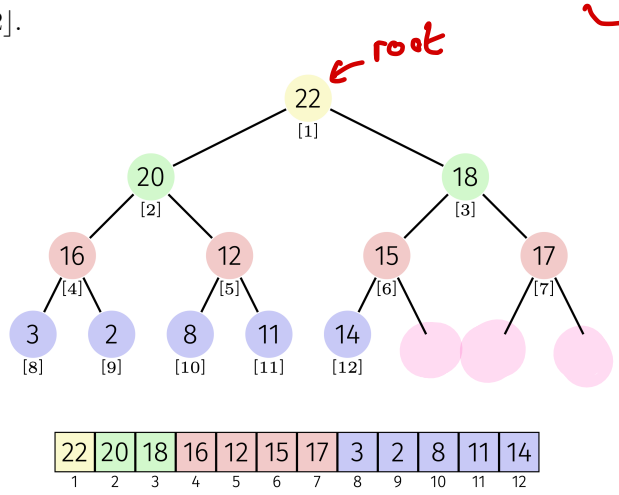


Abbildung 4.1: Algorithmus Aufsteigen

Indexiert man den ersten Eintrag des Arrays mit 0, muss man für Kinder und Eltern $i+1$ bzw. $i-1$ nehmen. Die Höhe des Heaps beträgt $\mathcal{O}(\log n)$. Nach dem Einfügen oder Löschen von Elementen muss man oft die Heapeigenschaften wiederherstellen. Beim Einfügen setzt man das Element an die erste freie Stelle und lässt es dann aufsteigen (4.2).

Algorithmus Aufsteigen(A, m)

Input: Array A mit mindestens m Elementen und Max-Heap-Struktur auf $A[1, \ldots, m-1]$
Output: Array A mit Max-Heap-Struktur auf $A[1, \ldots, m]$.
$v \leftarrow A[m]$ // Wert
$c \leftarrow m$ // derzeitiger Knoten (child)
$p \leftarrow \lfloor c/2 \rfloor$ // Elternknoten (parent)
while $c > 1$ and $v > A[p]$ **do**
  $A[c] \leftarrow A[p]$ // Wert Elternknoten → derzeitiger Knoten
  $c \leftarrow p$ // Elternknoten → derzeitiger Knoten
  $p \leftarrow \lfloor c/2 \rfloor$
$A[c] \leftarrow v$ // Wert → Wurzel des (Teil-)Baumes

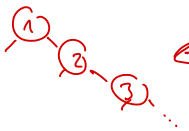
Abbildung 4.2: Algorithmus Aufsteigen(A,m)

Algorithmus Versickern(A, i, m)

Input: Array A mit Heapstruktur für die Kinder von i. Letztes Element m.
Output: Array A mit Heapstruktur für i mit letztem Element m.
while $2i \leq m$ **do**
  $j \leftarrow 2i$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j+1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    swap($A[i], A[j]$)
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

Abbildung 4.3: Algorithmus Versickern(A,i,m)

Beim Entfernen des Minimums bei einem Min-Heap ersetzt man das Minimum mit dem Wert ganz unten rechts und lässt es schliesslich versickern (4.3).



Binärer Suchbaum

Ein binärer Baum T (BST) besteht aus Knoten v die jeweils einen Wert/Schlüssel k abspeichern und zwei Zeiger enthalten: je einer zum linken T_l und zum rechten Teilbaum T_r. Die meisten Operationen verlaufen in $\mathcal{O}(\log n)$.

Die Suchbaumeigenschaften sind:

- Jeder Knoten v speichert einen Schlüssel k
- Schlüssel im linken Teilbaum T_l von v sind kleiner als der Schlüssel k im Knoten v
- Schlüssel im rechten Teilbaum T_r von v sind grösser als der Schlüssel k im Knoten v

Rot-Schwarz-Bäume

Jeder Rot-Schwarz-Baum ist auch ein Suchbaum. Rot-Schwarz-Bäume haben zusätlich noch folgende Eigenschaften, die Ihnen eine bessere Laufzeit bei einigen Operationen ermöglichen:

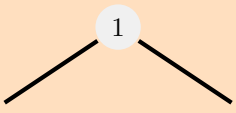
- rote Kanten gehen von Knoten zu seinem linken Kind (left-leaning)
- kein Knoten mit zwei roten Kanten
- jeder Pfad von Wurzel zu Blatt hat die gleiche Anzahl schwarze Kanten (*perfekt schwarz balanciert*)

Die Grundidee bei Operationen auf Rot-Schwarz-Bäumen ist, nach der Operation die oben genannten Eigenschaften mittels Rotatinen, Farbwechseln und Push-downs wiederherzustellen. Diese Grundidee wird

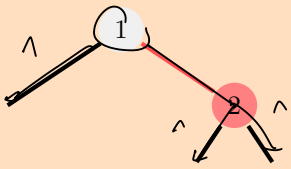
auch noch von diversen anderne baumartigen Datenstrukturen verwendet (AVL-Trees[2]). Diese Operationen und wann sie angewendet werden müssen ist am besten in den Vorlesungsslides 500 ff. nachzulesen[3]

Beispiel "Einfügen in Rot-Schwarz-Baum"

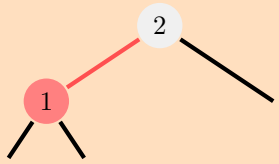
Fügen Sie die Zahlen $1, \ldots, 7$ der Reihe nach in einen (initial leeren) Rot-Schwarz-Baum ein und zeichnen Sie den Baum nach jedem Schritt.



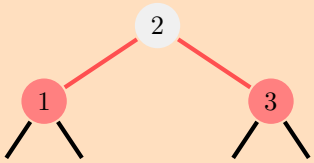
`insert(1)`



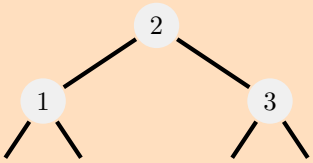
`insert(2): add`



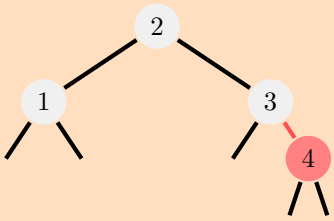
`insert(2): rotate_left`
(da rechtes Kind rot)



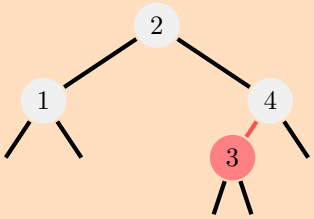
`insert(3): add`



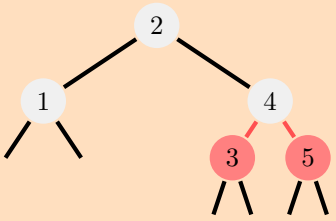
`insert(3): push_up`
(da zwei rote Kinder)



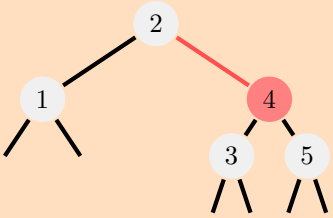
`insert(4): add`



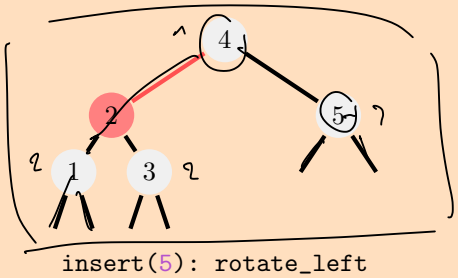
`insert(4): rotate_left`
(da rechtes Kind rot)



`insert(5): add`



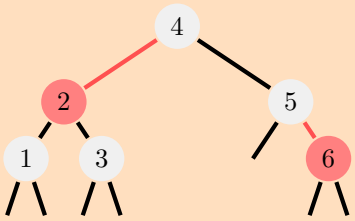
`insert(5): push_up`
(da zwei Kinder rot)



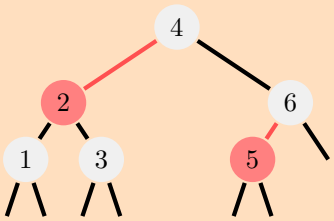
`insert(5): rotate_left`
(da rechtes Kind rot)

[2] nicht im Frühlingssemester 23 vorgekommen
[3] https://lec.inf.ethz.ch/DA/2023/slides/daLecture11.handout.pdf

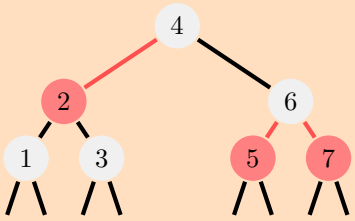
Beispiel “Einfügen in Rot-Schwarz-Baum (Weiterführung)”



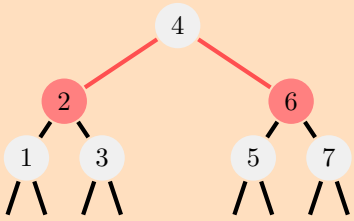
`insert(6): add`



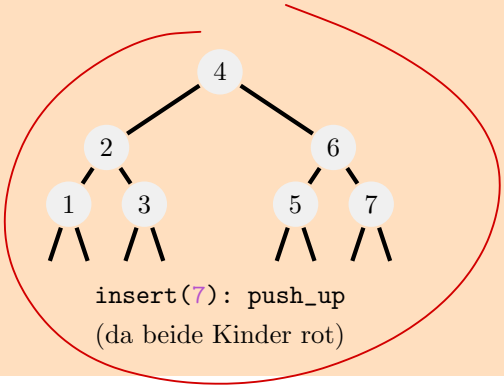
`insert(6): rotate_left`
(da rechtes Kind rot)



`insert(7): add`



`insert(7): push_up` (da zwei Kinder rot)



`insert(7): push_up`
(da beide Kinder rot)

Union-Find

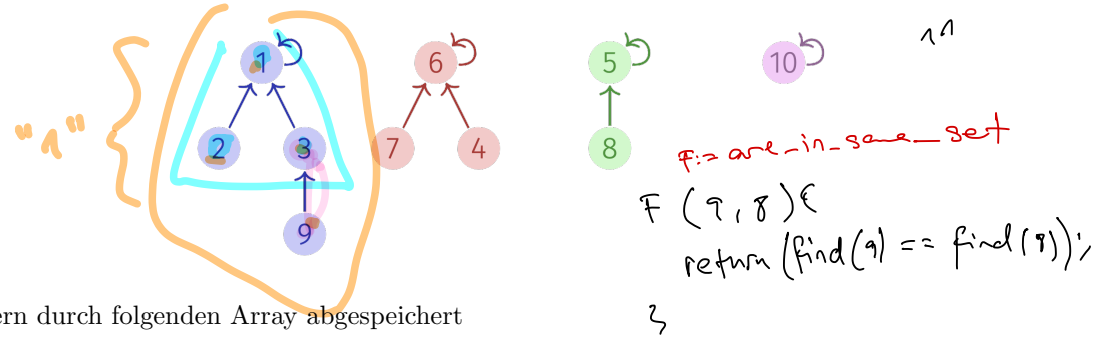
Die Union-Find-Datenstruktur wurde im Zusammenhand mit Kruskals Algorithmus zur berechnung von Minimalen Spannbäumen eingeführt. Man kann sie als eine Art implementation des abstrakten Datentyps der Menge anschauen. Das grundlegende Anliegen ist, dass man eine Menge von Teilmengen hat, und diese schrittweise zu einer Menge zusammenführen möchte. Die dafür benötigten Funktionen sind

- `find(v)` findet den Namen der Menge S, welche das Element v enthält. Im Kontext von Bäumen sind die Wurzeln der Name der Menge
- `union(Z,S)` vereinigt die beiden Mengen mit den Namen Z und S

Die Menge der Teilmengen

$$\{1,2,3,9\},\{7,6,4\},\{5,8\},\{10\}$$

wird duch die folgenden Bäume dargestellt



und intern durch folgenden Array abgespeichert

Index	**1**	2	**3**	4	**5**	**6**	7	8	9	**10**
Parent	1	1	1	6	5	6	6	5	**3**	10

die sich beim Vereinigen der Mengen so ängern, dass jeweils das Vorzeigelement (**im Array fettgedruckt**) neu nicht auf sich selbst, sindern ein element der anderen Menge zeigt. So würde in der obigen Darstellung nach einem `union(Blau, Grün)` der Knoten 5 auf (beispielsweise) die 2 oder zeigen.

Hashtabelle

Vokabular

Prehashing: Abbildung der Schlüssel zu den Ganzzahlen.

$$ph(k) : \mathcal{K} \mapsto \mathbb{N}$$

Oft ist dieser Schritt lediglich impliziert und man geht direkt von Schlüsseln aus, die sich hashen lassen.

Kollision:

$$h(k_i) = h(k_j), \quad i \neq j$$

Hashfunktion bildet zwei verschiedene Schlüssel auf den gleichen Index ab.

Chaining: Alle Werte bei denen Kollisionen vorkamen in diesselbe Linked-List speichern, die über den Index erreichbar ist. Vorteil: man bekommt mehr Schlüssel rein als die Indexmenge eigentlich zulassen würde und das Entfernen von Schlüsseln ist relativ einfach. Nachteil: Speicherverbrauch der Linked-List

Hashfunktion

Eine Hashfunktion h ist eine Abbildung aus der Schlüsselmenge $\mathcal{K}$ auf die Indexmenge eines Arrays (Hashtabelle).

$$h : \mathcal{K} \mapsto \{0, 1, \ldots, m-1\}$$

Hashtabelle

Der Belegungsfaktor α der Tabelle ist definiert als

$$\alpha := \frac{\#\text{Schlüssel}}{\#\text{Index}} = \frac{n}{m}$$

Da meistens $|\mathcal{K}| \gg m$ ist, kommt es zu sogenannten Kollisionen. Hier wird zwei Schlüsseln der gleiche Wert/Platz in der Hashtabelle zugewiesen. Um die Überläufer trotzdem zu speichern, wird eine Sondierungsfunktion benutzt.

Sondieren

Hier sei m die Grösse der Hashtabelle (die Anzahl möglicher Indizes), k der hashbare Schlüssel, j der j-te Schritt in einer Sondierungssequenz (wobei bei $j = 0$ angefangen wird), und $s(k, j)$ der Indexwert für den Schlüssel k im j-ten Sondierungsschritt. Wichtig hierbei ist, dass alle resultierenden gehashten Werte noch $\mod m$ gerechnet werden müssen, bevor man den Schlüssel an diesem Index in der Hashtabelle einfügen kann.

Sondierungssequenz Als Sondierungssequenz $S(j)$ bezeichnet man die Sequenz von Indizes, die man beim Versuch den Schlüssel k in der Hashtablle unterzubringen erhält.

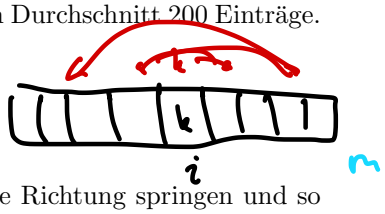
Lineares Sondieren

$$s(k, j) = h(k) \pm j$$

Bei Kollisionen in die vorgegebene Richtung wandern (rechts mit $+$ und nach links mit $-$) bis man auf einen freien Index kommt. Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 200 Einträge.

Quadratisches Sondieren

$$s(k, j) = h(k) + \left\lceil \frac{j}{2} \right\rceil^2 \cdot (-1)^{j+1}$$



Bei Kollisionen jeweils j^2 für jeden Wert j^2 erst in positive dann in negative Richtung springen und so weiter bis ein freier Index rauskommt. Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 22 Einträge.

Doppeltes Hashing

$$s(k, j) = h(k) + j \cdot h'(k)$$

Lineare Sondieren, nur halt nicht in 1-Schritten sondern in $h'(k)$-Schritten. Bei $\alpha = 0.95$ betrachtet die erfolglose Suche hier im Durchschnitt 20 Einträge. Tipp: Ganz am Anfang $s(k, j)$ explizit ausschreiben.

Kapitel 5

Grundlegende Algorithmen

Relevante Prüfungsaufgaben

- ☐ Prüfung 13.02.2023, Aufgabe 1
- ☐ Prüfung 25.08.2022, Aufgabe 1
- ☐ Prüfung 02.02.2022, Aufgabe 1
- ☐ Prüfung 27.08.2021, Aufgabe 1
- ☐ Prüfung 03.02.2021, Aufgabe 1
- ☐ Prüfung 21.08.2020, Aufgabe 1
- ☐ Prüfung 29.01.2020, Aufgabe 1

Suchalgorithmen

Das Suchproblem nennen wir die Aufgabe, einen Wert/Schlüssel (*search key*) in einer Menge von vergleichbaren Werten/Daten zu finden oder sagen zu können, dass dieser nicht in dieser Menge ist. Suchalgorithmen lösen dieses Problem. Vorab ein wichtiges Theoreme, das an den Prüfungen von Nutzen sein können.

- Jeder *vergleichsbasierte* Algorithmus zur Suche eines Werts in *unsortierten* Daten der Grösse n benötigt im schlechtesten Fall $\Omega(n)$ Vergleichsschritte.
  - jeder *vergleichsbasierte* Suchalgorithmus hat daher eine Laufzeit von $\Omega(n)$
  - Argument: jedes Element muss in mindestens einem Vergleich vorkommen

Lineare Suche

Die lineare Suche durchläuft das Array vom ersten bis zum letzten Element und vergleicht dies mit dem gesuchten Schlüssel.

- Bestenfalls: 1 Vergleich (das erste Element ist das gesuchte)
- Schlimmstenfalls: n Vergleiche (das letzte Element ist das gesuchte)
- Erwartungswert der Laufzeit bei Gleichverteilung: $\frac{n+1}{2}$
- Laufzeit: $\mathcal{O}(n)$

Binäre Suche

Voraussetzung für die Binäre Suche ist ein sortiertes Array. Der Algorithmus beginnt in der Mitte, vergleicht ob das Element grösser oder kleiner als der gesuchte Schlüssel ist und wiederholt dieses Vorgehen in der linken bzw. rechten Hälfte (Divide & Conquer). Besser als ein Vektor eignet sich hierfür ein Suchbaum (AVL-Bäume[1], Rot-Schwarz-Bäume). Der Algorithmus zur binären, sortierten Suche benötigt im schlechtesten Fall $\Omega(logn)$ Elementarschritte.

- Laufzeit: $\Theta(\log n)$

Auswahlalgorithmen

Das Auswahlproblem nennen wir die Aufgabe, den k-kleinsten Wert aus einer unsortierten Menge von n Werten wiederzugeben. Wichtige Spezialfälle sind:

- $k = 1$: das Minimum einer Menge
  - ein möglicher Algorithmus ist, durch die Menge zu iterieren und den bisher kleinsten Wert abzuspeichern und dann am Ende wiederzugeben.
- $k = n$: das Maximum einer Menge
  - ein möglicher Algorithmus ist, durch die Menge zu iterieren und den bisher grössten Wert abzuspeichern und dann am Ende wiederzugeben.
- $k = \frac{n}{2}$: der Median einer Menge
  - Hierzu ein Vergleich der möglichen Algorithmen:

$\mathcal{O}(n^2)$	k-fach Minimum finden
$\mathcal{O}(n \log n)$	Sortieren, dann auswählen
$\mathcal{O}(n)$ im Mittel	Quickselect mit zufälligem Pivot
$\mathcal{O}(n)$ im schl. Fall	Median-of-Medians (Blum)

Sortieralgorithmen

Das Sortierproblem nennen wir die Aufgabe, eine Menge von unsortierten, vergleichbaren Werten in sortierter Reihenfolge wiederzugeben. Ein wichtiges Theoreme und Information vorab, die an den Prüfungen von Nutzen sein können.

- *Vergleichsbasierte* Sortieralgorithmen benötigen im schlechtesten Fall und im Mittel mindestens $\Omega(n \log n)$ Schlüsselvergleiche und damit eine Laufzeit von $\Omega(n \log n)$
- Radix- und Bucketsort sind *nicht* vergleichsbasiert!

Bubblesort

Bei Bubblesort geht man den Array (schlimmstenfalls bis zu) n Mal von links nach rechts durch und betrachtet immer zwei benachbarte Elemente. Sind diese nicht sortiert, werden sie vertauscht. Wird in einem Durchlauf nichts vertauscht, ist das Array sortiert. Das schlimmstmögliche Szenario ist ein absteigend sortiertes Ausgangsarray. Dieser Algorithmus resultiert (schlimmstenfalls) in:

- $\Theta(n^2)$ Vergleiche

[1] nicht im Frühlingssemester 23 behandelt

- $\Theta(n^2)$ Vertauschungen

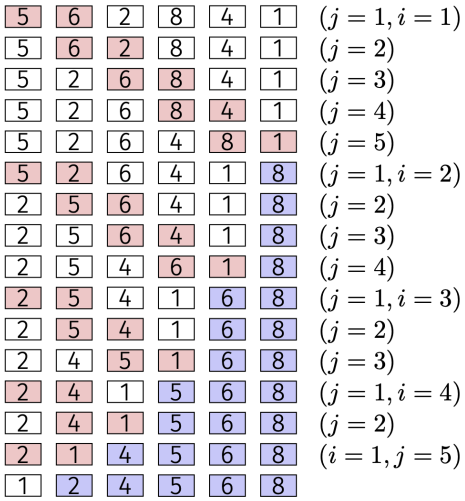


Abbildung 5.1: Bubblesort

Selectionsort

Das kleinste Element wird an die erste Stelle des Arrays getauscht. Das zweit-kleinste kommt an die zweite Stelle und so weiter. Bei der Implementation wird der Pointer für den linken Rand des Arrays in jedem Schritt um ein Element nach rechts verschoben.

- $\Theta(n^2)$ Vergleiche
- $\Theta(n)$ Vertauschungen

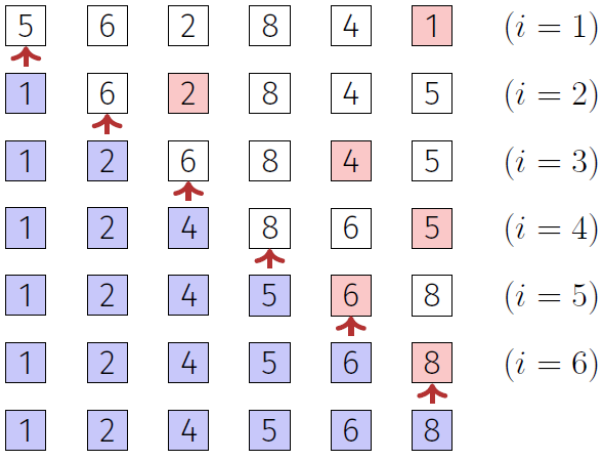


Abbildung 5.2: Selectionsort

Insertionsort

Insertionsort funktioniert so, wie man Spielkarten in der Hand sortiert. Die Schleife geht von $i = 1$ bis n. Man wählt das i-te Element des Arrays `arr` und fügt es mittels binärer Suche in $\texttt{arr}[0 \ldots i-1]$ ein. Wie bei Selectionsort hat man während des Algorithmus einen sortierten und einen unsortierten Teil innerhalb des Arrays. Auch Insertionssorts schlimmstmögliches Ausgangsarray ist absteigend sortiert.

- $\Theta(n \log n)$ Vergleiche
- $\Theta(n^2)$ Vertauschungen

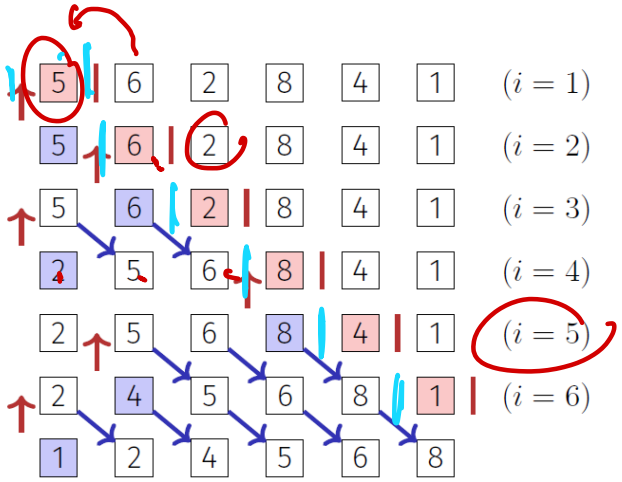


Abbildung 5.3: Insertionsort

Mergesort

Dieser Algorithmus arbeitet rekursiv. Der Input wird in zwei Teile aufgeteilt, mit welchen Mergesort erneut aufgerufen wird. Nachdem zwei Teile sortiert zurückkommen, werden sie (linear in Zeit) aufsteigend zusammengefügt. Der Algorithmus benötigt schlimmstenfalls $\Theta(n \log n)$ Vergleiche und immer $\Theta(n \log n)$ Vertauschungen.

Abbildung 5.4: Mergesort

Quicksort

Hier wird erneut Rekursion angewandt. Es wird ein Pivotelement gewählt (unten rot[2]) und anhand dessen zwei Teilarrays erstellt, in denen alle kleineren bzw. grösseren Elemente sind. Die Funktion ruft sich dann erneut auf diesen Arrays auf (Rekursion) und fügt sie schliesslich mit dem Pivot in der Mitte zusammen. Die Laufzeit beträgt im (seltenen) schlechtesten Fall $\Theta(n^2)$ und sie tritt dann ein, wenn die Pivotelemente jeweils Extrema sind. Im Mittel werden $\mathcal{O}(n \log n)$ Vergleiche benötigt.

[2]Die Pivotelemente 1, 4, 6 und 9 wurden hier nicht rot gefärbt, da sie keine weiteren Wechsel verursachen, also bereits richtig sortiert sind

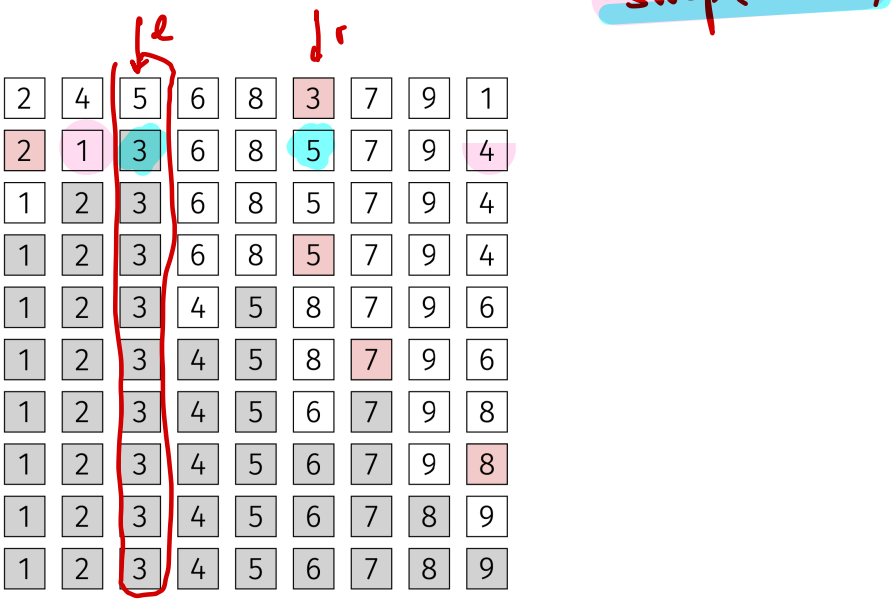


Abbildung 5.5: Quicksort

Heapsort

Alle Elemente werden in einen Max-Heap eingefügt. Nun vertauscht man die Wurzel mit dem letzten Knoten (konkret vertauscht man also das erste und letztes Element im Array), löscht die ehemalige Wurzel und stellt erneut die Max-Heap Eigenschaften her. Dieser Vorgang wird wiederholt, bis der ganze Heap aufgelöst wurde. Bewegt man dabei immer die Elemente im Array, wird dieses sortiert.

- Einfügen: $\mathcal{O}(\log n)$
- Löschen: $\mathcal{O}(\log n)$
- Sortieren: $\mathcal{O}(n \log n)$

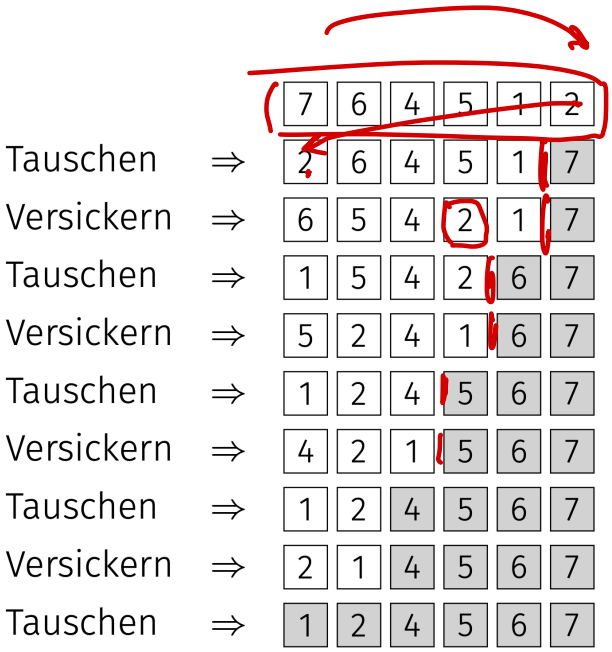


Abbildung 5.6: Heapsort

Radixsort

Dieser Algorithmus sortiert die Eingabe Ziffer für Ziffer, beginnend bei der signifikantesten. Im schlechtesten Fall erreicht Radixsort $\mathcal{O}(p \cdot n)$, wobei p die Anzahl der Ziffern ist.

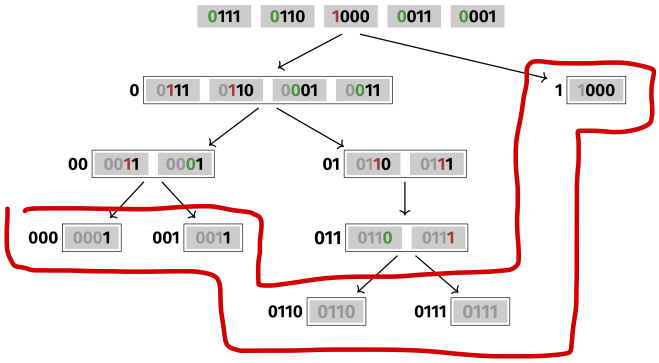


Abbildung 5.7: Radixsort

Bucketsort

Hier werden wiederholt Buckets erstellt und deren Inhalte in der richtigen Reihenfolge zusammengefügt. Der erste Bucket sortiert die Eingabe anhand der letzten Ziffer. Nach dem Zusammenfügen werden die Zahlen anhand der zweitletzten Ziffer sortiert und so weiter. Die Laufzeit beträgt $\mathcal{O}(n)$.

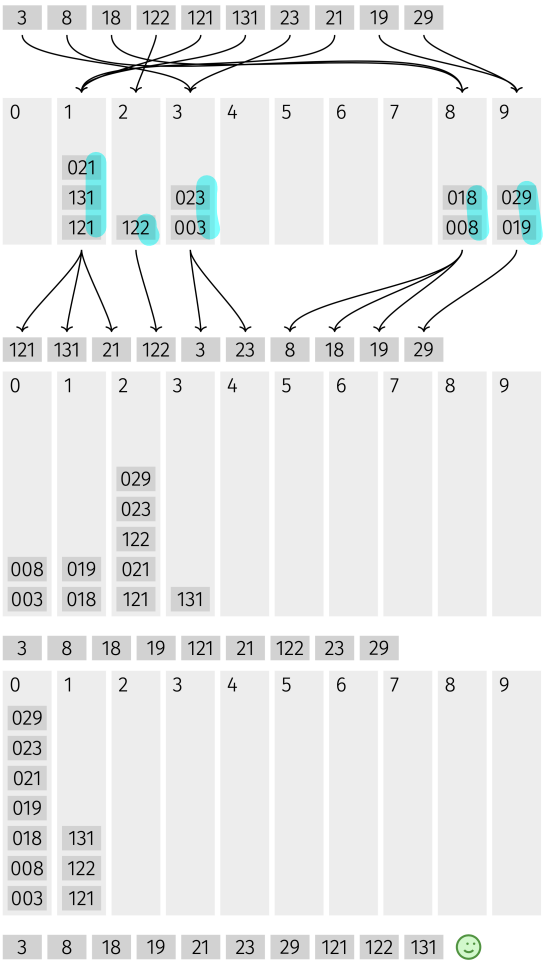


Abbildung 5.8: Bucketsort

Kapitel 6

Dynamische Programmierung vs. Gierige Algorithmen

Relevante Prüfungsaufgaben

- ☐ Prüfung 13.02.2023, Aufgabe 6
- ☐ Prüfung 25.08.2022, Aufgabe 4
- ☐ Prüfung 02.02.2022, Aufgabe 6
- ☐ Prüfung 27.08.2021, Aufgabe 5
- ☐ Prüfung 03.02.2021, Aufgabe 3
- ☐ Prüfung 21.08.2020, Aufgabe 3
- ☐ Prüfung 29.01.2020, Aufgaben 3, 5

Memoisation

Bevor ein Teilproblem gelöst/berechnet wird, wird die Existenz eines Zwischenresultats geprüft. Ist dieses vorhanden, wird es geladen und für weiterführende Berechnungen verwendet, falls nicht, wird gerechnet und das Resultat gespeichert. Dieser Vorgang nennt sich *Memoisation* und ist ein wichtiger Bestandteil vieler DP-Algotithmen.

Top-Down vs. Bottom-Up

Man unterscheidet bei den Ansätzen zwischen *Top-Down* und *Bottom-Up*.

Top-Down

Der Top-Down-Ansatz erinnert stark an Rekursion, unterscheidet sich aber darin, dass man nicht zwingend alle Berechnungen bei jedem Funktionsaufruf macht. Am Beispiel von Fibonacci-Zahlen würde man also die Exsitenz der n-ten Zahl prüfen (Memoisation), dann der $(n-1)$-ten usw., bis man die 2. und 1. aufruft, welche dann gespeichert werden.

Bottom-Up

Bei einem Bottom-Up-Ansatz ist der Algorithmus Iterativ anstatt Rekursiv. Dieser Ansatz ist für Dynamische Programmierung üblich. Es werden sofort die ersten zwei Fibonacci Zahlen gespeichert und dann in einer Schleife n Zahlen aus ihren jeweiligen Vorgängern berechnet.

Dynamische Programmierung

Die Grundidee der Dynamischen Programmierung (DP) ist das Aufteilen eines Problems in einfachere Teilprobleme. Die Lösungen derer werden zwischengespeichert und schliesslich zur Lösung des ursprünglichen Problems benutzt.

Lösungsschema

Ein DP-Problem kann (so gut wie) immer nach dem folgenden Schema gelöst werden.

1. Definieren und Erstellen der **DP-Tabelle**, in der die Resultate gespeichert werden.
   - Welche Dimension hat die Tabelle?
   - Was bedeutet jeder Eintrag?
2. Berechnung der **Randfälle** (Base Cases).
   - Welche Einträge sind unabhängig von anderen?
3. **Berechnungsreihenfolge** bestimmen.
   - Wie kann man Einträge berechnen, so dass bei jedem Schnritt die benötigten Operanden bereits vorhanden sind?
4. Rekonstruktion/Bestimmung der **Lösung**
   - Wie/Wo kann ich die Lösung auslesen?

Beispiel "Fibonacci-Folge"

Hier das Schema am klassischen Beispiel der Fibonacci-Folge:

1. Array mit den Dimensionen $n \times 1$, wobei der n-te Eintrag der n-ten Fibonacci Zahl entspricht
2. Hier sind die ersten beiden Zahlen der Reihenfolge $F_1 = 1$ und $F_2 = 1$ unabhängig von allen anderen, also werden diese zu Beginn eingetragen
3. F_i mit aufsteigendem i berechnen
4. Die gesuchte Fibonacci-Zahl ist an der Stelle n in der Tabelle

Die Laufzeit von DP-Algorithmen beträgt typischerweise

$$(\#\text{Tabelleneinträge}) \cdot (\text{Aufwand pro Eintrag})$$

Gierige Algorithmen (*Greedy Algorithms*)

Bei *Greedy Algorithms* wird in jeder Iteration der Schritt gewählt, der uns dem Ziel/Optimum in diesem Schritt am nächsten bringt. Probleme auf die gierige Weise zu lösen ist meist schneller, jedoch oft nicht die beste Art.

Huffman-Codierung

Huffman-Bäume dienen zum effizienten Speichern einer Folge von Zahlen, in dem man diese in eine binäre Folge codiert. In einem Codebaum gilt: je häufiger der Buchstabe vorkommt, desto näher an der Wurzel ist er. Erstellt wird ein Huffman-Baum auf gierige Weise von unten nach oben:

1. Huffman-Baum erstellen
   (a) Man startet mit der Menge W von Wörtern
   (b) Man erstellt einen Knoten, der ein Wort $\in W$ und dessen Häufigkeit speichert und fügt diese zusammen zu einer Menge Q
   (c) Man ersetzt Iterativ die beiden Knoten mit der geringsten Häufigkeit in Q mit einem neuen Elternknoten, der die kombinierte Häufigkeit seiner beiden Kinder speichert
   (d) Nun ist anstatt der beiden Kinderknoten der Vaterknoten in Q und man geht weiter zur nächsten Iteration
2. Huffman-Codierung rauslesen Zur Codierung geht man den Baum von der Wurzel bis zum gewünschten Wort (Blatt im Baum) hinunter.
   - Bei jedem linken Kind wird dem Code eine 0 angehängt, bei jedem rechten eine 1
   - Der resultierende Code ist das die Codierung des Worts.

Huffman-Codierungen sind (wenn man den gegebenen Algorithmus befolgt) *nicht* eindeutig.

Kruskals Algorithmus

Kruskals Algorithmus zur Bestimmung eines Minimalen Spannbaums (*Minimum Spanning Tree (MST)*) ist ebenfalls ein gieriger Algorithmus und funktioniert wie folgt:

1. die Kanten des Graphen werden erst aufsteigend nach ihrem Gewicht sortiert
2. dann wird über die sortierten Kanten iteriert. Wenn eine Kante zwei Knoten verbindet, die noch nicht durch einen Pfad vorheriger Kanten verbunden sind, wird diese Kante zum MST hinzugenommen.

Minimale Spannbäume sind (wenn man den gegebenen Algorithmus befolgt) *nicht* eindeutig. Die Summe der Kantengewichte aller Minimalen Spannbäume eines Graphen sind gleicht (denn sonst wär ja eine der Summen grösser und damit offensichtlich nicht minimal!).

Aufgabe "Dynamische Programmierung"

1. **Längste aufsteigende Teilfolge (LIS)**
   Gegeben sei ein Array mit Zahlen, beschreibe einen DP-Algorithmus, der die längste aufsteigende Teilfolge dieses Arrays berechnet.

   Für die Zahlen $50, 3, 10, 7, 40, 80$ wäre die LIS $3, 7, 40, 80$ mit der Länge 4.

2. **Rucksackproblem (*Knapsack Problem*)**
   Gegeben sind:

   - Menge von $n \in \mathbb{N}$ Gegenständen $\{1, ..., n\}$
   - jeder Gegenstand i hat einen Wert $v_i \in \mathbb{N}$ und Gewicht $w_i \in \mathbb{N}$
   - Das Maximalgewicht liegt bei $W \in \mathbb{N}$

   Gesucht ist eine Auswahl $I \subseteq \{1, ..., n\}$, die die Summe aller Werte maximiert aber das Maximalgewicht nicht überschreitet.

3. **Dice Throw**
   Gegeben sind n Würfel mit je m Seiten, nummeriert von 1 bis m. Auf wie viele Arten kann man die Augensumme X erhalten?

4. **Mars Mission**[a]
   Ein Marsroboter fährt von einer Startposition $S(1, 1)$ nach $Z(m, n)$. Er kann nur nach rechts und unten fahren und versucht auf seinem weg möglichst viele Gesteinsproben einzusammeln. Entwerfe einen Algorithmus, der die Maximale Anzahl Proben und den Pfad, den der Roboter fährt, berechnet.

S	9	2	5	11	8
17	21	32	5	15	3
2	2	3	8	1	5
8	2	8	11	15	9
0	5	3	10	4	Z

[a] Quelle: `https://lec.inf.ethz.ch/DA/2018/exercises/mars_mission.png`

Kapitel 7

Paralleles Programmieren

Beim parallelen Programmieren geht es darum, Aufgaben, die gleichzeitig ausgeführt werden können, auf verschiedene Prozessoren[1] zu verteilen. Dabei ist jedoch Vorsicht geboten, wenn die verschiedenen Programmzweige auf gleiche Daten zugreifen und sich damit gegenseitig in die Quere kommen könnten.

Für die Prüfungsvorbereitung macht es Sinn, das Thema des parallelen Programmieren in zwei Teile zu teilen, nämlich einen theoretischen Teil (Amdahl und Gustafson) und einen praktischen Teil (Paralleles Programmieren in C++). Deshalb sind hier die relevanten Prüfungsaufgaben auch separat angegeben.

Amdahl vs. Gustafson

Relevante Prüfungsaufgaben

- ☐ Prüfung 13.02.2023, Aufgabe 4
- ☐ Prüfung 21.08.2020, Aufgabe 6
- ☐ Prüfung 03.02.2021, Aufgabe 6
- ☐ Prüfung 29.01.2020, Aufgabe 6

Amdahl und Gustafson sind beides Modelle, mit denen man die Beschleunigung (den Speedup) bei Parallelisierung auf p Prozessoren berechnen kann. Während Amdahl von einem relativen sequenzielle Teil (abhängig von der Problemgrösse) ausgeht, betrachtet Gustafson den sequenzielle Teil als fix (unabhängig der Problemgrösse). Welches Modell besser passt, ist vom Problem abhängig.

Amdahls Gesetz

Amdahl teilt die nötige Rechenarbeit W in einen parallelisierbaren Teil W_p und einen nicht parallelisierbaren, sequenzielle Teil W_s auf. Die sequenzielle Ausführungszeit sei T_1, die parallele T_p mit p Prozessoren. Es gilt

$$T_p \geq T_1/p$$

Bei Gleichheit ist Perfektion erreicht, was eigentlich[2] unmöglich ist. Der Speedup S_p für p Prozessoren berechnet sich dann aus

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}$$

[1] ob diese virtuell(e Threads) sind oder nicht, ist uns egal.
[2] Cache-Effekte können solch Verhalten verursachen. Mehr dazu in späteren Semestern

Benutzt man den prozentualen Anteil λ, gegeben durch $W_s = \lambda W$, wodurch $W_p = (1-\lambda)W$, erhält man die Ungleichungen

$$S_p \leq \frac{1}{\lambda + \frac{1-\lambda}{p}}$$
$$S_\infty \leq \frac{1}{\lambda}$$

Beispiel "Amdahls Gesetz"

Ein Programm ist zu 80% parallelisierbar ($\implies \lambda = 1 - 0.8 = 0.2$). Die Zeit, das Programm auf einem Prozessor laufen zu lassen, beträgt 10 Zeiteinheiten. ($\implies T_1 = 10$). Der Speedup für 8 und ∞ Prozessoren (S_8 und S_∞) ist also gegeben durch

$$S_8 \leq \frac{1}{0.2 + \frac{1-0.2}{8}} \approx 3.33$$
$$S_\infty \leq \frac{1}{0.2 + \frac{1-0.2}{\infty}} = 5$$

Gustafsons Gesetz

Gustafsons Modell hält die Ausführungszeit T fest und variiert die Problemgrösse, wobei man davon ausgeht, dass der sequenzielle Teil (nicht *An*teil!) konstant bleibt und der parallelisierbare Teil mit der Problemgrösse wächst. Die Arbeit, die mit p Prozessoren in der Zeit T erledigt werden kann, ist gegeben als

$$W_s + p \cdot W_p = \lambda \cdot T + p(1-\lambda)T$$

Der Speedup bei p Prozessoren ist dann gegeben durch

$$\begin{aligned} S_p &= \frac{W_s + p \cdot W_p}{W_s + W_p} \\ &= p \cdot (1-\lambda) + \lambda \\ &= p - \lambda(p-1) \end{aligned}$$

Paralleles Programmieren

Relevante Prüfungsaufgaben

- ☐ Prüfung 25.08.2022, Aufgabe 7
- ☐ Prüfung 02.02.2022, Aufgabe 7
- ☐ Prüfung 27.08.2021, Aufgabe 7
- ☐ Prüfung 21.08.2020, Aufgabe 7
- ☐ Prüfung 29.01.2020, Aufgabe 7

Multithreading mit `std::threads`

Threads übergibt man beim Aufruf eine Funktion und die Liste der Parameter, die diese Funktion benötigt. Per Default wird alles per Wert übergeben. Um eine Variable `x` als Referenz zu übergeben,

benutzt man `std::ref(x)` in der Liste der Funktionsparameter. Das Erstellen eines Thread könnte beispielsweise folgendermassen aussehen

```
#include <thread>

void foo(int& x){
    x = 1;
}

int main(){

    int x = 0;

    std::thread t(foo, std::ref(x));        // open thread
    // x is being set to 1
    t.join();                               // close thread
}
```

Hat man es mit mehreren Threads zu tun, lohnt es sich, diese in einen `std::vector` zu packen

```
std::vector<std::thread> tv(4);
for(auto& t : tv){
    t = std::thread(foo, x);
}

// {good use for the threads goes here}

for(auto& t : tv){
    t.join();
}
```

Zugänge verwalten mit `std::mutex`

Soll ein Code Abschnitt ausgeführt werden, ohne dass ein anderer Thread auf die verwendeten Daten zugreifen kann, benutzt man einen `std::mutex`. Konkretes Beispiel:

```
#include <mutex>

std::mutex m;

m.lock();
// {critical section that operates
// on some variables goes here}
m.unlock();
```

Ein Bisschen so wie bei Pointern bei denen man sich mit Smart-Pointern die ganze Mühe mit der manuellen Deallokation (die gerne mal vergessen geht) sparen kann, kann man sich mit `std::lock_guard` die das ständige `.lock()` und `.unlock()` sparen. Hier ein Beispiel:

```
void foo(){
    std::lock_guard<std::mutex> guard(m)    // guard created
    // {critical section that operates      // and mutex locked
    // on some variables goes here}
}
                                            // guard destroyed
                                            // and mutex unlocked
```

Wenn man in einem Datensatz mehrere Elemente einzeln sperren muss, kann die Reihenfolge einen grossen Unterschied machen. Man sollte immer die gleiche Lock-Reihenfogle beibehalten, also beispielsweise mit aufsteigendem Index sperren. Dies kann Deadlocks verhindern.[3]

Race Conditions

Eine Race Condition tritt auf, wenn das Resultat eines Programms vom Scheduling abhängt. Hier wurden die Definitionen aus der 24. Lektion übernommen und adaptiert.[4]

Bad Interleaving

Fehlerhaftes Programmverhalten verursacht durch eine unglückliche Ausführungsreihenfolge eines Algorithmus mit mehreren Threads. Kann selbst dann auftreten, wenn Zugriffe zum Speicher anderweitig gut synchronisiert sind. Die Annahme, die man über die Atomizität von Operationen trifft sind fast immer falsch.

Data Races

Fehlerhaftes Programmverhalten verursacht durch ungenügend synchronisierten Zugriff zum Speicher, beispielsweise gleichzeitiges Lesen und Schreiben oder Schreiben und Schreiben zum gleichen Speicherbereich. Kann selbst dann auftreten, wenn es keine unglückliche Ausführungsreihenfolge der Instruktionen für den Fehler verantwortlich gemacht werden kann.

Locks

Deadlock

Zwei oder mehr Prozesse blockieren sich gegenseitig, weil jeder auf einen anderen wartet. Deadlocks erkennt man an Zyklen im Abhängigkeitsgraphen. Zur Vermeidung kann man den Prozessen Prioritäten zuordnen.

Starvation

Starvation ist der erfolglose Versuch, eine zwischenzeitlich freigegebene Ressource zu erhalten, ohne die der Prozess nicht fortgesetzt werden kann.

Livelock

Konkurrierende Prozesse erkennen einen potentielle Deadlock, machen aber keinen Fortschritt, diesen aufzulösen.

[3] Mehr hierzu findet man online unter dem Begriff "Dining Philosophers"
[4] https://lec.inf.ethz.ch/DA/2023/slides/daLecture24.handout.pdf

Threads wecken mit `std::condition_variable`

Wenn ein Thread auf etwas warten muss, kann man ihn mit `.wait(some_mutex, some_condition)` schlafen lassen. Möchte man den Thread wieder wecken (um ihn nachschauen zu lassen, ob er weiterarbeiten kann), kann man ihm von einem anderen Thread aus ein Signal (in Form eines `.notify_one()` oder `.notify_all()`) schicken, das ihn weckt. Der geweckte Threat schaut sich nun die Bedingung `some_condition` an und macht dort weiter wo er eingeschlafen ist, falls die Bedingung `true` zurückgibt. Um das ganze zu verwenden, muss man davor eine `std::condition_variable` einführen.

```
#include <mutex>
#include <condition_variable>

using guard = std::unique_lock<std::mutex>;

std::mutex m;
std::condition_variable cond;
```

Wichtig hierbei ist, dass man `std::condition_variable` und `some_condition` nicht verwechselt. Bei `std::condition_variable` handelt es sich um eine Condition Variable (also das Objekt, an dem sich alle Threads orientiern) und bei `some_condition` handelt es sich um eine Funktion (oft sogar eine Lambda-Funktion) die bestimmt, wann der Thread das nächste mal probiert, sich den `std::mutex m` zu schnappen (nämlich sobald diese Funktion `true` zurückgibt).

```
// INSIDE THREAD B

guard g(m);                                 // lock mutex
cond.wait(g, [&slice](Bread slice){         // relase mutex,...
                        return (slice.state == "toasted");
                    });                     // ...until slice is toasted
slice.apply_butter();                       // and then proceed
slice.apply_jam();                          // with recipe
```

```
// INSIDE THREAD A

guard g(k);             // lock mutex
Toaster.toast(slice);   // toast the slice of bread
cond.notify_one();      // notify thread B that slice is toasted
```

Falls die Bedingung in `cond.wait()` nicht erfüllt ist (also das Brot noch nicht getoasted), wartet Thread B bis zum nächsten `cond.notify_one()`. Beim Erhalt wird erneut geprüft ob die Bedingung erfüllt ist und falls nicht wird wieder gewartet. Als Alternative zu `notify_one()`, wo nur ein Thread benachrichtigt wird, gibt es auch `notify_all()`, wo alle Threads geweckt werden.