Algorithmen und Programmieren 3

Dirk Braun

1. Februar 2011

Inhaltsverzeichnis

1 Einleitung

Bei diesem Text handelt es sich um eine neuverfasste Version meiner Mitschriften aus der Vorlesung *Algorithmen und Programmieren 3: Datenstrukturen und Datenabstraktion* der Freien Universität Berlin im Wintersemester 2010/2011 durch Prof. Wolfgang Mulzer. Ich übernehme keine Verantwortung für inhaltliche Fehler. Ich würde mich jedoch sehr freuen, bei sämtlichen Problemen (von Rechtschreibfehler über Formulierungen bis hin zu inhaltlichen Fehlern) eine kurze Email zu erhalten. Meine Adresse lautet:

dirk.braun@fu-berlin.de

Sollten, neben der Vorlesung, weitere Quellen verwendet worden sein, so wird auf sie im Verzeichnisregister hinten hingewiesen.

1.1 Eine entscheidene Wahl

Um zu verstehen, wie wichtig es ist, für Daten die richtige Repräsentation zu verwenden, sehen wir uns ein Beispiel an. Wir stellen uns die folgende Situation vor: Für unser Programm werden *Polynome* benötigt. Dazu sollen darauf Operationen ausgeführt werden. Wir müssen also eine Klasse implementieren, in der wir das Polynome abspeichern können und können dann die benötigten Operationen hinzufügen. Wir definieren das Polynom wie folgt:

$$n \in \mathbb{N}: \quad a_0, a_1, a_2, \ldots, a_n \tag{1.1}$$

Wir benötigen hierfür $n-1$ Zahlen, womit für das Polynom vom Grad n der formale Ausdruck wie folgt definiert ist:

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n && (1.2) \\ &= \sum_{i=0}^{n} a_i x^i && (1.3) \end{aligned}$$

Wichtig zu verstehen ist, dass somit ein Polynom vom Grad n oder größer ist. Beispielsweise ist das Polynom $p(x) = 0$ vom Grad 0 aber auch 100! Genauso gilt für das Polynom

$$p(x) = x^5 + 2x^3 - 17 \tag{1.4}$$

dass es vom Grad ≥ 5 ist (Eine Grafische Darstellung des Polynoms ist in der Grafik 1.1 zu sehen). Als nächstes wollen wir uns ansehen welche Operationen benötigt werden. Damit können wir dann beriets entscheiden welche Art der Repräsentation benötigt wird.

Auswertung an der Stelle x_0 Wir wollen bestimmen, welchen Wert das Polynom $p(x)$ annimmt, wenn wir einen Wert für x einsetzen.

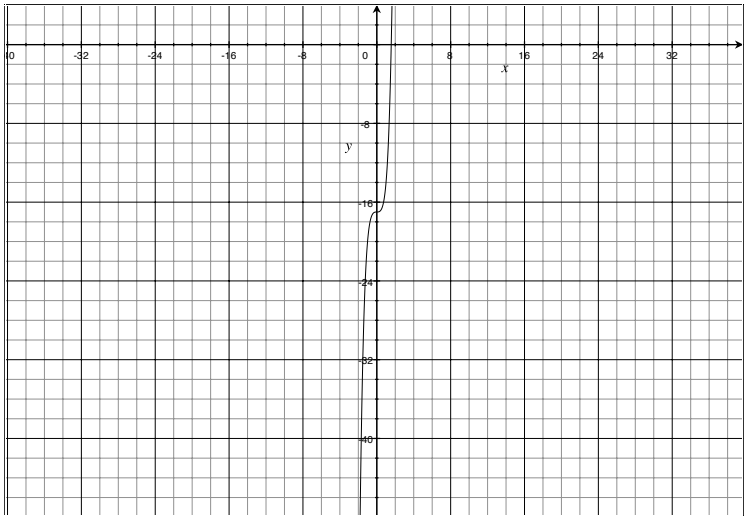

Abbildung 1.1: Das Polynom $x^5 + 2x^3 - 17$

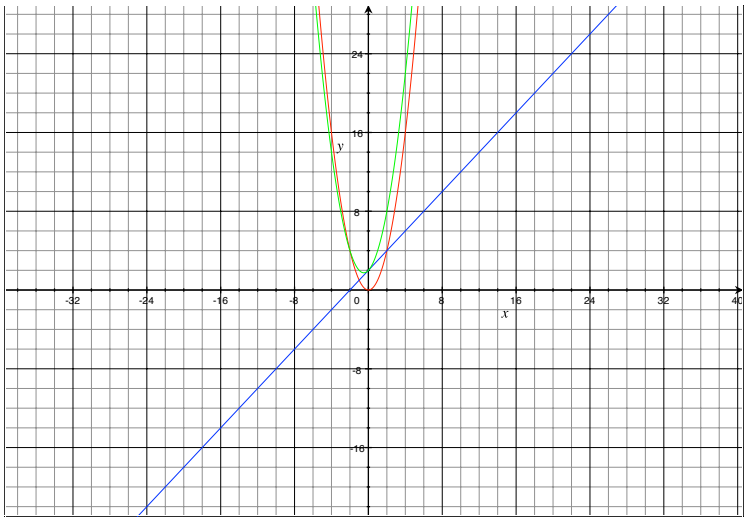

Abbildung 1.2: Drei Polynome: $f(x) = x^2$, $g(x) = x + 2$, sowie deren Summe $h(x) = (x^2) + (x + 2)$

Addition Zwei Polynome werden addiert, wie in Abbildung 1.2 zu sehen ist

Multiplikation Zwei Polynome werden ähnlich multipliziert wie sie addiert werden, zu erkennen in der Abbildung 1.3.

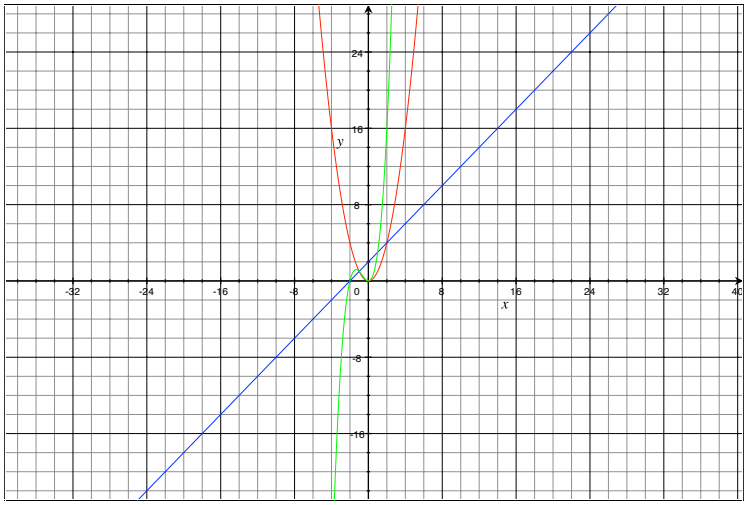

Abbildung 1.3: Drei Polynome: $f(x) = x^2$, $g(x) = x + 2$, sowie deren Summe $h(x) = (x^2) \cdot (x + 2)$

Die Frage die uns nun interessiert, lautet:

„*Mit welcher Darstellung lassen sich die Operationen besonders leicht implementieren und haben dabei eine möglichst gute, erwartete Laufzeit?*“

Ein erster Ansatz, der wahrscheinlich am schnellsten in den Sinn kommt, wäre die Koeffizienten eines Polynoms in einem Array zu speichern. Somit würde unsere Polynom-Klasse wie folgt aussehen:

Listing 1.1: KoeffizientenPolynom.java

```
1 class KoeffizientenPolynom {
2         Array koeffizienten;
3         // Operationen folgen
4 }
```

Wir können nun Polynome addieren, indem wir die einzelnen Koeffizienten addieren. Wir nehmen daher an, das wir die folgenden Koeffizienten haben:

$$a_0, a_1, \ldots, a_n \tag{1.5}$$

$$b_0, b_1, \ldots, b_n \tag{1.6}$$

$$c_0, c_1, \ldots, c_n \tag{1.7}$$

Somit können wir jeden einzelnen Koeffizienten wie folgt berechnen:

$$c_i = a_i + b_i \tag{1.8}$$

Im Code sähe das also, vereinfacht, folgendermaßen aus:

Listing 1.2: KoeffizientenPolynom.java

```
1  class KoeffizientenPolynom {
2          public Array koeffizienten;
3
4          public KoeffizientenPolynom addition(
               KoeffizientenPolynom summand) {
5                  KoeffizientenPolynom ergebnis = new
                       KoeffizientenPolynom();
6                  for  (int i = 0; i < summand.koeffizenten.
                       length, i++) {
7                          ergebnis.setObjectAtIndex(koeffizenten.
                               objectAtIndex + summand.
                               koeffizienten.objectAtIndex(i),i);
8                  }
9                  return ergebnis
10         }
11 }
```

Wie man also sieht, muss für jedes Koeffizientenpaar eine Addition durchgeführt werden. Daraus kann man schlißen dass bei einer Addition mit einem Polynom vom Grad n genau n Additionen durchgeführt werden müssen, was uns folgendes Ergebnis in der $\mathcal{O}$-Notation bringen würde:

$$\mathcal{O}(n) \tag{1.9}$$

1.2 Der Knackpunkt

Bisher können wir sehr zufrieden sein, da bekanntlich eine Lineare Laufzeit bereits ausreichend für größere Werte ist. Sehen wir uns nun an, wie es aussieht, wenn wir in ein Polynom einen Wert x_0 einsetzen. Eine naive Herangehensweise wäre nun das wir sämtliche Multiplikationen wie folgt berechnen:

$$p(x_0) = a_0 + a_1 \cdot x_0 + a_2 + x_0^2 + \cdots + a_n x_0^n \tag{1.10}$$

Für diese naive Heransgehenweise sollten wir uns die erwartete Laufzeitentwicklung ansehen. Dazu betrachten wir die erwartete Anzahl von Multiplikationen:

$$\frac{n\,(n+1)}{2} = \mathcal{O}\left(n^2\right) \tag{1.11}$$

Für mittlere Werte ist dieser Wert natürlich weniger von Nutzen. An dieser Stelle können jedoch optimieren, indem wir das Hornerschema einsetzen. Somit werden nur noch n

Additionen und n Multiplikationen benötigt, was wiederum in einer besseren Laufzeit von $\mathcal{O}(n)$ resultiert.

Würden wir den Algorithmus zur Addition auch für die Multiplikation einsetzen, und damit die folgende Rechnung durchführen:

$$c_i = \sum_{k=0}^{i} a_k \cdot b_i - k \tag{1.12}$$

mit $a_j = 0$ und $b_j = 0$ für $j > n$, so kämen wir an ein Problem. Damit wir ein c_i berechnen könnten, würden i Additionen und $i+1$ Multiplikationen nötig (es gelte $0 \leq i \leq n$). Weiter wäre zu erwarten, dass für $2n \geq i > n$, $2n-i$ Additionen *und* $2n-i+1$ Multiplikationen berechnet würden. Das bedeutet, es wäre eine Laufzeit von

$$\mathcal{O}(n^2) \tag{1.13}$$

zu erwarten!

1.3 Die Lösung

Wie man sieht, wird eine alternative Darstellung benötigt, die es uns erlaubt, einen Algorithmus für die Multiplikation von Polynomen zu verwenden, welcher besser ist als $\mathcal{O}(n^2)$. Eine bekannte Eigenschaft für ein Polynom ist, dass ein Polynom *eindeutig* dargestellt werden kann, da unter der Betrachtung von $z_0, z_1, \ldots, z_m$ mit $m+1$ paarweise verschiedenen Zahlen, $p(x)$ vom Grad $n \leq m$ ist und ferner p durch $p(z_0), p(z_1), \ldots, p(z_m)$ bestimmt werden kann. Dazu müssen wir $z_0, z_1, \ldots, z_m$ wie oben gezeigt, fixieren und müssen dann nur noch $p(z_0), p(z_1), \ldots, p(z_m)$ in einem Array speichern.

Listing 1.3: WertePolynom.java

```
1 class WertePolynom {
2         public Array werte;
3         // Operationen folgen
4 }
```

Wie man sieht, kann die Grundstruktur unserer alten Klasse beibehalten werden. Zunächst passen wir die Addition an. Dafür verwenden wir die Werte der Polynome, wie wir sie innerhalb des Arrays abgespeichert hatten. Wollen wir etwa das Polynom $r(x)$ erstellen so berechnen wir dies folgendermaßen:

$$r(z_i) = p(z_i) + q(z_i) \tag{1.14}$$

In Java hätten wir dies so wie im Listing 1.2 realisiert. Da wir genau m Additionen durchführen müssen erhalten wir für die Laufzeitabschätzung das Ergebnis (mit der Voraussetzung dass $m = n$):

$$\mathcal{O}(m) = \mathcal{O}(n) \tag{1.15}$$

Nun haben wir bereits das benötigte Mittel um eine Multiplikation von zwei Polynomen durchführen zu können. Das einzige was wir dafür machen müssen, ist die Addition durch eine Multiplikation zu tauschen:

$$r(z_i) = p(z_i) \cdot q(z_i) \tag{1.16}$$

bzw. in Java würden wir den folgenden Code verwenden:

Listing 1.4: WertePolynom.java

```
1  class WertePolynom {
2          public Array werte;
3          public WertePolynom addition(WertePolynom summand) {
4                  WertePolynom ergebnis = new WertePolynom();
5                  for  (int i = 0; i < summand.koeffizenten.
                      length, i++) {
6                        ergebnis.setObjectAtIndex(koeffizenten.
                           objectAtIndex + summand.
                           koeffizienten.objectAtIndex(i),i);
7                  }
8                  return ergebnis
9          }
10
11         public WertePolynom multiplikation(WertePolynom produkt
               ) {
12                 WertePolynom ergebnis = new WertePolynom();
13                 for  (int i = 0; i < summand.koeffizenten.
                      length, i++) {
14                       ergebnis.setObjectAtIndex(koeffizenten.
                           objectAtIndex * produkt.
                           koeffizienten.objectAtIndex(i),i);
15                 }
16                 return ergebnis
17         }
18 }
```

Die Multiplikation gilt, sofern $m = 2n$ ist. r ist nun eindeutig festgelegt worden, und erhalten die Laufzeitabschätzung $\mathcal{O}(2n)$.

Nun sollten wir uns eine kurze Übersicht verschaffen:

Operation	Koeffizientendarstellung	Wertedarstellung
Addition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Multiplikation	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Einesetzen von x_0	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$

Zwar ist unser Wunsch auf eine effektivere Multiplikation mit der Wertedarstellung erfüllt worden, jedoch lässt sich unser Trick mit dem Hornerschema dort nicht anwenden. Daher wird zwischen beiden Darstellungen gewechselt.

1.4 Wechsel zwischen den Darstellungen

Koeffizienten- $\longrightarrow$ Wertedarstellung Eine einfache und schnelle Weise um die Berechnungen durchzuführen ist die *schnelle Fouriertransformation*. Hierfür benötigen wir n Werte, die wir jeweils in $p(x)$ einsetzen. Der Aufwand dafür ist $\mathcal{O}(n^2)$, jedoch haben wir den Vorteil, dass wir die dafür benötigten Werte $z_0, \ldots, z_n$ selbst gewählt werden können! Wir definieren hierfür ω als eine primitive $n+1$ Einheitswurzel. Sie wird jedoch komplex dargestellt, und somit wird $\omega^{n+1} = 1$ und $w^k \neq 1$ für $k = 1 \cdot n$. Wir setzen nun $z_i := \omega^i$ für $i = 0, \ldots, n$. Wir wollen nun $p(\omega^0), p(\omega^1), \ldots, p(\omega^n)$ berechnen. Dazu nehmen wir nun an, dass $n+1$ eine Zweierpotenz sei. Alle anderen würden wir mit 0en auffüllen.

$$p(x) = \sum_{i=0}^{n} a_i x^i \tag{1.17}$$

$$= \sum_{i=0}^{\frac{n+1}{2}-1} a_{2i} x^{2i} + x \sum_{i=0}^{\frac{n+1}{2}-1} a_{2i+1} x^{2i} + x \tag{1.18}$$

$$= g(x^2) + x \cdot u(x^2) \tag{1.19}$$

Wir haben somit die Polynome g und u erstellt, die jeweils nur halbsoviele Koeffizienten haben, wie in p. Als letzten Schritt setzen wir in g und u die Einheitswurzeln $\omega^0, \omega^2, \omega^4, \ldots, \omega^{2n}$. Somit sind erhalten wir nur noch $\frac{n+1}{2}$ verschiedene Zahlen! Weiterhin gilt:

$$\omega^{n+1} = \omega^0 \tag{1.20}$$

$$\omega^{n+2} = \omega^2 \tag{1.21}$$

Wir erhalten somit eine Rekursion, und haben eine verbleibene Laufzeit von $\mathcal{O}(n \log n)$.

1.5 Algorithmen

Wir sollten uns, bevor wir mit den eigentlichen Betrachtungen anfangen, im klaren sein, was ein Algorithmus ist. Hierfür gibt es verschiedene Definitionen[19, S. 16][8, S. 162], wir wollen uns jedoch auf den folgenden Einigen:

Definition: Ein Algrithmus ist ein Rechenverfahren, welches eine Eingabe (die aus einem oder mehreren Werten besteht) in eine Ausgabe (bzw. ein oder mehrere Ergebnisse) überführt[4, S. 5]

Weiterhin erfüllt ein Algorithmus die folgenden Eigenschaften:

Endlich beschreibbar Der Algorithmus muss sich letztlich (zumindest theoretisch) programmieren und somit darstellen lassen, sodass die benötigten Zwischenschritte nachvollziehbar sind.

Effektiv Der Algorithmus sollte keine unnötigen Umwege gehen müssen, bzw. zumindest in „normaler“Zeit bearbeitbar sein

Allgmein Er darf sich nicht auf bestimmte Eingaben beschränken

Deterministisch Er soll, zumindest meistens, einen vorbestimmten Weg nehmen und auf die Eingaben möglichst immer den gleichen Weg gehen.

Für nähere Betrachtungen müssen wir Algorithmen jedoch noch weiter unterteilen. Dabei sollen die Algorithmen in die Kategorien *gut* und *schlecht* unterteilt werden. Die Regeln für uns, damit ein Algorithmus als „gut“ klassifizieren zu können:

Korrekt Der Algorithmus muss immer terminieren (darf also niemals in eine Endlosschleife kommen) und immer eine richtige Antwort liefern.

Schnell Natürlich darf ein einfacher Algorithmus zum beenden keine 100 Jahre bis zur Beendung brauchen.

Speichereffizient Die Zahl der zwischendurch benötigten Daten muss also so gering wie möglich gehalten werden.

Gut lesbar Der Algorithmus muss also möglichst schnell und möglichst ohne Hilfsmittel erfassbar sein.

1.5.1 Bewertung von Algorithmen

Da wir uns nun einig sind, wie Algorithmen klassifiziert werden können, müssen wir die Darstellung und unsere Mittel zur Klassifizierung definieren. Generell bleiben uns zwei große Möglichkeiten:

1. Experimentell: Wir können einen Algorithmus über verschiedene Testläufe mit verschiedenen Eingaben Testweise laufen lassen. Dies hat jedoch den Nachteil, dass wir letztlich beispielsweise die Korrektheit nicht endgültig nachweisen können. Auch ist die Geschwindigkeit und die Lesbarkeit abhängig, von der verwendeten Implentierung und Programmiersprache. Somit würden Bewertungen nicht allein auf Bezug des Algorithmus sondern unweigerlich auch auf die Programmiersprache/Implementation ausgeweitet.

2. Theoretisch: Wird der Code jedoch theoretisch abstrahiert, so lösen sich hier einige Probleme. Da dieser Code auch nicht vom Computer abhängig ist, auf dem er ausgeführt ist, wird ein standartisiertes, abstraktes Maschinenmodell verwendet. Weiterhin müssen keine festen verwendet werden, stattdessen gibt es die Möglichkeit sich dafür auf die $\mathcal{O}$-Notation zurück zu ziehen. Wir haben nun die Möglichkeit die worst-case Eingaben zu betrachten. Somit wählen wir dieses Mittel für unsere Betrachtungen.

1.5.2 Das Maschinenmodell - Die Registermaschine

Doch welches Maschinenmodell sollen wir hierfür verwenden? Wir benötigen vorallem eine Möglichkeit die Schritte zählen zu können und um den benötigten Speicherplatz messen zu können. Dies ist jedoch recht schwer, würden wir die Turingmaschine oder das λ-Kalkül bzw. den Markov Algorithmus[1] verwenden, wäre dies nur mittelmäßig möglich. Verwenden wir jedoch die Registermaschine (siehe Abbildung fig:RAM), so ist die Zahl der Schritte einfach abzählbar und der benötigte Speicherplatz ist leicht zu messen. Wie

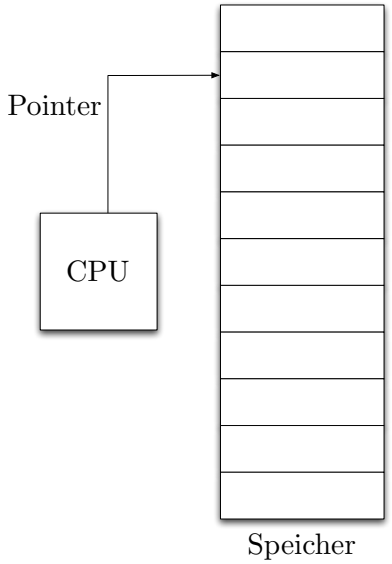

Abbildung 1.4: Die Registermaschine

man sieht, ist das Modell der Registermaschine (kurz: RAM) der Turingmaschine sehr ähnlich. Pro Speicherzelle kann eine ganze Zahl gespeichert werden, auch steht uns hierbei eine unbegrenzte Zahl von Speicherzellen zur Verfügung. Die CPU führt für uns die Operationen durch, und beherrscht die folgenden Arithmetischen Operationen:

- Addition
- Subtraktion
- Multiplikation
- Division
- Modulo

Weiterhin kann der Lesekopf absolut und relativ positioniert werden. Theoretisch könnten wir diese Maschine mit Assembler programmieren. Der Code, um beispielsweise das größte Element in einem Array zu finden, könnte wie folgt aussehen:

[1] http://de.wikipedia.org/wiki/Baum-Welch-Algorithmus

Listing 1.5: GreatestElement.asm

```
1  # Schreibe die Laenge des Arrays hinter die Eingabe
2          MOVE 0 -> [0]+1
3  # Initialisiere das aktuelle Maximum
4          MOVE[0] -> [0]+2
5
6  # Fange Schleife an, dekrementiere dabei die Laenge des Arrays
7  loop DEC [0]+1
8          JZ [0]+1 => ende # Es gibt keine weiteren Elemente mehr
               (die Laenge ist == 0)
9          JLE ([0]+1), [0]+2, skip # Das aktuelle Element <= das
              aktuelle groesste Element
10         MOVE [[0]+1] -> [0]+2
11 skip JMP loop
12 ende RETURN([0]+1)
```

Wir können nun die Anzahl der benötigten Schritte für einzelne Belegungen theoretisch durchrechnen, jedoch ist der Code auch nicht viel allgemeiner als würden wir uns den benötigten Code in Java ansehen. Auch kann das lesen von größeren Quelltexten sehr schwierig werden.

Verwenden wir stattdessen Pseudocode haben sogar noch den Vorteil, dass wir durch den leicht lesbaren Code auch gleich die Korrektheit überprüfen können. Der Pseudocode sieht für unser Beispiel wie folgt aus:

Listing 1.6: GreatestElement.pseudo

```
1  ArrayMax(length,A) {
2          currentMax <- A[length]
3          for counter:=length-1 downto 1 {
4                  if (A[counter] > currentMax) {
5                          currentMax <- A[counter]
6                  }
7          }
8          return currentMax
9  }
```

Zur Vereinheitlichung setzen wir folgende Regel fest: 1 Zeile Pseudocode soll konstant vielen RAM Anweiseungen entsprechen. Hierdurch kann beim Zählen der Schritte darauf verzichtet werden, das konstatnte Faktoren vorliegen, was uns die Arbeit vereinfacht. Wir drücken unsere Ergebnis nun, mathematisch Formal durch die $\mathcal{O}$-Notation aus.

1.6 Die O-Notation

Seien $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$, dann ist für den schlimmsten Fall:

$$f = \mathcal{O}(g) \quad \Leftrightarrow \quad \exists c > 0; n_0 \geq 1; \forall n \geq n_0 : f(n) \leq c \cdot g(n) \tag{1.22}$$

Somit zeigt die $\mathcal{O}$-Notation immer die oberste Grenze für eine Laufzeit an und ist somit der *worst-case*. . Für den besten Fall gilt:

$$f = \Omega(g) \quad \Leftrightarrow \quad \exists c < 0; n_0 \geq 1; \forall n \geq n_0 : f(n) \geq c \cdot g(n) \tag{1.23}$$

$$\tag{1.24}$$

Im Gegensatz zur $\mathcal{O}$-Notation, ist stellt die Ω-Notation also immer den besten Fall dar, den sogenannten *best-case*. Als letzten Fall gibt es:

$$f = \Theta(g) \quad \Leftrightarrow \quad f := \mathcal{O}(g(n)) \wedge f := \Omega(g) \tag{1.25}$$

Also zeigt die Θ-Notation den Fall, der zwischen $\mathcal{O}$ und Ω liegt an, den *average case*. Im folgenden Verwenden wir den Namen $\mathcal{O}$-Notation, für alle 3 Formen der Asymptotischen Darstellung.

Der Vorteil, den wir durch Anwendung der $\mathcal{O}$-Notation erhalten ist der, das wir hier die Laufzeit knapp, prägnant und unabhängig von den Details des Modells, angeben können.

Definition: Die wichtigsten Laufzeiten und ihre Namen lauten (von wünschenswert nach am wenigsten wünschenswert sortiert):

$\mathcal{O}(1)$ konstant

$\mathcal{O}(\log n)$ logarithmisch

$\mathcal{O}(n)$ linear

$\mathcal{O}(n \log n)$ quasilinear

$\mathcal{O}(n^2)$ quadratisch

$\mathcal{O}(n^3)$ kubisch

$\mathcal{O}(2^n)$ exponentiell

[12, S. 284-295][3, S. 41-56][4, S. 43-60]

1.7 Rekursionen

Eine bekannte Möglichkeit ein Problem zu lösen ist der Ansatz des *Divide and Conquer* (auch bekannt als *Teile und Herrsche Prinzip* oder *Divide et Impera*). Hierbei wird ein Problem in Teilprobleme unterteilt. Diese können dann schneller gelöst werden, die daraus resultierenden Teillösungen können dann zur Gesamtlösung zusammengesetzt werden.

Bekannte Beispiele hierfür sind:

- Fibonacci-Zahlen
- Binärer Baum

- Mergesort
- Quicksort
- Fakultät
- Türme von Hanoi

Am Beispiel der Fibonacci Zahlen stellen wir nun die Frage: „*Wie lange dauert es, um die n-te Fibonacci-Zahl zu berechnen?*“. Der bekannteste Ansatz ist der folgende:

Listing 1.7: Fibonacci1.pseudo

```
1 fib(n) {
2         if (n==0 OR n==1) {
3                 return 1
4         } else {
5                 return fib(n-1)+fib(n-2)
6         }
7 }
```

Wir können die Rekursion auch als Baum darstellen (Abbildung 1.5). Zu sehen ist welcher

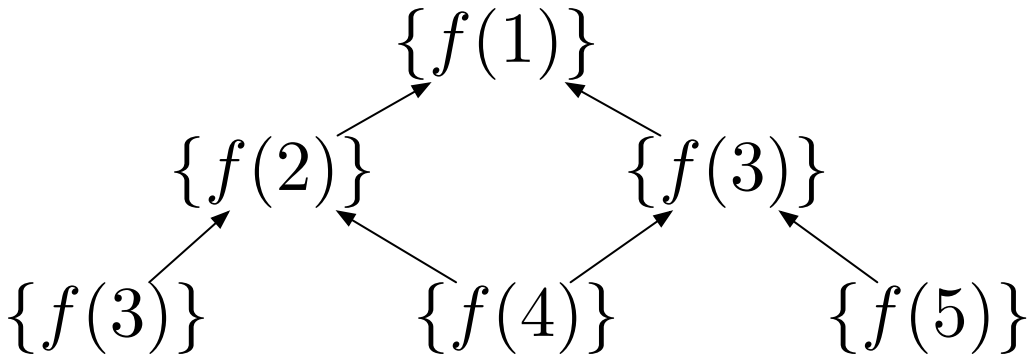

Abbildung 1.5: Beispielrekursionsbaum für die Berechnung der Fibonaccizahlen

Aufruf für die die jeweilige Zahl getan werden muss. So benötigt, wie man auch in den Quelltexten nachlesen kann, ist für die vierte Fibonacci Zahl sowohl die zweite, als auch die dritte berechnet werden.

Dabei können wir jeden Knoten als einzelnen Aufruf zählen und diese summieren. Eine Kante gilt wird zwischen den aufrufenden Funktionen gezogen. Weiterhin wird an die Knoten der jeweilige Aufwand für die Inkarnation geschrieben (in diesem Beispiel ist dies immmer $\mathcal{O}(1)$). Jedoch ergeben sich nun mehrere Fragen:

1. Wieviele Ebenen gibt es?
   Die Tiefe der Blätter ist zwischen n und $\frac{n}{2}$. Somit gibt es $\frac{n}{2}$ volle Ebenen!

2. Wie lautet der Aufwand in der i-ten Ebene?
   Sofern $i \leq \frac{n}{2}$ ist, lautet der Aufwand 2^i. Gilt $n \geq i > \frac{n}{2}$, dann ist der Aufwand $\leq 2^i$.

Damit wir nun den Aufwand abschätzen können, addieren wir den Aufwand pro Ebene über die Ebene:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\frac{n}{2}} 2^i && (1.26)\\ &= 2^{\frac{n}{2}} - 1 && (1.27)\\ &= \Omega\left(2^{\frac{n}{2}}\right) && (1.28)\\ T(n) &\leq \sum_{i=0}^{n-1} 2^i && (1.29)\\ &= 2^n - 1 && (1.30)\\ &= \mathcal{O}(2^n) && (1.31) \end{aligned}$$

1.7.1 Dynamische Programmierung

Wir erhalten also Exponentielle Laufzeit, diese ist jedoch nicht wünschenswert, da bereits für mittlere n die Rechenzeit sehr lang wird. Jedoch ist Auffällig, dass es Redundanzen gibt, da Werte berechnet werden, die bereits vorher berechnet wurden. Speichern wir diese Ergebnisse in einer Tabelle zwischen, so muss nur noch berechnet werden, was nicht in der Tabelle steht. Zwischenzeitlich berechnete n werden dann in die Tabelle eingetragen. Somit erhalten wir eine Laufzeit und einen Speicherverbauch von $\mathcal{O}(n)$, also ist beides linear. Noch effizienter ist wäre folgende Methode:

Listing 1.8: Fibonacci2.pseudo

```
1  fib(n) {
2          if (n==0 OR n==1) {
3                  return 1
4          } else {
5                  (a,b) <- (1,1)
6                  for (i = 2) to n {
7                          (a,b) <- (a+b,a)
8                  }
9                  return a
10         }
11 }
```

Da wir nur noch die letzten zwei Zwischenergebnisse zwischenspeichern, benötigen wir keine großen Arrays mehr, auf deren erste Elemente nicht mehr zurückgegriffen werden müssen. Somit ist zwar die Laufzeit weiterhin $\mathcal{O}(n)$, jedoch ist der Speicherverbrauch $\mathcal{O}(1)$. Wie man erkennt gibt es zwar kompliziertere Lösungen als rekursive Algorithmen,

jedoch sind diese generell besser als Rekursionen! Somit sollte, sofern möglich, nicht auf Rekursionen zurückgegriffen werden müssen.

2 Einfache Datenstrukturen

Als einfache Datenstrukturen bezeichnen wir vorallem die Strukturen die das Prinzip *FIFO* (*FIFO*) bzw. die *LIFO* (*LIFO*) benutzen. Bereits in dem Namen steckt bereits, in welcher Reihenfolge die einzelnen Elemente der Datenstruktur erreichbar sein sollen.

LIFO Die Elemente, die zuerst in die Struktur gegeben werden, werden insgesamt als erstes wieder erreichbar gemacht. Wir verwenden hierfür die *Queue*.

FIFO Die Elemente, die zuletzt in die Struktur gegeben wurden, werden als erstes wieder herausgenommen. Ein Beispiel ist der *Stack*

2.1 Stack

Der Stack ist die einfachste Datenstruktur. Die Grundidee ist, dass die Daten in einem Stapel nur übereinander gelegt werden. Dabei ist nur das Objekt erreichbar, welches zu letzt auf den Stack gelegt wurde, also oben auf liegt (Abbildung 2.1)

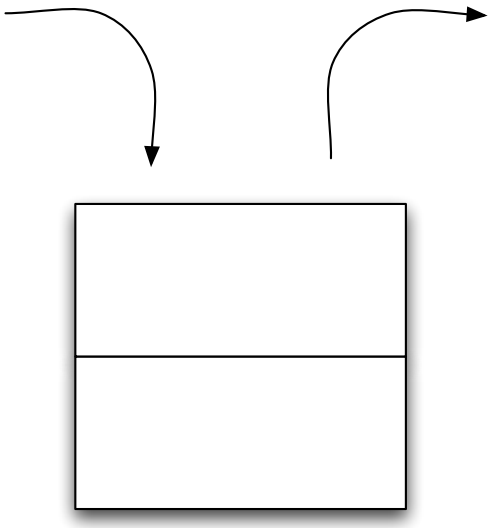

Abbildung 2.1: Ein Stack, zu sehen ist dass sich die Operationen jeweils nur oberste Element des Stapels beziehen.

Weiterhin können wir nun die Operationen bestimmen, die wir dringend für die Datenstruktur benötigen:

PUSH Legt e als oberstes Objekt auf den Stapel.
Parameter:

1. e - Das auf den Stapel zu legende Element

POP Das oberste Element des Stapels wird entfernt.
Rückgabewert:

- Das oberste, nun entfernte, Element des Stapels

SIZE Die Anzahl der Elemente die auf dem Stack liegen wird zurückgegeben.
Rückgabewert:

- Die Zahl der Elemente die aktuell auf dem Stack liegen

ISEMPTY Prüft, ob der Stack leer ist.
Rückgabewert:

- *falsch*, wenn die Anzahl der Elemente ≥ 1 ist, sonst *wahr*.

TOP Gibt das erste Element des Stacks zurück.
Rückgabewert:

- Das oberste Element des Stacks.

Diese Datenstruktur ist denkbar einfach mit Hilfe eines Arrays zu implementieren. Dabei muss nur gespeichert werden, welches das neueste Element des Stacks ist.

Listing 2.1: ArrayStack.java

```
1  class ArrayStack<E> {
2          E[] array;
3          int topElement;
4
5          // Den Konstruktur lassen wir der Einfachheit wegen weg
6
7          public void push(E element) {
8                  topElement += 1;
9                  array[topElement] = element;
10         }
11
12         public E pop() {
13                 if (topElement != 0) {
14                         topElement -= 1;
15                         return array[topElement+1];
16                 }
17         }
18
19         public boolean isEmpty() {
20                 return (0 == topElement);
```

```
21          }
22
23          public int size() {
24                  if (topElement == 0) {
25                          return 0;
26                  } else {
27                          return topElement;
28                  }
29          }
30
31          public E top() {
32                  return array[topElement];
33          }
34  }
```

An unserer Implementation gibt es leider zwei Probleme:

1. Zwar ist die Implementation sehr leicht zu realisieren. Jedoch ist die Größe des Stacks durch die maximale Größe des Arrays begrenzt. Bei der Erstellung des Stacks darf als größter Wert nur $2^{31} - 1$ gewählt werden, da dies der größte Integerwert ist und größere Datentypen für die Erstellung von Arrays in Java nicht zugelassen sind.

2. Auch sind Ausnahmesituationen noch nicht behandelt worden. Hat man beispielsweise keine Elemente im Stack, so ist es üblicher eine Ausnahme zu werfen. Wir würden also noch mindestens 2 Exception Klassen (StackFullException & StackEmptyException) schreiben müssen.

[4, S. 232-234][8, S. 188-203][19, S. 299-303]

2.2 Schlangen

Wie bereits beschrieben, ist die Schlange (engl. Queue) aufgabaut nach dem *FIFO*-Prinzip aufgebaut. Das Element, das als erstes in die Schlange gegeben wird, wird auch als erstes wieder ausgegeben (Abbildung 2.2)

2.2.1 Operationen

Die Operationen auf einer Warteschlange sind etwas komplexer. Hierbei müssen wir uns 2 Elemente merken, den Anfang und das Ende der Liste. Wir definieren als Anfang der Liste das Element das am frühsten hinzugefügt wurde, während das letzte Element der Liste das neueste sein soll. Die Operationen die wir nun benötigen sind:

ENQUEUE Füge e vor das letzte Element an, bewege den Pointer auf das letzte Element auf e.
Parameter:

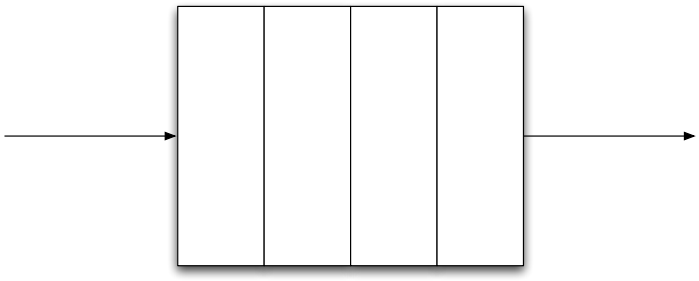

Abbildung 2.2: Skizze einer Warteschlange

1. e - Das auf den Stapel zu legende Element

DEQUEUE Entferne den Anfang der Liste, gebe jedoch den gespeicherten Weg zurück.
Rückgabewert:

- Das erste Element der Liste

SIZE Größe der Liste.
Rückgabewert:

- Anzahl der Elemente in der Schlange

ISEMPTY Prüft ob die Schlange leer ist.
Rückgabewert:

- *wahr*, wenn keine Werte mehr in der Schlange gespeichert werden, sonst *falsch*

FRONT Gibt das erste Element zurück.
Rückgabewert:

- Das erste Element der Liste

Auch eine Warteschlange kann natürlich mit einem Array realisiert werden, jedoch mit den gleichen Beschränkungen wie bei einem Stack.[4, S. 234f.][8, S. 204-212][19, S. 304-307]

2.3 Dynamische Arrays

Da eine Schlange auch über Arrays implementiert werden kann, hat dieser die gleichen Beschränkungen für große Listen. Diese kann jedoch auch recht speicherintensiv sein, wenn nur wenige Objekte darin gespeichert werden müssen, obwohl der Stack bzw. die Schlange für eine große Zahl von Elementen ausgelegt ist. Praktischer wäre es, einen Array zu entwickeln, der relativ klein anfängt, jedoch dann, je nach Bedarf wächst. Genau hier setzt die Lösung des Dynamischen Arrays an.

Sagen wir, es wird ein Array mit der Länge m eingerichtet. Nach einer gewissen Zeit, ist der Array nun so voll, dass $m - 1$ Elemente darin gespeichert werden. Nun wird ein

neuer Array angelegt, der doppelt so groß wie der alte ist, in diesen kopieren wir nun die Objekte aus dem kleineren Array und löschen ihn danach.

Listing 2.2: DynamicArray.java

```
1  class DynamicArray {
2          int m = 50;
3          int array[];
4          int lastObject;
5          public DynamicArray() {
6                  array = new int[m]
7                  lastObject = m;
8          }
9
10         public void add(int n) {
11                 if (lastObject+1 == array.length) {
12                         doubleListLength();
13                 }
14                 lastObject += 1;
15                 array[lastObject] = n;
16         }
17
18         public void doubleListLength() {
19                 int[] buffer = new int[array.length+m];
20                 for (int i = 0; i < array.length; i++) {
21                         buffer[i] = array[i];
22                 }
23                 array = buffer;
24         }
25 }
```

Natürlich ist m nach belieben anpassbar, weshalb dies auch je nach den erwarteten Datenmenge getan werden sollte. Die Frage ist jedoch, was für eine Laufzeit diese Verdopplung des Arrays hat. Dadurch dass das erstellen einer solchen Liste immer konstant viel Zeit benötigt gilt hier $\mathcal{O}(1)$. Da aber jedes Element kopiert werden muss, lautet die benötigte Laufzeit hier bei einem array mit n Elementen $\mathcal{O}(n)$

2.3.1 Amortisierte Analyse

Wir wollen nun ein Gedankenexperiment machen. Stellen wir uns vor, wir müssten pro hinzufügen Operation auf einem Array, einen Preis von 1 € zahlen. Wir haben jedoch einen Buchhalter dem wir zwar insgesamt pro Operation 3 € zahlen sollen, er jedoch die Differenz für uns auf ein Konto legt. Dabei soll die folgende Tabelle zeigen, wie sich das gesparte Geld das bei jeder Operation hinzugelegt wird summiert.

n	Angelegtes Geld in €
0	0
1	2
2	4
3	6
4	8
5	10
⋮	⋮

Wäre unser Array nun voll und die Daten würden in ein doppelt so großes Array verschoben werden müssen, so würde der Buchhalter nun das gesparte Geld verwenden, sodass wir nicht erneut zahlen müssten.

Die Amortisierte Analyse beim Stack

Wir wollen nun mit Hilfe unseres Buchhalters beweisen, dass jede Folge von m PUSH und POP Operationen immer $\mathcal{O}(m)$ Zeit benötigt.

Beweis: Da POP immer nur $\mathcal{O}(1)$ Zeit benötigt, besteht die Folge im schlimmsten Fall immer aus POP Operationen.
Würde der Buchhalter erneut 3 €pro Operation nehmen und einen Euro pro POP Operation benötigen, und den Rest in einem Array an der entsprechenden Stelle anlegen, so ergebe sich die Tabelle von oben. Auch hier wird ein dynamisches Array verwendet, welches k Elemente hat, so würde beim vergrößern ein Array mit der Größe $2 \cdot k$ angelegt. Somit werden jedoch nun nur noch $\frac{k}{2}$ PUSH-Operationen benötigt.

Nun stellt sich nur noch die Frage, wieviel Geld hierdurch gespart wurde. Die Antwort ist recht simpel, da sich nun ergibt:

$$\frac{k}{2} \cdot 2 \quad = \quad k \tag{2.1}$$

Hierdurch wurde nun Geld für das Verdoppeln gesammelt und für die PUSH-Operationen muss nun gezahlt werden:

$$3m\ e \quad = \quad \mathcal{O}(m) \tag{2.2}$$

Also Amortisieren sich die nun die Kosten pro Einzahlung. Also werden die hohen Kosten beim erstellen, durch spätere, niedrigere Kostenwieder ausgeglichen. Dieses Prinzip nennt man *Amortisierte Analyse* [4, S. 456-459]

2.4 Guter Softwarentwurf

Nachdem wir uns nun mehrfach über die Laufzeit von konkreten Algorithmen angesehen, etwa bei Operationen auf konkreten Datentypen, stellt sich nun die Frage, wodurch sich guter Code auszeichnet. Hierbei muss bereits der Softwareentwurf bestimmte Ziele verfolgen.

Wartbarkeit Unser Code muss sich leicht an neue Bedingen anpassen lassen. Dafür sind dann meist Veränderungen nötig, daher müssen diese sich leicht und schnell realisieren lassen.

Flexibilität Wichtig zu verstehen ist jedoch, dass solch tiefgreifende Änderungen nicht erst bei der Wartung auftreten müssen. Bereits während der Entwicklungsphase können sich Dinge ändern, etwa weil es neue Erkentnisse gibt. Daher müssen sich Implementierungsentscheidungen leicht ändern lassen.

geringe Fehleranfälligkeit

Übersichtlichkeit

2.4.1 Das Geheimnisprinzip

Ein bekanntes Mittel hierfür ist das *Geheimnisprinzip*. Hierbei trennen wir die Implementation eines Datentyps von seiner Definition. Wir wollen also so weit wie möglich für einen Datentypen nur noch eine Schnittstelle (auch *API* genennt) bereitstellen, die sich vollkommen neutral zu jeder Implementierung verhält.

2.4.2 Abstrakte Datentypen

Um nun diese Schnittstellen vorplanen und definieren zu können müssen wir uns von unseren echten Datentypen trennen und stattdessen *Abstrakte Datentypen* (kurz: ADT) entwickeln, die letztlich für unsere Schnittstelle die Definitionen der Methoden bereitstellen. Das hat die folgenden Vorteile:

Sicherheit

Flexibilität

Komfort

2.4.3 Das Geheimnisprinzip in Java

Es ist ganz hilfreich, sich die die Konzepte der Objektorientierten Programmierung erneut anzusehen, schließlich ist Java eine derartige Programmiersprache und wir wollen die Funktionen weiter ausnutzen.

- Ziel ist es, die Entitäten der realen Welt zu modellieren.
- Dabei repräsentieren die Objekte (also die Modelle) eine Menge von Attributen & Methoden, die letztlich zusammen gehören.
- Objekte sind Exemplare einer definierten Klasse
- Diese Objekte haben jeweils ein Innenleben und eine Identität.

Nicht zu vergessen ist, dass bestimmte Eigenschaften bzw. Methoden versteckt werden können, sodass sie von außen nicht erreichbar sínd. Dies nennt man Kopplung. Weiterhin lassen sich Klassen von anderen Klassen ableiten. Diese übernehmen (auch als *erben* bezeichnet) erlaubt es, die Definition der oberen Klasse zu erweitern. Das Typsystem in Java unterstützt somit auch *Polymorphie*.

Da es jedoch keine Mehrfachvererbung in Java gibt, haben wir nun ein Problem. Hierbei helfen uns jedoch die Interfaces weiter: Java erlaubt es uns mit Hilfe von abstrakten Objektklassen die Schnittstellen zu definieren. Hierfür gibt es zwei Stufen. Die erste und einfachste Schnittstelle ist das *Interface*. Diese erlaubt es, Methoden vorzudefinieren ohne sie zu implementieren. Da eine Klasse mehrere Interfaces implementieren kann, hilft sie uns in diser Misere. Im folgenden Beispielcode lässt sich das Interface für die Schlange ansehen (die gleichzeitig Generics verwendet um sie mit allen Datentypen kompatibel machen zu können).

Listing 2.3: Queue.java

```
1 public interface Queue<E> {
2         void enqueue(E element) throws QueueFullException;
3         E dequeue() throws QueueEmptyException;
4         E front() throws QueueEmptyException;
5         int size();
6         boolean isEmpty();
7 }
```

Die Implementierung ist nun entsprechend leicht vorzustellen.

Die Frage ist nun nur noch, wie äußert sich in der Programmierung *Guter Stil*? Hierbei gilt es nur eine Faustregel zu beachten: Während Variablen nur mit dem Interfacetyp deklariert werden sollten, so sind die konkreten Typen nur für die Erzeugung von Objekten zu benutzen (dies wäre mit dem Interface sowieso eh nicht möglich). Eine Ausnahme sollte sein, wenn eine spezielle Implementation weitere Methoden bereitstellt, die im Interface nicht definiert wurden. Somit sollte der Code normalerweise ungefähr so aussehen:

```
1 Queue<Integer>queue = new ArrayQueue<Integer>(100);
```

2.5 Die Prioritätswarteschlange

Nun soll die Queue aus dem Listing 2.3 um Prioritäten erweitert werden. Dies lässt sich im ersten Augenblick sehr leicht erledigen. Wir schreiben dazu ein neues Interface namens LNUQueue (für: *Low*, *Normal*, *Urgent*-Queue)

Listing 2.4: LNUQueue.java

```
1 enum Prio = {LOW, NORMAL, URGENT};
2
3 interface LNUQueue<E> extends Queue<E> {
4         void enqueue(E element, Prio p) throws
              QueueFullException;
```

```
5         void reset();
6 }
```

Somit wurde das alte Queue Interface um Prioritäten und die benötigten Methoden erweitert. Die Idee hinter der Schlange ist nun, dass die Standardpriorität *Normal* für jedes Element beim hinzufügen sei (sofern nicht anders angegeben). Reset soll die Schlange zurücksetzen. Da hierfür jedoch neue Klassen nötig sind, stellt sich nun die Frage ob es eine Möglichkeit gibt, die neuen Klassen zu programmieren ohne dabei, sehr viele Redundanzen einzubauen. Hier hilft uns ein weiteres Java Konzept weiter.[8, S. 320-332]

2.5.1 Abstrakte Klassen

Mit Hilfe der *Abstrakten Klassen* ist es möglich, einen Teil der Klassen bereits zu implementieren (während ein *Interface* dies nicht zulässt!), bestimmte Methoden jedoch der konkreten Klasse überlassen. Somit sieht der Code der abstrakten Klasse nun so aus:

Listing 2.5: AbstractLNUQueue<E>

```
1  private Queue<E> low, normal, urgent;
2
3  public void enqueue(E element, Prio p) throws
       QueueFullException {
4          switch(p) {
5                  case URGENT: urgent.enqueue(element); break;
6                  case NORMAL normal.enqueue(element); break;
7                  case LOW: low.enqueue(element); break;
8                  default: break;
9          }
10 }
```

Wir legen also nicht eine große Schlange an, sondern verwenden drei kleinere Schlangen, je nach Priorität(Abbildung 2.3). Die Behandlung der Exceptions muss *enqueue* somit nun nicht mehr machen, da die Exception weitergereicht wird, an die aufrufende Klasse und somit nun problemlos funktionieren würde.

2.5.2 Factory Pattern

Sehen wir uns jedoch die *reset* Funktion einmal an. Eine Möglichkeit wäre es, für alle Schlangen solange dequeue aufzurufen, bis diese leer sind. Dies kann bei vollen Schlangen jedoch recht Zeitaufwändig werden, da die Schlange somit sehr voll werden kann. Einfacher wäre es, einfach 3 neue Schlangen zu erstellen und diese zu speichern. Somit wird das Aufräumen der Garbage Collection überlassen. Das Problem ist jedoch, dass wir nicht wissen, welche Art von Queue hier verwendet werden soll. Außerdem ist es verboten Code wie den folgenden zu verwenden:

```
1 normal = new Queue<E>()
```

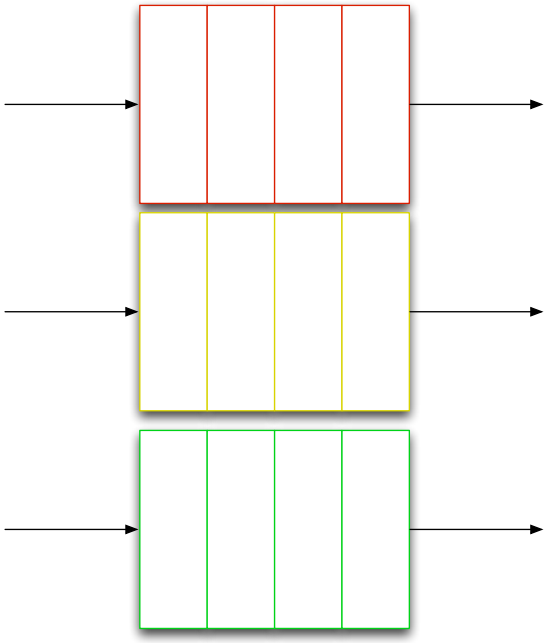

Abbildung 2.3: Die Prioritätswarteschlange.

Zur Erinnerung: Queue ist nur eine Schnittstelle, jedoch keine Implementation. Somit ist die Zuweisung verboten.

Wir verwenden daher ein Entwurfsmuster: das Factory Pattern. Hierbei wird, wie schon im Namen enthalten, eine Fabrik eingerichtet, welche die Erstellung der Objekte vornimmt. Somit können wir nun für *reset* folgende Implementierung vornehmen:

Listing 2.6: AbstractLNUQueue.java

```
1         public void reset() {
2                 low = buildQueue(LOW);
3                 normal = buildQueue(NORMAL);
4                 urgent = buildQueue(URGENT);
5         }
```

Wie man erkennt, benötigen wir eine neue Methode, die für uns das erstellen der Schlangen vornimmt. Sie wird wie folgt programmiert:

Listing 2.7: AbstractLNUQueue.java

```
1 protected abstract buildQueue(Prio p);
```

Wir überlassen die Implementation der buildQueue Funktion also der jeweiligen Unterklasse, da zu diesem Zeitpunkt nicht klar ist, welche Queue verwendet werden soll! [7, S. 110-170]

Bei einem Design Pattern handelt es sich um wiederverwendbare Lösungen für wiederkehrende Probleme im Software Entwurf. In der Objektorientierten Programmierung ist damit auch eine bestimmte Form von Beziehungen und Interaktionen gemeint. Dies sind Erfahrungen und Best Practices des Software Entwurfs. Dabei wird gleichezeitig die Termininologie gegeben um über einen Klassenentwurf nachzudenken und zu kommunizieren.

2.6 Abstrakte Datentypen

Im letzten Abschnitt haben nun eine Queue um drei Prioritäten erweitert. Diese Art von Schlange wird unter anderem verwendet bei:

- Scheduling
- Terminplanung
- Kürzeste Wege
- minimal aufspannenden Bäumen
- Sweepline Algorithmen

Wenn wir einen Abstrakten Datentyp spezifizieren wollen, muss uns klar sein, wie wir angeben können, welche Operationen durchgeführt werden sollen. Hier gibt es zwei Möglichkeiten:

verbal Wir beschreiben seine Funktion: „*Ein Datentyp aus einem total geordnetem Universum*“

Operationen Wir beschreiben, welche Operationen der Datentyp können soll und was beim Aufruf passieren soll: „*findMin() gibt das kleinste Element zurück, die Datenstruktur bleibt unverändert. Ist die Datenstruktur jedoch leer, wirf eine Exception.*“

2.6.1 Die Prioritätswerteschlange als Abstrakter Datentyp

Es soll nun besprochen werden, wie am Beispiel der Prioritätswarteschlange ein Abstrakter Datentyp definiert werden kann:

verbal Speichere Elemente aus einem total geordnetem Universum.

Operationen

- findMin()

  Vorraussetzung Datenstruktur $\neq \emptyset$

  Effekt Gibt das kleinste Element aus der Dataenstruktur zurück, lässt jedoch die Datenstruktur unverändert.

- deleteMin()

  Vorraussetzung Datenstruktur $\neq \emptyset$

Effekt Das kleinste Element wurd am Ende der Operation zurückgegben und aus der Datenstruktur entfernt.

- insert(x, y)

  Vorraussetzung -

  Effekt Fügt ein neues Element x mit der Priorität y in die Datenstruktur ein.

- isEmpty()

  Vorraussetzung -

  Effekt Gibt *true* zurück, wenn die Datenstruktur leer ist.

- size()

  Vorraussetzung -

  Effekt Gibt die Anzahl der Element in der Datenstruktur zurück.

Sofern sie benötigt wird, kann an dieser Stelle auch noch eine *Invariante* angegeben werden. Unter einer Invariante verstehen wir eine Bedingung, die über die gesamte Lebenszeit der Datenstruktur gilt.

Vorteile	Nachteile
Leicht verständlich	Viel Text
Allgemein gehalten	Kann unterspezifiziert sein
einfach	Kaum formal, eignet sich kaum für Beweise oder Computerprogramme

Unter einer *Unterspezifikation* verstehen wir, dass der abstrakte Datentyp (gewollt oder ungewollt) so offengehalten ist, dass er womöglich nicht so sicher beschrieben ist, dass für die Implementation alle benötigten Informationen bereitstehen. Somit können etwa Effekte auftreten, die bei der Spezifikation hätten ausgeschlossen werden müssen.

2.6.2 Modellierende Spezifikation

Eine weitere Möglichkeit für die Spezifikation von Abstrakten Datentypen ist die *Spezifikation!modellierend*. Hierzu wird entwickeln wir ein abstraktes Modell und beschreiben den Abstrakten Datentyp an Hand dieses Modells.

mathematisch Sei U ein total geordnetes Universum. Die Objekte seien endliche Multimengen $S \subseteq U$, Operationen sind *findMin, deleteMin, ...*, jedoch werden nun mathematisch beschrieben:

- findMin()

  Vorraussetzung $S \neq \emptyset$

  Effekt liefere $\min(S)$

- deleteMin()

Vorraussetzung $S \neq \emptyset$

Effekt liefere $\min(S)$, lösche $\min(S)$

programmiert Wir definieren hier mit Hilfe von Haskell: $(O, d(t)) \Rightarrow [t]$

- findMin: $(O, d(t)) \rightarrow (t, [t])$
- findMin xs

  Vorraussetzung $xs \neq []$

  Effekt minimum(xs)

  Hierbei ist es nicht wichtig, dass der Code letztlich wirklich Compilieren würde, wir nehmen ihn nur zur Definition. Wichtiger ist es, dass der Code eindeutig und für außenstehende Verständlich ist. Ist der Code jedoch wirklich ausführbar, so wäre dies natürlich hilfreich und schön, jedoch letztlich keine Vorraussetzung für diese Art der Spezifikation.

```
1 insert (O,dt) => t->[t])->[t]
2 -- Vorraussetzung: --
3 insert (x,xs) = x:xs
```

Vorteile	Nachteile
sehr genau	erfordert Vorkenntnisse
gut zu formalisieren und prüfen	zum Teil umständlich
kurz und knapp	*Überspezifizierungen* können hier auftreten

Unter der *Überspezifizierung* verstehen wir im Gegensatz zur *Unterspezifizierung*, dass die Spezifikation so genau geschrieben ist, dass sich nur bestimmte Implementierungen hierfür nutzen lassen oder verschiedene Dinge, die eigentlich möglich sein sollten, hierdurch verboten werden.

Eine weitere Möglichkeit zu Spezifikation stellt die algebraische Spezifikation dar. Hierbei bauen wir kein Modell mehr, sondern geben nur noch axiomatische Beziehungen zwischen den Operationen an. Diese müssen dann gelten. Beispielsweise für den Stapel ergibt sich somit:

- push$(S, x) \rightarrow S$
- pop $S \rightarrow S$
- top $S \rightarrow (x, S)$
- emptyStack S

Die Axiome spziefizieren wir nun wie folgt:

- pop(push$(S, x)) = S$

- top(push(S, x)) = (push(S, x),x)

Der Vorteil dieser Spezifizierung ist, dass sie zwar theoretisch sauber und gut formalisiert ist, jedoch sehr eine sehr hohe Abstraktion bietet. Somit kann die Spezifikation schnell kompliziert werden, je nachdem welche Axiome gewählt wurden. Somit eignet sich diese Art der Spezifikation komplexere Datentypen kaum noch.[19, S. 255-274]

2.7 Eine verkettete Liste mit mehreren Ebenen

Bei der Implementation der Prioriätswarteschlange gibt es mehrere Möglichkeiten. Zwei naive Ansätze wären beispielsweise:

- unsortierte verkettete Liste
- sortierte verkettete Liste

Bei der sortierten, verketteten Liste, verwenden wir als Parameter für das sortieren natürlich die Priorität der einzelnen Elemente.

Eine weitere Möglichkeit jedoch ist es, eine Liste mit mehreren Ebenen hierfür zu benutzen. Dazu müssen wir nur folgendes machen:

1. Fixiere einen Parameter m
2. Speichere die Elemenet als Folge von verketteten Listen. Wenn wir fertig sind, sollen sämtliche Listen (bis auf eine) eine Länge von *genau* m haben. In den einzelnen Listen sollen die jeweils kleinsten Elemente zu einer weiteren Liste verlinkt werden (siehe Abbildung 2.4).
3. Wenn die Struktur nun n Elemente hat, dann lautet die Anzahl der Listen l.

$$l := \left\lceil \frac{n}{m} \right\rceil \tag{2.3}$$

Nun müssen wir noch die jeweiligen Operationen beschreiben:

- insert(x): Füge x in die kürzeste Liste ein (bzw. lege eine neue an, wenn kein Platz an bei den aktuellen Listen frei ist), aktualisiere dann das jeweilige Minimum der jeweiligen Liste und passe entsprechend die Minimum-Liste.
- findMin(): durchlaufe die Minumliste, gib dann das kleinste Element zurück. Die Laufzeit hierfür lautet $\mathcal{O}(1)$.
- deleteMin(): Finde das kleinste Element (siehe findMin), lösche das kleinste Element der Liste und passe die MinimumListe erneut an. Die erwartete Laufzeit lautet: $\mathcal{O}(1 + m)$
- size(), isEmpty(): Zähle die Zahl der Elemente (möglichst in einem Attribut). Die Laufzeit ist auch hier: $\mathcal{O}(1)$

Wenn mit n die maximale Zahl von Elementen in der PriorityQueue bekannt ist, können wir $m = \sqrt{n}$ wählen. Somit haben alle Operationen nur noch die benötigte Laufzeit von $\mathcal{O}(\sqrt{n})$.

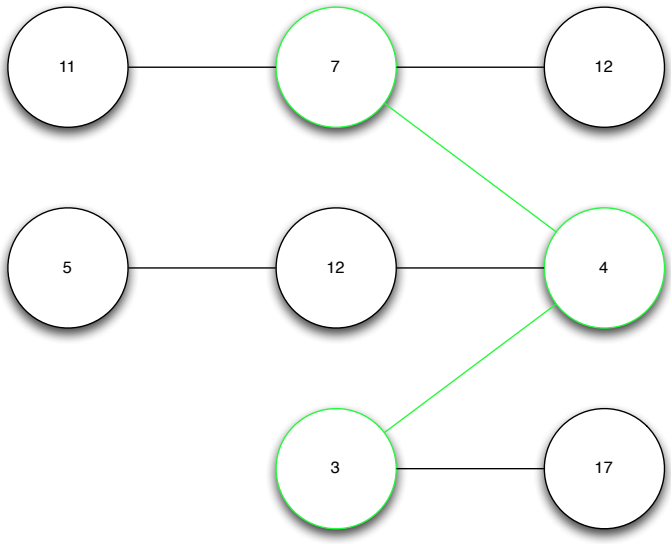

Abbildung 2.4: Eine verkettete Liste mit 3 Ebenen. Die grün markierten Elemente sind wiederum die jeweils kleinsten Elemente und deren Verkettung

2.8 Binärer Heap

Unter einem Binäreren Heap versteht man einen (zumeist) vollständigen, binäreren Baum. Die unterste Ebene, wird zuerst vollkommen aufgefüllt, bevor eine weitere begonnen wird. Unter Ausnutzung dieser Eigenschaft, lässt sich ein binärer Heap in einem linearen Array repräsentieren. Wir lassen jedoch das nullte Element der Liste leer und schreiben dort hingegen die größe (also die Anzahl der Elemente) des Heaps hinein. Somit kann die Position des linken Unterknoten des n-ten Knotens berechnet werden als $2 \cdot n$ bzw. der rechte mit $2 \cdot n + 1$ berechnet werden.

```
1 public int parent(int n) {
2         return n/2;
3 }
4 public int left(int n) {
5         return n*2;
6 }
7 public int right(int n) {
8         return 2*n+1;
9 }
```

Wir können zwei verschiedene Heaps erstellen. Der MinHeap und der MaxHeap, die sich an ihrer Sortierung unterschieden. Während beim MinHeap im Wurzelknoten eines Baumes das kleinste Element und die folgenden Knoten von den Werten immer größer werden. Beim MaxHeap ist es genau anders herum. Eín neues Element wir an der letzten freien Stelle eines Baumes hinzugefügt, woraufhin dann der Heap sortiert wird. [4, S.

151-169][3, S. 455-475]

2.9 Exkurs: Onlinealgorithmen

[1]

3 Wörterbücher

Eine weitere, *Datenstruktur*, die häufig benutzt wird ist, ist das Wörterbuch. Hierbei wird einem Element einer Schlüsselmenge $k \in K$ ein jeweils ein Wert v zugeordnet. Somit ergibt sich hierdurch für alle gespeicherten Werte ein Tupel (k, v). Dazu werden der Einfachheit wegen, ganze Zahlen verwendet (siehe Abbildung 3.1).

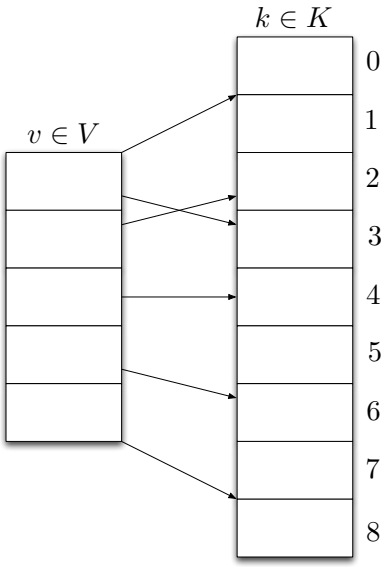

Abbildung 3.1: Eine einfache Zuweiseung zwischen Werten einer Wertemenge $v \in V$ und einem Schlüssel einer Schlüsselmenge $k \in K$.

Wie gewohnt, betrachten wir zunächst die Operationen des Datentyps an.

PUT Einem Schlüssel k wird der Wert v zugeordnet.

GET Der Wert, der dem Schlüssel k zugeordnet wurde, wird zurückgegeben.

REMOVE Der zu k gehörige Wert, wird gelöscht.

In Pseudo-Haskell programmiert ergibt sich nun:

```
1 put k v = elements[k] <- v -- O(1)
2 get k = elements[k] -- O(1)
3 remove k = elements[k] <- 0 -- O(1)
```

Wir haben jedoch zwei Probleme:

1. K kann sehr groß sein
2. K muss letztlich nicht aus Zahlen bestehen

Um all diesen Problemen entgegen zu wirken, setzen wir fest, dass die Schlüsselmenge K ausschließlich aus ganzen Zahlen besteht und nur begrenzen seine Größe. Somit benötigen wir letztlich eine Funktion h, die die Schlüssel k generiert. Diese Funktion nennt sich Hashfunktion h.

3.1 Hashfunktion

Wir wollen zunächst annehmen, wir wären uns über Verrfahren, welches h benutzt bereits einig. Daher entscheiden wie nun das Wörterbuch implementiert werden soll. Dies ist dann natürlich entsprechend leicht:

```
1 put k v = elements[h(k)] <- v -- O(1)
2 get k = elements[h(k)] -- O(1)
3 remove k = elements[h(k)] <- 0 -- O(1)
```

3.1.1 Kollisionsvermeidung

Nun ist es jedoch möglich, dass $h(k_1) = h(k_2)$ irgendwann auftritt. Somit hätten wir eine Kollision, und ein neuer Wert v_2 würde einen alten Wert v_1 überschreiben, der an der gleichen Stelle gespeichert wird. Damit wäre jedoch v_1 verloren gegangen. Somit müssen wir uns eine andere Implementation überlegen und können hieran auch die Hashfunktion entwickeln.

1. *Verkettung*: Die Elemente, die bei elements(i) gespeichert werden in einer verketteten Liste
2. *offene Adressierung*: Wen bei elements($h(k)$) bereits ein Wert steht, somit die berechnete Adresse also besetzt ist, so finde einen neuen Platz
3. *Kuckuck*: Ist der Platz bei elements($h(k)$) belegt, so schaffe dort Platz.

Bei allen Strategien, besteht die *Hashtabelle* aus einem Array der Länge n, der zugehörigen *Hashfunktion* und einer Strategie zur Kollisionsvermeidung.

3.1.2 Wahl der Funktion

Wir wollen nun jedem Schlüssel $k \in K$ einen Wert aus $\{0, \ldots, N-1\}$ zuordnen um somit die Zahl der der Kollisionen möglichst gering zu halten. k soll dazu möglichst zufällig gewählt sein. Somit soll eine mögliche Struktur, die sich hierdurch ergeben könnte, bereits im entstehen aufgelöst werden sollen. Dazu wird $h(k)$ in 2 Schritten berechnet:

1. Ein Hashcode wird für k berechnet, wir weisen also dem Schlüssel k eine ganze Zahl zu.
   Hierbei ist Java bereits bei den mitgelieferten Objekten hilfreich, da bereits in der Oberklasse *Object* die Funktion *hashCode()* implementiert ist. Sie übergibt in der ursrpünglichen Definition so implementiert, dass die Adresse der Speicheradresse des Objekts zurückgibt. Somit ist der Hash Code natürlich einmalig.da an jeder Speicheradresse nur ein Objekt liegen kann. Unter Umständen kann es jedoch von Nöten sein, diese Funktion zu überschreiben.

   Eine weitere Möglichkeit ist die Kompressionsfunktion, die einen Hashwert auf $\{0, \ldots, n-1\}$ abbildet. Dabei soll die Eingabe möglichst gut gesterut werden, um die Wahrscheinlichkeit einer Kollision zu minimieren. Um die Zahlen auf $n-1$ begrenzen zu können, verwenden wir die die Modulo Operation.

   $$h(z) \quad \rightarrow \quad z \mod n \tag{3.1}$$

   Dieser Ansatz ist zwar einfach, streut in der Realität aber nicht genügend. Daher nehmen wir nun eine beliebige Primzahl p mit der Vorraussetzung $p > n$:

   $$h(z) \quad \rightarrow \quad (z \mod p) \mod n \tag{3.2}$$

   Heuristisch gesehen, ist dieser Ansatz besser als der Ansatz in (3.1), jedoch wäre ein noch besserer Ansatz, wenn weiterhin $p > N$ gelte, wir aber zwei Variablen a und b wählen würden mit $a, b \in \{0, \ldots, p-1\}$ mit $a \neq 0$. So wäre es Möglich zu berechnen:

   $$h(z) \quad \rightarrow \quad ((a \cdot z + b) \mod p) \mod n \tag{3.3}$$

   Wir gehen davon aus, dass die Berechnung von h $\mathcal{O}(1)$ Zeit beansprucht. Somit haben wir nun eine (für unsere Zwecke) angebrachte Hashingfunktion gefunden und müssen uns nun nur noch um den zweiten Schritt Gedanken machen.

2. Auftretende Kollisionen behandeln

3.1.3 Kollisionsbehandlung

Wir sehen uns nun die Lösungsansätze bei Kollisionen etwas genauer an.

Verkettung

Eine Liste *elements* speichert die Einträge (k, v) an der Stelle $h(k) = i$ ab. Dafür wird bei *put*, die Liste der Schlüssel nach k durchsucht. Falls $h(k)$ bereits belegt wurde, so wird es an die Liste angefügt (siehe Abbildung 3.2). *get* durchsucht *elements* nach k an $h(k)$ und gibt dann den Wert zurück. *delete* löscht den Eintrag k. Der Platzbedarf für diesen Ansatz ist $\mathcal{O}(N + \text{Zahl der Einträge})$.

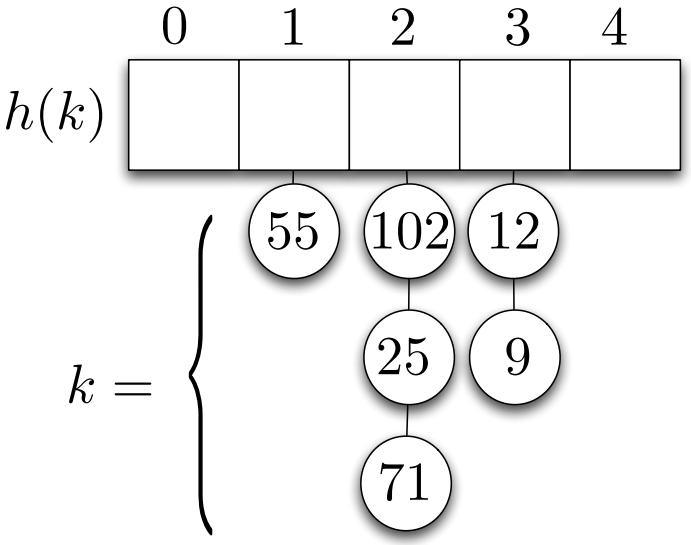

Abbildung 3.2: Eine Bildliche Darstellung der Verkettung für Wörterbücher.

Nehmen wir an, die Hashfunktion h verhält sich wie eine zufällige Funktion $k \in K$ wobei durch die Hashfunktion ein Wert $i \in \{0, \ldots, n-1$ berechnet wird. Die Wahrscheinlichkeit für einen bestimmten Wert beträgt nun:

$$\Pr(h(k)) \quad = \quad \frac{1}{n} \tag{3.4}$$

Stellen wir uns weiter vor, dass beim Anlegen der Hashtabelle ein h zufällig gewählt wird, wobei die genaue Wahl geheim sei, und für die nächsten Belegungen unbekannt sei. Diese würden dann unabhängig von den Vorherigen Belegungen hinzugefügt. Somit ergibt sich der Platzbedarf

$$\mathcal{O}(N + |S|) \tag{3.5}$$

S sei die aktuelle Menge von Einträgen, k ein fester fester Schlüssel, auf den die nächste Operation ausgeführt wird. Die Laufzeit lautet somit nun:

$$\mathcal{O}(1 + L) \tag{3.6}$$

Wir wollen uns nun einigen, um die Wahrscheinlichkeit von Kollisionen zu berechnen:

$$l_{k'} = \begin{cases} 1 & \text{falls } h(k') = h(k) \\ 0 & \text{sonst} \end{cases} \tag{3.7}$$

$$\rightsquigarrow \tag{3.8}$$

$$L \leq 1 + \sum_{k' \neq K, (k', k) \in S} l_{k'} \tag{3.9}$$

$$E[l_{K'}] = \mathcal{O} \cdot \Pr[h(k')] + 1 \cdot \Pr[h(k)] \tag{3.10}$$

$$= h(k) \tag{3.11}$$

$$= \Pr[h(K') = h(k)] \tag{3.12}$$

Wir bezeichnen $\frac{|S|}{N}$ als *Belegungsfaktor* (engl. *load factor*). Wenn der Belgegungsfaktor $\mathcal{O}(1)$ ist, bedeutet dies gleichzeitig, dass für den schlechtesten Hashingwert in der Verkettung erwartete die Laufzeit $\mathcal{O}(1)$ ist. In der Praxis will man einne Belegungsfaktor von 0,75 haben (dies lässt sich am besten mit Hilfe eines dynamischen Arrays realisieren).

Im nächsten Schritt wollen wir von der Annahme abweichen, dass h zufällig gewählt wurde. Dies ist in der realität eher unpraktisch. Kann h zufällig aus einer viel kleineren Menge von Funktionen gewählt werden, so ist der Beweis weiterhin gültig. Man nennt dies *Universelles Hashing*

Vorteile	Nachteile
einfach	benötigt Platz für die Listen
schnell	
es gibt einen Erwartungswert	

Bildlich gesehen, können die verketteten Listen innerhalb des Arrays auch als Eimer gesehen werden, in dem die weiteren Informationen einfach geworfen werden. Man spricht daher häufig auch von *Bucket Arrays*.[8, S. 273f.]

Offene Adressierung

Das Grundprinzip hinter dieser Idee ist, dass wir einen neuen Platz suchen, wenn der ursprünglich gewünschte belegt ist. Wir benötigen für das folgende Beispiel eine Liste. Sie soll 8 Elemente groß sein (siehe Abbildung 3.3). Um einen Schlüssel zu platzieren, berechnen wir nun den Modulo:

$$5 = 13 \mod 8 \tag{3.13}$$

$$1 = 9 \mod 8 \tag{3.14}$$

Somit ergibt sich nun die Belegung, wie sie in Abbildung 3.4 gezeigt wird. Nun soll der Schlüssel 85 hinzugefügt werden:

$$5 = 85 \mod 8 \tag{3.15}$$

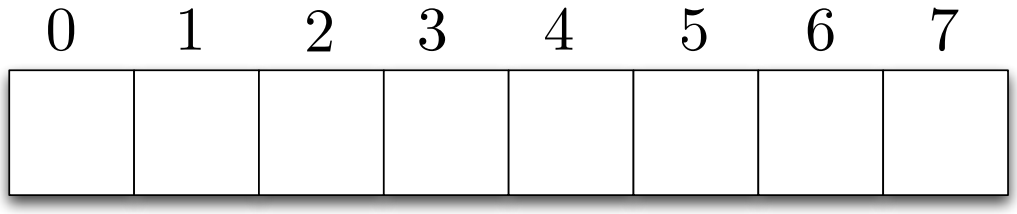

Abbildung 3.3: Eine einfache Liste

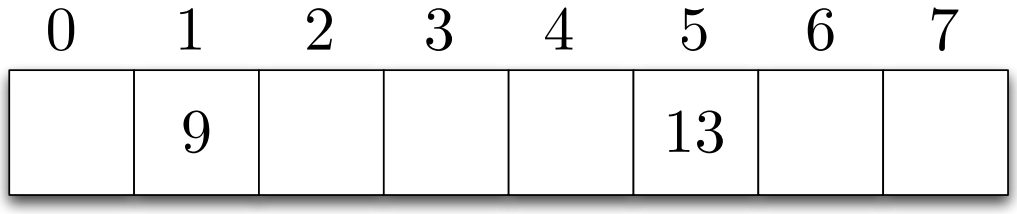

Abbildung 3.4: Die Zwischenbelegung der Schlüssel

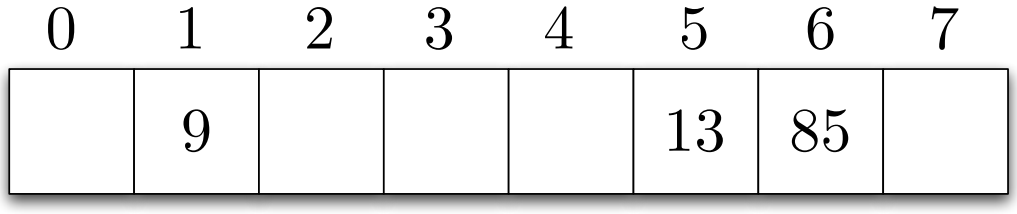

Abbildung 3.5: Neubelegung bei offener Adressierung: Da die 5 bereits mit der 13 belegt ist, wird die 85 auf das nächste, freie Feld gelegt

Somit würde 85 den Schlüssel 13 überschreiben, da sich beide Zahl modulo 8 5 ergeben. Stattdessen suchen wir bei der offenen Adressierung das nächst freie Feld und schreiben den Wert dort hinein (Abbildung 3.5). Wollen wir nun nach dem Wert für den Schlüssel 85 suchen, so berechnen wir erneut den modulo. Wir erhalten wieder die 5, sehen aber, dass auf dem Schlüsselarray an Stelle der 5 die 85 nicht abgelegt wurde also suchen wir die nächsten Stellen ab. Bereits auf *elements*(6) werden wir fündig. Wir können also solange Elemente suchen, bis wir am Ende des Arrays angelangt sind oder auf ein freies Feld stoßen, da bis hierhin zumindest der Schlüssel hätte abgelegt werden müssen. Somit kann der Schlüssel nicht gespeichert worden sein.

Beim Löschen von Elementen müssen wir jedoch nun vorsichtig sein. Statt die Stelle des Arrays komplett zu leeren müssen wir uns hier eher auf ein Füllzeichen einigen. Würden wir beispielsweise die 13 aus dem Array oben löschen, so wäre die 85 nicht mehr erreichbar. Einigen wir uns jedoch auf einen Platzhalter, so könnten wir dennoch weitersuchen, da hier zumindest vorher ein Element gestanden haben muss. Somit müssen wir über dieses Element hinwegsuchen. Einige Bemerkungen zur *Offenen Adressierung*[3, S. 237-244][4, S. 269-276]

1. Diese Methode nenn sich *lineares sondieren*(engl. *linear probing*)
2. Beim Löschen von Elementen können Lücken entstehen, die falsch verstanden werden können. Eine Lösung ist das Verwenden eines speziellen Eintrags *avail*, der als Platzhalter fungiert.
3. $|S| \leq |N|$ muss gelten (dazu wird ein dynamisches Array benötigt!)

Vorteile	Nachteile
geringer Platzbedarf	löschen ist kompliziert bzw. kann Fehler verursachen
	Clusterbildung von übrig bleibenden Elementen

Der *Clusterbildung* kann jedoch entgegen gearbeitet werden, indem sie regelmäßig neu aufgebaut wird. Dazu wird lettzlich eine amortisierte Laufzeit von $\mathcal{O}(1)$ erwartet.

Kuckuck

Soll das Element n_2 in $elements[h(k)]$ eingefügt werden, obwohl dort bereits n_1 liegt, so entferne n_1, lege an $h(k)$ n_2 ab. Finde für n_1 eine neue Stelle und lege es dort ab. Da dieses Verhalten an den *Kuckuck* erinnert, der seine Eier in fremde Nester legt, ist diese Methode nach ihm benannt.

Wir realisieren diese Kollisionsbehandlung, indem wir statt nur einer Hashfunktion, zwei Funktionen h_1 und h_2 implementieren. Möchten wir also einen Wert v_1 hinzufügen, so prüfen wir zunächst $elements[h_1(k_1)]$. Ist die Position frei, so können wir das Element dort einfügen. Liegt jedoch ein Wert v_2 an der Stelle, so nehmen wir v_2 aus der Liste und fügen stattdessen v_1 ein. Dafür werten wir nun den anderen Hash von k_2 aus und

beginnen damit von vorn. Um einen Kreisschluß zu verhindern hören wir nach $|elements|$ einfüge-Operationen auf.

Der Vorteil hier ist, das die $get()$ und $remove()$ Operationen nur an maximal 2 Stellen prüfen müssen ob das Element vorliegt, unabhängig von der Größe von $elements$. Das ergibt im schlimmsten Fall eine Laufzeit von $\mathcal{O}(1)$. Ist der Belegungfaktor klein genug und h_1, h_2 zufällig gewählt, so benötigt $put(k, v)$ sogar nur die erwartete, amortisierte Zeit von $\mathcal{O}(1)$.[16]

Während kleinere Schlüsselmengen somit sehr gut (teilweise sogar optimal) bearbeitet werden können, so sind große Universen eher eine Schwäche diese Hashingmethode.[5]

3.2 Iterator Pattern

Wir benötigen nun ein Konzept, das uns hilft die Einträge geordnet durchlaufen zu könen. Dabei soll die Möglichkeit gegebn werden, dass auch andere Objekte die gespeicherten Elemente durchlaufen können. Ein naiver Ansatz wäre es, die Elemente nur als Liste zurück geben zu können. Das widerspräche jedoch dem Geheimnisprinzip. Stattdessen wollen wir das Konzept der Iteratoren als Entwurfsmuster verwenden. Hirbei handelt es sich um eine neue Klasse, die die gespeicherten Objekte geordnet durchläuft und die Informationen bereitstellt. Dies ermöglicht es, dass wir in der konkreten Datenstruktur, für das jeweilige Objekt, weitere Hintergrundinformationen speichern, ohne dass diese dem Anwender unbedingt zugänglich gemacht werden. Somit ist das Geheimnisprinzip gewahrt.

Würden wir zunächst den Iterator direkt in die Dictionary Klasse implementieren, so wäre der Ansatz zunächst Lauffähig. Müssten wir jedoch an verschienden Stellen gleichzeitig (also durch verschiedene Klassen) zugreifen, so würde die aktuelle Position des Iterators jeweils überschreiben und für die anderen Klassen unbrauchbar. Daher lagern wir den Iterator in eine gesonderte Klasse aus. Somit können wir soviele Iteratoren erstellen, wie wir benötigen, da sie jeweils nur lesen müssen, besteht auch keine Gefahr, dass die Daten sich gegenseitig korumpieren. Java stellt für die Iteratoren bereits einige Schnittstellen bereits, somit ist es ein leichtes das Kozept letzlich zu verwenden.[7, S. 315-383]

3.3 Geordnetes Wörterbuch

Der abstrakte Datentyp des Wörterbuchs soll nun erweitert werden: unser Fokus liegt nun auf dem geordneten Wörterbuch. Hier gelten weiterhin die Bezeichnungen der Variablen k und v. Das Universum K sei nun aber total geordnet! Das Augenmerk soll nun auf der Speicherung von $S \subseteq K \times v$ liegen. Operationen wie $put()$, $get()$, $remove()$ sind mit ihren Definitionen weiterhin gültig. Jedoch kommen die folgenden Operationen nun hinzu:

min() Bestimme den kleinsten Schlüssel in S.

max() Bestimme den maximalen Schlüssel in S.

succ(k) Bestimme den kleinsten Schlüssel $k' \in S$ mit $k' > k$, also den Nachfolger von k.

pred(k) Bestimme den größten Schlüssel $k' \in S$ mit $k' < k$, also den Vorgänger von k.

Für die Implementierung eignet sich die Hashtabelle weniger, da *succ*() und *pred*() jeweils $\mathcal{O}(n)$ Zeit benötigen. Stattdessen wird eine sortierte, verkettete Liste verwendet.

3.3.1 Skiplisten

Für die Implementierung soll eine Hierachie gespeichert werden. Wir verketten hier jedes zweite Element erneut (s. Abbildung 3.6). Wird nun beispielsweise das Element 25

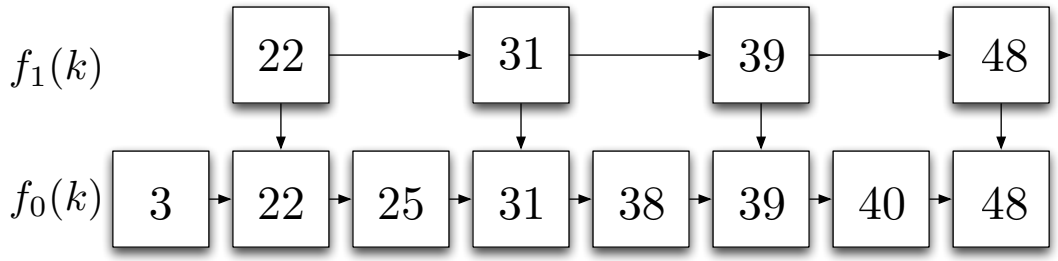

Abbildung 3.6: Eine einfache Skiplist. Hier ist jedes zweite Element erneut verkettet.

gesucht, so suchen wir in der Liste $f_1(k)$. Wir finden die Werte 22 und 31. Da 25 dazwischen liegt, müssen wir nun nur noch eine Ebene in den Listen nach $f_0(k)$ absteigen. Nun werden die dazwischen liegenden Werte kontrolliert. Hier treffen wir auf 25 und haben somit den gesuchten Schlüssel gefunden.

Einfacher wird das Suchen jedoch, wenn wir weitere Ebenen einziehen. So können wir bei $f_2(k)$ jedes vierte Element speichern, in $f_3(k)$ jedes achte und so weiter. Nun müssen wir jeweils nur das gesuchte Element umschließen. Finden ihn zwischenzeitlich können wir uns das weitersuchen in den untergeordneten Ebenen natürlich sparen.

Eine andere Idee zur Beleegung ist es, jeweils den Zufall bestimmen zu lassen ob der Schlüssel in die Ebene gespeichert wird oder nicht. Hierfür wird ein symbolischer Münzwurf durchgeführt:

```
1  add(k) {
2          wenn (!elements[k] existiert) {
3                  fuege k in ebene0 hinzu
4                  ebene <- 1
5                  solange (i = 0) {
6                          i <- zufallszahl % 2
7                          inkrementiere ebene
8                          fuege k in ebene hinzu
9                  }
10         }
11 }
```

Da die Liste geordnet ist, ist es uns letztlich egal, in welcher Ebene wir suchen, stattdessen müssen wir die Werte nur umschließen. Spätestens in der Ebene $f_0(k)$ finden wir dann den Wert (siehe Abbildung 3.7). [8, S. 398-406][17]

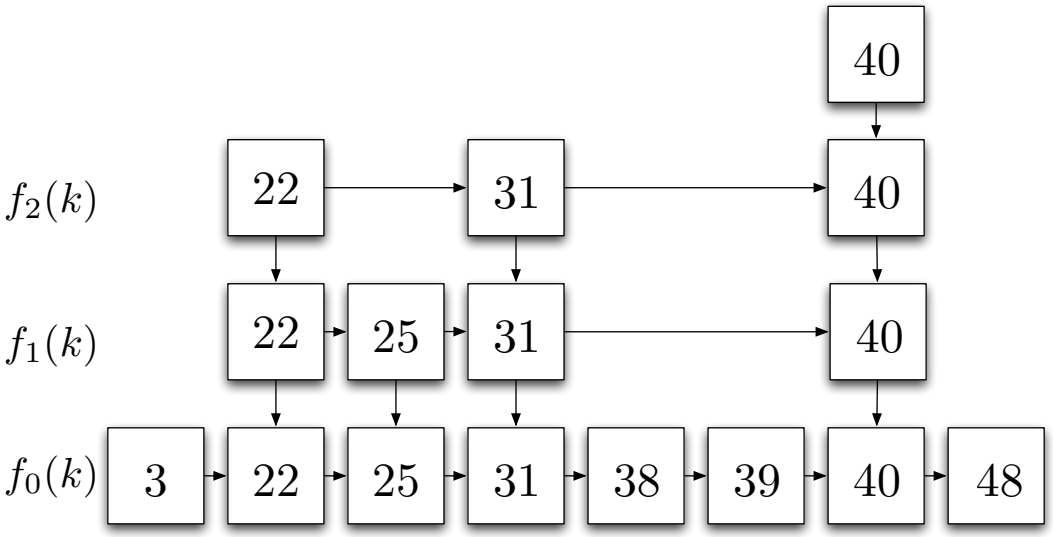

Abbildung 3.7: Eine Skipliste mit zufälliger Belegung.

4 Bäume

Nachdem wir uns zunächst nur mit linearen Datenstrukturen beschäftigt haben, die nur nach der jeweiligen Implementation über verschiedene Ebenen wuchsen, wollen wir uns nun mit echt mehrdimensionale Datentypen beschäftigen: *Bäume*. Hierbei handelt es sich um eine um eine geordnete Verkettung von Knoten eines Graphens, der wiederum, keine Kreise enthalten darf. Einen derartigen Graphen nennt man daher auch *zyklenfrei*.[15, S. 260]. Ein Beispiel ist in der Abbildung 4.1 zu sehen.

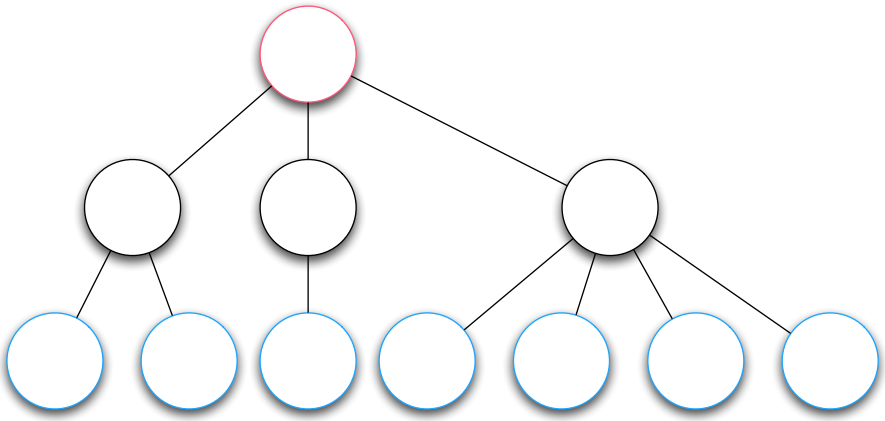

Abbildung 4.1: Ein einfacher Baum. Zu sehen ist die Wurzel sowie die Blattknoten.

Wie man erkennt, ist der Baum hieraisch aufgebaut, dass heißt, sofern es einen Nachfolger gibt, so ist er eindeutig als Nachfolger des Knotens definitiert. Der Heap ist dabei eine Spezialversion eines Baumes. Normalerweise ist die Anzahl der Kinderknoten in einem Baum nicht beschränkt. Der Heap ist jedoch hier eingeschränkt, da jeder Knoten pro Ebene nur maximal 2 Kinderknoten haben darf und dabei ausgeglichen sein muss!

4.1 Bezeichnungen

Wir führen für Bäume die folgenden Bezeichnugnen ein:

Wurzel Von diesem Knoten aus, wird der Baum aufgebaut. Er ist in der untersten Ebene (meist als Ebene 0 bezeichnet) des Datentyps angeordnet, wird jedoch in Zeichnungen meist oben eingezeichnet. Per Definition ist nur ein Wurzelknoten pro Baum erlaubt. Daher heißen Bäume mit einzelner Wurzel auch *gewurzelte Bäume*. Wir bezeichnen die Wurzel im folgenden mit r ($\equiv$ Root)

Blatt Hierbei handelt es sich um die äußere Knoten eines Baumes, die jedoch kein Wurzelknoten sind. Wenden wir die Symbolik eines realen Baumes an, so sind die auch hier die Blätter, die an den Ästen hängen und somit die Enden der Bäume darstellen. Der Baum hat somit keine Kinder. Um anzeigen zu können, dass es keine Kinderknoten gibt, sondern die Kanten jeweils auf NIL zeigen, verwenden wir als Symbol $\perp$.

Knoten Zwei Elemente eines Baumes werden als Knoten bezeichnet. Hierbei wird die Bezeichnung von den Graphen übernommen.

Pfad Die Beziehung zwischen zwei Knoten wird durch einen Pfad dargestellt. In der Graphischen Darstellung handelt es sich hierbei um einen Strich zwischen zwei Knoten. In der Bezeichnung würde eine Kante $\{u, v\}$ bedeuten, dass sie zwischen den Knoten u und v gezogen wird.

Suchbaum Baum in dem Schlüsselwerte (im Folgenden k genannt) gespeichert werden. Dabei sind die Wege geordnet, so dass klar ist, welche Pfade verfolgt werden müssen, wenn ein Schlüssel k gesucht wird.

Wald Ein Wald ist eine Sammlung von mehreren Bäumen.

Ausgeglichener Baum Ein Binärbaum ist ausgeglichen, wenn er bei n gespeicherten Elementen eine logrithmische Höhe hat.[19, S. 351]

Entarteter Baum Ein Baum der n Elemente hat, jedoch die Höhe $> \log n$ ist.[19, S. 352]

Höhe Die Höhe eines Baumes sei definiert als die Anzahl der Knoten, des längesten Weges von einem Knoten k bis r.

4.2 Wiederholung: Heap

Wie bereits zuvor besprochen ist der *Heap* ein spezieller Baum, mit besonderen Eigenschaften.

- Jeder Knoten darf nur 2 Kinderknoten besitzen.
- Der Heap ist immer ausgeglichen

Aufgrund dieser Eigenschaften lässt sich der Heap in einem linearen Array darstellen, ohne dass erst durch uns eine Konstruktion des Baumes als Klasse vorher geschehen muss. Somit lässt sich der Heap schnell und einfach implementieren und somit für einfache Bäume gut verwenden und stellt nach einiger Übung auch eine gute und schnelle Methode zum Sortieren von Elementen dar.

Sei $\{u, v\}$ nun eine Kante, sodass u im Baum näher an r liegt als v. Dann heißt u Elternknoten von v, während v der Kinderknoten von u ist.

4.2.1 Geordnete Wöterbücher als binäre Suchbäume

Die Einträge werden in den Knoten des Baumes gespeichert. Dabei hat jeder Knoten genau 2 Kinder, wobei hierfür auch $\perp$ als Möglichkeit in Betracht gezogen werden kann.

Definition Die Zahl aller Schlüssel im linken Teilbaum ist $\leq k \leq$ die Zahl aller Schlüssel im rechten Teilbaum.

Wir betrachten von nun an verschiedene Arten von Binären Suchbäumen. Wir beginnen zunächst mit einer verallgemeinerten Version des Binären Suchbaums, welche wir im Laufe der Zeit verfeinern.

4.3 Operationen

Wir definieren die Operationen des Binären Suchbaums wie folgt:

get(k) Lade einen Wert aus dem Baum. Ist der gesuchte Wert kleiner als der aktuelle Knoten, so suche links weiter, ansonsten gehe einen Schritt nach rechts. Ist man letztlich an einem Blattknoten angekommen, ohne auf den gesuchten Wert k getroffen zu sein, so werfe eine entsprechende Exception.

```
1  n <- root
2  while (n != NIL) {
3          if (n.k == k) {
4                  return(n.v)
5          }
6          if (n < n.k) {
7                  n <- n.left
8          } else {
9                  n <- n.right
10         }
11         throw NoSuchElementException
12 }
```

put(k,v) Analog zur get() Funktion, jedoch wird bei erfolgloser Suche ein neues Blatt eingefügt.

```
1  if (root == NIL) {
2          root <- node(k,v)
3          return
4  }
5  n <- root
6  while (true) {
7          if (n.k == k) {
8                  n.v <- v
9                  return
10         }
```

```
11          if (k < n.k) {
12                  if (n.left == NIL) {
13                          n.left <- (node(k,v))
14                          return
15                  } else {
16                          n <- n.left
17                  }
18          } else {
19                  if (n.right == NIL) {
20                          n.right <- node(k,v)
21                          return
22                  } else {
23                          n <- n.right;
24                  }
25          }
```

remove(k) Die get() Funktion wird benötigt um den zu löschenden Knoten zu suchen. Ist diese ein Blatt wird er einfach entfernt. Hängt ein Kinderknoten an ihm, so wird k durch den Kinderknoten ersetzt. Befinden sich zwei Kinderknoten an k, so wird im linken Teilbaum nach dem Vorgänger (also dem rechtesten Knoten) k' gesucht. Dieser wird als Blatt gelöscht und ersetzt letztlich. k.

```
1  n <- get(k):
2  if (n.left == NULL && n.right == NULL) {
3          k = NULL;
4  }
5  else if (n.left == NULL && n.right != NULL) {
6          k = n.right;
7  } else if (n.left != NULL && n.right == NULL){
8          k = n.left;
9  } else {
10         k' <- previous(k);
11         k = k';
12         k' = NULL;
13 }
```

[3, S. 253.272][4, S. 286-307]

4.4 Beispiele

4.4.1 Laufzeitbetrachtung

Die erwartete Laufzeit lautet $\mathcal{O}$(Höhe des Baumes). Dies lässt sich daran erkennen, dass wir den Baum sortiert haben. Dadurch ist klar, ob der aktuelle Wert größer oder kleiner

als der Wert des gesuchten Knotens ist. Somit sind die anderen Wege direkt zu ignorieren und können erspart werden. Der gesuchte Wert liegt im Blattknoten, wodurch die maximale Anzahl der zu durchsuchenden Knoten durch die Höhe des Baumes begrenzt ist.

4.5 Das Problem der Degenerierung

Das Problem ist jedoch, dass ein Baum degenrieren kann. Das bedeutet, dass er bei n Elementen eine Höhe von $\Omega(n)$ hat. Ein Beispiel kann in Abbildung 4.2 angesehen werden.[19, S. 352]

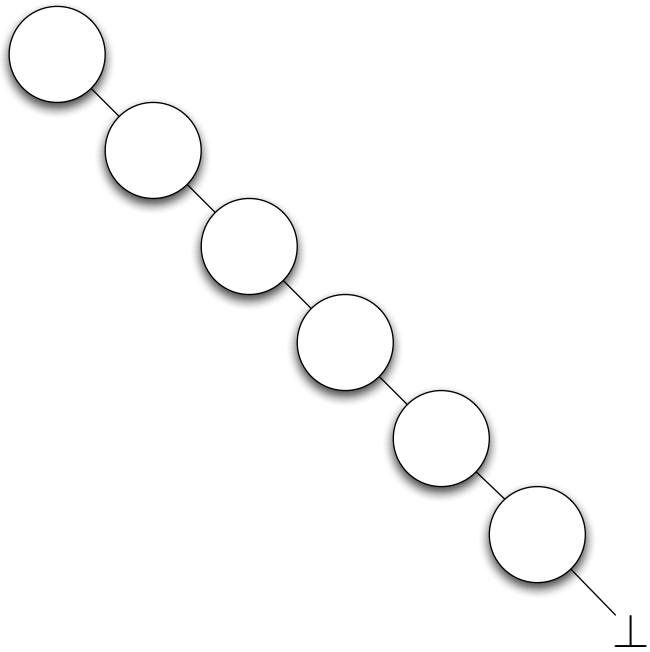

Abbildung 4.2: Ein entarteter Baum.

Entsprechend ergibt sich jedoch eine schlechtere Laufzeit, da das Suchen der Weerte nun länger dauern kann. Zwar könnte man dem schon vorbeugen, indem man die Werte zwar optimiert einfügt, jedoch kann nicht davon ausgegangen werden, da diese rechtzeitig zur Verfügung stehen.

4.6 AVL-Bäume

Wir wollen nun einen Baum entwickeln, der sich selbst optimiert. Ähnlich wie zu den Onlinealgorithmen in Kapitel 2.9, soll sich der Baum, beim Einfügen oder löschen neuer Knoten, selbst optimieren.

Ein perfekter Suchbaum hingegen hat eine Höhe von $\mathcal{O}(\log n)$. Dies würde bedeutet, dass alle Ebenen voll belegt sind. Wir müssen nun uns einen Weg überlegen, wie wir diesen Baum generieren können. Der Ansatz hier ist, dass wir put und remove so modifizieren, dass nach dem Einfügen des eines Elements der Baum jederzeit bestimmte Eigenschaften erfüllt und somit perfekt ist.

Würden wir die Eigenschaften und Vorraussetzungen des Binären Suchbaumes unverändert verwenden, so müsste der Baum nach jeder modifizierenden Operation neustrukturiert werden. Dies kann, je nach Veränderung, jedoch viele weitere Operationen vorraussetzen, da dieser stark umstrukturiert werden muss. Da wir dadurch nicht effizient sicherstellen können, dass der Baum jederzeit perfekt ist, müssen wir das Problem von anderer Seite angehen. Hierfür lockern wir die Anforderungen an den Baum so sehr, dass der Baum immer eine Höhe von $\mathcal{O}(\log n)$ hat, aber uns noch genügend strukturelle Flexibilität bietet, sodass die Operationen put() und remove() weiter effizient bleiben. Ein Beispiel ist der *AVL-Baum*.

Diese Bäume wurden 1962 von den russichen Forschern Georgy Adelson-Velskii und Evgenii Landis entwickelt.

AVL-Bäume sind *höhen-balancierte Binäre Suchbäume*. Dies bedeutet, dass wir für einen beliebigen Innenknoten v in T nehmen können. In diesem unterscheidet sich die Höhe der beiden Unterbäume um höchstens 1 (siehe Abbildung 4.3). Zur Erinnerung: Die Höhe des leeren Baumes wurde mit -1 definiert. Wir erinnern uns: Die *Höhe eines*

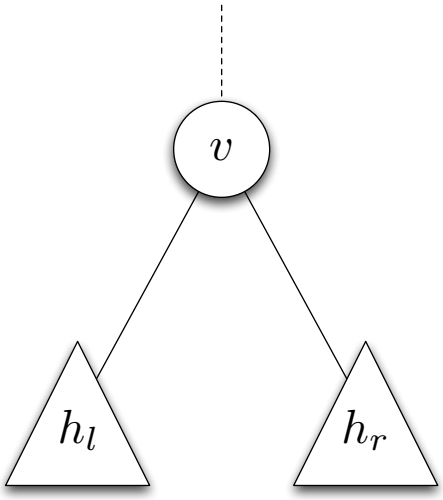

Abbildung 4.3: Ausschnitt aus einem beliebigen *AVL-Baum*. Es wurde ein innerer Knoten v gegriffen. Es gilt: $|h_l - h_r| \leq 1$.

Knotens wird bestimmt, indem die Zahl der besuchten Knoten auf dem Weg hoch zur Wurzel r gezählt wird. Somit hat r die größte Zahl, während Blätter eine Höhe von 0 haben (siehe Abbildung 4.4).

Hierdurch werden AVL-Bäume fast so gut wie perfekte Bäume.

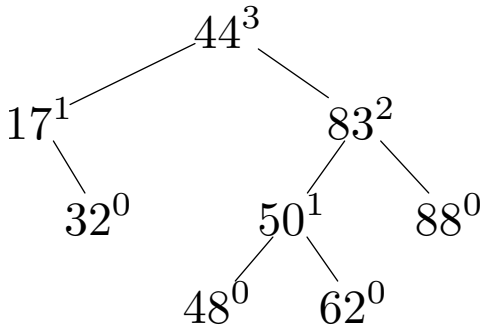

Abbildung 4.4: Ein AVL-Baum. Die Höhe der Knoten steht an den jeweiligen Knoten. Zu beachten ist, dass die Höhe eines Knotens jeweils der größte Wert ist, wodurch die Differenz zwischen der Höhe des Wurzelknotens r und der Höhe des linken Teilbaums von r erklärt wird.

Satz: Die Höhe eines *AVL-Baumes* mit n Knoten ist $\mathcal{O}(\log n)$.
Beweis: Sei n_h die minimale Anzahl der Knoten die ein *AVL-Baum* der Höhe h enthalten kann. So ergibt sich zunächst:

$$n_0 = 1 \tag{4.1}$$
$$n_1 = 2 \tag{4.2}$$
$$\vdots \tag{4.3}$$
$$n_h = 1 + n_{h-1} + n_{h-2}; h \geq 2 \tag{4.4}$$

Die Behauptung sei nun:

$$n_h \geq 2^{\lceil \frac{h}{2} \rceil} \tag{4.5}$$

Wir beweisen diesen Satz nun durch Induktion.

$$n_0 = 1 \geq 2\checkmark \tag{4.6}$$
$$n_1 = 2 \geq 2\checkmark \tag{4.7}$$
$$n_h = 1 + n_{h-1} + n_{h-2} \geq 2^{\lceil \frac{h-1}{2} \rceil} + 2^{\lceil \frac{h-2}{2} \rceil} \tag{4.8}$$
$$\geq 2 \cdot 2^{\lceil \frac{h}{2} \rceil - 1} \tag{4.9}$$
$$= 2^{\lceil \frac{h}{2} \rceil} \tag{4.10}$$
$$\rightsquigarrow n \geq 2^{\lceil \frac{h}{2} \rceil} \Rightarrow \log n \geq \frac{h}{2} \tag{4.11}$$
$$\Rightarrow h \leq 2 \cdot \log n = \mathcal{O}(\log n) \tag{4.12}$$

□

Bemerkung: Tatsächlich ist h sogar:

$$\frac{1,44}{\log 2} \log n \tag{4.13}$$

4.6.1 Rotationen

Doch wie stellen wir nun sicher, dass unser Baum T höhenbalanciert bleibt? Hierzu speichern wir mit jedem Knoten die Höhe des zugehörigen Teilbaums. Dies ist in Java Objekten sehr leicht zu realisieren, die Werte hingegen können dann schnell mit put() und remove() aktualisiert werden. Somit muss nur noch der Fall $|h_l - h_r| \geq 2$ behandelt werden. Ein Beispiel hierfür ist in der Abbildung 4.5 zu sehen.

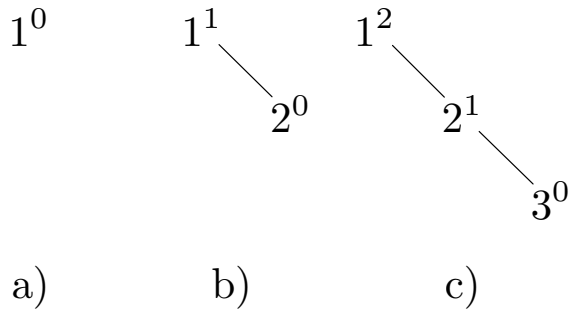

Abbildung 4.5: Ein AVL Baum wird aufgebaut. Die Höhe des jeweiligen Knotens im Baum wird an den jeweiligen Knoten rangeschrieben. Während bei a) der Wert 1 und bei b) 2 hinzugefügt wird, wird in c) der Wert angefügt. Wie man jedoch erkennt, ist in c) der Baum nicht mehr ausgeglichen.

4.6.2 Linksrotation

Nun muss der Baum aus Abbildung 4.5c) ausgeglichen werden, da die Höhe des linken Teilbaums von r sich so weit unterscheidet, dass die AVL Vorraussetzung verletzt wird. Der einfachste Griff ist hier, dass der Wurzelknoten nicht mehr 1 sondern 2 ist. Diese

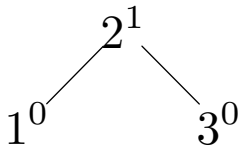

Abbildung 4.6: Der Baum aus Abbildung 4.6 wurde ausgeglichen und erfüllt nun die Spezifikationen für einen AVL-Baum

Modifizierung nennt sich *Linksrotation* (meist Abgekürzt als *l-Rotation*).

Die linksrotationen lassen sich lokal ausführen. Somit muss nicht ständig ein neuer Baum aufgebaut werden, sondern die Knoten nur „umgehangen“ werden. Dies passiert durch das korrigieren von konstant vielen Zeigern. Dadurch wird die Eigenschaft des Binären Suchbaums erfüllt.

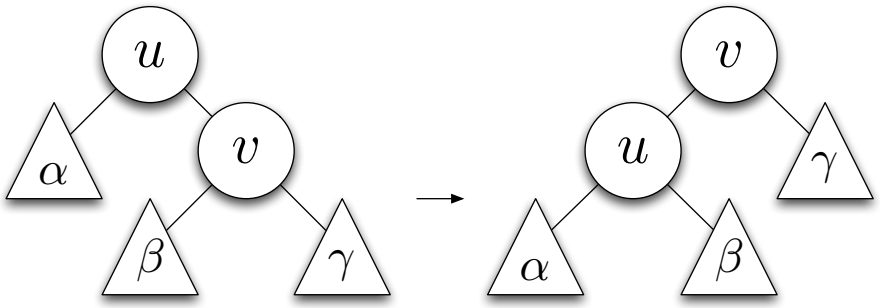

Abbildung 4.7: Eine beispielhafte Linksrotation im AVL-Baum.

4.6.3 Rechtsrotation

Analog zur *Linksrotation* gibt es noch die *Rechtsrotation*. Sie wird abgekürzt als *r-Rotation*. Dieser Ansatz kann in Abbildung 4.8 betrachtet werden.

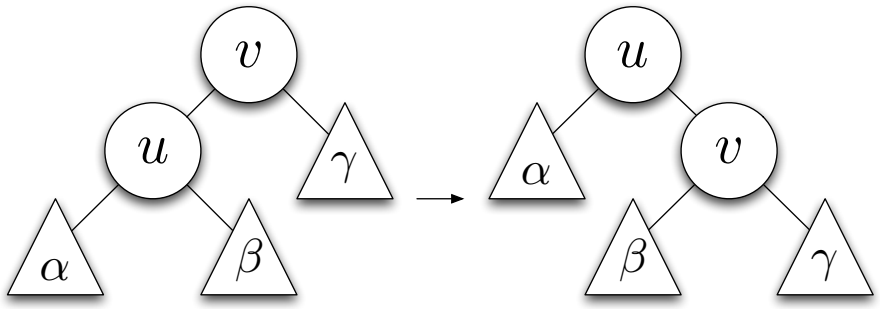

Abbildung 4.8: Eine beispielhafte Rechtsrotation im AVL-Baum.

Wenn bei v gilt $h_l - h_r \leq 0$, so ist der, aus einer *l-Rotation* resultierende Baum, ausgeglichen bei u und v (siehe Abbildung 4.9).

4.6.4 Mehrfache Rotationen

Diese Rotationen können jedoch zu problematischen Situationen führen (Abbildung 4.10): Hierbei verschieben wir das Problem nur von einem Teilbaum in den nächsten. Es reicht also nun nicht mehr aus, nur eine Rotation durchzuführen. Stattdessen müssen wir nun doppelt rotieren (Abbildung 4.11). Hierbei reicht es jedoch nicht aus, den Teilbaum aus Abbildung 4.10 zu betrachten, sondern wir müssen zumindest einen Teilbaum etwas weiter aufschlüsseln.

Sei nun v ein unausgeglichener Knoten (mit $|h_l - h_r| = 2$), und sind alle Nachbarn ausgeglichen, so gibt es eine (l, r, lr, rl)-Rotation bei v, die den Knoten ausgleicht. Dann sind v und alle seine Nachbarn ausgeglichen.

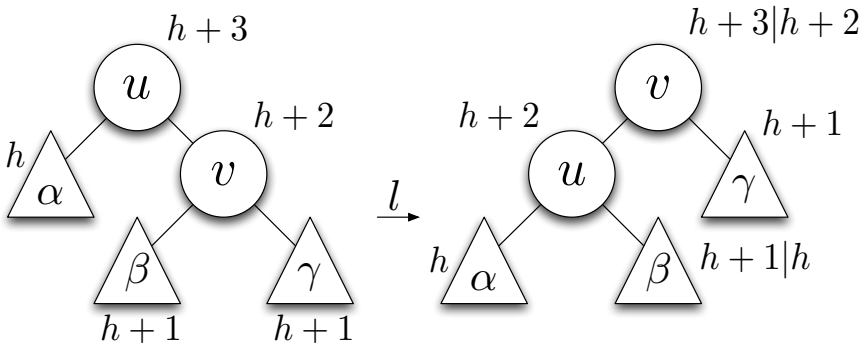

Abbildung 4.9: Der ausgeglichene Baum nach der Rotation.

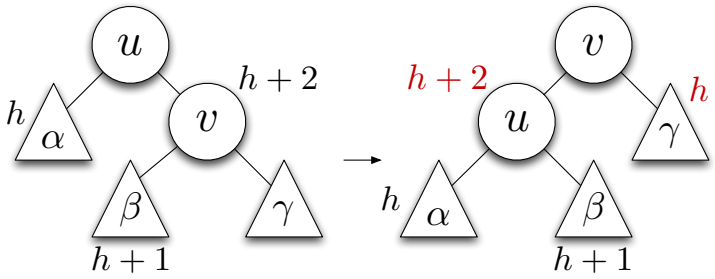

Abbildung 4.10: Die einfache Linksrotation im AVL löst unser Problem hier nicht.

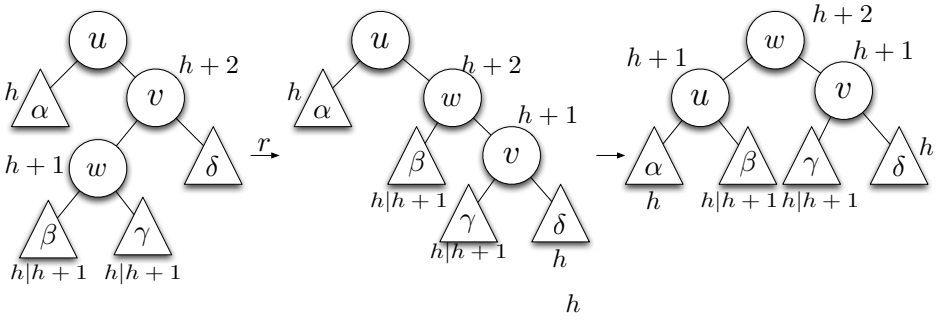

Abbildung 4.11: Bei diesem Baum muss mehrfach rotiert werden. Hierzu führen wir eine *rl*-Rotation durch.

Somit ergibt sich die folgende Beobachtung beim *AVL-Baum*: Das einfügen und löschen wird zunächst wie bei den Binären Suchbäumen durchgeführt. Danach muss dem Pfad bis zur Wurzel gefolgt werden, um Knoten zu finden die möglicherweise ausgeglichen werden müssen. Hierzu werden die notwendigen Rotationen dann ausgeführt.
Beispiel: Wir wollen nun mit einem leeren AVL Baum beginnen, und die Werte $\{10, 20, 30, 25, 35, 27\}$ nacheinander einfügen: [8, S. 431-436]

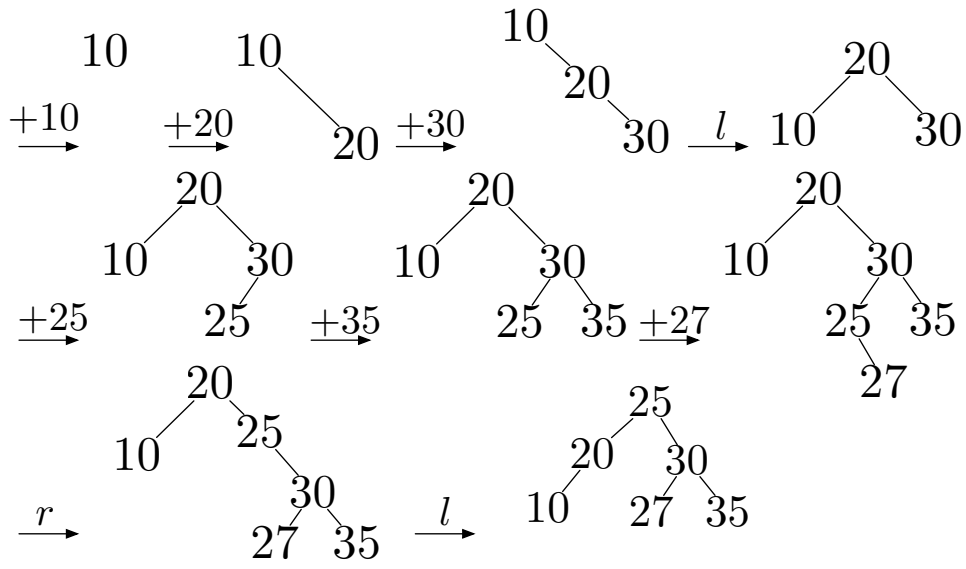

Abbildung 4.12: In einen leeren AVL-Baum werden nacheinander die Werte $\{10, 20, 30, 25, 35, 27\}$ eingefügt. Entsprechend muss der Baum somit, je nach Fall, rotiert werden.

4.6.5 Bewertung

Wir wollen nun die Betrachtung der AVL-Bäume abschließen, indem wir uns noch einmal die Vorteile und Nachteile dieser Bäume vor Augen führen.

Vorteile	Nachteile
Eine gute worst case Laufzeit von $\mathcal{O}(\log n)$	Es handelt sich um eine aufwendige Implementierung, wodurch sich leicht Fehler einschleichen können
Kompakte Darstellung des Baumes innerhalb der Klassen und des Speichers. Somit ist der *AVL-Baum* sehr platzsparend	Das Rebalancing findet sehr oft statt, weshalb bei der Implementation keine Rotation (links oder rechts) ausgelassen werden kann. Dies bedeutet wiederum viel Arbeit.

Abschließend sei noch gesagt, dass ein *AVL-Baum* in einen *Rot-Schwarz-Baum* gewandelt werden kann, indem bei ihm die Knoten entsprechend der Eigenschaften des Rot-Schwarz-Baumes gefärbt werden. Die Färbung eines Rot-Schwarz-Baumes ist letztlich jedoch nicht eindeutig, d.h. dass man bei einen Baum unter Umständen mehrere Möglichkeiten zur Färbung der Knoten hat.

4.7 Exkurs: Rot-Schwarz-Bäume

Wir wollen nun die Struktur des binären Suchbaumes weiter lockern. Das Ziel ist, dass wir die Zahl der nötigen Rotationen verringern. Hierfür fürben wir einige Knoten ein. Wir verwenden die Farben *rot* und *schwarz*, woher letztlich auch der Name *Rot-Schwarz-Baum* stammt. Für die Einfärbung der Knoten gelten die folgenden Regeln:[8, S. 463]

Wurzeleigenschaft Der Wurzelknoten ist schwarz.

Blatteigenschaft Jeder Blattknoten ist schwarz.

Eigenschaft innerer Knoten The Kinderknoten eines rot gefärbten Knotens sind schwarz.

Tiefeneigenschaft Alle Blätterknoten haben die gleiche schwarz-Tiefe. Das ist die Anzahl aller schwarzen Vorgänger im Baum, minus eins (da ein Knoten bereits Vorgänger von sich selbst ist).

Ein Beispiel für einen Rot-Schwarz-Baum kann man in Abbildung 4.13 sehen.

4.7.1 Laufzeitbetrachtungen

Ähnlich wie der *AVL-Baum*, hat der Rot-Schwarz-Baum eine Höhe von $\mathcal{O}(\log n)$.[8, S. 465]

4.7.2 Zusammenhang zwischen AVL-Bäumen und Rot-Schwarz-Bäumen

Jeder AVL-Baum ist ein Rot-Schwarz-Baum. Beim Einfügen und Löschen werden die Knoten rotiert und bei Bedarf neugefärbt. Der Vorteil im Rot-Schwarz-Baum gegenüber dem AVL-Baum ist, dass im Baum nur maximal 2-mal rotiert werden muss.[8, S. 463-480]

4.8 Exkurs: (a,b)-Bäume

Wir betrachten nun einen weiteren Suchbaum. Wir wollen die ursprüngliche Struktur des Binären Suchbaums nun noch weiter lockern. Der (a, b)-Baum ist ein *mehrweg Suchbaum*. Im Gegensatz zum binären Suchbaum können beim mehrweg Suchbaum mehr als zwei Pfade abzeigen, der *Grad der Knoten* kann also variieren. Der (a, b)-Baum führt die Variablen a und b ein.

$$a, b \in \mathbb{N} \quad \rightsquigarrow \quad b \geq 2 \cdot a - 1; \quad a \geq 2 \tag{4.14}$$

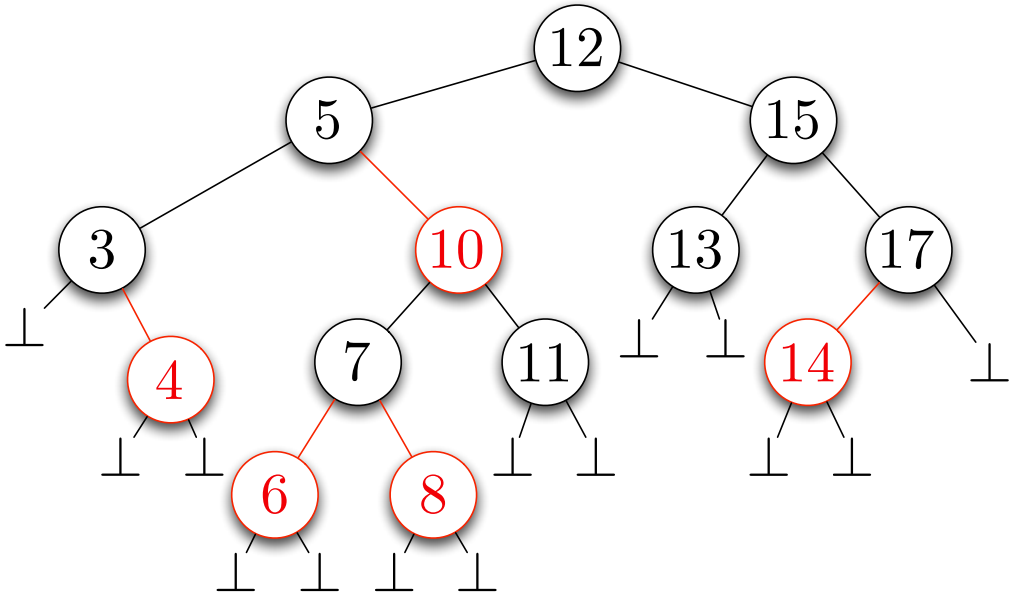

Abbildung 4.13: Ein Rot-Schwarz-Baum. Jeder Blattknoten hat 4 schwarze Elternknoten (inkl. sich selbst), wodurch die schwarz-Tiefe 3 ist. Wir haben die Kanten eines gefärbten Knotens ebenfalls gefärbt, dies dient ausschließlich der besseren Übersicht.

Im (a, b)-Baum wird der Grad der Knaten durch die Variablen a und b begrenzt. So ist definiert, dass für den Grad des Wurzelknotens r gilt

$$r \geq 2 \tag{4.15}$$

Der Grad eines inneren Knotens ist definiert als $\geq a$, während der Grad eines graden Knotens $\leq b$ ist. Letztlich speichert jeder Knoten zwischen $a-1$ und $b-1$ viele Einträge. Eine Ausnahme ist r, da dieser ≥ 1 Einträge speichert.

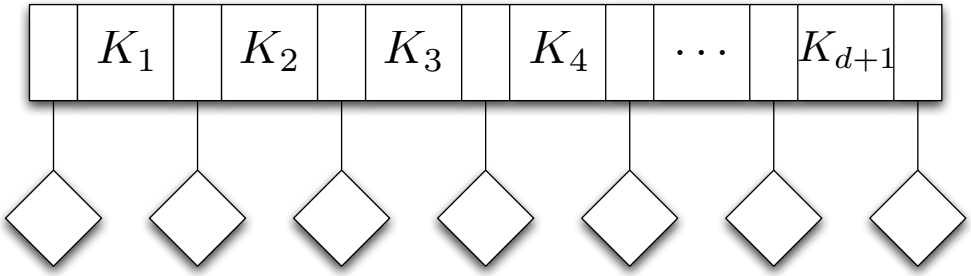

Abbildung 4.14: Ein Knoten eines Mehrwegbaumes.

Wie man in Abbildung 4.14, erkennen kann, führt von jedem eingetragenem Wert ein Wert ab. Somit lassen sich Suchen so durchführen wie man sie vom binären Suchbaum kennt. Die Tiefe (also die Anzahl der Knoten zur Wurzel) ist nun für jedes Blatt gleich. Die Höhe für einen (a, b)-Baum liegt zwischen $\mathcal{O}(\log n)$ und $\mathcal{O}(\log_a n)$.

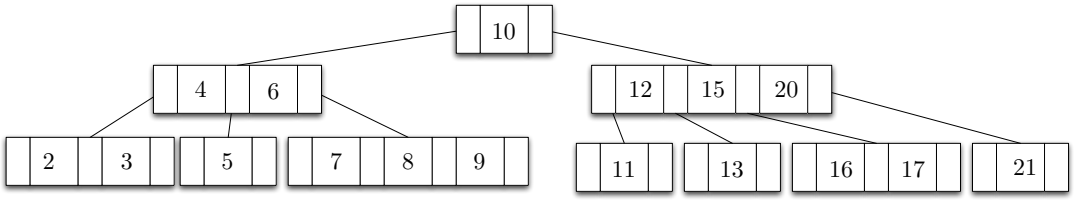

Abbildung 4.15: Beispiel für einen (2,4) Baum.

Wie man sich leicht vorstellen kann, wird ein Baum, welcher n Elemente speichert, bei geringer Höhe breiter und umgekehrt.

4.8.1 Suche im (a,b)-Baum

Die Suche im binären Suchbaum (a, b)-Baum wird so durchgeführt wie im binären Suchbaum. Jedoch kann es sein, dass wir mehrere Schlüssel pro Knoten vergleichen müssen. Somit haben wir eine Laufzeit von:

$$\mathcal{O}(b \log_a n) \tag{4.16}$$

4.8.2 Einfügen von Werten

Wir wollen in den Baum aus Abbildung 4.15 den Wert 18 einfügen. Hierzu laufen wir solange den Baum entlang, bis wir am Blattknoten angekommen sind. Hier fügen wir den Knoten ein und überprüfen die Größe des Knotens. Da hier < 4 Werte gespeichert wurden, sind wir fertig.

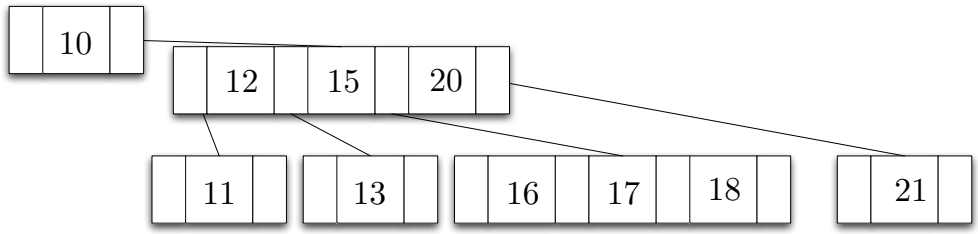

Abbildung 4.16: In den (a, b)-Baum aus Abbildung 4.15 wurde der Wert 18 eingefügt.

Nun soll in den Baum aus Abbildung 4.16 der Wert 19 eingefügt werden. Zunächst gehen wir vor, wie es eben gezeigt wurde, jedoch ist der neue Knoten nun zu voll (es kommt zum Überlauf). Daher muss er gespalten werden um einen Wert für den Vaterknoten zu haben der dort eingefügt werden kann. Bei Bedarf wird dieser erneut gespalten. Dieser Vorgang kann in Abbildung 4.17 betrachtet werden.

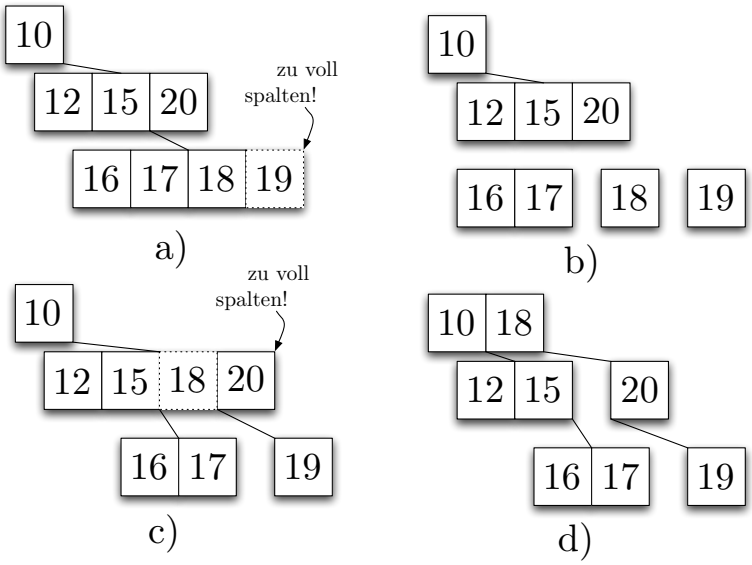

Abbildung 4.17: In den Baum aus Abbildung 4.16 wird der Wert 19 eingefügt. Der Blattknoten enthält in a) jedoch zu viele Elemente, weshalb er gespalten wird, und der Knoten 18 in den Vaterknoten verlegt wird. Auch dieser ist in c) zu voll weshalb 18 in Abbildung d) nach r verschoben wird.

4.8.3 Löschen von Werten

Der Algorithmus für das Löschen von Elementen im (a, b)-Baum ist jedoch komplexer als im binären Suchbaum,da es zu Unterläufen in den Knoten kommen kann. Zunächst wollen wir betrachten, wie r gelöscht wird. Wir übernehmen hierzu das Beispiel aus Abbildung 4.15. Zunächst wird die Position der 10 durch 9 (also den *Vorgänger*, das rechteste Blatt des Linkenteilbaums) übernommen. Kommt es beim Blatt zu keinem Unterlauf, kann 9 einfach entfernt werden.
Als nächstes wollen wir sehen, wie man aus einem Blatt löscht. Kommt es nach dem löschen nicht zu einem Unterlauf, es sind also $\geq a$ Elemente noch darin gespeichert, so wird der Wert einfach gelöscht.
Sind in einem Knoten jedoch nur noch $a - 1$ Schlüssel gespeichert, so kommt es zum *Unterlauf*.

4.8.4 Unterlaufbehandlung

Hierfür gibt es verschiedene Lösungsmöglichkeiten. Ein Weg ist es, sich einen Schlüssel von den Geschwisterknoten zu leihen. Hierzu wird vom Elternknoten ein Schlüssel übernommen, während der *Geschwisterknoten* einen Wert an den Elternknoten abgibt (siehe Abbildung 4.18).

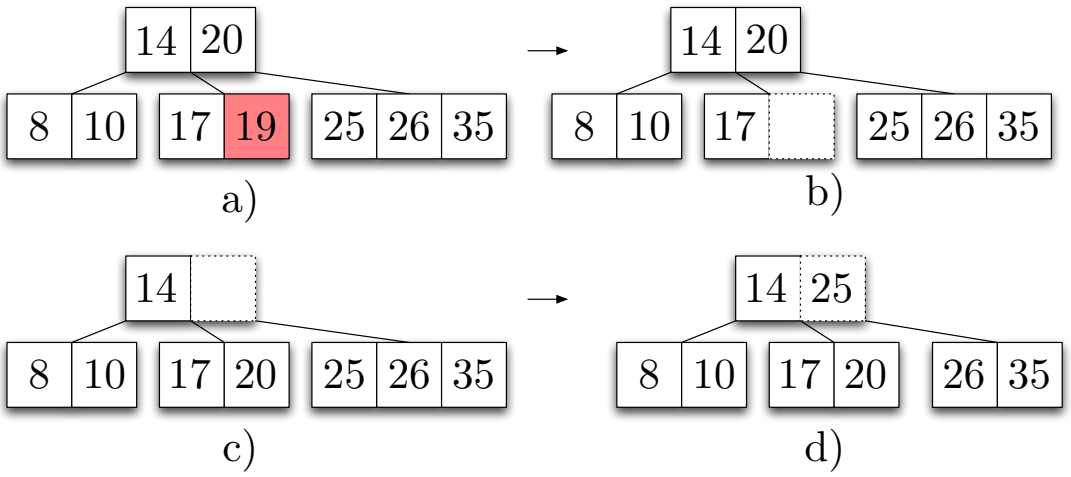

Abbildung 4.18: Löschen eines Knotens aus einem $(2, 4)$-Baum: a) Wird 19 gelöscht, so kommt es beim Blattknoten zu einem Unterlauf. b) Für einen Ausgleich, wird der Nachfolger aus dem Elternknoten im Blatt eingesetzt. c) Hierdurch kommt es erneut zu einem Unterlauf, der Nachfolger im Kinderknoten passt hier rein. d) Er wird in den Elternknoten verschoben. Danach ist der Baum erneut ausgeglichen worden. Somit hat das Blatt in der Mitte sich einen Schlüssel aus dem Geschwisterknoten geliehen.

Wenn dies jedoch nicht möglich ist, muss der Knoten mit einem Geschwisterknoten

verschmolzen werden. Dazu wird der Schlüssel, der im Elternknoten zwischen Geschwisterknoten liegt ebenfalls heran gezogen. Danach kann der Knoten bei Bedarf wieder gespalten werden (siehe Abbildung 4.19). Wiederhole den Prozess notfall mit dem Elternknoten. Die erwartete Laufzeit für das Löschen und ausgleichen lautet daher $\mathcal{O}(b \log_a n)$.[8, S. 461][19, S. 374ff.][4, S. 499ff.]

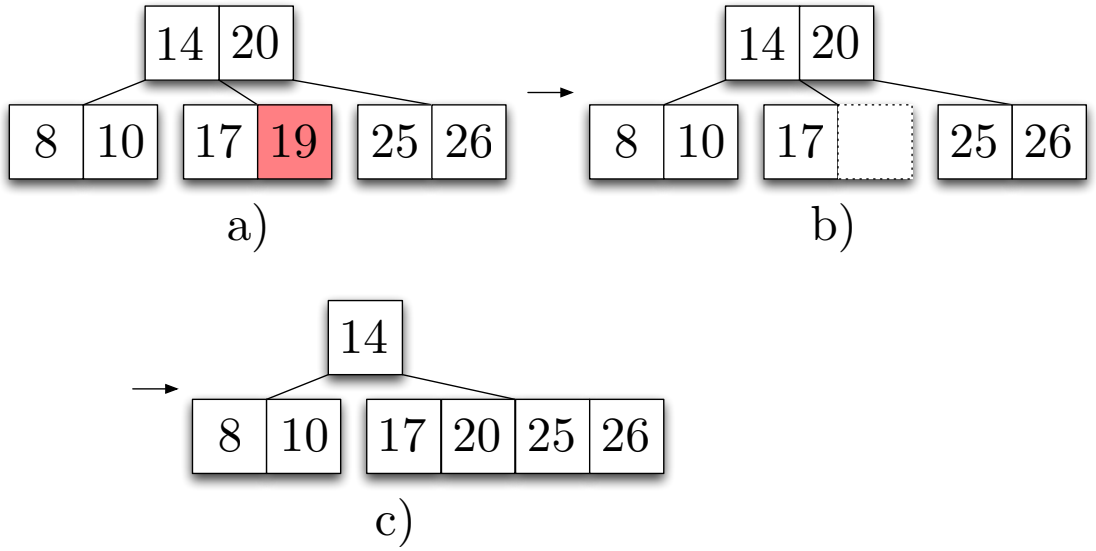

Abbildung 4.19: Löschen eines Knotens aus einem (a, b)-Baum. a) Erneut soll der Wert 19 gelöscht werden. In diesem Knoten kommt es zum Unterlauf. b) und c) Die Blattknoten werden nun verschmolzen, der Wert 20 aus der Wurzel hierfür mitverwendet, da die Suchbaumeigenschaft ansonsten verletzt wird.

4.8.5 Anwendung: Andere Bäume darstellen

Wie können wir nun die (a, b)-Bäume anders Repräsentieren lassen. Für die Darstellung von *AVL-Bäumen* eignen sich (2,3)-Bäume. Ein *Rot-Schwarz-Baum* hingegen kann mit Hilfe eines (2,4)-Baumes dargestellt werden. Dabei entscheidet die Anzahl der gefärbten Kinderknoten übe die Anzahl der abgehenden Pfade im Baum (siehe Abbildung 4.20).[8, S. 464]

4.8.6 Anwendung: Externe Datenstrukturen

Man findet große (a, b)-Bäume häufig in anderen Gebieten der Informatik wieder. Durch die gute Laufzeit beim suchen und modifizieren des Baumes eignen sich große (a, b)-Bäume für Dateisysteme oder Datenbanken. Hierbei sind Zugriffe auf die Festplatte die letztlich zeitintensivste Operation, wobei der Baum eine möglichst geringe Höhe haben soll, sodass nur wenige Vergleichsoperationen nötig sind. Ziel ist es, dass soviele Knoten

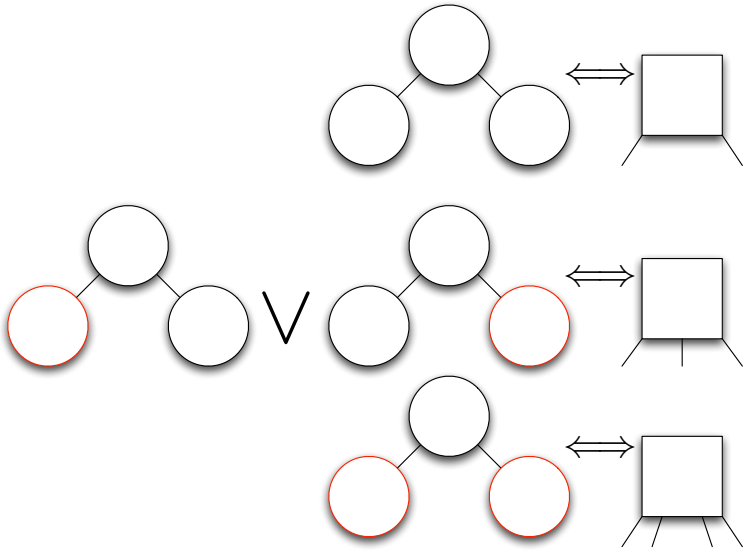

Abbildung 4.20: Das Verhältnis zwischen Rot-Schwarz und (a, b)-Bäumen wird gezeigt.

möglich in einen Sektor geschrieben werden, damit die Leseoperationen optimal ausgenutzt werden. Bekannte Beispiele für Datenbanken, die auf (a, b)-Bäumen basieren sind:

- Postgre-SQL
- MySQL

Der Einsatz in Dateisystemen wird vorallem im *BTRFS*[20] und *B-Tree-FS* verwendet. Letzteres basiert auf dem *B-Baum*. Dies ist ein spezialisierter (a, b)-Baum. Genauergesagt handelt es sich hier um einen $(a, 2a)$-Baum.

5 Zeichenketten

Wir wollen die Betrachtung von (abstrakten) Datenstrukturen nun beenden und uns nun Strings zuwenden. Hierzu wollen wir zunächst einige Definitionen aufstellen, die zumeist bereits aus der Vorlesung *Grundlagen der theoretischen Informatik* bekannt sein sollten.[13, S. 28]

Alphabet Sei Σ eine endliche Menge von Zeichen, so ist dies unser Alphabet. Beispiele hierfür sind:

- $\Sigma = \{0, 1\}$ ist das binär-Alphabet.
- $\Sigma = \{a, b, \ldots, z\}$ sei die Menge aller kleinen Buchstaben.
- Die Menge alle ASCII Zeichen: Dies ist das ASCII Alphabet.
- $\Sigma = \{G, T, C, A\}$ ist das Alphabet des Genetischen Codes.

Wörter Ein String, bzw. ein Wort ist eine endliche Folge von Symbolen die aus einem Σ gewählt werden. So sind etwa 010010110, 011, 10010111101 Wörter, die aus dem binär Alphabet $\Sigma = \{0, 1\}$ gebildet werden können. Jedoch kann 0101102 nicht daraus gebildet werden, da die 2 nicht im binär Alphabet vorkommt.

Hierzu gibt es nun einige Problemgebiete, die wir näher betrachten wollen.

- *Ähnlichkeit*: Wie ähnlich sind zwei Strings?
- *Suchen*: Finde alle vorkommen einer Zeichenkette in einer anderen Zeichenkette.
- *Effiziente Speicherung*: Wie kann eine Zeichenkette geschickt komprimiert werden?
- *Effiziente Wörterbücher*: Hierzu werden wir unter anderem *Tries* betrachten.

5.1 Effiziente Speicherung

Zur Vereinfachung wollen wir ein reduziertes Alphabet betrachten. Wir definieren für die folgende Betrachtung:

$$\Sigma = \{a, b, c, d\} \tag{5.1}$$

Weiterhin sehen wir den String s an:

$$s = abaaaccdaab \tag{5.2}$$

Ein naiver Ansatz ist für uns, dass wir jedem Zeichen $\lceil \log \Sigma \rceil$ viele Zeichen zuweisen. Für unser Beispiel wählen wir hierzu als Codierung:

$$a \rightarrow 00 \tag{5.3}$$
$$b \rightarrow 01 \tag{5.4}$$
$$c \rightarrow 10 \tag{5.5}$$
$$d \rightarrow 11 \tag{5.6}$$

Hierzu werden offensichtlich $\mathcal{O}(|s| \log |\Sigma|)$. Das ist unter Umständen jedoch verschwenderisch, da bestimmte Buchstaben häufiger vorkommen können als andere. Dies ist schnell nachvollziehbar, wenn wir uns überlegen, wie häufig im deutchen Wortschatz der Buchstabe e verwendet wird, und wie häufig etwa x oder y. Somit müsste e einen viel kürzeren Code bekommen als x oder y. Sollten einer dieser Buchstaben dennoch verwendung finden, so haben wir die Zeichen die nun mehr benötigt werden, bei den Verwendungen des e eingespart (vgl. Amortisierte Analyse).

Hierzu benötigen wir letztlich eine sogenannte *Codefunktion* C. Sie übersetzt ein Element eines Alphabets in eine Binärfolge.

$$C: \quad \Sigma \rightarrow \{0,1\}^* \tag{5.7}$$

Wir bezeichnen das n-te Zeichen eines Wortes mit G_n. Somit ist die Codierung eines Wortes s, mit der Länge n definiert als:

$$s = \varsigma_1\varsigma_2\varsigma_3 \ldots \varsigma_n \tag{5.8}$$
$$C(s) = C(\varsigma_1)C(\varsigma_2)C(\varsigma_3)\ldots C(\varsigma_n) \tag{5.9}$$

Da wir $C(s)$ auch zurückrechnen wollen, muss C dabei auch eindeutig dekodierbar sein, zusätzlich zur Vorraussetzung das ein möglichst kurzer Bitstring berechnet werden soll. Betrachten wir folgende Codes:

5.1.1 Eindeutigkeit

$$\Sigma = \{a,b,c\} \tag{5.10}$$
$$a \rightarrow 01 \tag{5.11}$$
$$b \rightarrow 010 \tag{5.12}$$
$$c \rightarrow 10 \tag{5.13}$$

Es stellt sich nun die Frage, ob diese Codierung eindeutig ist. Wir betrachten hierzu die Codierung der Strings ab und bc:

$$\begin{array}{ll} ab & \searrow \\ & \quad 01010 \\ bc & \nearrow \end{array} \tag{5.14}$$

Das Problem in diesem Beispiel ist, dass der Code $C(a)$ im Präfix von $C(b)$ steht. Dies ist ein Grundproblem bei unserer Betrachtung. Somit wäre beim dekodieren von 01010 nicht klar, ob es sich um $C(ab)$ oder $C(bc)$ handelt und somit um jeden Preis zu verhindern.

Definition : Wenn ein Code so geschaffen ist, dass kein Codewort Präfix eines anderen Codeworts ist, so ist er *Präfixfrei*. Diese lassen sich eindeutig (de-)kodieren. Präfixfreie Codes lassen sich als Bäume darstellen (siehe Abbildung 5.1).

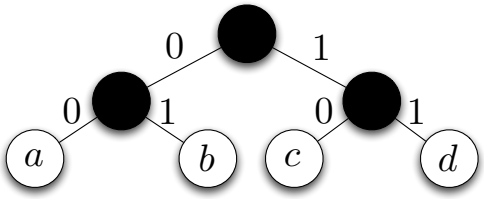

Abbildung 5.1: Beispiel für einen Baum, der für einen Präfixfreien Code für das Alphabet $\Sigma = \{a, b, c, d\}$ verwendet werden kann. Die Codes für die Zeichen in diesem Baum lauten somit: $a = 00$, $b = 01$, $c = 10$, $d = 11$.

5.2 Huffman-Code

Die Frage ist nun also, wie man einen Code erstellen kann, der bei häufig auftauchenden Zeichen einen kürzeren Code erstellt. Dazu soll dieser natürlich eindeutig, also präfixfrei, sein. Wir könnten die erwartete Häufigkeit für Strings nehmen und daraus einen Code generieren und ständig verwenden. So würden wir etwa die durchschnittliche Verteilung der Buchstaben in der deutschen Sprache berechnen und daraus einen Code erstellen. Müssten wir jedoch einen englischen Text oder Quelltext codieren, so würde diese Verteilung nicht mehr passen. Daher zählen wir für jeden Text (bzw. jede Datei) erneut die Häufigkeiten der Zeichen. Dies hat gleichzeitig den Vorteil, das wir keine Codierung für Zeichen vorbereiten müssen, die letztlich im Text nicht mehr vorkommen. Die Gesamtlänge des Codes ist somit

$$\sum_{G \in \Sigma} |C(\varsigma)| h_\varsigma \tag{5.15}$$

5.2.1 Allgemeines Entwurfsprinzip für den Algorithmus

Das Ziel ist nun, dass wir einen Binärbaum aufbauen. Dieser liefert dann einen Code ähnlich wie der Baum aus Abbildung 5.1. Hierzu zählen wir die Häufigkeit der verwendeten Zeichen im gegebenen Text. Danach können wir die Zeichen aus dem Zeichenalphabet Σ mit Hilfe des Baums codieren. Hierzu werden die Zeichen in Baumknoten gespeichert. Hinzu wird die entsprechende Häufigkeit des Zeichens gespeichert.

Als nächster Schritt soll der Baum zusammen gestellt werden. Hierzu werden die Elemente mit den kleinsten Häufigkeiten mit einem weiteren Knoten verbunden und dann dann erneut im Array gespeichert. Die Häufigkeit für diesen Knoten ergibt sich aus der Summe der häufigkeiten verbunden Knoten(siehe Abbildung 5.2). Dieser Schritt wird

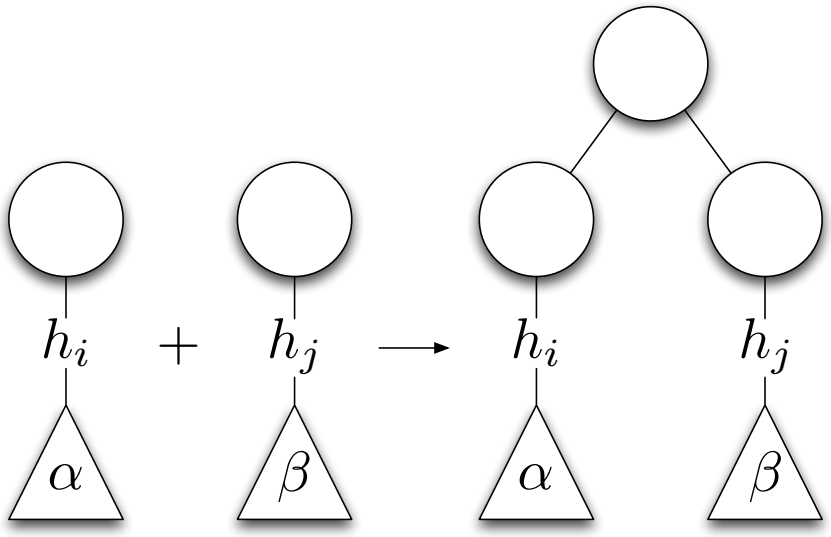

Abbildung 5.2: caption

solange wiederholt bis es nur noch ein Element im Array gibt, der Baum selbst (für ein Beispiel, siehe Abbildungen 5.3 und 5.4). Dieser Algorithmus und Baum wurde nach seinem Erfinder benannt: *David A. Huffmann*, weshalb es sich um den Huffmann-Baum und Huffmann-Code handelt. Das Prinzip hinter diesem Algorithmus ist der *Greedy-Algorithmus*. Das Konzept dahinter betrachten wir im Kapitel 5.3.

Satz: Huffmann Codes sind optimale Präfixcodes, d.h. sie minimieren einen Text so gut es geht, ohne dass Informationen verloren gehen.

$$\sum_{\varsigma\in\Sigma}^{|} C(\varsigma)|h_\varsigma \tag{5.16}$$

Lemma: Sei $\varsigma \in \Sigma$, sodass $h_\varsigma \leq h_\varsigma$, für alle $\varsigma \in \Sigma\backslash\{\varsigma_i\}$. Dann existiert ein optimaler Präfixcode, sodass im zugehörigen Baum ς_i und ς_j als Geschwister hängen.

Beweis: Sei C^* ein optimaler Präfixcode.

Schritt 1 Es existiert ein optimaler Präfixcode C^{**} , sodass das Blatt ς maximale Tiefe hat.

1. Das Gilt schon in C^* ✓.
2. Tausche ς_i mit einem Blatt maximaler Tiefe. Dadurch wird die Codelänge für ς_i größer, die Länge für das alte Blatt kürzer. Somit gleichen sich die

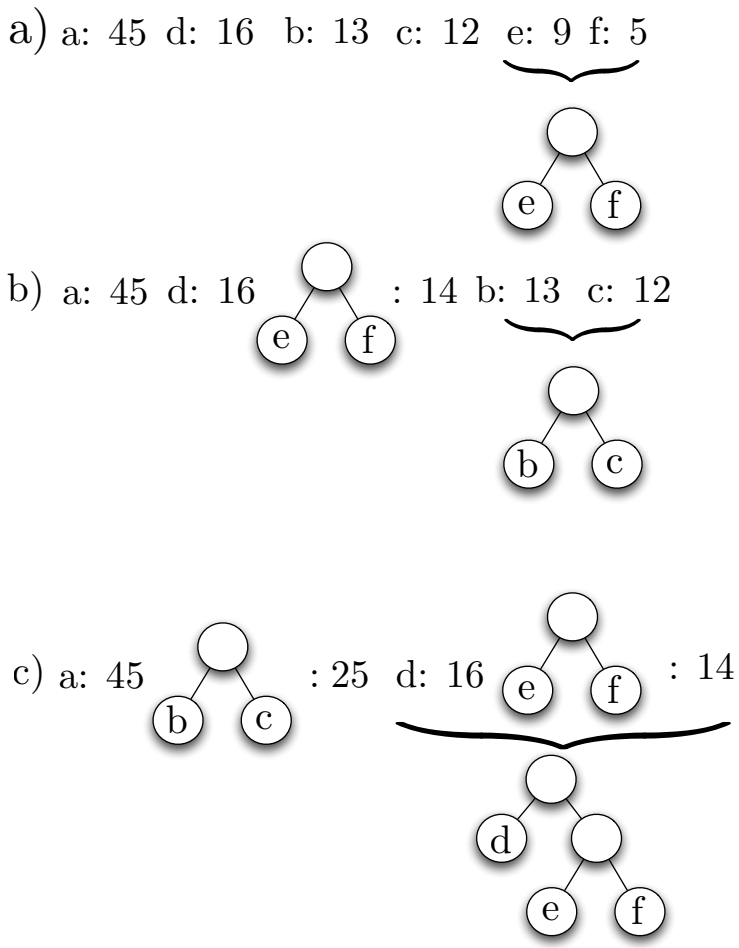

Abbildung 5.3: Erstellen eines Codebaumes. Zu Grunde liegen die Häufigkeiten: $a : 45$, $b : 13$, $c : 12$, $d : 16$, $e : 9$, $f : 5$

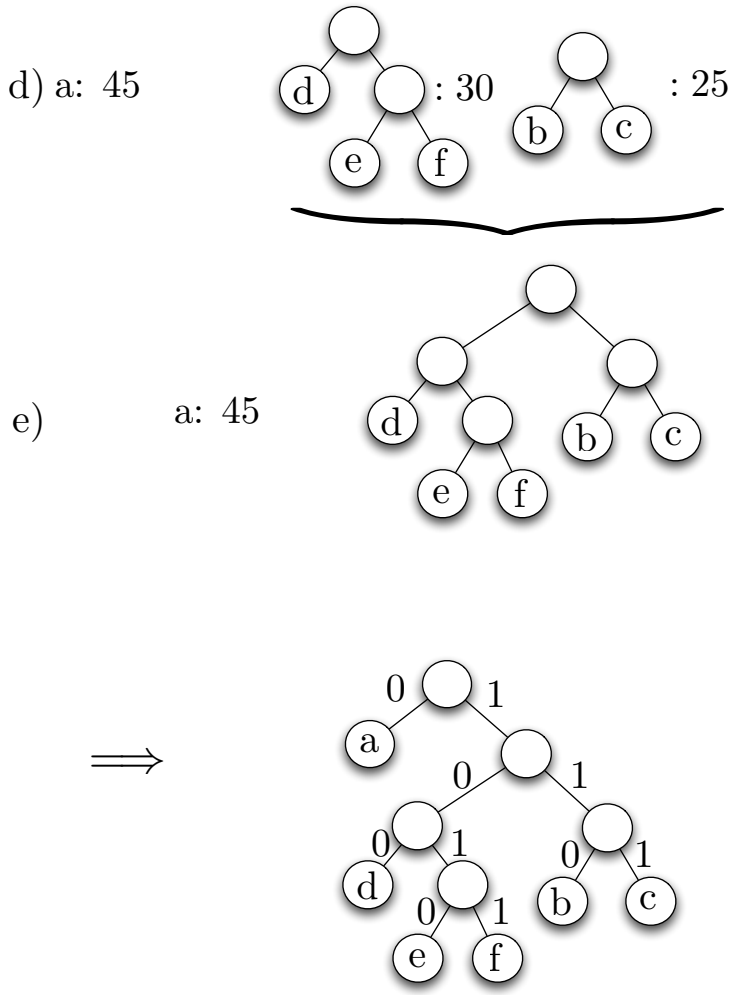

Abbildung 5.4: Fortsetzung von Abbildung 5.3. Somit ergeben sich die folgenden Codes: a: 0, b: 110, c: 111, d: 100, e: 1010, f: 1011

Veränderungen aus, der Code wird nicht länger, unter Umständen nur kürzer, im größten Teil aller Fälle änder sich die Gesamtlänge überhaupt nicht.

Schritt 2 Es existiert ein optimaler Präfixcode C^{***}, sodass ς_i und ς_j Geschwister sind.

- Gilt schon für C^{**}
- Sonst hat ς_i einen Geschwisterknoten, da C^{**} optimal ist. Daher müsste dieser Knoten mit ς_j getauscht werden. Dadurch wird die Gesamtlänge nicht größer.

□

Beweis (Satz): Wir führen hierzu einen Induktionsbeweis nach k über $|\Sigma|$.

Induktionsanfang $K = 2$ ✓

Induktionsschritt Wir nehmen an, dass der Huffmann Code H_c optimal ist für sämtliche Alphabete und Häufigkeiten der Größe $k-1$. Wir zeigen hierzu für alle Alphabete und Häufigkeiten der Größe k. Dazu wird ein *indirekter Beweis* geführt:

Nimm an, es existiert ein $\Sigma = \{\varsigma, \ldots, \varsigma_2\}$ und Häufigkeit $h_{\varsigma_1}, \ldots, h_{\varsigma_k}$ und dder hinzugehörigen Häufigkeit $h_{\varsigma_1}, \ldots, h_{\varsigma_k}$, sodass der Huffmanncode *nicht* optimal ist. Seien ς_i und ς_j die beiden Symbole, die der Huffmann Algorithmus zuerst vereint. Nach dem Lemma ergibt dies einen optimalen Code wo ς_i und ς_j Geschwister sind (siehe Hierzu Abbildung 5.5).

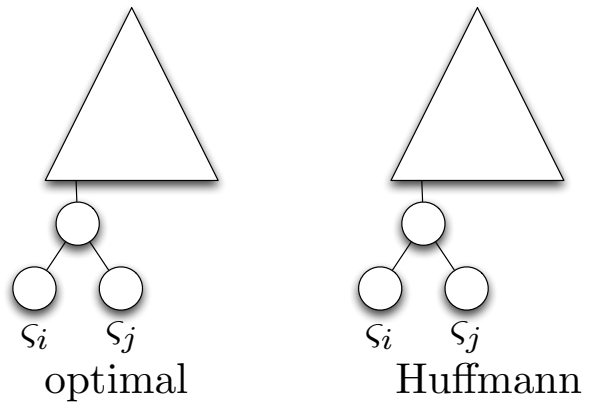

Abbildung 5.5: Der Huffmann Code und der optimale Code liefern den gleichen Baum, dies ist jeodch ein Widerspruch, da wir ursprünglich davon ausgingen, dass der optimale Code anders ist, als der, der sich aus dem Huffmann Algorithmus ergibt.

$$\sum_{\varsigma \in \Sigma} |h_c(\varsigma)| h_\varsigma \quad > \quad \sum_{\varsigma \in \Sigma} |\mathrm{Opt}(\varsigma)| h_\varsigma \tag{5.17}$$

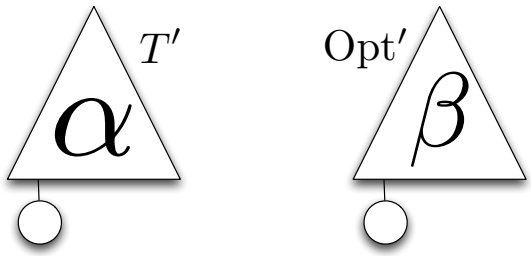

Abbildung 5.6: Die Bäume T' und Opt′ werden definiert.

Sei $\Sigma = \Sigma\{\varsigma_i, \varsigma_j\} \cup \{T\}$, $|\Sigma| = k - 1$ und $h_j = h_{\varsigma_j} = h_{\varsigma_j} + h_{\varsigma_i}$. Nun werden zwei Bäume T' und O' definiert (siehe Abbildung 5.6). T' ist der Huffmann Baum für Σ'. Aber C^* impliziert

$$\sum_{\varsigma\in\Sigma} |T(\varsigma)|h_\varsigma > \sum_{\varsigma\in\Sigma} |\text{Opt}'(\varsigma)|h_\varsigma \tag{5.18}$$

$$\sum_{\varsigma\in\Sigma} |T(\varsigma)|h_\varsigma = \sum_{\varsigma\in\Sigma} |\text{Opt}'(\varsigma)|h_{\varsigma_i} + h_{\varsigma_j} \tag{5.19}$$

$$\sum_{\varsigma\in\Sigma} |\text{Opt}(\varsigma)|h_\varsigma = \sum_{\varsigma\in\Sigma} |\text{Opt}'(\varsigma)|h_\varsigma + h_{\varsigma_i} + h_{\varsigma_j} \tag{5.20}$$

Damit wäre der Code echt besser als der Huffmann Code. Dies wiederspricht jedoch der Induktionsannahme und beendet somit den Beweis, so dass gezeigt ist, dass der Huffmann Code immer einen optimalen Code liefert.

□

[8, S. 566]

5.3 Greedy Algorithmen

Algorithmen, die nach dem *Greedy* (also gierigem) Algorithmus funktionieren, wählen in jedem Teilschritt den Weg, der das größtmögliche Vorankommen verspricht.

Dieses Vorgehen kann man häufig bei Kassierern beobachten. Diese greifen solange wie möglich die jeweils größten Münzen aus der Kasse. Bei einem Betrag von etwa 19,87 wird der Kassierer das Geld vorraussichtlich folgendermaßen herausnehmen:

1. 10 €(Schein) [Rest: 9,87 €]
2. 5 €(Schein) [Rest: 4,87 €]
3. 4 €(2× 2 €) [Rest: 0,87 €]

4. 0,5 €[Rest: 0,37 €]
5. 0,2 €[Rest: 0,17 €]
6. 0,1 €[Rest: 0,07 €]
7. 0,05 €[Rest: 0,02 €]
8. 0,02 €[Resst: 0,00 €]

[2, S. 157f.] In vielen Problemen liefern Greedy Algorithmen gute bzw. sehr gute Ergebnisse, bei anderen Problemen können diese jedoch eher ein schlechterer Ansatz sein.[4, S. 414-450][8, S. 567]

5.4 Datenkompression

Wie wir im Kapitel 5.2.1 gesehen haben, liefert der Huffmann-Algorithmus immer einen optimalen Code. Dazu erfüllt er die Anforderungen:

- *Präfixfrei*
- *Verlustfrei*
- Agiert auf *Zeichenebene*, codiert also jedes Zeichen einzeln.

Jedoch gibt es auch Algorithmen, die wiederum nicht jedes Zeichen einzeln codieren. Hierzu gehören:

Lauflängencodierung Da es sein kann, dass sich Zeichen hintereinander mehrfach wiederholen. Diese können dann etwa zusammengefasst werden:

$$s = aaabbbabaabaaaabbbaccacaccccacd \tag{5.21}$$
$$C(s) = 3a3b1a1b2a1b4a3b1a2c1a1c1a4c1a1c1d \tag{5.22}$$

Dies eignet sich vorallem für Bilder, wodurch neue Formate entstanden:

- PCX
- RLE

Wiederkehrende Muster Es kann sein, dass sich bestimmte Bitfolgen mehrfach in einer Datei bzw. einem Text finden. Diese könnten wiederum zu einem neuen Zeichen definiert werden. Diese Idee geht auf die Forscher *Jacob Ziv* und *Abraham Lempel* zurück. Verbesserungen gab es von *Terry Welch*. Dieser, ehemals patentierte Algorithmus, nennt sich daher *LZW-Algorithmus*. Anwendung findet er unter anderem in:

- compress
- PKZIP

- GIF

Ebenso können manche Anwendungen verlustbehaftete Informationen vertragen. Dies geschieht bei Bildern der Musik. Hier sind, im Gegensatz zum Text, nicht sämtliche Informationen unbedignt nötigt. Hier müssen nur relevante Versionen gespeichert werden (JPEG, MP3, MP4, FLV). Hier zukann unter anderem die *Fourier Transformation*, die *Diskrete Cosinus Transformation* oder die *Warclets Transformation* verwendet werden.

5.5 Ähnlichkeit von Texten

Wir wollen nun die Frage beantworten wie man feststellen kann, wie ähnlich sich zwei Zeichenketten sind. Beispielsweise kann das Programm '*diff*' zwei Dateien entgegennehmen, und Zeigen, welche Änderungen gemacht werden müssen um eine Datei in die andere umzuwandeln. Dies ist zumeist für *SCM* Systeme interessant, da hier den anderen Entwicklern die Änderungen schnell angezeigt werden.

5.5.1 Formalisierung

Wir definieren zwei Zeichenketten s und t wie folgt:

$$s = s_1 s_2 s_3 \dots s_k \tag{5.23}$$
$$t = t_1 t_2 t_3 \dots t_l \tag{5.24}$$

Als nächstes soll die Ähnlichkeit von zwei Zeichenketten definiert werden. Wir wollen zunächst zählen, was die Mindestzahl der benötigten Operationen ist, um s in t zu überführen. Dies nennt man den Editierabstand (auch bekannt als *Levenshtein-Metrik*).

Später wollen wir die längste, gemeinsame Teilfolge suchen. Wir definieren eine gemeinsame Teilfolge mit:

$$s_{i_1} \dots s_{i_2} \dots s_{i_3} \dots s_{i_j} \qquad i_1 \leq i_2 \leq i_j \tag{5.25}$$
$$t_{o_1} \dots t_{o_2} \dots t_{o_3} \dots t_{o_k} \qquad o_1 \leq o_2 \leq o_j \tag{5.26}$$

5.5.2 Elementare Operationen

Als weiteres müssen wir nun die Operationen definieren. Diese werden elementar gehalten, so dass die benötigten Schritte sich schnell zeigen. Dann kann die Zahl der Opertaionen gezählt werden, wodurch wir den gesuchten *Editierabstand* erhalten. Die Elementaroperationen lauten wie folgt und sollten selbsterklärend sein:

- Einfügen
- Löschen
- Vertauschen von Zeichen

Jedoch ist es leicht vorstellbar, dass das Vertauschen von Zeichen redundant ist, da es sich ebenso mit den anderen Operationen darstellen lässt. Jedoch lassen sich die Operationen nun besser zählen.

5.5.3 Fallbeispiel

Wir wollen uns nun die längste, gemeinsame Teilfolge in den Worten *NEUJAHRSTAG* und *EPIPHANIAS* ansehen. Wie man in Abbildung 5.7 sehen kann, ist lautet die längste, gemeinsame Teilfolge hier 3.

Abbildung 5.7: Die längste gemeinsame Teilfolge wird gezählt. Wie man sieht, lautet sie bei *NEUJAHRSTAG* und *EPIPHANIAS* 3. Die Teilfolgen wurden jeweils durch Farben gekennzeichnet.

5.5.4 Rekursiver Ansatz

Die nächste Frage soll nun sein, wie wir dieses Problem möglichst geschickt algorithmisch lösen. Ein Ansatz, welcher von links nach rechts verläuft würde dafür beispielsweise schwierig zu implementieren sein. Daher lesen wir die Zeichen mit einer Rekursion von hinten ein. Dazu betrachten wir zunächst die jeweils letzten Buchstaben der beiden Wörter. Wenn $s_k \neq t_l$ ist, so bestimmen wir dann die längste, gemeinsame Teillänge von s_1 bis s_{k-1} und t_1 bis t_l sowie zwischen s_1 bis s_k und t_1 bis t_{l-1}. Zum Schluss wird der jeweils größte Wert gewählt.
Sollte jedoch $s_k = t_l$, so wird als nächster Aufruf $s_1 \dots s_{k-1}$ und $t_1 \dots t_{l-1}$ berechnet. Am Ende wird der jeweils größte Wert + 1 zurückgegeben. Daraus würde sich etwa folgender Haskell Code ergeben:

Listing 5.1: recursive.hs

```
1 lgt s t
2         | s == [] || t == [] = 0
3         | last(s) == last(t) = lgt' init(s) init(t) 1
4         | otherwise = max (lgt' init(init(s)) init(t) 0) (lgt'
            init(s) init(init(t)) 0)
5
6 lgt' s t x
7         | s == [] || t == [] = x
8         | last(s) == last(t) = lgt' init(s) init(t) x+1
9         | otherwise = max (init(init(s)) init(t) x) (init(s)
            init(init(t)) x)
```

5.5.5 Dynamischer Ansatz

Da wir jedoch im Kapitel 1.7.1 gesehen haben, haben rekursive Lösungen meist eine schlechte Laufzeit haben, da hier häufige Ergebnisse mehrfach berechnet werden. Wir

entwickeln daher nun einen Ansatz mit *dynamischer Programmierung* und speichern dazu unsere Zwischenergebnisse. Es gibt letztlich zwei Lösungmöglichkeiten für uns:

- Wir implementieren weitere eine Rekursion, speichern jedoch Zwishcenergebnisse in einer Tabelle und sehen dort bei Bedarf nach.
- Wir füllen direkt die Tabelle von unten auf und verwenden dazu for-Schleifen.

Da die erste Lösung sich schnell aus dem rekursiven Code ergibt, betrachten wir den direkten Ansatz.

Listing 5.2: LargestCommonSubstring.java

```
1  public class LargestCommonSubstring {
2          private String s1, s2, gcs;
3          private long gcsLength;
4
5          public LargestCommonSubstring(String s1, String s2) {
6                  this.s1 = s1;
7                  this.s2 = s2;
8
9          }
10
11         private void computeGreatestCommonSubstring() {
12                 int s2l = s2.length()
13                 ArrayList<long> table = new ArrayList<long>();
14                 for (int i = 0; i < this.s1.length(); i++) {
15                         table.add(new ArrayList<long>(s2l));
16                 }
17                 for (int i = 0; i < this.s1.length(); i++) {
18                         for (int j = 0; i < s2l; j++) {
19                                 if(this.s1.charAt(i) == this.s2
                                       .charAt(j)) {
20                                         table.get(i).set(j,
                                           table.get(i).get(j)
                                           +1);
21                                 } else {
22                                         table.get(i).set(j,
                                           this.maximum(table.
                                           get(i-1).get(j),
                                           table.get(i).get(j
                                           -1)));
23                                 }
24                         }
25                 }
26         }
```

```
27
28          private long maximum(long l1, long l2) {
29                  if (l1 <= l2) {
30                          return l2;
31                  } else {
32                          return l1;
33                  }
34          }
35  }
```

Wichtig ist zu wissen, dass es auch Codebeispiele gibt, bei denen der Index 0 nicht verwendet wird und beim initialisieren auf den Wert 0 gesetzt wird. So werden wir etwa (der Übersicht wegen) im Beispiel im Abschnitt 5.5.6. Die erwartete Laufzeit lautet bei diesem Algorithmus $\mathcal{O}(k \cdot l)$

5.5.6 Beispiel

Wir wollen den längsten, gemeinsamen Teilstring der Wörter *HUND* und *FUNKE* ansehen.

			H	U	N	D
		0	1	2	3	4
	0	0	0	0	0	0
F	1	0	↑ ← 0	↑ ← 0	↑ ← 0	↑ ← 0
U	2	↑ ← 0	↑ ← 0	↖ +1 1	↑ ← 1	↑ ← 1
N	3	↑ ← 0	↑ ← 0	↑ ← 1	↖ +1 2	↑ ← 2
K	4	↑ ← 0	↑ ← 0	↑ ← 1	↑ ← 2	↑ ← 2
E	5	↑ ← 0	↑ ← 0	↑ ← 1	↑ ← 2	↑ ← **2**

Die Frage, die sich nun noch stellt ist, wie der längste gemeinsame Teilstring in der Tabelle gefunden wird, während die Länge des Strings in der Zelle (k, l) steht. Hierzu wird bei (k, l) angefangen, der Weg rückwärts gegangen. Wird ein diagonaler Pfad eingeschlagen, so wird das Zeichen ausgegeben. Das Ergebnis muss jedoch nochmal gedreht werden. Natürlich könnte der String auch entsprechend in einem *Stack* beim Aufbau der Tabelle gespeichert werden.

5.6 Stringsuche

Wir wollen nun wir ein weiteres, wichtiges Teilproblem bei Zeichenketten lösen. So ist es beispielsweise meist nützlich, in einem Text (bzw. einer beliebigen Zeichenkette) nach einem bestimmten Wort zu suchen.

Hierzu verwenden wir erneut unsere Definitionen zweier Strings:

$$s = s_1 s_2 \dots s_k \tag{5.27}$$
$$t = t_1 t_2 \dots t_l \tag{5.28}$$

Jedoch schränken wir nun die Länge von t ein, sodass dies ein Teilstring von s wird:

$$l \leq k \tag{5.29}$$

Wir wollen nun die Frage beantworten, ob t in s vorkommt, und wenn ja, an welcher Stelle. Der erste Einfall der uns hierzu kommt, ist selbstverständlich ein naiver Algorithmus.

Listing 5.3: StringSearchNaive.pseudo

```
1  for i=1 to k-l+1 {
2          j<- 1;
3          while (j <= l && s[i+j-1] == t[i]) {
4                  j++;
5          }
6          if (j == 'l+1') {
7                  return i;
8          }
9  }
10 return -1;
```

Dass dieser Ansatz bei einem sehr großen s unpraktisch ist, da so viele Vergleiche durchgeführt werden müssen. Wir müssen also einen Weg finden, die Zahl der Vergleiche zu beschleunigen. Hierzu nehmen wir uns ein Mittel, welches wir bereits aus dem Kapitel 3 kennen: die *Hashfunktion*.

5.6.1 Ansatz mit Hashfunktion - Rabin-Karp-Algorithmus

Wir wollen mit Hilfe der Hashfunktion nun nur noch die Teilstrings vergleichen. Wählen wir dazu eine gut verteilte Hashfunktion, so ist die Wahrscheinlichkeit gering, dass wir falsche Treffer erhalten. Da dies jedoch in keinem Fall auszuschließen ist, müssen wir bei der gleichheit der Hashwerte die Zeichenketten am Ende erneut einzeln vergleichen. Die Zahl der benötigten Vergleichsoperationen nimmt dadurch jedoch in einem großen Maße bereits ab!

Listing 5.4: HashStringSearch.pseudo

```
1  for i=1 to k-l+1 {
2          if (h(s[i]) == h(t[i])) {
3                  if s[i] == t[i] {
4                          return i;
5                  }
6          }
7  }
8  return -1;
```

Der Vergleich der Hashwerte benötigt nu $\mathcal{O}(1)$ Zeit und Kollisionen sind wie gesagt bei einer guten Verteilung selten zu erwarten. Daher muss können wir den Fall $h(s) == h(t) \wedge s \neq t$ in der folgenden Betrachtung vernachlässigen. Als Laufzeit für die heuristik erhalten wir jedoch $\mathcal{O}(k-l)$.

5.6.2 Problem mit der Hashfunktion

Wir haben jedoch noch ein Laufzeittechnisches Problem bei der Berechnung der Hashfunktion. Würden wir das Problem so angehen, wie es im Pseudocode gezeigt wurde, so würden wir an jeder Stelle den Hashwert ständig erneut berechnen müssen. Wir wünschen also statt einer *einmaligen, konstanten Laufzeit* eine *lineare* (jedoch letztlich *konstante*, da *amortisierte*) *Laufzeit*.

Die Lösung hierzu sieht so aus, dass wir h geschickt wählen, sodass $h(s[i])$ in $\mathcal{O}(1)$ Zeit berechnet werden kann. Hierzu müssen wir nur die Hashfunktion vom Substring, der ein Zeichen früher auftritt, kennen. Hierzu sehen wir uns zunächst die Hashfunktion von Strings erneut an:

$$h(s[1..i-1]) = \left(\sum_{i \in |s|}^{s[i-1]} |\Sigma| \, \varsigma_{i+j} \right) \mod p \tag{5.30}$$

Wollen wir also einen einen Teilstring $t_1 t_2 t_3$ in einem String $s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8$ suchen. Hierzu sehen die Operationen zunächst folgendermaßen aus:

$$h(s_1 s_2 s_3) = \left(s_1 \cdot 4^2 + s_2 \cdot 4^1 + s_3 \cdot 4^0\right) \mod p \tag{5.31}$$
$$h(s_2 s_3 s_4) = \left(s_2 \cdot 4^2 + s_3 \cdot 4^1 + s_4 \cdot 4^0\right) \mod p \tag{5.32}$$
$$h(s_3 s_4 s_5) = \ldots \tag{5.33}$$

Wir würden jedoch Zeit sparen, würden wir uns, wie gesagt, das vorhergehende Ergebnis zwischenspeichern. Daher lautet die allgemeine Lösung nun also:

$$h(s[i+1, \ldots, i+l-1]) = (|\Sigma| \, h(s[i, \ldots, i+l-1])) - \left(\Sigma^l s_i + s_{i+1}\right) \mod p \tag{5.34}$$

Somit können wir nun $h(s[i+1, \ldots, i+l])$ in $\mathcal{O}(1)$ aus $h(s[i, \ldots, i+l])$ berechen! Dieser Algorithmus geht auf *Richard Karp* und *Michael Rabin* zurück, und wird deswegen auch *Rabin-Karp-Algorithmus* genannt.[3, S. 911-915][4, S. 990-994]

5.6.3 Anwendung

Anwendung findet diesee Lösung unter anderen bei

- Passwort verschlüsselung und vergleich
- E-Mail Signaturen
- Prüfsummen

All diese Lösungen setzen auf dem sogenannten *Fingerprinting* von Strings. So werden nur Hashwerte statt Strings miteinander verglichen.

5.6.4 Bemerkungen

Man hofft auf eine Laufzeit von $\mathcal{O}(k + l)$. Dies tritt jedoch nur auf, wenn Kollisionen selten sind. Durch eine geeignete Wahl von p kann man $\mathcal{O}(k + l)$ erwartete worst-case Laufzeit erreichen.

5.7 Wörterbücher/Tries

Wir schließen das Kapitel Strings mit dieser Betrachtung nun ab. Das Problem ist nun, dass eine Menge von Zeichenketten gespeichert werden soll, dabei sollen jedoch die Operationen des *Abstrakten Datentypen Tries* unterstützt werden. Bei unserer Betrachtung sollen jedoch nun keine Einträge gespeichert werden, sondern nur die Speicherung von Schlüsseln. Hierfür haben wir mehrere Lösungen:

1. Wir nehmen eine bekannte Implementierung des Wörterbuchs und berechnen den Hash aus dem gegebenen Schlüssel und müssen diesen nur normal speichern. Diese Lösung ist aber unschön, da mehr als $\mathcal{O}(\log n)$ Zeit benötigt, da der Vergleich bereits länger dauert (dies ist jedoch nicht die konstante Zeit, sondern abhängig von der Länge des kürzeren Strings). Beim *AVL-Baum* wird somit etwa $\mathcal{O}(|s| \log n)$ Zeit benötigt, da die spezifische Struktur der Daten ignoriert wird.

2. Wir verwenden eine spezialisierte Datenstruktur für die interne Repräsentation. Das kann zum einen eine bessere Laufzeit liefern, jedoch auch mehr Struktur bringen. Dadurch erschließen sich weitere Anwendungsgebiete. Wollen wir Beispielsweise die Schlüssel $G = \{HALLO, HAND, HUND, TAG\}$ speichern, so entwickeln wir einen neuen Baum. Er wird in Abbildung 5.8 gezeigt.

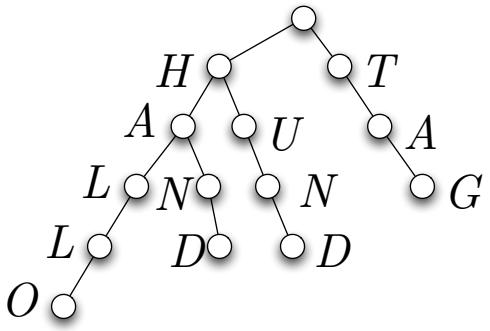

Abbildung 5.8: Ein *Trie* in dem die Schlüssel *HALLO, HAND, HUND* und *TAG* gespeichert wurden. Die Werte für das Wörterbuch stehen dann in den entsprechenden Blattknoten.

Es handelt sich bei dem Baum in Abbildung 5.8 um einen sogenannten *Trie*. Ein *Trie* ist ein *Mehrwegbaum*, da jeder innere Knoten einen bis $|\Sigma|$ Kinderknoten hat.

Die Kanten wurden beschriften mit den zugehörigen $\varsigma \in \Sigma$, sodass jeder innere Knoten höchstens eine Kindkante pro $\varsigma \in \Sigma$ hat. Die Blätter entsprachen den Wörtern die in dem Trie gespeichert wurden. Das Wort erhält man, indem man den jeweiligen Pfad von der Wurzel an folgt und die Zeichen aneinanderreiht.

Ein Problem an dieser Darstellung stellen jedoch noch Wörter da, die im Präfix eines anderen Wortes erscheinen. Diese verfälschen die Definition des Tries. Ein Beispiel wäre etwa das Wort *HALL*. Diesses wird jedoch durch den Schlüssel *HALLO* verdeckt. Dieses lässt sich jedoch leicht umgehen. Zum einen kann die Datenstruktur erweitert werden, sodass eine Markierung eingeführt wird, welche das Ende des Wortes anzeigt. Eine weitere Möglichkeit wäre es, diesen Fall unmöglich zu machen. Dazu wird ein neues Symbol ς eingeführt, für das gilt: $\varsigma \notin \Sigma$. So kann der Baum im Bild 5.8 um das Zeichen *§* erweitert werden und zum Baum in 5.9 werden.[8, S. 554ff.][13, S. 68-70]

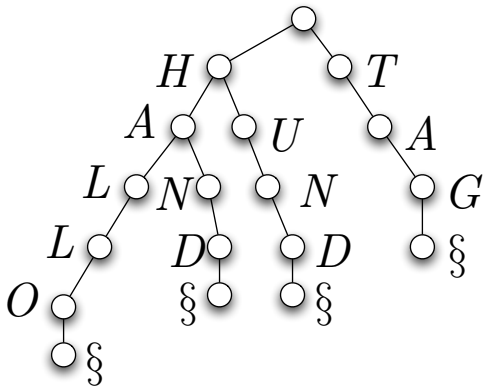

Abbildung 5.9: Der Trie aus 5.8 wird um das Abschlusssymbol § erweitert.

5.7.1 Analyse

Betrachten wir zunächst den *Speicherbedarf* eines Tries. Dazu definieren wir zunächst:

$$K = \{K_1, K_2, \ldots, K_l\} \tag{5.35}$$
$$K \in \Sigma^* \tag{5.36}$$

Der hier benötigte Platz ist somit $\mathcal{O}(\sum_{i=1}^{l} K_i)$. Daher gilt für die Operationen *put()*, *get()*, *pred()*, *succ()* und *remove()*, dass diese sich jeweils in Alphabet größe Ausführen implementieren lassen: $\mathcal{O}(|\Sigma| \cdot |K_i|)$, wobei k der Schlüssel ist, auf dem die jeweilige Operation ausgeführt wird. Bei *pred()*, *succ()* gilt $\mathcal{O}(|\Sigma||K| + |K'|)$, ergibt es sich aus k' des Vorgängers.[8, S. 554-557]

5.7.2 Komprimierte Tries (PATRICIA Tries)

Wie man in Abbildung 5.9 sieht, ist der einfache Trie sehr verschwenderisch, da die meisten Knoten eh nur einen Kinderknoten haben. Stattdessen wäre es natürlich einfacher, wenn diese zu einem Knoten zusammengefasst würden (s. Abbildung 5.10).

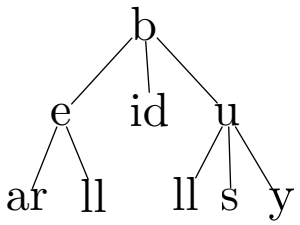

Abbildung 5.10: Ein komprimierter Trie.

Komprimierte Tries heißen auch *PATRICIA Tries*, wobei Patricia nicht als Name sondern für *Practical Algorithm to Retrieve Information Coded in Alphanumeric* steht. Hier werden zwar Knoten, jedoch kein Platz gespart, da der entsprechende Teilstring weiterhin abgespeichert. Jedoch werden hier nur $\mathcal{O}(l)$ statt $\mathcal{O}(\sum_{i=1}^{l} |K_i|)$ viele Knoten gespeiechert. Hier gibt es jedoch nur einen asymptotischen (also theoretischen) Vorteil, außer es werden Suffixbäume gespeichert.[8, S. 558-559]

5.7.3 Suffixbaum

Ein Suffixbau ist ein komprimierter Trie, der alle Suffixe eines gelesenen Strings speichert. Wollen wir etwa das Wort *MISSISSIPPI* in einem Suffixbaum speichern, legen wir zunächst die Suffixe des Wortes an.

M	:	ISSISSIPPI
I	:	SSISSIPPI, SSIPPI, IPPI, §
S	:	SISSIPPI, ISSIPPI, SIPPI
P	:	PI, I

Als nchstes generieren wir den dazugehörigen Suffixbaum, zu sehen ist er in Abbildung 5.11.

Ein String der Länge n hat $\mathcal{O}(n)$ Suffixe, also hat der Suffixbaum $\mathcal{O}(n)$ viele Knoten, aber die Gesamtgröße der Labels kann $\mathcal{O}(n^2)$ sein.

Suffixbäume eignen sich somit zum Suchen in sehr langen Texte. Da hier bereits die Startpositionen durch das erste Zeichen gefiltert werden, können Mustersuchen somit beschleunigt werden. Die Suche in einem Suffixbaum kann somit in $\mathcal{O}(|t|)$ Zeit durchgeführt werden. Die Größe eines Suffixbaumes kann jedoch quadratisch sein, was sich letztlich durch die Verwendung von Zeigern lösen lässt. Hierbei wird im Baum nicht mehr der jeweilige Teilstring, sondern nur noch die Start und Endpositionen eines Teilstrings gespeichert, wodurch $\mathcal{O}(n)$ Speicherungen durchgeführt werden müssen.

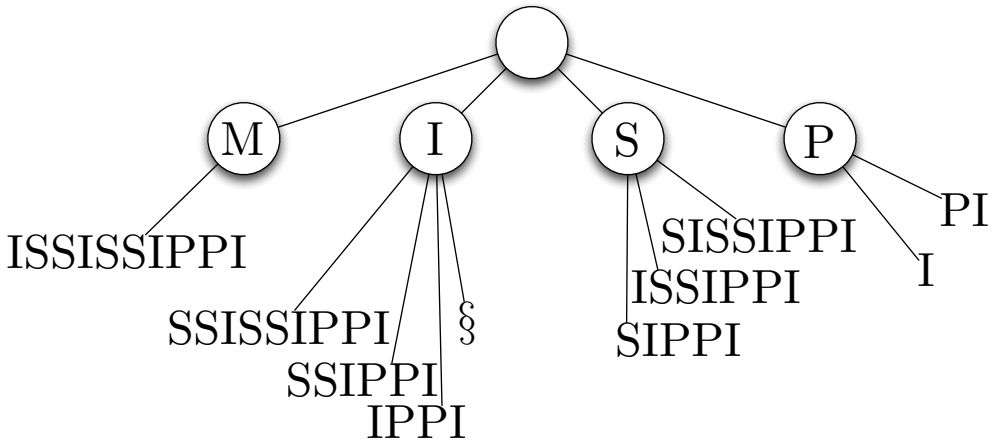

Abbildung 5.11: Der Suffixbaum für das Wort *MISSISSIPPI*

5.7.4 Generieren von Suffixbäumen

Als abschließende Frage, wollen wir uns noch um die Konstruktion von Suffixbäumen kümmern. Dieses Thema wird jedoch nur kurz angeschnitten. So dauert die naive Erstellung des einfachen Einfügens $\mathcal{O}(n^2)$ Zeit. Diese Operation kann jedoch verbessert werden, sodass nur $\mathcal{O}(n)$ Zeit benötigt wird. Dieser Algorithmus nennt wurde nach *Sander* benannt, jedoch hier nicht näher besprochen. [8, S. 560-563]

6 Graphen

Ein Graph $G = (V, E)$ besteht aus den Knoten V und den Kanten E. Die Kanten sind Teilmengen der Graphen. Einige Beispiele gibt es sind der Abbildung 6.1 zu sehen.

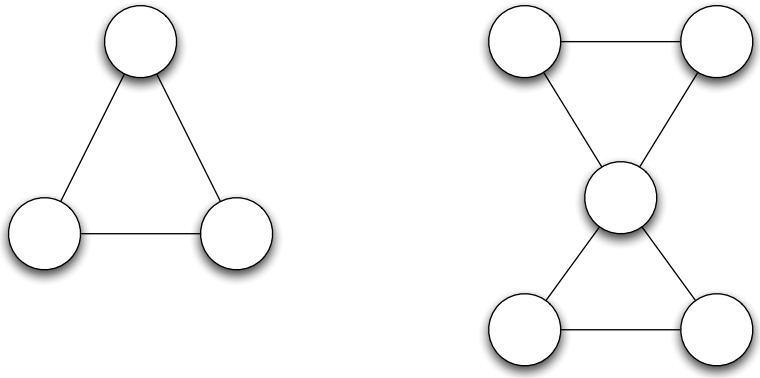

Abbildung 6.1: Zwei einfache, ungerichtete und ungewichtete Graphen.

6.1 Varianten

Weiter können die Regeln für Graphen erweitert oder eingeschränkt werden. Einige, wichtige Beispiele hierfür sind:

Multigraphen Erlauben Mehrfachkanten und Schleifen. Mehrfachkanten bedeutet, das zwischen zwei Knoten mehr als eine Kante gezogen werden darf(Abbildung 6.2).

gewichtete Graphen Die Kanten zwischen den Graphen bekommen Werte zugeordnet. Somit kann etwa bei einer Straßenkarte die Länge einer Straße an die Kante eingetragen werden (Abbildung 6.3).

gerichtete Graphen Die Kanten zwischen zwei Knoten dürfen nur noch in einer angegebene Richtung entlang gelaufen werden (Abbildung 6.4).

6.2 Beispiele

Nun wollen wir betrachten, in welchen bereichen Graphen angewendet werden:

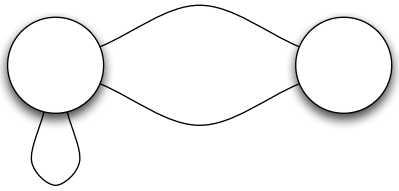

Abbildung 6.2: Ein Multigraph

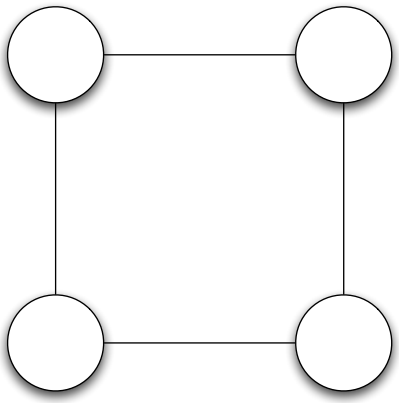

Abbildung 6.3: Ein gewichteteer Graph.

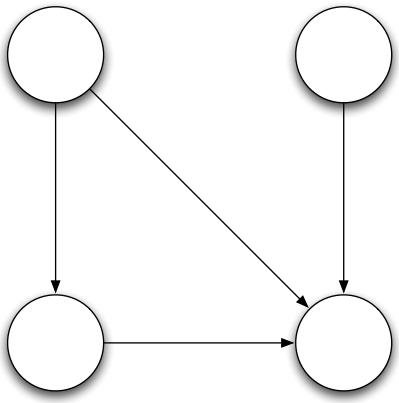

Abbildung 6.4: Ein gerichteter Graph.

- Bäume
- Soziale Netzwerke (zur Modellierung der Freundschaftsverhältnisse)
- Netzwerkgraphen
- Spielgraphen

6.3 Problemgebiete

Im Hinblick auf Graphen, werden höufig die folgenden Fragen gestellt:

- Gegeben sind zwei Knoten $u, v \in V$. Gibt es einen Weg von u nach v?
- Man kann den kürzesten, bzw. längsten Weg von u nach v suchen. Hierbei kann entweder die Zahl der zu besuchenden Kanten oder deren Gewichtung gezählt werden.
- Gegeben sein gewichteter Graph, in dem die Kanten Abhängigkeiten darstellen. Kann ein Graph so geordnet werden, dass kein Knoten von einem Nachfolger abhängig ist.
- Es sei erneut ein gewichteter Graph gegeben, jedoch sei die Frage gestellt, was die günstigste Möglichkeit ist, alle Knoten miteinander zu verbinden.
- Wie hoch ist die Zahl der Knoten die entfernt werden muss, um zwei Knoten $u, v \in V$ von einander zu trennen?

6.4 Darstellung von Graphen im Rechner

6.4.1 Adjazenzliste

Letztlich gibt es zwei Möglichkeiten um einfache Graphen darzustellen. Der einfachste Weg, um anzuzeigen, welche Knoten miteinander verbunden sind, ist es diese für jeden Knoten aufzulisten. Dieses Prinzip wird Adjazenzliste genannt (ein Beispiel gibt es in Abbildung 6.5).[8, S. 589-590] Der Platzbedarf in der Adjazenzliste ist $\mathcal{O}(|V| + |E|)$.

6.4.2 Adjazenzmatrix

Ein alternativer Ansatz für die Darstellung von Graphen sind Matrizen. Wird an der Stelle (i, j) eine 1 eingetragen, so existiert zwischen den Knoten i und j eine Kante. Als Beispiel kann man hierzu die Abbildung 6.6 ansehen. Der Platzbedarf in der Adjazenzmatrix ist $\mathcal{O}(|v|^2) = \mathcal{O}(n^2)$ groß, wenn n Knoten darin gespeichert werden.[8, S. 591-592]

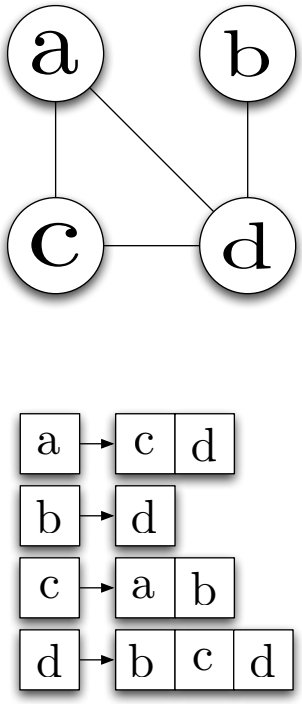

Abbildung 6.5: Eine Adjazenzliste für einen beispielhaften Graphen.

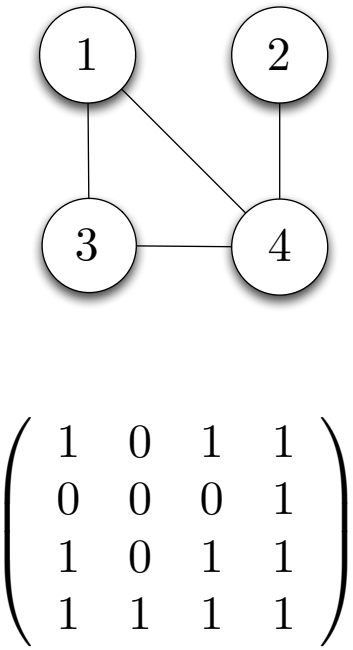

Abbildung 6.6: Eine Adjazenzmatrix für einen beispielhaften Graphen.

6.5 Definition des Abstrakten Datentyps

Bevor wir uns näher mit den Eigenschaften von Graphen beschäftigen, definieren wir die Operationen für den Datentyp.

vertices() Gibt, mit Hilfe eines Iterators, die Menge der Knoten zurück.

edges() Gibt, mit Hilfe eines Iterators, die Menge der Kanten zurück.

incidentEdges(v) Gibt alle benachbarten (inzidenten) Kanten von v zurück.

insertEdge(u,v) Fügt eine Kante zwischen den Knoten u und v ein.

removeEdge(e) Löscht die Kante e

newNode() Erstellt einen neuen Knoten

deleteNode(v) Löscht den Knoten v

endVertices(e) Findet die benachbarten Knoten von e

opposite(e,v) Gibt den benachbarten Knoten von v zurück, der über die Kante e verbunden ist.

get/setNodeInfo(u) Ändert die interne Beschreibung des Knotens u, bzw. gibt sie zurück

get/setEdgeInfo(e) Ändert die interne Beschreibung der Kante e, bzw. gibt sie aus

6.6 Durchsuchen von Graphen

Es sei ein Startknoten s gegeben. Nun sollen systematisch alle Knoten aufgezählt werden, die von s aus erreichbar sind. Hierzu gibt es zwei Möglichkeiten für den Ansatz, um den Graphen zu durchlaufen:

- Tiefensuchbäume (kurz: DFS)
- Breitensuchbäume (kurz: BFS)

6.6.1 Tiefensuche

Für die Tiefensuche, geht man einen Wegs so weit wie möglich. Hierzu müssen die Knoten gefärbt werden, und die besuchten Knoten werden auf einem Stack abgelegt. Der Pseudocode sieht somit folgendermaßen aus:

Listing 6.1: DFS.pseudo

```
1 s <- new Stack();
2 s.push(start);
3 while(!s.empty()) {
```

```
4           v <- s.pop();
5           v.visited<-true;
6           for (all e in incidentEdges(v)) do {
7                   w <- opposite(e,v);
8                   if (!w in s) {
9                           s.push(w)
10                  }
11          }
12  }
```

Die erwartete Laufzeit für den DFS Algorithmus lautet $\mathcal{O}(|v| + |E|)$ mit einer Adjanzenzliste, sonst $\mathcal{O}(|v|^2)$. [18, S. 726-729][12, S. 354-357][8, S. 593-604][4, S. 603-612][19, S. 435-439]

Anwendungen

Bei den folgenden Problemen, aber auch weiteren, wird der Tiefensuchalgorithnmus angewendet:

- Kann man einen Knoten v von u aus erreichen?
- Finden von Kreisen im Graph
- Finde einen aufspannenden Baum

Für das finden von kürzesten Wegen eignet sich der DFS Algorithmus jedoch nicht, hier wird die Breitensuche benötigt.

6.6.2 Breitensuche

Anstatt nun offensiv den jeweils nächsten Knoten zu besuchen, so wie es die Tiefensuche macht, geht die Breitensuche eher defensiv vor und sieht sich jeweils nur die Nachbarknoten an. Danach geht der Algorithmus zum ersten Nachbarknoten und schaut sich diese jeweils neu an. Für den Algorithmus wird statt eines Stacks, eine Schlange benötigt.

Listing 6.2: BFS.pseudo

```
1   i<-0;
2   while(!Q.empty()) {
3           for (all v in L) do {
4                   for (all e in G.incidentEdges(v)) do {
5                           if (e.unexplored) {
6                                   w<-v.endVertices(e)
7                                   if (w.unexplored) {
8                                           e.label(DISCOVERY_EDGE)
                                                ;
9                                           Q.insert(w);
10                                  } else {
```

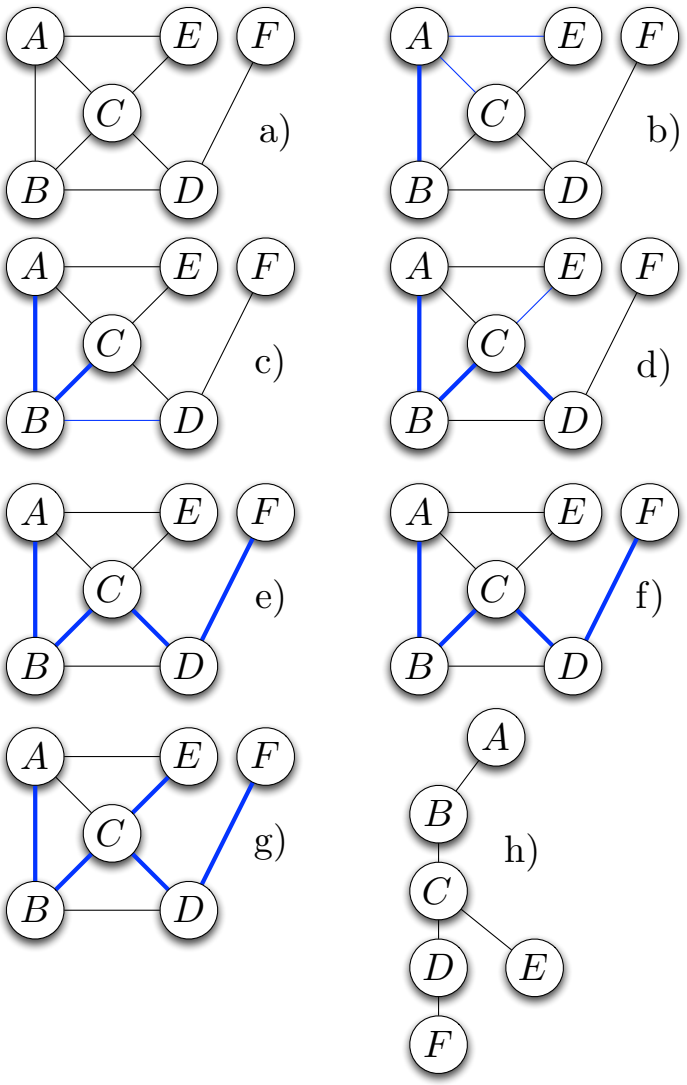

Abbildung 6.7: Der Algorithmus der Tiefensuche wird auf einen Graphen mit aufsteigender Adjenzenzliste angewendet. Mit blau-markierte Kanten stehen in diesem Schritt als besuchbar bereit. Gefettete Kanten wurden gewählt. In Schritt g) geht der Algorithmus solange zurück, bis er einen Knoten findet, noch nicht besucht wurde. Hier findet sich der Knoten E. In Abbildung h) ist der resultierende Baum dargestellt.

```
11                                                    e.label(CROSS_EDGE);
12                                              }
13                                        }
14                                  }
15                            }
16                            i <- i+1
17  }
```

Mit Hilfe des Breitensuchalgorithmus lassen sich die folgenden Probleme lösen:

- Prüfen ob G verbunden ist.
- Finden der Verbundenen Knoten von G.
- Prüfen, ob G Kreise enthält.

[18, S. 729-731][12, S. 357-359][8, S. 605-607][4, S. 594-602][3, S. 532-537][19, S. 431-435]

Laufzeitbetrachtung

Sowohl der Algorithmus für die Tiefensuche, als auch der Breitensuche haben eine Laufzeit von $\mathcal{O}(|V| + |E|)$.

6.7 Kürzeste Pfade

Ein übliches Problem für Graphen ist es, kürzeste Wege zu finden. Ein solcher Weg führt von einem Knoten v zu einem Knoten u in einem zusammenhängenden Graphen G.

Als Beispiel wollen wir uns den Plan eines öffentlichen Verkehrsnetzes ansehen (Abbildung 6.9).

Wir können uns für das Problem der *Kürzesten Pfade* die folgenden Fragen aufstellen:

1. Wie lauten die kürzesten Pfade, um von einem gewählten Bahnhof x andere Knoten zu erreichen. Dies ist das *Single Source Shortest Path* Problem (kurz: *SSSP*).
2. Wie kommen wir von einem gewählten Bahnhof x zu einem Bahnhof y, wobei dabei die geringste Zahl an weiteren Bahnhöfen besucht werden soll. Diese Frage nennt sich auch *Single Pair Shortest Path* Problem (kurz: *SPSP*).
3. Wie lauten die kürzesten Pfade von allen Knoten zu allen Knoten. Das ist das *All Pair Shortest Path* Problem (kurz: *APSP*).

Wir wollen uns zunächst dem *SSSP* Problem annehmen. Für den Fahrplan, wie wir ihn oben sehen, bei dem die Kanten nicht gewichtet wurden, reicht es aus, den Algorithmus der Breitensuche so zu modifizieren, als dass für jeden Knoten die Vorgänger und Abstände zum Startknoten s mitgespeichert werden.

Sind die Kanten jedoch gewichtet (etwa mit der Fahrzeit zwischen den Bahnhöfen) reicht das nicht mehr aus, da letztenendes andere Pfade kürzere Fahrzeiten enthalten können. Hier müssen wir den *Algorithmus von Dijkstra* anwenden.

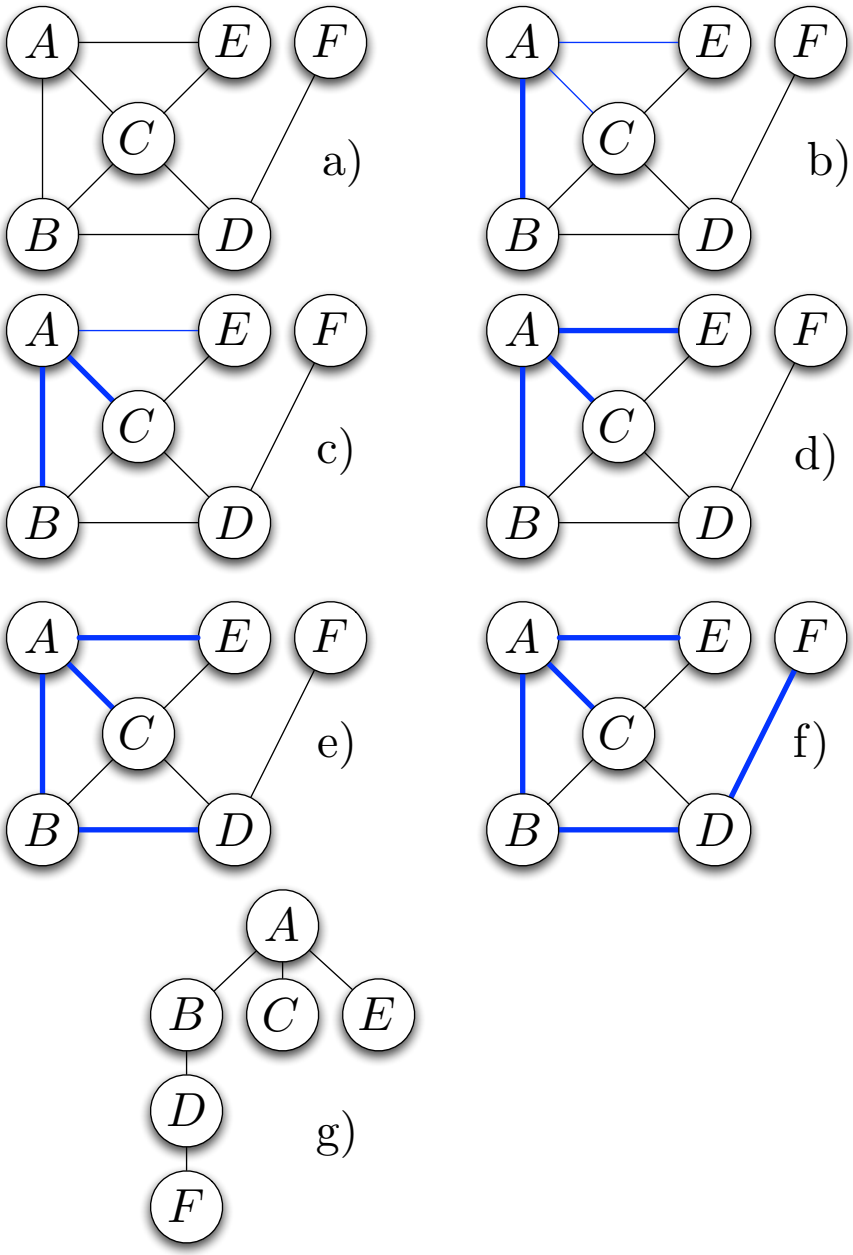

Abbildung 6.8: Der Algorithmus der Breitensuche wird auf einen Graphen mit aufsteigender Adjenzenzliste angewendet. Mit blau-markierte Kanten stehen in diesem Schritt als besuchbar bereit. Gefettete Kanten wurden gewählt. In Abbildung h) ist der resultierende Baum dargestellt.

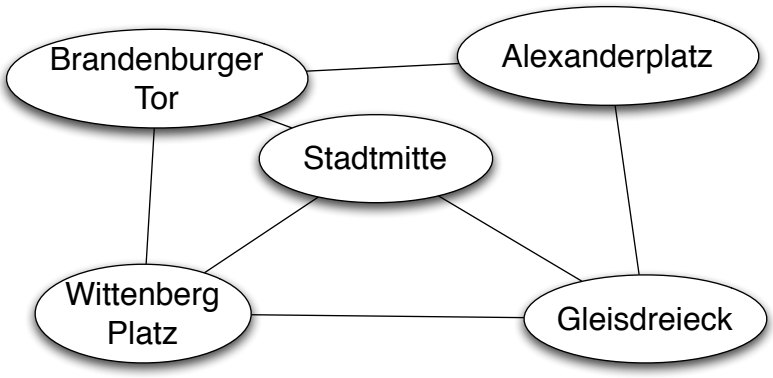

Abbildung 6.9: Ein fiktiver Fahrplan für Berlin. Wie man sieht, können die Stationen als Knoten gesehen werden, wobei die Kanten die direkte Erreichbarkeit von einem beliebigen Knoten aus darstellen.

6.7.1 Single Source Shortest Path - Dijkstras Algorithmus

Dieser Algorithmus geht auf *Edsger W. Dijkstra* zurück. Die Grundidee geht dabei erneut auf einen Breitensuchbaum zurück, welcher jedoch vorallem auf die Gewichte der Kanten achtet. Der Algorithmus geht dabei wieder Greedy vor.

Ein wichtiges Prinzip ist hier, dass die Entfernungen in den Kanten abgespeichert werden. Dabei wird die kleinste, bisher gefundene Summe der Gewichte die zum Knoten führt gespeichert. Sehen wir uns zunächst den Pseudocode für den Algorithmus an. Als Eingabe wird hier der Graph G übergeben, sowie der Startknoten s. Wir benötigen jedoch zwei Funktionen. Zum einen müssen wir eine Funktion in G hinzufügen, die uns für eine Kante zwischen den Knoten u und v das Gewicht ausgibt. Wir nennen sie *distance* Außerdem brauchen wir eine Funktion *audjust*, die die Warteschlange an die neuen Werte anpasst (also sortiert).

Listing 6.3: Dijkstra.pseudo

```
1  for all (v in V[G]\ s) do {
2          D[v] := −1
3  }
4  D[s] := 0;
5  Q := V;
6  while (!Q.isEmpty) do
7          u := Q.extractMinimum;
8          for all (v = G.adjacent(u)) do {
9                  if ((D[u] + G.distance(u,v)) < D[v]) {
10                         D[v] := D[u] + G.distance(u,v);
11                         Q.adjust(D[v]);
12                 }
13         }
```

Anstatt -1 für Knoten, deren Kantengewichte wir noch nicht kennen, wird auch meist ∞ verwendet. Nun wollen wir uns an unserem fiktivem Fahrplan einmal ansehen, wie der Algorithmus funktioniert. Dazu müssen (Abbildungen 6.10 & 6.11).

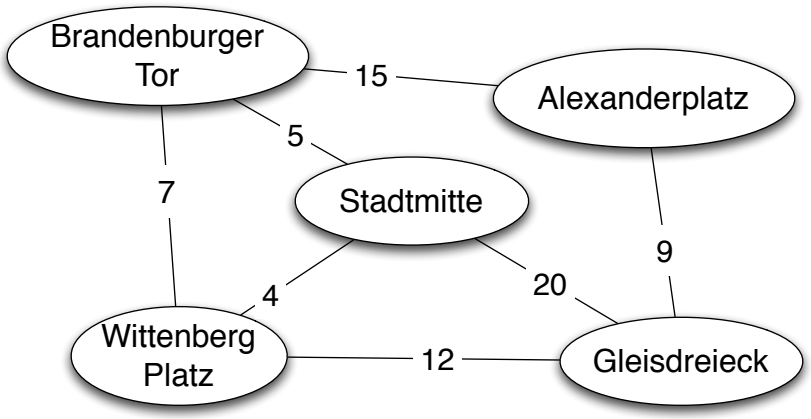

Abbildung 6.10: Der Fahrplan wurde um Gewichte erweitert.

Nun wollen wir den Algorithmus auf dem Graphen anwenden. Als Startknoten s wählen wir Stadtmitte. Somit hat der Dijkstra Algorithmus, der auf einen Graphen mit n Knoten angesetzt wird, im schlechtesten Fall eine Laufzeit von $\mathcal{O}(n^2 \cdot \log n)$.

6.7.2 Single Pair Shortest Path - A*-Algorithmus

Nun wollen wir den kürzesten (bzw. leichtesten) Pfad von einem Startknoten s zu einem Zielknoten t finden. Hierzu wird der A^*-Algorithmus verwendet. Dieser addiert die Pfade über einen Suchlauf solange, bis die Werte größer sind, als der bisher kürzeste Pfad und wählt dann einen weiteren Knoten und versucht dies dann ebenso.

Wir benötigen zusätzlich zum Graphen jedoch eine Heuristikfunktion h, sodass wir für jeden beliebigen Knoten v die Entfernung nach t abschätzen können. Wir bezeichnen h als *konsistent*, wenn für die Kante $e : a \to b$ gilt die neue Länge length' gilt:

$$e.\text{length'} \quad := \quad e.length + h(b) - h(a) \tag{6.1}$$

Somit können wir sicherstellen, dass die neuberechneten Längen nicht negativ sind, somit einfach verglichen werden können. Ein weiterer Vorteil der Heuristik ist, dass somit nun Kanten, die vom Ziel wegführen länger/schwerer werden, während zielführende Kanten kürzer/leicher werden, somit eher besucht werden, während der Rest schneller vernachlässigt werden kann.

Ein weiteres wichtiges Konzept ist die *Teleskopsumme*. Wir sagen hier aus, dass die Länge des Weges $s \to t$ um $h(t) - h(s)$ ändert. Somit ist ein kürzester Weg für die neuen Gewichte auch ein kürzester Weg für die alten Gewichte, da die Änderung nicht vom Weg sondern nur von den Knoten abhängt). Natürlich ist die Heuristik austauschbar,

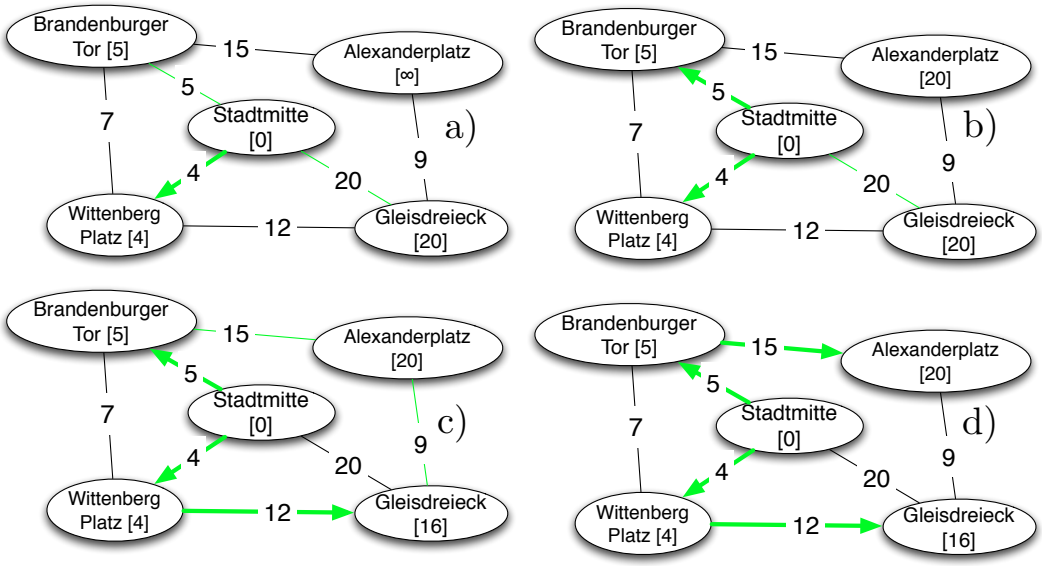

Abbildung 6.11: Der Algorithmus von Dijkstra wird auf den Graphen aus Abbildung 6.10 angewendet.

jedoch sollte hier eine gute Funktion gewählt werden, da sie entsprechend die Zahl der zu prüfenden Knoten verringern kann.

6.7.3 All Pair Shortest Path - Algorithmus von Floyd-Warshall

Wollen wir jedoch nicht nur die kürzesten Weg von einem beliebigen Knoten zu einem oder allen anderen Knoten finden, sondern für alle denkbaren Paare, so verwenden wir den *Algorithmus von Floyd-Warshall* . Er wurde von *Robert Floyd* und *Stephen Warshall* getrennt entwickelt und basiert auf *dynamischer Programmierung*. Die Grundidee liegt hier in der *Teilpfadoptimimalität*.
Satz: Sei ein Weg $v \to w$ in einem Graph G optimal, so sind auch die Zwischenschritte des Graphen optimal.
Beweis: Nehmen wir an, der Weg $v \longrightarrow w$ bestünde aus einem Weg $v \to t \to w$, so müssen die die Wege $v \to t$ und $t \to w$ auch optimal sein. Wäre dies nicht der Fall, so wäre auch auch $v \longrightarrow w$ nicht optimal!
Wir können also uns von einem Knoten aus, seine Nachbarn ansehen und somit zunächst die erreichbaren Nachbarknoten bestimmen. Aufgrund des oben beschriebenen Satzes, Können wir jedoch den kürzesten Weg zu einem Knoten bestimmen, indem wir den kürzesten Pfad zu den Zwischenknoten suchen. Kennen wir etwa den kürzesten Pfad $a \to b$, und wissen dass der kürzeste Pfad $b \to c$ existiert, so haben wir auch gleichzeitig den kürzesten Pfad für $a \longrightarrow c$.

Dem Algorithmus müssen wir nur einen Graphen G mit n Knoten übergeben, und erhalten dann entsprechend die jeweils kürzesten Wege für alle Paare des Graphen.

Listing 6.4: FloydWarshall.pseudo

```
1  G[0] <- G
2  for (k <- 1 to n) {
3          G[k] <- G[k-1]
4          for all (i,j in {1,...,n} with i != j && i,j != k) {
5                  if((v[i], v[k]) && (v[k], v[j]) in G[k-1]) {
6                          G[k-1].add(v[i],v[j])
7                  }
8          }
9  }
10 return G[n]
```

Sehen wir uns abschließend die Laufzeit des Algorithmus an. Wird der Algorithmus auf einen Graphen mit n Knoten angesetzt, so erhält man eine Laufzeit von $\mathcal{O}(n^3)$. Der Vorteil ist jedoch, dass selbst bei einem *dichten Graphen*, als einem Graphen mit $\Omega(n^2)$ Kanten, so lautet die Laufzeit weiterhin $\mathcal{O}(n^3)$.

6.8 Minimum Spanning Trees

Anstatt nun einzelne Pfade mit geringen Gewichten zu suchen, wollen wir einen Baum erstellen, dessen Kanten aus den kleinsten Gewichten besteht. Somit ist das Ergebnis ein *Minimal aufspannender Baum* (kurz: *MST*).

6.8.1 Algorithmus von Prim

Der Algorithmus von Prim, verwendet einen Startknoten s und sucht von dort aus jeweils die Kanten mit den kürzesten Gewichten, die von den bisher gefundenen Knoten erreicht werden können. Der Pseudocode für diesen Algorithmus sieht daher wie folgt aus[4, S. 634]:

Listing 6.5: Prim.pseudo

```
1  for (all v in G.V) {
2          v.key = -1;
3          u.p = NULL;
4  }
5  s.key = 0;
6  Q = G.V;
7  while (Q != EMPTY) {
8          e = q.extractMin();
9          for (all v in G.Adj[u]) {
10                 if (v in Q and w(u,v) < v.key) {
11                         v.p = u;
12                         v.key = w(u,v)
13                 }
```

```
14              }
15  }
```

Sehen wir uns hierzu unseren Graphen an: Wircht zu erkennen ist, dass der Algorithmus

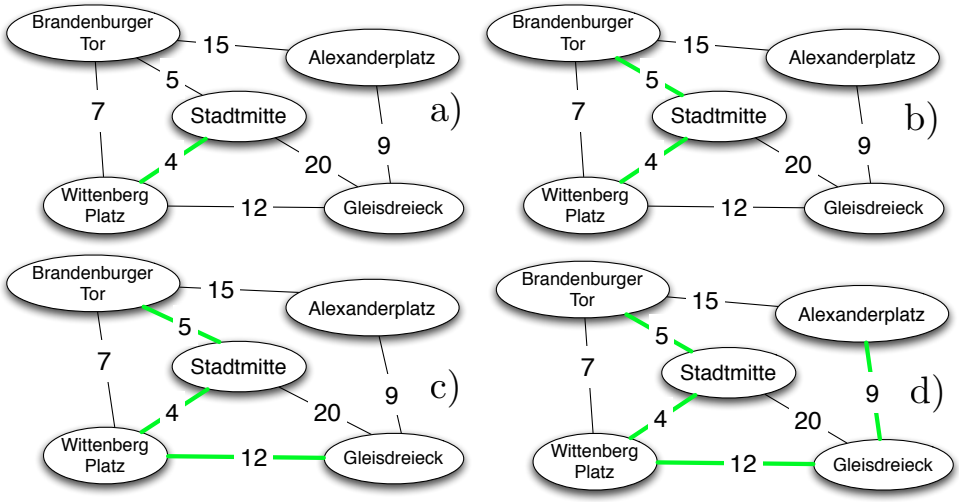

Abbildung 6.12: Der *Algorithmus von Prim* wird auf den Graphen aus Abbildung 6.10 angewendet. Als Startknoten s wird Wittenbergplatz gewählt.

keine Kreise geschlossen werden, da das Ergebnis ein Baum sein soll! Die Laufzeit für den Algorithmus von Prim lautet $\mathcal{O}(E + V \log V)$.

6.8.2 Algorithmus von Kruskal

Der Algorithmus von Kruskal ist denkbar einfach. Wir sortieren die Kanten nach ihren Gewichten aufsteigend, und markieren jeweils, wenn sie eine Zusammenhangskompente bilden, die keinen Kreis enthält (s. Abschnitt 4). Sehen wir uns dazu den Pseudocode an[4, S. 631]:

Listing 6.6: Kruskal.pseudo

```
1  A = EMPTY;
2  for (all v in Vertex) {
3          U.makeSet(v);
4  }
5  sort(U);
6  for (all (u,v) in E) {
7          if (U.findSet(u) != U.findSet(v)) {
8                  A = A++{(u,v)};
9                  U.union(u,v);
10         }
```

```
11 }
12 return A
```

Wie man erkennt, ist auch der Algorithmus von Kruskal ein *gieriger Algorithmus*. Sehen wir uns das Verhalten des Algorithmus von Kruskal nochmal auf unserem Fahrplan an.

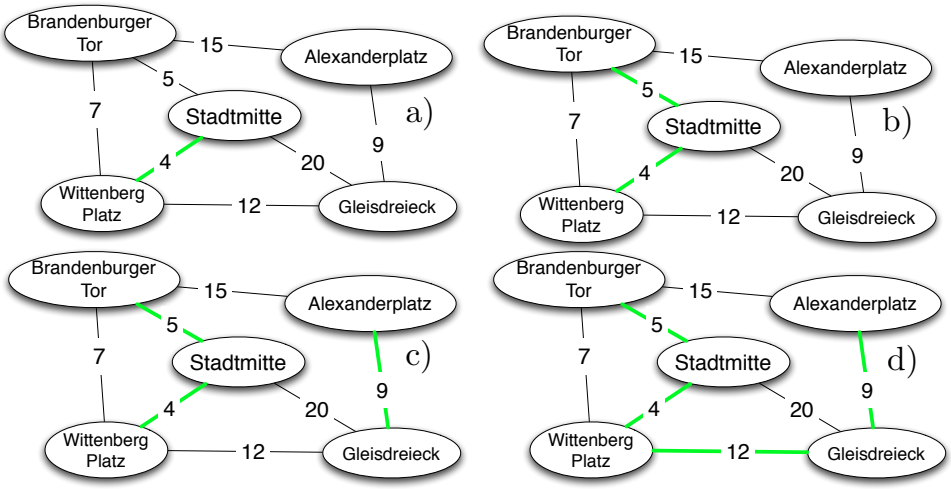

Abbildung 6.13: Der *Algorithmus von Kruskal* wird auf den Graphen aus Abbildung 6.10 angewendet. In Abbildung c) wird die Kante Brandenburger Tor ↔ Wittenbergplatz ausgelassen, da ansonsten ein Kreis entstehen würde und der endgültige Graph letztlich nicht mehr zyklenfrei wäre. Dies gilt ebenso für die Kanten Stadtmitte ↔ Gleisdreieck und Brandenburger Tor ↔ Alexanderplatz in Abbildung d).

Die Laufzeit des Algorithmus von Kruskal liegt bei $\mathcal{O}(E \log V)$.

6.8.3 Algorithmus von Borůvka

Kurz wollen wir den *Algorithmus von Borůvka* besprechen. Dieser Algorithmus stellt eine Mischung aus den Algorithmen von Prim und Kruskal dar. Er lässt mehrere Zusammenhangskomponenten gleichzeitig wachsen. Der Pseudocode hierzu sieht wie folgt aus:

Listing 6.7: Boruvka.pseudo

```
1 A = EMPTY;
2 while (|A| < n-1) {
3         B = EMPTY;
4         for (all c in (V,A)) {
5                 find lightest edge e with exactly one endpoint
                     in C
6                 B.add(e)
```

```
7           }
8           A.addAll(B);
9   }
```

Die Laufzeit für den Algorithmus lautet $\mathcal{O}(|E| \log |V|)$.

Lemma: Nach dem i-ten Durchlauf der while-Schleifen haben alle Zusammenhangskomponenten mindesten 2^i viele Knoten.

Beweis: wir beweisen durch Induktion.

Induktionsanfang Vor dem ersten Durchlauf hat jede Komponente $1 = 2^0$ Knoten.

Induktionsschritt In jedem Durchlauf wird jede Zusammenhangskomponente mit mindestens einer anderen Zusammenhangskomponente vereinigt, also verdoppelt sich die Größe um mindesten e. Somit gibt es nach $\mathcal{O}(\log |V|)$ Durchläufen nur noch eine Zusammenhangskomponente.

6.9 Union-FindStruktur

Die *Union-Find Struktur* verwaltet statt eines Baumes ein Wald von Kanten welcher später zu einem Wald von Bäumen erwächst. Hierzu werden die folgenden Operationen[1] definiert:

MAKESET Erstellt einen einzelnen Knoten für den Parameter u.

FINDSET Der Paramter u wird in den Bäumen gesucht. Die Wurzel des Baumes, der u enthält wird dann zurückgegeben.

UNION Die Bäume, deren Wurzel durch die Parameter U_1 und U_2 übergeben werden, werden vereinigt. Dazu wird die Wurzel von U_2 unter U_1 gehangen.

Wie man sich leicht vorstellen kann, können die Bäume sehr schnell entarten, da die Bäume, die mit der UNION Funktion verschmolzen werden von sehr unterschiedlicher Größe sein können. Dies kann, wie bereits besprochen, letztlich die Laufzeit beim Parsen des Baumes verschlechtern. Daher gibt es verschiedene Heuristiken um das Entarten des Baumes möglichst zu verhindern oder zumindest abzuschwächen.

6.9.1 Heuristiken

Union-by-size Die Höhe des Baumes wird in der Wurzel gespeichert. Werden zwei Bäume nun verschmolzen, wird der kleinere Baum unter den größeren gehangen.

Pfadkompression Mache den Baum so klein wie möglich, indem beim Ausführen der FINDSET Operation alle Knoten die zwischen dem gesuchten Knoten u und der Wurzel r des Baumes unter r gehangen werden.

[1] Für eine interaktive Darstellung des Datentyps sei http://www.cs.unm.edu/~rlpm/499/uf.html empfohlen.

Satz: Werden beide Heuristiken zusammen genutzt, so resultiert die amortisierte Laufzeit pro Operation in $\mathcal{O}(\alpha(n))$. $\alpha(n)$ sei weiterhin definiert als die *inverse Ackermannfunktion*[2], wodurch die Laufzeit nur sehr langsam wächst.

6.10 Spielgraphen

[2]Für eine Definition sei auf http://en.wikipedia.org/wiki/Ackermann_function verwiesen

Index

Abbildungsverzeichnis

Literaturverzeichnis

[1] S. Albers. Onlinealgorithmen - was ist es wert, die zukunft zu kennen? *Informatik Spektrum*, 4:438–443, 2010.

[2] M. Block. *Java Intensivkurs - In 14 Tagen lernen, Projekte erfolgreich zu realisieren.* Springer, 2007.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introductions to Algorithms 2nd Edition.* MIT Press, 2 edition, 2001.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 3 edition, 2009.

[5] M. Dietzfelbinger and U. Schellbach. Weaknesses of cuckoo hashing with a simple universal hash class: The case of large universes. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '09, pages 217–228, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] H.-D. Ehrich. On the theory of specification, implementation, and parametrization of abstract data types. *J. ACM*, 29:206–227, January 1982.

[7] E. Freeman, E. Freeman, K. Sierra, and B. Bates. *Head First Design Pattern.* O'Reilly, 2004.

[8] M. T. Goodrich and R. Tamassia. *Data Structures & Algorithms in Java.* Wiley, 2006.

[9] J. Guttag. Abstract data types and the development of data structures. *Commun. ACM*, 20:396–404, June 1977.

[10] J. V. Guttag. *Abstract data types and the development of data structures*, pages 453–479. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[11] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Commun. ACM*, 21:1048–1064, December 1978.

[12] G. Haggard, J. Schlipf, and S. Whitesides. *Discrete Mathematics for Computer Science.* Thompson Brooks/Cole, 2006.

[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation 3rd Edition.* Addison Wesley, 2007.

[14] J. Loeckx. Algorithmic specifications: a constructive specification method for abstract data types. *ACM Trans. Program. Lang. Syst.*, 9:646–661, October 1987.

[15] C. Meinel and M. Mundhenk. *Mathematische Grundlagen der Informatik*. Viweg+Teubner, 2009.

[16] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, ESA '01, pages 121–133, London, UK, 2001. Springer-Verlag.

[17] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[18] K. H. Rosen. *Discrete Mathematics and its Applications - Sixth Edition*. MacGraw-Hill, 2007.

[19] G. Saake and Kai-Uwe-Sattler. *Algorithmen und Datenstrukturen - Eine Einführung in Java*. dpunkt.Lehrbuch, 2006.

[20] U. Seidel. Datenmobile: Btrfs - das designierte linux-standard-filesystem. *iX Magazin für professionelle Informationstechnik*, 2:108–111, Februar 2011.