Algorithmen und Berechnungskomplexität I

Prof. Dr. Heiko Röglin

mit Ergänzungen von Prof. Dr. Thomas Kesselheim
und Prof. Dr. Anne Driemel

Institut für Informatik
Universität Bonn

18. Januar 2023

Inhaltsverzeichnis

Bitte senden Sie Hinweise auf Fehler im Skript und Verbesserungsvorschläge an die E-Mail-Adresse `driemel@cs.uni-bonn.de`.

Kapitel 1

Einleitung

In dieser Vorlesung werden wir uns mit dem Entwurf und der Analyse von *Algorithmen* beschäftigen. Ein Algorithmus ist eine Handlungsvorschrift zur Lösung eines Problems, die so präzise formuliert ist, dass sie von einem Computer ausgeführt werden kann. Algorithmen sind heute so allgegenwärtig, dass sie kaum wahrgenommen oder gewürdigt werden. Wie selbstverständlich nutzen wir Navigationsgeräte, um den besten Weg vom Start zum Ziel zu bestimmen, oder Suchmaschinen, um innerhalb kürzester Zeit riesengroße Datenmengen zu durchsuchen. Dass dies überhaupt möglich ist, liegt zum Teil an der immer besseren Hardware, zu einem viel größeren Teil liegt es aber an den cleveren Algorithmen, die für diese Anwendungen entwickelt wurden.

Die *Algorithmik* beschäftigt sich damit, Algorithmen für verschiedene Probleme zu entwerfen. Die obigen Beispiele zeigen bereits, dass es oft eine ganz wesentliche Herausforderung ist, nicht nur korrekte, sondern auch effiziente Algorithmen zu entwerfen, die so schnell wie möglich das richtige Ergebnis liefern. Dieser Aspekt gewinnt immer mehr an Bedeutung, da die zu verarbeitenden Datenmengen in vielen Bereichen rasant wachsen. So enthält beispielsweise die Datenbank des OpenStreetMap-Projektes, das frei nutzbare Geodaten sammelt, mittlerweile über sechs Milliarden Knoten[1] und in Googles Suchindex befinden sich geschätzt mehr als 55 Milliarden Webseiten[2]. Ohne die richtigen Algorithmen wäre es selbst mit modernster Hardware unmöglich, so große Datenmengen sinnvoll zu verarbeiten. Darüber hinaus werden wir sehen, dass selbst bei Problemen mit relativ kleinen Datenmengen der Wahl des richtigen Algorithmus oft eine entscheidende Bedeutung zukommt.

Wir werden in dieser Vorlesung viele grundlegende Techniken zum Entwurf und zur Analyse von Algorithmen kennenlernen. Unser Fokus wird dabei auf theoretischen Betrachtungen und formalen Analysen der Algorithmen liegen. Zu einer vollständigen Behandlung eines Algorithmus gehört auch eine Implementierung und experimentelle Evaluation, auf die wir jedoch in dieser Vorlesung nur am Rande eingehen werden. Stattdessen sei jeder Teilnehmerin und jedem Teilnehmer empfohlen, die betrachteten Algorithmen zu implementieren und mit ihnen zu experimentieren, auch wenn dies

[1] https://wiki.openstreetmap.org/wiki/Stats (abgerufen am 23.08.2020)
[2] http://www.worldwidewebsize.com/ (abgerufen am 23.08.2020)

nicht explizit als Übungsaufgabe genannt wird. Dies führt erfahrungsgemäß zu einem tieferen Verständnis der behandelten Themen.

Das Skript deckt den Inhalt der Vorlesung ab. Dennoch empfehle ich den Teilnehmerinnen und Teilnehmern zur Vertiefung weitere Literatur. Es gibt eine ganze Reihe von empfehlenswerten Büchern, die die Themen dieser Vorlesung behandeln und an denen sich dieses Skript in weiten Teilen orientiert. Das Standardwerk von Cormen et al. [1], welches auch auf Deutsch [2] erschienen ist, ist sehr zu empfehlen. Es ist ausführlich und deckt fast alle Themen der Vorlesung ab. Weniger ausführlich, aber dafür hervorragend geschrieben und für den Einstieg ideal geeignet ist die „Algorithms Illuminated"-Reihe von Tim Roughgarden [7, 8, 9, 10] (sofern man sich nicht daran stört, dass es nur eine englische Ausgabe gibt). Das Buch von Steven Skiena [12] ist eine eher praktisch orientierte Einführung in die Algorithmik. Ebenso ist das Buch von Kleinberg und Tardos [4] zu empfehlen.

1.1 Grundbegriffe

Bevor wir zum ersten Algorithmus dieser Vorlesung kommen, führen wir einige grundlegende Begriffe ein. Wir verzichten an dieser Stelle allerdings auf formale Definitionen und erläutern die Begriffe nur informell. Ein Algorithmus erhält eine *Eingabe* und produziert dazu eine *Ausgabe*. Unter einem *Problem* verstehen wir eine Beschreibung, die angibt, wie Eingaben und Ausgaben aufgebaut sind, und die für jede Eingabe festlegt, welches die gültigen Ausgaben sind. Beim Sortierproblem besteht die Eingabe beispielsweise aus einer Menge von Zahlen (oder allgemein aus einer Menge von Objekten, auf denen eine Ordnungsrelation definiert ist) und die gewünschte Ausgabe ist eine aufsteigend sortierte Permutation dieser Zahlen. Eingaben für ein Problem Π nennen wir auch *Instanzen des Problems* Π. So ist beispielsweise $\{5, 2, 3, 1, 9, 7\}$ eine Instanz des Sortierproblems, für die $(1, 2, 3, 5, 7, 9)$ die einzige gültige Ausgabe ist. Ein Beispiel, bei dem es zu einer Eingabe mehrere gültige Ausgaben geben kann, ist das Primfaktorproblem, bei dem die Eingabe eine natürliche Zahl ist und alle Primfaktoren dieser Zahl eine gültige Ausgabe darstellen.

Ein *Algorithmus* für ein Problem Π ist eine Handlungsvorschrift, die präzise genug formuliert ist, um von einem Rechner ausgeführt zu werden, und die zu jeder Eingabe in endlich vielen Rechenschritten eine gültige Ausgabe berechnet. Während in der Definition von Problemen der Zusammenhang zwischen einer Eingabe und den möglichen Ausgaben normalerweise nicht konstruktiv beschrieben wird, liefern Algorithmen eine konstruktive Möglichkeit, Eingaben in zugehörige Ausgaben zu transformieren. Wir sagen, dass ein Algorithmus *korrekt* ist, wenn er auf jeder Eingabe nach endlich vielen Rechenschritten mit einer gültigen Ausgabe terminiert.

Es gibt verschiedene Möglichkeiten, Algorithmen zu beschreiben. Eine naheliegende Möglichkeit ist es, Quellcode in einer beliebigen Programmiersprache anzugeben. Der Vorteil dieser Methode ist, dass der Algorithmus so eindeutig bis ins kleinste Detail beschrieben ist. Gerade bei komplizierten Algorithmen können aber genau diese unter Umständen unwichtigen Details von den wesentlichen Ideen und Aspekten des Algorithmus ablenken. Deshalb werden wir Algorithmen in *Pseudocode* beschreiben. Dieser

ist in seiner Syntax an Java angelehnt, enthält aber an einigen Stellen auch informelle Beschreibungen, um von unwichtigen Details zu abstrahieren.

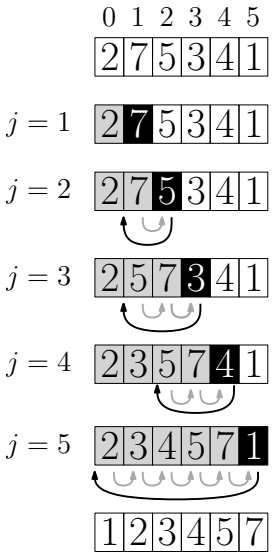
1.2 Ein erstes Beispiel: Insertionsort

Wir betrachten nun den ersten Algorithmus dieser Vorlesung. Es handelt sich dabei um Insertionsort (Sortieren durch Einfügen), einen einfachen Sortieralgorithmus, dessen Korrektheit intuitiv leicht einzusehen ist. Wir werden anhand dieses Algorithmus sehen, wie ein formaler Korrektheitsbeweis geführt werden kann und wie man die Laufzeit eines Algorithmus analysieren kann.

Die Eingabe für Insertionsort besteht aus einem Feld $a[0 \dots n-1]$ von ganzen Zahlen, das so permutiert werden soll, dass es die Zahlen in aufsteigender Reihenfolge enthält. Insertionsort ist durch das folgende Verfahren zum Sortieren von Spielkarten inspiriert. Alle zu sortierenden Karten werden zunächst auf einen Stapel auf den Tisch gelegt. Die Karten werden dann einzeln von diesem Stapel gezogen und direkt auf der Hand an der richtigen Stelle eingefügt. So ist sichergestellt, dass die Karten auf der Hand zu jedem Zeitpunkt sortiert sind. Insertionsort geht das Feld von links nach rechts durch und sorgt dafür, dass zu jedem Zeitpunkt der Bereich $a[0 \dots j-1]$ links von der aktuellen Position j aufsteigend sortiert ist. Dieser Bereich entspricht den Karten auf der Hand und der Bereich $a[j \dots n-1]$ entspricht den Karten, die sich noch auf dem Stapel befinden.

Der folgende Pseudocode unten links präzisiert diese Idee. Auf der rechten Seite ist ein Beispiel für eine Ausführung von Insertionsort angegeben.

```
Insertionsort(int[] a)
 1   for (int j = 1; j < a.length; j++) {
 2       // Füge a[j] in die bereits sortierte
             Sequenz a[0 ... j − 1] ein.
 3       int x = a[j];
 4       int i = j − 1;
 5       while ((i >= 0) && (a[i] > x)) {
 6           a[i + 1] = a[i];
 7           i−−;
 8       }
 9       a[i + 1] = x;
10   }
11   return a;
```



Theorem 1.1. *Die Ausgabe von* Insertionsort *ist stets eine aufsteigend sortierte Permutation der Eingabe.*

Beweis. Es sei $a[0 \ldots n-1]$ ein beliebiges Feld von ganzen Zahlen. Im Folgenden bezeichne $b[0 \ldots n-1]$ den initialen Inhalt dieses Feldes, den INSERTIONSORT als Eingabe erhält. Der Beweis beruht auf der folgenden Aussage.

> Bei jeder Überprüfung der Abbruchbedingung in Zeile 1 ($j < a.\text{length}$) ist $a[0 \ldots j-1]$ für das aktuelle j stets eine aufsteigend sortierte Permutation der Zahlen $b[0 \ldots j-1]$. Ferner gilt $a[j \ldots n-1] = b[j \ldots n-1]$.

Wir nennen eine solche Eigenschaft, die in jedem Schleifendurchlauf gilt, *Invariante*. Die Gültigkeit einer Invariante kann mithilfe vollständiger Induktion nachgewiesen werden. Für den Induktionsanfang muss gezeigt werden, dass die Invariante zu Beginn der ersten Iteration der for-Schleife gilt. Im Induktionsschritt muss gezeigt werden, dass unter der Annahme, dass die Invariante zu Beginn einer Iteration gilt, sie auch zu Beginn der nächsten Iteration gilt.

Wir zeigen zunächst den Induktionsanfang. Im ersten Schleifendurchlauf gilt $j = 1$. Der erste Teil der Invariante besagt somit, dass der aktuelle Inhalt von $a[0]$ eine sortierte Permutation von $b[0]$ ist. Da vor der for-Schleife keine Operationen durchgeführt werden, gilt $a[0] = b[0]$, womit die Aussage bewiesen ist. Der zweite Teil der Invariante besagt, dass $a[1 \ldots n-1] = b[1 \ldots n-1]$ gilt. Auch dies folgt daraus, dass vor der for-Schleife keinerlei Modifikationen des Feldes a vorgenommen werden.

Nehmen wir nun an, die Invariante gilt für ein $j \geq 1$. Dann ist zu Beginn der entsprechenden Iteration $a[0 \ldots j-1]$ eine aufsteigend sortierte Permutation der Zahlen $b[0 \ldots j-1]$ und es gilt $a[j \ldots n-1] = b[j \ldots n-1]$. In den Zeilen 3 bis 9 werden alle Einträge aus $a[0 \ldots j-1]$, die größer als das einzusortierende Element $a[j] = b[j]$ sind, um eine Position nach rechts geschoben. Das Element $b[j]$ wird dann an der dadurch frei gewordenen Position eingefügt. Am Ende des Schleifendurchlaufes ist somit $a[0 \ldots j]$ eine aufsteigend sortierte Permutation der Zahlen $b[0 \ldots j]$. Außerdem gilt $a[j+1 \ldots n-1] = b[j+1 \ldots n-1]$, da dieser Bereich des Feldes in dem Schleifendurchlauf nicht verändert wird. Zusammen bedeutet das, dass die Invariante auch zu Beginn der nächsten Iteration wieder gilt.

Im Prinzip müsste man natürlich auch noch formal beweisen, dass das Element $a[j] = b[j]$ wirklich an der richtigen Position einsortiert wird und dass alle größeren Elemente korrekt verschoben werden. Dies könnte man wiederum mithilfe einer Invariante für die while-Schleife tun. Die Leserin und der Leser mögen sich als Übung überlegen, welche Invariante dazu geeignet ist.

Aus der Gültigkeit der Invariante folgt direkt die Korrektheit des Algorithmus, denn für $j = n$ besagt die Invariante, dass $a[0 \ldots n-1]$ eine aufsteigend sortierte Permutation der Zahlen $b[0 \ldots n-1]$ ist. Da für $j = n$ die Abbruchbedingung der for-Schleife greift, ist dies auch die Ausgabe von INSERTIONSORT. □

Der obige Beweis war nicht schwierig und man hätte sicherlich auch weniger formal die einfach einzusehende Korrektheit von INSERTIONSORT beweisen können. Dennoch haben wir uns für diesen Beweis entschieden, da er das wichtige Konzept von Invarianten einführt. Dieses Konzept wird uns bei der Analyse komplizierterer Algorithmen oft wieder begegnen.

Um uns der Frage zu nähern, wie groß die *Laufzeit* von INSERTIONSORT ist, zählen wir zunächst für jede Zeile des Pseudocodes, wie oft sie ausgeführt wird. Dies ist natürlich davon abhängig, wie viele Zahlen zu sortieren sind. Ferner hängt insbesondere die Anzahl der Iterationen der while-Schleife davon ab, in welcher Reihenfolge die Zahlen in der Eingabe stehen. Sei eine beliebige Eingabe, die aus n Zahlen besteht, gegeben. Für $j \in \{1, \dots, n-1\}$ bezeichnen wir mit t_j, wie oft Zeile 5 in der entsprechenden Iteration der for-Schleife ausgeführt wird, d. h. wie oft die Bedingung der while-Schleife getestet wird. Mit dieser Notation können wir die Anzahl der Ausführungen der einzelnen Zeilen wie folgt angeben.

```
INSERTIONSORT(int[] a)                          Anzahl Ausführungen
 1  for (int j = 1; j < a.length; j++) {        n
 3      int x = a[j];                            n − 1
 4      int i = j − 1;                           n − 1
 5      while ((i >= 0) && (a[i] > x)) {         ∑_{j=1}^{n−1} t_j
 6          a[i + 1] = a[i];                     ∑_{j=1}^{n−1} (t_j − 1)
 7          i−−;                                 ∑_{j=1}^{n−1} (t_j − 1)
 8      }
 9      a[i + 1] = x;                            n − 1
10  }
11  return a;                                    1
```

Für $i \in \{1, \dots, 11\}$ bezeichnen wir mit c_i die Anzahl der Prozessorbefehle, die maximal notwendig sind, um Zeile i einmal auszuführen. Bei dem gegebenen Pseudocode ist es realistisch davon auszugehen, dass jedes c_i eine Konstante ist, die nicht von der Eingabe und insbesondere nicht von n abhängt. Damit können wir die Anzahl an Prozessorbefehlen, die INSERTIONSORT insgesamt benötigt, durch

$$c_1 n + (c_3 + c_4 + c_9)(n-1) + c_5 \left(\sum_{j=1}^{n-1} t_j \right) + (c_6 + c_7) \left(\sum_{j=1}^{n-1} (t_j - 1) \right) + c_{11}$$

nach oben abschätzen. Diesen Term werden wir im Folgenden auch als die Laufzeit von INSERTIONSORT auf der gegebenen Eingabe bezeichnen.

Diese Formel ist wegen der vielen Parameter, die in ihr vorkommen, recht unübersichtlich. Unser Ziel ist es, die Laufzeit von INSERTIONSORT nur in Abhängigkeit von der Anzahl der zu sortierenden Zahlen anzugeben. Dazu müssen wir die Parameter t_j abschätzen.

Optimistinnen und Optimisten könnten sagen, dass für jedes j im *Best Case* $t_j = 1$ gilt. Setzt man dies in die obige Formel ein, so ergibt sich

$$c_1 n + (c_3 + c_4 + c_5 + c_9)(n-1) + c_{11},$$

was für geeignete Konstanten $a, b \in \mathbb{R}$ durch $an + b$ nach oben beschränkt ist. Die Laufzeit von INSERTIONSORT wächst somit im Best Case nur *linear* mit der Anzahl der zu sortierenden Zahlen.

Realistinnen und Realisten werden an dieser Stelle direkt einwenden, dass $t_j = 1$ nur dann für alle j gilt, wenn die Eingabe bereits sortiert ist, wovon wir natürlich

im Allgemeinen nicht ausgehen können. Aus diesem Grund ist es sinnvoller, sich den *Worst Case* anzuschauen. Für jede Eingabe gilt $t_j \leq j+1$, da der Index i mit $j-1$ initialisiert und in jeder Iteration der while-Schleife um eins verringert wird. Setzt man $t_j = j+1$ (was eintritt, wenn die Zahlen in der Eingabe absteigend sortiert sind) in die obige Formel ein, so erhält man

$$\begin{aligned}
&c_1 n + (c_3 + c_4 + c_9)(n-1) + c_5\left(\sum_{j=1}^{n-1}(j+1)\right) + (c_6 + c_7)\left(\sum_{j=1}^{n-1} j\right) + c_{11} \\
&\leq c_5 \cdot \frac{n(n+1)}{2} + (c_6 + c_7) \cdot \frac{(n-1)n}{2} + (c_1 + c_3 + c_4 + c_9)n + c_{11} \\
&\leq (c_5 + c_6 + c_7)n^2 + (c_1 + c_3 + c_4 + c_9)n + c_{11},
\end{aligned}$$

was sich für entsprechende Konstanten $a, b, c \in \mathbb{R}$ als $an^2 + bn + c$ schreiben lässt. Die Laufzeit von Insertionsort wächst somit im Worst Case *quadratisch* mit der Anzahl der zu sortierenden Zahlen.

In der Algorithmik betrachtet man meistens den Worst Case, da eine kleine Schranke für die Worst-Case-Laufzeit eines Algorithmus eine starke Effizienzgarantie ist. Wir können garantieren, dass Insertionsort immer höchstens eine quadratische Laufzeit besitzt, und zwar vollkommen unabhängig von der Reihenfolge der Zahlen in der Eingabe.

Wir sind bei der Laufzeit von Insertionsort nicht näher auf die genauen Werte der Konstanten $c_1, \ldots, c_{11}$ eingegangen. Die Anzahl an Prozessorbefehlen, die für die einzelnen Zeilen des Pseudocodes benötigt werden, hängt genauso wie die tatsächliche Laufzeit vom Prozessor, vom Compiler und von weiteren Faktoren ab, die wir in einer theoretischen Analyse nicht genau fassen können und wollen. Wir sind hauptsächlich an der Größenordnung des Wachstums interessiert. Dies werden wir in Abschnitt 1.4 ausführlicher diskutieren. Die Quintessenz der obigen Analyse ist, dass die Worst-Case-Laufzeit von Insertionsort quadratisch ist.

1.3 Registermaschinen

Wir haben oben behauptet, dass jede Zeile des Pseudocodes von Insertionsort nur konstant viele Prozessorbefehle benötigt. Dies erscheint einleuchtend, wenn man an Prozessoren denkt, die in realen Rechnern benutzt werden. Bevor wir Algorithmen entwerfen und analysieren können, müssen wir jedoch zunächst das *Rechnermodell* angeben, auf das wir uns beziehen. Ein solches Modell legt fest, welche Operationen ein Algorithmus ausführen darf und welche Ressourcen (Rechenzeit, Speicherplatz, etc.) er dafür benötigt. Dies ist notwendig, da Aussagen, die sich auf irgendeinen konkreten Rechner beziehen, nicht allgemein genug sind und schnell an Bedeutung verlieren, wenn sich die verfügbare Hardware ändert. Wir betrachten deshalb mathematische Modelle, die nur die wesentlichen Aspekte realer Rechner abbilden und auf diese Weise von der konkreten Hardware abstrahieren.

Die Auswahl eines geeigneten Rechnermodells ist stets eine Gratwanderung. Auf der einen Seite sollte das Modell möglichst einfach sein, um theoretische Analysen zu

ermöglichen, auf der anderen Seite sollte es alle wesentlichen Aspekte realer Rechner abbilden, da ansonsten Aussagen, die in dem Modell zum Beispiel über die Laufzeit eines Algorithmus getroffen werden, nicht mit der Realität übereinstimmen.

Einen für viele Zwecke guten Kompromiss stellt das Modell der *Registermaschine* dar, das an eine rudimentäre Assemblersprache erinnert, die auf die wesentlichen Befehle reduziert wurde. In diesem Modell steht als Speicher eine unbegrenzte Anzahl an Registern zur Verfügung, die jeweils eine natürliche Zahl enthalten und auf denen grundlegende arithmetische Operationen durchgeführt werden können. Die Inhalte zweier Register können addiert, subtrahiert, multipliziert und dividiert werden. Ebenso können Registerinhalte kopiert werden und Register können mit Konstanten belegt werden. Darüber hinaus unterstützen Registermaschinen unbedingte Sprungoperationen (GOTO) und bedingte Sprungoperationen (IF), wobei als Bedingung nur getestet werden darf, ob ein Register kleiner, gleich oder größer als ein anderes Register ist. Jede der genannten Operationen benötigt eine Zeiteinheit zur Ausführung.

Wir verzichten auf eine formale Beschreibung des Modells der Registermaschine, da diese zum jetzigen Zeitpunkt wenig erhellend wäre. Leserinnen und Leser mit etwas Programmiererfahrung wird es nicht verwundern, dass man trotz des eingeschränkten Befehlssatzes mit einer Registermaschine alle Berechnungen durchführen kann, die man auch mit einem realen Rechner durchführen kann. Beispielsweise kann der oben besprochene Algorithmus Insertionsort mit etwas Aufwand auch als Registermaschinenprogramm formuliert werden. Natürlich werden wir die Algorithmen in dieser Vorlesung nicht als Registermaschinenprogramme angeben, da dies sehr umständlich wäre. Man sollte dieses Modell lediglich in Erinnerung behalten, wenn man die Laufzeit eines Algorithmus analysiert.

In einer Registermaschine kann jedes Register eine beliebig große natürliche Zahl speichern und unabhängig von der Größe der Zahlen können arithmetische Operationen auf den Registern in einer Zeiteinheit durchgeführt werden. Aus diesem Grund wird dieses Modell auch das *uniforme Kostenmodell* genannt. Dies ist nur für solche Algorithmen ein realistisches Modell, die ausschließlich kleine Zahlen verwenden, die auch in realen Prozessoren in ein Register passen. Für kryptographische Algorithmen, die arithmetische Operationen auf Zahlen mit mehreren Tausend Bits ausführen, ist es jedoch beispielsweise nicht realistisch davon auszugehen, dass jede Operation in einer Zeiteinheit realisiert werden kann. Für solche Anwendungen gibt es das *nicht uniforme Kostenmodell*, bei dem die Laufzeit der arithmetischen Operationen logarithmisch von der Größe der Zahlen abhängt. Auf dieses Modell gehen wir in dieser Vorlesung jedoch nicht weiter ein, da wir nur Probleme und Algorithmen mit kleinen Zahlen betrachten werden.

Ist das Modell der Registermaschine ein gutes Modell, um Algorithmen zu studieren? Geht es uns nur um die Frage, ob ein Problem generell mithilfe eines Rechners gelöst werden kann, so können wir diese Frage eindeutig mit ja beantworten. Registermaschinen können genau dieselben Probleme lösen wie reale Rechner. Geht es uns um die Frage, welche Probleme effizient gelöst werden können, so fällt die Antwort differenzierter aus. Grundsätzlich können alle Prozessorbefehle, die ein gängiger realer Rechner durchführen kann, durch konstant viele Befehle einer Registermaschine simuliert werden und umgekehrt. Das bedeutet, dass beispielsweise Insertionsort sowohl

auf einer Registermaschine als auch auf einem realen Rechner quadratische Laufzeit besitzt, wenngleich mit anderen konstanten Faktoren. In diesem Sinne ist das Modell der Registermaschine also ein realistisches Modell, um die Laufzeit von Algorithmen zu beurteilen. Dies hat dazu geführt, dass Registermaschinen zum Standard in der Algorithmik geworden sind.

Natürlich gibt es Aspekte realer Hardware, die nicht im Modell der Registermaschine abgebildet sind. Arbeitet man beispielsweise mit großen Datenmengen, die nicht komplett in den Hauptspeicher passen, so spielt es in der Praxis eine große Rolle, welche Daten auf die Festplatte oder SSD ausgelagert werden, da Festplattenzugriffe und auch Zugriffe auf SSDs sehr viel langsamer sind als Zugriffe auf den Hauptspeicher. Dies ist im Modell der Registermaschine nicht erfasst. Außerdem führen Registermaschinen per Definition sequentielle Programme aus. Nebenläufige Programmierung für Prozessoren mit mehreren Kernen ist nicht erfasst. Dennoch ist das Modell der Registermaschine für die allermeisten Anwendungen ein realistisches Modell. Man sollte aber die Limitierungen kennen, um es sinnvoll einsetzen zu können.

1.4 Größenordnungen

Wir haben bei der Analyse von Insertionsort bereits erwähnt, dass wir an den genauen Konstanten in der Laufzeit nicht interessiert sind, da sie vom Compiler und der konkreten Hardware abhängen. Ebenso spielt für uns meistens nur der Term in der Laufzeit eine Rolle, der am stärksten wächst, da die anderen Terme für große Eingaben an Bedeutung verlieren. Die Laufzeit von Insertionsort haben wir für geeignete Konstanten $a, b, c \in \mathbb{R}_{\geq 0}$ durch $an^2 + bn + c$ nach oben beschränkt. Für große n wird diese Laufzeit durch den quadratischen Term dominiert und dementsprechend hatten wir vereinfacht gesagt, dass die Worst-Case-Laufzeit von Insertionsort quadratisch in der Anzahl der zu sortierenden Zahlen wächst. In diesem Abschnitt lernen wir mit der *O-Notation* ein mathematisches Hilfsmittel kennen, um solche Sachverhalte zu formalisieren.

Definition 1.2. *Es seien $f, g\colon \mathbb{N} \to \mathbb{R}_{>0}$ zwei Funktionen.*

a) Wir sagen f wächst nicht schneller als *g und schreiben $f = O(g)$, wenn*

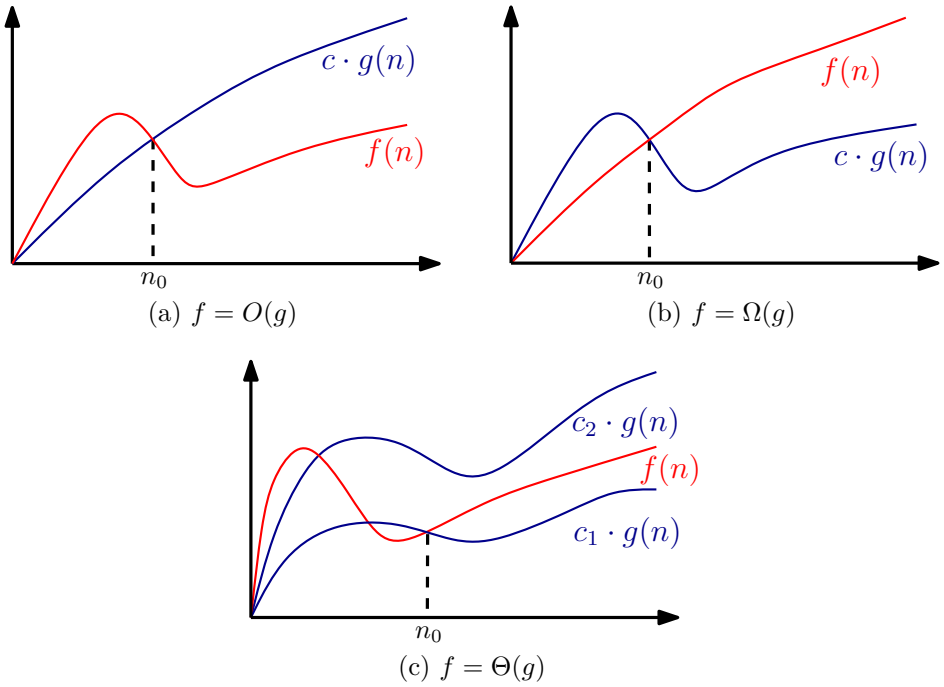
$$\exists c \in \mathbb{R} : \exists n_0 \in \mathbb{N} : \forall n \geq n_0 \colon f(n) \leq c \cdot g(n).$$

b) Wir sagen f wächst mindestens so schnell wie *g und schreiben $f = \Omega(g)$, wenn $g = O(f)$ gilt.*

c) Wir sagen f und *g* sind in der gleichen Größenordnung *und schreiben $f = \Theta(g)$, wenn $f = O(g)$ und $f = \Omega(g)$ gilt.*

Anschaulich bedeutet $f = O(g)$, dass die Funktion f ab einer bestimmten Konstante n_0 nur um maximal einen konstanten Faktor größer ist als die Funktion g. In Abbildung 1.1 ist die Definition noch einmal veranschaulicht.

Die in Definition 1.2 eingeführte Notation ist streng genommen nicht ganz korrekt, aber dennoch weit verbreitet. Für eine Funktion $g\colon \mathbb{N} \to \mathbb{R}_{>0}$ bezeichnet $O(g)$ formal korrekt



(a) $f = O(g)$ (b) $f = \Omega(g)$ (c) $f = \Theta(g)$

Abbildung 1.1: Illustration von Definition 1.2

eigentlich eine Menge von Funktionen, nämlich die Menge der Funktionen $f : \mathbb{N} \to \mathbb{R}_{>0}$, die die Bedingung aus Definition 1.2 a) erfüllen. Demzufolge wäre die formal korrekte Notation $f \in O(g)$ anstatt $f = O(g)$. Ebenso verhält es sich mit Ω und Θ. Da es sich bei $O(g)$ um eine Menge handelt, benutzen wir manchmal Formulierungen wie „die Laufzeit des Algorithmus *liegt in* $O(g)$“.

Wir haben oben gesehen, dass die Laufzeit von InsertionSort für das Sortieren von n Zahlen für geeignete Konstanten $a, b, c \in \mathbb{R}_{\geq 0}$ durch $an^2 + bn + c$ nach oben beschränkt ist. Für jedes $n \in \mathbb{N}$ gilt $an^2 + bn + c \leq (a+b+c)n^2$. Wir können also gemäß der obigen Definition einfach sagen, dass die Laufzeit von InsertionSort in $O(n^2)$ liegt. Da die obige Ungleichung für alle $n \in \mathbb{N}$ gilt, können wir $n_0 = 1$ setzen. Im Allgemeinen dient das n_0 dazu, kleine Werte von n auszublenden, da uns die Laufzeit in erster Linie für große Eingaben interessiert. Wir haben auch gesehen, dass es Eingaben gibt (solche, in denen die Zahlen in der Eingabe absteigend sortiert sind), für die die Laufzeit mindestens an^2 für eine geeignete Konstante $a \in \mathbb{R}_{>0}$ beträgt. Somit können wir auch sagen, dass die Worst-Case-Laufzeit von InsertionSort in $\Omega(n^2)$ und damit insgesamt in $\Theta(n^2)$ liegt. Wir haben ferner gesehen, dass InsertionSort für jede Eingabe mindestens n Schritte benötigt. Somit liegt die Laufzeit von InsertionSort für jede Eingabe in $\Omega(n)$.

Im Folgenden werden wir bei der Analyse von Laufzeiten stets die O-Notation einsetzen. Genauso wie beim Modell der Registermaschine muss man sich auch bei der O-Notation fragen, ob das richtige Abstraktionsniveau getroffen wurde. Ein Algorithmus, der $10^{100}n$ Schritte benötigt, ist für praktische Zwecke natürlich nicht besser als ein Algorithmus, der $2n^2$ Schritte benötigt, obwohl er eine lineare Laufzeit besitzt. Dennoch hat sich die O-Notation als Hilfsmittel in der Algorithmik durchgesetzt. Der

Grund dafür ist, dass bei den allermeisten Algorithmen keine so großen Konstanten auftreten und die Größenordnung des Wachstums tatsächlich die wichtigere Eigenschaft ist. Man sollte jedoch in Erinnerung behalten, dass die O-Notation eine zu starke Abstraktion darstellen kann, da es in praktischen Anwendungen oft durchaus eine Rolle spielt, die Konstanten in der Laufzeit zu optimieren. Ebenso ist die Fokussierung auf große Eingaben durch die Wahl von n_0 nicht immer gerechtfertigt. Zwar werden die zu verarbeitenden Datenmengen in vielen Anwendungen immer größer, aber auch hier gilt, dass für sehr große n_0 die praktische Relevanz schwindet. Generell hat sich die O-Notation aber sehr gut in der Informatik bewährt, um eine erste Einschätzung über die Effizienz eines Algorithmus zu erhalten.

Beispiele für die Verwendung der O-Notation

- Für jedes Polynom $p(n) = \sum_{i=0}^{d} c_i n^i$ mit $c_d > 0$ kann man analog zur Laufzeitabschätzung von INSERTIONSORT argumentieren, dass $p = \Theta(n^d)$ gilt.
- Für beliebige Konstanten $a > 1$ und $b > 1$ gilt $\log_a(n) = \Theta(\log_b(n))$, da $\log_a(n) = \log_b(n)/\log_b(a)$. Die beiden Funktionen unterscheiden sich also nur um den Konstanten Faktor $\log_b(a)$. Aus diesem Grund lassen wir in der O-Notation die Basis von Logarithmen weg (sofern sie konstant ist) und schreiben beispielsweise $O(n \log n)$ statt $O(n \log_2 n)$.
- Für beliebige Konstanten $a > 0$ und $b > 1$ gilt $\log_b(n^a) = \Theta(\log n)$ wegen $\log_b(n^a) = a \log_b n$.
- Gilt $f = O(1)$ für eine Funktion f, so folgt aus Definition 1.2, dass f für alle n durch eine Konstante nach oben beschränkt ist. Benötigt ein Algorithmus oder ein Teil eines Algorithmus eine konstante Laufzeit, die unabhängig von der Eingabe ist, so sagen wir deshalb auch, dass die Laufzeit in $O(1)$ liegt.

Der aufmerksamen Leserin und dem aufmerksamen Leser mag im ersten Beispiel aufgefallen sein, dass Definition 1.2 streng genommen nicht für jedes Polynom $p(n) = \sum_{i=0}^{d} c_i n^i$ mit $c_d > 0$ angewendet werden kann, da sie sich nur auf Funktionen bezieht, die ausschließlich positive Werte annehmen. Dies ist aber beispielsweise für das Polynom $p(n) = n^2 - 10n + 1$ nicht der Fall. Ebenso ist der Logarithmus im zweiten Beispiel problematisch, da er für $n = 1$ den Wert 0 annimmt. Diesen Funktionen ist jedoch gemein, dass es ein $n_0 \in \mathbb{N}$ gibt, sodass sie für alle $n \geq n_0$ positiv sind. Eine solche Funktion nennt man *asymptotisch positiv*. Um die O-Notation ohne großen Aufwand auf Polynome, Logarithmen und ähnliche Funktionen anwenden zu können, erweitern wir Definition 1.2 auf asymptotisch positive Funktionen.

Als Ergänzung zu Definition 1.2 führen wir nun noch die *o-Notation* ein, mit der man ausdrücken kann, dass eine Funktion echt langsamer wächst als eine andere.

Definition 1.3. *Es seien $f, g\colon \mathbb{N} \to \mathbb{R}_{>0}$ zwei Funktionen.*

a) Wir sagen f wächst langsamer als *g und schreiben $f = o(g)$, wenn*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

b) Wir sagen f wächst schneller als g *und schreiben* $f = \omega(g)$*, wenn* $g = o(f)$ *gilt.*

Auch Definition 1.3 kann problemlos auf asymptotisch positive Funktionen erweitert werden. Die Beweise für die Aussagen in den folgenden Beispielen folgen aus elementaren Argumenten der Analysis und sind der Leserin und dem Leser als Übung überlassen.

Beispiele für die Verwendung der o-Notation

- Es seien $p(n) = \sum_{i=0}^{d_1} c_i n^i$ mit $c_{d_1} > 0$ und $q(n) = \sum_{i=0}^{d_2} c_i n^i$ mit $c_{d_2} > 0$ zwei Polynome.
  - Für $d_1 < d_2$ gilt $p = o(q)$.
  - Für $d_1 > d_2$ gilt $p = \omega(q)$.
  - Für $d_1 = d_2$ gilt $p = \Theta(q)$.
- Für jedes $k > 0$ und jedes $\varepsilon > 0$ gilt $(\log_2(n))^k = o(n^\varepsilon)$.
- Für jedes $k > 0$ und jedes $a > 1$ gilt $n^k = o(a^n)$.
- Für jedes $a > 1$ und jedes $b > a$ gilt $a^n = o(b^n)$.

Kapitel 2

Methoden zum Entwurf von Algorithmen

Damit man nicht bei jedem Problem von neu beginnen und ad hoc nach einem möglichst guten Algorithmus suchen muss, gibt es eine Reihe von allgemeinen Methoden zum Entwurf von Algorithmen, mit deren Hilfe man für viele verschiedene Probleme effiziente Algorithmen konstruieren kann. Wir lernen in diesem Kapitel drei wichtige Methoden kennen.

2.1 Divide-and-Conquer

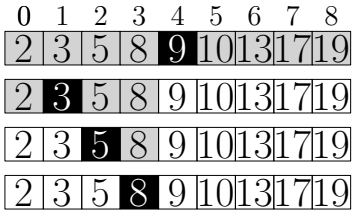
Divide-and-Conquer[1] ist für viele Probleme ein naheliegender Ansatz. Um eine gegebene Instanz zu lösen, teilt man diese zunächst in kleinere Teilinstanzen, löst diese und kombiniert anschließend die Lösungen der Teilinstanzen zu einer Lösung der gegebenen Instanz. Die Lösung der Teilinstanzen erfolgt dabei rekursiv. Das bedeutet, solange die Teilinstanzen noch eine gewisse Mindestgröße besitzen, werden sie nach dem gleichen Schema weiter in kleinere Instanzen zerlegt. Instanzen, die kleiner als die Mindestgröße sind, besitzen meistens entweder eine triviale Lösung oder sie können mit einem einfachen Algorithmus schnell gelöst werden.

Ein Algorithmus, der als einfacher Spezialfall von Divide-and-Conquer angesehen werden kann, ist *binäre Suche*. Dieser Algorithmus bestimmt für ein Feld $a[0 \ldots n-1]$ von aufsteigend sortierten Zahlen und eine Zahl x, ob x in a enthalten ist oder nicht. Binäre Suche testet zunächst, ob das mittlere Element des Feldes $a[m]$ für $m = \lfloor (n-1)/2 \rfloor$ kleiner als x ist. Ist dies der Fall, so kann sich x nicht in der linken Hälfte des Feldes befinden. Ist das mittlere Element größer als x, so kann sich x nicht in der rechten Hälfte des Feldes befinden. Der Teil des Feldes, in dem x noch gesucht werden muss, halbiert sich somit in beiden Fällen. Dieser Teil kann rekursiv nach demselben Prinzip durchsucht werden. Die Rekursion stoppt, wenn $a[m] = x$ gilt oder der noch zu durchsuchende Bereich leer ist. In letzterem Fall ist x nicht in a enthalten.

[1] Auf Deutsch wird Divide-and-Conquer auch als *teile und herrsche* bezeichnet.

Binäre Suche ist unten in Pseudocode mit einem Beispiel dargestellt. Die Variablen ℓ und r beschreiben den Bereich $a[\ell \dots r]$ des Feldes, in dem noch nach x gesucht werden muss. Der initiale Aufruf ist dementsprechend BinarySearch$(a, x, 0, n-1)$.

```
BinarySearch(int[] a, int x, int ℓ, int r)
1   if (ℓ > r) return false;
2   int m = ℓ + (r − ℓ)/2;
3   if (a[m] < x)
4       return BinarySearch(a,x,m + 1,r);
5   else if (a[m] > x)
6       return BinarySearch(a,x,ℓ,m − 1);
7   return true;
```



Suche nach $x = 8$

Zum Pseudocode sei noch angemerkt, dass wir davon ausgehen, dass bei Divisionen von ganzen Zahlen wie in Java die Nachkommastellen ignoriert werden. In Zeile 2 wird $(r - \ell)/2$ dementsprechend abgerundet.

Bei binärer Suche handelt es sich um einen einfachen Spezialfall von Divide-and-Conquer, bei dem die zu lösende Instanz lediglich in eine Teilinstanz mit einem halb so großen zu durchsuchenden Bereich zerlegt wird. Wir verzichten hier auf einen formalen Beweis der Korrektheit von binärer Suche, der mithilfe der Invariante geführt werden kann, dass sich das zu suchende Element x in jedem rekursiven Aufruf in dem aktiven Bereich $a[\ell \dots r]$ befindet, sofern es überhaupt in a enthalten ist. Viel mehr interessiert uns die Laufzeit von binärer Suche.

Theorem 2.1. *Die Laufzeit von binärer Suche beträgt* $O(\log n)$.

Beweis. Die wesentliche Beobachtung ist, dass der Bereich, der noch durchsucht werden muss, mit jedem rekursiven Aufruf im Wesentlichen halbiert wird. Betrachten wir einen beliebigen Aufruf von BinarySearch und sei $z = r - \ell + 1 > 0$ die Länge des noch zu durchsuchenden Bereichs. Ist z ungerade, so erfolgt der nächste rekursive Aufruf mit einem Bereich der Länge $(z-1)/2$. Ist z gerade, so erfolgt der nächste rekursive Aufruf mit einem Bereich der Länge $z/2 - 1$ oder $z/2$ je nachdem, ob im linken oder rechten Teil weitergesucht wird. In jedem Fall umfasst der neue Bereich höchstens $z/2$ Positionen.

Aus dieser Beobachtung lässt sich leicht ableiten, dass nach $k \in \mathbb{N}$ rekursiven Aufrufen nur noch ein Bereich der Länge höchstens $n/2^k$ durchsucht werden muss. Betrachten wir den vorletzten rekursiven Aufruf, dann gilt $n/2^k \geq 1$. Dies lässt sich umformen zu $k \leq \log_2(n)$. Da k stets ganzzahlig ist, folgt daraus $k \leq \lfloor \log_2(n) \rfloor$. Setzen wir $k = K := \lfloor \log_2(n) \rfloor + 1$, dann folgt daraus $n/2^k < 1$. Da auch die Länge des noch zu durchsuchenden Bereichs stets ganzzahlig ist, muss dieser Bereich demnach nach spätestens K rekursiven Aufrufen auf die Länge 0 zusammengeschrumpft sein. Dies impliziert, dass die Abbruchbedingung $\ell > r$ greift und kein weiterer rekursiver Aufruf erfolgt.

Dies zeigt, dass unabhängig von a und x maximal $K = O(\log n)$ rekursive Aufrufe erfolgen. Abgesehen von den rekursiven Aufrufen benötigt ein Aufruf von BINARY-SEARCH konstante Zeit. Außerdem erfolgt in jedem Aufruf maximal ein rekursiver Aufruf. Dies alles zusammen impliziert das Theorem. □

Dass die Analyse der Laufzeit von binärer Suche relativ einfach ist, ist der Tatsache geschuldet, dass in jedem Aufruf maximal ein rekursiver Aufruf erfolgt. Dies ist bei Divide-and-Conquer jedoch eher die Ausnahme. In den folgenden Abschnitten werden wir zwei Algorithmen kennenlernen, die mehrere rekursive Aufrufe erzeugen, und eine allgemeine Methode, um solchen Algorithmen zu analysieren.

2.1.1 Mergesort

Bei MERGESORT handelt es sich um einen weiteren Sortieralgorithmus. Genauso wie INSERTIONSORT erhält er als Eingabe ein Feld $a[0 \dots n-1]$ von ganzen Zahlen, das aufsteigend sortiert werden soll. Dazu zerlegt er das Feld in zwei Hälften, sortiert diese unabhängig voneinander und verschmilzt anschließend die sortierten Hälften zu einem insgesamt sortierten Feld. Das Sortieren der Hälften erfolgt wieder rekursiv, solange sie noch mindestens zwei Elemente enthalten.

Im Gegensatz zu binärer Suche erfolgen bei MERGESORT somit bei jedem Feld mit mindestens zwei Elementen zwei rekursive Aufrufe. Wir werden unten sehen, dass das Verschmelzen zweier sortierter Teilfelder zu einem insgesamt sortierten Feld in linearer Zeit durchgeführt werden kann. Im folgenden Pseudocode erhält die Methode MERGESORT ein Feld $a[0 \dots n-1]$ und zwei Indizes ℓ und r als Eingabe. Es wird dann der Bereich $a[\ell \dots r]$ des Feldes aufsteigend sortiert. Dementsprechend ist der initiale Aufruf MERGESORT$(a, 0, n-1)$. Ist $\ell \geq r$, so besteht der zu sortierende Bereich aus höchstens einem Element. In diesem Fall ist also nichts zu tun und die letzte Rekursionsebene ist erreicht.

Die Methode MERGE, die wir unten angeben werden, erhält als Eingabe ein Feld a und drei Indizes ℓ, m und r. Beim Aufruf dieser Methode sind die Teilfelder $a[\ell \dots m]$ und $a[m+1 \dots r]$ aufsteigend sortiert. Nach dem Aufruf der Methode ist das Teilfeld $a[\ell \dots r]$ insgesamt aufsteigend sortiert. Wir gehen im Pseudocode davon aus, dass bei den Aufrufen der Methoden MERGESORT und MERGE wie in Java Referenzen auf das Feld a übergeben werden. Änderungen, die innerhalb dieser Methoden an dem Feld vorgenommen werden, bleiben also auch nach dem Verlassen der Methoden erhalten.

```
MERGESORT(int[] a, int ℓ, int r)
1   if (ℓ < r) {
2       int m = ℓ + (r − ℓ)/2;
3       MERGESORT(a,ℓ,m);
4       MERGESORT(a,m + 1,r);
5       MERGE(a,ℓ,m,r);
6   }
```

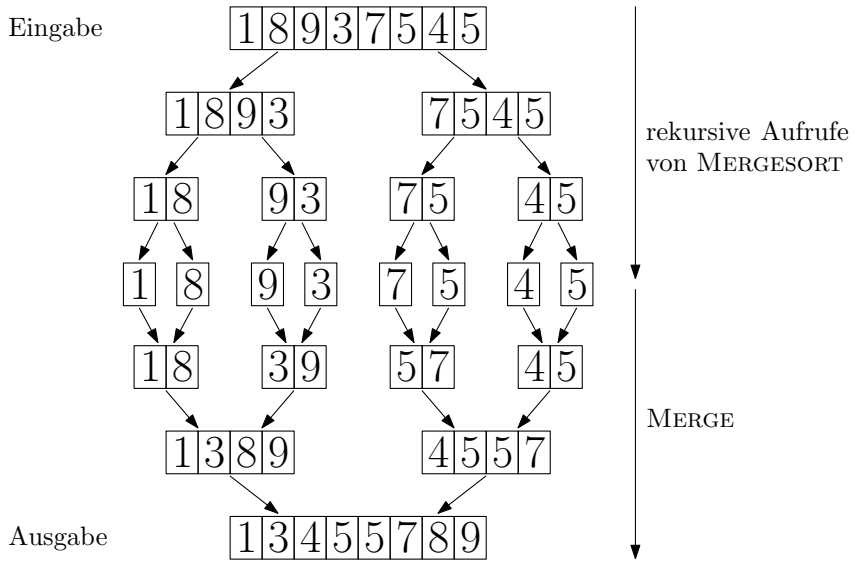


Abbildung 2.1: Illustration von MERGESORT

Der Algorithmus MERGESORT ist in Abbildung 2.1 an einem Beispiel veranschaulicht.

Die Methode MERGE kann wieder mit Spielkarten illustriert werden. Angenommen wir haben zwei sortierte Stapel von Spielkarten, die wir zu einem insgesamt sortierten Stapel verschmelzen wollen. Dazu vergleichen wir die beiden obersten Karten auf den Stapeln und legen die kleinere von beiden auf einen neuen Stapel. Anschließend vergleichen wir wieder die beiden obersten Karten auf den Stapeln und fügen die kleinere von beiden dem neu angelegten Stapel hinzu. Nach diesem Prinzip gehen wir vor, bis einer der beiden Stapel leer ist. Dann müssen nur noch die restlichen Karten des anderen Stapels dem neu angelegten Stapel hinzugefügt werden, der am Ende alle Karten in sortierter Reihenfolge enthält. Eine mögliche Implementierung dieses Verfahrens ist im Folgenden in Pseudocode dargestellt.

```
MERGE(int[] a, int ℓ, int m, int r)
 1   // Kopiere die beiden sortierten Teilfelder in left und right.
 2   int[] left = new int[m − ℓ + 1]; int[] right = new int[r − m];
 3   for (int i = 0; i < m − ℓ + 1; i++) { left[i] = a[ℓ + i]; }
 4   for (int i = 0; i < r − m; i++) { right[i] = a[m + 1 + i]; }
 5   // Solange beide Teilfelder noch Elemente enthalten, füge sie in a ein.
 6   int iL = 0; int iR = 0; int iA = ℓ;
 7   while ((iL < m − ℓ + 1) && (iR < r − m)) {
 8       if (left[iL] <= right[iR]) { a[iA] = left[iL]; iA++; iL++; }
 9       else { a[iA] = right[iR]; iA++; iR++; }
10   }
11   // Füge die übrig gebliebenen Elemente eines Teilfeldes in a ein.
12   while (iL < m − ℓ + 1) { a[iA] = left[iL]; iA++; iL++; }
13   while (iR < r − m) { a[iA] = right[iR]; iA++; iR++; }
```

Die Korrektheit von MERGE kann mithilfe der folgenden Invariante gezeigt werden. Nach jedem Durchlauf einer der while-Schleifen in den Zeilen 7, 12 und 13 enthält das Teilfeld $a[\ell \dots iA-1]$ die $iA-\ell$ kleinsten Elemente der Felder *left* und *right* in sortierter Reihenfolge. Ferner sind $left[iL]$ und $right[iR]$ die kleinsten Elemente aus den Feldern *left* und *right*, die sich nicht in dem Teilfeld $a[\ell \dots iA-1]$ befinden. Die Korrektheit von MERGESORT folgt direkt aus der Korrektheit von MERGE. Wir gehen darauf jedoch nicht weiter ein, da uns in diesem Abschnitt hauptsächlich die Laufzeit interessiert und die Korrektheit von MERGESORT der Leserin und dem Leser hoffentlich intuitiv klar ist.

Zur Analyse der Laufzeit gehen wir davon aus, dass $n = 2^k$ für ein $k \in \mathbb{N}$ gilt. Dies erleichtert die Analyse, da das Feld dann auf jeder Rekursionsebene in zwei gleich große Hälften geteilt werden kann und wir in den Rechnungen nicht auf- und abrunden müssen. Wir werden später sehen, dass die Einschränkung auf Zweierpotenzen asymptotisch keine Rolle spielt.

Wegen der rekursiven Struktur von Algorithmen, die auf Divide-and-Conquer basieren, ist es naheliegend auch die Laufzeit rekursiv zu analysieren. Sei $T(n)$ die Worst-Case-Laufzeit von MERGESORT auf Eingaben, die aus n Zahlen bestehen. Um $T(n)$ abzuschätzen, betrachten wir zunächst die Laufzeit von MERGESORT, ohne die beiden rekursiven Aufrufe in den Zeilen 3 und 4 zu berücksichtigen. Die Zeilen 1 und 2 können in konstanter Zeit ausgeführt werden. Die Laufzeit für MERGE in Zeile 5 liegt in $O(n)$. Um dies einzusehen, betrachten wir MERGE genauer. In den Zeilen 2 bis 4 werden insgesamt $r-\ell+1$ Einträge des Feldes a kopiert. Dies benötigt eine Laufzeit von $O(r-\ell)$. In jeder Iteration der while-Schleife in Zeile 7 wird entweder der Zähler iL oder der Zähler iR um eins erhöht. Nach spätestens $m-\ell+r-m = r-\ell = O(r-\ell)$ Schritten ist somit die Abbruchbedingung erfüllt. Da in jeder Iteration der while-Schleife nur konstant viele Operationen ausgeführt werden, benötigt die while-Schleife in Zeile 7 somit insgesamt nur eine Laufzeit von $O(r-\ell)$. Analog kann man für die while-Schleifen in den Zeilen 12 und 13 argumentieren. Da MERGE im initialen Aufruf von MERGESORT mit den Parametern $\ell = 0$ und $r = n-1$ aufgerufen wird, ergibt sich die gewünschte Laufzeit von $O(n)$.

Die beiden rekursiven Aufrufe erfolgen mit Teilfeldern der halben Länge. Sie besitzen somit per Definition jeweils eine Worst-Case-Laufzeit von $T(n/2)$. Damit können wir insgesamt festhalten, dass es eine Konstante $c \in \mathbb{R}_{>0}$ gibt, sodass

$$T(n) \leq 2 \cdot T(n/2) + cn \tag{2.1}$$

für jedes $n = 2^k$ mit $k \in \mathbb{N}$ gilt. Ferner gilt $T(1) \leq c$, da im Falle eines Feldes der Länge 1 keine rekursiven Aufrufe erfolgen.

Um die Worst-Case-Laufzeit von MERGESORT explizit nach oben abzuschätzen, müssen wir eine Funktion T finden, die die obigen Ungleichungen erfüllt. Hat man bereits einen Verdacht, wie diese Funktion aussieht, so kann man diesen mit vollständiger Induktion überprüfen, wie der Beweis des folgenden Theorems zeigt.

Theorem 2.2. *Die Laufzeit von* MERGESORT *liegt in* $O(n \log n)$.

Beweis. Wir müssen zeigen, dass es Konstanten $n_0 \in \mathbb{N}$ und $c^\star$ gibt, sodass

$$T(n) \leq c^\star \cdot n \log_2(n) \tag{2.2}$$

für alle $n \geq n_0$ gilt, wobei die Wahl der Basis des Logarithmus willkürlich ist, da sie in der O-Notation keine Rolle spielt. Wie oben erwähnt, werden wir nur den Fall betrachten, dass n eine Zweierpotenz ist.

Zunächst beobachten wir, dass die Ungleichung (2.2) für $n = 1$ nicht gilt, da die rechte Seite für $n = 1$ den Wert 0 annimmt. Um diesen Fall auszublenden, setzen wir $n_0 = 2$. Für den Induktionsanfang betrachten wir dementsprechend den Fall $n = 2$. Wird MERGESORT mit einem Feld der Länge zwei aufgerufen, so erfolgen zwei rekursive Aufrufe mit Feldern der Länge eins, die sofort terminieren. Insgesamt kann die Laufzeit für Felder der Länge zwei somit durch eine Konstante c' abgeschätzt werden. Der Induktionsanfang gilt demnach für jede Konstante $c^\star \geq c'$.

Sei nun $n = 2^k$ für ein $k \in \{2, 3, 4, \ldots\}$. Gemäß Ungleichung (2.1) und der Induktionsannahme $T(n/2) \leq c^\star \cdot (n/2) \log_2(n/2)$ gilt

$$\begin{aligned} T(n) &\leq 2 \cdot T(n/2) + cn \\ &\leq 2 \cdot (c^\star \cdot (n/2) \cdot \log_2(n/2)) + cn \\ &= c^\star \cdot n \log_2(n/2) + cn \\ &= c^\star \cdot n(\log_2(n) - 1) + cn \\ &= c^\star \cdot n \log_2(n) - c^\star n + cn. \end{aligned}$$

Für jede Konstante $c^\star \geq c$ folgt, wie gewünscht, $T(n) \leq c^\star \cdot n \log_2(n)$. Insgesamt folgt das Theorem somit für die Konstante $c^\star = \max\{c, c'\}$. □

Vorsicht bei Verwendung der O-Notation

Wir haben im vorangegangenen Beweis mit Konstanten gerechnet und außer in der Formulierung des Theorems keinen Gebrauch von der O-Notation gemacht. Warum dies notwendig ist, illustrieren wir mit dem folgenden falschen Beweis dafür, dass $T(n) = O(n)$ gilt. Wenn wir dies mit vollständiger Induktion beweisen wollen, ist der Induktionsanfang unproblematisch. Wir betrachten deshalb nur den Induktionsschritt. Für $n = 2^k$ mit $k \in \{2, 3, 4, \ldots\}$ gilt gemäß Ungleichung (2.1) und der Induktionsannahme $T(n/2) \leq O(n/2)$

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn \\ &= O(n/2) + O(n) \\ &= O(n) + O(n) \\ &= O(n). \end{aligned}$$

Die obige Rechnung ist korrekt, eignet sich aber nicht als Induktionsschritt, da die Konstanten, die sich hinter der O-Notation in der Induktionsannahme $T(n/2) \leq O(n/2)$ und der O-Notation im letzten Schritt der Rechnung $T(n) = O(n)$ verbergen, nicht dieselben sind.

Mit der obigen vollständigen Induktion haben wir also lediglich die triviale Aussage

$$\exists n_0 \in \mathbb{N} : \forall n \geq n_0 : \exists c \in \mathbb{R} : T(n) \leq cn$$

gezeigt, in der die Reihenfolge der Quantoren verglichen mit Definition 1.2 vertauscht ist.

In Theorem 2.2 haben wir nur eine obere Schranke für die Laufzeit von MERGESORT hergeleitet. Statt der Ungleichung (2.1) kann man aber auch leicht argumentieren, dass es eine Konstante $c' > 0$ gibt, für die $T(n) \geq 2T(n/2) + c'n$ für alle $n = 2^k$ mit $k \in \mathbb{N}$ gilt. Mit dieser Ungleichung kann man analog nachweisen, dass die Laufzeit von MERGESORT für jede Eingabe auch in $\Omega(n \log n)$ und somit insgesamt in $\Theta(n \log n)$ liegt. Im Gegensatz zu INSERTIONSORT, für dessen Laufzeit die Reihenfolge der Zahlen in der Eingabe eine große Rolle spielt, ist die Laufzeit von MERGESORT für jede Eingabe mit n Zahlen nahezu gleich. Während MERGESORT eine deutlich bessere Worst-Case-Laufzeit besitzt, ist INSERTIONSORT auf Eingaben, die bis auf wenige Einträge sortiert sind, besser.

2.1.2 Strassen-Algorithmus

Ein wichtiges Problem, das in vielen Anwendungen als Teilproblem auftaucht, ist die Berechnung des Produktes zweier Matrizen. Seien $A = (a_{ij})$ und $B = (b_{ij})$ zwei reelle $n \times n$-Matrizen und sei $C = A \cdot B$ das Produkt dieser Matrizen, das ebenfalls eine $n \times n$-Matrix $C = (c_{ij})$ ist. Für jedes $i, j \in \{1, \ldots, n\}$ gilt

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}. \tag{2.3}$$

Der naive Algorithmus zur Berechnung von C, der jeden Eintrag gemäß dieser Formel berechnet, besitzt eine Laufzeit von $\Theta(n^3)$, da zur Berechnung jedes der n^2 Einträge n Multiplikationen und $n - 1$ Additionen benötigt werden.[2]

Volker Strassen hat 1969 den ersten Algorithmus entworfen, der eine asymptotisch bessere Laufzeit besitzt. Zur Vereinfachung der Rechnungen gehen wir auch in diesem Abschnitt wieder davon aus, dass n eine Zweierpotenz ist. Dann können wir die Matrizen A, B und C jeweils in vier $(n/2) \times (n/2)$-Teilmatrizen zerlegen:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Man rechnet leicht nach, dass für diese Matrizen die folgenden Zusammenhänge gelten:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \end{aligned}$$

[2] Es hat sich bei der Multiplikation quadratischer Matrizen so etabliert, die Laufzeit nicht anhand der Größe der Eingabe anzugeben, sondern anhand der Anzahl von Zeilen und Spalten der gegebenen Matrizen.

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}.$$

Daraus ergibt sich ein naheliegender Divide-and-Conquer-Algorithmus SimpleProduct, der die Einträge von C gemäß der obigen Gleichungen bestimmt. Dieser berechnet rekursiv acht Produkte von $(n/2)\times(n/2)$-Matrizen. Anschließend addiert er je zwei dieser Produkte. Diese Additionen können ebenso wie das Erstellen der Teilmatrizen von A und B in Zeit $\Theta(n^2)$ durchgeführt werden. Bezeichnen wir nun mit $T(n)$ die Worst-Case-Laufzeit von SimpleProduct zur Multiplikation zweier $n \times n$-Matrizen, so ergibt sich

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{falls } n = 2^k \text{ für } k \in \mathbb{N}. \end{cases}$$

Wir werden im folgenden Abschnitt ein allgemeines Verfahren zur Lösung von Rekursionsgleichungen kennenlernen, nehmen an dieser Stelle aber schon vorweg, dass die Lösung der obigen Rekursionsgleichung $T(n) = \Theta(n^3)$ ist. Das bedeutet, dass der naheliegende Divide-and-Conquer-Algorithmus nicht besser ist als der naive Algorithmus, der jeden Eintrag gemäß (2.3) berechnet.

Strassen hat beobachtet, dass das Produkt zweier $n \times n$-Matrizen auch mithilfe von sieben statt acht Produkten von $(n/2) \times (n/2)$-Matrizen berechnet werden kann. Wir beschreiben seinen Algorithmus im Folgenden in Pseudocode, der diesmal weniger stark an eine konkrete Implementierung in Java angelehnt ist als bei Insertionsort und Mergesort.

Strassen(A,B)

1 Falls $n = 1$ gilt, berechne $A \cdot B$ mit einer Multiplikation und gib das Ergebnis zurück.

2 Ansonsten zerlege A und B wie oben beschrieben in jeweils vier $(n/2) \times (n/2)$-Matrizen A_{ij} und B_{ij}.

3 Berechne die folgenden zehn $(n/2) \times (n/2)$-Matrizen:

$$\begin{aligned} S_1 &= B_{12} - B_{22}, & S_2 &= A_{11} + A_{12}, & S_3 &= A_{21} + A_{22}, \\ S_4 &= B_{21} - B_{11}, & S_5 &= A_{11} + A_{22}, & S_6 &= B_{11} + B_{22}, \\ S_7 &= A_{12} - A_{22}, & S_8 &= B_{21} + B_{22}, & S_9 &= A_{11} - A_{21}, \\ S_{10} &= B_{11} + B_{12}. \end{aligned}$$

4 Berechne rekursiv die folgenden Produkte von $(n/2) \times (n/2)$-Matrizen:

$$\begin{aligned} P_1 &= A_{11} \cdot S_1, & P_2 &= S_2 \cdot B_{22}, & P_3 &= S_3 \cdot B_{11}, \\ P_4 &= A_{22} \cdot S_4, & P_5 &= S_5 \cdot S_6, & P_6 &= S_7 \cdot S_8, \\ P_7 &= S_9 \cdot S_{10}. \end{aligned}$$

5 Berechne die Teilmatrizen von C wie folgt:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6, & C_{12} &= P_1 + P_2, \\ C_{21} &= P_3 + P_4, & C_{22} &= P_5 + P_1 - P_3 - P_7. \end{aligned}$$

Es ist alles andere als offensichtlich, wie die Formeln im Strassen-Algorithmus zustande komme. Sind sie allerdings einmal beschrieben, so kann die Korrektheit des Algorithmus durch einfaches Nachrechnen gezeigt werden, auf das wir hier verzichten, da es wenig erhellend ist und wir uns hauptsächlich für die Laufzeit interessieren. Diese können wir wieder einfach rekursiv beschreiben:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{falls } n = 2^k \text{ für } k \in \mathbb{N}. \end{cases}$$

Wir werden im folgenden Abschnitt sehen, dass $T(n) = \Theta(n^{\log_2 7}) = O(n^{2,81})$ die Lösung dieser Rekursionsgleichung ist. Dadurch, dass nur sieben statt acht rekursive Aufrufe erfolgen, ergibt sich also eine essentielle Verbesserung der Laufzeit. Diese geht allerdings auf Kosten der Konstante in der O-Notation, da mehr Additionen und Subtraktionen durchgeführt werden als beim naiven Algorithmus, der das Produkt direkt gemäß (2.3) bestimmt. Deshalb ist der Strassen-Algorithmus in der Praxis erst ab einem gewissen n schneller als der naive Algorithmus.

2.1.3 Lösen von Rekursionsgleichungen

Wir haben in den vorangegangenen Abschnitten bereits einige Rekursionsgleichungen gesehen. Diese hatten gemein, dass $T(1) = \Theta(1)$ gilt. Für $n \geq 2$ haben wir die folgenden Gleichungen erhalten, wobei wir das Runden für den Moment wieder außer Acht lassen:

$$\begin{aligned} \textsc{BinarySearch}: &\quad T(n) = T(n/2) + \Theta(1), \\ \textsc{Mergesort}: &\quad T(n) = 2T(n/2) + \Theta(n), \\ \textsc{SimpleProduct}: &\quad T(n) = 8T(n/2) + \Theta(n^2), \\ \textsc{Strassen}: &\quad T(n) = 7T(n/2) + \Theta(n^2). \end{aligned}$$

Hat man eine Vermutung, wie die Lösung einer Rekursionsgleichung aussieht, so kann man diese mithilfe von vollständiger Induktion bestätigen. Um eine Vermutung zu entwickeln, ist es hilfreich, den *Rekursionsbaum* zu betrachten. Dieser Baum enthält für jeden rekursiven Aufruf einen Knoten, der mit der Größe der zu lösenden Teilinstanz beschriftet ist. Die Wurzel des Baumes repräsentiert den initialen Aufruf und die Kinder eines Knotens repräsentieren die rekursiven Aufrufe, die von ihm ausgehen. In Abbildung 2.2 ist der Rekursionsbaum von Mergesort dargestellt.

In Abbildung 2.2 haben wir ebenfalls notiert, dass der Rekursionsbaum eine Höhe von $\log_2(n)$ besitzt und dass es auf Ebene i genau 2^i Aufrufe von Mergesort gibt, die jeweils eine Laufzeit von höchstens $cn/2^i$ besitzen, wenn man die Laufzeit der rekursiven Aufrufe ignoriert. Um die Laufzeit von Mergesort zu bestimmen, genügt es die Laufzeiten aller Knoten im Rekursionsbaum zu addieren. Dies ergibt

$$T(n) \leq \sum_{i=0}^{\log_2 n} 2^i \cdot \frac{cn}{2^i} = cn \sum_{i=0}^{\log_2 n} 1 = cn(\log_2 n + 1).$$

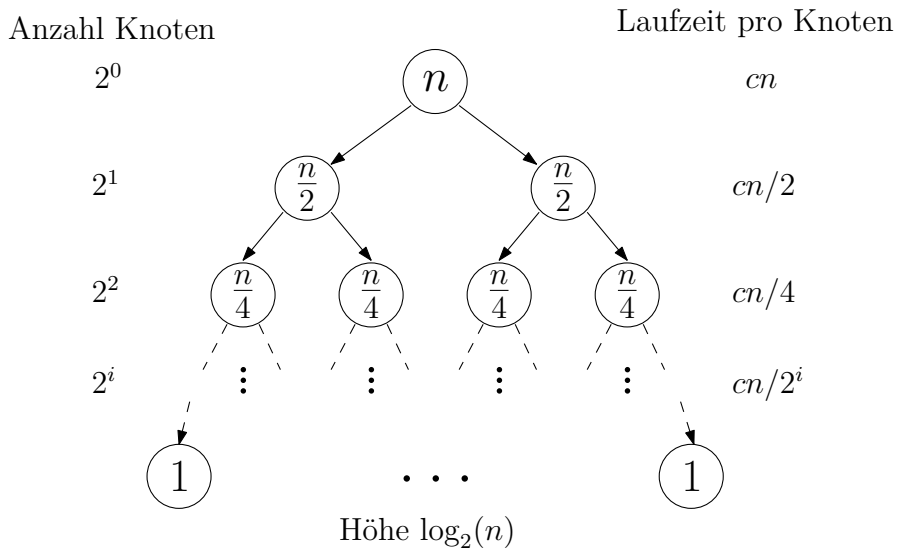


Abbildung 2.2: Der Rekursionsbaum von MERGESORT

Für $n = 2^k$ ist dies bereits ein formaler Beweis dafür, dass die Worst-Case-Laufzeit von MERGESORT in $O(n \log n)$ liegt. Da man auch hier wieder argumentieren kann, dass jeder Aufruf auf Level i eine Laufzeit von mindestens $c'n/2^i$ für eine geeignete Konstante $c' > 0$ besitzt, kann man analog argumentieren, dass die Worst-Case-Laufzeit sogar in $\Theta(n \log n)$ liegt.

Betrachten wir als weitere Beispiele die beiden Divide-and-Conquer-Algorithmen zur Matrixmultiplikation, die wir oben kennengelernt haben. Die Rekursionsbäume von beiden Algorithmen haben genauso wie der von MERGESORT die Höhe $\log_2 n$, da sich die Problemgröße n von Ebene zu Ebene halbiert. Auf Ebene i des Rekursionsbaumes erfolgen beim einfachen Algorithmus 8^i und beim Strassen-Algorithmus 7^i rekursive Aufrufe mit Instanzen der Größe $n/2^i$. Die Laufzeit eines Aufrufs auf Ebene i beträgt für beide Algorithmen höchstens $c(n/2^i)^2$. Für den naiven Algorithmus erhalten wir somit

$$T(n) \leq \sum_{i=0}^{\log_2 n} 8^i \cdot c\Big(\frac{n}{2^i}\Big)^2 = cn^2 \sum_{i=0}^{\log_2 n} \frac{8^i}{2^{2i}} = cn^2 \sum_{i=0}^{\log_2 n} 2^i = cn^2(2^{\log_2(n)+1} - 1) = O(n^3).$$

Für den Strassen-Algorithmus erhalten wir

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\log_2 n} 7^i \cdot c\Big(\frac{n}{2^i}\Big)^2 = cn^2 \sum_{i=0}^{\log_2 n} \frac{7^i}{2^{2i}} = cn^2 \sum_{i=0}^{\log_2 n} \Big(\frac{7}{4}\Big)^i \\ &= cn^2 \frac{\Big(\frac{7}{4}\Big)^{\log_2(n)+1} - 1}{\Big(\frac{7}{4}\Big) - 1} = O\Big(n^2 \Big(\frac{7}{4}\Big)^{\log_2 n}\Big) \\ &= O\Big(n^2 n^{\log_2(7/4)}\Big) = O\Big(n^2 n^{\log_2(7)-2}\Big) = O\Big(n^{\log_2 7}\Big). \end{aligned}$$

Wir haben in der letzten Rechnung ausgenutzt, dass $\sum_{i=0}^{n} q^i = \frac{q^{n+1}-1}{q-1}$ für $q > 1$ und $n \in \mathbb{N}$ gilt und dass $a^{\log_2 b} = b^{\log_2 a}$ für $a, b > 1$ gilt.

Das folgende Theorem verallgemeinert die obigen Betrachtungen.

Theorem 2.3. *Seien $a \geq 1$ und $b > 1$ Konstanten und sei $f\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Funktion. Ferner sei $T\colon \mathbb{N} \to \mathbb{R}$ definiert durch $T(1) = \Theta(1)$ und*

$$T(n) = aT(n/b) + f(n)$$

für $n \geq 2$. Dabei steht n/b entweder für $\lceil n/b \rceil$ oder $\lfloor n/b \rfloor$. Dann kann die Funktion T asymptotisch wie folgt beschrieben werden.

1. *Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für eine Konstante $\varepsilon > 0$ gilt, so gilt $T(n) = \Theta(n^{\log_b a})$.*
2. *Falls $f(n) = \Theta(n^{\log_b a})$ gilt, so gilt $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
3. *Falls $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für eine Konstante $\varepsilon > 0$ und $af(n/b) \leq cf(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n gilt, so gilt $T(n) = \Theta(f(n))$.*

Auf den ersten Blick wirkt Theorem 2.3 vielleicht etwas unübersichtlich. Letztendlich werden aber lediglich die Funktionen f und $n^{\log_b a}$ miteinander verglichen und im Wesentlichen bestimmt die größere von beiden die Lösung der Rekursionsgleichung. Im ersten Fall wächst f langsamer als die Funktion $n^{\log_b a}$. In diesem Fall ergibt sich $T(n) = \Theta(n^{\log_b a})$ als Lösung. Im zweiten Fall wachsen die Funktionen f und $n^{\log_b a}$ gleich schnell, was verglichen mit dem ersten Fall zu einem zusätzlichen Faktor von $\Theta(\log n)$ führt. Im dritten Fall wächst f schneller als $n^{\log_b a}$. Dann ist die Lösung $T(n) = \Theta(f(n))$.

Natürlich müssen noch einige technische Aspekte beachtet werden. Beispielsweise decken die drei Fälle nicht alle Möglichkeiten ab, da es im ersten Fall nicht genügt, dass f langsamer als $n^{\log_b a}$ wächst, sondern f muss mindestens um den Faktor n^{ε} langsamer wachsen. Funktionen wie $f(n) = n^{\log_b a}/\log_2 n$ sind also nicht durch Theorem 2.3 abgedeckt. Analog muss f im dritten Fall um mindestens den Faktor n^{ε} schneller als $n^{\log_b a}$ wachsen und die technische Bedingung $af(n/b) \leq cf(n)$ muss für große n erfüllt sein.

Wir werden Theorem 2.3 nicht beweisen. Ignoriert man das Runden, so kann der Beweis wieder durch die Betrachtung eines Rekursionsbaumes geführt werden. Abbildung 2.3 zeigt den entsprechenden Rekursionsbaum. Die Analyse ist sehr ähnlich zu unseren obigen Beispielen, wenngleich etwas technischer. Beachtet man die Rundungen, so werden die Details noch technischer, der Beweis basiert aber nach wie vor auf einer Analyse des Rekursionsbaumes. Bei Betrachtung des Rekursionsbaumes in Abbildung 2.3 wird die Rolle der Funktion $n^{\log_b a}$ noch etwas klarer. Es handelt sich hierbei um die Anzahl der Blätter des Rekursionsbaumes, da nach Anwendung der Logarithmengesetze gilt $a^h = a^{\log_b n} = n^{\log_b a}$, wobei h die Höhe des Baumes bezeichnet. Somit ergibt sich folgende Interpretation der drei Fälle des Theorems. Im ersten Fall dominieren die Blätter die Laufzeit, im zweiten Fall ist die Gesamtlaufzeit auf jedem der Level des Rekursionsbaumes in der gleichen Größenordnung, und im dritten Fall dominiert die Wurzel die Laufzeit

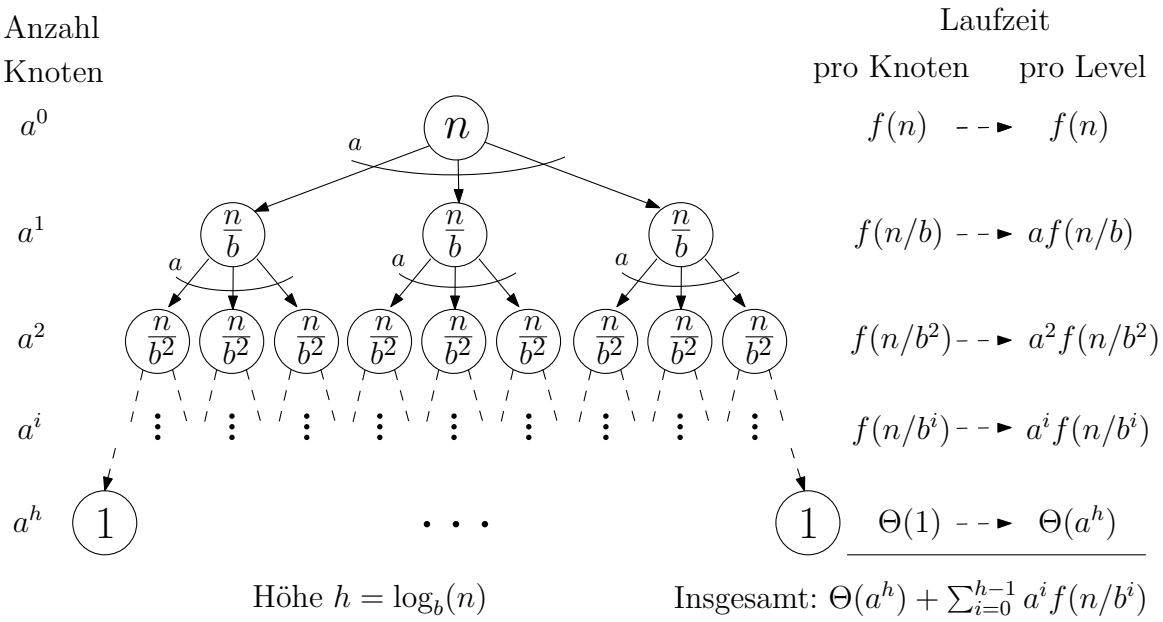


Abbildung 2.3: Der Rekursionsbaum in der Analyse von Theorem 2.3 unter der Annahme, dass n eine Potenz von b ist.

Beispiele für die Anwendung von Theorem 2.3

Wir gehen in allen Beispielen davon aus, dass $T(1) = \Theta(1)$ gilt.

- $T(n) = kT(n/k) + n$ für ein beliebiges $k \in \mathbb{N}$ mit $k \geq 2$
  Es gilt $a = b = k$ und damit $n^{\log_b a} = n$. Wegen $f(n) = n = \Theta(n^{\log_b a})$ gehört diese Rekursionsgleichung zum zweiten Fall und es ergibt sich $T(n) = \Theta(n \log n)$.
- $T(n) = T(2n/3) + 1$
  Es gilt $a = 1$, $b = 3/2$ und $f(n) = 1$. Dementsprechend gilt $n^{\log_{3/2} 1} = n^0 = 1 = \Theta(f)$ und der zweite Fall liefert $T(n) = \Theta(\log n)$.
- $T(n) = kT(n/k) + n \log_2 n$ für ein beliebiges $k \in \mathbb{N}$ mit $k \geq 2$
  Es gilt $a = b = k$. Zwar wächst $f(n) = n \log_2 n$ schneller als $n^{\log_b a} = n$, es kann jedoch trotzdem nicht der dritte Fall angewendet werden, da dieser voraussetzt, dass $f(n)$ um mindestens einen Faktor von n^ε für ein $\varepsilon > 0$ schneller wächst.
- $T(n) = 8T(n/2) + \Theta(n^2)$
  Dies ist die Rekursionsgleichung von SimpleProduct. Es gilt $a = 8$ und $b = 2$ und damit $n^{\log_b a} = n^3$. Es ist $f(n) = n^2 = O(n^{\log_b a - \varepsilon})$ für $\varepsilon = 1$ und damit greift der erste Fall, der $T(n) = \Theta(n^{\log_b a}) = \Theta(n^3)$ als Lösung liefert.
- $T(n) = 8T(n/2) + \Theta(n^4)$
  Es gilt $a = 8$ und $b = 2$ und damit $n^{\log_b a} = n^3$. Es ist $f(n) = n^4 = \Omega(n^{\log_b a + \varepsilon})$ für $\varepsilon = 1$ und damit greift der dritte Fall, da zusätzlich $8f(n/2) = n^4/2 \leq f(n)/2$ gilt. Es ergibt sich die Lösung $T(n) = \Theta(f(n)) = \Theta(n^4)$.

2.2 Greedy-Algorithmen

Nach Divide-and-Conquer-Algorithmen lernen wir in diesem Abschnitt mit *Greedy-Algorithmen* eine zweite wichtige Klasse von Algorithmen kennen. Greedy-Algorithmen lösen Probleme schrittweise und treffen dabei in jedem Schritt eine Entscheidung, die den größtmöglichen Gewinn im aktuellen Schritt verspricht.

Um diese vage Beschreibung zu illustrieren, betrachten wir ein Beispiel. Unsere Aufgabe ist es, einen gegebenen Betrag von z Cent mit möglichst wenigen Münzen zu bezahlen. Dazu stehen uns von jeder Sorte beliebig viele Münzen zur Verfügung. Beim Euro sind das Münzen mit den Werten 1, 2, 5, 10, 20, 50, 100 und 200 Cent. Ein Greedy-Algorithmus für dieses *Wechselgeldproblem* stellt die Münzen Schritt für Schritt zusammen. Dazu merkt er sich zu jedem Zeitpunkt in einer Variable z', die zu Beginn mit z initialisiert wird, wie viele Cent noch fehlen. Er fügt den bisher gewählten Münzen dann die größte Münze hinzu, deren Wert z' nicht übersteigt. Dieses Verfahren kann als Greedy-Algorithmus betrachtet werden, da in jedem Schritt versucht wird, möglichst viel von dem noch fehlenden Betrag mit einer Münze abzudecken.

Um 39 Cent zusammenzustellen wählt der Algorithmus also beispielsweise Münzen mit den Werten $20, 10, 5, 2, 2$ in dieser Reihenfolge. Es ist nicht möglich, den Betrag von 39 Cent mit weniger als fünf Münzen genau zu erreichen. Betrachtet man weitere Beispiele, so gelangt man schnell zu der Vermutung, dass der Algorithmus für jeden Betrag die minimale Zahl an Münzen findet. Dies ist tatsächlich der Fall.

Theorem 2.4. *Für Münzen mit den oben genannten Werten findet der Greedy-Algorithmus für jeden Betrag eine Lösung mit der kleinstmöglichen Anzahl an Münzen.*

Beweisskizze. Sei $z \in \mathbb{N}$ ein beliebiger Betrag, der mit so wenigen Münzen wie möglich genau erreicht werden soll. Für $i \in M := \{1, 2, 5, 10, 20, 50, 100, 200\}$ bezeichne $x_i \in \mathbb{N}_0$ die Anzahl an Münzen mit Wert i, die der Greedy-Algorithmus auswählt. Unser erstes Ziel ist es, die Lösung des Greedy-Algorithmus besser zu charakterisieren. Dazu beobachten wir, dass aus der Definition des Greedy-Algorithmus direkt für jedes $i \in M$ mit $i > 1$ die Ungleichung

$$\sum_{j \in M, j < i} j x_j < i$$

folgt. Würde diese Ungleichung nicht gelten, dann hätte der Algorithmus irgendwann den Restbetrag mit Münzen von Wert kleiner i dargestellt, obwohl er eine zusätzliche Münze mit Wert i auswählen konnte. Da der Algorithmus aber immer die Münze mit dem größten Wert wählt, muss die Ungleichung für die vom Greedy-Algorithmus berechnete Lösung erfüllt sein.

Man kann leicht argumentieren, dass diese Ungleichungen zusammen mit der Bedingung

$$\sum_{i \in M} i x_i = z$$

genau eine Lösung besitzen. Dies kann per Induktion über z wie folgt gezeigt werden. Für $z = 1$ ist die Behauptung trivial erfüllt, da es nur eine mögliche Lösung gibt und diese erfüllt die Ungleichungen. Für $z > 1$ beobachten wir, dass eine Lösung, die die Ungleichung

$$\sum_{j \in M, j<i} j x_j < i$$

für ein maximales $i \leq z$ erfüllt, mindestens eine Münze vom Wert i enthalten muss. Die Behauptung für z folgt aus dem Induktionsschritt für $z' = z - i$, zusammen mit diese Münze vom Wert i. Somit ist die Lösung, die der Greedy-Algorithmus berechnet, durch diese Ungleichungen eindeutig bestimmt.

Im zweiten Schritt des Beweises gehen wir davon aus, dass wir eine beliebige optimale Lösung $(y_i \in \mathbb{N}_0)_{i \in M}$ mit $\sum_{i \in M} i y_i = z$ und der kleinstmöglichen Zahl $\sum_{i \in M} y_i$ an Münzen vorliegen haben. Wir argumentieren, dass diese Lösung auch die obigen Ungleichungen erfüllen muss und somit der Lösung $(x_i \in \mathbb{N}_0)_{i \in M}$ des Greedy-Algorithmus entspricht. Dazu beobachten wir zunächst, dass $y_1 \leq 1$ gelten muss, da zwei 1-Cent-Münzen durch eine 2-Cent-Münze ersetzt werden könnten und die Lösung $(y_i \in \mathbb{N}_0)_{i \in M}$ somit nicht optimal wäre. Daraus folgt die Ungleichung für $i = 2$:

$$1 \cdot y_1 < 2$$

Ebenso muss $y_2 \leq 2$ gelten, da drei 2-Cent-Münzen durch eine 5-Cent-Münze und eine 1-Cent-Münze ersetzt werden könnten. Auch kann nicht gleichzeitig $y_1 = 1$ und $y_2 = 2$ gelten, da dann die 1-Cent-Münze und die beiden 2-Cent-Münzen durch eine 5-Cent-Münze ersetzt werden könnten. Aus $y_1 \leq 1$ und $y_2 \leq 2$ und der Tatsache, dass nicht beide Ungleichungen mit Gleichheit erfüllt sein können, folgt die Ungleichung für $i = 5$:

$$1 \cdot y_1 + 2 \cdot y_2 < 5$$

Dieses Argument kann man für jedes $i \in M$ analog fortsetzen. □

Das Wechselgeldproblem liefert auch direkt ein Beispiel dafür, dass Greedy-Algorithmen nicht immer optimale Lösungen berechnen. Für eine Währung, die nur Münzen mit den Werten 1, 3 und 4 enthält, wählt der Greedy-Algorithmus für $z = 6$ drei Münzen mit den Werten $4, 1, 1$ aus, obwohl der Betrag durch zwei 3-Cent-Münzen dargestellt werden kann. Die Korrektheit von Theorem 2.4 hängt also ganz entscheidend davon ab, welche Münzen in der betrachteten Währung vorhanden sind.

Auch für andere Probleme ist es natürlich nicht so, dass Greedy-Algorithmen immer optimale Lösungen finden. Dennoch lohnt es sich für viele Probleme zunächst darüber nachzudenken, ob sie mit einem Greedy-Algorithmus gelöst werden können, da Greedy-Algorithmen normalerweise sehr effizient sind und für erstaunlich viele Probleme optimale oder zumindest in einem gewissen Sinne gute Lösungen berechnen. Wir werden in diesem Abschnitt zwei weitere Probleme kennenlernen, für die das der Fall ist.

2.2.1 Optimale Auswahl von Aufgaben

Wir betrachten nun ein Problem, das auch als *Interval Scheduling* bekannt ist. Die Eingabe für das Problem besteht aus einer Menge $S = \{1, \ldots, n\}$ von Aufgaben. Jede Aufgabe i besitzt einen Startzeitpunkt $s_i \geq 0$ und einen Fertigstellungszeitpunkt $f_i > s_i$. Außerdem steht ein Prozessor zur Verfügung, der zu jedem Zeitpunkt maximal eine Aufgabe bearbeiten kann. Wählen wir Aufgabe i zur Bearbeitung aus, so kann in dem Zeitintervall $[s_i, f_i)$ keine weitere Aufgabe von dem Prozessor bearbeitet werden. Unsere Aufgabe ist es, eine größtmögliche Teilmenge der Aufgaben zu finden, die ohne Kollision von dem Prozessor bearbeitet werden kann. Formal sind wir also an einer größtmöglichen Teilmenge $S' \subseteq S$ interessiert, sodass für jedes Paar von Aufgaben $i \in S'$ und $j \in S'$ mit $i \neq j$ die Intervalle $[s_i, f_i)$ und $[s_j, f_j)$ disjunkt sind. Wir sagen dann auch, dass die Aufgaben aus S' paarweise nicht kollidieren.

Wir möchten die Teilmenge S' mit einem Greedy-Algorithmus finden. Es gibt mehrere naheliegende Möglichkeiten, wie dieser ausgestaltet sein kann. Eine Möglichkeit ist es, in jedem Schritt die Aufgabe mit dem kleinsten Startzeitpunkt zu wählen, die nicht mit den bisher gewählten Aufgaben kollidiert. Alternativ kann man auch in jedem Schritt die Aufgabe mit der kürzesten Dauer wählen, die nicht mit den bisher gewählten Aufgaben kollidiert. Beide Möglichkeiten sind unten in Pseudocode dargestellt.

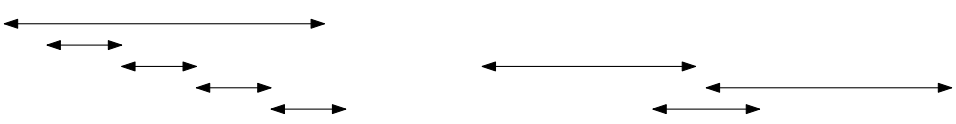
GreedyStart

```
1  S* = ∅;
2  while (S ≠ ∅) {
3     Wähle Aufgabe i ∈ S mit
        kleinstem Startzeitpunkt s_i.
4     S* = S* ∪ {i};
5     Lösche alle Aufgaben aus S,
        die mit i kollidieren.
6  }
7  return S*;
```

GreedyDauer

```
1  S* = ∅;
2  while (S ≠ ∅) {
3     Wähle Aufgabe i ∈ S mit der
        kürzesten Dauer f_i − s_i.
4     S* = S* ∪ {i};
5     Lösche alle Aufgaben aus S,
        die mit i kollidieren.
6  }
7  return S*;
```

Man erkennt schnell, dass keiner der beiden vorgeschlagenen Greedy-Algorithmen für jede Eingabe die optimale Lösung berechnet. Unten links ist eine Eingabe illustriert, in der die Aufgabe mit dem kleinsten Startzeitpunkt sehr lang ist. Der Algorithmus GreedyStart wählt nur diese Aufgabe aus, obwohl es möglich ist, alle vier übrigen Aufgaben zusammen zu bearbeiten. Unten rechts ist eine Eingabe illustriert, bei der die kürzeste Aufgabe mit zwei längeren Aufgaben kollidiert, die beide zusammen bearbeitet werden können. Statt der zwei langen Aufgaben wählt der Algorithmus GreedyDauer nur die kurze Aufgabe.



Es bleibt noch eine naheliegende Variante des Greedy-Algorithmus. Diese wählt in jedem Schritt die Aufgabe mit dem frühesten Fertigstellungszeitpunkt, die nicht mit den bisher gewählten Aufgaben kollidiert.

```
GreedyEnde
1  S* = ∅;
2  while (S ≠ ∅) {
3      Wähle Aufgabe i ∈ S mit dem frühesten Fertigstellungszeitpunkt f_i.
4      S* = S* ∪ {i};
5      Lösche alle Aufgaben aus S, die mit i kollidieren.
6  }
7  return S*;
```

Wir zeigen, dass dieser Greedy-Algorithmus für jede Eingabe eine größtmögliche Menge von Aufgaben auswählt. Dazu benötigen wir das folgende Lemma.

Lemma 2.5. *Es sei S eine Menge von Aufgaben und es sei $i \in S$ eine Aufgabe mit dem frühesten Fertigstellungszeitpunkt f_i. Dann gibt es eine optimale Auswahl $S' \subseteq S$ von paarweise nicht kollidierenden Aufgaben mit $i \in S'$.*

Beweis. Sei $S^\star \subseteq S$ eine größtmögliche Auswahl von paarweise nicht kollidierenden Aufgaben. Gilt $i \in S^\star$, so ist nichts mehr zu zeigen. Ansonsten sei $j \in S^\star$ die Aufgabe aus $S^\star$ mit dem frühesten Fertigstellungszeitpunkt f_j. Aus der Definition von i folgt $f_j \geq f_i$. Wir behaupten, dass auch $S' := (S^\star \setminus \{j\}) \cup \{i\}$ eine Menge von paarweise nicht kollidierenden Aufgaben ist. Da diese genauso groß ist wie die optimale Menge $S^\star$, ist auch S' optimal. Per Definition enthält S' die Aufgabe i, sodass damit das Lemma folgt.

Es bleibt zu zeigen, dass die Aufgaben aus S' paarweise nicht kollidieren. Zwei Aufgaben $k, k' \in S' \setminus \{i\}$ können nicht kollidieren, da sie beide auch in $S^\star$ enthalten sind. Kollisionen können also höchstens zwischen der Aufgabe i und einer Aufgabe $k \in S' \setminus \{i\}$ auftreten. Angenommen dies ist der Fall und es gibt eine Aufgabe $k \in S' \setminus \{i\}$, mit der Aufgabe i kollidiert. Da die Aufgaben j und k beide zu $S^\star$ gehören, sind die Intervalle $[s_j, f_j)$ und $[s_k, f_k)$ disjunkt. Aus der Wahl von j folgt $f_j \leq f_k$ und damit wegen der Überschneidungsfreiheit der Intervalle $s_j < f_j \leq s_k < f_k$. Mit $f_i \leq f_j$ folgt $s_i < f_i \leq s_k < f_k$ und damit sind auch die Intervalle $[s_i, f_i)$ und $[s_k, f_k)$ disjunkt. Dies ist ein Widerspruch zu der Annahme, dass die Aufgaben i und k kollidieren. □

Nun können wir die Korrektheit von GreedyEnde nachweisen.

Theorem 2.6. *Der Algorithmus* GreedyEnde *wählt für jede Instanz eine größtmögliche Menge von paarweise nicht kollidierenden Aufgaben aus.*

Beweis. Wir zeigen zum Beweis des Theorems die folgende Invariante: In Zeile 2 gilt stets, dass die Menge $S^\star$ der bereits ausgewählten Aufgaben mit Aufgaben, die sich noch in der Menge S befinden, zu einer optimalen Auswahl von Aufgaben erweitert

werden kann. Außerdem gibt es keine Aufgaben in S, die mit Aufgaben aus $S^\star$ kollidieren. Diese Invariante impliziert das Theorem, denn ist S die leere Menge, was am Ende der Fall ist, so folgt direkt, dass $S^\star$ eine optimale Auswahl von Aufgaben ist.

In der ersten Iteration gilt die Invariante, da $S^\star$ leer ist und S noch alle Aufgaben enthält. Sei nun die Invariante zu Beginn eines Schleifendurchlaufs erfüllt. Dann gibt es eine optimale Auswahl $\hat{S} \subseteq S^\star \cup S$ von Aufgaben, die alle Aufgaben aus $S^\star$ und gegebenenfalls zusätzliche Aufgaben aus S enthält. Außerdem kollidieren Aufgaben aus $S^\star$ nicht mit Aufgaben aus S. Das heißt, bei der Frage, welche Aufgaben aus S den Aufgaben aus $S^\star$ noch hinzugefügt werden müssen, um eine optimale Auswahl $\hat{S}$ zu erhalten, spielen die Aufgaben aus $S^\star$ keine Rolle. Die Menge $\hat{S} \setminus S^\star$ dieser Aufgaben ist eine größtmögliche Teilmenge von paarweise nicht kollidierenden Aufgaben aus S. Gemäß Lemma 2.5 existiert eine solche Menge, die die Aufgabe $i \in S$ mit dem frühesten Fertigstellungszeitpunkt f_i aller Aufgaben aus S enthält. Genau diese Aufgabe fügt GreedyEnde der Menge $S^\star$ hinzu. Anschließend werden aus S alle Aufgaben entfernt, die mit dieser Aufgabe i kollidieren. Dies garantiert zusammen, dass auch am Anfang der nächsten Iteration die Invariante wieder erfüllt ist. □

Nachdem wir uns davon überzeugt haben, dass der Algorithmus GreedyEnde auf jeder Eingabe das korrekte Ergebnis berechnet, betrachten wir nun noch seine Laufzeit. Um diese genau bestimmen zu können, konkretisieren wir den Pseudocode. Im folgenden Pseudocode besteht die Eingabe aus zwei Feldern $s[0 \ldots n-1]$ und $f[0 \ldots n-1]$, die für jede Aufgabe den Start- und den Fertigstellungszeitpunkt enthalten. Wir gehen ferner davon aus, dass die Aufgaben bezüglich ihres Fertigstellungszeitpunktes aufsteigend sortiert sind, dass also $f[0] \leq f[1] \leq \ldots \leq f[n-1]$ gilt.

```
GreedyEnde(int[] s, int[] f)
1   S* = {0};
2   k = 0;
3   for (int i = 1; i < n; i++) {
4       if (s[i] >= f[k]) {
5           S* = S* ∪ {i};
6           k = i;
7       }
8   }
9   return S*;
```

Da die for-Schleife in Zeile 3 nur $n-1$ mal durchlaufen wird und man n Zahlen mit Mergesort in Zeit $O(n \log n)$ sortieren kann, ergibt sich das folgende Theorem.

Theorem 2.7. *Die Laufzeit des Algorithmus* GreedyEnde *beträgt $O(n \log n)$, wobei $n = |S|$ die Anzahl der Aufgaben in der Eingabe bezeichnet. Sind die Aufgaben bereits aufsteigend nach ihrem Fertigstellungszeitpunkt sortiert, so beträgt die Laufzeit $O(n)$.*

2.2.2 Rucksackproblem mit teilbaren Objekten

Eine Eingabe für das *Rucksackproblem* besteht aus einer Menge von $n \in \mathbb{N}$ Objekten. Jedes Objekt $i \in \{1, \ldots, n\}$ besitzt einen *Nutzen* $p_i \in \mathbb{N}$ und ein *Gewicht* $w_i \in \mathbb{N}$. Außerdem ist eine *Kapazität* $t \in \mathbb{N}$ gegeben. Anschaulich soll eine Teilmenge der Objekte gefunden werden, die in den Rucksack passt und unter dieser Bedingung maximalen Nutzen besitzt. Um dieses Problem formal zu beschreiben, führen wir für jedes Objekt $i \in \{1, \ldots, n\}$ eine Indikatorvariable $x_i \in \{0, 1\}$ ein, die angibt, ob wir es in den Rucksack packen ($x_i = 1$) oder nicht ($x_i = 0$). Eine Belegung $(x_1, \ldots, x_n)$ dieser Indikatorvariablen ist *gültig*, wenn $\sum_{i=1}^{n} w_i x_i \leq t$ gilt. Wir werden im Folgenden Belegungen auch als *Lösungen* bezeichnen. Beim Rucksackproblem ist eine gültige Lösung $x = (x_1, \ldots, x_n) \in \{0, 1\}^n$ gesucht, die unter allen gültigen Lösungen die *Zielfunktion* $\sum_{i=1}^{n} p_i x_i$ maximiert.

Das Rucksackproblem wird uns in dieser Vorlesung und im nächsten Semester noch beschäftigen. Zunächst betrachten wir aber eine abgewandelte Problemstellung, bei der die Objekte *teilbar* sind. Der einzige Unterschied zu der obigen Formulierung ist, dass statt $x_i \in \{0, 1\}$ nur noch $x_i \in [0, 1]$ gefordert wird. Anschaulich bedeutet das, dass Objekte auch nur teilweise eingepackt werden dürfen und dann den entsprechenden anteiligen Nutzen bringen. Zur Abgrenzung bezeichnen wir eine Lösung $x = (x_1, \ldots, x_n) \in \{0, 1\}^n$ als *ganzzahlige Lösung* und eine Lösung $x = (x_1, \ldots, x_n) \in [0, 1]^n$ als *fraktionale Lösung*. Sofern wir es nicht anders angeben, ist mit dem Rucksackproblem die Problemvariante gemeint, in der Objekte nicht teilbar sind und nur ganzzahlige Lösungen erlaubt sind.

Wir entwerfen einen Greedy-Algorithmus, der das Rucksackproblem mit teilbaren Objekten löst. Dieser Algorithmus sortiert die Objekte zunächst absteigend gemäß ihrer *Effizienz* (dem Quotienten aus Nutzen und Gewicht) und packt dann die Objekte nach und nach ein, bis der Rucksack voll ist. Dabei wird das letzte Objekt gegebenenfalls nur anteilig eingepackt.

```
GreedyKP
1  Sortiere die Objekte gemäß ihrer Effizienz. Danach gelte
                        p_1/w_1 ≥ p_2/w_2 ≥ ... ≥ p_n/w_n.
2  for (int i = 1; i <= n; i++) { x_i = 0; }
3  int i = 1;
4  while ((t > 0) && (i <= n)) {
5      if (t >= w_i) { x_i = 1; t = t - w_i; }
6      else { x_i = t/w_i; t = 0; }
7      i++;
8  }
9  return (x_1, ..., x_n);
```

Die Leserin und der Leser werden feststellen, dass der Pseudocode schon deutlich von Java abweicht und wir beispielsweise x_i statt $x[i]$ schreiben. Damit entspricht der

Pseudocode der Notation, die wir auch unten in der Analyse des Algorithmus benutzen werden.[3]

Theorem 2.8. *Der Algorithmus* GreedyKP *berechnet für jede Instanz des Rucksackproblems mit teilbaren Objekten in Zeit $O(n \log n)$ eine optimale Lösung.*

Beweis. Die Laufzeit von GreedyKP wird durch den ersten Schritt dominiert, in dem die n Objekte sortiert werden müssen. Setzt man dazu beispielsweise Mergesort ein, so ergibt sich eine Laufzeit von $O(n \log n)$ für Zeile 1. Der restliche Algorithmus besitzt eine lineare Laufzeit von $O(n)$, da sowohl die for-Schleife in Zeile 2 als auch die while-Schleife in Zeile 4 maximal n mal durchlaufen werden.

Wir gehen im Folgenden davon aus, dass die Objekte bereits nach ihrer Effizienz sortiert sind. Sei $x = (x_1, \ldots, x_n) \in [0,1]^n$ die Lösung, die GreedyKP berechnet, und sei $x^\star = (x_1^\star, \ldots, x_n^\star) \in [0,1]^n$ eine optimale Lösung. Passen alle Objekte gemeinsam in den Rucksack, gilt also $\sum_{i=1}^n w_i \leq t$, so gilt $x = (1, 1, \ldots, 1)$. Dies ist in diesem Fall auch die optimale Lösung. Wir betrachten deshalb im Folgenden nur den Fall $\sum_{i=1}^n w_i > t$. Auch in diesem Fall hat die Lösung x eine einfache Struktur: es gibt ein $i \in \{1, \ldots, n\}$, sodass $x_1 = \ldots = x_{i-1} = 1$, $x_i < 1$ und $x_{i+1} = \ldots = x_n = 0$ gilt. Außerdem sieht man leicht, dass GreedyKP den Rucksack komplett ausfüllt, dass also $\sum_{i=1}^n x_i w_i = t$ gilt. Auch jede optimale Lösung muss den Rucksack komplett ausfüllen, das heißt es gilt insbesondere $\sum_{i=1}^n x_i^\star w_i = t$. Wäre dies nicht der Fall, so könnte der Nutzen gesteigert werden, indem ein $x_i^\star < 1$ erhöht wird.

Wir zeigen nun, dass wir die Lösung $x^\star$ Schritt für Schritt in die Lösung x transformieren können, ohne dabei den Nutzen zu verringern. Wir betrachten den kleinsten Index i mit $x_i^\star < 1$. Gilt $x_{i+1}^\star = \ldots = x_n^\star = 0$, so sind die Lösungen $x^\star$ und x identisch (die Leserin und der Leser überlegen sich, warum das der Fall ist), das heißt, die Lösung x ist optimal. Ansonsten gibt es einen Index $j > i$ mit $x_j^\star > 0$. Wir nehmen im Folgenden an, dass j der größte solche Index ist. Intuitiv können wir nun einfach den Anteil von Objekt j reduzieren und dafür den Anteil von Objekt i erhöhen. Da die Effizienz von Objekt i mindestens so groß ist wie die von Objekt j, verringert sich dadurch der Nutzen der Lösung nicht. Formal sei $\varepsilon = \min\{w_i - x_i^\star w_i, x_j^\star w_j\}$ das Gewicht, das wir von Objekt j zu Objekt i verlagern. Wir betrachten dementsprechend die Lösung $x' = (x_1', \ldots, x_n')$ mit

$$x_k' = \begin{cases} x_i^\star + \frac{\varepsilon}{w_i} & \text{falls } k = i, \\ x_j^\star - \frac{\varepsilon}{w_j} & \text{falls } k = j, \\ x_k^\star & \text{sonst.} \end{cases}$$

Diese Lösung ist gültig, denn

$$x_i' = x_i^\star + \frac{\varepsilon}{w_i} \leq x_i^\star + \frac{w_i - x_i^\star w_i}{w_i} = 1,$$
$$x_j' = x_j^\star - \frac{\varepsilon}{w_j} \geq x_j^\star - \frac{x_j^\star w_j}{w_j} = 0$$

[3] Auf Englisch wird das Rucksackproblem als *knapsack problem* bezeichnet und dementsprechend mit KP abgekürzt.

und

$$\sum_{k=1}^{n} x'_k w_k = \sum_{k=1}^{n} x^\star_k w_k + \frac{\varepsilon}{w_i} w_i - \frac{\varepsilon}{w_j} w_j = \sum_{k=1}^{n} x^\star_k w_k = t.$$

Außerdem gilt

$$\sum_{k=1}^{n} x'_k p_k = \sum_{k=1}^{n} x^\star_k p_k + \frac{\varepsilon}{w_i} p_i - \frac{\varepsilon}{w_j} p_j \geq \sum_{k=1}^{n} x^\star_k p_k,$$

wobei wir für die Ungleichung ausgenutzt haben, dass das Objekt i mindestens so effizient ist wie das Objekt j. Damit haben wir nachgewiesen, dass x' eine gültige Lösung ist, deren Nutzen mindestens so groß ist wie der von $x^\star$.

Den oben beschriebenen Transformationsschritt führen wir solange durch, wie er möglich ist. Ist er nicht mehr möglich, so haben wir die Lösung x erreicht. Dies muss nach spätestens n Schritten der Fall sein, da mit jedem Transformationsschritt eine Variable (entweder Variable x'_i oder Variable x'_j) auf 0 oder 1 gesetzt wird. Diese Variable wird in den folgenden Schritten nicht mehr verändert, wenn wir ausnutzen, dass j in dem obigen Argument größtmöglich gewählt wird. □

Das Rucksackproblem ist nicht nur ein rein theoretisches Problem, sondern es tritt in vielerlei Anwendungen zumindest als Teilproblem auf. Man denke beispielsweise an ein Speditionsunternehmen, das mehrere Aufträge und einen LKW mit begrenzter Kapazität hat. Die Auswahl, welche Aufträge angenommen werden sollten, um den Gewinn zu maximieren, lässt sich auf naheliegende Weise als Rucksackproblem formulieren. Tatsächlich sind in vielen Anwendungen die Objekte aber nicht teilbar. Dies bringt uns zu der Frage, ob der Algorithmus GreedyKP so modifiziert werden kann, dass er auch das Rucksackproblem (mit nicht teilbaren Objekten) löst. Wir werden im nächsten Semester noch ausführlich über das Rucksackproblem sprechen und sehen, dass es höchstwahrscheinlich keine einfache Abwandlung von GreedyKP für dieses Problem gibt. Mehr noch: Es gibt wahrscheinlich überhaupt keinen effizienten Algorithmus für das Rucksackproblem.

Wir werden deshalb unsere Anforderung aufgeben, stets eine optimale Lösung zu berechnen. Stattdessen geben wir uns damit zufrieden, Lösungen zu finden, die einen möglichst hohen (aber eben nicht immer optimalen) Nutzen besitzen. Der Algorithmus GreedyKP berechnet eine Lösung, in der es maximal ein geteiltes Objekt gibt. Alle anderen Objekte sind entweder vollständig oder gar nicht in der Lösung enthalten. Eine naheliegende Idee ist es, den Algorithmus so zu modifizieren, dass er das geteilte Objekt gar nicht einpackt. Dadurch erhält man eine gültige Lösung für das Rucksackproblem. Die einzige Änderung dazu erfolgt in Zeile 6 des Pseudocodes, in der wir $x_i = 0$ statt $x_i = \frac{t}{w_i}$ setzen. Diesen Algorithmus nennen wir IntGreedyKP, wobei der Zusatz „Int “ andeutet, dass eine ganzzahlige (auf Englisch *integer*) Lösung berechnet wird.

Wie gut sind die Lösungen, die IntGreedyKP berechnet, verglichen mit optimalen Lösungen für das Rucksackproblem? Ein einfaches Beispiel zeigt, dass die Lösung, die IntGreedyKP berechnet, beliebig schlecht sein kann. Um dies zu sehen, genügt es, eine Instanz mit zwei Objekten zu betrachten. Sei $p_1 = 2$, $w_1 = 1$, $p_2 = M$, $w_2 = M$ und $t = M$ für ein beliebig großes $M > 2$. Die optimale Lösung enthält nur das zweite

Objekt und erzielt einen Nutzen von M. Der Algorithmus IntGreedyKP packt nur das erste Objekt in den Rucksack, da dieses eine höhere Effizienz als das zweite besitzt und beide Objekte zusammen nicht in den Rucksack passen. Während der Nutzen der optimalen Lösung M ist, was wir beliebig groß wählen können, erreicht die Lösung von IntGreedyKP nur einen Nutzen von 2.

Um das Verhältnis der Nutzen der optimalen Lösung und der Lösung, die IntGreedyKP berechnet, beliebig groß werden zu lassen, wird in dem obigen Beispiel ausgenutzt, dass es ein Objekt gibt, das alleine schon sehr wertvoll ist. Um für Eingaben mit dieser Eigenschaft bessere Lösungen zu berechnen, erweitern wir den Algorithmus IntGreedyKP um einen weiteren Schritt und erhalten den Algorithmus ApproxKP. Dieser überprüft, ob die Lösung $x^\star$, die IntGreedyKP berechnet, einen höheren Nutzen besitzt als das wertvollste Objekt alleine. Ist dies der Fall, so wird $x^\star$ zurückgegeben, ansonsten die Lösung, die nur aus dem wertvollsten Objekt besteht. Wir gehen dabei davon aus, dass $w_i \leq t$ für alle Objekte i gilt. Objekte, für die dies nicht erfüllt ist, können ohne Beschränkung der Allgemeinheit aus der Instanz entfernt werden, da sie bereits alleine nicht in den Rucksack passen und dementsprechend in keiner gültigen Lösung vorkommen können.

ApproxKP
1 Berechne mit IntGreedyKP eine Lösung $x^\star = (x_1^\star, \ldots, x_n^\star)$.
2 $j = \arg\max_{i \in \{1,\ldots,n\}} p_i$; // Index eines Objektes mit maximalem Nutzen
3 **if** $(\sum_{i=1}^{n} p_i x_i^\star >= p_j)$ **return** $x^\star$;
4 **else return** $x' = (x'_1, \ldots, x'_n)$ mit $x'_i = \begin{cases} 0 & \text{falls } i \neq j, \\ 1 & \text{falls } i = j. \end{cases}$

Theorem 2.9. *Der Algorithmus* ApproxKP *berechnet auf jeder Eingabe für das Rucksackproblem mit n Objekten in Zeit $O(n \log n)$ eine gültige ganzzahlige Lösung, deren Nutzen mindestens halb so groß ist wie der Nutzen einer optimalen ganzzahligen Lösung.*

Beweis. Die Analyse der Laufzeit erfolgt analog zu GreedyKP. Abgesehen vom Sortieren der Objekte, welches mit Mergesort in Zeit $O(n \log n)$ durchgeführt werden kann, benötigen alle anderen Schritte zusammen nur eine lineare Laufzeit.

Wir bezeichnen für die gegebene Instanz mit OPT den Nutzen einer optimalen ganzzahligen Lösung. Zunächst leiten wir eine obere Schranke für OPT her. Dazu betrachten wir die Lösung $y \in [0,1]^n$, die GreedyKP für das Rucksackproblem mit teilbaren Objekten berechnet. Diese unterscheidet sich von der Lösung $x^\star \in \{0,1\}^n$, die IntGreedyKP berechnet, in maximal einem Element k, das in y teilweise und in $x^\star$ gar nicht ausgewählt ist. Mithilfe dieses k können wir den Nutzen der Lösung $x^\star$ als $\sum_{i=1}^{k-1} p_i$ schreiben, da diese Lösung genau die ersten $k-1$ Objekte enthält. Der Nutzen der Lösung y ist durch $\sum_{i=1}^{k} p_i$ nach oben beschränkt, da die Objekte $k+1, \ldots, n$ nicht in y enthalten sind.

Theorem 2.8 besagt, dass y eine optimale Lösung für das Rucksackproblem mit teilbaren Objekten ist. Daraus folgt $\sum_{i=1}^{n} y_i p_i \geq \text{OPT}$, denn gäbe es eine ganzzahlige

Lösung $x \in \{0,1\}^n$ mit einem höheren Nutzen als y, so wäre y nicht optimal, da x insbesondere eine Lösung für das Rucksackproblem mit teilbaren Objekten ist. Insgesamt folgt damit $\sum_{i=1}^{k} p_i \geq \text{OPT}$, da die Summe auf der linken Seite eine obere Schranke für den Nutzen von y ist.

Wir setzen nun den Nutzen der von ApproxKP berechneten Lösung und den Nutzen OPT der optimalen ganzzahligen Lösung zueinander in Beziehung. Der Nutzen der berechneten Lösung beträgt

$$\max\left\{\sum_{i=1}^{k-1} p_i, p_j\right\},$$

wobei j den Index eines Objektes mit maximalem Nutzen bezeichnet. Nutzen wir aus, dass $p_j \geq p_k$ gilt, da j per Definition ein Objekt mit maximalem Nutzen ist, so erhalten wir

$$\max\left\{\sum_{i=1}^{k-1} p_i, p_j\right\} \geq \frac{1}{2}\left(\sum_{i=1}^{k-1} p_i + p_j\right) \geq \frac{1}{2}\left(\sum_{i=1}^{k-1} p_i + p_k\right) = \frac{1}{2}\left(\sum_{i=1}^{k} p_i\right) \geq \frac{\text{OPT}}{2},$$

womit das Theorem bewiesen ist. □

Ein Algorithmus wie ApproxKP, der das Rucksackproblem nicht immer optimal löst, aber zumindest garantiert nie weniger als einen gewissen Anteil des optimalen Nutzens zu erreichen, nennt man auch *Approximationsalgorithmus*. Da der optimale Nutzen maximal doppelt so groß ist wie der erzielte, nennen wir ApproxKP auch einen 2-Approximationsalgorithmus. Wir werden im folgenden Abschnitt einen Algorithmus kennenlernen, der das Rucksackproblem optimal löst. Dieser besitzt allerdings im Allgemeinen eine wesentlich größere Laufzeit als ApproxKP.

2.3 Dynamische Programmierung

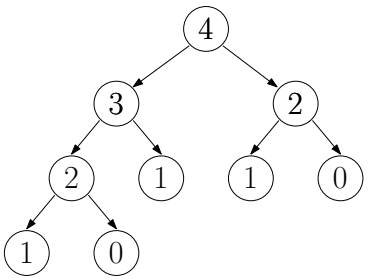
Eine weitere wichtige Technik zum Entwurf von Algorithmen ist *dynamische Programmierung*. Das Vorgehen dabei ist sehr ähnlich zu Divide-and-Conquer, denn auch bei dynamischer Programmierung wird eine gegebene Instanz zunächst in kleinere Teilinstanzen zerlegt, diese werden gelöst und anschließend werden die Lösungen der Teilinstanzen zu einer Lösung der gegebenen Instanz kombiniert. Der wesentliche Unterschied besteht darin, dass Lösungen für Teilinstanzen, die bereits berechnet sind, abgespeichert werden, damit sie nicht mehrfach berechnet werden müssen.

Wir illustrieren die Grundidee von dynamischer Programmierung an einem einfachen Beispiel, der Berechnung von *Fibonacci-Zahlen*. Für $n \in \mathbb{N}_0$ sei f_n die n-te Fibonacci-Zahl. Diese ist rekursiv wie folgt definiert:

$$f_n = \begin{cases} 0 & \text{falls } n = 0, \\ 1 & \text{falls } n = 1, \\ f_{n-1} + f_{n-2} & \text{falls } n \geq 2. \end{cases}$$

Diese Definition legt den folgenden rekursiven Algorithmus zur Berechnung der n-ten Fibonacci-Zahl nahe.

```
FibRek(int n)
 1   if (n == 0) return 0;
 2   else if (n == 1) return 1;
 3   else return FibRek(n − 1)+FibRek(n − 2);
```



Der Algorithmus ist offensichtlich korrekt, da er direkt die Definition der Fibonacci-Zahlen abbildet. Ein Blick auf den Rekursionsbaum, der oben rechts für $n = 4$ abgebildet ist, zeigt aber, dass der Algorithmus nicht besonders effizient ist. Die Leserin und der Leser überlegen sich als Übung, dass die Anzahl der rekursiven Aufrufe von FibRek zur Berechnung von f_n mindestens so groß wie f_n selbst ist und dass f_n bereits exponentiell wächst. Der Rekursionsbaum offenbart auch direkt den Grund für die Ineffizienz von FibRek: Einige Teilprobleme werden mehrfach gelöst. So erfolgen in dem obigen Beispiel zwei Aufrufe von FibRek(n) mit $n = 2$. Um den Algorithmus effizienter zu machen, sollte das Ergebnis von FibRek(2) nach dem ersten Aufruf gespeichert und beim zweiten Aufruf nicht neu berechnet werden. Bei größeren n ist der Gewinn, den man durch das Speichern einmal berechneter Ergebnisse erzielen kann, noch deutlich größer.

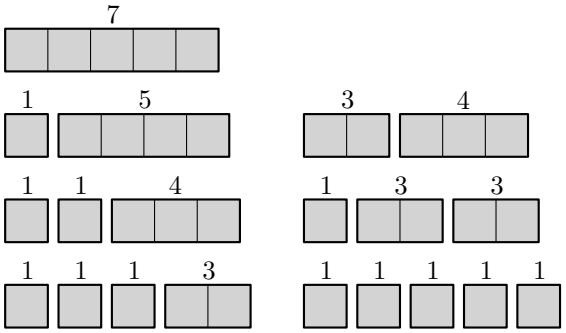
Im Unterschied zu den Divide-and-Conquer-Algorithmen, die wir in Abschnitt 2.1 kennengelernt haben, kommen im Rekursionsbaum von FibRek einige Teilprobleme mehrfach vor. Dies ist bei den Algorithmen, die wir in Abschnitt 2.1 kennengelernt haben, nicht der Fall, sodass ein Speichern der Ergebnisse der Teilprobleme dort keine Effizienzsteigerung liefert. Betrachtet man beispielsweise den Rekursionsbaum von Mergesort, so wird schnell klar, dass nie ein Teilproblem mehrfach gelöst wird.

2.3.1 Berechnung optimaler Zuschnitte

Als erstes Beispiel für dynamische Programmierung betrachten wir ein Planungsproblem, bei dem es darum geht, optimale Zuschnitte von Holzbrettern in einem Sägewerk zu berechnen. Die Eingabe für dieses *Zuschnittproblem* besteht aus einer Zahl $n \in \mathbb{N}$, die die Länge eines gegebenen Brettes angibt, und einer Tabelle, die für jede Länge $i \in \{1, \ldots, n\}$ einen Preis $p_i \in \mathbb{N}$ enthält. Dies ist der Preis, zu dem ein Brett der Länge i verkauft werden kann. Die Aufgabe besteht darin, das gegebene Brett der Länge n so in Teile zu zerlegen, dass der Gesamterlös so groß wie möglich ist. Formal suchen wir ein $\ell \in \{1, \ldots, n\}$ und eine Folge $i_1, \ldots, i_\ell \in \{1, \ldots, n\}$ von Längen mit $i_1 + \ldots + i_\ell = n$, sodass der Gesamterlös $p_{i_1} + \ldots + p_{i_\ell}$ größtmöglich ist.

In der folgenden Abbildung ist links eine Eingabe mit $n = 5$ dargestellt. Auf der rechten Seite werden alle Möglichkeiten dargestellt, wie ein Brett der Länge 5 in Teile zerlegt werden kann. Die Zahlen geben dabei den Preis der entsprechenden Bretter an. Man sieht, dass die optimale Lösung nicht eindeutig ist, denn die Lösungen (5), $(2, 3)$ und $(1, 2, 2)$ erreichen alle den optimalen Erlös von 7.

Länge i	1	2	3	4	5
Preis p_i	1	3	4	5	7



Ein algorithmischer Ansatz, um die optimale Lösung zu finden, ist es, alle möglichen Zuschnitte wie in dem obigen Beispiel zu testen. Dies ist allerdings keine gute Idee, da die Anzahl der möglichen Zuschnitte exponentiell mit n wächst. Stattdessen werden wir das Problem mittels dynamischer Programmierung lösen. Für $i \in \{1, \ldots, n\}$ bezeichnen wir im Folgenden mit R_i den maximalen Erlös, der durch Zuschneiden eines Brettes der Länge i erzielt werden kann. Die Berechnung dieser Werte R_i betrachten wir als die zu lösenden Teilprobleme. Dabei ist R_n der Wert der Lösung, die uns eigentlich interessiert. Es gilt $R_1 = p_1$, da ein Brett der Länge 1 nicht zerlegt werden kann und nur zum Preis p_1 verkauft werden kann. Außerdem definieren wir $R_0 = 0$, da dies die Beschreibung vereinfacht.

Das folgende Lemma zeigt, dass wir jedes R_i mithilfe der Lösungen R_j der kleineren Teilprobleme mit $j < i$ bestimmen können. Dies ist die zentrale Beobachtung, die wir zur Anwendung von Divide-and-Conquer und dynamischer Programmierung benötigen.

Lemma 2.10. *Für $i \in \{1, \ldots, n\}$ gilt*

$$R_i = \max\{p_j + R_{i-j} \mid j \in \{1, \ldots, i\}\}.$$

Beweis. Sei $j \in \{1, \ldots, i\}$ beliebig und sei $(i_1, \ldots, i_\ell)$ ein optimaler Zuschnitt von Brettern der Länge $i - j$. Es gelte also $i_1 + \ldots + i_\ell = i - j$ und $p_{i_1} + \ldots + p_{i_\ell} = R_{i-j}$. Dann ist $(j, i_1, \ldots, i_\ell)$ ein gültiger Zuschnitt von Brettern der Länge i, denn es gilt $j + i_1 + \ldots + i_\ell = i$. Da dieser Zuschnitt nicht besser als der optimale sein kann, gilt $R_i \geq p_j + p_{i_1} + \ldots + p_{i_\ell} = p_j + R_{i-j}$. Da dies für jedes $j \in \{1, \ldots, i\}$ der Fall ist, folgt

$$R_i \geq \max\{p_j + R_{i-j} \mid j \in \{1, \ldots, i\}\}. \tag{2.4}$$

Sei $(i_1, \ldots, i_\ell)$ nun ein optimaler Zuschnitt von Brettern der Länge i, es gelte also $i_1 + \ldots + i_\ell = i$ und $p_{i_1} + \ldots + p_{i_\ell} = R_i$. Dann ist $(i_2, \ldots, i_\ell)$ ein gültiger Zuschnitt von Brettern der Länge $i - i_1$. Da dieser nicht besser als optimal sein kann, gilt $p_{i_2} + \ldots + p_{i_\ell} \leq R_{i-i_1}$. Damit folgt

$$R_i = p_{i_1} + \ldots + p_{i_\ell} \leq p_{i_1} + R_{i-i_1} \leq \max\{p_j + R_{i-j} \mid j \in \{1, \ldots, i\}\}. \tag{2.5}$$

Aus den Ungleichungen (2.4) und (2.5) folgt zusammen direkt das Lemma. □

Intuitiv besagt Lemma 2.10, dass wir zur Berechnung des optimalen Zuschnittes eines Brettes der Länge i lediglich alle Möglichkeiten für die Länge j des ersten Brettes

im optimalen Zuschnitt testen müssen. Dieses Brett erzielt einen Erlös von p_j und das übriggebliebene Brett der Länge $i - j$ wird optimal zerlegt. Das Lemma legt den folgenden rekursiven Algorithmus zur Berechnung von R_n nahe.

```
ZuschnittRek(int i)
1   if (i == 0) return 0;
2   R = −1;
3   for (j = 1; j <= i; j++)
4       R = max{R, p_j + ZuschnittRek(i − j)};
5   return R;
```

Der initiale Aufruf ist ZuschnittRek(n). Da der Algorithmus genau Lemma 2.10 umsetzt, folgt seine Korrektheit direkt aus diesem Lemma. Betrachten wir den Rekursionsbaum von ZuschnittRek, so stellen wir genauso wie bei FibRek fest, dass er exponentiell mit n wächst, da dieselben Teilprobleme mehrfach gelöst werden. Die Leserin und der Leser sollten sich dies als Übung klar machen und bestimmen, wie genau die Anzahl der rekursiven Aufrufe mit n wächst. Auch ZuschnittRek können wir deutlich beschleunigen, indem wir einmal berechnete Ergebnisse speichern. Da bei der Berechnung von R_i nur auf R_j mit $j < i$ zurückgegriffen wird, können wir dies auch einfach iterativ realisieren wie der folgende Pseudocode zeigt.

```
Zuschnitt(int n)
1   R_0 = 0;
2   for (i = 1; i <= n; i++) {
3       R_i = −1;
4       for (j = 1; j <= i; j++)
5           R_i = max{R_i, p_j + R_{i−j}};
6   }
7   return R_n;
```

Theorem 2.11. *Der Algorithmus* Zuschnitt *berechnet in Zeit* $\Theta(n^2)$ *den maximal erreichbaren Erlös* R_n*.*

Beweis. Die Korrektheit folgt wieder direkt aus Lemma 2.10. Für die Analyse der Laufzeit rechnen wir zunächst aus, wie oft Zeile 5 ausgeführt wird. Aus den beiden verschachtelten for-Schleifen ergibt sich, dass dies

$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$$

mal passiert. Da alle anderen Zeilen höchstens so oft wie Zeile 5 ausgeführt werden, folgt daraus das Theorem. □

Der Algorithmus Zuschnitt berechnet lediglich den maximal erreichbaren Erlös R_n, nicht jedoch die optimale Zerlegung, die diesen Erlös erzielt. Implizit wird jedoch

auch eine optimale Zerlegung bestimmt. Um dies explizit zu machen, genügt es, in jedem Durchlauf der for-Schleife in einer Variablen s_i zu speichern, für welches j das Maximum angenommen wird. Dies entspricht dem folgenden Algorithmus

```
ZUSCHNITT2(int n)
 1   R_0 = 0;
 2   for (i = 1; i <= n; i++) {
 3       R_i = -1;
 4       for (j = 1; j <= i; j++) {
 5           if (p_j + R_{i-j} > R_i) {
 6               R_i = p_j + R_{i-j};
 7               s_i = j;
 8           }
 9       }
10   }
11   return R_n;
```

Die Variable s_i enthält die kleinste Länge $j \in \{1, \ldots, i\}$, für die es eine optimale Zerlegung von Brettern der Länge i mit einem Brett der Länge j gibt. Im folgenden Beispiel haben wir den Algorithmus ZUSCHNITT2 für ein Beispiel mit $n = 9$ ausgeführt.

Länge i	1	2	3	4	5	6	7	8	9
Preis p_i	1	3	5	5	9	10	10	11	13
s_i	1	2	3	1	5	1	2	3	1
R_i	1	3	5	6	9	10	12	14	15

Die optimale Zerlegung eines Brettes der Länge 9 besitzt den Wert 15. Die dazugehörige Zerlegung können wir mithilfe der Werte s_i rekonstruieren. Dazu betrachten wir zunächst den Wert $s_9 = 1$. Dieser besagt, dass es eine optimale Zerlegung von Brettern der Länge 9 gibt, in der ein Brett der Länge 1 benutzt wird. Es bleibt dann ein Brett der Länge 8, das noch optimal zerlegt werden muss. Der Wert $s_8 = 3$ besagt, dass es eine optimale Zerlegung von Brettern der Länge 8 gibt, die ein Brett der Länge 3 enthält. Es bleibt dann ein Brett der Länge 5, das noch optimal zerlegt werden muss. Der Wert $s_5 = 5$ besagt, dass es optimal ist, ein solches Brett ohne weiteres Zerlegen zu verkaufen. Dies ergibt insgesamt die Lösung $(1, 3, 5)$ mit einem Gesamterlös von $1 + 5 + 9 = 15$.

2.3.2 Rucksackproblem

Wir werden in diesem Abschnitt das Rucksackproblem mit unteilbaren Objekten mithilfe der dynamischen Programmierung lösen. Dazu überlegen wir zunächst, wie eine gegebene Instanz in kleinere Teilinstanzen zerlegt werden kann. Zunächst ist es naheliegend Teilinstanzen zu betrachten, die nur eine Teilmenge der Objekte enthalten. Das alleine reicht für unsere Zwecke aber noch nicht aus und wir gehen einen Schritt weiter.

Sei eine Instanz mit n Objekten gegeben. Wir bezeichnen wieder mit $p_1, \ldots, p_n \in \mathbb{N}$ die Nutzenwerte, mit $w_1, \ldots, w_n \in \mathbb{N}$ die Gewichte und mit $t \in \mathbb{N}$ die Kapazität des Rucksacks. Wir definieren zunächst $W = \max_{i \in \{1,\ldots,n\}} w_i$. Dann ist nW eine obere Schranke für das Gewicht jeder Lösung. Wir definieren für jede Kombination aus $i \in \{1, \ldots, n\}$ und $w \in \{0, \ldots, nW\}$ folgendes Teilproblem: finde unter allen Teilmengen der Objekte $1, \ldots, i$ mit Gewicht höchstens w die mit dem größten Nutzen. Es sei $P(i, w)$ der Nutzen dieser optimalen Auswahl von Objekten. Das bedeutet, $P(i, w)$ entspricht dem Nutzen einer optimalen Lösung für das Rucksackproblem mit Objekten $1, \ldots, i$ und Kapazität w.

Mit anderen Worten können wir die Eigenschaften von $P(i, w)$ auch wie folgt zusammenfassen: für jede Auswahl $I \subseteq \{1, \ldots, i\}$ mit $\sum_{j \in I} w_j \leq w$ gilt $\sum_{j \in I} p_j \leq P(i, w)$. Außerdem existiert eine solche Teilmenge I mit $\sum_{j \in I} p_j = P(i, w)$.

Wir werden nun das Rucksackproblem lösen, indem wir eine Tabelle konstruieren, die für jede Kombination aus $i \in \{1, \ldots, n\}$ und $w \in \{0, \ldots, nW\}$ den Wert $P(i, w)$ enthält. Wir konstruieren die Tabelle Schritt für Schritt und fangen zunächst mit den einfachen Randfällen der Form $P(1, w)$ an. Wir stellen uns hier also die Frage, welchen Nutzen wir mit einer Teilmenge von der Menge $\{1\}$ mit Gewicht höchstens w erreichen können. Ist $w < w_1$, so ist die leere Menge $\emptyset$ die einzige Teilmenge von $\{1\}$, die ein Gewicht von maximal w besitzt. Das heißt, in diesem Fall gilt $P(1, w) = 0$. Ist $w \geq w_1$, so ist die Auswahl $\{1\}$ gültig, und es gilt $P(1, w) = p_1$. Entsprechend setzen wir $P(1, w) = 0$ für $1 \leq w < w_1$ und $P(1, w) = p_1$ für $w_1 \leq w \leq nW$. Wir nutzen im Folgenden noch die Konvention $P(i, 0) = 0$ und $P(i, w) = -\infty$ für alle $i \in \{1, \ldots, n\}$ und $w < 0$.

Sei nun bereits für ein $i \geq 2$ und für alle $w \in \{0, \ldots, nW\}$ der Wert $P(i-1, w)$ bekannt. Wie können wir dann für $w \in \{0, \ldots, nW\}$ den Wert $P(i, w)$ aus den bekannten Werten berechnen? Gesucht ist eine Teilmenge $I \subseteq \{1, \ldots, i\}$ mit $\sum_{i \in I} w_i \leq w$ und größtmöglichem Nutzen. Wir unterscheiden zwei Fälle: entweder diese Teilmenge I enthält das Objekt i oder nicht.

- Falls $i \notin I$, so ist $I \subseteq \{1, \ldots, i-1\}$ mit $\sum_{j \in I} w_j \leq w$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt I maximalen Nutzen. Damit gilt $P(i, w) = P(i-1, w)$.
- Falls $i \in I$, so ist $I \setminus \{i\}$ eine Teilmenge von $\{1, \ldots, i-1\}$ mit Gewicht höchstens $w - w_i$. Unter allen Teilmengen, die diese Bedingungen erfüllen, besitzt $I \setminus \{i\}$ maximalen Nutzen. Wäre das nicht so und gäbe es eine Teilmenge $I' \subseteq \{1, \ldots, i-1\}$ mit Gewicht höchstens $w - w_i$ und größerem Nutzen als $I \setminus \{i\}$, so wäre das Gewicht von $I' \cup \{i\}$ höchstens w und der Nutzen wäre größer als der von I. Dies ist ein Widerspruch zur Wahl von I. Es gilt also $\sum_{j \in I \setminus \{i\}} p_j = P(i-1, w-w_i)$ und somit insgesamt $P(i, w) = P(i-1, w-w_i) + p_i$.

Wenn wir vor der Aufgabe stehen, $P(i, w)$ zu berechnen, dann wissen wir a priori nicht, welcher der beiden Fälle eintritt. Wir testen deshalb einfach, welche der beiden Alternativen eine Menge I mit größerem Nutzen liefert, d. h. wir setzen $P(i, w) = \max\{P(i-1, w), P(i-1, w-w_i) + p_i\}$. Um den Fall $w < w_i$, in dem der zweite

Fall $i \in I$ gar nicht eintreten kann, nicht explizit behandeln zu müssen, haben wir die Konvention $P(i, w) = -\infty$ für $i \in \{1, \ldots, n\}$ und $w < 0$ getroffen.

Ist $P(i, w)$ für alle Kombinationen von i und w bekannt, so gibt $P(n, t)$ den maximalen Nutzen an, der mit einem Rucksack der Kapazität $t \leq nW$ erreicht werden kann. Ist $t > nW$, so ist die Instanz trivial, da alle Objekte gemeinsam in den Rucksack passen. Der folgende Algorithmus fasst zusammen, wie wir Schritt für Schritt die Tabelle füllen.

```
DynKP
 1  // Sei P(i,0) = 0 und P(i,w) = −∞ für i ∈ {1,...,n} und w < 0.
 2  W = max_{i∈{1,...,n}} w_i;
 3  for (w = 1; w < w_1; w++) P(1,w) = 0;
 4  for (w = w_1; w <= nW; w++) P(1,w) = p_1;
 5  for (i = 2; i <= n; i++)
 6      for (w = 1; w <= nW; w++)
 7          P(i,w) = max{P(i−1,w), P(i−1,w−w_i) + p_i};
 8  return P(n,t);
```

In dem obigen Algorithmus würde es auch genügen, statt nW nur t als obere Grenze in den for-Schleifen zu verwenden, sofern $t \leq nW$ gilt. Mit der Grenze nW erhalten wir sogar für jede mögliche Wahl der Kapazität den Nutzen einer optimalen Lösung. Ebenso wie bei dem Algorithmus Zuschnitt2 können wir auch mithilfe von DynKP nicht nur den Nutzen einer optimalen Lösung bestimmen, sondern auch die Lösung selbst, wenn wir in Zeile 7 in entsprechenden Variablen protokollieren, von welchem der beiden Terme das Maximum angenommen wird. Die Details auszuarbeiten ist eine Übung für die Leserin und den Leser.

Theorem 2.12. *Der Algorithmus* DynKP *bestimmt in Zeit* $\Theta(n^2W)$ *den maximal erreichbaren Nutzen einer gegebenen Instanz des Rucksackproblems.*

Beweis. Die Korrektheit des Algorithmus folgt direkt aus unseren Vorüberlegungen. Formal kann man per Induktion über i zeigen, dass die Werte $P(i, w)$ genau die Bedeutung haben, die wir ihnen zugedacht haben, nämlich

$$P(i, w) = \max \left\{ \sum_{j \in I} p_j \;\middle|\; I \subseteq \{1, \ldots, i\}, \sum_{j \in I} w_j \leq w \right\}.$$

Das ist mit den Vorüberlegungen eine leichte Übung, auf die wir hier verzichten.

Die Laufzeit des Algorithmus ist auch nicht schwer zu analysieren. Die Tabelle, die berechnet wird, hat einen Eintrag für jede Kombination von $i \in \{1, \ldots, n\}$ und $w \in \{0, \ldots, nW\}$. Das sind insgesamt $\Theta(n^2W)$ Einträge und jeder davon kann in konstanter Zeit durch die Bildung eines Maximums berechnet werden. □

Beispiel für DynKP

Wir führen den Algorithmus DynKP auf der folgenden Instanz mit drei Objekten aus.

	i		
	1	2	3
p_i	2	1	3
w_i	3	2	4

Es gilt $W = 4$ und der Algorithmus DynKP berechnet die folgenden Werte $P(i, w)$.

	w												
i	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	0	2	2	2	2	2	2	2	2	2	2
2	0	0	1	2	2	3	3	3	3	3	3	3	3
3	0	0	1	2	3	3	4	5	5	6	6	6	6

Für einen Rucksack mit Kapazität 8 können wir beispielsweise an dem Eintrag $P(3, 8)$ direkt ablesen, dass ein Nutzen von 5 erreicht werden kann.

Kann der Algorithmus DynKP als effizient angesehen werden? Die Laufzeit $\Theta(n^2W)$ weist einen gravierenden Unterschied zu den Laufzeiten der Algorithmen auf, die wir bislang in der Vorlesung gesehen haben. Die Laufzeit hängt nämlich nicht nur von der Anzahl der Objekte in der Eingabe ab, sondern zusätzlich noch von dem maximalen Gewicht W. Im Gegensatz dazu hängen die Laufzeiten von den Sortieralgorithmen, die wir gesehen haben, nicht von der Größe der Zahlen ab, die zu sortieren sind, sondern lediglich von deren Anzahl. Ebenso hängen die Laufzeiten von GreedyKP und ApproxKP nur von der Anzahl der Objekte ab.

Dieser auf den ersten Blick kleine Unterschied hat große Auswirkungen. Bei den Algorithmen, die wir bisher gesehen haben, kann die Laufzeit polynomiell in der Eingabelänge beschränkt werden, wobei wir als Eingabelänge die Anzahl an Bits auffassen, die benötigt werden, um die Eingabe zu codieren. Betrachten wir hingegen beispielsweise eine Eingabe für das Rucksackproblem mit n Objekten, in der alle Gewichte, alle Nutzen und die Kapazität binär codierte Zahlen mit jeweils n Bits sind, dann beträgt die Eingabelänge $\Theta(n^2)$. Da wir mit n Bits Zahlen bis $2^n - 1$ darstellen können, gilt im Worst Case $W = 2^n - 1$. Die Laufzeit von DynKP beträgt dann $\Theta(n^2W) = \Theta(n^2 2^n)$, was exponentiell in der Eingabelänge ist.

Anders als bei den Algorithmen, die wir bisher gesehen haben, können bei DynKP also bereits Eingaben, die sich mit wenigen Bytes codieren lassen, zu einer sehr großen Laufzeit führen. Der Algorithmus DynKP löst aber zumindest alle Instanzen effizient, in denen alle Gewichte polynomiell in der Anzahl der Objekte n beschränkt sind. Instanzen, in denen $w_i \leq n^2$ für jedes i gilt, werden beispielsweise in Zeit $O(n^4)$ gelöst.

Kapitel 3

Sortieren

Die Standardbibliotheken nahezu jeder modernen Programmiersprache enthalten Methoden zum Sortieren von Objekten, sodass vermutlich die wenigsten Leserinnen und Leser nach ihrem Studium jemals selbst einen Sortieralgorithmus implementieren werden müssen. Dennoch ist es lohnenswert, sich mit solchen Algorithmen zu beschäftigen. Anhand von INSERTIONSORT und MERGESORT haben wir wichtige Konzepte und Methoden illustriert, die auch für andere Bereiche der Algorithmik nützlich und wichtig sind. In diesem Abschnitt lernen wir mit QUICKSORT einen der in der Praxis besten Sortieralgorithmen kennen, der in optimierter Form auch in der Java-Standardbibliothek implementiert ist.

Außerdem werden wir eine untere Schranke von $\Omega(n \log n)$ für die Worst-Case-Laufzeit eines jeden Sortieralgorithmus herleiten, der ausschließlich auf Vergleichen von Objekten basiert. Diese untere Schranke zeigt, dass MERGESORT bis auf konstante Faktoren ein optimaler vergleichsbasierter Sortieralgorithmus ist. Algorithmen, die ausschließlich auf Vergleichen von Objekten basieren, sind vorteilhaft, da sie nicht nur ganze Zahlen, sondern beliebige Objekte sortieren können, auf denen eine Ordnung definiert ist. Dies ist der Fall für INSERTIONSORT, MERGESORT und QUICKSORT. Zum Schluss werden wir noch sehen, dass es zum Sortieren von ganzen Zahlen Algorithmen mit linearer Laufzeit gibt. Im Gegensatz zu den anderen Sortieralgorithmen aus dieser Vorlesung basieren diese aber eben nicht ausschließlich auf Vergleichen, sondern sie inspizieren die einzelnen Ziffern der zu sortierenden Zahlen.

3.1 Quicksort

Der Algorithmus QUICKSORT gilt gemeinhin als der in der Praxis beste Sortieralgorithmus. Deshalb sollte ihn jede Informatikerin und jeder Informatiker im Laufe des Studiums gesehen haben. Bei QUICKSORT handelt es sich genauso wie bei MERGESORT um einen Divide-and-Conquer-Algorithmus. Soll ein Feld $a[0 \dots n-1]$ von Objekten (der Einfachheit halber gehen wir wieder von ganzen Zahlen aus) sortiert werden, so wählt QUICKSORT zunächst ein beliebiges Elemente $x = a[j]$ als sogenanntes *Pivotelement* aus. Für die Auswahl dieses Elementes gibt es verschiedene Strategien.

Eine Möglichkeit ist es, stets das letzte Element zu nehmen, also $j = n - 1$ zu setzen. Dann wird das Feld a mithilfe einer Methode PARTITION so permutiert, dass es ein $q \in \{0, \ldots, n-1\}$ gibt, für das die folgenden drei Eigenschaften gelten:

1. Für alle $i < q$ gilt $a[i] \leq x$.
2. Es gilt $a[q] = x$.
3. Für alle $i > q$ gilt $a[i] \geq x$.

Das Feld a wird also so permutiert, dass alle Einträge links vom Pivotelement x höchstens so groß wie x und alle Einträge rechts von x mindestens so groß wie x sind. Anschließend werden die beiden Teilfelder $a[0 \ldots q-1]$ und $a[q+1 \ldots n-1]$ unabhängig voneinander rekursiv sortiert, um ein insgesamt sortiertes Feld a zu erhalten.

Wir geben jetzt Pseudocode für den Algorithmus QUICKSORT an. Dieser bekommt als Eingabe neben dem Feld a noch zwei Indizes ℓ und r mit $0 \leq \ell \leq r \leq n-1$ übergeben. Die Aufgabe besteht darin, das Teilfeld $a[\ell \ldots r]$ zu sortieren. Der initiale Aufruf ist dementsprechend QUICKSORT$(a, 0, n-1)$.

```
QUICKSORT(int[] a, int ℓ, int r)
 1   if (ℓ < r) {
 2       int q =PARTITION(a, ℓ, r);
 3       QUICKSORT(a,ℓ,q − 1);
 4       QUICKSORT(a,q + 1,r);
 5   }
```

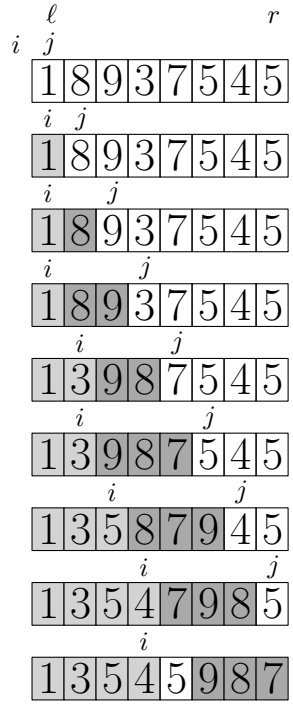
```
PARTITION(int[] a, int ℓ, int r)
 1   int x = a[r];
 2   int i = ℓ − 1;
 3   for (j = ℓ; j < r; j++) {
 4       if (a[j] <= x) {
 5           i++;
 6           vertausche a[i] und a[j];
 7       }
 8   }
 9   vertausche a[i + 1] und a[r];
10   return i + 1;
```

In dem obigen Pseudocode wird die Methode PARTITION aufgerufen. Diese erhält dieselben Parameter wie die Methode QUICKSORT und sie permutiert das Feld a so, dass die drei oben genannten Eigenschaften gelten. Der Rückgabewert von PARTITION ist die Position des Pivotelementes nach der Permutation. Er entspricht also dem Index q in der Formulierung der drei Eigenschaften.

Die Methode PARTITION kann auf viele verschiedene Arten realisiert werden. Man kann nach der Wahl des Pivotelementes x beispielsweise das Feld $a[\ell \ldots r]$ durchlaufen und dabei zwei Listen anlegen, die die Elemente kleiner gleich bzw. größer als x enthalten. Die Einträge dieser Listen könnte man anschließend nacheinander wieder in das Feld $a[\ell \ldots r]$ schreiben. Der Nachteil einer solchen Implementierung ist, dass für die Listen zusätzlicher Speicherplatz benötigt wird. Die Implementierung von PARTITION, die oben dargestellt ist, kommt hingegen ohne zusätzlichen Speicherplatz aus.

In der ersten Zeile wird das letzte Element $a[r]$ als Pivotelement x festgelegt. Die weitere Funktionsweise der Methode PARTITION ist auf den ersten Blick nicht so leicht zu

durchschauen. Zum besseren Verständnis betrachten wir deshalb zunächst ein Beispiel für die Ausführung von Partition. In der folgenden Abbildung ist der Zustand des Feldes $a[\ell \dots r]$ zu Beginn jeder Iteration der for-Schleife in Zeile 3 für ein Beispiel dargestellt. Die vorletzte Abbildung zeigt den Zustand von a zu dem Zeitpunkt, zu dem die Abbruchbedingung der for-Schleife greift. Die letzte Abbildung zeigt den Zustand von a, nachdem die Vertauschung in Zeile 9 durchgeführt wurde.



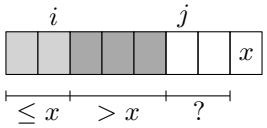
Wir haben in der obigen Abbildung einige Zellen hellgrau bzw. dunkelgrau markiert. Diese zeigen die Bereiche, für die bereits sichergestellt ist, dass sie ausschließlich Elemente kleiner gleich bzw. größer als das Pivotelement enthalten. Das folgende Lemma ist die zentrale Invariante für den Korrektheitsbeweis von Partition.

Lemma 3.1. *Am Anfang eines Schleifendurchlaufs der Methode* Partition *(Zeilen 3-8) gelten für das aktuelle j und das aktuelle i stets die folgenden Aussagen.*

1. *Für alle $k \in \{\ell, \dots, i\}$ gilt $a[k] \leq x$.*
2. *Für alle $k \in \{i+1, \dots, j-1\}$ gilt $a[k] > x$.*
3. *Es gilt $a[r] = x$.*

Die Aussagen gelten auch in dem Fall, dass die Abbruchbedingung der Schleife greift.

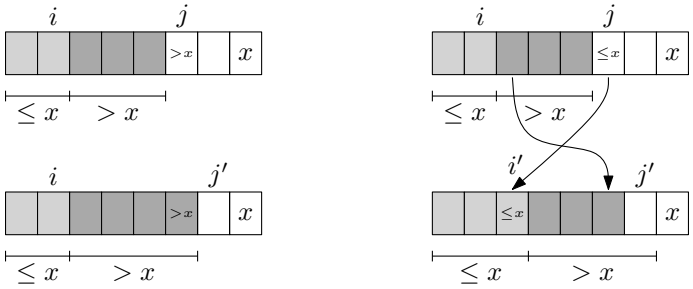
Beweis. Die folgende Abbildung veranschaulicht die Aussagen des Lemmas.



Am Anfang des ersten Schleifendurchlaufs, nach Initialisierung in Zeile 3, gilt $i = \ell - 1$ und $j = \ell$. Die erste Aussage bezieht sich somit auf alle $k \in \{\ell, \dots, \ell - 1\} = \emptyset$ und die zweite Aussage bezieht sich ebenfalls auf alle $k \in \{\ell, \dots, \ell - 1\} = \emptyset$. Damit sind die

ersten beiden Aussagen trivialerweise erfüllt. Da $x = a[r]$ in Zeile 1 gesetzt wird, gilt auch die dritte Aussage.

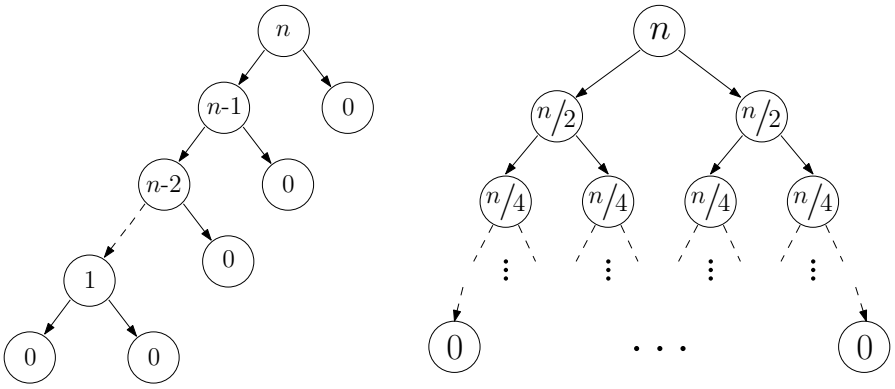
Es bleibt zu zeigen, dass die Invariante nach einem Schleifendurchlauf erhalten bleibt. Gehen wir also davon aus, dass die Invariante am Anfang eines Schleifendurchlaufs für ein entsprechendes j und i gilt. Ist $a[j] > x$, so wird innerhalb des Schleifendurchlaufs keine Veränderung an dem Feld a vorgenommen. Beim Anfang des nächsten Schleifendurchlaufs ist also lediglich der Index j um eins größer als vorher. Da die Eigenschaften 1 und 3 nicht von j abhängen, bleiben sie erhalten. Auch Eigenschaft 2 gilt weiterhin. Durch das Erhöhen von j wird der Bereich für k in Eigenschaft 2 von $\{i+1, \ldots, j-1\}$ auf $\{i+1, \ldots, j\}$ erweitert. Da wir den Fall $a[j] > x$ betrachten, gilt Eigenschaft 2 weiterhin. Dieser Fall ist in der folgenden Abbildung auf der linken Seite dargestellt.



Ist $a[j] \leq x$, so sind beim Anfang des nächsten Schleifendurchlaufs sowohl i als auch j um eins größer als vorher. Dieser Fall ist in der vorangegangenen Abbildung auf der rechten Seite dargestellt, wobei $i' = i+1$ und $j' = j+1$ definiert sei. Der Bereich für k in Eigenschaft 1 wird von $\{\ell, \ldots, i\}$ auf $\{\ell, \ldots, i+1\}$ erweitert. Durch die Vertauschung in Zeile 6 wird der Eintrag $a[j] \leq x$ an Position $i+1$ geschoben. Damit bleibt Eigenschaft 1 erhalten. Der Bereich für k in Eigenschaft 2 wird von $\{i+1, \ldots, j-1\}$ zu $\{i+2, \ldots, j\}$ geändert. An Position $a[j]$ befindet sich nach der Vertauschung in Zeile 6 das Element, das vorher an Position $i+1$ war. Da die Invariante gemäß unserer Annahme zu Beginn des Schleifendurchlaufs gilt, ist dieses Element größer als x. Damit bleibt auch Eigenschaft 2 erfüllt. Eigenschaft 3 bleibt erhalten, da $i < j < r$ gilt und der Eintrag $a[r]$ somit nicht geändert wird. □

Aus der Invariante folgt direkt die Korrektheit von Partition. Denn greift die Abbruchbedingung der for-Schleife für $j = r$, so besagt die Invariante, dass $a[k] \leq x$ für alle $k \in \{\ell, \ldots, i\}$ und $a[k] > x$ für alle $k \in \{i+1, \ldots, r-1\}$ gilt. Ferner gilt $a[r] = x$. Durch die letzte Vertauschung in Zeile 9 ergibt sich, dass Partition das Feld wie gewünscht permutiert.

Es bleibt noch die Laufzeit von Quicksort zu analysieren. Diese hängt entscheidend davon ab, welche Elemente als Pivotelemente gewählt werden. Betrachten wir als erstes den Fall, dass in jedem Schritt das größte Element des Feldes $a[\ell \ldots r]$ als Pivotelement gewählt wird. Dann ist die Aufteilung in die Teilprobleme in den Zeilen 3 und 4 von Quicksort so unausgeglichen wie möglich, denn es gilt $q = r$ und damit besitzt das erste Teilproblem, das in Zeile 3 rekursiv gelöst wird, eine Größe von $r - \ell$ und das zweite Teilproblem, das in Zeile 4 rekursiv gelöst wird, besitzt eine Größe von 0. Der Rekursionsbaum ist in der folgenden Abbildung auf der linken Seite dargestellt.



Da die Laufzeit von Partition linear in der Größe $r - \ell + 1$ des aktuellen Bereiches ist, gibt es eine Konstante c, sodass die Laufzeit von Quicksort in diesem Fall durch die folgende Formel beschrieben werden kann:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 0, \\ T(n-1) + cn & \text{falls } n > 0. \end{cases}$$

Eine leichte Übung zeigt, dass $T(n) = \Theta(n^2)$ die Lösung dieser Rekursionsgleichung ist. Ebenso kann man leicht durch Induktion beweisen, dass $\Theta(n^2)$ auch die Worst-Case-Laufzeit von Quicksort ist. Demnach ist das Schlechteste, was passieren kann, dass die beiden Teilprobleme extrem unbalanciert sind.

Ganz anders sieht die Situation aus, wenn immer der Median als Pivotelement gewählt wird. Dann haben beide Teilprobleme in etwa dieselbe Größe $\lfloor \frac{n-1}{2} \rfloor$ bzw. $\lceil \frac{n-1}{2} \rceil$. Dies ist in der Abbildung oben rechts angedeutet, wobei wir die Größen der Teilprobleme nur ungefähr angegeben haben. Also beispielsweise $n/2$ statt $\lfloor \frac{n-1}{2} \rfloor$. Dieser Rekursionsbaum erinnert stark an den von Mergesort und tatsächlich kann man mit einer Variante von Theorem 2.2 zeigen, dass Quicksort in diesem Fall eine Laufzeit von $\Theta(n \log n)$ besitzt. Hier haben wir allerdings noch nicht die Laufzeit für das Finden des Medians eingerechnet. Es gibt Algorithmen, die den Median einer Menge in linearer Zeit finden. Somit dauert das Finden des Medians asymptotisch genauso lang wie der Rest der Methode Partition, womit die Gesamtlaufzeit $\Theta(n \log n)$ tatsächlich erreicht wird. Für praktische Zwecke ist dieses Vorgehen jedoch ungeeignet, da durch das Finden des Medians in jedem Aufruf von Partition die Konstanten in der Laufzeit, von denen wir in der Θ-Notation abstrahieren, relativ groß werden.

Wählt man stets ein festes Element als Pivotelement, so kann man den Worst Case nicht ausschließen. In der obigen Methode Partition wird beispielsweise immer das letzte Element als Pivotelement gewählt. Ist die Eingabe bereits sortiert, so führt dies dazu, dass in jedem Schritt das größte Element des noch zu sortierenden Bereiches als Pivotelement gewählt wird. Eine Strategie, den Worst Case zu vermeiden, ist es, in jedem Schritt ein zufälliges Element als Pivotelement zu wählen. Diese Variante werden wir als RandomQuicksort bezeichnen. Sie kann im obigen Pseudocode einfach dadurch realisiert werden, dass zu Beginn der Methode Partition noch eine Zeile ergänzt wird, in der $a[r]$ mit $a[k]$ für ein uniform zufällig gewähltes $k \in \{\ell, \ldots, r\}$ vertauscht wird. Dann ist es unwahrscheinlich, dass ausgerechnet das größte (oder das kleinste) Element als Pivotelement gewählt wird. Dies führt in der Tat dazu, dass die

erwartete Laufzeit für jede Eingabe in der Größenordnung $\Theta(n \log n)$ liegt, was wir im Folgenden zeigen werden.

Exkurs: Wahrscheinlichkeitsrechnung

Wir werden die Analyse von RandomQuicksort so anschaulich wie möglich gestalten, ohne eine formale Einführung in die Wahrscheinlichkeitsrechnung zu geben. Leserinnen und Lesern, die an einer detaillierteren Einführung interessiert sind, seien das Buch von Michael Mitzenmacher und Eli Upfal [6] sowie das Skript zur Vorlesung randomisierte Algorithmen [11] empfohlen. Dort werden insbesondere die Aspekte der Wahrscheinlichkeitsrechnung dargestellt, die für die Algorithmik am wichtigsten sind.

Für ein Ereignis A bezeichnen wir mit $\mathbf{Pr}[A]$ die Wahrscheinlichkeit, dass es eintritt. Es gibt verschiedene Möglichkeiten, Wahrscheinlichkeiten zu interpretieren, wir wollen hier aber nur den *frequentistischen Wahrscheinlichkeitsbegriff* ansprechen. Bei diesem interpretiert man die Wahrscheinlichkeit für ein Ereignis als die relative Häufigkeit, mit der es bei einer großen Anzahl unabhängig wiederholter Zufallsexperimente auftritt. Bezeichne beispielsweise X die Augenzahl beim Wurf eines fairen Würfels, so gilt $\mathbf{Pr}[X = i] = 1/6$ für jedes $i \in \{1, 2, 3, 4, 5, 6\}$, da bei einer großen Anzahl an Würfen jede Augenzahl in etwa in einem Sechstel der Fälle auftritt. Bei RandomQuicksort wird in einer Iteration der Index k des Pivotelementes uniform zufällig aus der Menge $\{\ell, \ldots, r\}$ gewählt. Es gilt dann $\mathbf{Pr}[k = i] = 1/(r - \ell + 1)$ für alle $i \in \{\ell, \ldots, r\}$.

Eine *Zufallsvariable* X ist, grob gesprochen, eine Variable, deren Wert durch ein Zufallsexperiment bestimmt wird. Wirft man einen Würfel solange, bis er das erste Mal eine Sechs zeigt, so ist zum Beispiel die Zahl der benötigten Würfe eine Zufallsvariable. Ebenso ist auch die Laufzeit von RandomQuicksort eine Zufallsvariable. Der *Erwartungswert* $\mathbf{E}[X]$ einer Zufallsvariable X ist der Wert, den die Zufallsvariable im Durchschnitt annimmt, wenn man das Zufallsexperiment unendlich oft wiederholt. Formal lässt sich $\mathbf{E}[X]$ als die Summe über alle möglichen Werte von X darstellen, wobei jeder Wert mit der Wahrscheinlichkeit gewichtet ist, mit der er auftritt. Für eine Zufallsvariable X, die nur Werte aus $\mathbb{Z}$ annimmt, gilt dementsprechend

$$\mathbf{E}[X] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[X = i]. \tag{3.1}$$

Nimmt X nur Werte aus $\mathbb{N}_0$ an, so gilt außerdem

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i],$$

was durch eine elementare Umformung aus (3.1) folgt.

Beispiele für Erwartungswerte

- Sei X die Zufallsvariable, die den Ausgang eines fairen Würfelwurfs beschreibt. Dann gilt $\mathbf{Pr}[X = i] = 1/6$ für jedes $i \in \{1, \ldots, 6\}$ und $\mathbf{Pr}[X = i] = 0$ für jedes $i \notin \{1, \ldots, 6\}$. Damit gilt für den Erwartungswert von X

$$\mathbf{E}[X] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[X = i] = \sum_{i=1}^{6} i \cdot \mathbf{Pr}[X = i] = \sum_{i=1}^{6} \frac{i}{6} = 3{,}5.$$

Wirft man den Würfel sehr oft hintereinander, so wird man tatsächlich beobachten, dass sich der durchschnittliche Wert mehr und mehr 3,5 annähert.

- Bezeichne nun X die Zufallsvariable, die angibt, wie oft man einen fairen Würfel werfen muss, bis er das erste Mal eine Sechs zeigt. Um den Erwartungswert zu bestimmen, betrachten wir zunächst für jedes $i \in \mathbb{N}$ die Wahrscheinlichkeit, dass $X \geq i$ gilt.

  - Es gilt $\mathbf{Pr}[X \geq 1] = 1$, denn es wird in jedem Fall mindestens ein Wurf benötigt.
  - Es gilt $\mathbf{Pr}[X \geq 2] = 5/6$, denn genau dann, wenn der erste Wurf eine andere Zahl als Sechs zeigt, werden mindestens zwei Würfe benötigt. Die Wahrscheinlichkeit hierfür beträgt 5/6.
  - Es gilt $\mathbf{Pr}[X \geq 3] = (5/6)^2$, denn genau dann, wenn die ersten beiden Würfe jeweils eine andere Zahl als Sechs zeigen, werden mindestens drei Würfe benötigt. Der erste Wurf zeigt mit einer Wahrscheinlichkeit von 5/6 eine andere Zahl als Sechs, ebenso der zweite. Da beide Würfe unabhängig sind, ist die Wahrscheinlichkeit, dass beide Würfe andere Zahlen als Sechs zeigen, das Produkt der Einzelwahrscheinlichkeiten.
  - Mit einer analogen Überlegung erhält man $\mathbf{Pr}[X \geq i] = (5/6)^{i-1}$ für jedes $i \in \mathbb{N}$.

  Damit ergibt sich

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] = \sum_{i=1}^{\infty} (5/6)^{i-1} = \frac{1}{1 - 5/6} = 6.$$

  Im Durchschnitt werden also sechs Würfe benötigt, bis der Würfel das erste Mal eine Sechs zeigt.

- Eine Zufallsvariable X, die nur die Werte 0 und 1 mit positiver Wahrscheinlichkeit annimmt, nennen wir *0-1-Zufallsvariable*. Für ihren Erwartungswert gilt

$$\mathbf{E}[X] = \sum_{i=-\infty}^{\infty} i \cdot \mathbf{Pr}[X = i] = 1 \cdot \mathbf{Pr}[X = 1] = \mathbf{Pr}[X = 1]. \tag{3.2}$$

Eine wichtige Eigenschaft, die wir im Folgenden nutzen werden, ist die *Linearität des Erwartungswertes*. Diese besagt, dass für zwei beliebige Zufallsvariablen X und Y stets $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$ gilt. Dies ist sogar dann der Fall, wenn die Zufallsvariablen voneinander abhängen.

Für eine feste Eingabe ist die Laufzeit von RandomQuicksort eine Zufallsvariable. Wir sind daran interessiert, die erwartete Laufzeit (also den Erwartungswert dieser Zufallsvariable) nach oben zu beschränken. Auch bei RandomQuicksort betrachten

wir den Worst Case, d. h. für jedes n schätzen wir die erwartete Laufzeit für jede mögliche Eingabe mit n Zahlen nach oben ab. Der Worst Case bezieht sich dabei also genauso wie bei den bisher untersuchten Algorithmen auf die Wahl der schlechtesten Eingabe, nicht jedoch auf die Zufallsentscheidungen des Algorithmus.

Theorem 3.2. *Die erwartete Laufzeit von* RandomQuicksort *beträgt auf jeder Eingabe mit n Zahlen $\Theta(n \log n)$.*

Beweis. Wir analysieren, wie oft RandomQuicksort zwei Zahlen aus der Eingabe miteinander vergleicht. Dies passiert in Zeile 4 der Methode Partition, in der das jeweilige Pivotelement mit einem Eintrag des Feldes a verglichen wird. Bezeichne N die Anzahl der Ausführungen dieser Zeile, so beträgt die Laufzeit von RandomQuicksort insgesamt $\Theta(N)$. Aus diesem Grund genügt es, nur die Anzahl der Vergleiche zweier Zahlen aus der Eingabe zu beschränken.

Wir betrachten eine beliebige Eingabe $a[0 \ldots n-1]$ bestehend aus n Zahlen. Für $i \in \{1, \ldots, n\}$ bezeichne y_i die i-t größte Zahl aus der Eingabe a, d. h. $(y_1, \ldots, y_n)$ ist ein aufsteigend sortierter Vektor, der genau die Zahlen aus der Eingabe enthält. Der Einfachheit halber gehen wir davon aus, dass die Zahlen in der Eingabe paarweise verschieden sind, es gilt also $y_1 < \ldots < y_n$.

Für jedes Paar $i, j \in \{1, \ldots, n\}$ mit $i < j$ definieren wir eine Zufallsvariable $X_{ij} \in \{0, 1\}$, die genau dann den Wert 1 annimmt, wenn während der Ausführung von RandomQuicksort irgendwann die Zahlen y_i und y_j miteinander verglichen werden. Ansonsten nimmt sie den Wert 0 an.

Wir sind an dem Erwartungswert der Zufallsvariable X interessiert, die die Anzahl an Vergleichen von RandomQuicksort insgesamt beschreibt. Diese können wir als Summe über die X_{ij} schreiben, da Quicksort jeden Vergleich höchstens einmal durchführt. Dies liegt daran, dass bei einem Vergleich in Zeile 4 von Partition eines der Elemente das aktuelle Pivotelement ist, welches im nächsten rekursiven Aufruf in keinem der beiden Teilprobleme mehr enthalten ist. Das bedeutet insbesondere, dass sich das Element, mit dem das Pivotelement in Zeile 4 verglichen wird, im nächsten rekursiven Aufruf in einem Teilproblem befindet, das das Pivotelement nicht mehr enthält. Demzufolge kann derselbe Vergleich im weiteren Verlauf des Algorithmus kein zweites Mal mehr auftreten. Wir nutzen die Linearität des Erwartungswertes und erhalten

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{Pr}[X_{ij} = 1].$$

Im letzten Schritt dieser Umformung haben wir ausgenutzt, dass der Erwartungswert einer Zufallsvariable, die nur die Werte 0 und 1 annehmen kann, gemäß (3.2) der Wahrscheinlichkeit entspricht, dass sie den Wert 1 annimmt.

Wir bestimmen nun die Wahrscheinlichkeit dafür, dass y_i und y_j miteinander verglichen werden. Dazu betrachten wir die Sequenz von Pivotelementen, die im Laufe der Ausführung von RandomQuicksort gewählt werden. Uns interessiert insbesondere das erste Pivotelement x, das im Laufe des Algorithmus aus der Menge $\{y_i, y_{i+1}, \ldots, y_j\}$ gewählt wird. Genau dann, wenn $x = y_i$ oder $x = y_j$ gilt, werden die Elemente y_i und y_j

im Laufe von RandomQuicksort miteinander verglichen. Um dies zu begründen, betrachten wir die folgenden beiden Fälle.

- Sei $x \neq y_i$ und $x \neq y_j$. Da Quicksort in jedem Aufruf nur Vergleiche mit dem Pivotelement durchführt und bis zu diesem Zeitpunkt weder y_i noch y_j Pivotelement waren, wurden y_i und y_j bisher noch nicht miteinander vergleichen. In diesem Durchlauf gilt $y_i < x < y_j$. Das bedeutet, dass y_i dem linken und y_j dem rechten Teilproblem zugewiesen werden. Somit kann es auch in keinem der weiteren rekursiven Aufrufe mehr zu einem Vergleich von y_i und y_j kommen.
- Sei x entweder y_i oder y_j. Da bisher kein Element zwischen y_i und y_j als Pivotelement gewählt wurde, sind in dem rekursiven Aufruf, in dem x gewählt wird, noch beide Elemente in dem aktuellen Teilproblem enthalten. Somit findet in diesem Aufruf ein Vergleich von y_i und y_j statt.

Wir müssen also nur noch die Wahrscheinlichkeit bestimmen, dass das erste Pivotelement, das aus der Menge $\{y_i, y_{i+1}, \ldots, y_j\}$ gewählt wird, entweder y_i oder y_j ist. Solange kein Element aus dieser Menge als Pivotelement gewählt wurde, sind in jedem rekursiven Aufruf entweder noch alle Elemente dieser Menge in dem aktuellen Teilproblem enthalten oder gar keins. Daraus und aus der uniform zufälligen Wahl des Pivotelementes folgt, dass jedes Element der Menge die gleiche Wahrscheinlichkeit besitzt, als erstes Element der Menge Pivotelement zu werden. Damit folgt $\mathbf{Pr}[X_{ij} = 1] = \frac{2}{j-i+1}$. Insgesamt erhalten wir

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{Pr}[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^{n} \sum_{i=1}^{n+1-k} \frac{2}{k} = \sum_{k=2}^{n} \frac{2(n+1-k)}{k} \\
&= (n+1)\left(\sum_{k=2}^{n} \frac{2}{k}\right) - 2(n-1) \\
&= (2n+2)(H_n - 1) - 2(n-1),
\end{aligned}$$

wobei $H_n = \sum_{i=1}^{n} \frac{1}{i}$ die *n-te harmonische Zahl* bezeichnet. Nutzt man aus, dass $\ln n \leq H_n \leq \ln n + 1$ gilt, so impliziert das $\mathbf{E}[X] = \Theta(n \log n)$ und genauer

$$\begin{aligned}
\mathbf{E}[X] &= (2n+2)(H_n - 1) - 2(n-1) \\
&\leq 2n \ln n + 2 \ln n - 2n + 2 \\
&= 2n \ln n - \Theta(n).
\end{aligned}$$

Damit haben wir gezeigt, dass die Anzahl an Vergleichen (d. h. Ausführungen von Zeile 4 von Partition) im Erwartungswert $\Theta(n \log n)$ beträgt. Dies überträgt sich mit der obigen Argumentation auch auf die Laufzeit von RandomQuicksort. □

Bemerkenswert ist bei der obigen Analyse, dass die erwartete Laufzeit von RandomQuicksort für jede Eingabe der Länge n identisch ist, dass also der Best Case und der Worst Case zusammenfallen. Intuitiv ist dies allerdings nicht überraschend, denn durch die uniform zufällige Wahl der Pivotelemente spielt die Reihenfolge der Zahlen in der Eingabe keine Rolle.

3.2 Eigenschaften von Sortieralgorithmen

Abgesehen von der Laufzeit unterscheiden sich die präsentierten Sortieralgorithmen auch durch weitere Eigenschaften. Oft ist man daran interessiert, dass Sortieralgorithmen *stabil* arbeiten. Damit ist gemeint, dass die relative Ordnung gleicher Elemente beim Sortieren erhalten bleibt. Enthält das zu sortierende Feld also beispielsweise zwei Einsen, so soll diejenige, die zu Beginn als erste von beiden im Feld steht, auch nach dem Sortieren noch die erste sein. Diese Eigenschaft ist in vielen Anwendungen wichtig, in denen Objekte sortiert werden, die außer ihrem Wert noch weitere Eigenschaften besitzen. Betrachten wir als Beispiel den Fall, dass uns ein Feld von Objekten vorliegt, die jeweils einen Vor- und einen Nachnamen besitzen. Ist dieses Feld bereits alphabetisch nach Vornamen sortiert und sortieren wir es mit einem stabilen Sortieralgorithmus alphabetisch nach Nachnamen, so ist garantiert, dass zwei Personen mit demselben Nachnamen in der Reihenfolge ihrer Vornamen sortiert sind. Nutzen wir einen instabilen Sortieralgorithmus, so können sie anschließend in beliebiger Reihenfolge stehen.

Es ist leicht zu sehen, dass die Algorithmen InsertionSort und MergeSort, so wie wir sie implementiert haben, stabil sind. Unsere Implementierung von QuickSort hingegen ist nicht stabil. Die Leserin und der Leser mögen sich ein kleines Beispiel überlegen, bei dem unsere Implementierung von QuickSort die relative Ordnung zweier gleicher Objekte vertauscht.

Oft unterscheidet man Sortieralgorithmen auch danach, ob sie *in situ* (auch *in-place*) arbeiten. Ein Sortieralgorithmus arbeitet in situ, wenn er neben dem Feld a nur konstant viel Speicher benötigt. Gerade beim Sortieren von sehr großen Datenmengen sind Algorithmen, die in situ arbeiten, vorteilhaft. InsertionSort arbeitet in situ. MergeSort hingegen benötigt zusätzlichen Speicher der Größe $\Theta(n)$ für die Felder *left* und *right* in der Methode Merge. Bei QuickSort ist man zunächst versucht zu sagen, dass unsere Implementierung in situ arbeitet, da die Methode Partition keinen zusätzlichen Speicherplatz benötigt. Streng genommen ist dies aber nicht korrekt, da Speicherplatz für den Rekursionsstack benötigt wird. Im Worst Case kann dieser bis zu $\Theta(n)$ Ebenen enthalten. Bei RandomQuickSort sind es im Erwartungswert jedoch nur $\Theta(\log n)$ Ebenen, was auch für große Datenmengen noch akzeptabel ist.

3.3 Untere Schranke für die Laufzeit vergleichsbasierter Sortieralgorithmen

Regelmäßig steht man nach dem Entwurf und der Analyse eines Algorithmus vor der Frage, ob man sich mit der erreichten Laufzeit zufrieden geben soll oder ob es sich lohnt, nach einem besseren Algorithmus zu suchen. Wir kennen mit MergeSort nun beispielsweise einen Sortieralgorithmus mit einer Worst-Case-Laufzeit von $\Theta(n \log n)$. Die Frage, ob es einen schnelleren Sortieralgorithmus gibt, der vielleicht sogar eine lineare Laufzeit $\Theta(n)$ besitzt, ist naheliegend. In einer solchen Situation ist es gut, untere Schranken für die Worst-Case-Laufzeit eines jeden Algorithmus zu kennen, der

das betrachtete Problem löst. Wissen wir, dass jeder Sortieralgorithmus eine Worst-Case-Laufzeit von $\Omega(n \log n)$ besitzt, so lohnt es sich nicht, Zeit in die Suche nach asymptotisch besseren Sortieralgorithmen zu investieren. Tatsächlich hat es sich als ausgesprochen schwierig herausgestellt, solche unteren Schranken zu beweisen. Glücklicherweise ist das Sortierproblem aber eines der wenigen Probleme, für das eine untere Schranke bekannt ist.

Wir betrachten in diesem Abschnitt nur sogenannte *vergleichsbasierte Sortieralgorithmen*. Dies sind Sortieralgorithmen, die einzig durch Vergleiche von jeweils zwei Objekten Informationen gewinnen dürfen. Ein solcher Algorithmus kann in Java also beispielsweise für alle Klassen eingesetzt werden, die das Interface `Comparable` implementieren. Wir gehen bei der folgenden Argumentation der Einfachheit halber wieder davon aus, dass ein Feld $a[1 \dots n]$ von n ganzen Zahlen als Eingabe gegeben ist. Ein vergleichsbasierter Sortieralgorithmus darf sein Verhalten lediglich von Vergleichen von zwei Zahlen aus a abhängig machen, er darf aber beispielsweise nicht mit den Zahlen aus a rechnen oder die einzelnen Ziffern betrachten. Der Einfachheit halber beschäftigen wir uns in diesem Abschnitt nur mit deterministischen Algorithmen, also solchen, die keine zufälligen Entscheidungen treffen. Die untere Schranke für die Laufzeit, die wir zeigen werden, ist zwar auch für die erwartete Laufzeit von randomisierten Algorithmen gültig, um uns jedoch auf das wesentliche Argument konzentrieren zu können, lassen wir solche Algorithmen außen vor. Wir gehen ferner davon aus, dass das Feld keine Zahl mehrfach enthält, dass also eine eindeutige Sortierung der Elemente aus a existiert. Da wir eine untere Schranke zeigen wollen, ist diese Annahme ohne Beschränkung der Allgemeinheit: Können wir bereits Felder mit dieser Eigenschaft nicht schneller als in Zeit $\Omega(n \log n)$ sortieren, so können wir allgemeine Felder erst recht nicht schneller sortieren.

Sei $(\ell_1, \dots, \ell_n)$ die eindeutige Permutation der Zahlen $1, \dots, n$, für die $a[\ell_1] < \dots < a[\ell_n]$ gilt. Wir nennen $(\ell_1, \dots, \ell_n)$ den *Ordnungstyp* von a. Wir werden im Folgenden ausnutzen, dass ein Sortieralgorithmus sich auf zwei Eingaben, die verschiedene Ordnungstypen besitzen, nicht identisch verhalten darf, da er ansonsten mindestens eine der beiden Eingaben nicht korrekt sortieren würde.

Jeder vergleichsbasierte Algorithmus induziert für ein festes n einen *Entscheidungsbaum*. Ein solcher Baum abstrahiert von allen Operationen außer den Vergleichen von zwei Elementen aus dem Feld a. Jeder innere Knoten des Baumes repräsentiert einen Vergleich und er ist mit $i{:}j$ beschriftet, wenn die Elemente $a[i]$ und $a[j]$ miteinander verglichen werden. Das weitere Verhalten des Algorithmus hängt im Allgemeinen davon ab, ob $a[i] < a[j]$ oder $a[i] > a[j]$ gilt. Deshalb besitzt jeder innere Knoten genau zwei Söhne, die diese beiden Fälle repräsentieren. Für jeden Knoten v des Entscheidungsbaumes bestimmen wir alle Ordnungstypen, für die Knoten v erreicht wird, und wir beschriften jeden Knoten mit der Menge dieser Ordnungstypen. Gibt es mindestens zwei Ordnungstypen, die zu demselben Knoten gehören, so muss der Algorithmus in diesem Knoten einen weiteren Vergleich durchführen, da er sich ansonsten für diese Ordnungstypen identisch verhalten würde. Die Knoten, zu denen nur noch ein Ordnungstyp passt, bilden die Blätter des Baumes. Hat der Algorithmus einen solchen Knoten erreicht, so kann er durch die getätigten Vergleiche eindeutig den Ordnungstyp rekonstruieren und die Eingabe sortieren. In Abbildung 3.1 ist ein Beispiel für

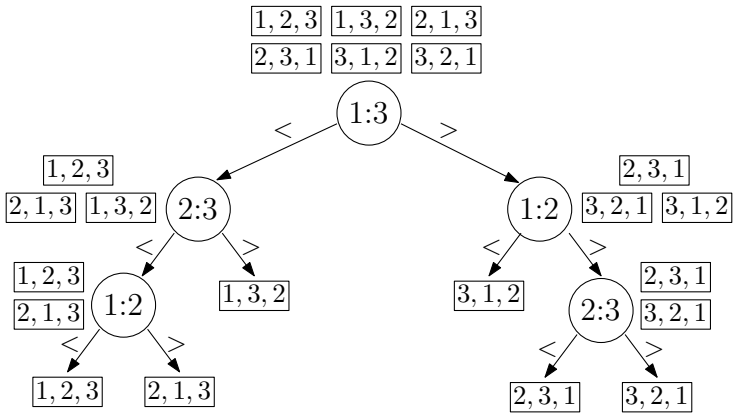


Abbildung 3.1: Ein Beispiel für einen Entscheidungsbaum: Die Wurzel ist mit allen sechs möglichen Ordnungstypen beschriftet. Als erstes wird das erste mit dem dritten Element verglichen. Ist das erste Element kleiner als das dritte, so sind nur noch die Ordnungstypen $(1, 2, 3)$, $(2, 1, 3)$ und $(1, 3, 2)$ möglich. In diesem Fall wird als nächstes das zweite mit dem dritten Element verglichen. Ist das zweite Element größer als das dritte, so bleibt insgesamt nur noch der Ordnungstyp $(1, 3, 2)$ als Möglichkeit. Ansonsten bleiben noch $(1, 2, 3)$ und $(2, 1, 3)$ als mögliche Ordnungstypen und der Algorithmus muss einen weiteren Vergleich durchführen.

einen Entscheidungsbaum zu sehen.

Wir können jeden vergleichsbasierten Sortieralgorithmus für festes n in einen Entscheidungsbaum überführen. Mithilfe dieses Entscheidungsbaumes können wir Schranken für die Anzahl der Vergleiche herleiten, die der Algorithmus durchführt. Der Best Case entspricht dem kürzesten Weg von der Wurzel zu einem der Blätter, während der Worst Case dem längsten solchen Weg entspricht. In Abbildung 3.1 ist beispielsweise zu erkennen, dass der betrachtete Algorithmus für $n = 3$ mindestens zwei und höchstens drei Vergleiche durchführen muss, um die Eingabe zu sortieren.

Theorem 3.3. *Jeder vergleichsbasierte Sortieralgorithmus benötigt zum Sortieren von Feldern der Länge n im Worst Case $\Omega(n \log n)$ Vergleiche. Damit beträgt insbesondere seine Worst-Case-Laufzeit $\Omega(n \log n)$.*

Beweis. Sei ein beliebiger vergleichsbasierter Sortieralgorithmus gegeben. Wir betrachten für ein beliebiges n den Entscheidungsbaum, den er induziert. Dieser besitzt für jeden möglichen Ordnungstyp ein Blatt. Da es $n!$ Ordnungstypen gibt, handelt es sich somit um einen Baum mit $n!$ Blättern. Es bezeichne h die Höhe des Baumes, also die Anzahl an Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter. Die Höhe h entspricht wie oben bereits beschrieben der Anzahl der Vergleiche im Worst Case.

Da jeder innere Knoten in diesem Baum den Grad 2 besitzt, besitzt der Baum höchstens 2^h Blätter. Da es $n!$ Blätter gibt, muss demnach $2^h \geq n!$ und damit $h \geq \log_2(n!)$ gelten. Das Theorem folgt, da $\log_2(n!) = \Theta(n \log n)$ gilt (der Beweis dieser Gleichung ist der Leserin und dem Leser als Übung überlassen). □

3.4 Sortieren in Linearzeit

Wir haben im vorangegangenen Abschnitt gesehen, dass jeder vergleichsbasierte Sortieralgorithmus eine Worst-Case-Laufzeit von $\Omega(n \log n)$ besitzt. In diesem Abschnitt werden wir sehen, dass man diese untere Schranke schlagen kann, wenn man die Möglichkeit besitzt, Informationen über die zu sortierenden Objekte zu gewinnen, die über bloße Vergleiche zweier Objekte hinausgehen.

Wir betrachten zunächst den Fall, dass es sich bei den zu sortierenden Objekten um Zahlen aus dem Bereich $\{0, \ldots, M-1\}$ für ein gegebenes $M \in \mathbb{N}$ handelt. Wissen wir beispielsweise, dass jede zu sortierende Zahl eine Dezimalzahl mit ℓ Stellen ist, so können wir $M = 10^\ell$ wählen. Der folgende Algorithmus MSORT sortiert Felder $a[0 \ldots n-1]$, die ausschließlich Elemente aus $\{0, \ldots, M-1\}$ enthalten, auf einfache Art und Weise. Er legt für jedes $k \in \{0, \ldots, M-1\}$ eine Liste an und fügt die Elemente aus a nach und nach in die entsprechenden Listen ein. Am Ende hängt er alle Listen aneinander.

```
MSORT(int[] a)
1  Initialisiere ein Feld der Länge M, das für jedes k ∈ {0,...,M−1}
   eine leere verkettete Liste L[k] enthält. Für jede Liste wird ein Zeiger
   auf ihr letztes Element verwaltet, sodass Elemente in konstanter Zeit
   an das Ende einer Liste eingefügt werden können.
2  for (int i = 0; i < n; i++)
3      Füge a[i] an das Ende der Liste L[a[i]] an.
4  Erzeuge ein Feld b[0...n−1] durch das Aneinanderhängen der Lis-
   ten L[0], L[1], ..., L[M−1].
5  return b;
```

Lemma 3.4. *Der Algorithmus* MSORT *sortiert n Zahlen aus der Menge $\{0, \ldots, M-1\}$ in Zeit $O(n+M)$. Ferner ist* MSORT *ein stabiler Sortieralgorithmus.*

Beweis. Die Korrektheit ergibt sich unmittelbar aus dem Algorithmus. Für die Laufzeit betrachten wir die Schritte einzeln. Schritt 1 besitzt eine Laufzeit von $O(M)$, die for-Schleife besitzt eine Laufzeit von $O(n)$ und Schritt 4 besitzt eine Laufzeit von $O(n+M)$, da sich in den M Listen insgesamt n Objekte befinden. Die Stabilität ergibt sich daraus, dass Elemente in Zeile 3 stets ans Ende der entsprechenden Liste angefügt werden. □

Ist M eine Konstante, die nicht von der Anzahl n der zu sortierenden Zahlen abhängt, so besitzt der Algorithmus MSORT eine lineare Laufzeit $O(n)$. Für große M ist der Algorithmus dennoch unpraktikabel. Man denke beispielsweise an das Sortieren von Dezimalzahlen mit 10 Stellen, für das MSORT 10^{10} Listen anlegt. Wir können den Algorithmus deutlich verbessern, indem wir die Ziffern der zu sortierenden Zahlen einzeln betrachten. Dies führt zum Algorithmus RADIXSORT, der für Felder $a[0 \ldots n-1]$ angewendet werden kann, deren Einträge Zahlen mit ℓ Ziffern in M-adischer Darstellung sind ($M = 10$ entspricht dem Dezimal- und $M = 2$ dem Binärsystem). Dabei

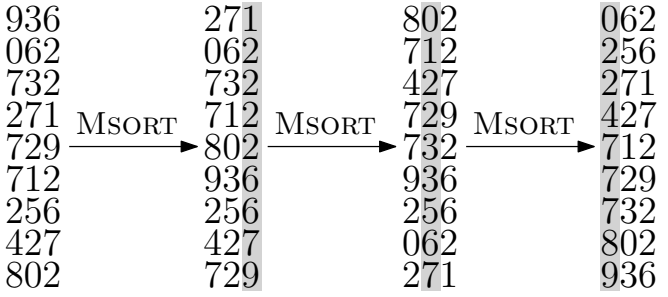


Abbildung 3.2: Ein Beispiel für die Ausführung von RADIXSORT

betrachten wir $M \in \mathbb{N}$ und $\ell \in \mathbb{N}$ als gegebene Parameter. Der Algorithmus RADIXSORT sortiert das Feld zunächst mit MSORT nach der letzten (der niedrigstwertigen) Ziffer. Anschließend sortiert er das Feld nach der vorletzten Ziffer und so weiter.

```
RADIXSORT(int[] a)
1   for (int i = ℓ; i >= 1; i--)
2       Sortiere das Feld a mit MSORT bezüglich der i-ten Ziffer.
```

Abbildung 3.2 zeigt, wie RADIXSORT ein Feld mit Dezimalzahlen mit jeweils drei Ziffern sortiert.

Theorem 3.5. *Der Algorithmus* RADIXSORT *sortiert n Zahlen mit jeweils ℓ Ziffern in M-adischer Darstellung in Zeit $O(\ell(n + M))$.*

Beweis. Die Abschätzung der Laufzeit folgt direkt aus Lemma 3.4 und der Tatsache, dass MSORT in RADIXSORT ℓ mal aufgerufen wird. Zum Beweis der Korrektheit des Algorithmus nutzen wir die folgende Invariante: In Zeile 1 von RADIXSORT ist das Feld a stets bezüglich der $\ell - i$ niedrigstwertigen Ziffern sortiert. Diese Invariante kann einfach gezeigt werden und sie basiert essentiell darauf, dass MSORT ein stabiler Sortieralgorithmus ist. Aus der Invariante folgt direkt die Korrektheit, denn zu dem Zeitpunkt, zu dem die Abbruchbedingung der for-Schleife greift, gilt $i = 0$. Das heißt, das Feld a ist bezüglich aller ℓ Ziffern sortiert. □

Für konstante Werte für M und ℓ ist die Laufzeit von RADIXSORT linear in der Anzahl n der zu sortierenden Zahlen. Für Dezimalzahlen mit jeweils 10 Ziffern können wir beispielsweise $\ell = 10$ und $M = 10$ wählen und erhalten eine viel bessere Laufzeit als mit MSORT alleine.

Kapitel 4

Dynamische Mengen

In einer *Datenstruktur* werden Daten so gespeichert, dass gewisse Operationen auf diesen Daten möglich sind. Setzt man eine Datenstruktur als Hilfsmittel in einem Algorithmus ein, so ist es für den Algorithmus nicht wichtig, wie die Daten in der Struktur intern gespeichert werden, da er nur über die eindeutig definierten Operationen auf die Daten zugreifen und sie verändern kann. Eine Datenstruktur ist also vergleichbar mit einer Klasse in der objektorientierten Programmierung, die die internen Daten kapselt und Zugriffe nur über Methoden erlaubt.

In vielen Algorithmen benötigt man Datenstrukturen, um Mengen zu verwalten, beispielsweise die Menge aller Einträge in einer Datenbank. Wir gehen davon aus, dass jedes Element x der Menge aus einem *Schlüssel* x.key und dem eigentlichen Datensatz besteht. Bei einer Literaturdatenbank könnte der Schlüssel beispielsweise die ISBN sein. Die Schlüssel stammen dabei aus einer Menge U, die wir das *Universum* nennen. Wir gehen davon aus, dass in der Menge zu keinem Zeitpunkt zwei Elemente mit demselben Schlüssel vorhanden sind. Eine Datenstruktur, die die folgenden Operationen unterstützt, nennen wir *dynamische Menge*.

- Search(k): Testet, ob ein Element mit dem Schlüssel k in der Menge vorhanden ist und gibt dieses gegebenenfalls zurück.
- Insert(x): Fügt das Element x unter dem Schlüssel x.key in die Menge ein.
- Delete(k): Löscht das Element mit dem Schlüssel k aus der Menge, sofern es existiert.

Oft ist das Universum U geordnet und dynamische Mengen unterstützen noch weitere Operationen. Insbesondere die folgenden Operationen sind für viele Anwendungen nützlich.

- Minimum: Gibt das Element der Menge mit dem kleinsten Schlüssel zurück.
- Maximum: Gibt das Element der Menge mit dem größten Schlüssel zurück.

Wir werden uns in diesem Kapitel damit beschäftigen, welche Datenstrukturen es für dynamische Mengen gibt und welche Laufzeiten die oben genannten Operationen in diesen Datenstrukturen besitzen.

4.1 Felder und Listen

Bereits aus der ersten Vorlesung über Programmierung sollten der Leserin und dem Leser verkettete Listen bekannt sein. Diese eigenen sich auf naheliegende Weise als Datenstruktur für dynamische Mengen. Wir wollen dies hier nur kurz andiskutieren, ohne jedoch noch einmal im Detail zu wiederholen, wie eine verkettete Liste implementiert werden kann.

Um die Operation SEARCH zu realisieren, muss die Liste solange Element für Element durchgegangen werden, bis der gewünschte Schlüssel gefunden oder das Ende der Liste erreicht wird. Damit besitzt SEARCH eine Laufzeit von $O(n)$, wobei n die Anzahl der Elemente bezeichnet, die bereits in der verketteten Liste gespeichert sind. Auf dieselbe Art und Weise kann auch die Operation DELETE realisiert werden. Der einzige Unterschied zu SEARCH ist, dass DELETE das Element, nachdem es gefunden wurde, durch das Umsetzen eines Zeigers aus der Liste entfernen muss. Dementsprechend beträgt auch die Laufzeit von DELETE $O(n)$. Die Operation INSERT benötigt nur eine konstante Laufzeit von $O(1)$ (sofern nicht getestet wird, ob ein Element mit demselben Schlüssel bereits in der Liste enthalten ist).

Da das Suchen und Löschen von Elementen in Listen relativ lange dauert, sind Listen nur dann als Datenstruktur für dynamische Mengen zu empfehlen, wenn sehr viele Elemente eingefügt werden, aber nur selten Elemente gesucht oder gelöscht werden.

Bei Feldern (Arrays) ergibt sich zunächst das Problem, dass wir schon bei der Initialisierung eine obere Grenze für die maximale Größe der Menge festlegen müssen. Da das Universum U in der Regel sehr groß ist (es könnte beispielsweise aus allen ASCII-Zeichenketten der Länge 30 und damit aus 128^{30} Elementen bestehen), ist es unpraktikabel, ein Feld der Länge $|U|$ anzulegen. Stattdessen kann man vorher eine Schätzung vornehmen, wie viele Elemente die dynamische Menge maximal enthalten wird, und das Feld am Anfang entsprechend initialisieren. Stellt sich dann später heraus, dass die Schätzung überschritten wird, so muss das Feld vergrößert werden, was in der Regel nur durch das Anlegen eines neuen größeren Feldes und das Kopieren der vorhandenen Elemente in dieses neue Feld realisiert werden kann. Diese Datenstruktur nennen wir ein *dynamisches Feld*.

Der folgende Pseudocode zeigt eine mögliche Implementierung einer dynamischen Menge mithilfe eines Feldes, wobei wir der Einfachheit halber davon ausgehen, dass das Universum U, aus dem die Schlüssel stammen, aus den ganzen Zahlen besteht. In dem Pseudocode bezeichnen a und n Variablen, auf die von allen Methoden aus zugegriffen werden kann. Dabei ist a das Feld, in dem die Elemente gespeichert werden, und n gibt an, wie viele Elemente bereits in a vorhanden sind. Die Methoden INIT, die zur Initialisierung der Datenstruktur dient, und SEARCH können auf die naheliegende Art und Weise implementiert werden. Der Parameter *size*, der der Methode INIT übergeben wird, ist die initiale Schätzung der maximalen Größe der Menge.

```
INIT(int size)
1   n = 0;
2   a = new Element[size];
```

```
SEARCH(int k)
1   for (int i = 0; i < n; i++)
2       if (a[i].key == k) return a[i];
3   return null;
```

Ist das Feld voll und soll ein neues Element eingefügt werden, so wird ein neues Feld erzeugt, das doppelt so groß wie das bisherige Feld ist. So ist sichergestellt, dass nicht bei jedem Einfügen ein neues Feld erzeugt werden muss. Dies ist in der folgenden Methode dargestellt.

```
INSERT(Element x)
1   if (n == a.size) {
2       Element[] b = new Element[2*a.size];
3       b[0...a.size − 1] = a[0...a.size − 1];
4       a = b;
5   }
6   a[n] = x;
7   n++;
```

Wird ein Element aus dem Feld gelöscht, so werden alle Elemente rechts davon eine Position nach links geschoben. So ist sichergestellt, dass keine Löcher entstehen. Dies ist in der folgenden Methode dargestellt.

```
DELETE(int k)
1   for (int i = 0; i < n; i++)
2       if (a[i].key == k) {
3           for (int j = i; j < n − 1; j++)
4               a[j] = a[j + 1];
5           a[n − 1] =null;
6           n−−;
7           return;
8       }
```

Die Laufzeit von SEARCH beträgt $O(n)$, da das Feld Element für Element durchsucht wird. Die Laufzeit von DELETE beträgt ebenfalls $O(n)$, da auch bei DELETE das Feld erst Element für Element durchsucht wird und anschließend noch $O(n)$ Verschiebungen durchgeführt werden. Die Laufzeit von INSERT beträgt $O(1)$, wenn das Feld groß genug ist, um ein weiteres Element aufzunehmen, und $O(n)$ sonst, da in letzterem Fall n Einträge in das neue Feld kopiert werden.

In manchen Anwendungen kann es störend sein, dass die Laufzeit von Einfügeoperationen sehr stark variiert. Ist man hingegen nur an der Gesamtlaufzeit für alle Einfügeoperationen zusammen interessiert, so ist es sinnvoll, die durchschnittliche Laufzeit pro Einfügeoperation zu betrachten. Diese nennen wir auch die *amortisierte Laufzeit*.

Lemma 4.1. *Die amortisierte Laufzeit von* Insert *beträgt in einem dynamischen Feld* $O(1)$.

Beweis. Es sei n die Anzahl der Einfügeoperationen. Werden zwischendurch Elemente gelöscht, so kann dies die Laufzeit für das Einfügen nur verringern, da es weniger Einfügeoperationen gibt, bei denen ein neues Feld angelegt werden muss. Aus diesem Grunde gehen wir davon aus, dass zwischen den Einfügeoperationen keine Elemente aus der Menge gelöscht werden.

Wir gehen im Folgenden der Einfachheit halber davon aus, dass das dynamische Feld zu Beginn mit einer Größe von 1 initialisiert wird. Der Beweis kann aber für jede andere Konstante analog geführt werden. Ist die initiale Größe 1, so ist die Größe des aktuellen Feldes stets eine Zweierpotenz. Für jedes $k \in \mathbb{N}_0$ erfolgt beim Einfügen des (2^k+1)-ten Elementes eine Vergrößerung des Feldes von 2^k auf 2^{k+1}. Die Größe des Feldes nach dem Einfügen des n-ten Elementes beträgt dementsprechend 2^ℓ für $\ell = \lceil \log_2 n \rceil$. Es erfolgen demnach ℓ Vergrößerungen des Feldes. Zusammen mit der Initialisierung und der Laufzeit aller Einfügeoperationen ergibt sich somit für die n Einfügeoperationen eine Gesamtlaufzeit von

$$O\left(\sum_{i=1}^{\ell} 2^i\right) + O(n) = O(2^{\ell+1} - 2) + O(n) = O(2^{\log_2(n)}) + O(n) = O(n).$$

Damit folgt das Lemma, da die durchschnittliche Laufzeit zum Einfügen eines Elementes $O(1)$ beträgt. □

Die Laufzeiten der Operationen sind bei einem dynamischen Feld vergleichbar mit denen bei einer verketteten Liste. In Anwendungen, in denen oft nach Elementen gesucht wird, bietet es sich an, das dynamische Feld sortiert zu halten, um eine binäre Suche zu ermöglichen. Dadurch wird erreicht, dass Search nur noch eine Laufzeit von $O(\log n)$ besitzt. Auch Delete kann mithilfe einer binären Suche etwas effizienter gestaltet werden, die Laufzeit beträgt aber wegen des Verschiebens der Elemente rechts von dem zu löschenden Element weiterhin $O(n)$. Die amortisierte Laufzeit von Insert erhöht sich nun allerdings ebenfalls auf $O(n)$, da nun auch beim Einfügen zunächst die richtige Stelle gefunden und anschließend alle Elemente rechts von dieser Stelle verschoben werden müssen, um dem neuen Element Platz zu machen.

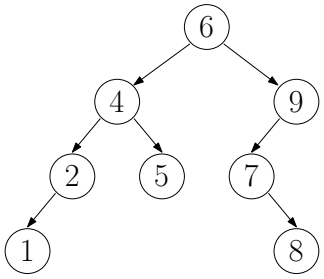
Die folgende Tabelle fasst die Laufzeiten, die wir mit Feldern und verketteten Listen erreicht haben, noch einmal zusammen.

	Search	Insert	Delete
verkettete Liste	$O(n)$	$O(1)$	$O(n)$
dynamisches Feld	$O(n)$	$O(n)$, amortisiert $O(1)$	$O(n)$
sortiertes dynamisches Feld	$O(\log n)$	$O(n)$	$O(n)$

4.2 Suchbäume

Ein *binärer Suchbaum* ist eine weitere Datenstruktur, mit der eine dynamische Menge realisiert werden kann. Die Daten werden dabei in den Knoten eines binären Baumes

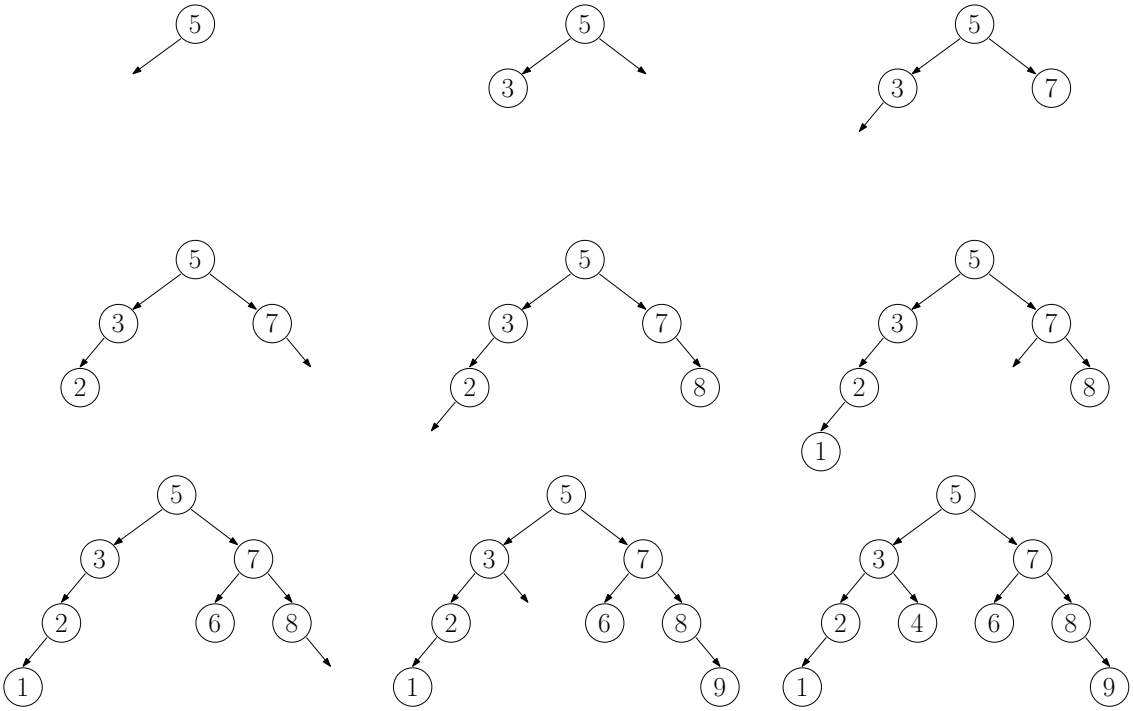
gespeichert. Die wesentliche Eigenschaft eines binären Suchbaumes ist, dass für jeden Knoten v mit Inhalt x gilt, dass alle im linken (rechten) Teilbaum von v gespeicherten Daten Schlüssel besitzen, die kleiner (größer) als der Schlüssel von x sind. Die folgende Abbildung zeigt ein Beispiel für einen binären Suchbaum.



Durch die Ordnung der Daten in einem binären Suchbaum kann die Methode SEARCH einfach realisiert werden. Man vergleicht den zu suchenden Schlüssel mit dem Schlüssel an der Wurzel. Ist dies bereits der Schlüssel, nach dem gesucht wird, so wird das Element an der Wurzel zurückgegeben. Ansonsten gibt das Ergebnis des Vergleiches vor, ob im linken oder rechten Teilbaum weitergesucht werden muss. Dies ist im folgenden Pseudocode dargestellt, der zusätzlich zum zu suchenden Schlüssel k auch noch einen Knoten v des Baumes als Eingabe erhält. Der Schlüssel k wird dann in dem Teilbaum gesucht, dessen Wurzel der Knoten v ist. SEARCH ist eine rekursive Methode und der initiale Aufruf ist SEARCH(*root*,k), wobei *root* die Wurzel des binären Suchbaumes bezeichnet. Für einen Knoten v des Baumes bezeichnet außerdem v.key den Schlüssel des Elementes, das in Knoten v gespeichert ist, und $v.left$ und $v.right$ bezeichnen das linke bzw. rechte Kind von v in dem Baum. Ist kein Element mit Schlüssel k in dem Baum vorhanden, so gibt SEARCH **null** zurück.

```
SEARCH(Node v, int k)
1   if ((v == null) || (v.key == k)) return v;
2   if (v.key > k) return SEARCH(v.left,k);
3   return SEARCH(v.right,k);
```

Um ein neues Element in den Baum einzufügen, wird ein neues Blatt erzeugt. Ist der Baum noch leer, so ist das neue Blatt gleichzeitig die Wurzel und der einzige Knoten des Baumes. Ansonsten wird zunächst nach dem Schlüssel des einzufügenden Elementes in dem Baum gesucht. Ist bereits ein Element mit demselben Schlüssel im Baum vorhanden, so wird es mit dem neu einzufügenden Element überschrieben. Ansonsten erreicht die Suche einen Null-Zeiger. Anschaulich gesprochen wird das neue Blatt an genau diese Stelle im Baum gehängt wo die Suche diesen Null-Zeiger findet, das heißt unter den entsprechenden Elternknoten. Die genaue Implementierung überlassen wir der Leserin und dem Leser als Übung. Wir zeigen stattdessen als Beispiel, was passiert, wenn nacheinander die Elemente 5, 3, 7, 2, 8, 1, 6, 9 und 4 in einen anfangs leeren Baum eingefügt werden. Dabei ist in jedem Bild der Null-Zeiger eingezeichnet, den die Suche hervorbringt, und an dessen Stelle das neue Blatt im nächsten Schritt gehängt wird.



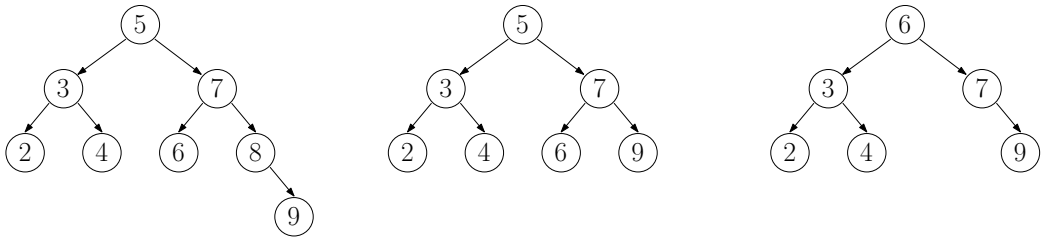
Um die Operation DELETE(k) zu realisieren, suchen wir zunächst mit SEARCH nach einem Knoten mit Schlüssel k in dem Baum. Ist ein solcher Knoten v vorhanden, so unterscheiden wir drei Fälle.

1. Ist v ein Blatt, so kann es einfach aus dem Baum entfernt werden, indem der Zeiger, der auf v zeigt, auf **null** gesetzt wird.

2. Besitzt v genau ein Kind w, so wird v aus dem Baum gelöscht, indem der Zeiger, der auf v zeigt, auf w gesetzt wird.

3. Besitzt v zwei Kinder w_1 und w_2, so suchen wir im rechten Teilbaum von v den Knoten u mit dem kleinsten Schlüssel, indem wir zunächst zum rechten Kind von v und anschließend stets zum linken Kind gehen, solange bis wir einen Knoten erreicht haben, der kein linkes Kind besitzt. Dann tauschen wir die Daten in v und u und löschen u. Da u kein linkes Kind besitzt, führt uns dies in einen der ersten beiden Fälle.

Auch hier überlassen wir die Implementierungsdetails dem Leser als Übung. Wir illustrieren die drei Fälle im Folgenden an einem Beispiel. Wir löschen aus dem Baum, den wir oben aufgebaut haben, die Element 1, 8 und 5. Dies entspricht den Fällen 1, 2 und 3.



Lemma 4.2. *Wird aus einem binären Suchbaum ein Element mit der obigen Methode* DELETE *gelöscht, so ist der resultierende Baum wieder ein binärer Suchbaum.*

Beweis. Im ersten Fall ist klar, dass die Ordnungseigenschaft des Suchbaumes erhalten bleibt, da lediglich ein Blatt gelöscht wird. Im zweiten Fall gilt für alle Knoten x und y des Suchbaumes, dass x nach dem Löschen des Knotens v genau dann in dem linken (rechten) Teilbaum von y liegt, wenn dies bereits vor dem Löschen der Fall war. Damit folgt die Ordnungseigenschaft nach dem Löschen aus der Ordnungseigenschaft vor dem Löschen. Die Korrektheit des dritten Falles folgt zusätzlich daraus, dass das Element, das in den Knoten v verschoben wird, einen größeren Schlüssel besitzt als alle Elemente im linken Teilbaum von v und einen kleineren Schlüssel als alle Elemente im rechten Teilbaum von v. □

Theorem 4.3. *Die Laufzeit der Operationen* SEARCH*,* INSERT *und* DELETE *in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.*

Beweis. Die Knoten, die von der Methode SEARCH besucht werden, bilden einen Pfad, der an der Wurzel beginnt und in jedem Schritt eine Ebene weiter nach unten geht. Somit werden von SEARCH maximal $h+1$ viele Knoten besucht, woraus sich die Laufzeit von $O(h+1)$ direkt ergibt. Dasselbe Argument kann auch auf die Methode INSERT angewendet wird, weil diese auch zunächst eine Suche durchführt und anschließend mit konstant vielen Operationen ein neues Blatt in den Baum einfügt.

Auch die Methode DELETE besitzt eine Laufzeit von $O(h+1)$. Zunächst wird das zu löschende Element in Zeit $O(h+1)$ mittels SEARCH gesucht. Anschließend sind in den Fällen 1 und 2 nur konstant viele Operationen zum Löschen notwendig. In Fall 3 wird ausgehend von dem zu löschenden Element ein weiterer Pfad in dem Baum konstruiert, um das Element mit dem kleinsten Schlüssel im rechten Teilbaum zu finden. Auch dieser Pfad enthält maximal $h+1$ viele Knoten. Anschließend kann noch einmal Fall 2 eintreten welcher aber, wie eben diskutiert, nur konstant viele Operationen benötigt. Insgesamt beträgt die Laufzeit von DELETE damit $O(h+1)$. □

Die Laufzeit, die wir in Theorem 4.3 angegeben haben, kann nicht direkt mit den Laufzeiten verglichen werden, die wir im vorangegangenen Abschnitt für dynamische Felder und verkettete Listen gezeigt haben, da sie nicht direkt von der Anzahl n der Elemente, sondern der Höhe des Baumes abhängt. Zunächst überlegen wir uns, wie groß die Höhe eines binären Suchbaumes mindestens ist.

Theorem 4.4. *Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.*

Beweis. Wir nummerieren die Ebenen des Baumes von 0 bis h. Das heißt, die Wurzel befindet sich auf Ebene 0, ihre Kinder auf Ebene 1 und so weiter. Dann gibt es für jedes $i \in \{0, 1, \ldots, h\}$ höchstens 2^i Knoten auf Ebene i. Ein Baum der Höhe h kann also insgesamt höchstens

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

Knoten enthalten. Da der Baum n Knoten enthält muss $n \leq 2^{h+1} - 1$ gelten. Daraus folgt $h \geq \log_2(n+1) - 1$. Da h eine ganze Zahl ist, gilt diese Ungleichung immer noch, wenn die rechte Seite aufgerundet wird. □

Aus dem obigen Argument folgt nicht nur, dass jeder binäre Suchbaum mit n Elementen mindestens die Höhe $\Omega(\log n)$ besitzt, sondern es folgt auch, dass ein binärer Baum mit n Elementen, der bis auf eventuell die letzte Ebene vollständig ist (d. h. jeder Knoten des Baumes, der sich nicht auf der vorletzten oder letzten Ebene befindet, besitzt genau zwei Kinder), die Höhe $\Theta(\log n)$ besitzt. In einem solchen Baum können demnach alle relevanten Operationen in logarithmischer Zeit ausgeführt werden. Leider ist im Allgemeinen jedoch nicht garantiert, dass binäre Suchbäume vollständig sind. Werden beispielsweise n Daten in sortierter Reihenfolge in einen anfangs leeren Suchbaum eingefügt, so entartet der Baum zu einer Liste mit Höhe $n - 1$. Im Worst Case benötigen die Operationen also sogar eine Laufzeit von $\Omega(n)$.

4.2.1 AVL-Bäume

Die Diskussion am Ende des vorangegangenen Abschnittes legt es nahe, die Operationen INSERT und DELETE so zu implementieren, dass der Suchbaum zu jedem Zeitpunkt möglichst vollständig ist. Dazu muss der Suchbaum nach dem Einfügen oder Löschen eines Knotens eventuell umgebaut oder *balanciert* werden. Dies sollte natürlich nicht zu lange dauern, damit die Laufzeiten von INSERT und DELETE nicht zu groß werden. Es ist nicht effizient möglich, den Baum bis auf die letzte Ebene vollständig zu halten. Stattdessen ist es aber möglich, relativ effizient zu jedem Zeitpunkt die folgende *AVL-Eigenschaft* zu garantieren.

Definition 4.5. *Ein binärer Suchbaum heißt* AVL-Baum *(nach Adelson-Velskii und Landis, die diese Art von Suchbaum 1962 vorgeschlagen haben), wenn für jeden Knoten v des Baumes gilt, dass sich die Höhen des linken und des rechten Teilbaumes um maximal 1 unterscheiden. Seien T_L und T_R die Teilbäume von v und seien $h(T_L)$ und $h(T_R)$ ihre Höhen. Dann bezeichne* $\text{bal}(v) = h(T_L) - h(T_R)$ *die* Balance *von Knoten v. In AVL-Bäumen gilt* $\text{bal}(v) \in \{-1, 0, 1\}$ *für jeden Knoten v.*

Bevor wir uns damit beschäftigen, wie die AVL-Eigenschaft nach jedem Einfügen und Löschen eines Knotens sichergestellt werden kann, zeigen wir zunächst, dass die Höhe eines jeden AVL-Baumes logarithmisch in der Anzahl der Knoten ist.

Theorem 4.6. *Ein AVL-Baum mit $n \in \mathbb{N}$ Knoten besitzt eine Höhe von mindestens $\lceil \log_2(n+1) - 1 \rceil$ und höchstens $1{,}4405 \log_2(n+1)$.*

Beweis. Die untere Schranke gilt sogar für alle binären Suchbäume und folgt direkt aus Theorem 4.4. Zum Beweis der oberen Schranke bezeichne $A(h)$ die geringste Anzahl an Knoten, die ein AVL-Baum der Höhe h haben kann. Ein Baum mit Höhe 0 besteht nur aus der Wurzel (leere Bäume ohne Knoten sind für diesen Beweis nicht von Interesse, wir definieren ihre Höhe als -1). Dementsprechend gilt $A(0) = 1$. Man sieht leicht,

dass $A(1) = 2$ gilt, da jeder Baum der Höhe 1 mindestens zwei Knoten enthalten muss und die AVL-Eigenschaft erfüllt.

Sei ein AVL-Baum der Höhe $h \geq 2$ gegeben. Die beiden Teilbäume T_L und T_R links bzw. rechts von der Wurzel sind für sich genommen ebenfalls AVL-Bäume. Da der gesamte Baum eine Höhe von h besitzt, muss einer dieser beiden Teilbäume eine Höhe von genau $h-1$ besitzen. Damit die AVL-Eigenschaft für die Wurzel erfüllt ist, muss der andere Teilbaum eine Höhe von mindestens $h-2$ besitzen. Dementsprechend gilt für $h \geq 2$ die Rekursionsformel

$$A(h) = 1 + A(h-1) + A(h-2). \tag{4.1}$$

Die ersten Werte der Folge $A(h)$ lauten 1, 2, 4, 7, 12, 20, 33, ... Vergleichen wir dies mit der Fibonacci-Folge aus Abschnitt 2.3, die mit den Zahlen 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... beginnt, so liegt die Vermutung nahe, dass $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$ gilt. Dies kann leicht mithilfe von Induktion nachgewiesen werden. Der Induktionsanfang für $h \in \{0, 1\}$ folgt direkt aus den oben angegebenen Werten. Der Induktionsschritt für $h \geq 2$ folgt aus

$$A(h) = 1 + A(h-1) + A(h-2) = 1 + f_{h+2} - 1 + f_{h+1} - 1 = f_{h+3} - 1,$$

wobei wir im ersten Schritt die Rekursionsformel (4.1), im zweiten die Induktionsannahme und im dritten die Definition der Fibonacci-Zahlen eingesetzt haben.

Für die Fibonacci-Zahlen gibt es eine explizite Formel. Für $h \geq 0$ gilt

$$f_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right].$$

Diese Formel nachzuweisen, ist eine gute Übung für die Leserin und den Leser, um noch einmal vollständige Induktion zu üben.

Aus der Definition von A folgt, dass $n \geq A(h)$ für jeden AVL-Baum mit n Knoten und Höhe h gilt. Demnach gilt

$$n + 1 \geq A(h) + 1 = f_{h+3} = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right].$$

Aus $\left| \frac{1-\sqrt{5}}{2} \right| < 1$ folgt

$$n + 1 \geq \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right].$$

Durch Umformen ergibt sich hieraus

$$\begin{aligned} h + 3 &\leq \frac{\log_2(\sqrt{5}(n+1)+1)}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} \\ &< 1{,}4405 \cdot \log_2(\sqrt{5}(n+1)+1) \end{aligned}$$

$$\begin{aligned}
&\leq 1{,}4405 \cdot \log_2((\sqrt{5}+1)(n+1)) \\
&= 1{,}4405 \cdot \log_2(n+1) + 1{,}4405 \cdot \log_2(\sqrt{5}+1) \\
&< 1{,}4405 \cdot \log_2(n+1) + 2{,}5.
\end{aligned}$$

Daraus folgt

$$h < 1{,}4405 \cdot \log_2(n+1),$$

womit das Theorem bewiesen ist. □

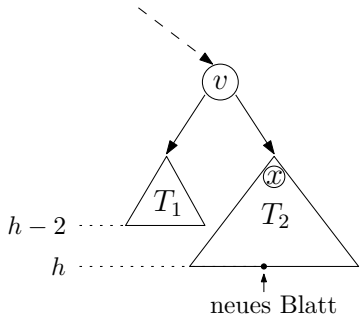
Wir beschreiben nun, wie die Operationen INSERT und DELETE so gestaltet werden können, dass sie die AVL-Eigenschaft zu jedem Zeitpunkt aufrecht erhalten. In einem AVL-Baum speichern wir in jedem Knoten v zusätzlich zu dem Element, das er enthält, auch seine aktuelle Balance $\text{bal}(v)$. Wir müssen deshalb insbesondere beschreiben, wie diese Werte nach dem Einfügen oder Löschen eines Knotens aktualisiert werden.

Beim Aufruf von INSERT wird das neue Element zunächst genauso wie in einen normalen binären Suchbaum eingefügt. Anschließend müssen die Balance-Werte aktualisiert werden. Gleichzeitig muss getestet werden, ob durch das Einfügen die AVL-Eigenschaft zerstört wurde. Ist dies der Fall, so muss der Baum *rebalanciert* werden. Dazu gehen wir ausgehend von dem neu eingefügten Blatt Knoten für Knoten nach oben in Richtung der Wurzel. Nur für Knoten, die auf dem so erzeugten Pfad liegen, können sich die Balancen geändert haben. Für alle anderen Knoten in dem Baum hat sich die Balance durch das Einfügen nicht geändert. Diese erfüllen also weiterhin die AVL-Eigenschaft.

Wir betrachten nur den Fall, dass wir auf dem Pfad zur Wurzel einen Knoten v von seinem rechten Kind aus erreichen. Den Fall, dass wir den Knoten v von seinem linken Kind aus erreichen, werden wir nicht diskutieren, da er analog behandelt werden kann. Wir gehen davon aus, dass für alle Knoten auf dem Pfad unterhalb von v die AVL-Eigenschaft bereits wieder hergestellt wurde, und beschreiben nun, wie sie an v wiederhergestellt werden kann, sofern sie verletzt ist. Ist die Höhe des rechten Teilbaumes von v durch das Einfügen des neuen Blattes nicht größer geworden, so kann die Rebalancierung abgeschlossen werden, da sich durch das Einfügen weder die Balance von v noch die von den Knoten oberhalb von v geändert hat. Damit erfüllen diese Knoten immer noch die AVL-Eigenschaft. Wir gehen also im Folgenden davon aus, dass durch das Einfügen des neuen Blattes die Höhe des rechten Teilbaumes von v um eins größer geworden ist. Wir unterscheiden die folgenden Fälle, wobei $\text{bal}(v)$ die in Knoten v gespeicherte Balance bezeichnet. Es handelt sich bei $\text{bal}(v)$ also um die Balance vor dem Einfügen des neuen Blattes.

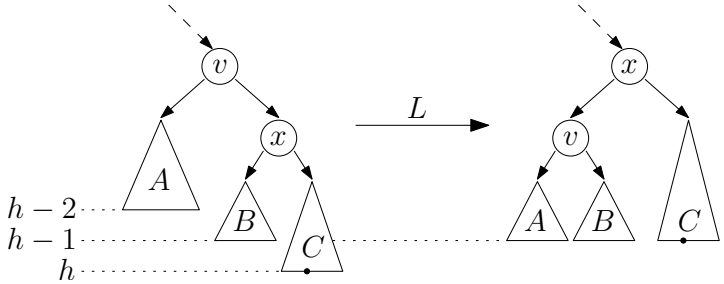
- $\text{bal}(v) = 1$: Wir korrigieren die Balance von v und setzen $\text{bal}(v) = 0$. In diesem Fall hat sich die Höhe des Teilbaumes mit Wurzel v nicht geändert. Damit bleiben die Balancen aller Knoten oberhalb von v unverändert, weshalb die Rebalancierung beendet werden kann.
- $\text{bal}(v) = 0$: Wir korrigieren die Balance von v und setzen $\text{bal}(v) = -1$. In diesem Fall ist die Höhe des Teilbaumes mit Wurzel v um eins größer geworden. Ist v die Wurzel des Baumes, dann ist die Rebalancierung abgeschlossen. Ansonsten gehen wir zum Elternknoten von v und setzen die Rebalancierung dort fort.

- $\text{bal}(v) = -1$: Der Knoten ist nun außer Balance, denn die aktuelle Balance von v beträgt -2. Diese Situation ist in der folgenden Abbildung dargestellt.



Im Folgenden bezeichne x das rechte Kind von v. Dieses muss existieren, da die aktuelle Balance von v negativ ist und der rechte Teilbaum von v somit nicht leer sein kann.

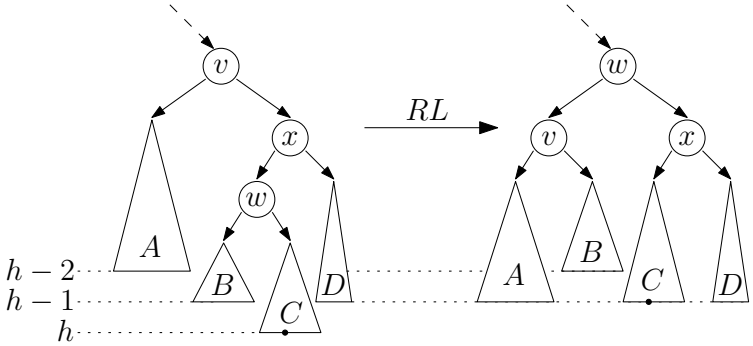
– 1. Unterfall: Die Höhe des rechten Teilbaumes von x ist durch das Einfügen um eins größer geworden. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, wird eine *Linksrotation* durchgeführt. Diese ist in der folgenden Abbildung dargestellt.



Die Linksrotation erhält die Suchbaumeigenschaft. Die Rebalancierung ist abgeschlossen, da die Höhe des betrachteten Teilbaumes nach der Linksrotation wieder genauso groß ist wie vor dem Einfügen und sich die Balancen der Knoten oberhalb dieses Teilbaumes somit nicht geändert haben.

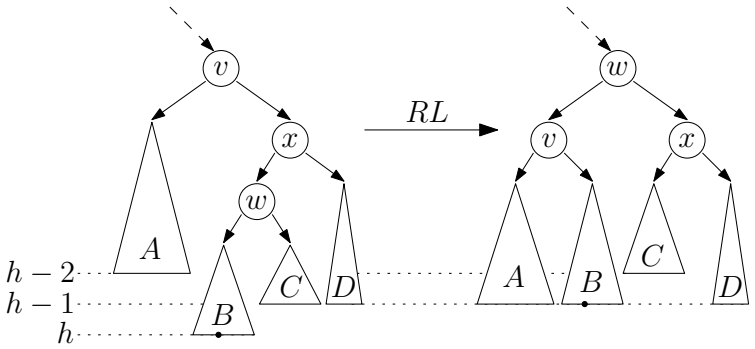
In den folgenden Unterfällen bezeichne w das linke Kind von x.

– 2. Unterfall: Die Höhen des linken Teilbaumes von x sowie des rechten Teilbaumes von w sind durch das Einfügen jeweils um eins größer geworden. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, genügt eine *Doppelrotation* (eine RL-Rotation, zunächst rechts an x und dann links an v) wie in der folgenden Abbildung dargestellt.



Die Rebalancierung wird beendet, da die Höhe des betrachteten Teilbaumes nach der Doppelrotation wieder genauso groß ist wie vor dem Einfügen und sich die Balancen der Knoten oberhalb dieses Teilbaumes somit nicht geändert haben. Man beachte, dass der Teilbaum B in der obigen linken Abbildung nicht auf Höhe $h-2$ enden kann, da sonst der Knoten w außer Balance wäre. Wir gehen jedoch davon aus, dass alle Knoten unterhalb von v die AVL-Eigenschaft (wieder) erfüllen.

– 3. Unterfall: Die Höhen des linken Teilbaumes von x sowie des linken Teilbaumes von w sind durch das Einfügen jeweils um eins größer geworden. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, genügt eine *Doppelrotation* (eine RL-Rotation, zunächst rechts an x und dann links an v) wie in der folgenden Abbildung dargestellt.

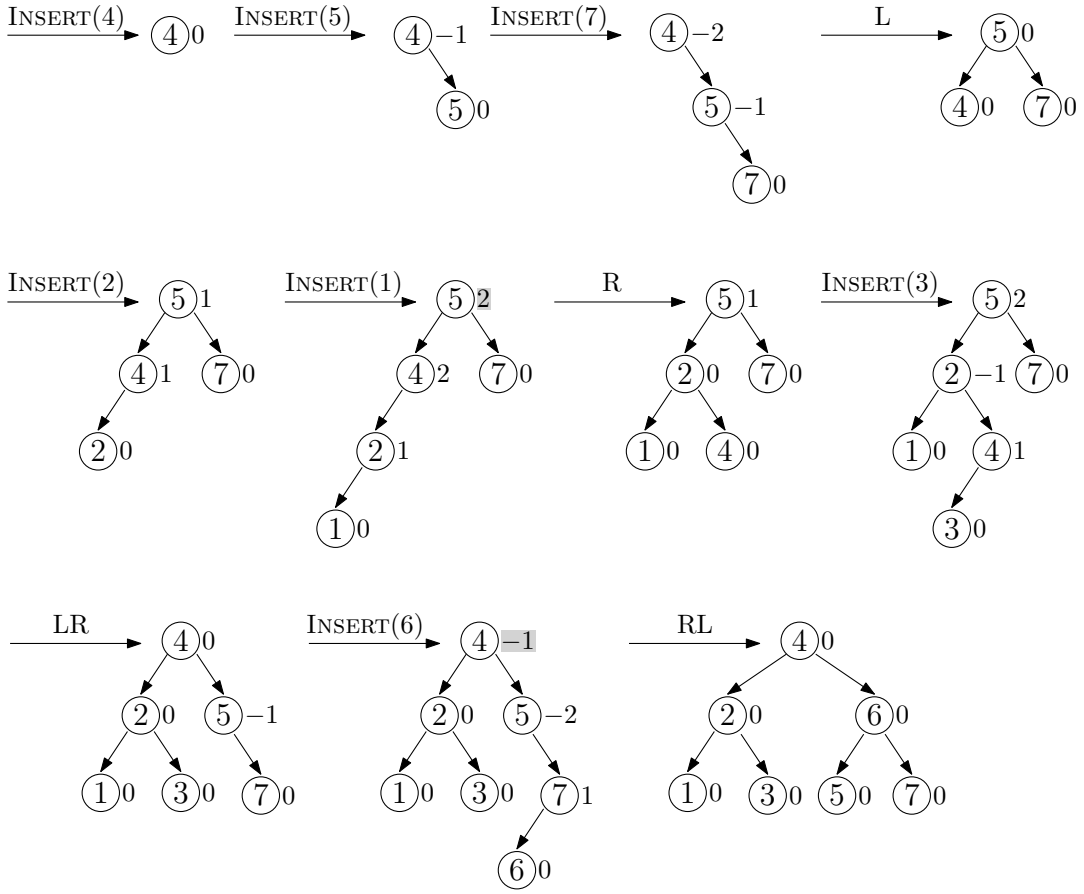


Die Rebalancierung wird beendet, da die Höhe des betrachteten Teilbaumes nach der Doppelrotation wieder genauso groß ist wie vor dem Einfügen und sich die Balancen der Knoten oberhalb dieses Teilbaumes somit nicht geändert haben.

– 4. Unterfall: Die Höhe des linken Teilbaumes von x ist durch das Einfügen um eins größer geworden und w besitzt kein Kind. Dann muss w der neu eingefügte Knoten sein. Dieser Unterfall kann nur erreicht werden, wenn die vier Teilbäume A, B, C und D leer sind. Dann gilt (analog zu der Abbildung im 3. Unterfall) nach einer Doppelrotation $\text{bal}(w) = \text{bal}(v) = \text{bal}(x) = 0$. Die Rebalancierung wird beendet, da die Höhe des betrachteten Teilbaumes nach der Doppelrotation wieder genauso groß ist wie vor dem Einfügen und sich die Balancen der Knoten oberhalb dieses Teilbaumes somit nicht geändert haben.

Die Rebalancierung kann bis zur Wurzel des Baumes propagiert werden, wenn stets der Fall $\text{bal}(v) = 0$ eintritt. Sobald jedoch einmal der Fall $\text{bal}(v) = -1$ eintritt und eine Rotation oder Doppelrotation durchgeführt wird, ist die Rebalancierung beendet. Wird der Knoten v von seinem linken Kind aus erreicht, so verläuft die Fallunterscheidung analog. Dabei werden Linksrotationen zu Rechtsrotationen und RL-Doppelrotationen zu LR-Doppelrotationen. Da pro erreichtem Knoten nur konstant viele Schritte notwendig sind (insbesondere für die Rotationen und Doppelrotationen) dauert das Einfügen eines neuen Elementes in einen AVL-Baum $O(h)$, wobei h die Höhe des Baumes bezeichnet.

Die folgende Abbildung zeigt ein Beispiel, in dem die Zahlen 4, 5, 7, 2, 1, 3, und 6 in dieser Reihenfolge in einen anfangs leeren AVL-Baum eingefügt werden. Neben jedem Knoten ist seine Balance gezeigt. Die beiden grau hinterlegten Balancen werden beim Einfügen tatsächlich gar nicht gesetzt, da die entsprechenden Knoten bei der Rebalancierung nicht erreicht werden. Im Falle der ersten grau hinterlegten Balance (nach dem Einfügen von 1) wird der Wert während des Einfügens beispielsweise auf 1 gelassen, was vor dem Einfügen und nach der Rotation auch der korrekte Wert ist.



Auch das Löschen eines Knotens verläuft zunächst genauso wie das Löschen in einem normalen binären Suchbaum. Anschließend muss auch hier getestet werden, ob die AVL-Eigenschaft noch gilt, und es muss gegebenenfalls eine Rebalancierung durchgeführt werden. Wieder gehen wir ausgehend von dem gelöschten Knoten nach oben in Richtung der Wurzel. Nur für Knoten, die auf dem so erzeugten Pfad liegen, können sich die Balancen geändert haben. Für alle anderen Knoten in dem Baum hat sich die Balance durch das Löschen nicht geändert. Diese erfüllen also weiterhin die AVL-Eigenschaft.

Wir betrachten nur den Fall, dass wir auf dem Pfad zur Wurzel einen Knoten v von seinem linken Kind aus erreichen. Den Fall, dass wir den Knoten v von seinem rechten Kind aus erreichen, werden wir nicht diskutieren, da er analog behandelt werden kann. Wir gehen davon aus, dass für alle Knoten auf dem Pfad unterhalb von v die AVL-Eigenschaft bereits wieder hergestellt wurde, und beschreiben nun, wie sie an v

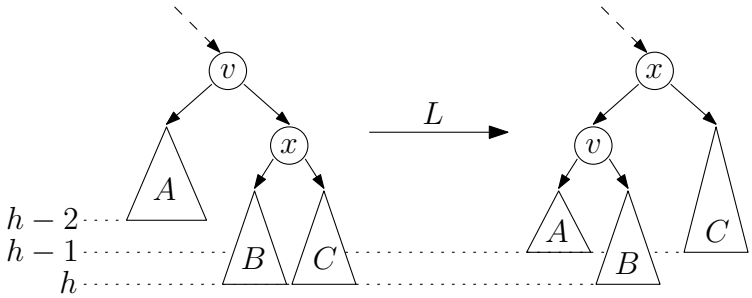
wiederhergestellt werden kann, sofern sie verletzt ist. Ist die Höhe des linken Teilbaumes von v durch das Löschen nicht kleiner geworden, so kann die Rebalancierung abgebrochen werden, da sich durch das Löschen weder die Balance von v noch die von den Knoten oberhalb von v geändert hat. Damit erfüllen diese Knoten immer noch die AVL-Eigenschaft. Wir gehen also im Folgenden davon aus, dass durch das Löschen des Knotens die Höhe des linken Teilbaumes von v um eins kleiner geworden ist. Wir unterscheiden die folgenden Fälle, wobei $\text{bal}(v)$ die in Knoten v gespeicherte Balance bezeichnet. Es handelt sich bei $\text{bal}(v)$ also um die Balance vor dem Löschen des Knotens.

- $\text{bal}(v) = 1$: Wir korrigieren die Balance von v und setzen $\text{bal}(v) = 0$. Ist v die Wurzel des Baumes, dann ist die Rebalancierung abgeschlossen. Ansonsten gehen wir zum Elternknoten von v und setzen die Rebalancierung dort fort.

- $\text{bal}(v) = 0$: Wir korrigieren die Balance von v und setzen $\text{bal}(v) = -1$. In diesem Fall hat sich die Höhe des Teilbaumes mit Wurzel v nicht geändert. Damit bleiben auch die Balancen aller Knoten oberhalb von v unverändert, weshalb die Rebalancierung beendet werden kann.

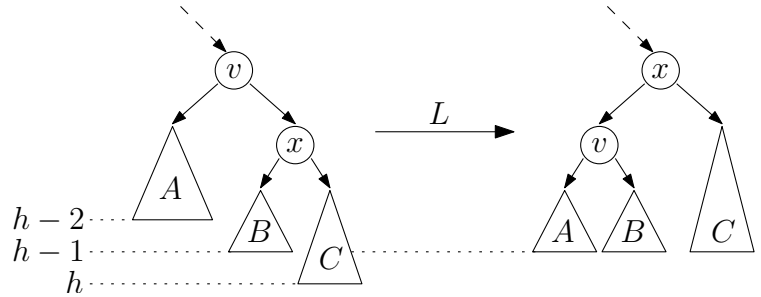
- $\text{bal}(v) = -1$: Der Knoten ist nun außer Balance, denn die aktuelle Balance von v beträgt -2. Im Folgenden bezeichne x das rechte Kind von v.

  - 1. Unterfall: Die beiden Teilbäume von x haben die gleiche Tiefe, d. h. $\text{bal}(x) = 0$. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, genügt eine Linksrotation wie in der folgenden Abbildung dargestellt.



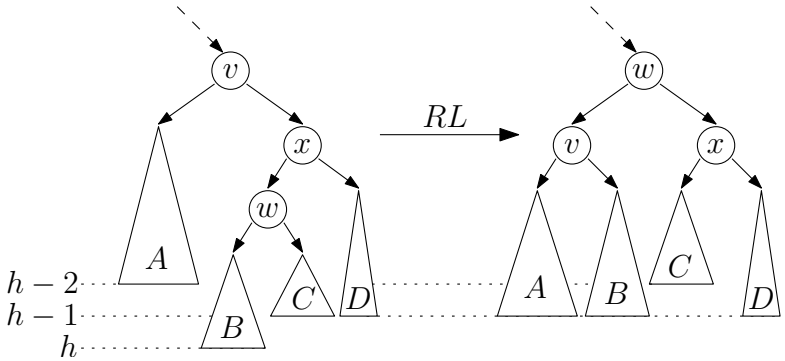
    Die Rebalancierung ist abgeschlossen, da die Höhe des betrachteten Teilbaumes nach der Linksrotation wieder genauso groß ist wie vor dem Löschen und sich die Balancen der Knoten oberhalb dieses Teilbaumes somit nicht geändert haben.

  - 2. Unterfall: Die Höhe des rechten Teilbaumes von x ist um eins größer als die des linken Teilbaumes, d.h. $\text{bal}(x) = -1$. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, genügt eine Linksrotation wie in der folgenden Abbildung dargestellt.



Sofern v nicht die Wurzel des Baumes war, muss die Rebalancierung am ehemaligen Elternknoten von v und jetzigen Elternknoten von x fortgesetzt werden, da die Höhe des Teilbaumes um eins kleiner geworden ist.

– 3. Unterfall: Die Höhe des linken Teilbaumes von x ist um eins größer als die des rechten Teilbaumes, d.h. $\text{bal}(x) = 1$. Es bezeichne w das linke Kind von x. Um die AVL-Eigenschaft an Knoten v wiederherzustellen, genügt eine Doppelrotation wie in der folgenden Abbildung dargestellt.



Für einen Teilbaum T bezeichne $h(T)$ seine Höhe. Aus den Balancen von v und x ergibt sich, dass die Höhen $h(A)$ und $h(D)$ übereinstimmen. Ebenso folgt $h(A) = h(D) = \max\{h(B), h(C)\}$. Abhängig von der Balance $\text{bal}(w)$ gilt $\min\{h(B), h(C)\} = \max\{h(B), h(C)\}$ oder $\min\{h(B), h(C)\} = \max\{h(B), h(C)\} - 1$.

Sofern v nicht die Wurzel des Baumes war, muss die Rebalancierung am ehemaligen Elternknoten von v und jetzigen Elternknoten von w fortgesetzt werden, da die Höhe des Teilbaumes um eins kleiner geworden ist.

Genauso wie beim Einfügen wird also auch beim Löschen ein Pfad von einem Blatt bis potentiell zur Wurzel verfolgt. Wird der Knoten v von seinem rechten Kind aus erreicht, so verläuft die Fallunterscheidung analog. Dabei werden Linksrotationen zu Rechtsrotationen und RL-Doppelrotationen zu LR-Doppelrotationen. Da die Höhe des Baumes h ist, werden auf diesem Pfad maximal $h + 1$ Knoten erreicht. Pro Knoten benötigt Delete nur eine konstante Laufzeit, da maximal eine Rotation oder Doppelrotation durchgeführt werden muss. Insgesamt ergibt sich zusammen mit Theorem 4.6 das folgende Ergebnis.

Theorem 4.7. *Die Operationen* Search, Insert *und* Delete *benötigen in AVL-Bäumen mit n Daten eine Laufzeit von $O(\log(n))$.*

4.2.2 B-Bäume

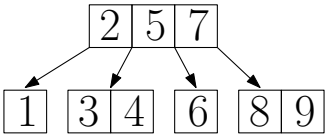
Wir haben gesehen, dass AVL-Bäume bereits alle wesentlichen Operationen einer dynamischen Menge relativ effizient unterstützen. Wir werden nun mit *B-Bäumen* eine weitere Datenstruktur kennenlernen, die asymptotisch dieselben Laufzeiten erreicht, die aber für viele praktische Zwecke besser geeignet ist, da sie die *Speicherhierarchie* realer Rechner besser berücksichtigt. Bei großen Datenmengen tritt das Problem auf, dass nicht alle Daten in den Hauptspeicher passen, sondern teilweise auf die Festplatte ausgelagert werden müssen. Zugriffe auf die Festplatte sind aber selbst beim Einsatz von Solid-State-Drives um einige Größenordnungen langsamer als Zugriffe auf den Hauptspeicher. Dieser Effekt wird bei AVL-Bäumen nicht berücksichtigt und je nach Größe des Datensatzes und Hauptspeichers kann es passieren, dass beim Suchen, Einfügen oder Löschen $\Theta(\log n)$ Zugriffe auf die Festplatte erfolgen, wenn jeder erreichte Knoten erst von der Festplatte ausgelesen werden muss.

B-Bäume machen sich zunutze, dass der Speicher realer Rechner in sogenannte *Seiten* eingeteilt ist, die in der x86-Architektur beispielsweise eine Größe von 4kB haben. Werden Daten auf der Festplatte adressiert, so werden keine einzelnen Bytes, sondern stets ganze Seiten in den Hauptspeicher verschoben. Alle Daten einer Seite zu lesen dauert also kaum länger als ein einzelnes Byte zu lesen. Aus diesem Grund enthält ein B-Baum in jedem Knoten nicht nur einen einzelnen Schlüssel, sondern mehrere, die in derselben Seite gespeichert werden, sodass sie gemeinsam gelesen und geschrieben werden können.

Enthält ein innerer Knoten v eines B-Baumes ℓ Schlüssel, so besitzt er $\ell + 1$ Kinder $u_1, \ldots, u_{\ell+1}$. Seien $v.\text{key}(1) \leq \ldots \leq v.\text{key}(\ell)$ die Schlüssel von v und sei k_i für jedes $i \in \{1, \ldots, \ell + 1\}$ ein beliebiger Schlüssel in dem Unterbaum mit Wurzel u_i. Dann gilt

$$k_1 \leq v.\text{key}(1) \leq k_2 \leq v.\text{key}(2) \leq \ldots \leq v.\text{key}(\ell) \leq k_{\ell+1}.$$

Einen Baum mit dieser Eigenschaft nennt man auch einen *verallgemeinerten Suchbaum*. Die folgende Abbildung zeigt ein Beispiel für einen solchen Baum.



Definition 4.8. *Ein verallgemeinerter Suchbaum T heißt* B-Baum der Ordnung t *(für ein $t \in \mathbb{N}$ mit $t \geq 2$), falls die folgenden Eigenschaften erfüllt sind.*

1. *Alle Blätter haben dieselbe Tiefe.*
2. *Jeder Knoten mit Ausnahme der Wurzel enthält mindestens $t - 1$ Daten und jeder Knoten einschließlich der Wurzel enthält höchstens $2t - 1$ Daten.*
3. *Ist der Baum nicht leer, so enthält die Wurzel mindestens ein Datum.*

In der Praxis orientiert sich t an der Zahl an Schlüsseln, die innerhalb einer Seite gespeichert werden können. Eine Wahl im drei- oder vierstelligen Bereich ist in Anwendungen nicht unüblich.

Theorem 4.9. *Für die Höhe h eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt*

$$h \leq \log_t \left(\frac{n+1}{2}\right).$$

Beweis. Da der Baum nicht leer ist, enthält die Wurzel mindestens ein Datum. Jeder andere Knoten enthält per Definition mindestens $t-1$ Daten. Dementsprechend besitzt die Wurzel mindestens zwei Kinder, sofern sie nicht bereits ein Blatt ist. Ebenso besitzt jeder andere innere Knoten mindestens t Kinder. Daraus folgt, dass es auf Tiefe 0 einen Knoten gibt (die Wurzel), auf Tiefe 1 mindestens 2 Knoten (die Kinder der Wurzel), auf Tiefe 2 mindestens $2t$ Knoten, auf Tiefe 3 mindestens $2t^2$ Knoten und so weiter. Allgemein folgt mit dieser Argumentation, dass es auf Tiefe $i \leq h$ mindestens $2t^{i-1}$ Knoten gibt. Da die Wurzel mindestens ein Datum enthält und jeder andere Knoten mindestens $t-1$, folgt

$$n \geq 1 + (t-1)\sum_{i=1}^{h}(2t^{i-1}) = 1 + 2(t-1)\sum_{i=0}^{h-1} t^i = 1 + 2(t-1)\frac{t^h - 1}{t-1} = 2t^h - 1.$$

Das Theorem folgt direkt durch Umstellen dieser Ungleichung nach t. □

Wir können die obere Schranke für die Höhe von B-Bäumen aus dem vorangegangenen Theorem auch als

$$\log_t \left(\frac{n+1}{2}\right) = \frac{\log_2((n+1)/2)}{\log_2 t} = \frac{\log_2(n+1) - 1}{\log_2 t} \approx \frac{\log_2(n+1)}{\log_2 t}$$

schreiben. Für die Höhe eines AVL-Baumes mit n Daten haben wir in Theorem 4.6 eine untere Schranke von $\lceil \log_2(n+1) - 1 \rceil$ nachgewiesen. Damit ist die Höhe eines B-Baumes um mindestens einen Faktor von ungefähr $\log_2 t$ kleiner als die eines AVL-Baumes mit derselben Anzahl an Daten. Da die Anzahl der besuchten Knoten beim Suchen, Einfügen und Löschen und damit auch die Zahl der Speicherzugriffe sowohl in AVL- als auch in B-Bäumen linear von der Höhe abhängt, ist dies auch in etwa die Einsparung an Speicherzugriffen, die mit einem B-Baum gegenüber einem AVL-Baum erzielt wird.

Suchen

Wir beschreiben nun, wie die Operationen Search, Insert und Delete in einem B-Baum realisiert werden können. Suchen in allgemeinen Suchbäumen verläuft analog zu binären Suchbäumen. Bei einem binären Suchbaum gibt der Schlüssel eines Knotens an, ob im linken oder rechten Teilbaum weitergesucht werden muss. Bei einem verallgemeinerten Suchbaum geben die Schlüssel ebenfalls den Teilbaum an, in dem weitergesucht werden muss.

Die folgende Methode gibt den Knoten zurück, in dem das gesuchte Datum gespeichert ist, und den Index, an welcher Stelle sich das Datum in diesem Knoten befindet. Für einen Knoten v bezeichne $v.\ell$ die Anzahl an Daten, die in v gespeichert sind,

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$ die Schlüssel dieser Daten und $v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$ die Kinder von v.

```
B-Tree-Search(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-Tree-Search(v.child(i),k);
```

Die Methode Search besucht höchstens $h+1$ Knoten, wenn h die Höhe des B-Baumes bezeichnet. Es werden also maximal $O(h) = O(\log_t(n))$ verschiedene Seiten angefragt. Innerhalb eines Knotens wird jeweils der richtige Teilbaum gesucht, in dem die Suche fortgesetzt wird. Dazu wird eine Laufzeit von $O(t)$ benötigt, sodass die Gesamtlaufzeit von Search durch $O(th) = O(t \log_t(n))$ beschränkt ist. Werden die Schlüssel innerhalb eines Knotens in einem sortierten Feld gespeichert, so kann die Laufzeit pro Knoten mithilfe von binärer Suche sogar auf $O(\log t)$ reduziert werden.

Einfügen

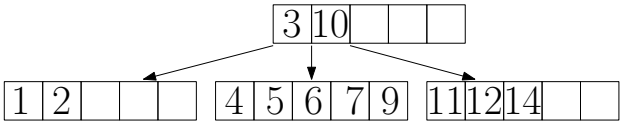
Als nächstes müssen die Operationen Insert und Delete beschrieben werden. Hier liegt die Schwierigkeit darin, zu garantieren, dass nach dem Einfügen oder Löschen eines Datums die Eigenschaften eines B-Baumes aus Definition 4.8 wieder erfüllt sind. Ähnlich wie bei AVL-Bäumen ist es dazu nach dem Einfügen oder Löschen gegebenenfalls notwendig, gewisse Reparaturoperationen durchzuführen.

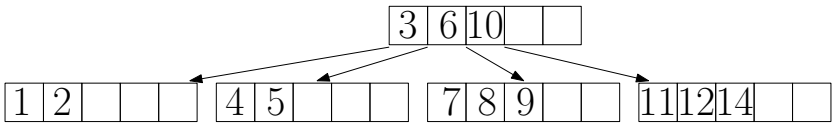
Zunächst beschreiben wir die Methode Insert. Ähnlich wie bei AVL-Bäumen beschreiben wir diese nicht in Pseudocode, sondern abstrakter. Der Leserin und dem Leser sei aber auch hier als Übung empfohlen, B-Bäume zu implementieren.

Es sei x das Datum mit Schlüssel $k = x.\text{key}$, das in den B-Baum eingefügt werden soll. Wir suchen zunächst mittels Search nach dem Schlüssel k. Wird dieser gefunden, so wird einfach das entsprechende Datum überschrieben. Ansonsten endet die erfolglose Suche von Search in einem Blatt v des B-Baumes. Wir fügen diesem Blatt das Datum x hinzu, sofern es nicht bereits $2t - 1$ Daten enthält. Der Schlüssel k bestimmt die Position von x innerhalb des Blattes, das heißt, die bereits vorhandenen Daten müssen gegebenenfalls verschoben werden, um für das neue Datum Platz zu schaffen. Da wir bei der Suche nach k das Blatt v erreicht haben, wird durch diese Einfügung die Suchbaumeigenschaft nicht verletzt.

Falls das Blatt v bereits $2t - 1$ Schlüssel enthält, so kann das Datum x nicht hinzugefügt werden, ohne die B-Baum-Eigenschaft zu verletzen. In diesem Fall teilen wir den Knoten v mit den Schlüsseln $v.\text{key}(1), \ldots, v.\text{key}(2t-1)$ vor dem Einfügen zunächst in zwei Knoten u_1 und u_2 auf. Der Knoten u_1 enthält die $t - 1$ Daten mit den Schlüsseln $v.\text{key}(1), \ldots, v.\text{key}(t-1)$ und der Knoten u_2 enthält die $t - 1$ Daten mit den Schlüsseln $v.\text{key}(t+1), \ldots, v.\text{key}(2t-1)$. Ist v nicht die Wurzel des Baumes, so wird das Datum mit dem Schlüssel $v.\text{key}(t)$ in den Elternknoten von v geschoben. Nach

dem Aufteilen des Knotens v wird das Datum x in einen der neuen Knoten u_1 oder u_2 eingefügt, abhängig davon, ob $x.\text{key} < v.\text{key}(t)$ oder $x.\text{key} > v.\text{key}(t)$ gilt. Dies ist in der folgenden Abbildung an einem Beispiel dargestellt. Wird dort der Schlüssel 8 eingefügt, so muss das entsprechende Blatt zunächst aufgeteilt werden, was in der zweiten Abbildung zu sehen ist.



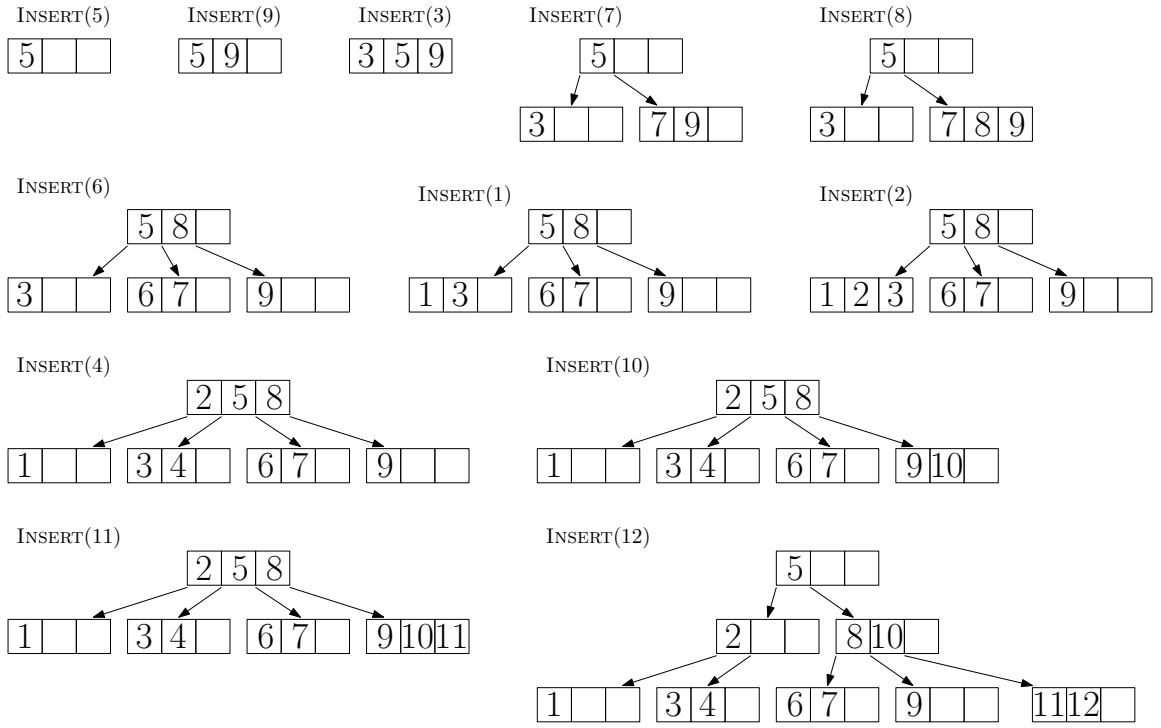


Besitzt der Elternknoten w von v bereits $2t-1$ Schlüssel, so ist anders als in dem obigen Beispiel kein Platz, um das Datum mit dem Schlüssel $v.\text{key}(t)$ dorthin zu schieben. Dann wird mit dem Elternknoten w genauso verfahren wie mit dem Knoten v. Das heißt, der Knoten w wird in zwei Knoten aufgeteilt, die jeweils $t-1$ Schlüssel enthalten, und das Datum mit dem Schlüssel $w.\text{key}(t)$ wird wiederum in den Elternknoten von w verschoben. Dieses Prozedere wird solange wiederholt, bis entweder ein Knoten erreicht ist, der noch Platz für einen weiteren Schlüssel besitzt, oder bis die Wurzel des B-Baumes erreicht ist.

Wird ein weiterer Schlüssel in die Wurzel r des Baumes eingefügt und besitzt die Wurzel bereits $2t-1$ Daten, so wird sie genau wie oben beschrieben in zwei Knoten u_1 und u_2 mit den Daten mit den Schlüsseln $r.\text{key}(1), \ldots, r.\text{key}(t-1)$ bzw. $r.\text{key}(t+1), \ldots, r.\text{key}(2t-1)$ aufgespalten. Das Datum mit dem Schlüssel $r.\text{key}(t)$ kann nicht in den Elternknoten von r verschoben werden, da ein solcher nicht existiert. Stattdessen wird ein neuer Knoten angelegt, der ausschließlich das Datum mit dem Schlüssel $r.\text{key}(t)$ enthält. Dieser Knoten wird die neue Wurzel des Baumes und u_1 und u_2 werden seine beiden Kinder. Dies ergibt einen neuen gültigen B-Baum, da die Wurzel als einziger Knoten weniger als $t-1$ Daten enthalten darf.

Da auch beim Einfügen nur ein Pfad von der Wurzel zu einem der Blätter betrachtet wird (bei der Suche zunächst von oben nach unten, anschließend gegebenenfalls ein zweites Mal von unten nach oben) ergibt sich wie bei SEARCH eine obere Schranke von $O(h) = O(\log_t(n))$ für die Anzahl an Seitenzugriffen. Die Laufzeit pro Knoten beträgt $O(t)$, da das Hinzufügen eines zusätzlichen Datums zu einem Knoten und das Aufteilen eines Knotens, wenn er bereits $2t-1$ Daten besitzt, in Linearzeit bezogen auf die Anzahl an Daten in dem Knoten durchgeführt werden kann. Daraus ergibt sich auch für das Einfügen eine Laufzeit von $O(th) = O(t \log_t(n))$.

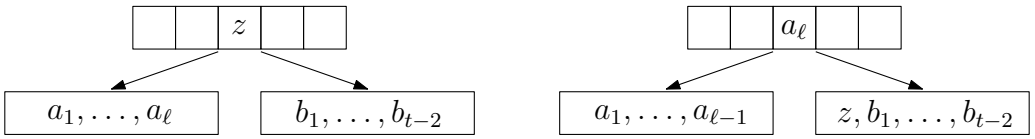
Wir betrachten im Folgenden ein Beispiel, in dem die Schlüssel 5, 9, 3, 7, 8, 6, 1, 2, 4, 10, 11 und 12 in einen anfangs leeren B-Baum der Ordnung $t = 2$ eingefügt werden. In einem solchen Baum besitzt jeder Knoten mindestens ein Datum und höchstens drei Daten.



Löschen

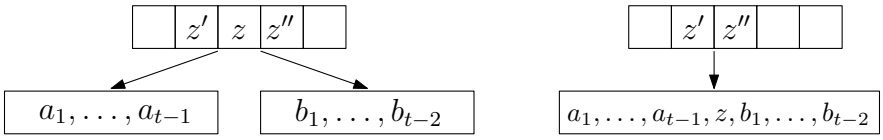
Wir beschreiben nun, wie die Methode DELETE(k) in einem B-Baum realisiert werden kann. Zunächst wird dazu in dem Baum das Datum mit Schlüssel k gesucht. Es bezeichne v den Knoten, in dem sich dieses Datum befindet, und es sei k der i-te Schlüssel des Knotens v. Ist v kein Blatt, so suchen wir in dem Baum zunächst das Datum mit dem größten Schlüssel $a < k$. Dazu gehen wir zum i-ten Kind von v und anschließend immer zum sich am weitesten rechts befindenden Kind des aktuellen Knotens, solange bis ein Blatt u erreicht wird. In diesem Blatt ist das Datum mit dem größten Schlüssel das gesuchte Datum. Wir ersetzen im Knoten v dann den Schlüssel k und das dazugehörige Datum durch den Schlüssel a und das dazugehörige Datum und löschen dieses aus dem Blatt u. Wegen der Wahl von a bleibt die Suchbaumeigenschaft erhalten, denn für alle anderen Schlüssel b gilt entweder $b < a < k$ oder $a < k < b$, das heißt die Relation zwischen a und b ist dieselbe wie zwischen k und b.

Das Blatt u besitzt nach dem Löschen einen Schlüssel weniger als zuvor. Sind dennoch mindestens $t - 1$ Schlüssel in u vorhanden, so liegt weiterhin ein B-Baum vor und DELETE(k) ist abgeschlossen. Ansonsten testen wir, ob ein Nachbarknoten von u mindestens t Daten enthält. (Ist u das i-te Kind seines Elternknotens, so bezeichnen wir das $(i-1)$-te und das $(i+1)$-te Kind als seine Nachbarknoten, sofern sie existieren.) Ist das der Fall, so kann die B-Baum-Eigenschaft durch eine Rotation eines Datums wieder hergestellt werden. In der folgenden Abbildung ist der Fall dargestellt, dass der linke Nachbar von u die Schlüssel $a_1, \ldots, a_\ell$ mit $\ell \geq t$ enthält. Wir bezeichnen die Schlüssel von u mit $b_1, \ldots, b_{t-2}$.



Nach dieser Rotation besitzt der linke Nachbar von u noch $\ell - 1 \geq t - 1$ Daten und der Knoten u besitzt $t-1$ Daten. Es ist leicht zu sehen, dass die Suchbaumeigenschaft erhalten bleibt. Damit liegt wieder ein B-Baum vor und das Löschen ist abgeschlossen.

Besitzen alle Nachbarn von u nur genau $t-1$ Daten, so verschmelzen wir einen der Nachbarn mit dem Knoten u. Dies ist in der folgenden Abbildung illustriert.



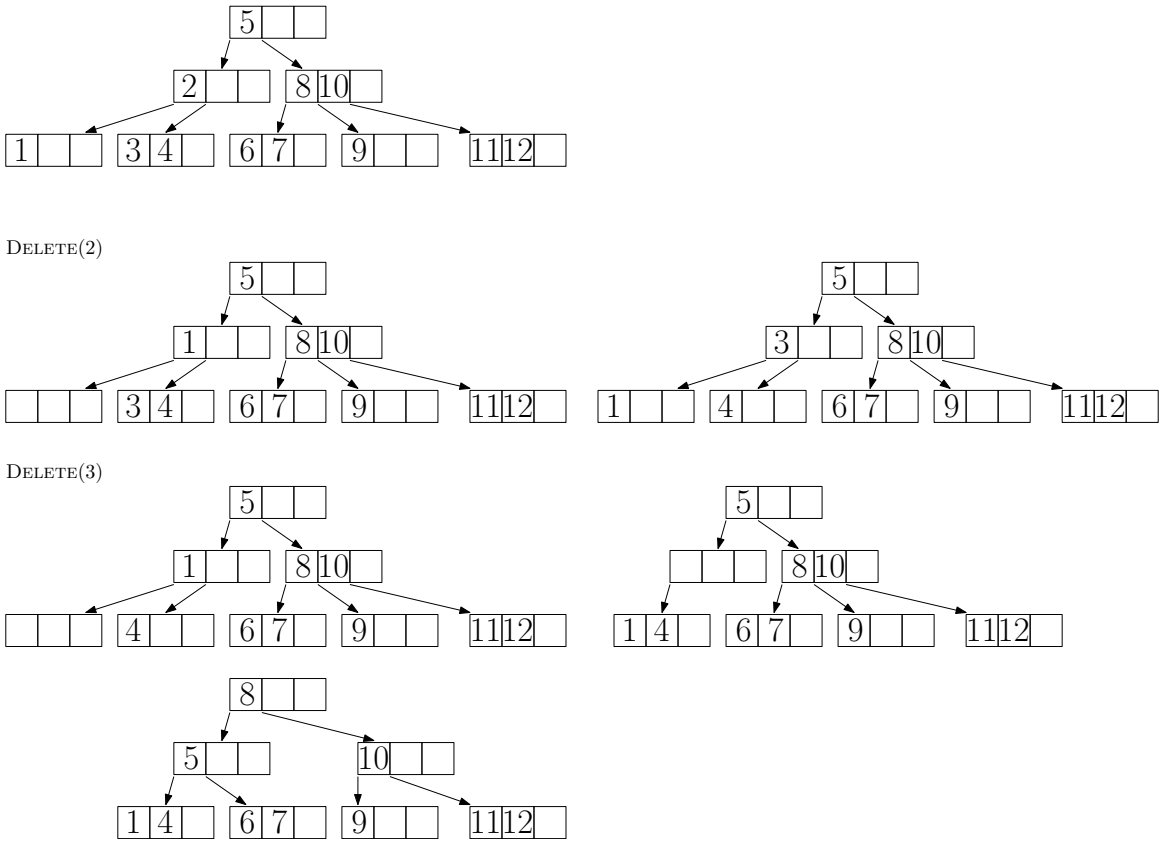
Der neue Knoten besitzt $2t-2$ Daten und die Suchbaumeigenschaft ist nach wie vor erfüllt. Allerdings wurde im Elternknoten von u ein Datum gelöscht. Enthält dieser nun weniger als $t-1$ Schlüssel, so wird mit diesem Knoten analog fortgefahren. Dadurch kann das Löschen unter Umständen bis zur Wurzel propagiert werden. Besitzt die Wurzel nur ein Datum, das gelöscht wird, so sinkt die Tiefe des Baumes um eins. Wendet man die obigen Reparaturoperationen auf innere Knoten des Baumes an, so ist darauf zu achten, dass bei einer Rotation auch die Teilbäume korrekt umgehängt werden. Dies wird in dem Beispiel weiter unten noch einmal illustriert.

Analog zum Einfügen kann man auch hier wieder argumentieren, dass die Laufzeit $O(th) = O(t \log_t(n))$ beträgt und $O(h) = O(\log_t(n))$ Seitenzugriffe getätigt werden. Zusammengefasst ergibt sich das folgende Theorem.

Theorem 4.10. *Die Operationen* Search, Insert *und* Delete *benötigen in einem B-Baum der Ordnung t mit n Daten eine Laufzeit von $O(th) = O(t \log_t(n))$ und sie benötigen $O(h) = O(\log_t(n))$ Seitenzugriffe, wenn alle Daten eines Knotens in derselben Seite gespeichert sind.*

Die folgende Abbildung zeigt als Beispiel, wie aus dem B-Baum, den wir oben konstruiert haben, die Schlüssel 2 und 3 gelöscht werden. Beim Löschen von 2 wird zunächst der größte Schlüssel, der kleiner als 2 ist, an die Position von 2 geschoben. In dem Beispiel ist das der Schlüssel 1. Der Knoten, in dem der Schlüssel 1 vorher gespeichert war, besitzt danach keinen Schlüssel mehr. Da sein Nachbarknoten zwei Schlüssel besitzt, kann die B-Baum-Eigenschaft durch eine Rotation wiederhergestellt werden.

Beim Löschen von 3 wird auch zunächst das Datum mit Schlüssel 1 an die Stelle geschoben, an der sich das Datum mit Schlüssel 3 befand. Danach besitzt der Knoten, in dem der Schlüssel 1 vorher gespeichert war, keinen Schlüssel mehr. Da nun sein einziger Nachbar nur noch einen Schlüssel besitzt, kann keine Rotation durchgeführt werden. Stattdessen werden die Knoten zu einem gemeinsamen Knoten zusammengefügt. Danach besitzt der Elternknoten dieser Knoten keinen Schlüssel mehr. Dies kann wiederum durch eine Rotation behoben werden.



4.3 Hashing

Ist das Universum U, aus dem die Schlüssel stammen, nicht allzu groß, so gibt es eine ebenso einfache wie effiziente Möglichkeit, eine dynamische Menge zu realisieren. Man legt dazu einfach ein Feld der Länge $|U|$ an, das für jeden möglichen Schlüssel $x \in U$ einen Eintrag enthält, in dem das zu x gehörige Datum oder, falls in der dynamischen Menge kein solches existiert, ein Null-Zeiger gespeichert ist. Suchen, Einfügen und Löschen können dann in konstanter Zeit durchgeführt werden, da bei allen Operationen lediglich ein Zugriff auf die entsprechende Position des Feldes erfolgt.

Das Problem mit dem obigen Vorgehen ist natürlich, dass das Universum U der möglichen Schlüssel in den allermeisten Anwendungen extrem groß ist, sodass es praktisch unmöglich ist, ein Feld der Länge $|U|$ anzulegen. Dies gilt insbesondere dann, wenn die Anzahl der tatsächlich gespeicherten Daten sehr viel kleiner als $|U|$ ist.

Um diese Idee dennoch realisieren zu können, bedient man sich sogenannter *Hashfunktionen*. Eine solche Funktion $h\colon U \to \{0, 1, \ldots, m-1\}$ bildet das Universum U auf die Menge $\{0, 1, \ldots, m-1\}$ ab, wobei m typischerweise sehr viel kleiner als $|U|$ gewählt ist. Dann kann die obige Idee mit einem Feld der Länge m umgesetzt werden. Der einzige Unterschied ist, dass ein Element mit Schlüssel k an der Position $h(k)$ anstatt an der Position k in dem Feld abgelegt wird. Eine solche Datenstruktur wird *Hashtabelle* genannt. Die Funktion h sollte effizient auswertbar sein. Wir gehen im Folgenden davon aus, dass der Wert $h(k)$ für jeden Schlüssel k in konstanter Zeit berechnet werden

kann.

Da m normalerweise kleiner als $|U|$ ist, gibt es keine injektive Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$, das heißt es gibt für jede Hashfunktion h zwei verschiedene Schlüssel $k_1 \in U$ und $k_2 \in U$ mit $h(k_1) = h(k_2)$. Wir sagen dann, dass die Schlüssel k_1 und k_2 *kollidieren*. Dies ist natürlich dann problematisch, wenn zwei verschiedene Daten mit diesen Schlüsseln in der Hashtabelle gespeichert werden sollen. Wir werden im Folgenden Möglichkeiten kennenlernen, wie mit solchen Kollisionen umgegangen werden kann. Außerdem werden wir analysieren, wie viele Kollisionen es gibt, und uns mit der Frage beschäftigen, wie eine gute Hashfunktion gewählt werden kann.

Nachdem wir mit balancierten Suchbäumen bereits eine Möglichkeit kennengelernt haben, um dynamische Mengen zu realisieren, stellt sich die Frage, wozu Hashing überhaupt noch benötigt wird. Tatsächlich ist Hashing aber eine Datenstruktur, die in unzähligen Anwendungen zum Einsatz kommt. Dies sind insbesondere Anwendungen, in denen große Datenmengen verarbeitet werden, und in denen nur die Methoden Search, Insert und ggf. Delete benötigt werden. Anders als Suchbäume enthalten Hashtabellen keine weiteren Informationen über die Ordnung der Schlüssel, sondern sie unterstützen genau die drei genannten Operationen. Mit einer geschickten Wahl der Hashfunktion kann erreicht werden, dass die Laufzeit dieser Operationen in einer Hashtabelle zwar nicht im Worst Case jedoch in Anwendungen typischerweise nur $O(1)$ (statt wie in Suchbäumen $O(\log n)$) beträgt. Eine typische Anwendung ist ein Router, der eine Blacklist verwaltet und IP-Adressen aus dieser blockiert. Diese Blacklist, die sich typischerweise im Laufe der Zeit verändert, kann in einer dynamischen Menge gespeichert werden. Für jedes eintreffende Paket gleicht der Router die IP-Adresse mit der Blacklist mithilfe einer Search-Operation ab. Dies muss sehr schnell gehen und wird dementsprechend oft durch eine Hashtabelle realisiert.

4.3.1 Hashfunktionen

Es gibt sehr viele theoretische und experimentelle Arbeiten über die Wahl einer guten Hashfunktion. Wir werden in dieses Thema in dieser Vorlesung nicht tief einsteigen und lediglich kurz andiskutieren, wie eine Hashfunktion aussehen könnte und welche Eigenschaften sie besitzen sollte.

Grundsätzlich sollte eine Hashfunktion die Schlüssel aus U möglichst gleichmäßig auf die Menge $\{0, 1, \ldots, m-1\}$ verteilen, das heißt, die Menge $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ sollte für jedes i in etwa gleich groß sein. Man könnte sich auf den Standpunkt stellen, dass theoretisch jede Hashfunktion, bei der diese Eigenschaft gegeben ist, gleich gut ist. Dies ist aber für echte Daten, die oft bestimmte Strukturen aufweisen, nicht der Fall. Sind die Schlüssel beispielsweise Namen von Personen, so könnte man $m = 26$ wählen und jeden Namen auf seinen ersten Buchstaben abbilden. Dann gibt es theoretisch zwar genauso viele Zeichenketten, die auf X abgebildet werden, wie Zeichenketten, die auf S abgebildet werden, dennoch wird es in einer echten Datenbank typischerweise sehr viel mehr Namen geben, die mit S beginnen, als solche, die mit X beginnen.

Handelt es sich bei den Schlüsseln um nichtnegative ganze Zahlen, kann man einfache Hashfunktionen mit der *Divisionsmethode* und der *Multiplikationsmethode* finden.

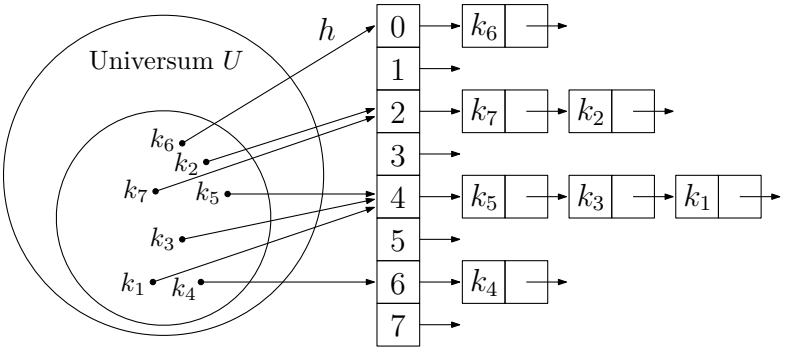
Bei der Divisionsmethode setzt man $h(k) = k \bmod m$, das heißt, der Hashwert eines Schlüssels ist der Rest, den er bei Division durch m lässt. Bei dieser Methode sollte vermieden werden, die Größe m der Hashtabelle als Zweierpotenz zu wählen. Gilt nämlich $m = 2^i$, so hängt der Hashwert eines Schlüssels nur von den i Bits mit der geringsten Wertigkeit ab. Je nach Anwendung kann dies zu sehr vielen Kollisionen führen. Es hat sich bei der Divisionsmethode empirisch als besser herausgestellt, die Größe m der Hashtabelle als eine Primzahl zu wählen.

Bei der Multiplikationsmethode wählt man zunächst eine Konstante $A \in (0, 1)$. Anschließend berechnet man für einen Schlüssel k die Nachkommastellen von kA mittels $kA - \lfloor kA \rfloor$. Dieser Wert liegt in dem Intervall $[0, 1)$. Um eine Position in der Hashtabelle zu erhalten, multipliziert man diesen Wert mit m und rundet anschließend ab. Man setzt also $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$. Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich empirisch und theoretisch als gut herausgestellt.

Oft handelt es sich bei den Schlüsseln nicht um Zahlen, sondern um Zeichenketten. In Java besitzt jede Klasse eine Methode namens `hashCode()`, die für jedes Objekt einen Hashwert vom Typ `int` liefert. Wie diese Methode implementiert ist, hängt von der konkreten Klasse ab. In der Klasse `String` ist die Methode so implementiert, dass eine Zeichenkette $s_{\ell-1}s_{\ell-2}\ldots s_1 s_0$ auf den Hashwert $\sum_{i=0}^{\ell-1} s_i \cdot 31^i$ abgebildet wird, wobei wir s_i in der Formel als den numerischen Wert des Zeichens s_i (aufgefasst als `char`) interpretieren. In der Implementierung in Java wird mit dem primitiven Datentyp `int` gerechnet. Dies führt bei längeren Zeichenketten schnell zu Überläufen, was jedoch auch beabsichtigt ist.

4.3.2 Hashing mit verketteten Listen

Eine Möglichkeit, mit Kollisionen umzugehen, ist *Hashing mit verketteten Listen*. Dabei wird ein Feld $T[0 \ldots m-1]$ angelegt, in dem jeder Eintrag entweder der Null-Zeiger oder ein Zeiger auf das erste Element einer verketteten Liste ist. Die verkettete Liste an Position $i \in \{0, \ldots, m-1\}$ enthält alle Daten in der dynamischen Menge mit einem Schlüssel k mit $h(k) = i$. Die folgende Abbildung zeigt ein Beispiel.



Zum Suchen, Einfügen und Löschen von Daten wird auf die entsprechenden Operationen der verketteten Liste zurückgegriffen.

- SEARCH(k): Gib das Element mit dem Schlüssel k in der Liste $T[h(k)]$ zurück, sofern es existiert.

- INSERT(x): Füge das Element x an den Anfang der Liste $T[h(x.\text{key})]$ ein, falls es noch nicht enthalten ist.
- DELETE(k): Lösche das Element mit dem Schlüssel k aus der Liste $T[h(k)]$, sofern es existiert.

Im Folgenden bezeichne n die Anzahl an Daten, die in der Hashtabelle gespeichert sind. Alle Operationen besitzen eine Laufzeit, die schlimmstenfalls linear in der Länge der relevanten Liste ist. Die Laufzeit von INSERT ist sogar konstant, sofern nicht getestet wird, ob ein Element mit demselben Schlüssel bereits in der Liste enthalten ist. Im Worst Case kann es jedoch passieren, dass die Schlüssel aller n Daten kollidieren und somit alle Daten in derselben Liste gespeichert sind. In diesem Fall können wir die Laufzeiten von SEARCH und DELETE nur durch $O(n)$ abschätzen.

Es ist für große n allerdings ausgesprochen unwahrscheinlich, dass alle Schlüssel kollidieren, wenn die Hashfunktion sinnvoll gewählt ist. Aus diesem Grund sind Hashtabellen in der Praxis deutlich effizienter als die obige Abschätzung für den Worst Case vermuten lässt. Um dies theoretisch zu untermauern, gehen wir im Folgenden davon aus, dass die Hashfunktion jeden Schlüssel zufällig auf ein Element aus $\{0, 1, \ldots, m-1\}$ abbildet. Konkret betrachten wir eine Hashtabelle mit n Daten, wobei jedes Datum unabhängig von den anderen in einer zufälligen Liste $T[i]$ für $i \in \{0, 1, \ldots, m-1\}$ gespeichert ist. Dabei besitzt jedes $i \in \{0, 1, \ldots, m-1\}$ dieselbe Wahrscheinlichkeit $1/m$ gewählt zu werden. Etwas anschaulicher könnte man auch sagen, dass n Daten nacheinander in eine anfangs leere Hashtabelle eingefügt werden und dass beim Einfügen eines jeden Datums ein fairer Würfel mit m Seiten geworfen wird, der die Position des Datums in der Hashtabelle bestimmt. Diese Annahme nennt man *uniformes Hashing*.

Uniformes Hashing ist der Idealfall, den wir durch eine geschickt auf die Anwendung abgestimmte Hashfunktion zu erreichen versuchen, da er intuitiv die bestmögliche Streuung der Daten auf die Positionen der Hashtabelle garantiert, die man mit einer Hashfunktion sinnvoll erreichen kann. Man muss aber natürlich bei allen Schranken, die wir zeigen werden, im Hinterkopf behalten, dass uniformes Hashing nur eine idealisierte Annahme ist, die in Anwendungen nicht immer gerechtfertigt ist. Dennoch bieten die Schranken einen guten Anhaltspunkt für die Laufzeit, die wir in Anwendungen erwarten.

Wir betrachten im Folgenden eine Hashtabelle, in die die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden und wir gehen von uniformem Hashing aus. Das heißt, jedes Datum befindet sich an einer uniform zufälligen Position unabhängig von den anderen. Mit $\alpha = n/m$ bezeichnen wir den *Auslastungsfaktor* der Hashtabelle. Wir werden im Folgenden die erwartete Laufzeit für das Suchen nach Daten in der Hashtabelle genauer analysieren. Auf die Operationen INSERT und DELETE gehen wir nicht näher ein, da ihre Laufzeiten durch die Laufzeit von SEARCH dominiert sind.

Theorem 4.11. *Unter der Annahme des uniformen Hashings benötigt eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, im Erwartungswert eine Laufzeit von* $\Theta(1 + \alpha)$.

Beweis. Es sei k der Schlüssel, nach dem erfolglos gesucht wird, und es sei $i = h(k)$ sein Hashwert. Uns interessiert die erwartete Länge der Liste $T[i]$, da diese die erwartete

Laufzeit der Suche bestimmt. Für jeden in der Hashtabelle gespeicherten Schlüssel k_j mit $j \in \{1, \ldots, n\}$ definieren wir eine Zufallsvariable X_j wie folgt:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$, da es sich bei X_j um eine 0-1-Zufallsvariable handelt.

Dann ist die Länge $|T[i]|$ der Liste $T[i]$ eine Zufallsvariable, die der Summe $\sum_{j=1}^{n} X_j$ entspricht. Für die erwartete Länge gilt demnach

$$\mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j] = \frac{n}{m} = \alpha,$$

wobei die zweite Gleichung aus der Linearität des Erwartungswertes folgt.

Somit folgen wir bei der Suche nach dem Schlüssel k im Erwartungswert $1 + \alpha$ vielen Zeigern, wobei der Summand 1 für den Null-Zeiger des letzten Eintrages der Liste $T[i]$ steht. □

Nun betrachten wir noch den Fall, dass wir eine erfolgreiche Suche nach einem Schlüssel in der Hashtabelle durchführen. Legen wir uns auf einen konkreten Schlüssel k_i für ein i fest, so können wir die Laufzeit fast genauso analysieren wie in Theorem 4.11. Wir betrachten aber stattdessen den Fall, dass wir ein $i \in \{1, \ldots, n\}$ uniform zufällig auswählen und nach dem Schlüssel k_i suchen. Auf den ersten Blick scheint dieses Szenario auch nicht wesentlich anders zu sein, als nach einem festen Schlüssel zu suchen. Es gibt jedoch einen Unterschied: lange Listen, die viele Schlüssel haben, werden mit höherer Wahrscheinlichkeit ausgewählt. Dies könnte dazu führen, dass die erwartete Laufzeit steigt. Das folgende Theorem zeigt, dass dies nicht der Fall ist.

Theorem 4.12. *Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von* $\Theta(1 + \alpha)$.

Beweis. Wir gehen davon aus, dass die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden. Die entscheidende Frage ist, wie vielen Zeigern wir folgen, wenn wir nach einem zufällig ausgewählten Schlüssel k_i suchen. Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel nach k_i in die Liste $T[h(k_i)]$ eingefügt werden.

Für jedes $i \in \{1, \ldots, n\}$ und jedes $j \in \{i+1, \ldots, n\}$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } h(k_j) = h(k_i), \\ 0 & \text{falls } h(k_j) \neq h(k_i). \end{cases}$$

Es gilt $\mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[h(k_i) = h(k_j)] = 1/m$ und demnach auch $\mathbf{E}[X_{ij}] = 1/m$, da es sich bei X_{ij} um eine 0-1-Zufallsvariable handelt.

Für ein festes $i \in \{1, \ldots, n\}$ können wir die erwartete Anzahl an Zeigern, denen wir bei der Suche nach dem Schlüssel k_i folgen müssen, wie folgt beschreiben:

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}\left[X_{ij}\right] = 1 + \frac{n-i}{m}.$$

Da wir k_i uniform zufällig wählen, bilden wir den Durchschnitt über alle i. Damit können wir die erwartete Anzahl an Zeigern, denen wir folgen müssen, wie folgt beschreiben:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) = \frac{1}{n}\left(n + \sum_{i=1}^{n}\frac{n-i}{m}\right) = 1 + \frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$$
$$= 1 + \frac{1}{nm}\sum_{i=1}^{n-1} i = 1 + \frac{1}{nm}\cdot\frac{n(n-1)}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} = \Theta(1+\alpha).$$

Damit ist das Theorem bewiesen. □

Aus den Theoremen 4.11 und 4.12 folgt, dass die Operationen SEARCH, INSERT und DELETE unter der Annahme des uniformen Hashings im Erwartungswert in Zeit $O(1)$ durchgeführt werden können, wenn $n = O(m)$ gilt.

4.3.3 Geschlossenes Hashing

Wir werden nun eine weitere Möglichkeit kennenlernen, um mit Kollisionen umzugehen. Ein Nachteil des Hashings mit verketteten Listen ist, dass zusätzlicher Speicherplatz für die Zeiger benötigt wird. Beim *geschlossenen Hashing* (auch *Hashing mit offener Adressierung*) werden alle Daten innerhalb des Feldes T gespeichert. Dazu betrachten wir Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\},$$

wobei wir davon ausgehen, dass $h(k,0), \ldots, h(k,m-1)$ für jeden Schlüssel $k \in U$ eine Permutation der Menge $\{0, 1, \ldots, m-1\}$ ist. Wenn ein Datum mit Schlüssel k eingefügt werden soll, dann wird zunächst getestet, ob die Position $h(k,0)$ in dem Feld T bereits belegt ist. Ist dies nicht der Fall, so wird das Datum an dieser Position gespeichert. Ansonsten wird die Position $h(k,1)$ getestet und das Datum dort eingefügt, sofern diese Position noch frei ist. Ist dies nicht der Fall, so wird als nächstes die Position $h(k,2)$ getestet und so weiter. Dieses Verfahren, das auch *Sondieren* genannt wird, ist im folgenden Pseudocode dargestellt.

```
INSERT(x)
 1   for (int i = 0; i < m; i++) {
 2       j = h(x.key, i);
 3       if (T[j] == null) {
 4           T[j] = x;
 5           return j;
 6       }
 7   }
 8   return „Überlauf“;
```

Die Methode INSERT gibt entweder die Position des neu eingefügten Elementes x oder eine Fehlermeldung zurück, falls die Hashtabelle bereits voll ist. Im Gegensatz zu Hashing mit verketteten Listen können beim geschlossenen Hashing maximal m Daten gespeichert werden, wobei m die Größe der Hashtabelle bezeichnet.

Die Methode SEARCH kann auf ähnliche Art und Weise realisiert werden. Wird nach einem Datum mit Schlüssel k gesucht, so wird erst getestet, ob sich ein solches Datum an der Position $h(k, 0)$ befindet. Ist diese Position leer, so gibt es kein Datum mit dem Schlüssel k in der Hashtabelle. Befindet sich dort ein anderes Datum, so muss weiter an der Position $h(k, 1)$ nach dem Datum gesucht werden und so weiter. Die folgende Methode gibt entweder die Position des Datums mit Schlüssel k oder -1 zurück, falls kein solches Datum in der Hashtabelle vorhanden ist.

```
SEARCH(k)
 1   for (int i = 0; i < m; i++) {
 2       j = h(k, i);
 3       if ((T[j] ≠ null) && (T[j].key == k)) {
 4           return j;
 5       } else if (T[j] == null) {
 6           return −1; //„nicht vorhanden“;
 7       }
 8   }
 9   return −1; // „nicht vorhanden“;
```

Solange zwischendurch keine Einträge aus der Hashtabelle gelöscht werden, erfüllt die Methode SEARCH ihren Zweck. Sie geht die Positionen in der Hashtabelle in derselben Reihenfolge durch wie die Methode INSERT. Wenn sie bei dieser Sondierung auf eine freie Position stößt, kann kein Datum mit Schlüssel k in der Hashtabelle vorhanden sein, da es ansonsten von INSERT dort eingefügt worden wäre.

Wir müssen nun aber beim Löschen von Elementen aufpassen. Die naheliegende Implementierung, die das zu löschende Datum zunächst sucht und anschließend den entsprechenden Eintrag in der Hashtabelle auf den Null-Zeiger setzt, beeinflusst die Korrektheit von SEARCH, wie das folgende Beispiel zeigt. Es gelte der Einfachheit halber $h(k, i) = i$ für alle Schlüssel $k \in U$ und für alle $i \in \{0, \ldots, m-1\}$. Werden

dann die Daten x_1 und x_2 mit $x_1.\text{key} \neq x_2.\text{key}$ nacheinander eingefügt, so wird x_1 an Position 0 gespeichert. Beim Einfügen von x_2 wird auch zunächst die Position 0 getestet. Da diese bereits belegt ist, wird als nächstes die Position 1 getestet, an der x_2 dann auch eingefügt wird. Wird anschließend nach $x_2.\text{key}$ gesucht, so wird ebenfalls zunächst die Position 0 getestet. Da diese mit einem anderen Schlüssel belegt ist, wird anschließend die Position 1 getestet, an der $x_2.\text{key}$ gefunden wird. Wird nun x_1 dadurch gelöscht, dass $T[0]$ auf den Null-Zeiger gesetzt wird, und wird anschließend noch einmal nach x_2 gesucht, so findet sich an Position $h(x_2.\text{key}, 0) = 0$ ein Null-Zeiger und die Suche wird fälschlicherweise mit dem Ergebnis abgebrochen, dass x_2 nicht in der Hashtabelle vorhanden ist.

Um dieses Problem zu umgehen, verwalten wir für jeden Eintrag $T[i]$ der Hashtabelle zusätzlich ein Bit Del$[i]$, das mit **false** initialisiert wird und angibt, ob an der Position i irgendwann ein Datum war, das gelöscht wurde. Die Methode DELETE kann dann wie folgt implementiert werden.

```
DELETE(k)
1   j = SEARCH(k);
2   if (j ≠ −1) {
3       Del[j] = true;
4       T[j] = null;
5   }
```

Die Methode SEARCH muss dann entsprechend angepasst werden. Sie darf erst dann abbrechen, wenn eine leere Position gefunden wird, an der noch keine Löschung stattgefunden hat. Zeile 5 des Pseudocodes von SEARCH muss dazu entsprechend durch

$$\text{} \} \textbf{ else if } ((T[j] == \textbf{null}) \;\&\&\; (\text{Del}[j] == \textbf{false})) \; \{$$

ersetzt werden.

Geschlossenes Hashing ist ungeeignet, wenn viele Löschungen auftreten, denn wenn aus sehr vielen Positionen bereits Daten gelöscht wurden, können erfolglose Suchen sehr lange dauern. In dem Extremfall, dass bereits aus jeder Position einmal ein Datum gelöscht wurde, beträgt die Laufzeit einer erfolglosen Suche sogar immer $\Theta(n)$. Aus diesem Grund sollte geschlossenes Hashing nur dann angewendet werden, wenn gar keine oder nur sehr wenige Löschungen auftreten.

Sondierungsreihenfolgen

Wir werden nun einige gängige Möglichkeiten diskutieren, wie eine Hashfunktion

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

realisiert werden kann. Wir gehen dabei davon aus, dass uns bereits eine „normale“ Hashfunktion $h'\colon U \to \{0, \ldots, m-1\}$ vorliegt.

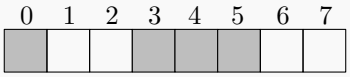
Beim *linearen Sondieren* werden für einen Schlüssel $k \in U$ nacheinander die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$ getestet. Formal setzen wir

$$h(k,i) = (h'(k) + i) \bmod m.$$

Diese einfache Herangehensweise besitzt den Nachteil, dass die belegten Zellen in dem Feld T die Tendenz haben, längere zusammenhängende Blöcke zu bilden, was sich negativ auf die Laufzeit der Operationen auswirkt.

Beispiel für lineare Sondierung

Es sei $m = 8$ und die Positionen 0, 3, 4 und 5 seien belegt.



Nun fügen wir ein neues Datum x mit Schlüssel k ein und gehen davon aus, dass der Funktionswert $h'(k)$ uniform zufällig aus der Menge $\{0, 1, \ldots, 7\}$ gewählt wird. Das Datum x wird an einer der vier freien Positionen eingefügt. Diese haben aber nicht dieselbe Wahrscheinlichkeit. Es sei Z die zufällige Position, an der das Element x eingefügt wird. Dann gelten die folgenden Aussagen.

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0, 1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$
- $\mathbf{Pr}[Z = 6] = \mathbf{Pr}[h'(k) \in \{3, 4, 5, 6\}] = 4/8 = 1/2$
- $\mathbf{Pr}[Z = 7] = \mathbf{Pr}[h'(k) = 7] = 1/8$

Demzufolge ist es also relativ wahrscheinlich, dass die bestehende Kette verlängert wird und das neue Element an Position 6 eingefügt wird.

Eine Alternative zum linearen Sondieren ist das *quadratische Sondieren*. Dabei werden die folgenden Positionen beim Sondieren durchlaufen, wobei jeweils modulo m gerechnet wird:

$$h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$$

Gilt für zwei Schlüssel k_1 und k_2 beispielsweise $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt also insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$. Dadurch wird der Tendenz, dass sich zusammenhängende Ketten von belegten Zellen bilden, vorgebeugt.

Eine weitere Möglichkeit, die im Allgemeinen noch bessere Eigenschaften besitzt, ist das *doppelte Hashing*. Bei diesem gehen wir davon aus, dass neben der Hashfunktion h' noch eine weitere Hashfunktion $h'' \colon U \to \{0, \ldots, m-1\}$ vorliegt. Die Hashfunktion h ist dann definiert als

$$h(k,i) = h'(k) + i \cdot h''(k) \bmod m.$$

Das doppelte Hashing ist nicht für jede Wahl von h' und h'' sinnvoll. Beispielsweise sollte die Funktion h'' nicht den Wert 0 annehmen. Wir gehen an dieser Stelle aber

nicht weiter ins Detail. Interessierte Leserinnen und Leser können zum Beispiel in dem Buch von Cormen et al. [1] tiefer in die Materie einsteigen.

Analyse von geschlossenem Hashing

Da geschlossenes Hashing ohnehin ungeeignet ist, wenn viele Daten gelöscht werden, gehen wir in der Analyse davon aus, dass keine Löschungen auftreten. Ebenso wie beim Hashing mit verketteten Listen treffen wir auch hier wieder die idealisierte Annahme des *uniformen Hashings*. Diese ist hier sogar weitergehend, denn wir gehen nicht nur davon aus, dass beim Einfügen eines Datum die erste getestete Position uniform zufällig gewählt ist, sondern wir gehen sogar davon aus, dass beim Sondieren die Positionen in einer uniform zufälligen Reihenfolge getestet werden. Wieder bezeichnen wir mit $\alpha = n/m$ den Auslastungsfaktor der Hashtabelle.

Theorem 4.13. *Unter der Annahme des uniformen Hashings untersucht eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, beim geschlossenen Hashing im Erwartungswert höchstens* $1/(1-\alpha)$ *Positionen.*

Beweis. Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Da die erste betrachtete Position beim uniformen Hashing zufällig gewählt wird, ist sie mit Wahrscheinlichkeit n/m belegt. Genau dann, wenn dies der Fall ist, muss eine zweite Position betrachtet werden. Es gilt also $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Genau dann, wenn die ersten beiden Positionen belegt sind, muss eine dritte Position betrachtet werden. Ist die erste Position belegt (Wahrscheinlichkeit n/m), so gibt es für die Wahl der zweiten Position noch $m-1$ Möglichkeiten. Von diesen $m-1$ Positionen sind $n-1$ belegt. Die Wahrscheinlichkeit, dass die zweite Position belegt ist, beträgt also unter der Annahme, dass die erste belegt ist, $(n-1)/(m-1)$. Dementsprechend gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren, dass die folgende Formel gilt:

  $$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

  Für $i > n$ gilt natürlich sogar $\mathbf{Pr}[X \geq i] = 0$. Die obige Abschätzung genügt jedoch für unsere Zwecke.

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha},$$

womit das Theorem bewiesen ist. □

Aus dem vorangegangenen Theorem folgt insbesondere, dass das Einfügen eines neuen Schlüssels unter der Annahme des uniformen Hashings im Erwartungswert eine Laufzeit von $O(1/(1-\alpha))$ besitzt.

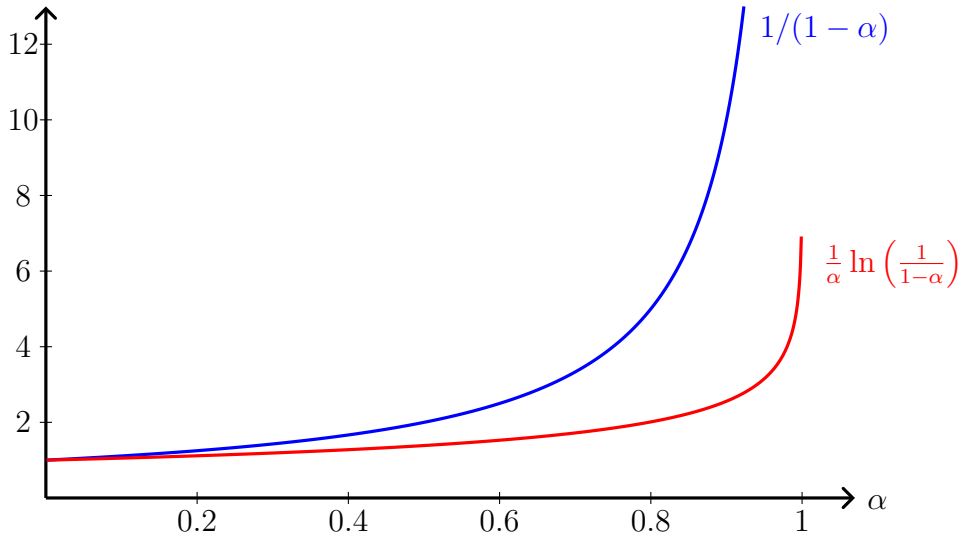
Theorem 4.14. *Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens* $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$ *Positionen.*

Beweis. Die Suche nach einem Schlüssel x erzeugt die gleiche Sequenz von Positionen wie das Einfügen des Schlüssels. Beim Einfügen des i-ten Schlüssels beträgt der Auslastungsfaktor $\alpha_i = (i-1)/m$ und demnach beträgt gemäß Theorem 4.13 die Anzahl der Positionen, die betrachtet werden, im Erwartungswert höchstens $1/(1-\alpha_i) = 1/(1-(i-1)/m)$. Die Bildung des Durchschnitts über alle Schlüssel ergibt nun

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-(i-1)/m} &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{k=m-n}^{m}\frac{1}{x}\,dx \\
&= \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) \\
&= \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right),
\end{aligned}$$

womit das Theorem bewiesen ist. □

Zur Verdeutlichung der beiden vorangegangenen Theoreme stellt das folgende Diagramm die Terme $1/(1-\alpha)$ und $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$ in Abhängigkeit vom Auslastungsfaktor α dar.

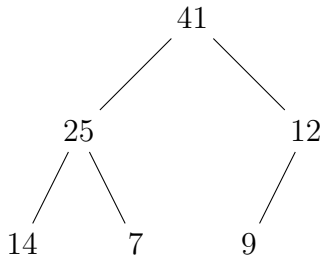


4.4 Heaps

Nun lernen wir mit *Heaps* eine Datenstruktur kennen, die auch dynamische Mengen verwalten kann, jedoch andere Operationen bereitstellt. Insbesondere fehlen die Operationen Search und Delete. Stattdessen ist es sehr leicht, das Maximum (sog. Max-Heap) oder Minimum (sog. Min-Heap) der gespeicherten Schlüssel zu ermitteln und zu entfernen. Wir werden Heaps nutzen um einen Sortieralgorithmus zu entwerfen und um Prioritätswarteschlangen zu realisieren. Letztere sind sehr hilfreich für effiziente Implementierungen von Greedy-Algorithmen.

In einem *Max-Heap* werden die Schlüssel in einem Binärbaum angeordnet, jedoch nicht wie in einem Suchbaum. Stattdessen ist zu jedem Zeitpunkt die folgende Bedingung erfüllt: Für jeden Knoten v mit Inhalt x gilt, dass in beiden Teilbäumen von v ausschließlich Daten gespeichert sind, deren Schlüssel höchstens so groß wie der Schlüssel von x sind. Anders formuliert enthält die Wurzel immer den größten im Max-Heap gespeicherten Schlüsselwert und die beiden Teilbäume der Wurzel sind wiederum Max-Heaps. Anders als in Suchbäumen wird jedoch keine Bedingung daran gestellt, wie sich die Schlüssel auf die Teilbäume verteilen.

Darüber hinaus ist der Binärbaum bis auf möglicherweise die letzte Ebene vollständig. Die letzte Ebene wird von links nach rechts gefüllt. Die folgende Abbildung zeigt ein Beispiel für einen Heap mit den Schlüsselwerten 7, 9, 12, 14, 25, 41.



Aufgrund der vollständigen Besetzung aller Ebenen lässt sich ein Heap leicht in einem Feld speichern. Wir bezeichnen im Folgenden mit heap-size die Anzahl der Elemente im Heap und gehen davon aus, dass sie in einem Feld a an den Positionen $1, \ldots,$ heap-size gespeichert sind. Die Wurzel befindet sich an Position 1. Die Kinder des Knotens an Position i werden an den Stellen $2i$ und $2i + 1$ gespeichert. Umgekehrt findet sich der Elternknoten zu dem an Position $i > 1$ gespeicherten Knoten an Stelle $\lfloor i/2 \rfloor$. Die Heap-Bedingung lässt sich demzufolge so umformulieren, dass im Feld a für alle i gelten muss

$$a[i].\text{key} \geq a[2i].\text{key} \quad \text{und} \quad a[i].\text{key} \geq a[2i+1].\text{key}. \tag{4.2}$$

Für solche i mit $2i >$ heap-size oder $2i + 1 >$ heap-size entfällt die entsprechende Bedingung in (4.2). Insbesondere ist die Bedingung trivialerweise für alle i mit $2i >$ heap-size erfüllt. Dies sind genau die Blätter des Baumes. Abbildung 4.1 zeigt ein Beispiel.

Analog lässt sich ein *Min-Heap* definieren. Hier enthält in jedem Teilbaum die Wurzel den kleinsten gespeicherten Schlüsselwert. Aus mathematischer Sicht ist dies jedoch

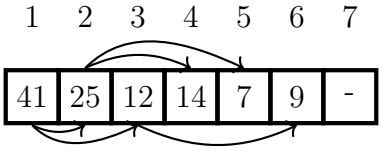


Abbildung 4.1: Max-Heap, wie er im Feld gespeichert wird

nichts anderes als eine Umkehrung der Ordnungsrelation. Deshalb konzentrieren wir uns im Folgenden auf Max-Heaps und die Bezeichnung *Heap* bezieht sich immer auf einen Max-Heap.

4.4.1 Heap-Eigenschaft herstellen

Um einen Heap aufzubauen, werden wir die Heap-Eigenschaft von unten nach oben im Baum herstellen. Betrachten wir also den Fall, dass in jedem Teilbaum die Heap-Eigenschaft gilt, jedoch möglicherweise nicht an der Wurzel. Das heißt, die Wurzel des Baums enthält nicht den maximalen Schlüssel aber beide Teilbäume sind gültige Heaps. Nun müssen wir den Schlüssel der Wurzel im Heap „versinken“ lassen. Dazu stellen wir fest, dass es nur drei Möglichkeiten gibt, wo sich das Maximum aller Schlüssel befinden kann: In der Wurzel selbst – das heißt, wir haben schon einen gültigen Heap – oder in einem der Kinder der Wurzel. Weitere Knoten sind ausgeschlossen, da die beiden Teilbäume unterhalb der Wurzel die Heap-Eigenschaft erfüllen.

Steht das Maximum nicht in der Wurzel, so können wir den Schlüssel in der Wurzel mit dem größeren Schlüssel ihrer Kinder vertauschen. Nun ist die Heap-Eigenschaft an der Wurzel sichergestellt, möglicherweise aber an demjenigen Kind verletzt, das wir gerade für den Tausch herangezogen haben. Dort können wir die Eigenschaft jedoch rekursiv herstellen, weil in diesem Teilbaum die Heap-Eigenschaft überall außer möglicherweise an der Wurzel erfüllt ist. Genau so geht die Subroutine MAX-HEAPIFY vor. Sie bekommt als Argument den Index der Wurzel des Teilbaums, für den die Heap-Eigenschaft überall gilt außer an der Wurzel, und stellt die Heap-Eigenschaft im gesamten Teilbaum her.

```
MAX-HEAPIFY(int i)
1   ℓ = 2i; r = 2i + 1; largest = i;
2   if (ℓ ≤ heap-size && a[ℓ].key > a[largest].key)
3       largest = ℓ;
4   if (r ≤ heap-size && a[r].key > a[largest].key)
5       largest = r;
6   if (largest ≠ i) {
7       vertausche a[i] und a[largest];
8       MAX-HEAPIFY(largest);
9   }
```

Sind n Daten im Heap enthalten, so beträgt die Laufzeit von MAX-HEAPIFY $O(\log n)$, denn mit jedem rekursiven Aufruf verdoppelt sich der Index. Die Schritte abgesehen vom rekursiven Aufruf benötigen nur konstante Zeit.

Lemma 4.15. *Gilt vor dem Aufruf von* MAX-HEAPIFY(i) *Ungleichung* (4.2) *für alle* $j > i$*, so gilt sie anschließend für alle* $j \geq i$*.*

Beweis. Wir führen den Beweis per Induktion über i, jedoch von großen zu kleinen Werten. Als Induktionsanfang nutzen wird, dass im Fall $2i >$ heap-size Ungleichung (4.2) trivialerweise erfüllt ist, weil i keine Kinder besitzt.

Für den Induktionsschritt betrachten wir nun ein i mit $2i \leq$ heap-size. Die Induktionsvoraussetzung besagt nun Folgendes: Wird MAX-HEAPIFY(i') für ein $i' > i$ aufgerufen und gilt zuvor Ungleichung (4.2) für alle $j > i'$, so gilt sie anschließend für alle $j \geq i'$. Wir müssen nun folgern, dass nach Aufruf von MAX-HEAPIFY(i), wenn zuvor Ungleichung (4.2) für alle $j > i$ gilt, anschließend Ungleichung (4.2) für alle $j \geq i$ gilt.

Hierfür unterscheiden wir zwei Fälle. Ist largest $= i$, so gilt Ungleichung (4.2) bereits vor der Ausführung von MAX-HEAPIFY(i) und der Heap bleibt unverändert. Anderenfalls gilt largest $> i$ und wir vertauschen $a[i]$ mit $a[\text{largest}]$. Dadurch ist Ungleichung (4.2) für i erfüllt. Die einzige Position aus dem Bereich $i, \ldots,$ heap-size, für die die Heap-Eigenschaft möglicherweise verletzt sein kann, ist largest. Wegen largest $> i$ dürfen wir die Induktionsvoraussetzung nutzen. Laut dieser stellt MAX-HEAPIFY(largest) sicher, dass Ungleichung (4.2) für alle $j \geq$ largest gilt. Für alle Positionen aus dem Bereich $i, \ldots,$ largest $- 1$ gilt die Ungleichung weiterhin, da sich für diese Positionen durch den Aufruf von MAX-HEAPIFY(largest) nichts verändert. □

Ist uns bereits ein Feld $a[1], \ldots, a[n]$ von Daten gegeben, können wir durch wiederholten Aufruf von MAX-HEAPIFY dieses Feld in einen Heap umbauen.

```
BUILD-HEAP()
1   for (i = ⌊a.length/2⌋; i ≥ 1; i−−) {
2       MAX-HEAPIFY(i);
3   }
```

Die Laufzeit von BUILD-HEAP ist beschränkt durch $O(n \log n)$, denn es gibt $O(n)$ Schleifendurchläufe, in denen jeweils MAX-HEAPIFY ausgeführt wird, dessen Laufzeit $O(\log n)$ beträgt.

Lemma 4.16. *Nach der Ausführung von* BUILD-HEAP *gilt Ungleichung* (4.2) *für alle* i*.*

Beweis. Diese Aussage kann leicht mit einer Invariante gezeigt werden. Für das aktuelle i gilt in Zeile 1 jeweils, dass Ungleichung (4.2) für alle $j > i$ erfüllt ist. Dabei können wir ausnutzen, dass für $i > \lfloor a.\text{length}/2 \rfloor$ Ungleichung (4.2) immer gilt, weil die entsprechenden Knoten keine Kinder haben. Lemma 4.15 besagt nun, dass nach dem Aufruf MAX-HEAPIFY(i) Ungleichung (4.2) für alle $j \geq i$ erfüllt ist. Gemäß der Invariante gilt nach Beendigung der Schleife Ungleichung (4.2) für alle $i > 0$, also im gesamten Heap. Dies ist genau die Aussage des Lemmas. □

4.4.2 Heapsort

Mittels eines Heaps können wir sehr einfach das Sortierproblem lösen. In der Wurzel finden wir immer das maximale Element, das an die letzte Stelle in der sortierten Folge gehört. Dieses müssen wir entfernen und auf den verbleibenden Elementen wieder einen Heap aufbauen.

Durch ein geschicktes Vorgehen schafft es der Algorithmus ohne zusätzlichen Speicherplatz auszukommen. Er verkürzt den Bereich, der im Feld für den Heap vorgesehen ist, um einen Eintrag. Den Schlüssel, der aus dem für den Heap vorgesehenen Bereich herausfällt, setzt er in die Wurzel und an die nun frei gewordene Stelle setzt er das Maximum. Mit einem Aufruf von MAX-HEAPIFY kann nun die Heap-Eigenschaft wiederhergestellt werden. Dieses Verfahren wird wiederholt, bis der Heap leer ist.

```
HEAPSORT(int[] a)
      // Das Feld a besitzt die Positionen 1, ..., n.
 1    BUILD-HEAP();
 2    heap-size = n;
 3    while (heap-size > 0) {
 4         vertausche a[1] und a[heap-size];
 5         heap-size−−;
 6         MAX-HEAPIFY(1);
 7    }
```

Theorem 4.17. HEAPSORT *sortiert jedes Feld mit n Einträgen in Zeit $O(n \log n)$.*

Beweis. Die Laufzeit ist leicht abgeschätzt: Wie oben gezeigt, benötigt BUILD-HEAP $O(n \log n)$ Operationen. Die while-Schleife wird n-mal durchlaufen, denn in jedem Durchlauf reduziert sich die Größe des Heaps um 1. In jedem Durchlauf wird einmal MAX-HEAPIFY(1) aufgerufen. Dies benötigt $O(\log n)$ Operationen. Die übrigen Schritte benötigen konstant viele Operationen.

Zur Korrektheit kann folgende Invariante verwendet werden: Jeweils in Zeile 2 bilden $a[1], \ldots, a[\text{heap-size}]$ einen Heap, der die heap-size kleinsten Elemente enthält. Ferner enthalten $a[\text{heap-size} + 1], \ldots, a[n]$ die größten $n -$ heap-size vielen Elemente in sortierter Reihenfolge. Diese Invariante kann per Induktion bewiesen werden, wie wir es bereits zuvor vielfach gesehen haben. □

Somit haben wir einen weiteren Sortieralgorithmus mit Laufzeit $O(n \log n)$ kennengelernt. Im Gegensatz zu Mergesort benötigt er nur konstant viel zusätzlichen Speicherplatz, ist also *in situ*. Sehr bemerkenswert ist, dass der gesamte Effizienzgewinn darin begründet ist, dass wir eine passende Datenstruktur ausgewählt haben. Dies zeigt, wie wichtig diese Wahl im Entwurf von effizienten Algorithmen ist.

4.4.3 Prioritätswarteschlangen

Als zweites Anwendungsbeispiel von Heaps werden wir nun Prioritätswarteschlagen kennenlernen. Hierbei handelt es sich um eine Art von dynamischer Menge. Für eine Menge bestehend aus Schlüssel/Daten-Paaren stehen uns folgende Operationen zur Verfügung:

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.
- FIND-MAX(): Gib ein Element mit dem größtem Schlüssel zurück.
- EXTRACT-MAX(): Entferne ein Element mit dem größtem Schlüssel und gib dieses Element zurück.
- INCREASE-KEY(i, k): Erhöhe den Schlüssel des an Stelle i gespeicherten Objektes auf k.

Bevor wir zu INSERT kommen, betrachten wir zunächst die anderen Operationen. Die Operation FIND-MAX ist in unserem Heap leicht implementiert. Wir geben einfach das an Stelle $a[1]$ gespeicherte Element zurück. Dies benötigt nur konstante Zeit.

Bei der Operation EXTRACT-MAX müssen wir zusätzlich auch das Datum mit maximalem Schlüssel entfernen. Eine Möglichkeit dafür haben wir bereits bei Heapsort kennengelernt. Wir verschieben das letzte zum Heap zugehörige Element des Feldes an Position 1 und rufen anschließend MAX-HEAPIFY auf.

```
EXTRACT-MAX()
1   if (heap-size > 0) {
2       max = a[1];
3       a[1] = a[heap-size];
4       heap-size−−;
5       MAX-HEAPIFY(1);
6       return max;
7   }
```

Bei n gespeicherten Daten hat EXTRACT-MAX eine Laufzeit von $O(\log n)$.

Bei einem Aufruf von INCREASE-KEY müssen wir einen Schlüssel erhöhen. Dies kann zur Folge haben, dass die Heap-Eigenschaft verletzt ist. In solchen Fällen haben wir bislang Max-Heapify eingesetzt, das jeweils das Startelement im Heap „versinken“ lässt. Im Falle der Vergrößerung eines Schlüssels müssen wir genau das Gegenteil machen, nämlich das aktuelle Element möglicherweise in Richtung der Wurzel verschieben. Hierzu tauscht INCREASE-KEY es so lange wie erforderlich jeweils mit dem Elternknoten.

```
INCREASE-KEY(i,k)
1   a[i].key = k;
2   while (i > 1 && a[i].key > a[⌊i/2⌋].key) {
3       vertausche a[i] und a[⌊i/2⌋];
4       i = ⌊i/2⌋;
5   }
```

Auch INCREASE-KEY hat eine Laufzeit von $O(\log n)$. Eine wichtige Eigenschaft ist, dass als Parameter die Position im Feld übergeben wird, nicht jedoch das Datum x oder sein Schlüssel. Im Normalfall wird man jedoch den Schlüssel eines bestimmten Datums x verändern wollen. Diesen im Heap zu finden braucht jedoch lineare Zeit, weil der Heap im Wesentlichen unstrukturiert ist. Deshalb ist es erforderlich, für die Daten im Heap nachzuhalten, an welchen Stellen sie sich befinden, wenn ihr Schlüssel geändert werden soll.

Um schließlich ein Element in den Heap einzufügen, können wir es zunächst mit Schlüssel $-\infty$ ans Ende des Feldes setzen, ohne die Heap-Eigenschaft zu verletzen. Nun können wir INCREASE-KEY nutzen, um den Schlüssel auf den eigentlich geforderten Wert zu setzen.

```
INSERT(x,k)
1   heap-size++;
2   a[heap-size] = x;
3   a[heap-size].key = −∞;
4   INCREASE-KEY(heap-size, k);
```

Dies benötigt offensichtlich nur eine konstante Anzahl von Schritten über INCREASE-KEY hinaus, insgesamt also auch $O(\log n)$ Schritte. Zusammenfassend können wir das folgende Theorem festhalten.

Theorem 4.18. *Die Laufzeit der Operationen* EXTRACT-MAX, INCREASE-KEY *und* INSERT *beträgt in einem Heap mit n gespeicherten Daten $O(\log n)$. Die Operation* FIND-MAX *benötigt nur konstante Laufzeit.*

Kapitel 5

Graphenalgorithmen

Graphen spielen in allen Bereichen der Informatik eine wichtige Rolle, da sie zur Modellierung unterschiedlichster Probleme eingesetzt werden können. Ein *ungerichteter Graph* $G = (V, E)$ besteht aus einer endlichen *Knotenmenge* V und einer *Kantenmenge* E. Eine *Kante* $e = \{u, v\} \in E$ ist eine zweielementige Teilmenge der Knotenmenge V. In einem *gerichteten Graphen* $G = (V, E)$ ist V ebenfalls eine endliche Knotenmenge, eine Kante $e = (u, v) \in E$ ist nun aber ein geordnetes Paar von zwei Knoten aus V. Geht aus dem Kontext klar hervor, dass wir über ungerichtete Graphen sprechen, so benutzen wir manchmal auch dort für Kanten die Notation (u, v) statt $\{u, v\}$. Gemeint ist bei ungerichteten Graphen aber stets die zweielementige Teilmenge und nicht das geordnete Paar. Sowohl ungerichtete als auch gerichtete Graphen sind in vielen Anwendungen *gewichtet*. Das bedeutet, dass eine Funktion $w\colon E \to \mathbb{R}$ gegeben ist, die jeder Kante $e \in E$ ein *Gewicht* $w(e)$ zuweist.

In einem ungerichteten Graphen $G = (V, E)$ heißen zwei Knoten $u \in V$ und $v \in V$ *adjazent*, wenn $\{u, v\} \in E$ gilt. Der Knoten $u \in V$ und die Kante $\{u, v\} \in E$ heißen *inzident* (ebenso der Knoten v und die Kante $\{u, v\}$). Die Anzahl an inzidenten Kanten eines Knotens $v \in V$ bezeichnen wir als seinen *Grad* $d(v)$. Bei gerichteten Graphen $G = (V, E)$ unterscheiden wir für einen Knoten $v \in V$ seine *eingehenden* und seine *ausgehenden* Kanten. Dabei heißt jede Kante der Form $(v, u) \in E$ von v ausgehende und jede Kante der Form $(u, v) \in E$ in v eingehende Kante. Für eine Kante $(u, v) \in E$ sagen wir, dass v ein *direkter Nachfolger* von u und u ein *direkter Vorgänger* von v ist. Die Anzahl der aus einem Knoten $v \in V$ ausgehenden Kanten bezeichnen wir als seinen *Outgrad* $d^+(v)$ und die Anzahl der in einen Knoten $v \in V$ eingehenden Kanten bezeichnen wir als seinen *Ingrad* $d^-(v)$.

Eine Folge $v_0, \ldots, v_\ell \in V$ von Knoten heißt *Weg von* v_0 *nach* v_ℓ *der Länge* $\ell \in \mathbb{N}_0$, wenn $(v_{i-1}, v_i) \in E$ für alle $i \in \{1, \ldots, \ell\}$ gilt. Mit der oben erwähnten Interpretation von (v_{i-1}, v_i) als $\{v_{i-1}, v_i\}$ bei ungerichteten Graphen, gilt diese Definition sowohl für gerichtete als auch für ungerichtete Graphen. Anstatt der Folge von Knoten bezeichnen wir auch die entsprechende Folge von Kanten $(v_0, v_1), \ldots, (v_{\ell-1}, v_\ell) \in E$ als Weg der Länge ℓ. Ein Weg $v_0, \ldots, v_\ell$ heißt *einfach*, wenn alle Knoten auf dem Weg paarweise verschieden sind. Der Weg heißt *Kreis*, wenn $v_0 = v_\ell$ und $\ell \geq 1$ gilt, und er heißt

einfacher Kreis, wenn $v_0 = v_\ell$ gilt und alle anderen Knoten paarweise verschieden und verschieden von v_0 sind. In ungerichteten Graphen verlangen wir von Kreisen zusätzlich, dass sie einfach sind und mindestens die Länge drei haben ($\ell \geq 3$).

Ein Graph, der keinen Kreis enthält, heißt *kreisfrei*, *azyklisch* oder *Wald*. Ein ungerichteter Graph heißt *zusammenhängend*, wenn es zwischen jedem Paar von Knoten einen Weg gibt. Ein zusammenhängender Graph, der keinen Kreis besitzt, heißt *Baum*.

Wir werden uns in diesem Kapitel mit sogenannten *Graphenalgorithmen* beschäftigen. Das sind Algorithmen, die Probleme lösen, die im Kontext von Graphen auftreten. Eine grundlegende Frage in vielen Anwendungen ist beispielsweise, ob ein gegebener ungerichteter Graph $G = (V, E)$ zusammenhängend ist. Modelliert der Graph eine Straßenkarte, so ist es zum Beispiel naheliegend, für zwei Knoten $u, v \in V$ nach dem kürzesten Weg von u nach v zu fragen. Wir werden effiziente Algorithmen für diese und weitere Probleme kennenlernen. Vorher müssen wir uns allerdings noch Gedanken dazu machen, welche Datenstrukturen es gibt, um Graphen zu speichern.

Eine naheliegende und universelle Datenstruktur für Graphen sind *Adjazenzmatrizen*. Sei $G = (V, E)$ ein Graph mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$. Bei der Adjazenzmatrix $A = (a_{ij})$ von G handelt es sich um eine $n \times n$-Matrix. Ist der Graph nicht gewichtet, so gilt

$$a_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E, \\ 0 & \text{sonst.} \end{cases}$$

Interpretieren wir (v_i, v_j) bei einem ungerichteten Graphen wieder als $\{v_i, v_j\}$, so ist diese Definition sowohl für gerichtete als auch für ungerichtete Graphen gültig. Bei ungerichteten Graphen handelt es sich bei A um eine symmetrische Matrix. Ist der Graph gewichtet, so setzt man den Eintrag a_{ij} auf das Gewicht $w((v_i, v_j))$, sofern die Kante (v_i, v_j) in dem Graphen existiert. Ist das nicht der Fall, so wird dies entsprechend in der Matrix vermerkt (beispielsweise durch ein zusätzliches Bit pro Eintrag oder je nach Graph durch einen bestimmten Wert, der nicht als Gewicht in Frage kommt).

Adjazenzmatrizen haben eine Größe von $\Theta(n^2)$. *Adjazenzlisten* bieten eine platzsparendere Möglichkeit, Graphen mit $o(n^2)$ Kanten darzustellen. Eine Adjazenzliste ist ein Feld von n verketteten Listen. Für jeden Knoten $v \in V$ gibt es eine Liste in dem Feld, in der bei ungerichteten Graphen alle zu v adjazenten Knoten und bei gerichteten Graphen alle direkten Nachfolger von v gespeichert sind. Diese Datenstruktur kann wieder kanonisch an gewichtete Graphen angepasst werden. Sie benötigt einen Speicherplatz von $\Theta(n + m)$, wobei m die Anzahl der Kanten von G bezeichnet.

5.1 Tiefen- und Breitensuche

Bei *Tiefen- und Breitensuche* handelt es sich um zwei Algorithmen, die die Knoten eines gegebenen Graphen in einer bestimmten Reihenfolge durchlaufen. Bei diesen Durchläufen werden wichtige strukturelle Informationen über den Graphen gesammelt, wie zum Beispiel, ob er zusammenhängend oder azyklisch ist.

5.1.1 Tiefensuche

Der Algorithmus *Tiefensuche* oder *Depth-First Search (DFS)* startet die Exploration des Graphen an einem beliebigen Knoten. Von dort aus folgt er Kanten, solange bis er einen Knoten erreicht, dessen Nachbarn bereits alle besucht wurden. Von diesem aus erfolgt dann ein Backtracking. Die genaue Arbeitsweise ist im folgenden Pseudocode dargestellt. Der Methode DFS wird als Eingabe ein Graph $G = (V, E)$ übergeben. Dieser kann sowohl gerichtet als auch ungerichtet sein. Wir gehen davon aus, dass jeder Knoten u des Graphen vier Attribute besitzt, die wir setzen können. Das sind die Farbe u.color, der Vorgänger $u.\pi$, der Zeitpunkt $u.d$ (für *discover*), zu dem die Tiefensuche den Knoten das erste Mal erreicht, und der Zeitpunkt $u.f$ (für *finish*), zu dem die Bearbeitung des Knotens u abgeschlossen ist. Außerdem benutzen wir time als globale Variable.

```
DFS(G = (V, E))
 1   for each (u ∈ V) {
 2       u.color = weiß;
 3       u.π = null;
 4   }
 5   time = 0;
 6   for each (u ∈ V)
 7       if (u.color == weiß) DFS-Visit(G, u);
```

Die folgende Methode DFS-Visit wird von DFS aufgerufen.

```
DFS-Visit(G = (V, E), u)
 1   time++;
 2   u.d = time;
 3   u.color = grau;
 4   for each ((u, v) ∈ E) {
 5       if (v.color == weiß) {
 6           v.π = u;
 7           DFS-Visit(G, v);
 8       }
 9   }
10   u.color = schwarz;
11   time++;
12   u.f = time;
```

Die Funktionsweise von Tiefensuche und die Bedeutung der Attribute der Knoten sind schnell erklärt. Zu Beginn wird in DFS jedem Knoten die Farbe Weiß zugewiesen. Ein Knoten u bleibt solange weiß, bis er von der Tiefensuche das erste Mal erreicht wird. Dann wird seine Farbe in DFS-Visit von weiß auf grau gesetzt. Sind alle Knoten abgearbeitet, die von u aus erreicht werden können, so ist die Bearbeitung von u

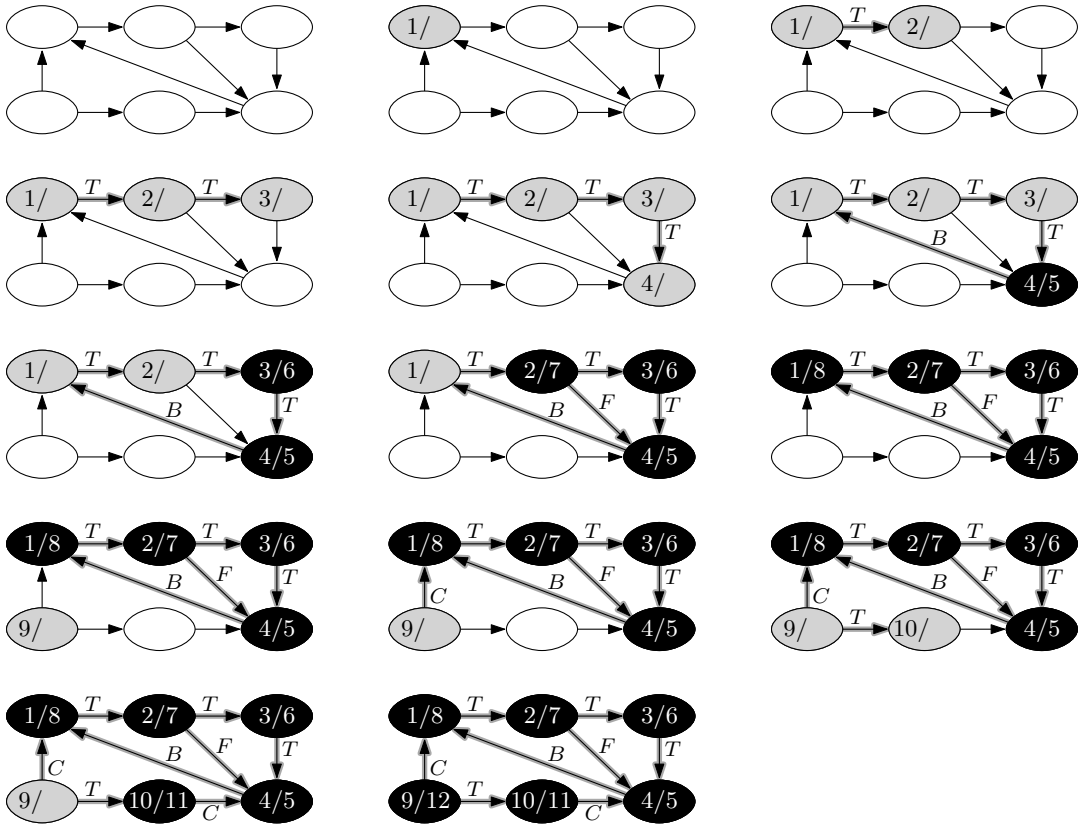


Abbildung 5.1: Beispiel für die Ausführung von DFS. Bei den beiden Werten innerhalb eines Knotens u handelt es sich um $u.d$ und $u.f$. Die mit T markierten Kanten codieren die Vorgängerbeziehung.

abgeschlossen und seine Farbe wird in Zeile 10 von DFS-Visit von grau auf schwarz gesetzt. Die for-Schleife in den Zeilen 6 und 7 von DFS stellt sicher, dass jeder Knoten von der Tiefensuche erreicht wird, da sie für jeden weißen Knoten (also für jeden Knoten, der noch nicht betrachtet wurde), einen Aufruf von DFS-Visit generiert. Der Vorgänger $u.\pi$ bezeichnet den Knoten, von dem aus u in der Tiefensuche erreicht wird. Ist $u.\pi$ der Null-Zeiger, so wurde u von keinem anderen Knoten erreicht, sondern der entsprechende Aufruf von DFS-Visit wurde in Zeile 7 von DFS generiert. Die Attribute $u.d$ und $u.f$ speichern die Zeitpunkte, zu denen der Knoten u das erste Mal erreicht wird bzw. komplett abgearbeitet ist. Dazu wird der Zähler time verwaltet, der nach jedem solchen Ereignis erhöht wird.

In Abbildung 5.1 ist die Ausführung von Tiefensuche an einem Beispiel dargestellt. Dort ist ebenfalls zu sehen, dass Tiefensuche die Menge der Kanten in die vier Mengen T, B, F und C partitioniert. Bevor wir diese Einteilung im Detail beschreiben, beweisen wir zunächst eine grundlegende Aussage über die Attribute $u.d$ und $u.f$ der Knoten $u \in V$. Wir sagen im Folgenden, dass ein Knoten $v \in V$ ein DFS-Nachfolger von $u \in V$ ist, wenn es einen Weg $u = v_0, v_1, \ldots, v_\ell = v$ gibt, sodass $v_i.\pi = v_{i-1}$ für alle $i \in \{1, \ldots, \ell\}$ gilt. Anschaulich ist v also genau dann ein DFS-Nachfolger von u, wenn die Tiefensuche von u aus einem Weg zu v folgt.

Lemma 5.1. *Ein Knoten $v \in V$ ist genau dann ein DFS-Nachfolger eines Knotens $u \in$*

V, wenn der Knoten u zum Zeitpunkt $v.d$ des ersten Besuches von v grau ist.

Beweisidee. Sei ein beliebiger Zeitpunkt fixiert und seien $u_1, \ldots, u_\ell$ die Knoten, die zu diesem Zeitpunkt grau sind. Diese seien so sortiert, dass $u_1.d < \ldots < u_\ell.d$ gilt. Mit vollständiger Induktion kann man leicht nachweisen, dass $u_1, \ldots, u_\ell$ ein einfacher Weg ist und dass der Knoten u_i für jedes $i \in \{1, \ldots, \ell\}$ genau die DFS-Vorgänger $u_1, \ldots, u_{i-1}$ besitzt. Weiterhin kann man nachweisen, dass zum aktuellen Zeitpunkt nur von dem Knoten u_ℓ aus ein neuer Knoten erreicht werden kann. Daraus folgt das Lemma, denn wird von u_ℓ aus ein neuer Knoten v erreicht, so besitzt er genau die momentan grauen Knoten $u_1, \ldots, u_\ell$ als DFS-Vorgänger. □

Theorem 5.2. *Für jeden (gerichteten oder ungerichteten) Graphen $G = (V, E)$ gilt für jedes Paar $u \in V$ und $v \in V$ von zwei verschiedenen Knoten genau eine der folgenden drei Aussagen.*

1. *Die Intervalle $[u.d, u.f]$ und $[v.d, v.f]$ sind disjunkt und weder u ist ein DFS-Nachfolger von v noch andersherum.*
2. *Es gilt $[u.d, u.f] \subseteq [v.d, v.f]$ und u ist ein DFS-Nachfolger von v.*
3. *Es gilt $[v.d, v.f] \subseteq [u.d, u.f]$ und v ist ein DFS-Nachfolger von u.*

Beweis. Wir betrachten nur den Fall, dass $u.d < v.d$ gilt, da der Fall $u.d > v.d$ symmetrisch ist und der Fall $u.d = v.d$ wegen $u \neq v$ nicht eintreten kann. Da in dem betrachteten Fall der Knoten u vor dem Knoten v besucht wird, ist u kein DFS-Nachfolger von v. Gilt $v.d > u.f$, so sind die Intervalle $[u.d, u.f]$ und $[v.d, v.f]$ disjunkt. In diesem Fall ist der Knoten u bereits schwarz, wenn der Knoten v erreicht wird. Gemäß Lemma 5.1 ist v demnach kein DFS-Nachfolger von u. Damit tritt die erste Aussage des Theorems ein (nicht aber die zweite oder dritte).

Gilt $v.d < u.f$, so ist der Knoten u zu dem Zeitpunkt, zu dem v erreicht wird, grau. Gemäß Lemma 5.1 ist v damit ein DFS-Nachfolger von u. Außerdem gilt $v.f < u.f$, denn die Tiefensuche arbeitet erst alle von v ausgehenden Kanten ab, bevor ein Backtracking zu u erfolgt. Damit ist gezeigt, dass die dritte Aussage des Theorems eintritt (nicht aber die erste oder zweite). □

Wir beschreiben nun die Einteilung der Kanten, die die Tiefensuche vornimmt. Dazu gehen wir davon aus, dass wir uns in einem Aufruf DFS-Visit(G, u) befinden und in Zeile 4 eine Kante (u, v) betrachten. Bei ungerichteten Graphen interessiert uns nur das erste Mal, dass wir die Kante $(u, v) = \{u, v\}$ betrachten. Dann tritt genau einer der folgenden vier Fälle ein.

1. Ist v.color = weiß, so wurde der Knoten v bislang noch nicht erreicht und die Tiefensuche wird mit ihm fortgesetzt. In diesem Fall ist (u, v) eine *Tree-* oder *T-Kante*. Dies sind genau die Kanten, für die $v.\pi = u$ gilt.
2. Ist v.color = grau, so wurde der Knoten v bereits in der Tiefensuche erreicht, seine Bearbeitung ist aber noch nicht abgeschlossen. Dies bedeutet, dass der Knoten u in der Tiefensuche über einen Weg von v aus erreicht wurde. In diesem Fall ist (u, v) eine *Back-* oder *B-Kante*.

3. Ist v.color = schwarz, so wurde der Knoten v bereits in der Tiefensuche erreicht und seine Bearbeitung ist komplett abgeschlossen. Ist $v.d > u.d$, so wurde v über einen Weg von u erreicht. In diesem Fall ist (u, v) eine *Forward-* oder *F-Kante.*
4. Ist v.color = schwarz, so wurde der Knoten v bereits in der Tiefensuche erreicht und seine Bearbeitung ist komplett abgeschlossen. Ist $v.d < u.d$, so wurde weder v über einen Weg von u noch u über einen Weg von v erreicht. In diesem Fall ist (u, v) eine *Cross-* oder *C-Kante.*

Die folgende Beobachtung zeigt, dass bei ungerichteten Graphen nur zwei verschiedene Arten von Kanten auftreten können.

Lemma 5.3. *In einem ungerichteten Graphen erzeugt die Tiefensuche nur T- und B-Kanten.*

Beweis. Es sei $\{u, v\}$ eine beliebige Kante des Graphen. Ohne Beschränkung der Allgemeinheit gelte $u.d < v.d$. Bevor der Knoten u komplett abgearbeitet ist, wird die Kante $\{u, v\}$ betrachtet. Passiert dies das erste Mal ausgehend von u, so ist v zu diesem Zeitpunkt noch weiß und $\{u, v\}$ wird eine T-Kante. Wird v von u aus nicht direkt über die Kante $\{u, v\}$, sondern über Zwischenknoten erreicht, so wird die Kante $\{u, v\}$ das erste Mal von v ausgehend betrachtet. Sie wird dann eine B-Kante, da u zu diesem Zeitpunkt noch grau ist. □

Die Einteilung der Kanten bei einer Tiefensuche liefert ein einfaches Kriterium, um zu testen, ob der gegebene Graph einen Kreis enthält.

Theorem 5.4. *Ein ungerichteter oder gerichteter Graph G ist genau dann kreisfrei, wenn bei der Tiefensuche keine B-Kante erzeugt wird.*

Beweis. Falls die Tiefensuche eine B-Kante (u, v) erzeugt, so ist v noch grau, wenn u erreicht wird. Das bedeutet, dass u gemäß Lemma 5.1 ein DFS-Nachfolger von v ist. Demnach gibt es einen Weg von v nach u, der zusammen mit der Kante (u, v) einen Kreis bildet.

Wir müssen nun noch die andere Richtung zeigen: Wenn ein Kreis in G existiert, so wird von der Tiefensuche eine B-Kante erzeugt. Wir betrachten nur den Fall, dass G gerichtet ist. Für ungerichtete Graphen kann das Argument leicht angepasst werden. Sei C ein Kreis und sei v der erste Knoten des Kreises, der von der Tiefensuche erreicht wird. Es sei (u, v) die Kante auf dem Kreis, von der aus v erreicht wird, d. h. u ist der Vorgänger von v auf dem Kreis C. Zu diesem Zeitpunkt sind alle anderen Knoten auf dem Kreis noch weiß. Daraus folgt, dass alle Knoten auf dem Kreis besucht werden, bevor v abgearbeitet ist. Um dies zu zeigen, führen wir einen Widerspruchsbeweis und gehen davon aus, dass es einen Knoten w auf dem Kreis gibt, der nicht erreicht wird, solange v grau ist. Sei w der erste solche Knoten, der auf dem Kreis nach v kommt. Dann wird der Vorgänger x von w besucht, während v grau ist. Da es die Kante (x, w) gibt, wird w besucht, solange x noch grau ist. Solange x noch grau ist, ist auch v grau. Dies führt zu einem Widerspruch, da w besucht wird, während v grau ist. Somit wird insbesondere der Knoten u besucht, während v grau ist. Von dort wird die Kante (u, v) betrachtet und als B-Kante markiert. □

Sei nun $G = (V, E)$ ein ungerichteter Graph. Gibt es einen Weg von $v \in V$ nach $u \in V$, so sagen wir, dass u von v aus erreichbar ist und wir schreiben $v \rightsquigarrow u$. Es ist eine gute Übung und Wiederholung für die Leserin und den Leser, zu beweisen, dass es sich bei $\rightsquigarrow$ um eine Äquivalenzrelation handelt. Das bedeutet, dass $\rightsquigarrow$ die Knoten des Graphen in Klassen einteilt, sodass es zwischen je zwei Knoten aus derselben Klasse einen Weg und zwischen Knoten aus verschiedenen Klassen keinen Weg gibt. Wir nennen diese Klassen die *Zusammenhangskomponenten* des Graphen. Ein Graph ist genau dann zusammenhängend, wenn alle seine Knoten zur selben Zusammenhangskomponente gehören.

Das folgende Theorem besagt, dass wir anhand der T-Kanten die Zusammenhangskomponenten eines Graphen identifizieren können.

Theorem 5.5. *In einem ungerichteten Graphen bilden die T-Kanten einen Wald, dessen Zusammenhangskomponenten genau die Zusammenhangskomponenten des Graphen sind.*

Beweis. Man überlegt sich leicht mit einem Widerspruchsbeweis, dass es keinen Kreis von T-Kanten geben kann, da eine T-Kante nur dann zwischen zwei Knoten eingefügt wird, wenn einer davon noch nicht besucht wurde. Zwischen Knoten aus verschiedenen Zusammenhangskomponenten gibt es keinen Weg. Sie können also insbesondere nicht durch T-Kanten verbunden sein. Es bleibt somit nur zu zeigen, dass es zwischen zwei Knoten aus derselben Zusammenhangskomponente einen Weg über T-Kanten gibt. Sei eine beliebige Zusammenhangskomponenten fixiert und sei u der erste Knoten aus dieser Komponente, der von der Tiefensuche besucht wird. Ähnlich wie im Beweis von Theorem 5.4 kann man argumentieren, dass jeder Knoten v, der von u aus erreichbar ist, ein DFS-Nachfolger von u wird. Das bedeutet, es gibt einen Weg aus T-Kanten von u zu jedem Knoten v der Zusammenhangskomponente. Damit gibt es zwischen zwei beliebigen Knoten v_1 und v_2 der Zusammenhangskomponente einen Weg aus T-Kanten (man verbinde dazu einfach die beiden Wege über u). □

Um zu testen, ob ein gegebener ungerichteter Graph zusammenhängend ist, genügt es also, die Tiefensuche an einem beliebigen Knoten zu starten und zu testen, ob alle anderen Knoten über T-Kanten erreicht werden. Dies ist äquivalent dazu, dass in Zeile 7 von DFS lediglich ein Aufruf von DFS-Visit erfolgt. Allgemein entspricht die Anzahl dieser Aufrufe der Anzahl an Zusammenhangskomponenten.

Theorem 5.6. *Die Laufzeit von Tiefensuche auf einem Graphen $G = (V, E)$ beträgt $\Theta(|V| + |E|)$, wenn der Graph als Adjazenzliste gegeben ist.*

Beweis. In der for-Schleife in DFS werden alle Knoten einmal betrachtet. Ohne die Aufrufe von DFS-Visit besitzt DFS somit eine Laufzeit von $\Theta(|V|)$. Jede Kante wird bei gerichteten Graphen genau einmal und bei ungerichteten Graphen genau zweimal in Zeile 4 von DFS-Visit betrachtet. Daher besitzen alle Aufrufe von DFS-Visit zusammen eine Laufzeit von $\Theta(|V| + |E|)$. □

5.1.2 Breitensuche

Breitensuche oder *Breadth-First Search (BFS)* ist genauso wie Tiefensuche ein Verfahren, das die Knoten eines gegebenen Graphen besucht. Im Gegensatz zur Tiefensuche werden bei einer Breitensuche, die an einem Knoten s gestartet wird, zunächst nur die direkten Nachfolger von s (bzw. die zu s adjazenten Knoten) besucht. Erst wenn diese Knoten alle besucht wurden, werden die direkten Nachfolger der direkten Nachfolger von s (bzw. die Knoten, die zu s adjazenten Knoten adjazent sind) besucht und so weiter.

Der Pseudocode von Breitensuche ist im Folgenden dargestellt. Dabei kann G sowohl einen gerichteten als auch einen ungerichteten Graphen bezeichnen. Wieder besitzt jeder Knoten u die Attribute u.color und $u.\pi$ mit derselben Bedeutung wie bei Tiefensuche. Weiterhin existiert ein Attribut $u.d$, das nun aber eine andere Bedeutung als bei der Tiefensuche hat. Es gibt nicht mehr den Zeitpunkt an, zu dem der Knoten u das erste Mal besucht wird, sondern die Ebene, auf der das passiert. Ist u ein direkter Nachfolger des Knoten s, an dem die Tiefensuche gestartet wird, so gilt $u.d = 1$. Ist u ein direkter Nachfolger eines direkten Nachfolgers von s, so gilt $u.d = 2$ und so weiter. Wir gehen davon aus, dass wir eine Queue Q haben, in die Knoten mittels Q.enqueue eingefügt und aus der Knoten mittels Q.dequeue entfernt werden können. Queues arbeiten nach dem FIFO-Prinzip. Das heißt, der erste Knoten, der zur Queue hinzugefügt wird, ist auch der erste Knoten, der wieder entfernt wird.

```
BFS(G = (V, E),s)
 1   for each (u ∈ V \ {s}) {
 2       u.color = weiß;  u.π = null;  u.d = ∞;
 3   }
 4   s.color = grau;  s.π = null;  s.d = 0;
 5   Q = ∅;
 6   Q.enqueue(s);
 7   while (Q ≠ ∅) {
 8       u = Q.dequeue();
 9       for each ((u, v) ∈ E) {
10           if (v.color == weiß) {
11               v.color = grau;  v.π = u;  v.d = u.d + 1;
12               Q.enqueue(v);
13           }
14       }
15       u.color = schwarz;
16   }
```

In der Queue befinden sich in Zeile 7 genau die grauen Knoten. Ein Knoten ist grau, wenn die Breitensuche bereits eine Kante zu ihm gesehen, ihn aber noch nicht abgearbeitet hat. Das FIFO-Prinzip der Queue stellt sicher, dass die Knoten in der oben beschriebenen Reihenfolge besucht werden. Abbildung 5.2 zeigt ein Beispiel für eine

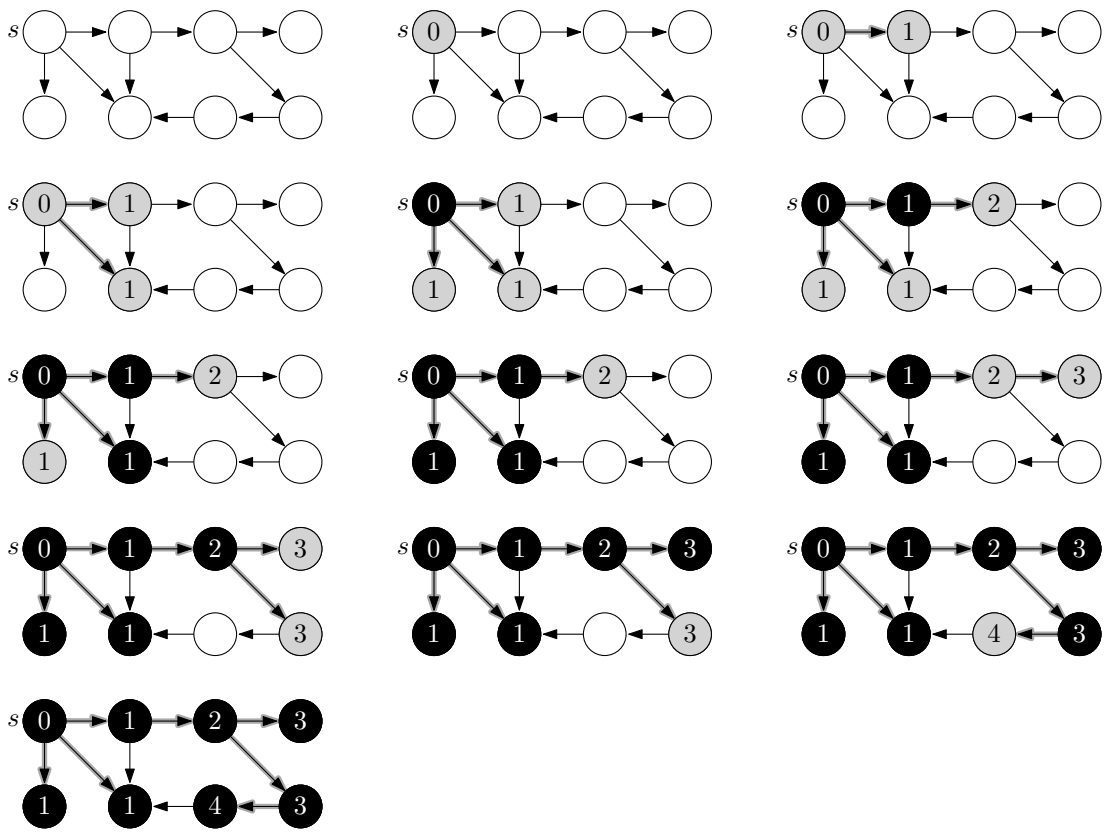


Abbildung 5.2: Beispiel für die Ausführung von BFS. Bei dem Wert innerhalb eines Knotens u handelt es sich um $u.d$.

Breitensuche. Die Laufzeit lässt sich ähnlich wie die der Tiefensuche abschätzen.

Theorem 5.7. *Die Laufzeit von Breitensuche auf einem Graphen $G = (V, E)$ beträgt $O(|V| + |E|)$, wenn der Graph als Adjazenzliste gegeben ist.*

Beweis. Jeder Knoten wird höchstens einmal zur Queue hinzugefügt, da er beim Hinzufügen grau gefärbt wird und somit in den weiteren Iterationen nicht mehr in Betracht kommt. Dementsprechend wird jede Kante in Zeile 9 für gerichtete Graphen maximal einmal und für ungerichtete Graphen maximal zweimal betrachtet. Aus diesen Beobachtungen ergibt sich leicht das Theorem. □

Eine wesentliche Eigenschaft von Breitensuche ist, dass sie jeden Knoten auf einem kürzesten Weg von s aus erreicht. Um dies zu formalisieren, definieren wir für einen gegebenen (gerichteten oder ungerichteten) Graphen $G = (V, E)$ für jedes Paar $u, v \in V$ von Knoten $\delta(u, v)$ als die Länge des kürzesten Weges von u nach v. Wir betrachten in diesem Abschnitt nur ungewichtete Graphen und fassen die Länge eines Weges stets als die Anzahl seiner Kanten auf. Gibt es keinen Weg von u nach v, so setzen wir $\delta(u, v) = \infty$. Unser Ziel ist es, das folgende Theorem zu beweisen.

Theorem 5.8. *Sei $G = (V, E)$ ein beliebiger Graph und sei $s \in V$ ein beliebiger Knoten. Nachdem die Breitensuche* $\mathrm{BFS}(G, s)$ *abgeschlossen ist, gilt $u.d = \delta(s, u)$ für jeden Knoten $u \in V$. Für jeden Knoten $u \in V$ mit $\delta(s, u) < \infty$ kann ein kürzester*

Weg von s zu u rückwärts von u aus konstruiert werden, indem man vom aktuellen Knoten v stets zu seinem Vorgänger $v.\pi$ geht.

Um das Theorem zu beweisen, benötigen wir zunächst einige Hilfsaussagen.

Lemma 5.9. *Es sei $G = (V, E)$ ein beliebiger Graph und es sei $s \in V$ ein beliebiger Knoten. Für jede Kante $(u, v) \in E$ gilt $\delta(s, v) \leq \delta(s, u) + 1$. Ist die Kante $(u, v) \in E$ in einem kürzesten Weg von s nach v enthalten, so gilt sogar $\delta(s, v) = \delta(s, u) + 1$.*

Beweis. Es genügt den Fall $\delta(s, u) < \infty$ zu betrachten. In diesem Fall ist u von s aus erreichbar. Wegen $(u, v) \in E$ ist dann auch v von s aus erreichbar. Fügen wir an den kürzesten Weg von s nach u noch die Kante (u, v) an, so erhalten wir einen Weg von s nach v der Länge $\delta(s, u) + 1$. Der kürzeste Weg von s nach v kann nicht länger sein als dieser Weg. Damit folgt der erste Teil des Lemmas.

Um den zweiten Teil des Lemmas zu beweisen, sei P ein kürzester Weg von s nach v, der die Kante (u, v) enthält. Sei P' der Teilweg von P ohne diese Kante. Dieser Weg muss ein kürzester Weg von s nach u sein. Gäbe es nämlich einen kürzeren solchen Weg P'', so ergäbe dieser kombiniert mit der Kante (u, v) auch einen kürzeren Weg von s nach v als P. Da der Weg P genau eine Kante mehr besitzt als der Weg P', folgt der zweite Teil des Lemmas. □

Wir zeigen zunächst, dass die Werte $u.d$, die die Breitensuche berechnet, durch die Werte $\delta(s, u)$ nach unten beschränkt sind.

Lemma 5.10. *Sei $G = (V, E)$ ein beliebiger Graph und sei $s \in V$ ein beliebiger Knoten. Nachdem die Breitensuche $\mathrm{BFS}(G, s)$ abgeschlossen ist, gilt $u.d \geq \delta(s, u)$ für jeden Knoten $u \in V$.*

Beweis. Wir beweisen die Aussage induktiv über die Anzahl an Knoten, die bislang in die Queue eingefügt wurden. Nach dem Einfügen von s in die Queue Q gilt $s.d = \delta(s, s) = 0$ und $u.d = \infty \geq \delta(s, u)$ für alle Knoten $u \in V \setminus \{s\}$. Angenommen die Aussage gilt nach einer bestimmten Zahl an Einfügungen und nun wird in Zeile 12 während der Bearbeitung eines Knotens u ein weiterer Knoten v der Queue hinzugefügt. Dies bedeutet, dass die Kante (u, v) im Graphen existiert. In Zeile 11 wird $v.d = u.d + 1$ gesetzt und aus Lemma 5.9 folgt $\delta(s, v) \leq \delta(s, u) + 1$. Aufgrund der Induktionsannahme gilt außerdem $u.d \geq \delta(s, u)$. Zusammen ergibt dies, wie gewünscht,

$$v.d = u.d + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$ □

Als nächstes weisen wir nach, dass sich die d-Werte der Knoten in der Queue zu jedem Zeitpunkt um maximal eins unterscheiden.

Lemma 5.11. *Enthält die Queue Q während der Ausführung von $\mathrm{BFS}(G, s)$ zu einem Zeitpunkt die Knoten $v_1, \ldots, v_r$ in dieser Reihenfolge (wobei v_1 von diesen Elementen das erste ist, das eingefügt wurde), so gilt $v_r.d \leq v_1.d + 1$ und $v_i.d \leq v_{i+1}.d$ für alle $i \in \{1, \ldots, r - 1\}$.*

Beweis. Wir zeigen das Lemma mithilfe vollständiger Induktion über die Anzahl an dequeue-Operationen. Zu Beginn enthält die Queue nur den Knoten s und die Aussagen des Lemmas sind trivialerweise erfüllt.

Gehen wir nun davon aus, dass die Aussagen zu einem Zeitpunkt gelten und dass in Zeile 8 eine dequeue-Operation ausgeführt wird. Mit dieser Operation wird lediglich der Knoten v_1 aus der Queue entfernt. Danach ist v_2 der neue erste Knoten in der Queue und die Aussagen des Lemmas sind noch immer erfüllt, denn es gilt $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$.

Nachdem in Zeile 8 der Knoten $u = v_1$ aus der Queue entfernt wurde, wird seine Adjazenzliste durchlaufen und es werden gegebenenfalls zu ihm adjazente Knoten in Zeile 12 an das Ende der Queue angefügt. Es seien $v_{r+1}, \ldots, v_{r+\ell}$ diese Knoten, für die $v_{r+1}.d = \ldots = v_{r+\ell}.d = v_1.d + 1$ gilt. Daraus folgt $v_{r+i}.d = v_1.d + 1 \leq v_2.d + 1$ für alle $i \in \{1, \ldots, \ell\}$. Ferner gilt aufgrund der Induktionsannahme $v_r.d \leq v_1.d + 1 = v_{r+i}.d$. Damit ist gezeigt, dass die Aussagen im Lemma weiterhin gelten. □

Beweis von Theorem 5.8. Wir führen einen Widerspruchsbeweis und gehen davon aus, dass es einen Knoten $u \in V$ gibt, für den $u.d \neq \delta(s, u)$ gilt. Lemma 5.10 lässt nur die Möglichkeit $u.d > \delta(s, u)$ zu. Wir betrachten einen kürzesten Weg P von s nach u und gehen ohne Beschränkung der Allgemeinheit davon aus, dass $w.d = \delta(s, w)$ für alle Knoten w auf diesem Weg vor u gilt. Ist dies nicht der Fall, so könnten wir u als den ersten Knoten auf dem Weg wählen, für den diese Gleichheit nicht gilt.

Sei nun w der Knoten, der auf dem Weg P direkt vor dem Knoten u besucht wird. Gemäß dem zweiten Teil von Lemma 5.9 gilt $\delta(s, u) = \delta(s, w) + 1$. Wir betrachten nun den Zeitpunkt, zu dem der Knoten w in Zeile 8 von BFS aus der Queue entfernt wird. Die Wahl von u stellt sicher, dass zu diesem Zeitpunkt $w.d = \delta(s, w)$ gilt. Gemäß Lemma 5.11 wurden vor w nur Knoten x mit $x.d \leq w.d = \delta(s, w)$ aus der Queue entfernt. Ist der Knoten u zu diesem Zeitpunkt schon grau, so wäre er von einem solchen Knoten x erreicht worden und es würde

$$u.d \leq x.d + 1 \leq w.d + 1 = \delta(s, w) + 1 = \delta(s, u)$$

gelten. Mit Lemma 5.10 folgt daraus $u.d = \delta(s, u)$, was im Widerspruch zur Wahl von u steht. Das bedeutet, wenn der Knoten w in Zeile 8 aus der Queue entfernt wird, ist der Knoten u noch unbesucht. Dann wird er aber beim anschließenden Durchlauf der for-Schleife in die Queue eingefügt und es wird

$$u.d = w.d + 1 = \delta(s, w) + 1 = \delta(s, u)$$

gesetzt, was ebenfalls im Widerspruch zur Wahl von u steht.

Dass wir für jeden Knoten $u \in V$, der von s aus erreichbar ist, einen kürzesten Weg erhalten, indem wir den Kanten der Form $(v.\pi, v)$ folgen, folgt direkt daraus, dass für jede solche Kante $v.d = v.\pi.d + 1$ gilt. Dies besagt nämlich, dass der Weg, den wir auf diese Weise zu einem Knoten u erhalten, genau die Länge $u.d = \delta(s, u)$ besitzt. □

5.2 Minimale Spannbäume

In diesem Abschnitt bezeichne $G = (V, E)$ stets einen ungerichteten zusammenhängenden Graphen mit positiven Kantengewichten $w : E \to \mathbb{R}_{>0}$. Wir erweitern die Funktion w auf Teilmengen von E und definieren das Gewicht einer Teilmenge $E' \subseteq E$ als

$$w(E') = \sum_{e \in E'} w(e).$$

Definition 5.12. *Eine Kantenmenge $T \subseteq E$ heißt* Spannbaum *von G, wenn der Graph (V, T) ein Baum (das heißt zusammenhängend und kreisfrei) ist. Ein Spannbaum $T \subseteq E$ heißt* minimaler Spannbaum *von G, wenn es keinen Spannbaum von G mit einem kleineren Gewicht als $w(T)$ gibt.*

Die Berechnung eines minimalen Spannbaumes ist ein Problem, das in vielen Anwendungen wie beispielsweise der Planung von Telekommunikations- und Stromnetzen auftritt. Wir entwerfen in diesem Abschnitt einen effizienten Algorithmus zur Berechnung eines minimalen Spannbaumes. Vorher beschreiben wir noch eine Datenstruktur, die für diesen Algorithmus benötigt wird.

5.2.1 Union-Find-Datenstrukturen

Für den Algorithmus zur Berechnung eines minimalen Spannbaumes und auch für viele andere Algorithmen benötigen wir eine sogenannte *Union-Find-* oder *Disjoint-Set-Datenstruktur*. Diese Datenstruktur dient zur Verwaltung einer Menge von disjunkten Mengen $S_1, \ldots, S_k$. Für jede von diesen disjunkten Mengen S_i gibt es einen beliebigen Repräsentanten $s_i \in S_i$, und die Datenstruktur muss die folgenden Operationen unterstützen.

- Make-Set(x): Erzeugt eine neue Menge $\{x\}$ mit Repräsentant x. Dabei darf x nicht bereits in einer anderen Menge enthalten sein.
- Union(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.
- Find(x): Liefert den Repräsentanten der Menge S mit $x \in S$ zurück.

Beispiel für die Operationen einer Union-Find-Datenstruktur

Im folgenden Beispiel soll $x\colon A$ bedeuten, dass wir eine Menge A gespeichert haben, deren Repräsentant x ist. Welches Element nach einer Union-Operation Repräsentant der neuen Menge wird, ist nicht eindeutig bestimmt. Es wurde jeweils ein beliebiges Element dafür ausgewählt.

Operation	Zustand der Datenstruktur
	$\emptyset$
MAKE-SET(1)	$1:\{1\}$
MAKE-SET(2), MAKE-SET(3)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$
MAKE-SET(4), MAKE-SET(5)	$1:\{1\}$, $2:\{2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
FIND(3)	Ausgabe: 3, keine Zustandsänderung
UNION(1,2)	$2:\{1,2\}$, $3:\{3\}$, $4:\{4\}$, $5:\{5\}$
UNION(3,4)	$2:\{1,2\}$, $3:\{3,4\}$, $5:\{5\}$
FIND(1)	Ausgabe: 2, keine Zustandsänderung
UNION(1,5)	$2:\{1,2,5\}$, $3:\{3,4\}$
FIND(5)	Ausgabe: 2, keine Zustandsänderung
UNION(5,4)	$3:\{1,2,3,4,5\}$

Wir betrachten nun eine Realisierung einer Union-Find-Datenstruktur, die immer dann eingesetzt werden kann, wenn am Anfang feststeht, wie oft MAKE-SET aufgerufen wird, das heißt, wie viele Elemente sich maximal in den Mengen befinden. Bezeichne n diese Anzahl, die die Datenstruktur bei der Initialisierung als Parameter übergeben bekommt. Wir legen ein Feld $A[1 \dots n]$ an und speichern für jedes der n Elemente den Repräsentanten der Menge, in der es sich gerade befindet. Am Anfang initialisieren wir das Feld mit $A[1] = 1, \dots, A[n] = n$, das heißt, wir gehen davon aus, dass MAKE-SET bereits für jedes Element einmal aufgerufen wurde und dass sich jedes Element in einer eigenen Menge befindet.

Mit dieser Information allein könnten wir bereits eine Union-Find-Datenstruktur implementieren. Allerdings würde jede Vereinigung eine Laufzeit von $\Theta(n)$ benötigen, da wir bei einer Vereinigung zweier Mengen jedes Element darauf testen müssten, ob es zu einer dieser Mengen gehört und gegebenenfalls den entsprechenden Eintrag im Feld A anpassen müssten. Deshalb speichern wird noch weitere Informationen.

Wir legen ein Feld $L[1 \dots n]$ an und speichern in jedem der Einträge einen Zeiger auf eine verkettete Liste. Wenn ein Element i Repräsentant seiner Menge ist, so enthält die verkettete Liste $L[i]$ alle Elemente in der Menge, die i repräsentiert. Zusätzlich speichern wir in einem weiteren Feld size$[1 \dots n]$ die Kardinalität der Listen von L ab. Ist ein Element i nicht Repräsentant seiner Menge, so ist $L[i]$ der Null-Zeiger und size$[i] = 0$. Zu Beginn repräsentieren alle Elemente ihre eigenen einelementigen Teilmengen, das heißt, jede Liste $L[i]$ enthält nur i selbst als Element und außerdem gilt size$[i] = 1$.

Die n Aufrufe von MAKE-SET haben wir mit dieser Initialisierung bereits erledigt. Diese benötigen insgesamt eine Laufzeit von $\Theta(n)$, da wir nur die Felder A, L und size korrekt initialisieren müssen. Wir überlegen uns nun, wie man UNION und FIND implementiert. FIND(x) ist sehr einfach; wir geben lediglich $A[x]$ zurück. Dafür wird nur eine konstante Laufzeit in der Größenordnung $O(1)$ benötigt.

Die UNION-Operation wird wie folgt realisiert.

```
UNION(x, y)
1  i = FIND(x); j = FIND(y);
2  if (size[i] > size[j]) Vertausche i und j.
3  for each (z ∈ L[i]) { A[z] = j; }
4  Hänge L[i] an L[j] an.
5  size[j] = size[j] + size[i];
6  L[i] = null;
7  size[i] = 0;
```

Der Repräsentant der vereinigten Menge ist also der alte Repräsentant der größeren Menge und die Einträge im Feld A werden für alle Elemente der kleineren Menge entsprechend angepasst. Die Laufzeit beträgt somit $\Theta(\min\{\text{size}[i], \text{size}[j]\})$, wobei i und j die Repräsentanten der beiden zu vereinigenden Mengen bezeichnen. Genauso wie bei dynamischen Feldern interessieren uns auch bei dieser Union-Find-Datenstruktur die amortisierten Kosten.

Lemma 5.13. *Jede Folge von $(n-1)$ UNION- und f FIND-Operationen kann in Zeit $O(n\log(n)+f)$ durchgeführt werden, wobei n die Anzahl an Elementen bezeichnet.*

Beweis. Da FIND-Operationen in Zeit $O(1)$ ausgeführt werden können, ist nur zu zeigen, dass alle UNION-Operationen zusammen eine Laufzeit von $O(n\log n)$ besitzen. Wir wählen zunächst eine geeignete Konstante $c > 0$, sodass jede UNION-Operation höchstens c mal so viele Rechenschritte benötigt wie die Kardinalität der kleineren Menge. Für $i \in \{1,\dots,n-1\}$ bezeichne X_i die kleinere Menge, die bei der i-ten UNION-Operation auftritt. Dann können wir die Laufzeit für alle UNION-Operationen durch

$$c \cdot \sum_{i=1}^{n-1} |X_i| = c \cdot \sum_{i=1}^{n-1} \sum_{x \in X_i} 1 = c \cdot \sum_{j=1}^{n} |\{X_i \mid j \in X_i\}| \tag{5.1}$$

nach oben abschätzen. Wir schauen also für jedes Element j, bei wie vielen UNION-Operationen es in der kleineren Menge auftritt, und summieren diese Anzahlen auf.

Tritt ein Element j zum ersten Mal in der kleineren Menge auf, so hat diese Kardinalität 1. Danach liegt j in einer Menge der Größe mindestens 2. Ist diese Menge dann später wieder die kleinere Menge einer UNION-Operation, so liegt j hinterher in einer Menge der Größe mindestens 4 und so weiter. Wir können festhalten, dass das Element j, nachdem es s mal in der kleineren Menge einer UNION-Operation lag, in einer Menge der Größe mindestens 2^s liegt. Da es nur n Elemente gibt, muss $2^s \leq n$ und somit $s \leq \log_2 n$ gelten. Also gilt für jedes Element j

$$|\{X_i \mid j \in X_i\}| \leq \log_2 n.$$

Zusammen mit (5.1) ergibt sich

$$c \cdot \sum_{j=1}^{n} \log_2 n = O(n\log n)$$

als obere Schranke für die Laufzeit aller UNION-Operationen. □

5.2.2 Algorithmus von Kruskal

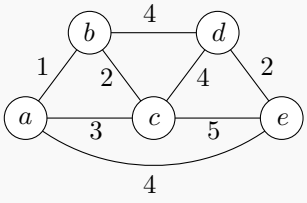
Mithilfe einer Union-Find-Datenstruktur können wir nun den *Algorithmus von Kruskal* zur Berechnung eines minimalen Spannbaumes angeben. Er ist nach Joseph Kruskal benannt, der diesen Algorithmus 1956 veröffentlichte.

```
KRUSKAL(G, w)
 1  Teste mittels DFS, ob G zusammenhängend ist. Falls nicht, Abbruch.
 2  for each (v ∈ V) MAKE-SET(v);
 3  T = ∅;
 4  Sortiere die Kanten in E gemäß ihrem Gewicht.
    Danach gelte E = {e_1, ..., e_m} mit w(e_1) ≤ w(e_2) ≤ ... ≤ w(e_m).
    Außerdem sei e_i = (u_i, v_i).
 5  for (int i = 1; i ≤ m; i++) {
 6      if (FIND(u_i) ≠ FIND(v_i)) {
 7          T = T ∪ {e_i};
 8          UNION(u_i, v_i);
 9      }
10  }
11  return T
```

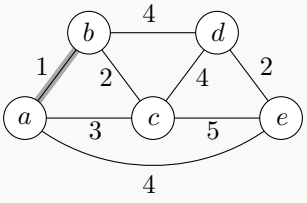
Die Mengen, die in der Union-Find-Datenstruktur im Algorithmus verwaltet werden, entsprechen den Zusammenhangskomponenten des Graphen $G = (V, T)$. Wir gehen die Kanten in der Reihenfolge ihres Gewichtes durch und fügen eine Kante zur Menge T hinzu, falls sie nicht innerhalb einer bereits zusammenhängenden Menge verläuft.

Bevor wir die Korrektheit beweisen und die Laufzeit analysieren, demonstrieren wir das Verhalten an einem Beispiel.

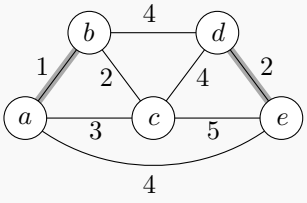
Beispiel für den Algorithmus von Kruskal



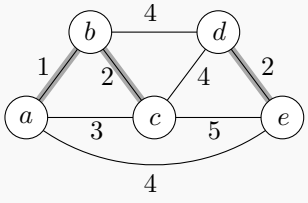
$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}$



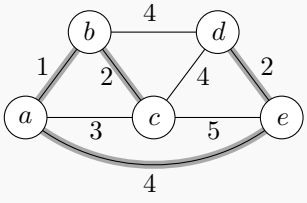
$\{a, b\}, \{c\}, \{d\}, \{e\}$



$\{a, b\}, \{c\}, \{d, e\}$



$\{a, b, c\}, \{d, e\}$



$\{a, b, c, d, e\}$

Die Korrektheit des Algorithmus von Kruskal folgt aus dem folgenden Theorem

Theorem 5.14. *Für zusammenhängende Graphen berechnet der Algorithmus von Kruskal einen minimalen Spannbaum.*

Beweis. Wir benutzen die folgende Invariante, die aus zwei Teilen besteht. Wir behaupten, dass die Invariante am Anfang jedes Schleifendurchlaufs, also nach Überprüfung der Abbruchbedingung der Schleife in Zeile 5, gilt (auch dann wenn die Abbruchbedingung greift und die Schleife nicht durchlaufen wird).

1. Die Mengen, die in der Union-Find-Datenstruktur gespeichert werden, entsprechen den Zusammenhangskomponenten des Graphen (V, T).
2. Es gibt eine Kantenmenge $S \subseteq \{e_i, \ldots, e_m\}$, sodass $T \cup S$ ein minimaler Spannbaum von G ist. Insbesondere ist (V, T) azyklisch.

Ist diese Invariante korrekt, so ist auch der Algorithmus korrekt. Das folgt unmittelbar aus dem zweiten Teil der Invariante. Nach dem letzten Schleifendurchlauf gilt $i = m+1$. Somit muss S leer sein und damit T ein minimaler Spannbaum.

Nun müssen wir nur noch die Korrektheit der Invariante zeigen. Dazu überlegen wir uns zunächst, dass die Invariante vor dem ersten Schleifendurchlauf gilt. Der erste Teil der Invariante gilt, weil die Menge T bislang noch leer ist. Die Zusammenhangskomponenten von (V, T) sind also genau die einelementigen Mengen. Dies ist auch in der Union-Find-Struktur so abgebildet. Für den zweiten Teil der Invariante nutzen wir $T = \emptyset$ und $i = 1$. Wir können dann S als einen beliebigen minimalen Spannbaum von G wählen.

Schauen wir uns nun einen Schleifendurchlauf an und nehmen an, dass die Invariante zu Beginn erfüllt ist. Seien nun T und i die Werte der Variablen zu Beginn des Durchlaufs.

Hinsichtlich des ersten Teils der Invariante stellen wir fest: Falls wir Kante e_i nicht in T einfügen, so ändern sich weder die Mengen in der Union-Find-Datenstruktur noch die Zusammenhangskomponenten von (V, T). Teil eins der Invariante bleibt dann also erhalten. Fügen wir e_i in T ein, so fallen die Zusammenhangskomponenten von u_i und v_i in (V, T) zu einer gemeinsamen Komponente zusammen. Diese Veränderung bilden wir in der Union-Find-Datenstruktur korrekt ab. Also bleibt auch dann der erste Teil erhalten.

Für den zweiten Teil nutzen wir die Induktionsvoraussetzung, dass es eine Menge $S \subseteq \{e_i, \ldots, e_m\}$ gibt, so dass $T \cup S$ ein minimaler Spannbaum ist. Nun können wir einige Fälle unterscheiden.

Falls $T \cup \{e_i\}$ einen Kreis enthält, wird e_i nicht zu T hinzugefügt. Gleichzeitig kann nicht $e_i \in S$ gelten, denn dann wäre $T \cup S$ kein Baum. Also gilt $S \subseteq \{e_{i+1}, \ldots, e_m\}$. Damit gilt die Invariante auch beim nächsten Erreichen von Zeile 5.

Falls $T \cup \{e_i\}$ keinen Kreis enthält, fügen wir e_i zu T hinzu. Wenn auch $e_i \in S$ gilt, gibt es nichts weiter zu zeigen, denn e_i wird somit nur von S nach T verschoben. Der interessante Fall ist somit, dass $e_i \notin S$ gilt.

Wir wissen, dass $(V, T \cup S)$ zusammenhängend ist. Wegen $e_i \notin S$ muss $T \cup S \cup \{e_i\}$ einen Kreis C enthalten. Gleichzeitig wissen wir aber, dass $T \cup \{e_i\}$ keinen Kreis enthält. Somit muss es eine Kante $e_j \in S$ geben, die in C liegt. Wir behaupten nun, dass $T' := T \cup (S \setminus \{e_j\}) \cup \{e_i\}$ auch ein minimaler Spannbaum ist.

Zunächst können wir festhalten, dass (V, T') zusammenhängend ist. Der Graph $(V, T \cup S)$ ist nach Voraussetzung zusammenhängend. Gibt es in $(V, T \cup S)$ zwischen zwei Knoten $u, v \in V$ einen Weg, der die Kante e_j nicht benutzt, so gibt es diesen Weg auch in (V, T'). Gibt es in $(V, T \cup S)$ einen Weg P zwischen u und v, der die Kante $e_j = (w_1, w_2)$ benutzt, so erhält man auch in (V, T') einen Weg zwischen u und v, indem man P nimmt und die Kante e_j durch den Weg zwischen w_1 und w_2 über die Kante e_i ersetzt.

Daraus folgt nun direkt, dass (V, T') ein Spannbaum ist. Durch das Entfernen der Kante e_j zerfällt der Spannbaum $(V, T \cup S)$ in zwei Zusammenhangskomponenten $V_1 \subset V$ und $V_2 \subset V$, sodass $w_1 \in V_1$ und $w_2 \in V_2$. Das Hinzufügen von e_j verbindet diese beiden Komponenten wieder zu einem neuen Spannbaum.

Wir müssen noch zeigen, dass der neue Spannbaum auch minimal ist. Dafür nutzen wir die Tatsache, dass $i < j$ und dass wir die Kanten nach Gewicht sortiert haben. Somit gilt nämlich $w(e_i) \leq w(e_j)$ und deshalb

$$w(T') = w(T \cup S) + w(e_i) - w(e_j) \leq w(T \cup S). \qquad \square$$

Nachdem wir die Korrektheit des Algorithmus von Kruskal bewiesen haben, analysieren wir nun noch die Laufzeit.

Theorem 5.15. *Für einen ungerichteten zusammenhängenden Graphen $G = (V, E)$ benötigt der Algorithmus von Kruskal eine Laufzeit von $O(|E| \log |E|)$.*

Beweis. Die Tiefensuche im ersten Schritt benötigt eine Laufzeit von $O(|V| + |E|)$. Da der Graph laut Voraussetzung zusammenhängend ist, gilt $|E| \geq |V| - 1$, also $O(|V| + |E|) = O(|E|)$. Das Sortieren der Kanten benötigt Zeit $O(|E| \log |E|)$. Die for-Schleife wird $|E|$ mal durchlaufen und abgesehen von Union und Find werden in jedem Durchlauf nur konstant viele Operationen durchgeführt, sodass wir eine Laufzeit von $O(|E|)$ für alle Operationen in der Schleife abgesehen von Union und Find ansetzen können.

Wir führen in der Union-Find-Datenstruktur $2|E|$ Find-Operationen und $|V| - 1$ Union-Operationen durch. Mit Lemma 5.13 ergibt sich demnach eine Laufzeit von $O(|V| \log |V| + 2|E|) = O(|E| \log |E|)$ für die Operationen der Union-Find-Datenstruktur. $\square$

5.3 Kürzeste Wege

In diesem Abschnitt sei $G = (V, E)$ stets ein gerichteter Graph mit Kantengewichten $w : E \to \mathbb{R}$. Da wir uns in diesem Abschnitt mit kürzesten Wegen beschäftigen,

werden wir $w(e)$ auch als die *Länge der Kante* e bezeichnen. Außerdem werden wir für eine Kante $(x, y) \in E$ auch $w(x, y)$ statt der formal korrekten Notation $w((x, y))$ verwenden.

Definition 5.16. *Es seien $s \in V$ und $t \in V$ zwei beliebige Knoten und es sei $P = (v_0, v_1, \ldots, v_\ell)$ ein Weg von s nach t. Wir definieren die* Länge von P *als $w(P) = \sum_{i=0}^{\ell-1} w(v_i, v_{i+1})$. Wir sagen, dass P ein* kürzester Weg von s nach t *ist, falls es keinen Weg P' von s nach t mit $w(P') < w(P)$ gibt. Wir nennen die Länge $w(P)$ des kürzesten Weges P die* Entfernung von s nach t *und bezeichnen diese mit $\delta(s, t)$. Existiert kein Weg von s nach t, so gelte $\delta(s, t) = \infty$.*

Die naheliegende Anwendung von Algorithmen zur Berechnung kürzester Wege sind Navigationsgeräte. Es gibt jedoch auch zahlreiche andere Anwendungsgebiete, in denen kürzeste Wege gefunden werden müssen. Insbesondere auch solche, in denen negative Kantengewichte auftreten können.

Wir sind an der Lösung der folgenden zwei Probleme interessiert.

1. Im *Single-Source Shortest Path Problem (SSSP)* ist zusätzlich zu G und w ein Knoten $s \in V$ gegeben und wir möchten für jeden Knoten $v \in V$ einen kürzesten Weg von s nach v und die Entfernung $\delta(s, v)$ berechnen.

2. Im *All-Pairs Shortest Path Problem (APSP)* sind nur G und w gegeben und wir möchten für jedes Paar $u, v \in V$ von Knoten einen kürzesten Weg von u nach v und die Entfernung $\delta(u, v)$ berechnen.

Für den Spezialfall, dass $w(e) = 1$ für alle Kanten $e \in E$ gilt, haben wir mit Breitensuche bereits einen Algorithmus kennengelernt, der das SSSP löst. Breitensuche lässt sich allerdings nicht direkt auf den Fall verallgemeinern, dass beliebige Kantengewichte gegeben sind.

Bevor wir Algorithmen für das SSSP und das APSP beschreiben, betrachten wir zunächst einige strukturelle Eigenschaften von kürzesten Wegen.

5.3.1 Grundlegende Eigenschaften von kürzesten Wegen

Für die Korrektheit der Algorithmen, die wir vorstellen werden, ist es wichtig, dass jeder Teilweg eines kürzesten Weges selbst wieder ein kürzester Weg ist.

Lemma 5.17. *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Für jedes Paar i, j mit $0 \leq i \leq j \leq \ell$ ist $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ ein kürzester Weg von v_i nach v_j.*

Beweis. Wir zerlegen den Weg $P = P_{0\ell}$ in die drei Teilwege P_{0i}, P_{ij} und $P_{j\ell}$. Dies schreiben wir als

$$P = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

Dann gilt insbesondere $w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{j\ell})$.

Nehmen wir nun an, dass P_{ij} kein kürzester Weg von v_i nach v_j ist. Dann gibt es einen Weg P'_{ij} von i nach j mit $w(P'_{ij}) < w(P_{ij})$. Mit diesem erhalten wir ebenfalls einen anderen Weg P' von v_0 nach v_ℓ:

$$P' = v_0 \overset{P_{0i}}{\rightsquigarrow} v_i \overset{P'_{ij}}{\rightsquigarrow} v_j \overset{P_{j\ell}}{\rightsquigarrow} v_\ell.$$

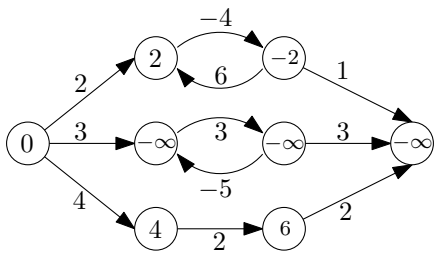
Für diesen Weg gilt $w(P') = w(P) - w(P_{ij}) + w(P'_{ij}) < w(P)$ im Widerspruch zur Voraussetzung, dass P ein kürzester Weg von v_0 nach v_ℓ ist. Also kann es keinen kürzeren Weg von v_i nach v_j als P_{ij} geben. □

Aus dem vorangegangenen Lemma erhalten wir direkt das folgende Korollar.

Korollar 5.18. *Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$. Dann gilt*

$$\delta(v_0, v_\ell) = \delta(v_0, v_{\ell-1}) + w(v_{\ell-1}, v_\ell).$$

Wir haben auch negative Kantengewichte erlaubt, da diese in einigen Anwendungsgebieten tatsächlich auftreten. Dies führt jedoch zu zusätzlichen Schwierigkeiten, insbesondere dann, wenn der Graph einen Kreis mit negativem Gesamtgewicht enthält. Dann ist nämlich für einige Paare von Knoten kein kürzester Weg mehr definiert, da man beliebig oft den Kreis mit negativem Gesamtgewicht entlang laufen könnte. Die folgende Abbildung zeigt einen Graphen mit einem negativen Kreis. Die Zahlen in den Knoten entsprechen dabei den Entfernungen der Knoten vom linken Knoten.



Im Kontrast dazu halten wir fest, dass ein kürzester Weg in einem Graphen ohne Kreis mit negativem Gesamtgewicht keinen Knoten mehrfach besuchen muss und damit auch aus höchstens $n - 1$ Kanten besteht. Ist $P = (v_0, \ldots, v_\ell)$ nämlich ein kürzester Weg von $v_0 \in V$ nach $v_\ell \in V$ und gäbe es v_i und v_j mit $i < j$ und $v_i = v_j$, so wäre der Teilweg $P_{ij} = (v_i, \ldots, v_j)$ ein Kreis, der laut Voraussetzung keine negativen Gesamtkosten besitzt. Das heißt, er hat entweder Kosten 0 oder echt positive Kosten. In jedem Falle können wir einen neuen Weg P' von v_0 nach v_ℓ konstruieren, indem wir den Kreis überspringen: $P' = (v_0, \ldots, v_i, v_{j+1}, \ldots, v_\ell)$. Für diesen neuen Weg P' gilt $w(P') \leq w(P)$.

Der *Algorithmus von Dijkstra*, den wir für das SSSP kennenlernen werden, funktioniert nur für Graphen, die keine negativen Kantengewichte besitzen. Der *Floyd-Warshall-Algorithmus* für das APSP funktioniert auch für negative Kantengewichte, solange es keine negativen Kreise gibt. Gibt es einen negativen Kreis, so kann der Floyd-Warshall-Algorithmus zumindest benutzt werden, um die Existenz eines solchen zu detektieren.

5.3.2 Single-Source Shortest Path Problem

In diesem Abschnitt betrachten wir nur Eingaben, in denen alle Kantengewichte nichtnegativ sind. Im Folgenden bezeichne $G = (V, E)$ stets einen gerichteten Graphen mit Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$ und $s \in V$ bezeichne einen beliebigen Startknoten, von dem aus kürzeste Wege zu allen anderen Knoten berechnet werden sollen. Der Einfachheit halber gehen wir davon aus, dass es von s zu jedem Knoten aus V einen Weg gibt. Außerdem gelte $n = |V|$ und $m = |E|$.

Algorithmus von Dijkstra

Für das SSSP werden wir den Algorithmus von Dijkstra kennenlernen. Dieser verwaltet für jeden Knoten $v \in V$ zwei Attribute $d(v) \in \mathbb{R}$ und $\pi(v) \in V \cup \{\textbf{null}\}$. Am Ende wird $d(v) \in \mathbb{R}$ die Entfernung von s zu v angeben. Ein kürzester Weg lässt sich dann aus den Werten rekonstruieren, die in π gespeichert sind: Sofern $d(v) < \infty$, ist $\pi(v)$ der letzte Zwischenknoten auf dem Weg zu v. Der gesamte Weg ergibt sich aus dem kürzesten Weg von s zu $\pi(v)$ sowie der Kante $(\pi(v), v)$. Wir initialisieren $\pi(v) = \textbf{null}$ für alle $v \in V$ und $d(v) = \infty$ für alle $v \in V \setminus \{s\}$ sowie $d(s) = 0$.

Der Ablauf des Algorithmus ist nun wie folgt: In einer Menge $S \subseteq V$ sind diejenigen Knoten gespeichert, für die bereits ein kürzester Weg gefunden wurde. Das heißt, dass $d(v) = \delta(s, v)$ für $v \in S$. Initial ist S leer und vergrößert sich im Laufe der Zeit. Für $v \notin S$ ist $d(v)$ zu verstehen als die Länge des kürzesten Weges von s zu v, den der Algorithmus bisher gefunden hat. Somit gilt lediglich $d(v) \geq \delta(s, v)$. Konkreter wird $d(v)$ die Länge des kürzesten Weges von s nach v sein, der lediglich Knoten aus S als Zwischenknoten verwendet.

Wir gehen davon aus, dass der Graph G in Adjazenzlistendarstellung gegeben ist. Wie Zeile 4 effizient realisiert werden kann, diskutieren wir später.

```
DIJKSTRA(G, w, s)
 1   for each (v ∈ V) { d(v) = ∞; π(v) = null; }
 2   d(s) = 0; S = ∅;
 3   while (S ≠ V) {
 4       Finde u ∈ V \ S, für das d(u) minimal ist.
 5       S = S ∪ {u};
 6       for each ((u, v) ∈ E) {
 7           if (d(v) > d(u) + w(u, v)) {
 8               d(v) = d(u) + w(u, v);
 9               π(v) = u;
10           }
11       }
12   }
```

Ein Beispiel für die Ausführung des Algorithmus von Dijkstra findet sich in Abbildung 5.3.

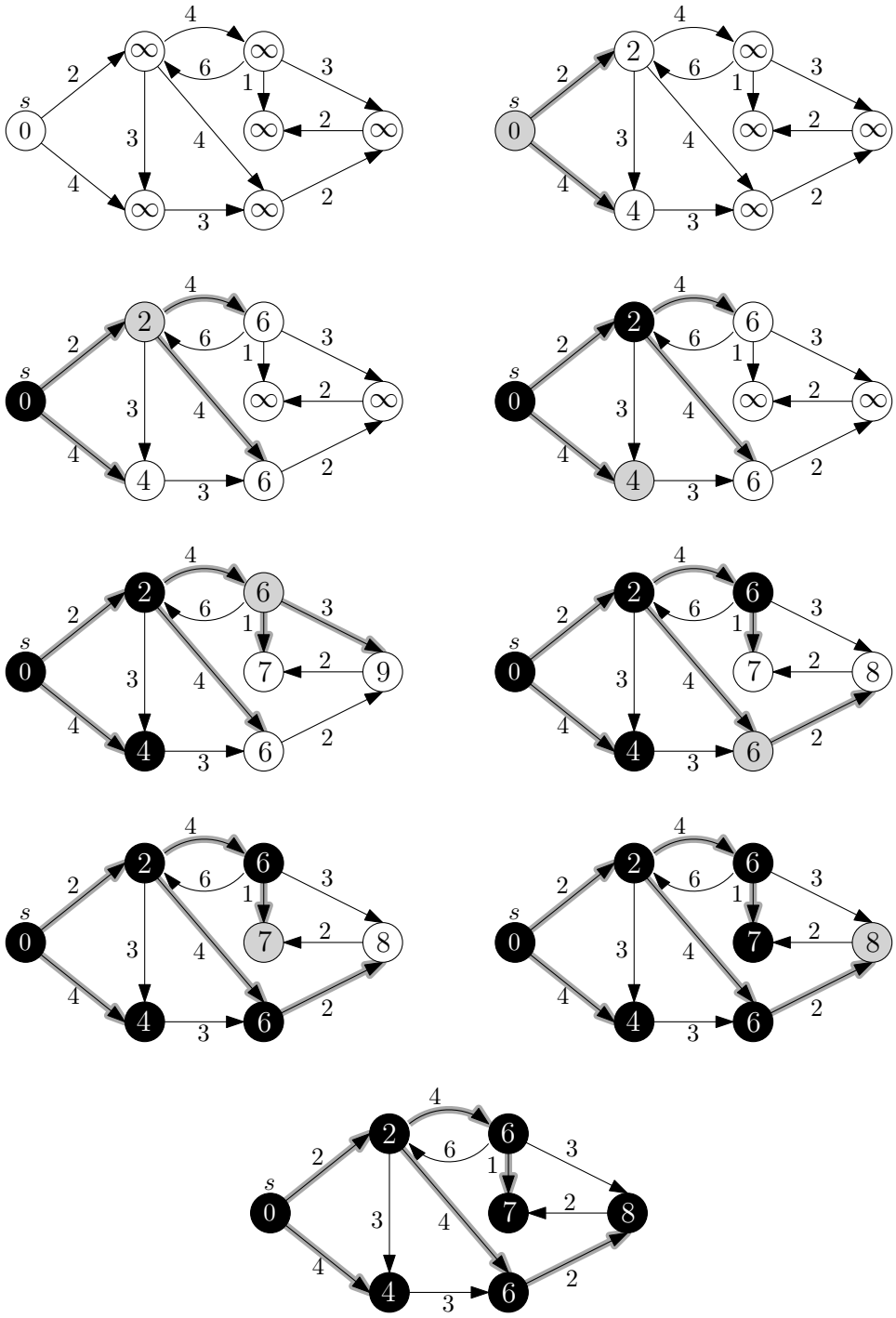


Abbildung 5.3: Beispiel für den Ablauf des Dijkstra-Algorithmus. Schwarze Knoten gehören zu S und der graue Knoten ist jeweils der Knoten u, der aus Q extrahiert wird. Eine graue Kante (u, v) gibt an, dass $\pi(v) = u$ gilt.

Theorem 5.19. *Der Algorithmus von Dijkstra terminiert auf gerichteten Graphen $G = (V, E)$ mit nichtnegativen Kantengewichten $w : E \to \mathbb{R}_{\geq 0}$ und Startknoten $s \in V$ in einem Zustand, in dem $d(v) = \delta(s, v)$ für alle $v \in V$ gilt.*

Beweis. Wir beweisen das Theorem mithilfe der folgenden Invariante, bei der wir die Konvention $w((u, v)) = \infty$ für $(u, v) \notin E$ verwenden.

> Am Ende jeder Iteration der while-Schleife gilt:
>
> 1. $\forall v \in S : d(v) = \delta(s, v)$,
> 2. $\forall v \in V \setminus S : d(v) = \min\{\delta(s, x) + w(x, v) \mid x \in S\}$.

Gilt diese Invariante, so folgt aus ihr direkt das Theorem: In jedem Schritt der while-Schleife fügen wir einen weiteren Knoten zu der Menge S hinzu. Nach n Durchläufen der while-Schleife muss $S = V$ gelten und der Algorithmus terminiert. Die Invariante besagt dann, dass $d(v) = \delta(s, v)$ für alle Knoten $v \in S = V$ gilt.

Am Ende des ersten Schleifendurchlaufs gilt $S = \{s\}$ und $d(s) = \delta(s, s) = 0$. Somit ist der erste Teil der Invariante erfüllt. Den zweiten Teil der Invariante können wir in diesem Fall umschreiben zu: $\forall v \in V \setminus \{s\} : d(v) = \delta(s, s) + w(s, v) = w(s, v)$. Die Gültigkeit dieser Aussage folgt daraus, dass für jeden direkten Nachfolger v von s die Zeilen 7 bis 10 ausgeführt werden. Diese setzen $d(v) = \min\{d(s) + w(s, v), \infty\} = w(s, v)$.

Wir müssen nun nur noch zeigen, dass die Invariante erhalten bleibt. Gehen wir also davon aus, dass wir uns in einem Durchlauf der while-Schleife befinden und am Ende des vorherigen Durchlaufs die Invariante galt. Während dieser Iteration fügen wir zunächst den Knoten $u \in V \setminus S$, der momentan unter allen Knoten aus $V \setminus S$ den kleinsten Wert $d(u)$ besitzt, zur Menge S hinzu. Dann betrachten wir alle Nachbarn v von u und aktualisieren den Wert $d(v)$, falls der aktuelle Weg über die Kante (u, v) kürzer ist.

Wir beweisen zunächst den ersten Teil der Invariante. Für alle Knoten $v \in S$, mit $v \neq u$, folgt aus der Induktionsvoraussetzung, dass am Anfang des Schleifendurchlaufs $d(v) = \delta(s, v)$ gilt. Es ist möglich, dass im aktuellen Schleifendurchlauf v als Nachbar von u betrachtet wird. Da v in S liegt und also in einem früheren Durchlauf der while-Schleife in Zeile 4 ausgewählt wurde, galt zu dem Zeitpunkt schon $d(v) \leq d(u)$, und somit bliebt der Wert $d(v)$ unverändert. (Der Wert von $d(u)$ kann sich nach der Bearbeitung von v noch verringern, er kann aber nicht kleiner werden als der d-Wert eines Knoten der schon abgearbeitet ist, wie man leicht per Induktion zeigen kann.)

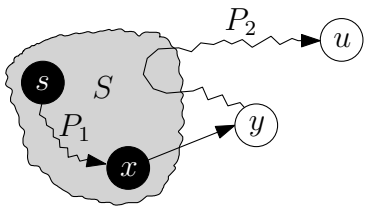
Nun müssen wir noch zeigen, dass auch für den Knoten u, der in diesem Schleifendurchlauf zu S hinzugefügt wird, $d(u) = \delta(s, u)$ gilt. Hierfür zeigen wir, dass sowohl $d(u) \geq \delta(s, u)$ als auch $d(u) \leq \delta(s, u)$ gilt. Wir betrachten dafür den Zustand am Anfang des Schleifendurchlaufs.

Um die Ungleichung $d(u) \geq \delta(s, u)$ zu zeigen, nutzen wir zunächst, dass gemäß dem zweiten Teil der Invariante $d(u) = \min\{\delta(s, x)+w(x, u) \mid x \in S\}$ gilt. Für jeden Knoten

$x \in S$ können wir einen (nicht unbedingt kürzesten) Weg von s zu u konstruieren, indem wir einen kürzesten Weg von s zu x um die Kante (x, u) erweitern. Deshalb gilt $\delta(s, u) \leq \delta(s, x) + w(x, u)$ für alle $x \in S$ und damit insbesondere auch für dasjenige x, das den Ausdruck minimiert.

Der komplexere Teil ist, zu zeigen, dass auch $d(u) \leq \delta(s, u)$ gilt. Intuitiv bedeutet dies, dass wir in den bisherigen Schleifendurchläufen, in denen Wege über Knoten in S betrachtet wurden, tatsächlich einen kürzesten Weg zu u gefunden haben und dass es nicht noch einen besseren Weg über Knoten gibt, die zur Zeit nicht in S sind.

Sei dazu P ein kürzester Weg von s nach u und sei y der erste Knoten auf dem Weg P, der nicht zur Menge S gehört. Da u selbst noch nicht zu S gehört, muss es einen solchen Knoten y geben. Weiterhin muss y einen Vorgänger haben, da der erste Knoten auf dem Weg P der Knoten $s \in S$ ist. Wir nennen diesen Vorgänger x. Den Teilweg von P von s zu x nennen wir P_1 und den Teilweg von y zu u nennen wir P_2. Diese Teilwege können auch leer sein. Die Notation ist in der folgenden Abbildung veranschaulicht.



Wegen Lemma 5.17 muss P_1 ein kürzester Weg von s nach x sein. Somit gilt am Anfang des Schleifendurchlaufs

$$\delta(s, u) = w(P_1) + w(x, y) + w(P_2) = \delta(s, x) + w(x, y) + w(P_2) \overset{(1)}{=} d(y) + w(P_2) \geq d(y),$$

wobei wir bei (1) den zweiten Teil der Invariante für $y \in V \setminus S$ ausnutzen. Diese besagt, dass $d(y)$ der Länge des kürzesten Weges von s zu y über einen direkten Vorgänger aus S entspricht. Da x ein solcher Vorgänger aus S ist, und laut Lemma 5.17 der entsprechende Teilweg von P ein kürzester Weg von s nach y ist, gilt hier Gleichheit.

Schließlich, da u am Anfang des Schleifendurchlaufs ein Knoten aus $V \setminus S$ mit kleinstem d-Wert ist, muss auch $d(u) \leq d(y)$ gelten. Folglich gilt also $\delta(s, u) \geq d(y) \geq d(u)$.

Zusammengenommen zeigt dies, dass am Anfang des Schleifendurchlaufs $d(u) = \delta(s, u)$ gilt. Da sich der Wert $d(u)$ während des Schleifendurchlaufs nicht ändert, gilt dies auch am Ende. Damit ist der erste Teil der Invariante gezeigt.

Der zweite Teil der Invariante folgt nun daraus, dass vorher $d(v) = \min\{\delta(s, x) + w(x, v) \mid x \in S\}$ für alle $v \in V \setminus S$ galt und dass nun für jeden direkten Nachfolger v von u Zeilen 7 bis 10 aufgerufen werden. Wir haben eben gezeigt, dass $d(u) = \delta(s, u)$ gilt. Dies führt dazu, dass am Ende des Schleifendurchlaufs gilt

$$\begin{aligned} d(v) &= \min\{\min\{\delta(s, x) + w(x, v) \mid x \in S\}, \delta(s, u) + w(u, v)\} \\ &= \min\{\delta(s, x) + w(x, v) \mid x \in S \cup \{u\}\}. \end{aligned}$$

□

Kürzeste-Wege-Bäume

Wir haben gezeigt, dass nach dem Durchlauf des Algorithmus von Dijkstra $d(v) = \delta(s, v)$ für alle $v \in V$ gilt. Wie bereits erwähnt, lassen sich aus den Werten $\pi(v)$ die kürzesten Wege rekonstruieren. Dies ist tatsächlich bemerkenswert: Wir berechnen n kürzeste Wege. Weil jeder von diesen Wegen bis zu $n-1$ Kanten enthält, liegt es nahe, dass für alle Wege insgesamt Speicherplatz $O(n^2)$ benötigt wird. Tatsächlich benötigen wir aber nur Speicherplatz $O(n)$.

Statt nämlich alle n Wege separat zu speichern, haben wir einen Kürzeste-Wege-Baum mit Wurzel s gemäß der folgenden Definition konstruiert.

Definition 5.20. *Wir nennen $G' = (V', E')$ mit $V' \subseteq V$ und $E' \subseteq E$ einen* Kürzeste-Wege-Baum mit Wurzel s*, wenn die folgenden Eigenschaften erfüllt sind.*

1. *V' ist die Menge der Knoten, die von s aus in G erreichbar sind.*
2. *G' ist ein gewurzelter Baum mit Wurzel s. (Das bedeutet, dass G' azyklisch ist, selbst wenn wir die Richtung der Kanten ignorieren, und dass alle Kanten von s weg zeigen.)*
3. *Für alle $v \in V'$ ist der eindeutige Weg von s zu v in G' ein kürzester Weg von s nach v in G.*

Kürzeste-Wege-Bäume benötigen Speicherplatz $O(n)$, da wir für jeden Knoten nur seinen Vorgänger speichern müssen. Sie sind also eine platzsparende Möglichkeit, um alle kürzesten Wege von s zu den anderen Knoten des Graphen zu codieren.

Um den Korrektheitsbeweis für den Algorithmus von Dijkstra zu vervollständigen, müsste man also zeigen, dass die Vorgänger-Relation, die in π berechnet wird, wirklich einen Kürzesten-Wege-Baum mit Wurzel s ergibt. Dies folgt unmittelbar aus dem folgenden Lemma, dessen Beweis wir der Leserin und dem Leser als Übung überlassen.

Lemma 5.21. *Betrachte das Ergebnis des Algorithmus von Dijkstra. Es seien*

$$V_\pi = \{v \in V \mid \pi(v) \neq \textbf{null}\} \cup \{s\} \quad \textit{und} \quad E_\pi = \{(\pi(v), v) \in E \mid v \in V_\pi \setminus \{s\}\}.$$

Dann ist der Graph (V_π, E_π) ein Kürzeste-Wege-Baum mit Wurzel s.

Implementierung und Laufzeit Wir analysieren nun die Laufzeit des Algorithmus von Dijkstra. Dazu müssen wir uns zunächst überlegen, wie wir Zeile 4 realisieren. Wir benötigen eine Datenstruktur, die die Menge $Q = V \setminus S$ verwalten kann. Sie sollte die folgenden drei Operationen möglichst effizient unterstützen:

- Insert(x, d): Füge ein neues Element x mit Schlüssel $d \in \mathbb{R}$ in die Menge Q ein. (Benötigt zur Initialisierung.)
- Extract-Min(): Entferne aus Q ein Element mit dem kleinsten Schlüssel und gib dieses Element zurück. (Benötigt für Zeile 4.)

- DECREASE-KEY(x, d_1, d_2): Ändere den Schlüssel des Objektes $x \in Q$ von d_1 auf $d_2 < d_1$. (Benötigt für Zeile 8.)

Dies entspricht den Prioritätswarteschlangen, die wir in Abschnitt 4.4.3 eingeführt haben mit dem einzigen Unterschied, dass wir die Ordnung der Schlüssel in den Operationen umkehren, d. h. wir haben eine Methode EXTRACT-MIN statt EXTRACT-MAX und eine Methode DECREASE-KEY statt INCREASE-KEY. Hierfür ist keine wesentliche Veränderung an der Datenstruktur notwendig. Gemäß Theorem 4.18 benötigen die drei oben genannten Operationen eine Laufzeit von $O(\log n)$.

Theorem 5.22. *Die Laufzeit des Algorithmus von Dijkstra beträgt $O((n+m)\log n)$, wenn der Graph als Adjazenzliste gegeben ist.*

Beweis. Die Initialisierung benötigt eine Laufzeit von $O(n \log n)$, da jeder Knoten in die Prioritätswarteschlange mithilfe der Methode INSERT eingefügt werden muss. Berücksichtigt man, dass zu Beginn $d(u) = \infty$ für alle Knoten $u \in V \setminus \{s\}$ gilt, so kann der initiale Heap auch direkt in Zeit $O(n)$ aufgebaut werden.

Da die Adjazenzliste eines Knotens $v \in V$ dann das einziges Mal durchlaufen wird, wenn der Knoten v der Menge S hinzugefügt wird, werden Zeilen 7 bis 10 für jede Kante $e = (u, v) \in E$ genau einmal aufgerufen. Hier wird gegebenenfalls der Wert $d(v)$ reduziert. Dies entspricht einem Aufruf von DECREASE-KEY, welcher Laufzeit $O(\log n)$ benötigt. Alle Aufrufe von Zeilen 7 bis 10 zusammen haben somit eine Laufzeit von $O(m \log n)$. Wir benötigen außerdem genau n Aufrufe von EXTRACT-MIN. Diese haben eine Gesamtlaufzeit von $O(n \log n)$. Damit folgt das Theorem. □

5.3.3 All-Pairs Shortest Path Problem

In diesem Abschnitt beschäftigen wir uns mit dem APSP. Dazu sei ein gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$ gegeben. Im Gegensatz zum Dijkstra-Algorithmus funktioniert der Floyd-Warshall-Algorithmus, den wir gleich kennenlernen werden, auch dann, wenn es negative Kantengewichte gibt, solange kein Kreis mit negativem Gesamtgewicht im Graphen enthalten ist. Enthält der Graph einen solchen Kreis, so stellt der Algorithmus dies fest.

Für den Floyd-Warshall-Algorithmus gehen wir davon aus, dass $V = \{1, \ldots, n\}$ gilt, und wir untersuchen Wege, die nur bestimmte Knoten benutzen dürfen. Sei $P = (v_0, \ldots, v_\ell)$ ein kürzester Weg von v_0 nach v_ℓ. Wir haben bereits beim SSSP argumentiert, dass es für jedes Paar von Knoten u und v in einem Graphen ohne negativen Kreis stets einen kürzesten Weg von u nach v gibt, der keinen Knoten mehrfach besucht. Wir nehmen an, dass dies auf P zutrifft. Das heißt, die Knoten $v_0, \ldots, v_\ell$ sind paarweise verschieden. Wir nennen $v_1, \ldots, v_{\ell-1}$ die *Zwischenknoten von P*. Der Floyd-Warshall-Algorithmus basiert auf dynamischer Programmierung. Er löst für jedes Paar $i, j \in V$ von Knoten und jedes $k \in \{0, \ldots, n\}$ das Teilproblem, einen kürzesten Weg von i nach j zu finden, der nur Zwischenknoten aus der Menge $\{1, \ldots, k\}$ besitzt.

Wir betrachten für jedes Paar $i, j \in V$ alle einfachen Wege von i nach j, die nur Zwischenknoten aus der Menge $\{1, \ldots, k\}$ für ein bestimmtes k benutzen dürfen. Sei P_{ij}^k

der kürzeste solche Weg von i nach j und sei $\delta_{ij}^{(k)}$ seine Länge. Wir überlegen uns, wie wir den Weg P_{ij}^k und $\delta_{ij}^{(k)}$ bestimmen können, wenn wir für jedes Paar $i, j \in V$ bereits den Weg P_{ij}^{k-1} kennen, also den kürzesten Weg von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k-1\}$ benutzen darf. Hierzu unterscheiden wir zwei Fälle.

- Enthält der Weg P_{ij}^k den Knoten k nicht als Zwischenknoten, so enthält er nur Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten und somit ist P_{ij}^k auch dann ein kürzester Weg, wenn wir nur diese Knoten als Zwischenknoten erlauben. Es gilt dann also $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$ und $P_{ij}^k = P_{ij}^{k-1}$.
- Enthält der Weg P_{ij}^k den Knoten k als Zwischenknoten, so zerlegen wir ihn wie folgt in zwei Teile:
$$P_{ij}^k = i \overset{P}{\rightsquigarrow} k \overset{P'}{\rightsquigarrow} j.$$
Da P_{ij}^k ein einfacher Weg ist, kommt k nur einmal vor und ist somit nicht in den Teilwegen P und P' enthalten. Diese müssen laut Lemma 5.17 kürzeste Wege von i nach k bzw. von k nach j sein, die nur Knoten aus $\{1, \ldots, k-1\}$ als Zwischenknoten benutzen. Somit gilt in diesem Fall
$$\delta_{ij}^{(k)} = \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}.$$
und
$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j.$$

Aus dieser Beobachtung können wir direkt induktive Formeln zur Berechnung der Entfernungen $\delta^{(k)}(i,j)$ und der Wege P_{ij}^k herleiten. Wir betrachten zunächst nur die Entfernungen. Für alle $i, j \in V$ gilt

$$\delta_{ij}^{(k)} = \begin{cases} w(i,j) & \text{für } k = 0, \\ \min\{\delta_{ij}^{(k-1)}, \delta_{ik}^{(k-1)} + \delta_{kj}^{(k-1)}\} & \text{für } k > 0. \end{cases}$$

Bei der obigen Formel nehmen wir an, dass $w(i,j) = \infty$ für $(i,j) \notin E$ gilt. Für $k = n$ erlauben wir alle Knoten $\{1, \ldots, n\}$ als Zwischenknoten. Demzufolge besteht für $n = k$ keine Einschränkung mehr und es gilt $\delta(i,j) = \delta_{ij}^{(n)}$ für alle $i, j \in V$.

Wir weisen an dieser Stelle nochmals explizit darauf hin, dass wir für diese Rekursionsformel essentiell ausgenutzt haben, dass es im Graphen G keinen Kreis mit negativem Gesamtgewicht gibt. Gibt es nämlich einen solchen Kreis, der von einem Knoten $i \in V$ erreichbar ist und von dem aus ein Knoten $j \in V$ erreichbar ist, so ist der kürzeste Weg von i nach j nicht mehr einfach. Er würde stattdessen den Kreis unendlich oft durchlaufen und demzufolge die Knoten auf dem Kreis mehrfach besuchen. Dies bedeutet insbesondere, dass P_{ij}^k im Allgemeinen nicht definiert ist.

Floyd-Warshall-Algorithmus

Der Floyd-Warshall-Algorithmus nutzt die induktive Darstellung der $\delta_{ij}^{(k)}$, die wir soeben hergeleitet haben, um das APSP zu lösen. Wir gehen davon aus, dass der Graph

als $(n \times n)$-Adjazenzmatrix $W = (w_{ij})$ gegeben ist. Die Adjazenzmatrix W enthält bei gewichteten Graphen an der Stelle w_{ij} das Gewicht der Kante (i, j). Existiert diese Kante nicht, so gilt $w_{ij} = \infty$. Des Weiteren gilt $w_{ii} = 0$ für alle $i \in V$.

```
FLOYD-WARSHALL(W)
 1   D^(0) = W;
 2   for (int k = 1; k ≤ n; k++) {
 3       Erzeuge (n × n)-Nullmatrix D^(k) = (δ_ij^(k)).
 4       for (int i = 1; i ≤ n; i++) {
 5           for (int j = 1; j ≤ n; j++) {
 6               δ_ij^(k) = min{δ_ij^(k-1), δ_ik^(k-1) + δ_kj^(k-1)};
 7           }
 8       }
 9   }
10   return D^(n);
```

Theorem 5.23. *Der Floyd-Warshall-Algorithmus löst das APSP für Graphen ohne Kreise mit negativem Gesamtgewicht, die als Adjazenzmatrix dargestellt sind und n Knoten enthalten, in Zeit $\Theta(n^3)$.*

Beweis. Die Korrektheit des Algorithmus folgt direkt aus der induktiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben, da der Algorithmus die Entfernungen $\delta_{ij}^{(k)}$ exakt anhand dieser Formel berechnet. Die Laufzeit von $\Theta(n^3)$ ist leicht ersichtlich, da wir drei ineinander geschachtelte for-Schleifen haben, die jeweils über n verschiedene Werte zählen, und innerhalb der innersten Schleife nur Operationen ausgeführt werden, die konstante Zeit benötigen. Das Erstellen der Nullmatrizen benötigt insgesamt ebenfalls Zeit $\Theta(n^3)$. □

Wenn wir zusätzlich zu den Entfernungen auch die kürzesten Wege ermitteln wollen, so können wir dies auch wieder mittels einer induktiv definierten Vorgängerfunktion π tun. Sei $\pi_{ij}^{(k)}$ der Vorgänger von j auf dem Weg P_{ij}^k, das heißt auf dem kürzesten Weg von i nach j, der als Zwischenknoten nur Knoten aus $\{1, \ldots, k\}$ benutzen darf. Da für $k = 0$ überhaupt keine Zwischenknoten erlaubt sind, gilt für alle Knoten $i, j \in V$

$$\pi_{ij}^{(0)} = \begin{cases} \textbf{null} & \text{falls } i = j \text{ oder } w_{ij} = \infty, \\ \text{i} & \text{falls } i \neq j \text{ und } w_{ij} < \infty. \end{cases}$$

Das heißt, $\pi_{ij}^{(0)}$ ist genau dann gleich i, wenn es eine direkte Kante von i nach j gibt. Für $k > 0$ orientieren wir uns an der rekursiven Formel für $\delta_{ij}^{(k)}$, die wir oben hergeleitet haben. Ist $\delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}$, so ist auch $P_{ij}^k = P_{ij}^{k-1}$ und somit ist der Vorgänger von j auf dem Weg P_{ij}^k derselbe wie der auf dem Weg P_{ij}^{k-1}. Ist $\delta_{ij}^{(k)} < \delta_{ij}^{(k-1)}$, so gilt

$$P_{ij}^k = i \overset{P_{ik}^{k-1}}{\rightsquigarrow} k \overset{P_{kj}^{k-1}}{\rightsquigarrow} j,$$

und damit ist insbesondere der Vorgänger von j auf dem Weg P_{ij}^k derselbe wie auf dem Weg P_{kj}^{k-1}. Zusammenfassend halten wir für $k > 0$ fest

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{falls } \delta_{ij}^{(k)} = \delta_{ij}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{falls } \delta_{ij}^{(k)} < \delta_{ij}^{(k-1)}. \end{cases}$$

Die $\pi_{ij}^{(k)}$ können zusammen mit den $\delta_{ij}^{(k)}$ im Floyd-Warshall-Algorithmus berechnet werden, ohne dass sich die Laufzeit signifikant ändert.

Negative Kreise Wir wollen nun noch kurz diskutieren, wie man den Floyd-Warshall-Algorithmus benutzen kann, um die Existenz eines Kreises mit negativem Gesamtgewicht zu detektieren. Dazu müssen wir uns nur überlegen, was die Einträge $\delta_{ii}^{(k)}$ auf der Diagonalen der Matrix $D^{(k)}$ bedeuten. Für $k = 0$ werden diese Einträge alle mit 0 initialisiert. Gibt es nun für $k > 0$ einen Kreis mit negativem Gesamtgewicht, der den Knoten i enthält und sonst nur Knoten aus der Menge $\{1, \ldots, k\}$, so gilt $\delta_{ii}^{(k)} < 0$ und somit auch $\delta_{ii}^{(n)} < 0$, da es einen Weg negativer Länge von i nach i gibt. Also genügt es, am Ende des Algorithmus, die Einträge auf der Hauptdiagonalen zu betrachten. Ist mindestens einer von diesen negativ, so wissen wir, dass ein Kreis mit negativem Gesamtgewicht existiert. In diesem Fall gibt die vom Algorithmus berechnete Matrix $D^{(n)}$ nicht die korrekten Abstände an.

Vergleich mit Algorithmus von Dijkstra Das APSP kann auch dadurch gelöst werden, dass wir einen Algorithmus für das SSSP für jeden möglichen Startknoten einmal aufrufen. Hat der Algorithmus für das SSSP eine Laufzeit von $O(f)$, so ergibt sich zur Lösung des APSP eine Laufzeit von $O(nf)$. Für den Dijkstra-Algorithmus ergibt sich also eine Laufzeit von $O(nm \log n)$.

Für Graphen mit nicht-negativen Kantengewichten stellt sich somit die Frage, ob es effizienter ist, den Floyd-Warshall-Algorithmus mit Laufzeit $O(n^3)$ oder den Dijkstra-Algorithmus mit Laufzeit $O(nm \log n)$ zur Lösung des APSP zu nutzen. Diese Frage lässt sich nicht allgemein beantworten. Für dichte Graphen mit $m = \Theta(n^2)$ vielen Kanten ist der Floyd-Warshall-Algorithmus schneller. Für Graphen mit $m = o(n^2/\log n)$ Kanten ist hingegen die n-malige Ausführung des Dijkstra-Algorithmus schneller.

5.4 Flussprobleme

In diesem Abschnitt lernen wir mit Flussproblemen eine weitere wichtige Klasse von algorithmischen Graphenproblemen kennen, die in vielen Bereichen Anwendung findet. Ein *Flussnetzwerk* ist ein gerichteter Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten $s, t \in V$ und einer *Kapazitätsfunktion* $c : V \times V \to \mathbb{N}_0$, die jeder Kante $(u, v) \in E$ eine ganzzahlige nicht-negative Kapazität $c(u, v)$ zuweist. Wir definieren $c(u, v) = 0$ für alle $(u, v) \notin E$. Außerdem nennen wir den Knoten s die *Quelle* und den Knoten t die *Senke*. Wir definieren $n = |V|$ und $m = |E|$. Außerdem nehmen wir an, dass jeder Knoten von der Quelle s zu erreichen ist. Das bedeutet insbesondere, dass $m \geq n - 1$ gilt.

Definition 5.24. *Ein* Fluss *in einem Flussnetzwerk ist eine Funktion $f : V \times V \to \mathbb{R}$, die die folgenden beiden Eigenschaften erfüllt.*

1. Flusserhaltung*: Für jeden Knoten $u \in V \setminus \{s, t\}$ gilt*
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$
2. Kapazitätsbeschränkung*: Für alle Knoten $u, v \in V$ gilt*
$$0 \leq f(u, v) \leq c(u, v).$$

Wir definieren den Wert *eines Flusses $f : V \times V \to \mathbb{R}$ als*
$$|f| := \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$
und weisen explizit darauf hin, dass die Notation $|.|$ hier keinen Betrag bezeichnet, sondern den Wert des Flusses gemäß obiger Definition.

Das Flussproblem, das wir uns in diesem Abschnitt anschauen werden, lässt sich nun wie folgt formulieren:

> Gegeben sei ein Flussnetzwerk G. Berechne einen maximalen Fluss in G,
> d. h. einen Fluss f mit größtmöglichem Wert $|f|$.

Anschaulich ist die Frage also, wie viele Einheiten Fluss man in dem gegebenen Netzwerk maximal von der Quelle zur Senke transportieren kann.

Als erste einfache Beobachtung halten wir fest, dass der Fluss, der in der Quelle entsteht, komplett in der Senke ankommt und nicht zwischendurch versickert.

Lemma 5.25. *Sei f ein Fluss in einem Flussnetzwerk G. Dann gilt*
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) = \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v).$$

Beweis. Als erstes beobachten wir, dass die folgenden beiden Summen gleich sind:
$$\sum_{u \in V} \sum_{v \in V} f(v, u) = \sum_{u \in V} \sum_{v \in V} f(u, v).$$
Dies folgt einfach daraus, dass wir sowohl links als auch rechts über jedes (geordnete) Knotenpaar genau einmal summieren. Nun nutzen wir die Flusserhaltung für alle Knoten $u \in V \setminus \{s, t\}$ und erhalten aus obiger Gleichung:
$$\begin{aligned}
\sum_{u \in V \setminus \{s,t\}} \sum_{v \in V} f(v, u) + \sum_{u \in \{s,t\}} \sum_{v \in V} f(v, u) &= \sum_{u \in V \setminus \{s,t\}} \sum_{v \in V} f(u, v) + \sum_{u \in \{s,t\}} \sum_{v \in V} f(u, v) \\
\iff \quad \sum_{u \in \{s,t\}} \sum_{v \in V} f(v, u) &= \sum_{u \in \{s,t\}} \sum_{v \in V} f(u, v) \\
\iff \quad \sum_{v \in V} f(v, s) + \sum_{v \in V} f(v, t) &= \sum_{v \in V} f(s, v) + \sum_{v \in V} f(t, v) \\
\iff \quad \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) &= \sum_{v \in V} f(v, t) - \sum_{v \in V} f(t, v),
\end{aligned}$$
woraus mit der Definition von $|f|$ direkt das Lemma folgt. □

5.4.1 Anwendungsbeispiele

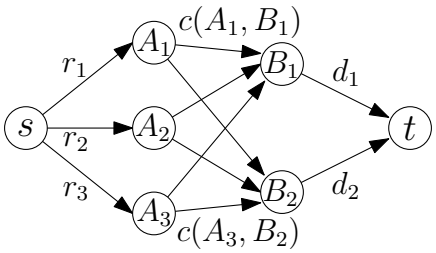
Bevor wir Algorithmen zur Lösung des Flussproblems vorstellen, präsentieren wir ein Anwendungsbeispiel, das wir aus Kapitel 4.5 von [2] übernommen haben. Nehmen wir an, in den Seehäfen $A_1, \ldots, A_p$ liegen Bananen zur Verschiffung bereit. Es gibt Zielhäfen $B_1, \ldots, B_q$, zu denen die Bananen transportiert werden sollen. Für $i = 1, \ldots, p$ bezeichne r_i, wie viele Tonnen Bananen im Hafen A_i bereit stehen, und für $j = 1, \ldots, q$ bezeichne d_j wie viele Tonnen Bananen am Zielhafen B_j angefordert wurden. Für $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es zwischen den Häfen A_i und B_j eine Schifffahrtslinie, die maximal $c(A_i, B_j)$ Tonnen Bananen transportieren kann. Es kann auch sein, dass es zwischen zwei Häfen A_i und B_j keine Schifffahrtslinie gibt; in diesem Falle definieren wir $c(A_i, B_j)$ als 0. Die folgenden Fragen stellen sich dem Handelsunternehmen:

1. Ist es möglich, alle Anforderungen zu erfüllen?
2. Falls nein, wie viele Tonnen Bananen können maximal zu den Zielhäfen transportiert werden?
3. Wie sollen die Bananen verschifft werden?

Alle diese Fragen können als Flussproblem modelliert werden. Dazu konstruieren wir einen Graphen $G = (V, E)$ mit den Knoten $V = \{s, t, A_1, \ldots, A_p, B_1, \ldots, B_q\}$. Es gibt drei Typen von Kanten:

- Für $i = 1, \ldots, p$ gibt es eine Kante (s, A_i) mit Kapazität r_i.
- Für $j = 1, \ldots, q$ gibt es eine Kante (B_j, t) mit Kapazität d_j.
- Für jedes $i = 1, \ldots, p$ und $j = 1, \ldots, q$ gibt es eine Kante (A_i, B_j) mit Kapazität $c(A_i, B_j)$.

Die folgende Abbildung zeigt ein Beispiel für $p = 3$ und $q = 2$.



Sei $f : E \to \mathbb{R}$ ein maximaler Fluss im Graphen G. Man kann sich anschaulich schnell davon überzeugen, dass man mithilfe dieses Flusses alle drei Fragen, die oben gestellt wurden, beantworten kann.

1. Wir können alle Anforderungen befriedigen, wenn der Wert $|f|$ des Flusses gleich $\sum_{j=1}^{q} d_j$ ist.

2. Wir können maximal $|f|$ Tonnen Bananen zu den Zielhäfen transportieren.
3. Der Fluss $f(e)$ auf Kanten $e = (A_i, B_j)$ gibt an, wie viele Tonnen Bananen entlang der Schifffahrtslinie von A_i nach B_j transportiert werden sollen.

Wir verzichten an dieser Stelle auf einen formalen Beweis, dass dies die korrekten Antworten auf die obigen Fragen sind, möchten aber die Leserin und den Leser dazu anregen, selbst darüber nachzudenken.

Flussprobleme können auch dazu genutzt werden, andere Probleme zu modellieren, die auf den ersten Blick ganz anders geartet sind. Ein Beispiel (siehe Übungen) ist es, zu entscheiden, ob eine Fußballmannschaft bei einer gegebenen Tabellensituation und einem gegebenen restlichen Spielplan gemäß der alten Zwei-Punkte-Regel noch Meister werden kann.

5.4.2 Algorithmus von Ford und Fulkerson

Der folgende Algorithmus von Ford und Fulkerson berechnet für ein gegebenes Flussnetzwerk $G = (V, E)$ mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ einen maximalen Fluss. Wir gehen in diesem Abschnitt davon aus, dass das Flussnetzwerk dergestalt ist, dass für kein Paar von Knoten $u, v \in V$ die beiden Kanten (u, v) und (v, u) in E enthalten sind. Dies ist keine wesentliche Einschränkung, da jedes Netzwerk, das diese Eigenschaft verletzt, einfach in ein äquivalentes umgebaut werden kann, das keine entgegengesetzten Kanten enthält. Es ist eine gute Übung, sich zu überlegen, wie dies funktioniert.

```
FORD-FULKERSON(G, c, s ∈ V, t ∈ V)
1   Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2   while (∃ flussvergrößernder Weg P) {
3       Erhöhe den Fluss f entlang P.
4   }
5   return f;
```

Wir müssen noch beschreiben, was ein *flussvergrößernder Weg* ist und was es bedeutet, den Fluss entlang eines solchen Weges zu erhöhen. Dazu führen wir den Begriff des *Restnetzwerkes* ein.

Definition 5.26. *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}_0$ und sei f ein Fluss in G. Das dazugehörige* Restnetzwerk $G_f = (V, E_f)$ *ist auf der gleichen Menge von Knoten V definiert wie das Netzwerk G. Wir definieren eine Funktion* $\mathrm{rest}_f : V \times V \to \mathbb{R}$ *mit*

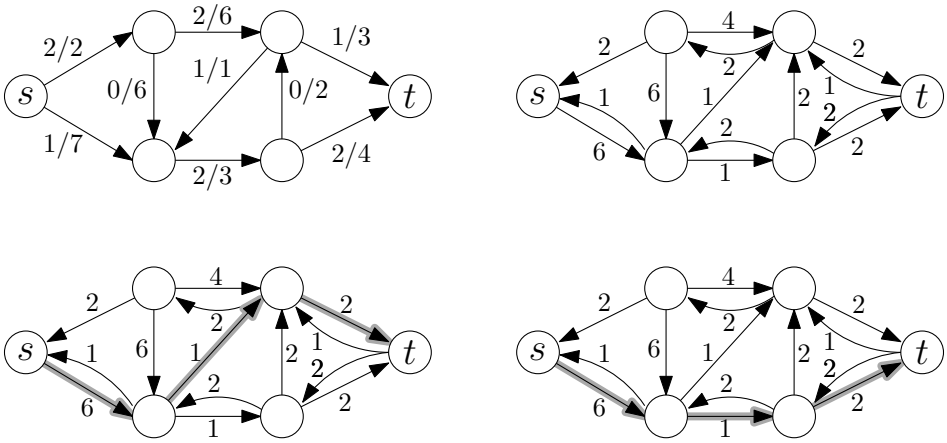
$$\mathrm{rest}_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E, \\ f(v, u) & \text{falls } (v, u) \in E, \\ 0 & \text{sonst.} \end{cases}$$

Die Kantenmenge E_f ist definiert als

$$E_f = \{(u, v) \in V \times V \mid \mathrm{rest}_f(u, v) > 0\}.$$

Ein flussvergrößernder Weg *ist ein einfacher Weg von s nach t im Restnetzwerk G_f.*

Die folgende Abbildung zeigt oben links ein Beispiel für ein Flussnetzwerk mit einem Fluss f. Dabei bedeutet die Beschriftung a/b an einer Kante e, dass $f(e) = a$ und $c(e) = b$ gilt. Rechts oben ist das zugehörige Restnetzwerk dargestellt, und in der unteren Reihe sind zwei flussvergrößernde Wege zu sehen.



Nun müssen wir noch klären, was es bedeutet, den Fluss entlang eines Weges zu erhöhen.

Definition 5.27. *Sei G ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$, sei $f : V \times V \to \mathbb{R}$ ein Fluss und sei P ein einfacher Weg im Restnetzwerk G_f von s nach t. Wir bezeichnen mit $f \uparrow P : V \times V \to \mathbb{R}$ den Fluss, der entsteht, wenn wir f entlang P erhöhen. Dieser Fluss ist definiert durch*

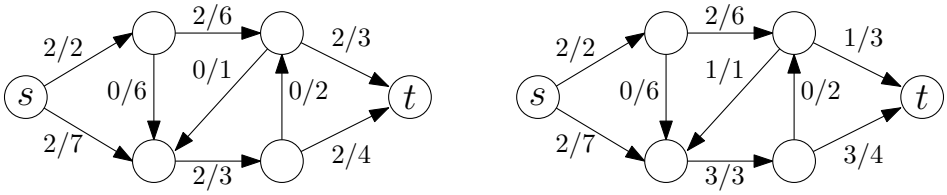
$$(f \uparrow P)(u,v) = \begin{cases} f(u,v) + \delta & \textit{falls } (u,v) \in E \textit{ und } (u,v) \in P, \\ f(u,v) - \delta & \textit{falls } (u,v) \in E \textit{ und } (v,u) \in P, \\ f(u,v) & \textit{sonst,} \end{cases}$$

wobei

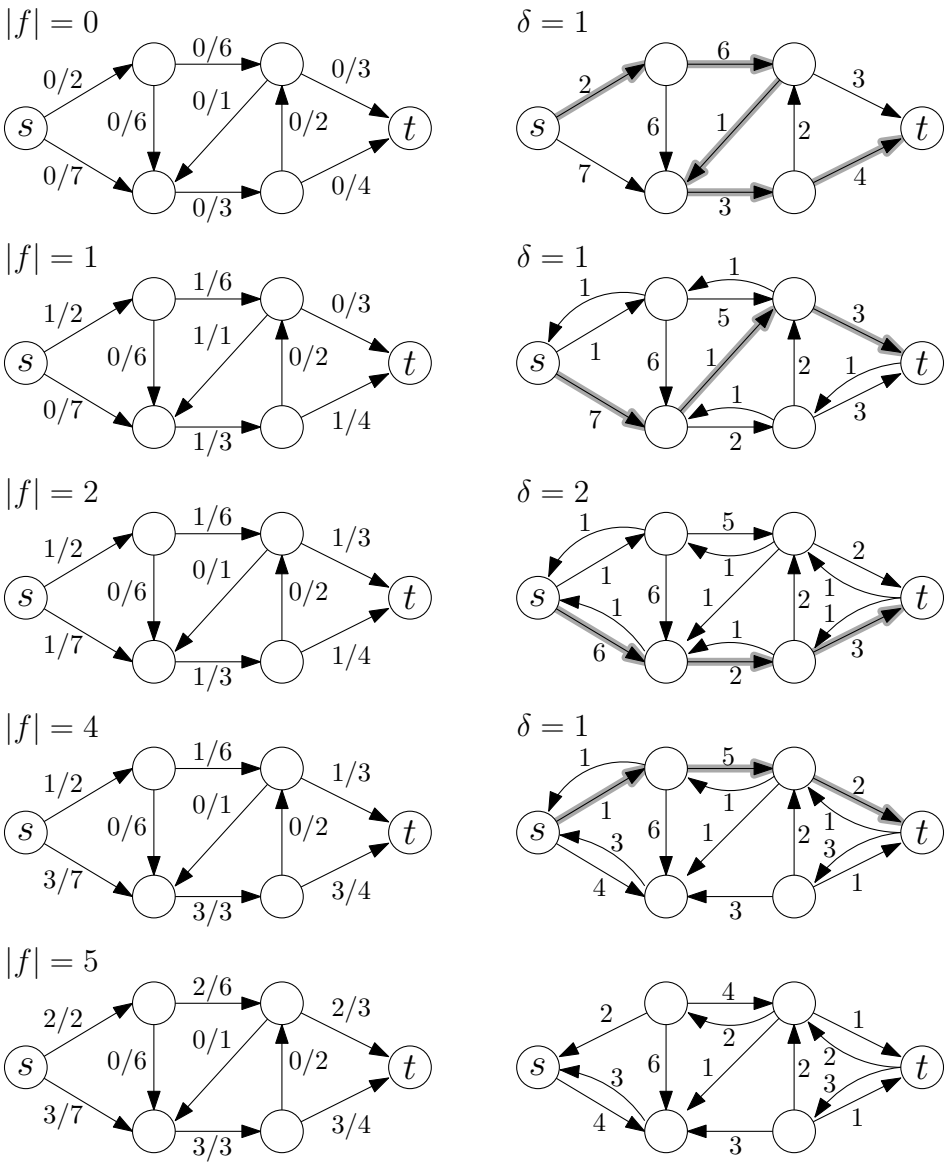
$$\delta = \min_{e \in P}(\mathrm{rest}_f(e)).$$

Aus der Definition von E_f folgt, dass $\delta > 0$ gilt.

In der folgenden Abbildung sind die Flüsse dargestellt, die entstehen, wenn wir den Fluss in obiger Abbildung entlang der beiden oben dargestellten Wege erhöhen. In beiden Beispielen ist $\delta = 1$.



Die folgende Abbildung zeigt ein Beispiel für die Ausführung des Algorithmus von Ford und Fulkerson.



Korrektheit

Um die Korrektheit des Algorithmus von Ford und Fulkerson zu zeigen, müssen wir zunächst zeigen, dass $f \uparrow P$ wieder ein Fluss ist.

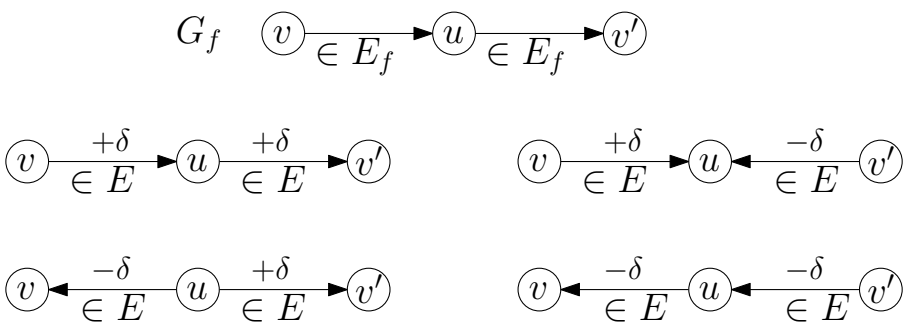
Lemma 5.28. *Sei f ein Fluss in einem Netzwerk G und sei P ein Weg von s nach t im Restnetzwerk G_f. Die Funktion $f \uparrow P : V \times V \to \mathbb{R}$ ist wieder ein Fluss. Für diesen Fluss gilt*

$$|f \uparrow P| = |f| + \delta.$$

Beweis. Wir müssen zeigen, dass die beiden Eigenschaften aus Definition 5.24 erfüllt sind.

- *Flusserhaltung*: Sei $u \in V \setminus \{s,t\}$. Falls u auf dem Weg P nicht vorkommt, so gilt $f(u,v) = (f \uparrow P)(u,v)$ und $f(v,u) = (f \uparrow P)(v,u)$ für alle Knoten $v \in V$ und somit folgt die Flusserhaltung für u in $f \uparrow P$ aus der Flusserhaltung für u im Fluss f.

  Kommt u auf dem Weg P vor, so gibt es genau eine Kante $e \in P$ der Form (v,u) für ein $v \in V$ und es gibt genau eine Kante e' der Form (u,v') für ein $v' \in V$. Die Eindeutigkeit folgt daraus, dass P per Definition einfach ist und somit den Knoten u nur einmal besucht. Die Kanten e und e' sind die einzigen zu u inzidenten Kanten, auf denen sich der Fluss ändert. Wir brauchen also nur diese beiden Kanten näher zu betrachten. Es gilt entweder $(v,u) \in E$ oder $(u,v) \in E$. Ebenso gilt entweder $(v',u) \in E$ oder $(u,v') \in E$. Wir schauen uns alle vier Fälle einzeln an und überprüfen, dass die Flusserhaltung erhalten bleibt. In der folgenden Abbildung ist oben der relevante Ausschnitt des Restnetzwerkes dargestellt und darunter die vier Fälle. In jedem der Fälle ist leicht zu sehen, dass die Flusserhaltung an u erhalten bleibt.



- *Kapazitätsbeschränkung*: Seien $u, v \in V$ beliebig so gewählt, dass $e = (u,v) \in E_f$ eine Kante auf dem Weg P in G_f ist. Wir zeigen, dass $0 \leq (f \uparrow P)(e) \leq c(e)$ gilt. Dafür unterscheiden wir zwei Fälle.

  – Ist $e = (u,v) \in E$, so erhöhen wir den Fluss auf e um δ. Aufgrund der Definition von δ und wegen $f(e) \geq 0$ und $\delta \geq 0$ gilt

  $$0 \leq f(e) + \delta \leq f(e) + \mathrm{rest}_f(e) = f(e) + (c(e) - f(e)) = c(e).$$

  – Ist $e' = (v,u) \in E$, so verringern wir den Fluss auf e' um δ. Aufgrund der Definition von δ und wegen $f(e') \leq c(e')$ und $\delta \geq 0$ gilt

  $$c(e') \geq f(e') - \delta \geq f(e') - \mathrm{rest}_f(e) \geq f(e') - f(e') = 0.$$

Es bleibt zu zeigen, dass $|f \uparrow P| = |f| + \delta$ gilt. Dazu beobachten wir, dass der einfache Weg P genau eine zu s inzidente Kante enthält. Diese ist von der Form $(s,v) \in E_f$. Gilt $(s,v) \in E$, so gilt $(f \uparrow P)(s,v) = f(s,v) + \delta$, woraus direkt die gewünschte Aussage $|f \uparrow P| = |f| + \delta$ folgt. Man kann leicht argumentieren, dass der Fall $(v,s) \in E$ nicht eintreten kann. Selbst wenn er eintritt, gilt aber wegen $(f \uparrow P)(v,s) = f(v,s) - \delta$ die gewünschte Aussage. □

Aus Lemma 5.28 folgt direkt, dass die Funktion f, die der Algorithmus von Ford und Fulkerson verwaltet, nach jeder Iteration ein Fluss ist. Wir zeigen nun, dass der Algorithmus, wenn er terminiert, einen maximalen Fluss berechnet hat. Dazu führen wir zunächst die folgende Definition ein.

Definition 5.29. *Sei $G = (V, E)$ ein Flussnetzwerk mit Kapazitäten $c : V \times V \to \mathbb{N}$. Sei $s \in V$ die Quelle und $t \in V$ die Senke. Ein* Schnitt *von G ist eine Partition der Knotenmenge V in zwei Teile $S \subseteq V$ und $T \subseteq V$ mit $s \in S$, $t \in T$ und $T = V \setminus S$. Wir nennen*

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

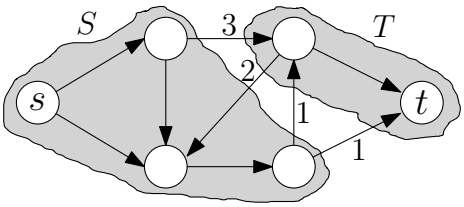
die Kapazität *des Schnittes (S,T). Ein Schnitt (S,T) heißt* minimal*, wenn es keinen Schnitt (S',T') mit $c(S',T') < c(S,T)$ gibt.*

Für einen Fluss f und einen Schnitt (S,T) sei

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

der Fluss über den Schnitt.

Die folgende Abbildung zeigt einen Schnitt mit Kapazität 5.



Anschaulich ist leicht einzusehen, dass die Kapazität $c(S,T)$ für jeden Schnitt (S,T) eine obere Schranke dafür ist, wie viel Fluss von der Quelle zur Senke geschickt werden kann, und dass für einen Fluss f der Fluss über den Schnitt (S,T) genau $|f|$ sein muss. Formal zeigen wir das in folgendem Lemma.

Lemma 5.30. *Sei (S,T) ein Schnitt eines Flussnetzwerkes G und sei f ein Fluss in G. Dann gilt*

$$|f| = f(S,T) \leq c(S,T).$$

Beweis. Ähnlich zu dem Beweis von Lemma 5.25 summieren wir zunächst über alle Kanten, die innerhalb der Menge S verlaufen. Daraus ergibt sich

$$\sum_{u \in S} \sum_{v \in S} f(u,v) = \sum_{u \in S} \sum_{v \in S} f(v,u).$$

Durch Abtrennen der Summanden für $u = s$ erhalten wir

$$\sum_{v \in S} f(s,v) + \sum_{u \in S \setminus \{s\}} \sum_{v \in S} f(u,v) = \sum_{v \in S} f(v,s) + \sum_{u \in S \setminus \{s\}} \sum_{v \in S} f(v,u)$$

$$\iff \sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(v,u) - \sum_{u\in S\setminus\{s\}} \sum_{v\in S} f(u,v). \tag{5.2}$$

Für jeden Knoten $u \in S \setminus \{s\}$ gilt die Flusserhaltung. Dafür müssen wir aber auch die Kanten betrachten, die von u in die Menge T führen, und die Kanten, die von der Menge T zu u führen. Wir erhalten für alle $u \in S \setminus \{s\}$

$$\sum_{v\in S} f(u,v) + \sum_{v\in T} f(u,v) = \sum_{v\in S} f(v,u) + \sum_{v\in T} f(v,u).$$

Damit können wir (5.2) schreiben als

$$\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u). \tag{5.3}$$

Für die Quelle s gilt entsprechend

$$\begin{aligned} &\sum_{v\in S} f(s,v) + \sum_{v\in T} f(s,v) = |f| + \sum_{v\in S} f(v,s) + \sum_{v\in T} f(v,s) \\ \iff &\sum_{v\in S} f(s,v) - \sum_{v\in S} f(v,s) = |f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v). \end{aligned} \tag{5.4}$$

Wir setzen (5.4) in (5.3) ein und erhalten

$$\begin{aligned} &|f| + \sum_{v\in T} f(v,s) - \sum_{v\in T} f(s,v) = \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(u,v) - \sum_{u\in S\setminus\{s\}} \sum_{v\in T} f(v,u) \\ \iff &|f| = \sum_{u\in S} \sum_{v\in T} f(u,v) - \sum_{u\in S} \sum_{v\in T} f(v,u) = f(S,T). \end{aligned}$$

Die Ungleichung folgt einfach aus obiger Gleichung, da $f(u,v) \le c(u,v)$ und $f(v,u) \ge 0$ für alle $u, v \in V$:

$$f(S,T) \le \sum_{u\in S} \sum_{v\in T} f(u,v) \le \sum_{u\in S} \sum_{v\in T} c(u,v) = c(S,T). \qquad \square$$

Mithilfe von Lemma 5.30 zeigen wir das folgende Theorem, aus dem direkt folgt, dass der Algorithmus von Ford und Fulkerson einen maximalen Fluss gefunden hat, wenn er terminiert.

Theorem 5.31. *Sei f ein Fluss in einem Netzwerk G. Dann sind die folgenden drei Aussagen äquivalent.*

a) f ist ein maximaler Fluss.

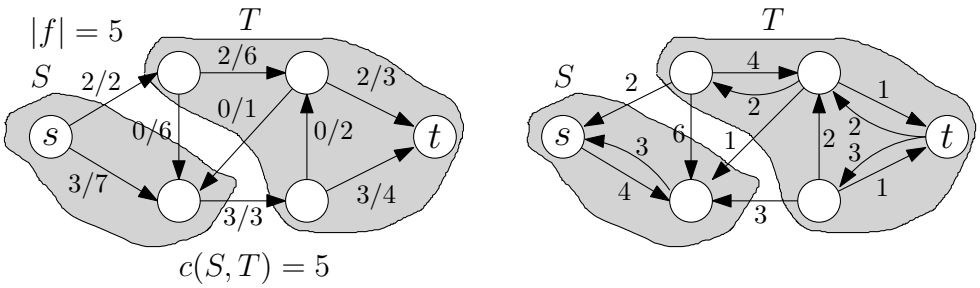
b) Das Restnetzwerk G_f enthält keinen flussvergrößernden Weg.

c) Es gibt einen Schnitt (S,T) mit $|f| = c(S,T)$.

Dieses Theorem besagt insbesondere, dass es einen Schnitt (S,T) geben muss, dessen Kapazität so groß ist wie der Wert $|f|$ des maximalen Flusses f. Zusammen mit Lemma 5.30, das besagt, dass es keinen Schnitt (S',T') mit kleinerer Kapazität als $|f|$ geben kann, folgt die wichtige Aussage, dass die Kapazität des minimalen Schnittes gleich dem Wert des maximalen Flusses ist. Die Korrektheit des Algorithmus von Ford und Fulkerson bei Terminierung folgt einfach daraus, dass der Algorithmus erst dann terminiert, wenn es keinen flussvergrößernden Weg mehr gibt. Nach dem Theorem ist das aber genau dann der Fall, wenn der Fluss maximal ist.

Beweis von Theorem 5.31. Wir werden zeigen, dass b) aus a) folgt, dass c) aus b) folgt und dass a) aus c) folgt. Diese drei Implikationen bilden einen Ringschluss und damit ist gezeigt, dass die drei Aussagen äquivalent sind.

- **a) ⇒ b)** Wir führen einen Widerspruchsbeweis und nehmen an, dass es einen flussvergrößernden Weg P in G_f gibt. Dann können wir den Fluss f entlang P erhöhen und erhalten den Fluss $f \uparrow P$, welcher nach Lemma 5.28 einen um $\delta > 0$ größeren Wert als f hat. Damit kann f kein maximaler Fluss sein.
- **c) ⇒ a)** Sei f ein Fluss mit $|f| = c(S,T)$ für einen Schnitt (S,T) und sei f' ein maximaler Fluss. Dann gilt $|f| \leq |f'|$. Wenn wir Lemma 5.30 für den Fluss f' und den Schnitt (S,T) anwenden, dann erhalten wir $|f'| \leq c(S,T)$. Zusammen ergibt das
$$c(S,T) = |f| \leq |f'| \leq c(S,T),$$
woraus $c(S,T) = |f| = |f'|$ folgt. Damit ist auch f ein maximaler Fluss.
- **b) ⇒ c)** Laut Voraussetzung gibt es keinen flussvergrößernden Weg, das heißt, es gibt keinen Weg von s nach t im Restnetzwerk G_f. Wir teilen die Knoten V des Restnetzwerkes in zwei Klassen S und T ein:
$$S = \{v \in V \mid \text{es gibt Weg in } G_f \text{ von } s \text{ nach } v\} \quad \text{und} \quad T = V \setminus S.$$
Da der Knoten t in G_f nicht von s aus erreichbar ist, gilt $s \in S$ und $t \in T$. Damit ist (S,T) ein Schnitt des Graphen. Die Situation ist in der folgenden Abbildung dargestellt.



Wir argumentieren nun, dass $|f| = c(S,T)$ für diesen Schnitt (S,T) gilt.

  - Sei $(u,v) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(u,v) = c(u,v)$.
  - Sei $(v,u) \in E$ eine Kante mit $u \in S$ und $v \in T$. Da u in G_f von s aus erreichbar ist, v aber nicht, gilt $(u,v) \notin E_f$. Mit der Definition des Restnetzwerkes folgt $\text{rest}_f(u,v) = 0$, also $f(v,u) = 0$.

Somit gilt mit Lemma 5.30

$$\begin{aligned} c(S,T) &= \sum_{u \in S} \sum_{v \in T} c(u,v) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &= f(S,T) = |f| \leq c(S,T). \end{aligned}$$

Damit folgt $|f| = f(S,T) = c(S,T)$. □

Laufzeit

Wir haben den Korrektheitsbeweis noch nicht abgeschlossen. Wir haben lediglich gezeigt, dass der Algorithmus *partiell korrekt* ist. Das bedeutet, wir haben gezeigt, dass der Algorithmus das richtige Ergebnis liefert, falls er terminiert. Wir haben aber noch nicht gezeigt, dass er wirklich auf jeder Eingabe terminiert. Das erledigen wir nun implizit bei der Laufzeitabschätzung.

Theorem 5.32. *Für ganzzahlige Kapazitäten $c : V \times V \to \mathbb{N}_0$ ist die Anzahl an Iterationen der while-Schleife im Algorithmus von Ford und Fulkerson durch $C = \sum_{e \in E} c(e)$ nach oben beschränkt. Die Laufzeit des Algorithmus beträgt $O(mC)$.*

Beweis. Man kann sehr einfach (z. B. mithilfe einer Invariante) zeigen, dass der aktuelle Fluss f, den der Algorithmus verwaltet, stets ganzzahlig ist, dass also stets $f : V \times V \to \mathbb{N}_0$ gilt. Dies folgt einfach daraus, dass für einen ganzzahligen Fluss f und einen flussvergrößernden Weg P stets $\delta \in \mathbb{N}$ gilt. Demzufolge steigt der Wert des Flusses mit jeder Iteration der while-Schleife um mindestens eins an. Da der Wert eines jeden Flusses durch $\sum_{e \in E} c(e)$ beschränkt ist, garantiert dies die Terminierung nach höchstens C vielen Schritten.

Für einen gegebenen Fluss f kann das Restnetzwerk G_f in Zeit $O(m)$ berechnet werden. Ein Weg P von s nach t in G_f kann mittels Tiefensuche in Zeit $O(m)$ gefunden werden. Ist ein solcher Weg P gefunden, so kann in Zeit $O(m)$ der neue Fluss $f \uparrow P$ berechnet werden. Insgesamt dauert also jede Iteration Zeit $O(m)$, woraus das Theorem folgt. □

Aus diesem Theorem und seinem Beweis folgt direkt, dass es bei ganzzahligen Kapazitäten auch stets einen ganzzahligen maximalen Fluss gibt. Dies ist eine sehr wichtige Eigenschaft für viele Anwendungen, weswegen wir sie explizit als Korollar festhalten.

Korollar 5.33. *Sind alle Kapazitäten ganzzahlig, so gibt es stets einen ganzzahligen maximalen Fluss.*

5.4.3 Algorithmus von Edmonds und Karp

Die Laufzeit $O(mC)$, die wir in Theorem 5.32 bewiesen haben, ist nicht besonders gut, weil sie mit den Kapazitäten der Kanten wächst. Dies steht im Gegensatz zu den Laufzeiten von zum Beispiel den Algorithmen von Kruskal und Dijkstra. Für diese Algorithmen haben wir Laufzeitschranken bewiesen, die nur von der Größe des Graphen abhängen, nicht jedoch von den Gewichten oder Längen der Kanten. Eine Laufzeit wie $O(mC)$, die von den Kapazitäten abhängt, heißt *pseudopolynomielle Laufzeit*. Wir werden diesen Begriff im nächsten Semester weiter vertiefen. Hier sei nur angemerkt, dass man, wenn möglich, Laufzeiten anstrebt, die nur von der Größe des Graphen, nicht jedoch von den Kapazitäten abhängen, da diese auch bei einer kleinen Codierungslänge sehr groß werden können. Dies haben wir analog bereits in Abschnitt 2.3.2 beim dynamischen Programm für das Rucksackproblem diskutiert.

Wir werden nun den *Algorithmus von Edmonds und Karp* kennenlernen. Dabei handelt es sich um eine leichte Modifikation des Algorithmus von Ford und Fulkerson, deren Laufzeit nur von der Größe des Graphen abhängt. Bei der Beschreibung des Algorithmus von Ford und Fulkerson haben wir offen gelassen, wie die flussvergrößernden Wege gewählt werden, wenn es mehrere zur Auswahl gibt. Unsere Analyse von Korrektheit und Laufzeit war unabhängig von dieser Wahl. Wir zeigen nun, dass für eine geeignete Wahl die Laufzeit reduziert werden kann.

```
EDMONDS-KARP(G, c, s ∈ V, t ∈ V)
1  Setze f(e) = 0 für alle e ∈ E.   // f ist gültiger Fluss mit Wert 0
2  while (∃ flussvergrößernder Weg P) {
3      Wähle einen flussvergrößernden Weg P mit so wenig Kanten wie möglich.
4      Erhöhe den Fluss f entlang P.
5  }
6  return f;
```

Der einzige Unterschied zum Algorithmus von Ford und Fulkerson ist also, dass immer ein kürzester flussvergrößernder Weg gewählt wird, wobei die Länge eines Weges als die Anzahl seiner Kanten definiert ist. Wir werden in diesem Zusammenhang auch von der *Distanz von s zu $v \in V$ im Restnetzwerk G_f* sprechen. Damit meinen wir, wie viele Kanten der kürzeste Weg (d. h. der Weg mit den wenigsten Kanten) von s nach v in G_f hat.

Theorem 5.34. *Der Algorithmus von Edmonds und Karp besitzt (auch für Graphen mit nicht ganzzahligen Kapazitäten) eine Laufzeit von $O(m^2 n) = O(n^5)$.*

Beweis. Eine Iteration der while-Schleife kann immer noch in Zeit $O(m)$ durchgeführt werden. Einen kürzesten flussvergrößernden Weg findet man nämlich zum Beispiel, indem man von s startend eine Breitensuche im Restnetzwerk durchführt. Diese benötigt Zeit $O(m)$. Zum Beweis des Theorems genügt es also zu zeigen, dass der Algorithmus nach $O(mn)$ Iterationen terminiert. Dazu zeigen wir zunächst die folgende strukturelle Eigenschaft über die Distanzen im Restnetzwerk.

Lemma 5.35. *Sei $x \in V$ beliebig. Die Distanz von s zu x in G_f wird im Laufe des Algorithmus nicht kleiner.*

Beweis. Wir betrachten eine Iteration der while-Schleife, in der der Fluss f entlang des Weges P zu dem Fluss $f \uparrow P$ erhöht wird. Die Kantenmenge $E_{f\uparrow P}$ unterscheidet sich von der Kantenmenge E_f wie folgt.

- Für jede Kante $(u, v) \in P \subseteq E_f$ verringert sich $\text{rest}_f(u, v)$ um δ, das heißt $\text{rest}_{f\uparrow P}(u, v) = \text{rest}_f(u, v) - \delta$. Eine Kante mit $\text{rest}_f(u, v) = \delta$ heißt *Flaschenhalskante* und sie ist in $E_{f\uparrow P}$ nicht mehr enthalten.
- Für jede Kante $(u, v) \in P \subseteq E_f$ erhöht sich die Restkapazität $\text{rest}_f(v, u)$ der entgegengesetzten Kante um δ, das heißt $\text{rest}_{f\uparrow P}(v, u) = \text{rest}_f(v, u) + \delta$. War $\text{rest}_f(v, u) = 0$, so war $(v, u) \notin E_f$, nun gilt aber $(v, u) \in E_{f\uparrow P}$.

Wir spalten den Übergang von E_f zu $E_{f\uparrow P}$ in mehrere Schritte auf. Zunächst fügen wir alle neuen Kanten, die gemäß dem zweiten Aufzählungspunkt entstehen, nach und nach ein. Anschließend entfernen wir die Kanten nach und nach gemäß dem ersten Aufzählungspunkt, um die Kantenmenge $E_{f\uparrow P}$ zu erhalten. Nach jedem Einfügen und Löschen einer Kante untersuchen wir, wie sich die Distanz von s zu x geändert hat.

Als erstes betrachten wir den Fall, dass wir eine Kante (v, u) einfügen. Gemäß dem zweiten Aufzählungspunkt bedeutet das, dass die Kante (u, v) auf dem Weg P liegen muss. In dem Algorithmus wurde der Weg P so gewählt, dass er ein kürzester Weg von s nach t ist. Dies bedeutet insbesondere, dass die Teilwege von P, die von s nach u und v führen, kürzeste Wege von s nach u bzw. v sind. Die Länge des Teilweges zu u bezeichnen wir mit ℓ. Dann ist die Länge des Teilweges nach v genau $\ell + 1$. Somit verbindet die neue Kante einen Knoten mit Distanz $\ell + 1$ von s mit einem Knoten mit Distanz ℓ von s. Dadurch kann sich jedoch die Distanz von s zu keinem Knoten des Graphen verringern. Um die Distanz zu verringern, müsste die neue Kante einen Knoten mit Distanz k von s für ein $k \geq 0$ mit einem Knoten mit Distanz $k+i$ von s für ein $i \geq 2$ verbinden. Damit folgt, dass selbst nach dem Einfügen aller neuen Kanten der Knoten x immer noch die gleiche Distanz von s hat.

Es ist leicht einzusehen, dass das Löschen von Kanten keine Distanzen verringern kann. Deshalb hat der Knoten x nach dem Löschen mindestens die gleiche Distanz von s wie in G_f. □

Mithilfe von Lemma 5.35 können wir nun das Theorem beweisen. Bei jeder Iteration der while-Schleife gibt es mindestens eine Flaschenhalskante (u, v). Diese wird aus dem Restnetzwerk entfernt. Sei ℓ die Distanz von s zu u, bevor die Kante entfernt wird. Die Distanz zu v ist dann $\ell + 1$. Es ist möglich, dass diese Kante zu einem späteren Zeitpunkt wieder in das Restnetzwerk eingefügt wird, aber nur dann, wenn die Kante (v, u) in einer späteren Iteration auf dem gewählten flussvergrößernden P' liegt. Da sich die Distanz von s zu v nicht verringern kann, muss zu diesem Zeitpunkt die Distanz von s nach v immer noch mindestens $\ell + 1$ sein. Da P' ein kürzester Weg ist, können wir aus Lemma 5.17 wieder folgern, dass die Distanz von s zu u zu diesem Zeitpunkt mindestens $\ell + 2$ sein muss.

Zusammenfassend können wir also festhalten, dass das Entfernen und Wiedereinfügen einer Kante (u, v) im Restnetzwerk die Distanz von s zu u um mindestens 2 erhöht. Da die maximale Distanz $n - 1$ ist, kann eine Kante demnach nicht öfter als $n/2$ mal entfernt werden. In jeder Iteration wird mindestens eine Kante entfernt und es gibt $2m$ potentielle Kanten im Restnetzwerk. Damit ist die Anzahl an Iterationen der while-Schleife nach oben beschränkt durch $\frac{n}{2} \cdot 2m = nm$. □

Kapitel 6

Algorithmische Geometrie

In vielen Anwendungen kommen die Eingabedaten aus einer geometrischen Umgebung, oder sie können geometrisch interpretiert werden. Ein einfaches Beispiel sind Koordinaten, die eine Position in der Ebene beschreiben, zum Beispiel in der Geographie. Ein anderes Beispiel sind mehrdimensionale Vektoren, welche die Eigenschaften eines Produktes beschreiben, wobei jeder Dimension eine festen Eigenschaft zugeordnet ist. Eigenschaften können zum Beispiel die Länge, Breite und Höhe der Verpackung oder der Preis sein. Eine solche Vektorbeschreibung wird zum Beispiel im Maschinellen Lernen oft verwendet. Triangulationen, sogenannte Dreiecksnetze, von Punktmengen kommen in vielen Anwendungen in der Computer Grafik vor. Sie werden zum Beispiel benutzt, um aus abgetasteten Punktwolken einer Oberfläche ein Drahtgittermodell (im Englischen auch *Mesh* genannt) zu konstruieren. In der Numerik werden sie benutzt, um Funktionen zu approximieren. Auch in der Ebene sind sie ein wichtiges Werkzeug, das es uns erlaubt, geometrische Zusammenhänge herzustellen. Die algorithmischen Probleme, die dadurch entstehen, werden in der Algorithmischen Geometrie untersucht. Wir werden uns in diesem Kapitel einer kleinen Auswahl von klassischen Problemen aus diesem Gebiet widmen, die sich mit Punkten in der Ebene befassen. Für weitergehendes Material sei auf die Literatur verwiesen [3].

In der Algorithmischen Geometrie verwenden wir ein abgewandeltes Modell der Registermaschine, die in einem Register statt einer ganzen Zahl eine reelle Zahl speichern kann. Wir nehmen weiterhin an, dass eine unbegrenzte Anzahl dieser Register zur Verfügung steht und auf diesen Registern grundlegende arithmetische Operationen durchgeführt werden können, sowie die üblichen Sprungoperationen und Kopieroperationen.

6.1 Delaunay Triangulation

Viele Anwendungen benutzen die Delaunay-Triangulation als Triangulation einer Menge von Punkten. Benannt ist sie nach dem russischen Mathematiker Boris Delaunay oder Delone (beides sind aus dem Russischen Делоне übertragene Schreibweisen). Wir

wollen in diesem Abschnitt zunächst grundlegende Eigenschaften von geometrischen Graphen und Triangulationen herleiten und uns dann damit beschäftigen, wie die Delaunay-Triangulation berechnet werden kann.

6.1.1 Geometrische Graphen

Wir betrachten ungerichtete ungewichtete Graphen, deren Knoten Punkte in der Ebene sind. Sei $V \subseteq \mathbb{R}^2$ eine Menge von n Punkten in der Ebene. In einem geometrischen Graphen $G = (V, E)$ betrachten wir für jedes Knotenpaar $(u, v) \in E$ die Strecke $\overline{uv} = \{(1-t)u + tv \mid t \in [0, 1]\}$ als die Kante zwischen den Knoten u und v. Wir sagen, dass zwei Kanten sich kreuzen, wenn sie einen Schnittpunkt haben, der nicht einer der Endpunkte ist. Um Spezialfälle zu vermeiden, nehmen wir an, dass keine drei Punkte auf einer gemeinsamen Geraden liegen. Der Graph G ist *kreuzungsfrei* wenn keine zwei Kanten seiner Kantenmenge sich kreuzen. Die Kanten eines kreuzungsfreien geometrischen Graphen unterteilen die Ebene in eine endliche Anzahl von Flächen. Der leere Graph hat nur eine Fläche, die Ebene.

Theorem 6.1. *Sei $G = (V, E)$ ein kreuzungsfreier geometrischer Graph mit v Knoten, e Kanten, f Flächen und c Zusammenhangskomponenten. Es gilt $v + f = e + c + 1$.*

Beweis. Wir zeigen das Theorem mithilfe struktureller Induktion. Die Induktion geht über die Menge der Knoten und Kanten des Graphen. Offenbar lässt sich jeder kreuzungsfreie geometrische Graph durch sequentielles Hinzufügen von einzelnen Knoten und Kanten erzeugen.

Für den Induktionsanfang beobachten wir, dass die Aussage auch für den leeren Graphen mit $v = 0, f = 1, e = 0, c = 0$ gilt.

Allgemein sei G also ein kreuzungsfreier geometrischer Graph mit v Knoten, e Kanten, f Flächen und c Zusammenhangskomponenten und sei die Aussage für G wahr. Im Induktionsschritt fügen wir entweder einen isolierten Knoten hinzu, oder wir verbinden zwei bestehende Knoten mit einer Kante. Sei G' der so entstandene Graph und seien v', e', f', c' die Anzahl der Knoten, Kanten, Flächen und Zusammenhangskomponenten.

Wir zeigen, dass die Aussage in beiden Fällen auch für G' gilt. Im ersten Fall fügen wir einen isolierten Knoten hinzu. Dadurch erhöht sich neben der Anzahl der Knoten auch die Anzahl der Zusammenhangskomponenten um eins, es gilt $v' = v + 1$, $c' = c + 1$, $e' = e$, $f' = f$. Im zweiten Fall fügen wir eine Kante hinzu, es gilt also $v' = v$ und $e' = e + 1$. Hier unterscheiden wir zwei Unterfälle: (a) die neue Kante schließt einen Kreis, (b) zwei Zusammenhangskomponenten von G werden durch die neue Kante zu einer Zuammenhangskomponente verbunden. Im Fall (a) gilt $f' = f + 1$, $c' = c$. Im Fall (b) gilt $f' = f$, $c' = c - 1$. □

Theorem 6.1 ist auch unter dem Namen *Eulersche Polyederformel* bekannt, weil sich mit ihrer Hilfe Eigenschaften von Polyedern herleiten lassen, die hier aber nicht weiter wichtig sind. Wir können damit eine wichtige Eigenschaft kreuzungsfreier geometrischer Graphen zeigen, die wir wie folgt formulieren.

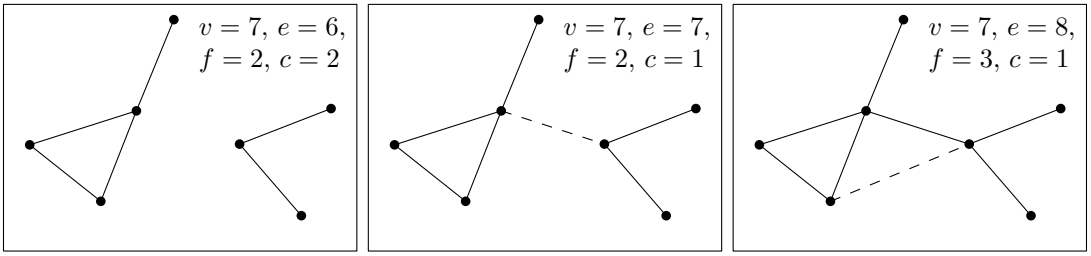


Abbildung 6.1: Drei Graphen, die zwei verschiedene Fälle im Beweis von Theorem 6.1 illustrieren. Die neu hinzugefügte Kante ist jeweils gestrichelt. Links: Der Graph hat zwei Flächen und zwei Zusammenhangskomponenten. Mitte: Anzahl der Komponenten wird um eins verringert. Rechts: Anzahl der Flächen wird um eins erhöht.

Theorem 6.2. *Ein kreuzungsfreier geometrischer Graph in der Ebene mit n Knoten hat $O(n)$ Kanten und Flächen.*

Beweis. Wir zeigen das Theorem für zusammenhängende Graphen. Da die Knotenmengen von verschiedenen Zusammenhangskomponenten des Graphen disjunkt sind und keine Kanten zwischen den Zusammenhangskomponenten verlaufen, folgt die Aussage auch für Graphen die nicht zusammenhängend sind.

Sei G ein kreuzungsfreier zusammenhängenden Graphen G mit v Knoten, e Kanten und f Flächen. Im Folgenden nehmen wir an, dass $e \geq 3$. Sei d_i die Anzahl der Kanten, die inzident zur i-ten Fläche sind. Wobei die Flächen beliebig nummeriert seien. Wir können die Summe der d_i über alle Flächen auf zwei verschiedene Arten abschätzen. Zum einen ist jede Kante zu höchstens zwei Flächen inzident, zum anderen ist jede Fläche zu mindestens drei Kanten inzident, also ist

$$3f \;\leq\; \sum_{i=1}^{f} d_i \;\leq\; 2e$$

Daraus folgt $f \leq \frac{2}{3}e$. Gleichzeitig wissen wir aus Theorem 6.1 mit $c = 1$, dass $v - e + f = 2$. Setzen wir obige Ungleichung ein, dann erhalten wir

$$e = f + v - 2 \leq v + \frac{2}{3}e - 2$$

Dies können wir umformen zu

$$e \leq 3(v - 2)$$

Also ist $e \in O(n)$. Aus $f \leq \frac{2}{3}e$ folgt nun auch $f \in O(n)$. □

Definition 6.3 (Triangulation)**.** *Eine Triangulation einer Menge von Punkten S in der Ebene ist ein kreuzungsfreier geometrischer Graph mit Knotenmenge S und mit inklusions-maximaler Kantenmenge E. Das heißt, wir können keine Kante zu E hinzufügen ohne dass eine Kreuzung entsteht.*

Jede Triangulation einer Punktmenge S teilt die Ebene in Dreiecke und in einen unbeschränkten Bereich, welchen wir die äußere Fläche nennen. Im Folgenden treffen

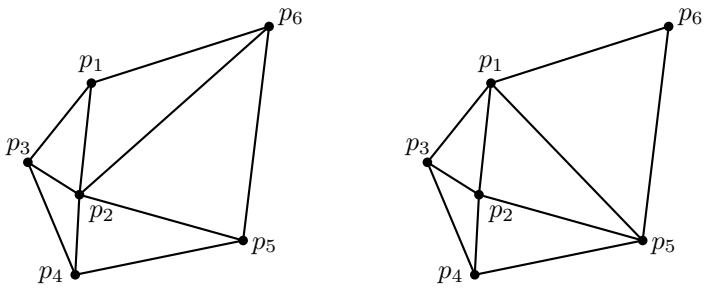


Abbildung 6.2: Zwei verschiedene Triangulationen derselben Punktmenge.

wir zwei wichtige Annahmen. Keine vier Punkte aus S liegen auf einem gemeinsamen Kreis. Keine drei Punkte aus S liegen auf einer gemeinsamen Geraden. Wir sagen über eine Punktmenge S, die diese zwei Annahmen erfüllt, dass sie in *allgemeiner Lage* liegt.

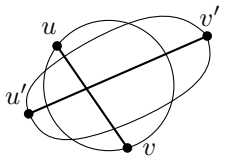
Definition 6.4 (Delaunay-Triangulation)**.** *Eine Triangulation einer Menge von Punkten S in der Ebene ist eine Delaunay-Triangulation wenn für jedes Dreieck der Triangulation gilt, dass sein Umkreis keine Punkte aus S in seinem Inneren enthält.*

Wir wollen nun zeigen, dass die Delaunay-Triangulation von Punkten in allgemeiner Lage eindeutig ist. Die folgende Charakterisierung der Kanten der Delaunay-Triangulation hilft uns dabei.

Definition 6.5 (Delaunay-Kante)**.** *Eine Kante (p, q) zwischen zwei Punkten der Menge S ist eine Delaunay-Kante genau dann wenn es einen Kreis gibt, der die zwei Endpunkte p und q auf dem Rand hat und keine weiteren Punkte von S in seinem Inneren enthält.*

Lemma 6.6. *Sei S eine Menge von Punkten in der Ebene in allgemeiner Lage und seien $e = \{u, v\}$ und $e' = \{u', v'\}$ zwei verschiedene Delaunay-Kanten von S, dann können sich e und e' nicht kreuzen.*

Beweis. Beweis durch Widerspruch. Angenommen, die zwei Kanten kreuzen sich. Da beides Delaunay-Kanten sind, existiert für jede der beiden Kanten ein Kreis, der keine der beiden Endpunkte der anderen Kante in seinem Inneren enthält.



Dies ist aber nicht für beide Kanten möglich, wenn sich die beiden Kanten kreuzen, denn dann müssten sich die zwei Kreise an mehr als zwei verschiedenen Punkten schneiden. Kreise können sich aber an maximal zwei verschiedenen Punkten schneiden. □

Theorem 6.7. *Die Menge der Kanten der Delaunay-Triangulation einer Menge S von Punkten in allgemeiner Lage ist genau die Menge der Delaunay-Kanten von S.*

Beweis. Wir zeigen Inklusion der Mengen in beide Richtungen. Für jedes Dreieck der Delaunay-Triangulation nach Definition 6.4 gilt, dass sein Umkreis keinen weiteren Punkt enthält. Daher ist jede der Kanten der Delaunay-Triangulation auch eine Delaunay-Kante nach Definition 6.5. Es bleibt, die Inklusion in die andere Richtung zu zeigen. Diesen Teil der Aussage zeigen wir mit einem Widerspruchsbeweis. Angenommmen, es existiert eine Delaunay-Kante e, die nicht in der Delaunay-Triangulation enthalten ist. Da die Delaunay-Triangulation eine Triangulation ist, ist ihre Kantenmenge nach Definition 6.3 eine maximale Menge kreuzungsfreier Kanten. Da e nicht in der Kantenmenge enthalten sei, muss e also eine Kante e' der Delaunay-Triangulation kreuzen, sonst wäre die Kantenmenge nicht maximal. Nach Lemma 6.6 ist dies nicht möglich, denn e und e' sind beides Delaunay-Kanten. □

Theorem 6.7 zeigt, dass die Delaunay-Triangulation für Punkte in allgemeiner Lage eindeutig ist. Die dadurch hergeleitete Charakterisierung der Delaunay-Kanten werden wir im Folgenden wiederholt für die Entwicklung von effizienten Algorithmen verwenden. Die Eindeutigkeit der Delaunay-Triangulation wird auch noch im nächsten Abschnitt über Voronoi-Diagramme wichtig sein.

6.1.2 Greedy-Algorithmus von Fortune

In diesem Abschnitt werden wir einen einfachen Algorithmus kennenlernen, der aus einer beliebigen Triangulation die Delaunay-Triangulation berechnen kann. Wir betrachten dazu die Operation des Diagonalenwechsels, aus dem Englischen hergeleitet auch Flip genannt, die wir wie folgt definieren.

Definition 6.8 (Flip)**.** *Angenommen, (a, b, c) und (a, c, d) sind benachbarte Dreiecke einer Triangulation T. Wir bezeichnen die Kanten (a, c) und (b, d) als Diagonalen des Vierecks (a, b, c, d) sofern sie innerhalb des Vierecks verlaufen. Ein Flip ersetzt (a, c) durch (b, d) in T. Der Flips ist zulässig, wenn beide Kanten Diagonalen sind. In diesem Fall nennen wir (b, d) die Flip-Diagonale von (a, c) in T.*

Definition 6.9 (Delaunay-Flip)**.** *Sei (a, c) die gemeinsame Kante zweier Dreiecke (a, b, c) und (a, c, d) einer Triangulation, sodass der Flip von (a, c) zu (b, d) zulässig ist. Ist d im Umkreis von (a, b, c) enthalten, so sprechen wir von einem Delaunay-Flip.*

Wir werden zeigen, dass durch einen Delaunay-Flip, die Delaunay-Eigenschaft lokal hergestellt wird, dass also bezüglich der Eckpunkte der beiden involvierten Dreiecke eine Delaunay-Triangulation entsteht. Allerdings kann die Delaunay-Eigenschaft bezüglich der Gesamtmenge der Punkte noch immer verletzt sein. Es ist sogar möglich, dass keine der beiden betrachteten Diagonalen eine Delaunay-Kante der Gesamtmenge der Punkte ist. Dennoch werden wir sehen, dass man immer mithilfe einer endlichen Abfolge von Delaunay-Flips die globale Delaunay-Eigenschaft herstellen kann. Wir werden sogar sehen, dass es egal ist, in welcher Reihenfolge die Delaunay-Flips durchführt werden. Die Analyse des folgenden Greedy-Algorithmus geht zurück auf Fortune.

GreedyFlips(**Triangulation** T)

```
1  while (∃ Kante e in T, die einen Delaunay-Flip erlaubt) {
2      Ersetze e durch die entsprechende Flip-Diagonale in T.
3  }
```

Uns geht es zunächst nicht darum zu zeigen, wie man den Greedy-Algorithmus effizient implementieren kann, sondern zunächst geht es nur darum zu zeigen, dass der Algorithmus überhaupt terminiert und damit das korrekte Ergebnis liefert. Für die Analyse des Algorithmus führen wir das Konzept von Konflikten ein. Intuitiv löst ein Delaunay-Flip einen Konflikt auf und wir wollen zeigen, dass sich die Anzahl der Konflikte mit jedem Delaunay-Flip verringert.

Definition 6.10 (Konfliktfunktion)**.** *Sei $C(a,b,c)$ der Umkreis des Dreiecks (a,b,c). Sei T eine Triangulation von S. Wir definieren die Funktion Φ auf der Menge der Triangulationen von S.*

$$\Phi(T) = \sum_{(a,b,c)\ Dreieck\ in\ T} \Big| \{\ p \in S\ \mid\ p \in C(a,b,c)\ \} \Big|$$

Wenn ein Punkt aus S im Umkreis eines Dreiecks von T enthalten ist, dann sagen wir dieser Punkt ist mit dem Dreieck im Konflikt. $\Phi(T)$ ist die Anzahl der Konflikte in T.

Desweiteren verwenden wir die folgende geometrische Eigenschaft der Umkreise der Dreiecke, die in einen Delaunay-Flip involviert sind. Der Beweis dieses Lemmas ist nicht schwierig, aber auch nicht besonders interessant. Daher werden wir ihn hier nicht besprechen.

Lemma 6.11. *Sei $C(a,b,c)$ der Umkreis des Dreiecks (a,b,c). Seien a,b,c,d gegeben wie in Definition 6.8. Es gilt*

(i) $C(a,b,d) \cup C(b,c,d) \subseteq C(a,b,c) \cup C(a,c,d)$

(ii) $C(a,b,d) \cap C(b,c,d) \subseteq C(a,b,c) \cap C(a,c,d)$

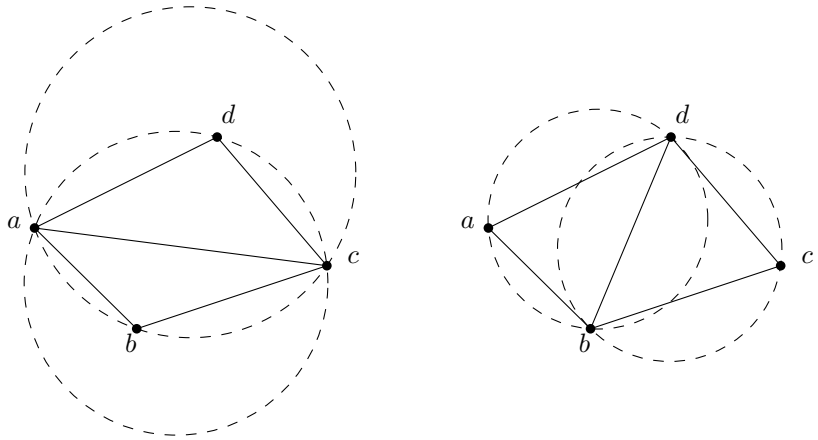


Abbildung 6.3: Delaunay-Flip im Viereck (a,b,c,d).

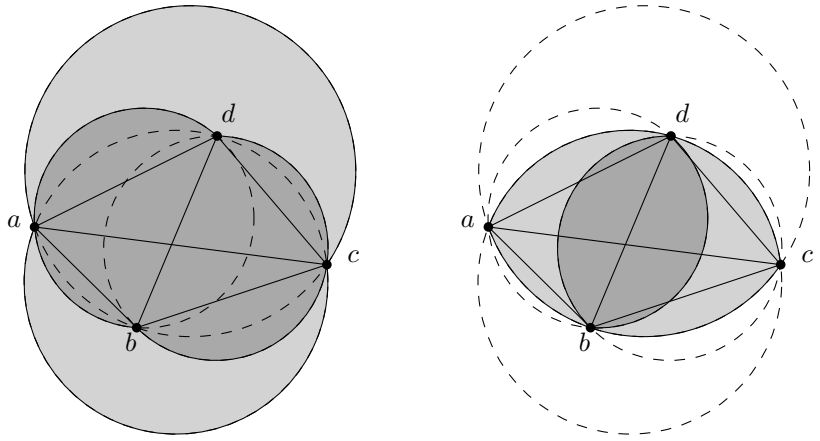


Abbildung 6.4: Illustration zu den zwei Fällen in Lemma 6.11.

Aus dem zweiten Teil des Lemmas folgt direkt, dass bei einem Delaunay-Flip initial auch b mit dem Dreieck (a, c, d) im Konflikt sein muss, da b im Schnitt der beiden Umkreise enthalten sein muss. Es werden durch den Delaunay-Flip also immerhin zwei Konflikte aufgelöst. Es ist allerdings zunächst nicht offensichtlich, dass durch den Delaunay-Flip nicht auch neue Konflikte entstehen können.

Lemma 6.12. *Sei T' eine Triangulation, die man aus T durch einen Delaunay-Flip erhält. Dann gilt $\Phi(T') \leq \Phi(T) - 2$.*

Beweis. Aus dem ersten Teil von Lemma 6.11 folgt, dass ein Punkt aus S, der mit einem der neuen Dreiecke (a, b, d) oder (b, c, d) im Konflikt ist, auch vorher im Konflikt mit einem der Dreiecke (a, b, c) oder (a, c, d), die beide entfernt wurden, war. Aus dem zweiten Teil des Lemmas folgt, dass ein Punkt, der mit *beiden* neuen Dreiecken in Konflikt ist, auch mit beiden Dreiecken, die entfernt wurden, im Konflikt war. Alle anderen Konflikte, mit Dreiecken die in beiden Triangulationen vorkommen, bleiben unberührt. Die Anzahl der Konflikte verringert sich um mindestens zwei, da der Konflikt der Dreiecke (a, b, c) und (a, c, d) mit dem Punkt d, beziehungsweise b, aufgelöst wird. □

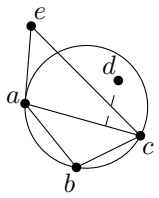
Der Greedy-Algorithmus terminiert, wenn in einem der Schritte $\Phi(T) \leq 0$ gilt. In diesem Fall gibt es keine Konflikte mehr und T ist die Delaunay-Triangulation. Wir zeigen als nächstes, dass es immer möglich ist einen zulässigen Delaunay-Flip zu finden, solange es noch Konflikte in der Triangulation gibt. Es bleibt dann noch zu zeigen, dass der Algorithmus in einer endlichen Anzahl von Schritten terminiert.

Lemma 6.13. *Sei T eine Triangulation einer Punktmenge S mit $\Phi(T) > 0$, dann gibt es zwei benachbarte Dreiecke in T, die einen Delaunay-Flip erlauben.*

Beweis. Sei d ein Punkt aus S und (a, b, c) ein Dreieck das mit d in Konflikt steht und sei das Dreieck so gewählt, dass der minimale Abstand von d zu einem Punkt im Dreieck minimiert wird. Ohne Beschränkung der Allgemeinheit sei (a, c) die Kante des Dreiecks, die den Abstand zu d minimiert. Angenommen, das Dreieck (a, c, d) ist

nicht in der Triangulation T enthalten. Dann muss die Kante (a, c) mit einem anderen Punkt e ein Dreieck in T bilden.

Falls e nicht im Umkreis von (a, b, c) liegt, dann liegt d im Umkreis von (a, c, e), ist also im Konflikt mit (a, c, e).



Dann wäre aber der Abstand von d zu (a, b, c) nicht minimal, da die Strecke von d zu dem Punkt mit kleinstem Abstand das Dreieck (a, c, e) kreuzen würde und somit der Abstand zu (a, c, e) kleiner wäre. Es muss also so sein, dass e im Umkreis von (a, b, c) liegt. Somit gibt es einen Delaunay-Flip im Viereck (a, b, c, e). □

Als nächstes wollen wir zeigen, dass der Algorithmus terminiert. Mithilfe von Lemma 6.11 werden wir zeigen, dass die Anzahl der Delaunay-Flips, die der Algorithmus im schlimmsten Fall durchführt, durch die Anzahl der Konflikte der initialen Triangulation beschränkt ist.

Lemma 6.14. *Sei T eine Triangulation einer Menge S von n Punkten. Der Greedy-Algorithmus führt maximal $O(n^2)$ Delaunay-Flips durch.*

Beweis. Aus Theorem 6.2 folgt, dass die Anzahl der Dreiecke in T in $O(n)$ ist. Daher kann es höchstens $O(n^2)$ viele Konflikte geben, da jedes Paar von Dreieck und Punkt nur einen Konflikt erzeugen kann. Aus Lemma 6.11 folgt, dass sich die Anzahl der Konflikte mit jedem Diagonalenwechsel um mindestens zwei verringert. □

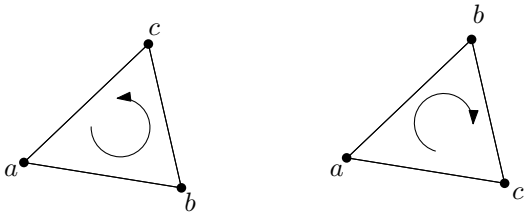
6.1.3 Halbkanten-Datenstruktur

Bevor wir besprechen, wie eine Triangulation so im Computer repräsentiert werden kann, dass Delaunay-Flips effizient erkannt und durchgeführt werden können, wollen wir kurz auf grundlegende geometrische Operationen eingehen. In der Praxis wollen wir numerische Probleme vermeiden, die bei Berechnungen mit reellen Zahlen entstehen können. Deshalb verwenden wir nur möglichst einfache arithmetische Operationen $(+, -, *, /)$ und vermeiden trigonometrische Funktionen $(\sin, \cos)$.

Orientierung eines Dreiecks: Gegeben drei Punkte $a, b, c \in \mathbb{R}^2$, ist das Tupel (a, b, c) gegen den Uhrzeigersinn orientiert? Hierfür können wir die Formel der Dreiecksfläche mit Vorzeichen verwenden. Für Punkte $a = (a_1, a_2), b = (b_1, b_2), c = (c_1, c_2)$ die gegen den Uhrzeigersinn orientiert sind, ist der (positive) Flächeninhalt des Dreiecks bestimmt durch

$$\frac{1}{2} \det \begin{pmatrix} a_1 & a_2 & 1 \\ b_1 & b_2 & 1 \\ c_1 & c_2 & 1 \end{pmatrix} = \frac{(c_1 - a_1)(c_2 + a_2) + (b_1 - c_1)(b_2 + c_2) + (a_1 - b_1)(a_2 + b_2)}{2}$$

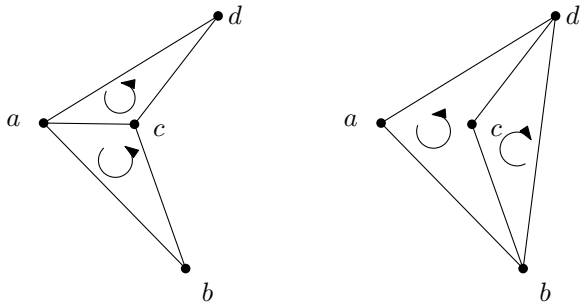
Falls das Tupel (a, b, c) nicht gegen den Uhrzeigersinn orientiert ist, dann ergibt obige Formel eine negative Zahl. Falls die drei Punkte auf einer gemeinsamen Geraden liegen, dann ergibt die obige Formel null.



Punkt links von einer Geraden: Gegeben drei Punkte $a, b, c \in \mathbb{R}^2$, liegt c links von der Geraden durch a und b, die in Richtung von a nach b orientiert ist? Das ist der Fall genau dann wenn das Tupel (a, b, c) gegen den Uhrzeigersinn orientiert ist.

Punkt im Dreieck: Gegeben $q \in \mathbb{R}^2$ und drei Punkte $a, b, c \in \mathbb{R}^2$, ist q im Dreieck (a, b, c) enthalten? Wir nehmen an, dass das Tupel (a, b, c) gegen den Uhrzeigersinn orientiert ist. Der Punkt q ist genau dann im Dreieck enthalten, wenn alle drei Tupel (a, b, q), (b, c, q), und (c, a, q) gegen den Uhrzeigersinn orientiert sind. Wir können uns vorstellen, dass wir die Dreieckskanten entgegen dem Uhrzeigersinn ablaufen, wenn der Punkt q immer links von der aktuellen Kante liegt, dann ist er im Dreieck enthalten.

Flip zulässig: Angenommen, wir wollen einen Flip einer Diagonalen (a, c) in einem Viereck (a, b, c, d) durchführen. Um zu testen, ob die Kante (b, d) innerhalb des Vierecks (a, b, c, d) verläuft, testen wir ob die beiden neuen Dreiecke (b, c, d) und (a, b, d) jeweils entgegen dem Uhrzeigersinn orientiert sind. Wenn dies für beide der Fall ist, dann ist der Flip zulässig. Alternativ könnte man auch die Innenwinkel des Vierecks an den Eckpunkten a und c bestimmen. Der Flip ist zulässig genau dann wenn beide Innenwinkel kleiner als 180° sind. Je nach Implementierung kann dies allerdings leichter zu numerischen Problem führen.



Obiges Beispiel zeigt ein Viereck in dem der Flip von (a, c) zu (b, d) nicht zulässig ist, da die Kante (b, d) nicht innerhalb des Vierecks verläuft. Das Dreieck (a, b, d) ist gegen den Uhrzeigersinn orientiert, aber das Dreieck (b, c, d) ist im Uhrzeigersinn orientiert.

Punkt im Kreis: Gegeben ein Punkt $q = (q_1, q_2) \in \mathbb{R}^2$ und ein Kreis C mit Mittelpunkt $p = (p_1, p_2)$ und Radius r, ist q im Inneren von C enthalten? Das ist der Fall genau dann, wenn gilt

$$(p_1 - q_1)^2 + (p_2 - q_2)^2 < r^2.$$

Punkt im Umkreis eines Dreiecks: Gegeben ein Punkt $q \in \mathbb{R}^2$ und drei Punkte $a, b, c \in \mathbb{R}^2$, ist q im Umkreis von (a, b, c) enthalten? Die kartesischen Koordinaten des Umkreismittelpunktes und der (quadratische) Radius des Umkreises lassen sich aus den Koordinaten der Punkte a, b, c herleiten.

Als nächsten werden wir eine Datenstruktur für Triangulationen kennenlernen, die sogenannte *Halbkanten-Datenstruktur*. Die Datenstruktur speichert ein Objekt für jeden Knoten der Triangulation und zwei Objekte für jede Kante – sogenannte Halbkanten, die jeweils eine Richtung repräsentieren. Die Objekte der Datenstruktur werden durch Zeiger miteinander verknüpft. Eine Halbkante $\overrightarrow{uv}$ welche vom Knoten u zum Knoten v verläuft, wird durch einen Zeiger mit der Halbkante $\overrightarrow{vu}$ verknüpft, welche dieselbe Triangulationskante in umgekehrter Richtung repräsentiert. In diesem Kontext sprechen wir von $\overrightarrow{uv}$ und $\overrightarrow{vu}$ als Zwillingskanten.

Die Flächen der Triangulation werden nicht explizit gespeichert. Jede Halbkante ist implizit mit der Fläche assoziiert, die zu ihrer linken Seite liegt wenn man entlang der Richtung der Halbkante schaut. Die Halbkanten, die auf diese Weise mit einer Fläche assoziiert sind, sind mithilfe von Zeigern zyklisch verkettet. Die Zeiger und Objekte lassen sich wie folgt zusammenfassen.

Objekte und Zeiger der Halbkanten-Datenstruktur

- **Halfedge**

  start: Jede Halbkante speichert einen Zeiger auf ihren Startknoten.

  twin: Die zwei Halbkanten, die eine feste Kante jeweils in eine der beiden Richtungen repräsentieren, speichern jeweils einen Zeiger aufeinander.

  next: Jede Halbkante speichert einen Zeiger auf ihren Nachfolger entlang des Randes der Fläche, die zu ihrer Linken liegt. Durch diese Relation sind Halbkanten, die einem Dreieck zugeordnet sind, zyklisch gegen den Uhrzeigersinn entlang des Randes der Fläche geordnet. Halbkanten, die an der äußeren Fläche liegen, sind dagegen im Uhrzeigersinn geordnet.

  outer: Jede Halbkante speichert eine boolesche Variable, die angibt, ob die Fläche zu ihrer Linken die äußeren Fläche der Triangulation ist

- **Node**

  coords: Jeder Knoten speichert seine eigenen Koordinaten.

  edge: Jeder Knoten speichert einen Zeiger auf *eine* der Halbkanten, welche diesen Knoten als Startknoten haben.

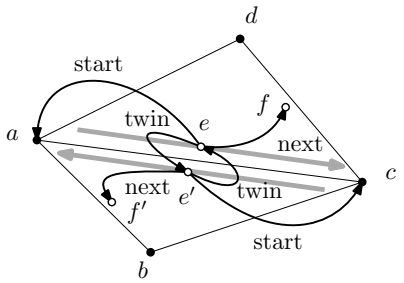
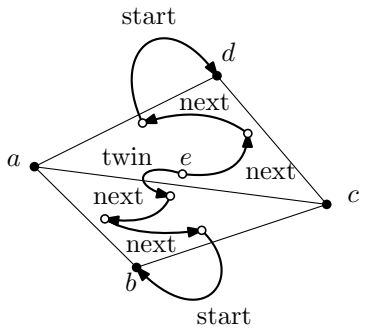


Abbildung 6.5: Zwillingskanten $e = \overrightarrow{ac}$ und $e' = \overrightarrow{ca}$ und ihre Zeiger auf andere Objekte der Datenstruktur: Halbkante f ist die nächste Kante nach e im oberen Dreieck. Halbkante f' ist die nächste Kante nach e' im unteren Dreieck.

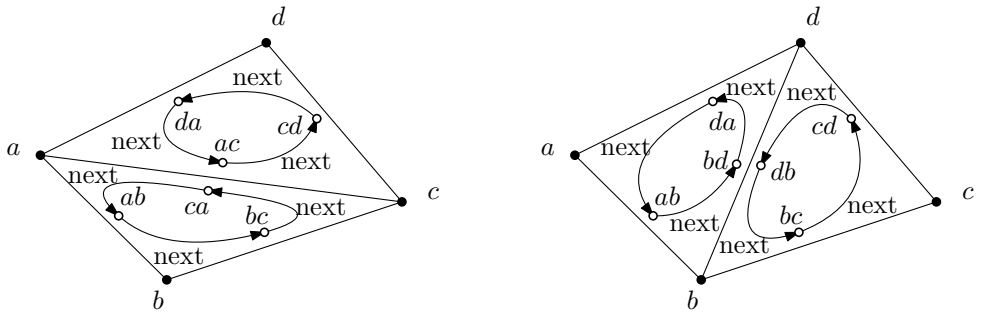
Für eine Triangulation, die in einer Halbkanten-Datenstruktur gespeichert ist, können wir nun einen Algorithmus angeben, der testet, ob eine Kante einen Delaunay-Flip erlaubt. Der folgende Pseudocode tut genau das. Die Laufzeit ist in $O(1)$, wobei ein Zeiger auf die Kante, die getestet werden soll, als Argument übergeben wird.

```
IsLocallyDelaunay(Halfedge e)
1  if(e.twin.outer or e.outer){ return True;}
2  a = e.start;
3  c = e.next.start;
4  d = e.next.next.start;
5  b = e.twin.next.next.start;
6  return (d ∉ C(a,b,c) and b ∉ C(a,c,d));
```



Als nächstes wollen wir einen Algorithmus angeben, der den Diagonalenwechsel in der Halbkanten-Datenstruktur in konstanter Zeit durchführt. Der Algorithmus bekommt einen Zeiger auf die Diagonale, die gewechselt werden soll, als Argument übergeben.

```
Flip(Halfedge ac)
1  b = ac.twin.next.next.start;
2  d = ac.next.next.start;
3  Initialisiere Zwillingskanten bd und db mit Knoten b und d.
4  //Aktualisiere next-Zeiger der Halbkanten
5  cd = ac.next; ab = ac.twin.next;
6  da = cd.next; bc = ab.next;
7  bd.next = da; da.next = ab; ab.next = bd;
8  db.next = bc; bc.next = cd; cd.next = db;
9  //Aktualisiere Zeiger auf ausgehende Kante
10 a.edge=ab; c.edge=cd;
11 return bd;
```



Der Greedy-Algorithmus lässt sich nun mithilfe dieser beiden Algorithmen IsLocallyDelaunay und Flip realisieren. Dafür nehmen wir an, dass die Triangulation T in einer Halbkanten-Datenstruktur gegeben ist, mit einem Zeiger auf eine Kante e auf der äußeren Fläche. In jedem Durchlauf der while-Schleife, führen wir, ausgehend vom Startknoten von e, eine Breitensuche auf dem Graph der Triangulation aus (siehe Übung). Für jede von der Breitensuche besuchten Kante, wird IsLocallyDelaunay aufgerufen, um zu entscheiden, ob die zwei benachbarten Dreiecke einen Delaunay-Flip erlauben. Wenn IsLocallyDelaunay False zurück gibt, dann wird Flip auf dieser Kante aufgerufen und ein neuer Schleifendurchlauf mit einer erneuten Breitensuche gestartet. Wenn ein Viereck nur eine zulässige Diagonale hat, dann gibt der Algorithmus IsLocallyDelaunay auf dieser Diagonalen True zurück. Der Algorithmus benötigt insgesamt $O(n^3)$ Schritte, da wir im schlimmsten Fall $O(n)$ Zeit benötigen um mithilfe der Breitensuche einen Delaunay-Flip zu finden.

Lemma 6.15. *Sei eine Triangulation T mit n Punkten in einer Halbkanten-Datenstruktur und ein Zeiger auf eine Kante e auf der äußeren Fläche gegeben. Die Laufzeit des Algorithmus GreedyFlips ist in $O(n^3)$.*

6.1.4 Inkrementeller Algorithmus

In diesem Abschnitt werden wir einen Algorithmus analysieren, der die Delaunay-Triangulation einer gegebenen Menge von n Punkten in der Ebene in $O(n^2)$ Zeit berechnet. Der Algorithmus fügt die Punkte nacheinander zu der Triangulation hinzu und stellt in jedem Schritt die globale Delaunay-Eigenschaft der Triangulation wieder her. Dabei wird ausgenutzt, dass die einzigen Konflikte, die der Algorithmus auflösen muss, mit dem neu eingefügten Punkt zusammenhängen. Diese Eigenschaft wollen wir zunächst genauer analysieren. Dafür beweisen wir das folgende Lemma.

Lemma 6.16 (Stern-Lemma)**.** *Seien $p_1, \ldots, p_n$ Punkte in der Ebene in allgemeiner Lage und sei DT_i die Delaunay-Triangulation der Punkte $p_1, \ldots, p_i$. Betrachte ein $i > 3$.*

(i) Jedes Dreieck, das in DT_i enthalten ist, aber nicht in DT_{i-1}, muss p_i als einen seiner Eckpunkte haben.

(ii) Jedes Dreieck, das in DT_{i-1} enthalten ist, aber nicht in DT_i, muss mit p_i im Konflikt stehen.

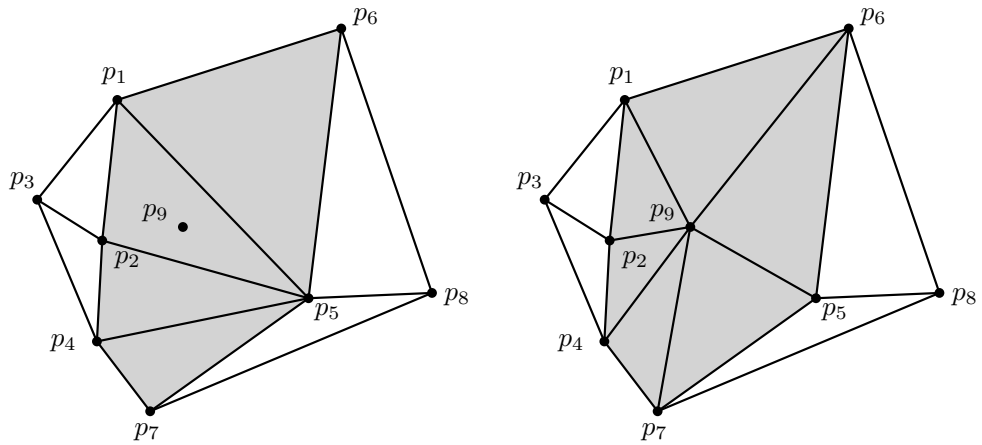


Abbildung 6.6: Illustration zu Lemma 6.16. Links: Delaunay-Triangulation der Punkte $p_1, \ldots, p_8$, Dreiecke, die entfernt werden müssen, sind grau unterlegt. Die grauen Dreiecke sind mit p_9 im Konflikt. Rechts: Delaunay-Triangulation der Punkte $p_1, \ldots, p_9$. Dreiecke, die neu eingefügt werden müssen, sind grau unterlegt. Die grauen Dreiecke sind inzident zu p_9.

Beweis. Wir zeigen beide Aussagen durch Umkehrschluss mithilfe von Theorem 6.7. Betrachte ein Dreieck D in DT_i, welches nur Punkte aus $p_1, \ldots, p_{i-1}$ als Eckpunkte hat. Das Dreieck ist mit keinem Punkt aus $p_1, \ldots, p_i$ im Konflikt. Damit sind seine Kanten Delaunay-Kanten bezüglich der Menge $p_1, \ldots, p_{i-1}$. Somit muss D ein Dreieck in DT_{i-1} sein. Damit ist (i) bewiesen. Der zweite Teil des Lemmas folgt mit einem ähnlichen Argument. Wenn ein Dreieck aus DT_{i-1} nicht mit p_i im Konflikt steht, dann steht es mit keinem Punkt aus der Menge $p_1, \ldots, p_i$ im Konflikt und muss somit in DT_i enthalten sein. □

Aus Lemma 6.16(i) folgt, dass beim Übergang von DT_{i-1} zu DT_i alle neuen Dreiecke zu dem neuen Punkt p_i inzident sein müssen. Als nächstes wollen wir einen Algorithmus angeben, der zur Triangulation DT_{i-1} den Punkt p_i hinzufügt, und die Delaunay-Eigenschaft wiederherstellt, sodass daraus die Delaunay-Triangulation DT_i entsteht. Der Algorithmus berechnet in dieser Weise nacheinander die Triangulationen $DT_3, \ldots, DT_n$. Der Pseudocode des Algorithmus ist wie folgt.

```
DelaunayTriangulation(S)

1  Seien p_1, ..., p_n die Punkte in S.
2  Initialisiere T als Triangulation mit Dreieck (p_1, p_2, p_3).
3  for (i = 4 ; i ≤ n ; i++){
4      Füge p_i in T ein und ergänze Kantenmenge zu einer Triangulation.
5      foreach(Halfedge e bildet mit p_i ein Dreieck in T){
6          RestoreDelaunay(e);
7      }
8  }
```

Der inkrementelle Algorithmus benutzt die rekursive Funktion RestoreDelaunay, welche auf Kanten der ursprünglichen Fläche aufgerufen wird, in der der neue Punkt p_i eingefügt wurde. Diese Kanten bilden nach dem Einfügen ein neues Dreieck mit p_i. Die Funktion RestoreDelaunay testet, für eine gegebene Kante, ob diese einen Delaunay-Flip erlaubt. Wenn ja, dann wird die Kante durch ihre Flip-Diagonale ersetzt. Dadurch entstehen zwei neue Dreiecke, die beide zu p_i inzident sind. Die Funktion ruft sich dann rekursiv jeweils auf der Kante des neuen Dreiecks auf, die gegenüberliegend zu p_i liegt. Der Pseudocode ist wie folgt. Eine Illustration ist in Abbildung 6.7 dargestellt. Abbildung 6.8 zeigt die Sequenz von Delaunay-Flips, die vom Algorithmus durchgeführt werden, anhand eines Beispiels.

```
RestoreDelaunay(Halfedge e)

1   if(not IsLocallyDelaunay(e)){
2       Halfedge g = Flip(e);
3       RestoreDelaunay(g.next.next);
4       RestoreDelaunay(g.twin.next);
5   }
```

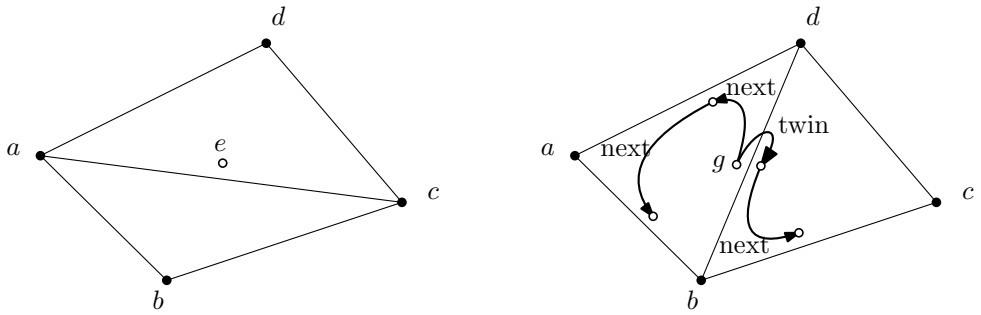


Abbildung 6.7: Illustration zu RestoreDelaunay. Falls die Kante e einen Delaunay-Flip erlaubt, wird sie durch ihre Flip-Diagonale g ersetzt. Die Funktion ruft sich dann rekursiv auf den beiden Kanten auf, welche die neuen Dreiecke mit d bilden.

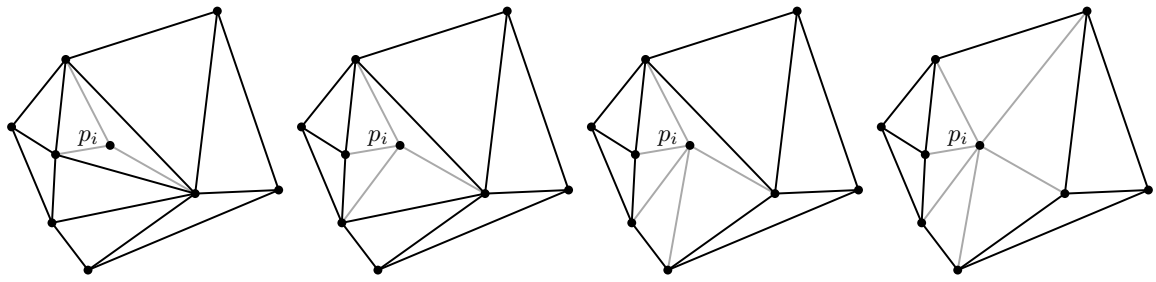


Abbildung 6.8: Sequenz von Delaunay-Flips nach Einfügen von Punkt p_i.

Lemma 6.17. *Betrachte einen Durchlauf der for-Schleife von* DelaunayTriangulation*, in dem der Punkt p_i hinzugefügt wird.* RestoreDelaunay *wird rekursiv auf genau den Kanten e aufgerufen, die zu einem Zeitpunkt ein Dreieck mit p_i in der aktuellen Triangulation bilden.*

Beweis. Wir zeigen die Behauptung mithilfe Induktion über die Rekursionstiefe. Am Anfang wird RestoreDelaunay vom Algorithmus DelaunayTriangulation aus aufgerufen. Hier ist die Behauptung trivial erfüllt, da die foreach-Schleife genau über die Halbkanten läuft, die mit p_i ein Dreieck bilden. Betrachten wir einen beliebigen rekursiven Aufruf von RestoreDelaunay, der von diesem initialen Aufruf herrührt, so können wir per Induktionvoraussetzung annehmen, dass die Kante e mit p_i ein Dreieck in der aktuellen Triangulation bildet. Falls ein Flip durchgeführt wird, wird die Kante e durch die Kante g ausgetauscht. Danach bildet p_i jeweils ein Dreieck mit beiden Kanten, mit denen RestoreDelaunay rekursiv aufgerufen wird. □

Lemma 6.18. *Betrachte einen Durchlauf der for-Schleife in* DelaunayTriangulation*, in dem der Punkt p_i hinzugefügt wurde. Wenn am Anfang des Schleifendurchlaufs T die Delaunay-Triangulation von $p_1, \ldots, p_{i-1}$ war, dann ist am Ende des Schleifendurchlaufs kein Dreieck von T mit p_i im Konflikt.*

Beweis. Sei T die Triangulation am Ende des Durchlaufs der for-Scheife. Betrachte einen Kreis w aus den Kanten die mit p_i in T ein Dreieck bilden. Aus Lemma 6.17 folgt, dass IsLocallyDelaunay auf jeder Kante e von w aufgerufen wurde und True zurückgegeben hat, sonst wäre e durch einen Flip ersetzt worden. Die anliegenden Dreiecke der Triangulation, die eine Kante auf dem Kreis w haben, sind also nicht im Konflikt mit p_i. Aus Lemma 6.16 (ii) folgt somit, dass w in DT_i enthalten ist, also eine Delaunay-Kante ist.

Angenommen es gibt ein Dreieck (a, b, c) in T das mit p_i im Konflikt steht, dann müsste mindestens eine Kante $(p_i, a), (p_i, b)$ oder (p_i, c) in DT_i sein, da (a, b, c) kein Dreieck in DT_i sein kann und laut Lemma 6.16 (i) alle neuen Dreiecke in DT_i zu p_i inzident sind. Diese Kante müsste w kreuzen, da a, b, c ausserhalb von w liegen. Dies ist ein Widerspruch, da Delaunay-Kanten sich nicht kreuzen (Lemma 6.6). □

Wir haben noch nicht ausgeführt, wie der neue Punkt p_i der Triangulation eingefügt wird und die Kantenmenge zu einer Triangulation ergänzt werden kann (Zeile 4 in DelaunayTriangulation). Dafür müssen wir die Fläche in der Triangulation finden, die den neuen Punkt enthält. Für einen gegebenen Punkt und ein gegebenes Dreieck lässt sich in $O(1)$ Zeit testen, ob der Punkt in dem Dreieck enthalten ist. Das Finden der Fläche kann mittels einer Breitensuche über die Kanten der Triangulation in $O(n)$ Zeit geschehen. Für jede besuchte Kante testen wir die höchstens zwei anliegenden Dreiecke. Wenn ein Dreieck gefunden wird, das den Punkt p_i enthält dann müssen Triangulationskanten von p_i zu allen Knoten des Dreiecks hinzugefügt werden. Wenn kein Dreieck gefunden wird, das den Punkt p_i enthält, dann muss der Punkt in der äußeren Fläche enthalten sein. In diesem Fall müssen nur diese Kanten (a, c) mit dem neuen Punkt p_i verbunden werden, wo das Tupel (a, c, p_i) entgegen dem Uhrzeigersinn orientiert ist. Abbildung 6.9 zeigt ein Beispiel.

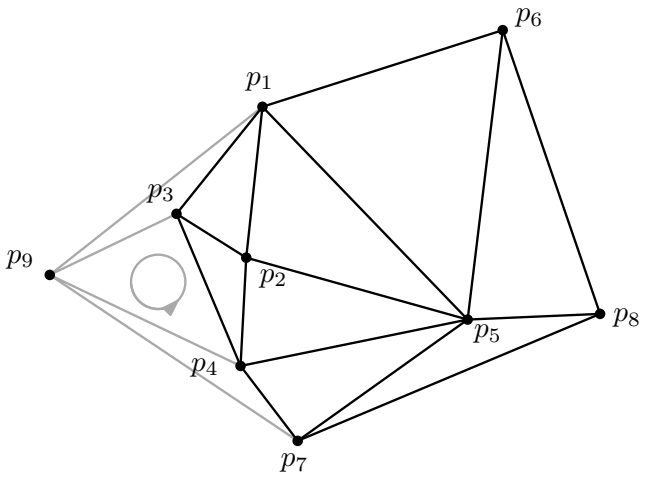


Abbildung 6.9: Einfügen des Punktes p_9 in die äußere Fläche einer Triangulation von Punkten $p_1, \ldots, p_8$.

Theorem 6.19. *Die Delaunay-Triangulation von n Punkten in allgemeiner Lage in der Ebene kann in $O(n^2)$ Zeit und mit $O(n)$ Speicherplatz berechnet werden.*

Beweis. Die Korrektheit von DELAUNAY-TRIANGULATION folgern wir aus Lemmas 6.18 und 6.16 mithilfe von Induktion über die for-Schleife. Betrachten wir einen Durchlauf der for-Schleife, in dem p_i hinzugefügt wird. Offenbar folgt aus Lemma 6.17, dass nur solche Dreiecke aus DT_{i-1} entfernt werden, die mit p_i im Konflikt stehen. Aus Lemma 6.18 folgt, dass die Dreiecke, die nach Einfügen von p_i verblieben sind, nicht im Konflikt mit p_i stehen und aus Lemma 6.16 folgt, dass sie deswegen in DT_i sein müssen. Aus 6.16 folgt auch, dass alle neuen Dreiecke in DT_i zu p_i inzident sein müssen. Die vom Algorithmus berechnete Triangulation ist die einzige Triangulation die diese Bedingungen erfüllt.

Als nächstes analysieren wir die Laufzeit. In einem Durchlauf der for-Schleife von DELAUNAYTRIANGULATION wird ein Punkt p_i in die Triangulation eingefügt und danach rekursiv eine Sequenz von Flips durchgeführt. Das Einfügen kann in $O(n)$ Zeit durchgeführt werden, da die Triangulation höchstens $O(n)$ Kanten enthält und das Testen, ob ein konkretes Dreieck den Punkt p_i enthält, in konstanter Zeit durchgeführt werden kann. Wir behaupten, dass die Anzahl der Diagonalenwechsel die im Durchlauf einer for-Schleife durchgeführt werden durch $O(n)$ begrenzt ist. Das folgt daraus, dass jede Kante höchstens einmal ersetzt wird, und zwar durch eine Kante, die mit p_i inzident ist. Eine Kante, die mit p_i inzident ist, wird im weiteren Verlauf nicht mehr gewechselt. Da der Algorithmus insgesamt n Durchläufe der for-Schleife durchführt, ergibt sich eine Laufzeit von $O(n^2)$.

Zuletzt analysieren wir den Speicherplatz. Zu jedem Zeitpunkt im Algorithmus wird nur die Halbkanten-Datenstruktur der aktuellen Triangulation mitgeführt. Laut Theorem 6.2 ist die Anzahl der Kanten und somit die Anzahl der Kanten in der Triangulation und somit ist die Anzahl der Halbkanten-Objekte und Knoten-Objekte in $O(n)$. Jedes Objekt der Datenstruktur speichert eine konstante Anzahl von Zeigern und Variablen. Der Speicherplatz, den der Algorithmus benutzt, ist also in $O(n)$. □

Wir haben also gesehen, dass der inkrementelle Algorithmus zur Berechnung der Delaunay-Triangulation eine Laufzeit von $O(n^2)$ im schlimmsten Fall hat. Ähnlich wie bei Quicksort kann man zeigen, dass die Laufzeit mithilfe von Randomisierung verbessert werden kann. Wir beschränken uns hier in unserer Betrachtung auf die Anzahl der vom Algorithmus durchgeführten Flips. Die asymptotische Laufzeit ändert sich dadurch nicht, da in jedem Schritt noch die Fläche gefunden werden muss, in der sich der neu hinzuzufügende Punkt befindet. Dieses Problem nennt sich Punktlokalisierung. Wir werden uns später in der Vorlesung noch weiter damit beschäftigen. Dennoch ist die Analyse der erwarteten Anzahl von Flips exemplarisch für die Analyse von randomisierten geometrischen Algorithmen.

Wir nehmen an, dass die Reihenfolge, in der die Punkte eingefügt werden, gleichverteilt zufällig über alle Permutationen der Eingabe gewählt ist. Wir werden zeigen, dass die erwartete Anzahl der Flips, die nach dem Einfügen eines Punktes in die Delaunay-Triangulation durchgeführt werden, konstant ist. Ingesamt führt der Algorithmus also im Erwartungsfall nur $O(n)$ Flips durch, wobei der Erwartungswert über die zufällig gewählte Permutation der Eingabe gebildet wird.

Theorem 6.20. *Sei die Einfügereihenfolge der n Punkte zufällig gleichverteilt über alle möglichen Permutationen gewählt. Dann ist die erwartete Anzahl der Flips, die der Algorithmus* DelaunayTriangulation *insgesamt durchführt, in $O(n)$.*

Beweis. Seien $p_1, \ldots, p_n$ die Punkte in der Reihenfolge wie sie vom Algorithmus zur Delaunay-Triangulation hinzugefügt werden. Die Permutation ist zufällig gleichverteilt über alle $n!$ möglichen Permutationen gewählt. Wir dürfen annehmen, dass diese zufällige Reihenfolge schrittweise wie folgt generiert wurde. Wir wählen zuerst den Punkt p_n zufällig gleichverteilt aus der Gesamtmenge der Punkte aus. Danach wählen wir für $j = 1, \ldots n-2$ den Punkt p_{n-j} jeweils zufällig gleichverteilt aus der verbleibenden Restmenge aus. (Nachdem wir den Punkt p_2 ausgewählt haben, steht der Punkt p_1 fest, da er als einziger in der Restmenge verblieben ist.)

Der Algorithmus initialisiert die Delaunay-Triangulation T mit DT_3, welche nur das Dreieck der ersten drei Punkte p_1, p_2, p_3 enthält. Bis dahin wurden noch keine Flips ausgeführt. Betrachten wir also die Sequenz der vom Algorithmus erzeugten Delaunay-Triangulationen DT_i rückwärts von $i = n, \ldots, 4$. Betrachten wir ein festes i. Der Punkt p_i, der zu DT_{i-1} hinzugefügt wird um DT_i zu erhalten, ist zufällig gleichverteilt in der Menge der Punkte $S = \{p_1, \ldots, p_i\}$. Das heißt, jeder der Punkte in S hat die gleiche Wahrscheinlichkeit, als Punkt p_i ausgewählt zu werden.

Die Menge S ist genau die Menge der Punkte, die in DT_i als Knoten vorkommen. Wir halten also die Menge S fest und damit ist auch die Delaunay-Triangulation DT_i fest.

Laut dem Stern-Lemma (Lemma 6.16) ist die Anzahl der Flips, die nach dem Hinzufügen von p_i zu DT_{i-1} vom Algorithmus durchgeführt werden, gleich dem Knotengrad von p_i in DT_i. Aus Theorem 6.2 folgt, dass die Anzahl der Kanten in DT_i in $O(i)$ und somit der durchschnittliche Knotengrad konstant ist. Der erwartete Knotengrad eines gleichverteilt zufällig gewählten Knoten in DT_i ist also konstant. Da wir p_i gleichverteilt zufällig unter den Knoten in DT_i auswählen, ist sein erwarteter Knotengrad und somit die erwartete Anzahl von Flips für die Herstellung der Delaunay-Eigenschaft

nach Einfügen von p_i konstant. Die erwartete Anzahl von Flips, die der Algorithmus insgesamt durchführt, ist also in $O(n)$. □

6.2 Nächste-Nachbarn-Suche

Ein klassisches Problem der Algorithmischen Geometrie ist das Problem des nächsten Postamtes (Knuth, 1973 [5]). Gegeben eine Menge S von n Punkten in der Ebene und ein Punkt q, welcher Punkt aus der Menge S ist der am nächsten zu q liegende? Gesucht ist eine Datenstruktur für die Menge S, welche die Anfrage mit einem vorher unbekannten Punkt q möglichst effizient beantworten kann. Man stelle sich vor, dass jeder Punkt in S die Position eines Postamtes beschreibt und man sich am Punkt q befindet und ein Paket senden will. Welches Postamt am nächsten liegt, wird dabei als der Punkt mit dem kleinsten Euklidischen Abstand zu q festgelegt. Das nächste Postamt wird auch als der nächste Nachbar von q in S bezeichnet.

Definition 6.21 (Euklidischer Abstand)**.** *Für $a = (a_1, a_2) \in \mathbb{R}^2$ und $b = (b_1, b_2) \in \mathbb{R}^2$ ist der Euklidische Abstand gegeben als $\|a - b\| = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2}$.*

Definition 6.22 (Nächster Nachbar)**.** *Gegeben eine Menge $S \subseteq \mathbb{R}^2$ und ein Punkt $q \in \mathbb{R}^2$, der nächste Nachbar von q in S ist der Punkt $p \in S$, sodass $\|q - p\|$ minimiert wird.*

6.2.1 Voronoi-Diagramm

Sei $S \subseteq \mathbb{R}^2$ eine Menge von n Punkten in der Ebene. Betrachte für einen Punkt $p \in S$ die Menge der Punkte, für die p der eindeutige nächste Nachbar in S ist:

$$\left\{ x \in \mathbb{R}^2 \;\middle|\; \forall q \neq p \in S \|x - p\| < \|x - q\| \right\}$$

Das Voronoi-Diagramm ist die Unterteilung der Ebene $\mathbb{R}^2$ in diese Teilmengen der Ebene für alle Punkte in S. Im Folgenden werden wir genaue Eigenschaften und Form dieser Mengen und des daraus resultierenden Diagramms herleiten. Wir werden auch einen direkten Zusammenhang zu der Delaunay-Triangulation von S herstellen.

Definition 6.23. *Der Bisektor $B(p, q)$ zwischen zwei Punkten $p \in \mathbb{R}^2$ und $q \in \mathbb{R}^2$ ist die Menge*

$$B(p, q) = \left\{ x \in \mathbb{R}^2 \;\middle|\; \|p - x\| = \|q - x\| \right\}$$

Der Bisektor enthält genau die Punkte, für die der Abstand zu p gleich dem Abstand zu q ist. Für feste p und q erfüllen diese Punkte eine Geradengleichung, wie sich leicht überprüfen lässt.

$$\|p - x\| = \|q - x\|$$

$$
\begin{aligned}
\Leftrightarrow \quad & \|p - x\|^2 = \|q - x\|^2 \\
\Leftrightarrow \quad & \langle p - x, p - x \rangle = \langle q - x, q - x \rangle \\
\Leftrightarrow \quad & \langle p, p \rangle + \langle x, x \rangle - 2 \langle p, x \rangle = \langle q, q \rangle + \langle x, x \rangle - 2 \langle q, x \rangle \\
\Leftrightarrow \quad & \langle p, p \rangle - 2 \langle p, x \rangle = \langle q, q \rangle - 2 \langle q, x \rangle \\
\Leftrightarrow \quad & 2 \langle q, x \rangle - 2 \langle p, x \rangle = \langle q, q \rangle - \langle p, p \rangle \\
\Leftrightarrow \quad & \langle 2(q - p), x \rangle = \langle q, q \rangle - \langle p, p \rangle \\
\Leftrightarrow \quad & \langle w, x \rangle = u
\end{aligned}
$$

mit $w = 2(q - p) \in \mathbb{R}^2$ und $u = \langle q, q \rangle - \langle p, p \rangle \in \mathbb{R}$.

Der Bisektor $B(p, q)$ unterteilt die Ebene in die Mengen

$$B_-(p, q) = \left\{ x \in \mathbb{R}^d \;\middle|\; \langle w, x \rangle < u \right\}, \; B_+(p, q) = \left\{ x \in \mathbb{R}^d \;\middle|\; \langle w, x \rangle > u \right\},$$

und den Bisektor selbst. B_- enthält alle die Punkte die näher an p liegen als an q und B_+ enthält alle die Punkte die näher an q liegen als an p.

Definition 6.24 (Voronoi-Region)**.** *Die Voronoi-Region eines Punktes p in der Menge S ist die Menge der Punkte, für die p der eindeutige nächste Nachbar ist:*

$$\mathcal{V}(p, S) = \bigcap_{\substack{q \in S \\ p \neq q}} B_-(p, q)$$

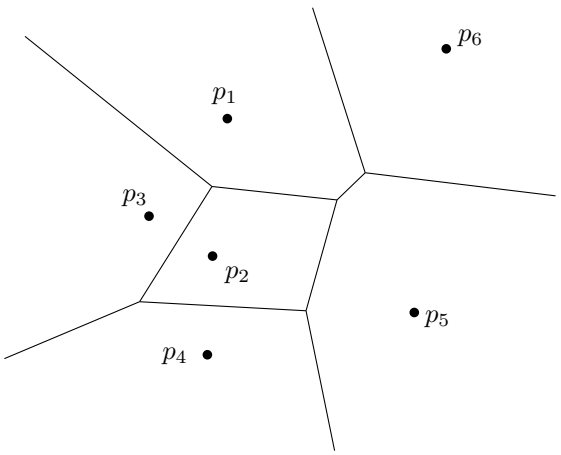


Abbildung 6.10: Beispiel eines Voronoi-Diagramms.

Aus obigen Definitionen folgt direkt, dass die Voronoi-Region von p bezüglich S genau die Menge der Punkte ist, für die p der eindeutige nächste Nachbar in S ist.

Voronoi-Regionen können unbeschränkt sein. Die Ränder der Voronoi-Regionen setzen sich zusammen aus Teilen der Bisektoren von Paaren von Punkten aus S. In der Ebene formen diese kreuzungsfreien geometrischen Graphen mit Knoten und Kanten, wobei manche dieser Kanten unbeschränkt sind (Strahlen). Wir definieren einen *abstrakten Knoten* im Unendlichen, den wir als zweiten Endpunkt dieser unbeschränkten Kanten betrachten. Wir nennen die Knoten dieses Graphen Voronoi-Knoten und die Kanten

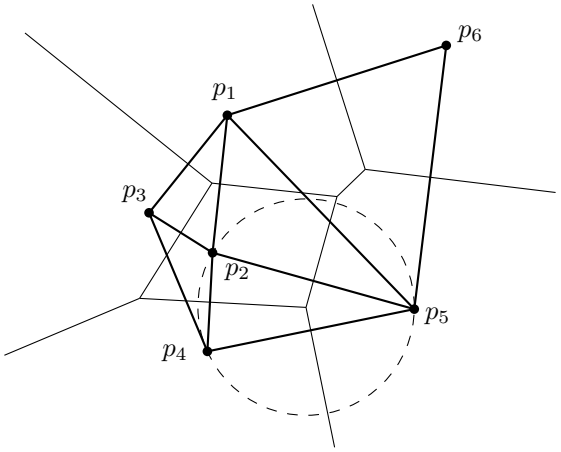


Abbildung 6.11: Voronoi-Diagramm und Delaunay-Triangulation einer Punktmenge.

werden Voronoi-Kanten genannt. Den Graphen nennen wir das Voronoi-Diagramm von S.

Es gibt einen natürlichen Zusammenhang zwischen den Voronoi-Knoten und den Delaunay-Kanten derselben Punktmenge, den wir in folgendem Theorem formulieren.

Theorem 6.25. *Sei S eine endliche Menge von Punkten in allgemeiner Lage in der Ebene, mit $|S| \geq 3$. Für den Graph des Voronoi-Diagramms von S und den Graph der Delaunay-Triangulation von S gilt:*

(i) Drei Punkte in $a, b, c \in S$ bilden ein Dreieck in der Delaunay-Triangulation genau dann wenn es ein Voronoi-Knoten gibt, der zu den Voronoi-Regionen $\mathcal{V}(a, S)$, $\mathcal{V}(b, S)$ und $\mathcal{V}(c, S)$ inzident ist.

(ii) Zwei Punkte $a, b \in S$ sind genau dann durch eine Delaunay-Kante verbunden, wenn es eine Voronoi-Kante gibt, die zu den Voronoi-Regionen $\mathcal{V}(a, S)$ und $\mathcal{V}(b, S)$ inzident ist.

Bevor wir das Theorem beweisen, wollen wir noch eine Anmerkung zu Teil (ii) machen. Die Voronoi-Kante muss nicht unbedingt ihre zugehörige Delaunay-Kante kreuzen. Ein Beispiel dafür ist in Abbildung 6.11 zu sehen. Die Delaunay-Kante zwischen den Punkten p_3 und p_4 ist disjunkt zur entsprechenden Voronoi-Kante. Der abstrakte Voronoi-Knoten im Unendlichen wird aus unserer Betrachtung ausgenommen. Er entspricht in diesem Sinne der äußeren Fläche der Delaunay-Triangulation.

Beweis. Wir zeigen zunächst (i). Seien $a, b, c \in S$ paarweise verschieden. Der Umkreismittelpunkt z des Dreiecks (a, b, c) ist der gemeinsame Schnittpunkt der Bisektoren $B(a, b), B(a, c)$, und $B(b, c)$. Die folgenden beiden Aussagen sind äquivalent:

(1) Die nächsten Nachbarn von z in S sind a, b und c.

(2) Der Kreis mit Mittelpunkt z und a, b, c auf dem Rand hat keinen anderen Punkt aus S in seinem Inneren.

Genau dann, wenn beide Aussagen auf z zutreffen, bildet (a, b, c) ein Dreieck in der Delaunay-Triangulation und gibt es ein Voronoi-Knoten, der zu den Voronoi-Regionen $\mathcal{V}(a, S)$, $\mathcal{V}(b, S)$ und $\mathcal{V}(c, S)$ inzident ist. Daraus folgt (i).

Als nächstes zeigen wir (ii). Seien $a, b \in S$ verschieden. Sei x ein Punkt auf dem Bisektor $B(a, b)$. Die folgenden beiden Aussagen sind äquivalent:

(3) Die nächsten Nachbarn von x in S sind a und b.

(4) Der Kreis mit Mittelpunkt x und Punkten a und b auf dem Rand hat keinen anderen Punkt aus S in seinem Inneren.

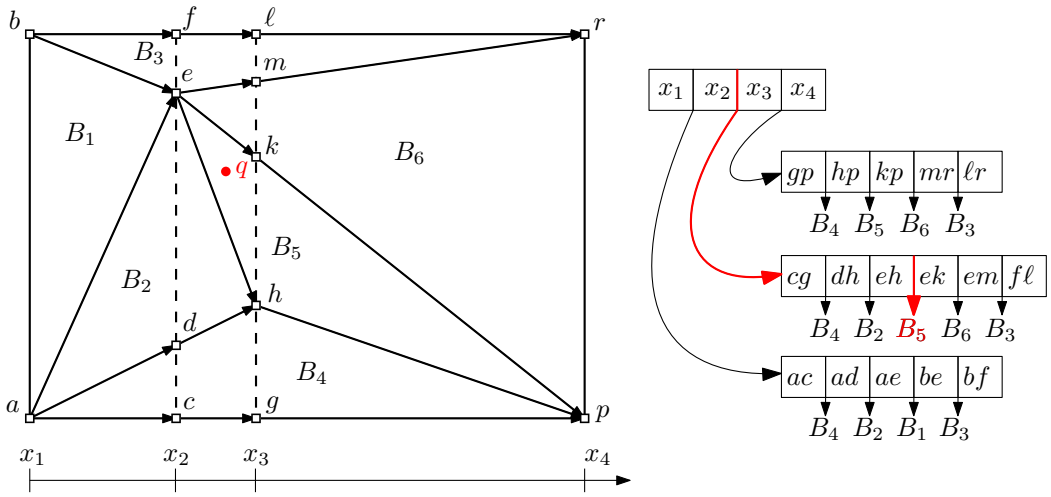
Genau dann, wenn ein Punkt x mit den obigen Eigenschaften existiert, dann ist (a, b) Delaunay-Kante (und somit nach Theorem 6.7 Kante der Delaunay-Triangulation) und ebenso existiert die Voronoi-Kante die zu den Voronoi-Regionen $\mathcal{V}(a, S)$ und $\mathcal{V}(b, S)$ inzident ist. Daraus folgt (ii). □

Aufgrund von Theorem 6.25 können wir mithilfe einer Breitensuche über die Dreiecke der Delaunay-Triangulation in der Halbkanten-Datenstruktur das Voronoi-Diagramm in $O(n)$ Zeit berechnen. Damit folgt aus Theorem 6.19, dass das Voronoi-Diagram einer Menge von n Punkten in allgemeiner Lage in der Ebene in $O(n^2)$ Zeit und mit $O(n)$ Speicherplatz berechnet werden kann.

6.2.2 Punktlokalisierung

Gegeben sei eine Triangulation T in der Ebene und ein Punkt q, welche Fläche von T enthält q? Dieses Problem wird auch Punktlokalisierung genannt. Gesucht ist eine Datenstruktur, die diese Art von Anfragen effizient beantworten kann. Im Allgemeinen ist T keine Triangulation sondern ein kreuzungsfreier geometrischer Graph, zum Beispiel das Voronoi-Diagramm einer Punktmenge. Die ebene Unterteilung eines solchen Graphen mit n Knoten lässt sich aber immer mit $O(n)$ zusätzlichen Kanten zu einer Triangulation erweitern. Wir nehmen im Folgenden an, dass eine solche Triangulation gegeben ist.

Wir betrachten zunächst eine einfache Datenstruktur für Punktlokalisierung. Wir unterteilen die Ebene in vertikale Streifen, indem wir eine vertikale Gerade durch jeden Knoten von T ziehen. Seien $x_1 < \cdots < x_k$ die entsprechenden x-Koordinaten der Vertikalen. Betrachten wir die Kanten von T beschränkt auf einen Streifen (x_i, x_{i+1}), dann sehen wir, dass diese Kanten den Streifen in trapezförmige und dreieckige Bereiche aufteilen. Wir orientieren diese Kanten von links nach rechts indem wir den Schnittpunkt der Kante mit x_i und x_{i+1} in dieser Reihenfolge betrachten. Wir sortieren die Kanten für jeden Streifen in der lexikographischen Reihenfolge der y-Koordinaten ihrer Endpunkte und speichern sie in dieser Reihenfolge in einem Array. Wir erhalten ein Array per Streifen. Zeiger auf diese Arrays speichern wir in einem weiteren Array, das die Koordinaten $x_1, \ldots, x_k$ aufsteigend sortiert enthält.



Eine Anfrage mit einem Punkt q können wir nun wie folgt beantworten. Wir suchen zuerst mit binärer Suche nach dem vertikalen Streifen, der q enthält und finden den Zeiger auf das entsprechende Array. In einem zweiten Schritt führen wir eine binäre Suche auf diesem Array aus, um den Bereich des Streifens zu finden, der q enthält. Ob der Punkt q im Streifen (x_i, x_{i+1}) oberhalb oder unterhalb einer Kante (u, v) liegt die zwischen den beiden Vertikalen bei x_i und x_{i+1} verläuft, können feststellen, indem wir die Orientierung des Tupels (u, v, q) betrachten: ist das Tupel gegen den Uhrzeigersinn orientiert, dann liegt q oberhalb, sonst liegt q unterhalb der Kante. Wir nennen dieses Verfahren die *doppelte binäre Suche*.

Um den Speicherplatz der Datenstruktur zu analysieren, gilt es herauszufinden, wieviele Schnittpunkte die erzeugten vertikalen Geraden mit den Kanten von T haben können. Sei n die Anzahl der Knoten der Triangulation T. Im schlimmsten Fall werden $O(n^2)$ Schnittpunkte erzeugt, da wir n Vertikalen und $O(n)$ Triangulationskanten haben und es $O(n^2)$ Paare geben kann, die jeweils einen Schnittpunkt erzeugen. Wir müssen die Vertikalen einmal nach der x-Koordinate sortieren und dann in jedem der n Streifen $O(n)$ Kanten sortieren und in die Arrays schreiben. Die Laufzeit, um die Datenstruktur zu bauen, ist also in $O(n^2 \log n)$. Der Speicherplatz ist in $O(n^2)$. Die Anfragezeit in dieser einfachen Datenstruktur ist $O(\log n)$, da wir nur zwei binäre Suchen jeweils auf einem Array der Länge $O(n)$ ausführen. Leider kann der Speicherplatz im schlimmsten Fall tatsächlich quadratisch in n sein, was die Datenstruktur für die meisten praktischen Anwendungen ungeeignet macht.

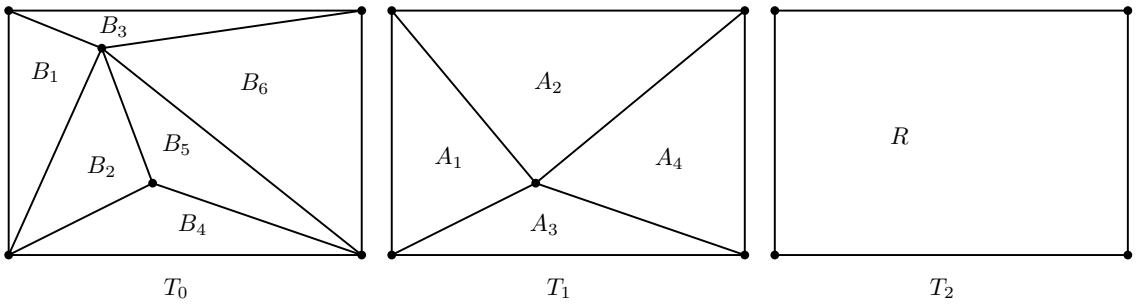
Zusammenfassend kennen wir jetzt also zwei Datenstrukturen für Punktlokalisierung. Die *lineare Suche*, welche wir im Abschnitt über die Delaunay-Triangulation benutzt haben, führt eine Breitensuche auf den Dreiecken der Triangulation aus und testet für jedes Dreieck, ob es den Anfragepunkt enthält.

	Anfragezeit	Speicherplatz
lineare Suche	$O(n)$	$O(n)$
doppelte binäre Suche	$O(\log n)$	$O(n^2)$

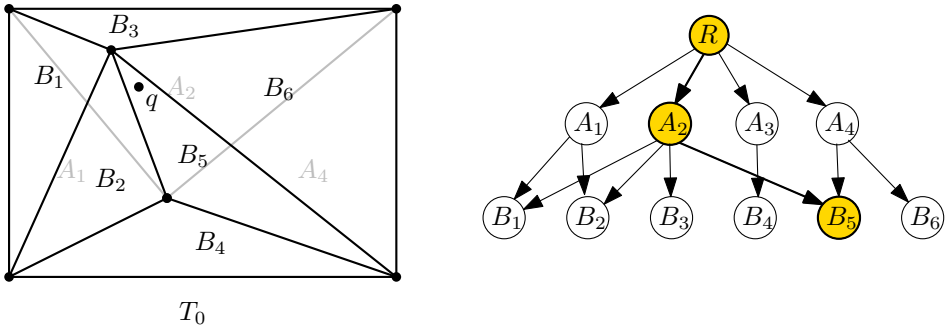
Im nächsten Abschnitt werden wir eine Datenstruktur für Punktlokalisierung in Triangulationen kennenlernen, die Speicherplatz $O(n)$ und Anfragezeit $O(\log n)$ hat.

6.2.3 Triangulationshierachie

Wir befassen uns nun mit einer weiteren Datenstruktur für Punktlokalisierung in einer Triangulation T von n Punkten. Wir nehmen der Einfachheit halber an, dass die äußeren Kanten der Triangulierung ein Rechteck R bilden. Die Datenstruktur basiert auf einer Sequenz von Triangulationen $T_0, \ldots, T_h$, ausgehend von $T_0 = T$, in der die Anzahl der Dreiecke in jedem Schritt reduziert wird, wobei die äußeren Kanten immer das gleiche Rechteck R bilden.



Auf den Dreiecken dieser Triangulationen wird dann ein gerichteter kreisfreier Graph G erstellt, den wir die *Suchstruktur* nennen. Zwei Dreiecke in Triangulationen T_i und T_{i-1} sind in G durch eine Kante verbunden, wenn entweder zwei ihrer Kanten sich kreuzen, oder eines im anderen enthalten ist. Wir nennen eine solche Datenstruktur eine *Triangulationshierarchie*. Eine Suchanfrage mit einem Punkt q startet im Wurzelknoten der Suchstruktur und folgt in jedem Schritt der ersten ausgehenden Kante zu einem Knoten, der q enthält. Wenn der Algorithmus an einem Knoten angekommen ist, der keine ausgehenden Kanten hat, dann wird ein Zeiger zu dem Dreieck dieses Knotens zurückgegeben.



Theorem 6.26. *Sei G der Graph der Suchstruktur einer Triangulationshierarchie für eine Triangulation mit n Knoten. Sei d der maximale Outgrad von G und sei h die Länge des längesten Weges von der Wurzel zu einem Knoten auf der untersten Ebene. Die Anfragezeit mit einem Punkt $q \in \mathbb{R}^2$ in G ist in $O(dh)$.*

Beweis. Der Algorithmus für die Suchanfrage ist oben beschrieben. Wir nehmen an, dass der Graph G als Adjazenzliste gespeichert ist. In jedem Schritt des Algorithmus werden höchstens d ausgehende Kanten des aktuellen Knotens getestet. Das Testen, ob ein Dreieck einen Punkt enthält kann in konstanter Zeit durchgeführt werden. Die

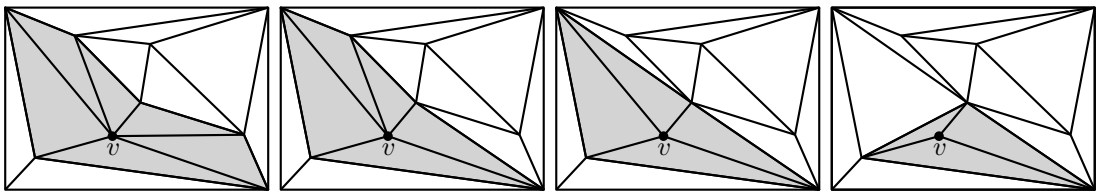
Anzahl der Schritte entspricht dann der Länge des Pfades von der Wurzel zu einem Knoten auf der untersten Ebene der Triangulationshierarchie, da der Algorithmus in jedem Schritt einer Kante zu einem Dreieck in einer tieferen Ebene der Triangulationshierarchie folgt. □

Als nächstes wollen wir eine Triangulationshierarchie mit Speicherplatz $O(n)$ und Anfragezeit $O(\log n)$ bauen. Die Datenstruktur geht zurück auf Kirkpatrick. Dafür definieren wir zunächst das Konzept der unabhängigen Knotenmenge in einem Graphen.

Definition 6.27 (Unabhängige Knotenmenge)**.** *Eine unabhängige Knotenmenge eines Graphen $G = (V, E)$ ist eine Menge $S \subseteq V$, mit der Eigenschaft dass keine zwei Kanten aus S durch eine Kante in G verbunden sind.*

Der Algorithmus von Kirkpatrick generiert die Sequenz von Triangulationen schrittweise. T_{i+1} wird aus T_i generiert, indem eine unabhängige Knotenmenge entfernt wird, wobei jeder Knoten aus der entfernten Menge einen konstanten Knotengrad hat. Um in jedem Schritt wieder eine Triangulation zu erhalten, müssen (1) neue Triangulationskanten eingefügt werden, wenn der entfernte Knoten einen Knotengrad größer als drei hat. Desweiteren wollen wir, (2) dass die Anzahl der Knoten der Triangulation in jedem Schritt um einen konstanten Faktor schrumpft.

Wir besprechen zunächst, wie wir das erste Ziel erreichen können. Sei v ein Knoten, den wir aus einer Triangulation T mitsamt seinen inzidenten Kanten entfernen wollen. Wir nehmen an, dass v nicht auf der äußeren Fläche liegt. Wenn der Knotengrad von v größer als drei ist, würde durch das Entfernen eine Fläche mit mehr als drei Kanten erzeugen, wir hätten also keine Triangulation mehr. Um sicherzustellen, dass wir eine Triangulation erhalten, führen wir vor dem Entfernen von v mit einem Greedy-Algorithmus so lange zulässige Diagonalenwechsel auf den Kanten inzident zu v durch, bis der Knotengrad von v drei ist. Man kann zeigen, dass immer ein zulässiger Diagonalenwechsel einer Kante von v existiert, sofern der Knotengrad von v mindestens 4 ist und v nicht auf der äußeren Fläche liegt. Der Algorihmus kann in konstanter Zeit implementiert werden, sofern der Knotengrad von v konstant ist.



Wir wollen nun ein paar grundlegende Eigenschaften von Triangulationen zeigen, die uns dabei helfen, das zweite Ziel zu erreichen.

Lemma 6.28. *Sei $k \geq 5$, eine Triangulation T mit $n \geq 4$ Knoten hat mindestens $\lceil \frac{k-5}{k+1} n \rceil$ Knoten mit Knotengrad höchstens k.*

Beweis. Sei m die Anzahl der Knoten in der Triangulation T, die Knotengrad größer als k haben, für ein $k \geq 5$. Sei d_i der Knotengrad des $i$ten Knotens. Betrachten wir

die Summe der Knotengrade über alle Knoten von T, dann gilt

$$\sum_{i=1}^{n} d_i \geq m(k+1)$$

Gleichzeitig wissen wir, dass jede Kante zu genau 2 Knoten inzident ist. Also gilt $\sum_{i=1}^{n} d_i = 2e$. Aus der Analyse im Beweis von Theorem 6.2 folgt, dass $e \leq 3(n-2)$. Zusammen implizieren diese Ungleichungen, dass

$$6n - 12 \geq 2e = \sum_{i=1}^{n} d_i \geq m(k+1)$$

und somit gilt

$$m \leq \frac{6n-12}{k+1}$$

Es gibt $n - m$ Knoten, die Knotengrad höchstens k haben. Für diese Anzahl gilt

$$n - m \geq n - \frac{6n-12}{k+1} = \frac{k-5}{k+1}n + \frac{12}{k+1} \geq \frac{k-5}{k+1}n$$

Da die Anzahl der Knoten eine natürliche Zahl ist, dürfen wir aufrunden. □

Einsetzen mit $k = 11$ ergibt das folgende Korollar.

Korollar 6.29. *Eine Triangulation T mit $n \geq 4$ Knoten hat mindestens $\lceil \frac{n}{2} \rceil$ Knoten mit Knotengrad höchstens 11.*

Als nächstes wollen wir zeigen, dass wir eine geeignete unabhängige Knotenmenge in linearer Zeit berechnen können. Dafür benutzen wir den folgenden Greedy-Algorithmus.

```
IndependentSet(Triangulation T)
 1   Sei W die Teilmenge der Knoten von T mit Knotengrad(v) ≤ 11,
 2   U = ∅;
 3   while(|W| ≥ 5){
 4       Sei v ∈ W innerer Knoten in T.
 5       Füge v zu U hinzu.
 6       Entferne v und seine benachbarten Knoten aus W.
14   } return U;
```

Lemma 6.30. *Es existiert eine natürliche Zahl C, sodass für jede Triangulation mit $n \geq C$ Knoten, von denen höchstens 4 auf der äußeren Fläche liegen, ein Algorithmus existiert, der in Laufzeit $O(n)$ eine unabhängige Knotenmenge S mit $|S| = \lceil \frac{n}{25} \rceil$ berechnet, wobei keiner der Knoten von S auf der äußeren Fläche von T liegt und jeder Knoten in S Knotengrad höchstens $k = 11$ in T hat.*

Beweis. Sei die Triangulation in einer Halbkanten-Datenstruktur gegeben. Wir können mithilfe einer Breitensuche die Menge der Knoten mit Knotengrad höchstens k in $O(n)$ Zeit identifizieren. Sei diese Menge W. Aus Korollar 6.29 folgt $|W| \geq \lceil \frac{n}{2} \rceil$. Allerdings kann die Menge W auch Knoten enthalten, die auf der äußeren Fläche liegen.

Wir nutzen den Greedy-Algorithmus IndependentSet, um aus der Menge W eine geeignete unabhängige Knotenmenge zu wählen. In jedem Schritt wählt wir einen Knoten aus W, der nicht auf der äußeren Fläche liegt, und wir entfernen ihn und alle seine direkten Nachbarn aus W. In jedem Schritt entfernen wir also höchstens $k+1$ Knoten aus W und erweitern unsere unabhängige Knotenmenge S um genau einen Knoten. Sei w_i die Größe von W nach i Schritten. Es gilt

$$w_i \geq w_0 - i(k+1)$$

Sei $m = |S|$ die Größe der vom Algorithmus bestimmten unabhängigen Knotenmenge. Nach m Schritten terminiert also der Algorithmus, da wir keinen weiteren Knoten aus W wählen können. Da höchstens vier Knoten auf der äußeren Fläche liegen dürfen, die wir nicht wählen dürfen und wir alle anderen übrigen Knoten in der Menge W noch wählen dürften, gilt also $w_m < 5$. Daraus folgt

$$5 > w_m \geq w_0 - m(k+1)$$

Wegen Korollar 6.29 gilt $w_0 \geq \frac{n}{2}$ und da $k = 11$ gewählt wurde, folgt

$$5 > \frac{n}{2} - 12m$$

Diese Ungleichung können wir nach m umstellen. Für hinreichend große C gilt für $n \geq C$

$$m > \frac{n}{24} - \frac{5}{12} > \frac{n}{25}$$

Insbesondere ist dies der Fall für $C = 251$, wie sich durch Umformen obiger Ungleichung verifizieren lässt. Da m eine natürliche Zahl ist, dürfen wir aufrunden. Es gilt also $m \geq \left\lceil \frac{n}{25} \right\rceil$ für hinreichend große n. □

Wir zeigen nun, dass die Datenstruktur die eingangs behaupteten Schranken bezüglich Anfragezeit und Speicherplatz einhält. Die Datenstruktur bekommt als Eingabe eine Triangulation, in der die äußere Fläche nur vier Kanten hat. Für Eingaben, die diese Eigenschaft nicht erfüllen, können wir ein großes Rechteck hinzufügen, welches alle Punkte der Triangulation enthält, und entsprechende Kanten hinzufügen, um eine solche Triangulation zu erhalten.

Um die Datenstruktur zu bauen, generieren wir aus der Eingabe T_0 eine Sequenz von Triangulationen $T_1, \ldots, T_h$ mit absteigender Anzahl von Knoten. Zusätzlich bauen wir die Suchstruktur G auf den Dreiecken der Triangulationen wie oben beschrieben. Dabei berechnen wir Triangulation T_i aus Triangulation T_{i-1}, sofern die Anzahl der Knoten in T_{i-1} mindestens C ist, indem wir den Algorithmus aus Lemma 6.30 an um eine unabhängige Knotenmenge S zu berechnen. Für jeden Knoten v in S führen wir Diagonalenwechsel auf den inzidenten Kanten durch, bis der Knotengrad von v gleich 3

ist. Dann entfernen wir den Knoten v mitsamt seinen verbleibenden inzidenten Kanten aus der Triangulation. Für jedes Dreieck aus T_{i-1}, das v als Eckpunkt hatte, und jedes neue Dreieck in T_i, das durch Flips von v entstanden ist, wird getestet, ob sie sich schneiden. Wenn dem so ist, dann wird eine Kante in der Suchstruktur hinzugefügt. Die Anzahl der Paare von Dreiecken, die pro Knoten in S getestet werden müssen, ist konstant, da den Knotengrad von v in T_{i-1} höchstens 11 war. Am Ende fügen wir noch das Rechteck R als Wurzelknoten in unserer Suchstruktur hinzu und verbinden es mithilfe von Kanten mit allen Dreiecken aus T_h.

```
KIRKPATRICK(Triangulation T, int n)
 1   i = 0;
 2   T_0 = COPY(T);
 3   while(n > C){
 4       S = INDEPENDENTSET(T);
 5       foreach(v ∈ S){
 6           while(Knotengrad(v) > 3){
 7               Führe zulässigen Flip auf einer Kante von v aus.
 8           }
 9           Entferne v mit seinen Kanten aus T.
10       }
11       n = n − |S|;
12       i = i + 1;
13       T_i = COPY(T);
14   }
```

Theorem 6.31. *Sei T eine Triangulation mit $n \geq 4$ Knoten in der Ebene, von denen höchstens 4 auf der äußeren Fläche liegen. Wir können in Laufzeit $O(n)$ eine Datenstruktur bauen, die in Zeit $O(\log n)$ das Dreieck bestimmt, das den gegebenen Anfragepunkt enthält. Die Datenstruktur benutzt Speicherplatz in $O(n)$.*

Beweis. Wieviele Triangulationen werden vom Algorithmus erstellt? Aus Lemma 6.30 folgt, dass die Anzahl der Knoten in jedem Schritt um mindestens einen Faktor $\frac{1}{25}$ kleiner wird. Der Prozess terminiert sobald höchstens $C = 251$ Knoten übrig sind. Wir suchen also nach der kleinsten natürlichen Zahl h, sodass gilt

$$\left(\frac{24}{25}\right)^h n \;\leq\; C$$

Das ist äquivalent zu

$$h \cdot \log_2\left(\frac{25}{24}\right) \;\geq\; \log_2\left(\frac{n}{C}\right)$$

Die kleinste natürliche Zahl h, die diese Bedingungen erfüllt ist

$$h \;=\; \left\lceil \frac{\log_2\left(\frac{n}{C}\right)}{\log_2\left(\frac{25}{24}\right)} \right\rceil \in O(\log n)$$

Sei n_i die Anzahl der Knoten in der Triangulation T_i. Die Konstruktion dauert in jedem Schritt $O(n_i)$ Zeit. Die Gesamtlaufzeit ist also asymptotisch beschränkt durch

$$\sum_{i=0}^{h} n_i \leq \sum_{i=0}^{h} \left(\frac{24}{25}\right)^h n \leq n \cdot \sum_{i=0}^{\infty} \left(\frac{24}{25}\right)^i \leq 25n$$

wobei der letzte Schritt aus der Formel für die geometrische Reihe folgt. Diese Schranke und Analyse gelten gleichermaßen auch für den Speicherplatz.

Aus Theorem 6.26 folgt, dass die Anfragezeit linear in der maximalen Länge eines Pfades von der Wurzel zu einem Dreieck in T_0 ist, da der Outgrad konstant ist. Nach der Analyse oben ist die Länge des Pfades höchstens $h+1$ und liegt somit $O(\log n)$. □

Literaturverzeichnis

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: **Introduction to Algorithms**. The MIT Press, 3. Auflage, 2009.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: **Algorithmen - Eine Einführung.** De Gruyter Oldenbourg, 4. Auflage, 2013.

[3] Rolf Klein, Anne Driemel und Herman Haverkort: **Algorithmische Geometrie — Grundlagen, Methoden, Anwendungen**. Springer, 2022.

[4] Jon M. Kleinberg und Éva Tardos: **Algorithm design**. Addison-Wesley, 2005.

[5] D.E. Knuth: **The Art of Computer Programming, Volume III: Sorting and Searching**. Addison–Wesley, 1973.

[6] Michael Mitzenmacher und Eli Upfal: **Probability and Computing**. Cambridge University Press, 2005.

[7] Tim Roughgarden: **Algorithms Illuminated (Part 1): The Basics**. Algorithms Illuminated. Soundlikeyourself Publishing, LLC, 2017.

[8] Tim Roughgarden: **Algorithms Illuminated (Part 2): Graph Algorithms and Data Structures**. Algorithms Illuminated. Soundlikeyourself Publishing, LLC, 2018.

[9] Tim Roughgarden: **Algorithms Illuminated (Part 3): Greedy Algorithms and Dynamic Programming**. Algorithms Illuminated Series. Soundlikeyourself Publishing, LLC, 2019.

[10] Tim Roughgarden: **Algorithms Illuminated (Part 4): Algorithms for NP-Hard Problems**. Soundlikeyourself Publishing, LLC, 2020.

[11] Heiko Röglin: **Randomisierte und approximative Algorithmen**, Vorlesungsskript, Universität Bonn, Wintersemester 2011/12. `http://www.roeglin.org/teaching/WS2011/RandomisierteAlgorithmen/RandomisierteAlgorithmen.pdf`.

[12] Steven S. Skiena: **The Algorithm Design Manual**. Springer, 2. Auflage, 2008.