Algorithmen und Datenstrukturen

Steffen Börm

Stand 17. Juli 2014, 9 Uhr 30

Inhaltsverzeichnis

1 Einleitung **5**

2 Algorithmen und ihre Eigenschaften **7**
- 2.1 Beispiel: Suchen in Arrays . . . 7
- 2.2 Binäre Suche . . . 8
- 2.3 Beispiel: Sortieren von Arrays . . . 14
- 2.4 Mergesort . . . 17
- 2.5 Quicksort . . . 25
- 2.6 Landau-Notation . . . 36
- 2.7 Beispiel: Suchen in vollständig geordneten Mengen . . . 40
- 2.8 Korrektheit . . . 46
- 2.9 Zusammenfassung . . . 52

3 Grundlegende Datenstrukturen **53**
- 3.1 Listen . . . 53
- 3.2 Keller . . . 60
- 3.3 Warteschlange . . . 62
- 3.4 Bäume . . . 63
- 3.5 Balancierte Bäume . . . 74
- 3.6 Dynamische Datenstrukturen in Arrays . . . 86
- 3.7 Heapsort . . . 91

4 Graphen **101**
- 4.1 Definition eines Graphen . . . 101
- 4.2 Breitensuche . . . 104
- 4.3 Tiefensuche . . . 109
- 4.4 Optimale Pfade in gewichteten Graphen . . . 116
- 4.5 Verbesserte Fassung des Dijkstra-Algorithmus' . . . 124
- 4.6 Optimale Pfade zwischen allen Knoten . . . 128

5 Konstruktionsprinzipien **137**
- 5.1 Teile und herrsche: Karatsuba-Multiplikation . . . 137
- 5.2 Tiefensuche: Sudoku . . . 139
- 5.3 Dynamisches Programmieren: Rucksack-Problem . . . 141

Index **143**

1 Einleitung

Eine der wichtigsten Fragen der Informatik lautet „Wie bringe ich den Computer dazu, zu tun, was ich von ihm will?“ Die Antwort besteht darin, eine Rechenvorschrift zu formulieren, die zu dem gewünschten Ergebnis führt und aus einer Reihe von Schritten besteht, die so einfach sind, dass der Computer sie ausführen kann. Eine solche Rechenvorschrift bezeichnet man als *Algorithmus* (nach dem Mathematiker Abu Dscha'far Muhammad ibn Musa *al-Chwarizmi*, der im 9. Jahrhundert ein Buch mit Rechenverfahren veröffentlichte).

Eine grundlegende Frage ist natürlich, ob es überhaupt möglich ist, für eine gegebene Aufgabe einen Lösungsalgorithmus anzugeben. Man kann beweisen, dass es Aufgaben gibt, für die *kein* solcher Algorithmus existiert.

Im Rahmen dieser Vorlesung interessieren wir uns eher für Aufgaben, die sich mit dem Computer lösen lassen, für die also mindestens ein Algorithmus existiert. Bei derartigen Aufgaben stellt sich die Frage, ob der Algorithmus auch praktisch brauchbar ist, ob er also beispielsweise schnell genug arbeitet und ob sein Speicherbedarf sich in einem vernünftigen Rahmen hält.

Es hat sich gezeigt, dass in der Praxis bestimmte typische Aufgabenstellungen immer wieder auftreten, beispielsweise muss sehr häufig in einer größeren Datenmenge ein Eintrag gefunden werden, der bestimmten Bedingungen genügt. Für diese Aufgabenstellungen hat die Informatik eine ganze Reihe von Algorithmen entwickelt, die zum Teil sehr unterschiedliche Eigenschaften in Hinblick auf den Rechenaufwand aufweisen. Abhängig von der Architektur des Rechnersystems kann auch die Reihenfolge und Art der Rechenoperationen sich auf die Zeit auswirken, die der Algorithmus benötigt, beispielsweise kosten Divisionen meistens mehr Zeit als Additionen, und Speicherzugriffe auf benachbarte Adressen erfolgen oft schneller als solche auf weit voneinander entfernte.

Bei vielen Algorithmen ist für die Geschwindigkeit entscheidend, in welcher Form die zu verarbeitenden Daten vorliegen. Beispielsweise kann ein Suchalgorithmus sehr viel schneller arbeiten, wenn die zu durchsuchenden Daten sortiert sind. Deshalb bietet es sich an, Algorithmen immer im Zusammenhang mit den *Datenstrukturen* zu diskutieren, auf denen sie arbeiten. Das hat natürlich zur Folge, dass neben dem Rechenaufwand auch der Speicherbedarf berücksichtigt werden muss.

Beispielsweise legen viele Datenbankprogramme neben den eigentlichen Datensätzen auch mindestens einen sogenannten *Index* an, dabei handelt es sich um ein sortiertes Verzeichnis, mit dessen Hilfe sich schnell Datensätze nach bestimmten Kriterien auswählen lassen (etwa vergleichbar mit dem Stichwortverzeichnis eines Buchs). In diesem Fall ist abzuwägen, ob der erhöhte Speicherbedarf in einem sinnvollen Verhältnis zu der reduzierten Rechenzeit steht.

Dementsprechend ist es die Aufgabe einer Programmiererin oder eines Programmie-

rers, für eine gegebene Aufgabenstellung einen gut geeigneten Algorithmus auszuwählen und ihn so zu implementieren, dass er auf dem vorliegenden Rechnersystem möglichst effizient arbeitet.

Diese Aufgabe ist offenbar nur dann zu lösen, wenn man eine Reihe von typischen Algorithmen für typische Aufgabenstellungen kennt und weiß, welche Eigenschaften sie aufweisen und wie man sie geschickt umsetzt. Um auch neuen Aufgabentypen gewachsen zu sein, ist es sinnvoll, die allgemeinen Prinzipien zu kennen, mit denen sich Algorithmen und Datenstrukturen entwickeln lassen. Ziel der vorliegenden Vorlesung ist es, diese Kenntnisse zu vermitteln.

Das Suchen in Datenmengen und das Sortieren dieser Datenmengen sind Aufgabenstellungen, die in der Praxis immer wieder auftreten, deshalb werden wir uns ihnen relativ ausführlich widmen. Ein weiteres wichtiges Aufgabenfeld sind *Graphen*, mit deren Hilfe sich beispielsweise Verkehrsnetze beschreiben lassen, um kürzeste Verbindungen zwischen beliebigen Orten zu finden. Graphen sind allerdings so allgemein, dass sie sich bei der Behandlung überraschend vieler Aufgaben gewinnbringend nutzen lassen.

Auf der Seite der Konstruktionsprinzipien werden die *Rekursion*, insbesondere in Gestalt des *Teile-und-Herrsche-Ansatzes*, sowie das *dynamische Programmieren* im Mittelpunkt stehen. Für die Analyse der entwickelten Algorithmen werden wir einige grundlegende Techniken für Komplexitätsabschätzungen behandeln.

Danksagung

Ich bedanke mich bei Prof. Dr. Klaus Jansen für das Skript seiner Vorlesung „Algorithmen und Datenstrukturen“, das die Grundlage des vorliegenden Skripts bildet. Außerdem gilt mein Dank Marcin Pal, Jessica Gördes, Fabian Fröhlich, Sven Christophersen, Dirk Boysen, Sönke Fischer, Jonathan Schilling, Ingmar Knof, Paul Hein, Steffen Strohm, Malin Rau und Sebastian Rau für Korrekturen und Verbesserungsvorschläge.

2 Algorithmen und ihre Eigenschaften

Bevor wir daran gehen, einen Algorithmus zu implementieren, müssen wir entscheiden, welcher der in Frage kommenden Algorithmen für die gegebene Aufgabenstellung überhaupt die richtige Wahl ist. Dazu empfiehlt es sich, Kriterien zu entwickeln, mit denen sich die Eignung eines Algorithmus' beurteilen lässt, mit denen wir also entscheiden können, ob er für unsere Aufgabe gut oder schlecht geeignet ist.

Von zentraler Bedeutung sind dabei die Struktur der Aufgabe und die des Rechners, auf dem die Aufgabe gelöst werden soll.

2.1 Beispiel: Suchen in Arrays

Als erstes Beispiel untersuchen wir eine Aufgabenstellung, die in der Praxis relativ häufig auftritt: Wir suchen in einer gegebenen Datenmenge ein bestimmtes Element, beispielsweise ein Stichwort in einem Stichwortverzeichnis oder einen Namen in einem Adressbuch.

Damit wir unsere Ergebnisse möglichst allgemein anwenden können, formulieren wir die Aufgabenstellung abstrakt in der Sprache der Mathematik:

> Gegeben sind $n \in \mathbb{N}$, $y \in \mathbb{Z}$ sowie $x_0, x_1, \ldots, x_{n-1} \in \mathbb{Z}$.
> Gesucht ist ein $j \in \{0, \ldots, n-1\}$ mit $x_j = y$.

Wir beschränken uns zunächst auf den Fall, dass y sowie $x_0, \ldots, x_{n-1}$ ganze Zahlen sind, um die Verallgemeinerung unserer Ergebnisse kümmern wir uns später.

Wenn wir davon ausgehen, dass die Zahlen $x_0, \ldots, x_{n-1}$ in einem Array gespeichert sind, können wir die Aufgabe lösen, indem wir einfach jede Zahl mit y vergleichen und aufhören, falls der Vergleich positiv ausfällt. Eine mögliche Implementierung dieser *linearen Suche* ist in Abbildung 2.1 dargestellt.

Falls y in dem Array vorkommt, wird der zugehörige Index zurückgegeben. Anderenfalls erhalten wir den Wert -1, den das aufrufende Programm natürlich geeignet interpretieren muss.

Eine wichtige Eigenschaft eines Algorithmus' ist sicherlich die *Laufzeit*, also die Zeit, die er benötigt, um eine gegebene Aufgabe zu lösen. In unserem Fall können wir die Laufzeit grob abschätzen, indem wir zählen, wieviele Rechenoperationen höchstens ausgeführt werden. Da in der Programmiersprache C fast alle Operationen durch C-Operatoren wiedergegeben werden, zählen wir einfach die Operatoren.

Lemma 2.1 (Rechenaufwand) *Der Algorithmus* `linear_search` *benötigt höchstens* $4n + 2$ *Operationen.*

```c
int
linear_search(int n, int y, const int *x)
{
  int j;
  for(j=0; j<n; j++)
    if(y == x[j])
      return j;
  return -1;
}
```

Abbildung 2.1: Lineare Suche in einem Array

Beweis. In jeder Iteration der Schleife fallen nicht mehr als 4 Operationen an:

Ausdruck	Operationen
`j<n`	1
`j++`	1
`y == x[j]`	2

Da `j` in jedem Schleifendurchlauf inkrementiert wird und die Schleife spätestens abbricht, falls die Bedingung `j<n` nicht mehr gilt, wird die Schleife höchstens n-mal durchlaufen, so dass inklusive der Initialisierung `j=0` und der abschließenden Prüfung von `j<n` insgesamt nicht mehr als $4n + 2$ Operationen anfallen. ∎

2.2 Binäre Suche

In der Praxis werden wir häufig viele Suchoperationen in großen Datenmengen durchführen müssen, also sind wir daran interessiert, den Rechenaufwand möglichst zu reduzieren. Ein Ansatz besteht darin, die Zahlen $x_0, \ldots, x_{n-1}$ so zu organisieren, dass das Suchen einfacher wird.

Besonders elegant lässt sich das Problem lösen, wenn wir voraussetzen, dass die Zahlen *sortiert* sind, dass wir also nur die folgende vereinfachte Aufgabenstellung lösen müssen:

> Gegeben sind $n \in \mathbb{N}$, $y \in \mathbb{Z}$ sowie $x_0, \ldots, x_{n-1} \in \mathbb{Z}$ mit $x_0 \leq x_1 \leq \ldots \leq x_{n-1}$.
> Gesucht ist ein $j \in \{0, \ldots, n-1\}$ mit $x_j = y$.

Falls für ein $k \in \{0, \ldots, n-1\}$ die Beziehung $y < x_k$ gilt, ergibt sich aus

$$y < x_k \leq x_{k+1} \leq x_{k+2} \leq \ldots \leq x_{n-1}$$

bereits, dass das gesuchte j nur in der Menge $\{0, \ldots, k-1\}$ liegen kann. Wir können also mit einem einzigen Vergleich mehrere Indizes ausschließen.

Falls $y \not< x_k$ und $y \neq x_k$ gelten sollten, folgt bereits $x_k < y$, also

$$x_0 \leq x_1 \leq \ldots \leq x_k < y,$$

so dass wir in diesem Fall ebenfalls gleich mehrere Indizes ausschließen und unsere Suche auf $\{k+1, \ldots, n-1\}$ einschränken dürfen.

Besonders effizient wird diese Vorgehensweise, wenn wir das k als (abgerundete) Hälfte der Gesamtlänge des Arrays wählen.

Erinnerung 2.2 (Gauß-Klammer) *Das Ab- und Aufrunden einer Zahl beschreiben wir durch die* Gauß-Klammern*: Für $x \in \mathbb{R}$ definieren wir*

$$\begin{aligned} \lfloor x \rfloor &:= \max\{k \in \mathbb{Z} \ : \ k \leq x\}, && \textit{(untere Gauß-Klammer)} \\ \lceil x \rceil &:= \min\{k \in \mathbb{Z} \ : \ x \leq k\}. && \textit{(obere Gauß-Klammer)} \end{aligned}$$

Man kann leicht nachprüfen, dass diese beiden ganzen Zahlen auch durch die Ungleichungen

$$\begin{aligned} \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1, \\ \lceil x \rceil - 1 < x \leq \lceil x \rceil \end{aligned}$$

eindeutig bestimmt sind.

Mit der unteren Gauß-Klammer setzen wir $k = \lfloor n/2 \rfloor$, denn dann wird sowohl für $y < x_k$ als auch für $y > x_k$ die Menge der der noch zu prüfenden Indizes mindestens halbiert.

Wir können diesen Schritt wiederholen, bis entweder die Lösung gefunden oder die Menge der zu prüfenden Indizes leer ist. Diese Menge ist immer von der Form

$$M := \{a, a+1, \ldots, b-2, b-1\} \tag{2.1}$$

für $0 \leq a \leq b \leq n$. Die Menge m enthält genau $b-a$ Elemente und ist deshalb für $a = b$ leer. Wir wählen als „Mittelpunkt“ $k = \lfloor (b+a)/2 \rfloor$ und prüfen, ob $y = x_k$ gilt. Falls ja, sind wir fertig. Falls nein, prüfen wir, ob $y < x_k$ gilt. In diesem Fall wiederholen wir die Prozedur für die Menge

$$M_1 := \{a, a+1, \ldots, k-2, k-1\},$$

ansonsten für die Menge

$$M_2 := \{k+1, k+2, \ldots, b-2, b-1\}.$$

Da aus $a < b$ bereits $a \leq k < b$ folgt, enthalten die Mengen M_1 und M_2 weniger Elemente als M, so dass sicher gestellt ist, dass wir nach einer endlichen Anzahl von Schritten entweder j gefunden haben oder leere Mengen erreichen.

Der resultierende *binäre Suchalgorithmus* ist in Abbildung 2.3 zusammengefasst.

Natürlich stellt sich wieder die Frage nach dem Rechenaufwand. Angesichts der Tatsache, dass wir die Aufgabenklasse erheblich eingeschränkt haben, erwarten wir, dass der neue Algorithmus Vorteile gegenüber dem alten bietet.

Den Ausschlag gibt dabei die Anzahl der Schleifendurchläufe, denn der Aufwand pro Schleifendurchlauf lässt sich relativ einfach beschränken. Die Schleifenbedingung ist so

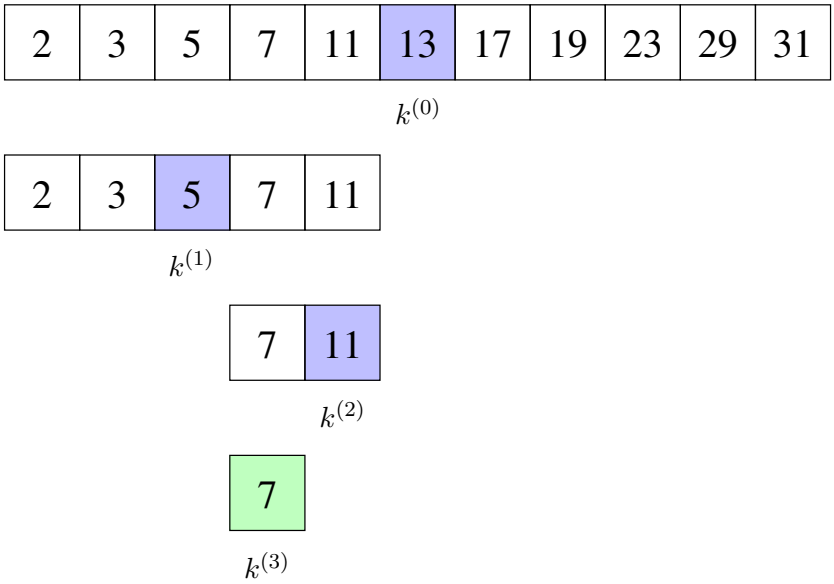


Abbildung 2.2: Binäre Suche in einem sortierten Array nach dem Wert 7 an Position 5.

```
1  int
2  binary_search(int n, int y, const int *x)
3  {
4    int a, b, k;
5    a = 0; b = n;
6    while(a < b) {
7      k = (b + a) / 2;
8      if(y == x[k])
9        return k;
10     else if(y < x[k])
11       b = k;
12     else
13       a = k + 1;
14   }
15   return -1;
16 }
```

Abbildung 2.3: Binäre Suche in einem Array

formuliert, dass die Schleife höchstens durchlaufen wird, solange in Zeile 6 die Bedingung `a < b` erfüllt ist, solange also die durch (2.1) definierte Menge nicht leer ist.

Um den Algorithmus mathematisch präzise analysieren zu können, müssen wir berücksichtigen, dass die Variablen `a` und `b` in jedem Schleifendurchlauf ihre Werte verändern können. Dazu bezeichnen wir mit $a^{(i)}$ und $b^{(i)}$ die Werte, die diese Variablen

nach dem i-ten Durchlauf aufweisen.

Die einzelnen Durchläufe bezeichnet man auch als *Iterationen* (von dem lateinischen Wort *iterare* für „wiederholen“).

Bevor die Schleife durchlaufen wird, gilt wegen Zeile 5 gerade

$$a^{(0)} = 0, \qquad b^{(0)} = n.$$

Falls während der i-ten Iteration die Bedingung in Zeile 10 erfüllt ist, erhalten wir

$$a^{(i)} = a^{(i-1)}, \qquad b^{(i)} = \lfloor (b^{(i-1)} + a^{(i-1)})/2 \rfloor,$$

anderenfalls

$$a^{(i)} = \lfloor (b^{(i-1)} + a^{(i-1)})/2 \rfloor + 1, \qquad b^{(i)} = b^{(i-1)}.$$

Die Frage ist, nach wievielen Iterationen $a^{(i)} = b^{(i)}$ gilt und damit die Bedingung in Zeile 6 dazu führt, dass die Schleife beendet wird.

Lemma 2.3 (Halbierung) *Seien $a, b \in \mathbb{N}$ mit $0 \leq a \leq b \leq n$ gegeben, sei $k = \lfloor (b + a)/2 \rfloor$. Dann gelten*

$$k - a \leq \frac{b-a}{2}, \qquad b - (k+1) \leq \frac{b-a}{2}.$$

Falls $b - a = 2^p - 1$ für ein $p \in \mathbb{N}$ gilt, folgt sogar $k - a = b - (k+1) = 2^{p-1} - 1$.

Beweis. Da wir k durch Abrunden definiert haben, gilt nach Erinnerung 2.2

$$k \leq \frac{b+a}{2} < k + 1,$$

so dass wir

$$\begin{aligned} k - a &\leq \frac{b+a}{2} - a = \frac{(b+a) - 2a}{2} = \frac{b-a}{2}, \\ b - (k+1) &< b - \frac{b+a}{2} = \frac{2b - (b+a)}{2} = \frac{b-a}{2} \end{aligned}$$

erhalten. Damit ist die erste Aussage bewiesen.

Wenden wir uns nun dem Sonderfall $b - a = 2^p - 1$ zu. Dann gilt

$$k = \left\lfloor \frac{b+a}{2} \right\rfloor = \left\lfloor \frac{2a + 2^p - 1}{2} \right\rfloor = \lfloor a + 2^{p-1} - 1/2 \rfloor = a + 2^{p-1} - 1$$

und wir erhalten

$$\begin{aligned} k - a &= a + 2^{p-1} - 1 - a = 2^{p-1} - 1, \\ b - (k+1) &= b - a - 2^{p-1} = 2^p - 1 - 2^{p-1} = 2^{p-1} - 1, \end{aligned}$$

also die zweite Aussage. ∎

Angewendet auf unser Beispiel bedeutet das Lemma, dass

$$b^{(i)} - a^{(i)} \leq \left\lfloor \frac{b^{(i-1)} - a^{(i-1)}}{2} \right\rfloor \tag{2.2}$$

für das Ergebnis des i-ten Durchlaufs gilt. Wir dürfen abrunden, da $b^{(i)}$ und $a^{(i)}$ in unserem Algorithmus immer ganze Zahlen sind. In jedem Schritt reduziert sich also der Abstand zwischen $a^{(i)}$ und $b^{(i)}$ um mindestens die Hälfte. Wir sind daran interessiert, herauszufinden, für welches i der Abstand zum ersten Mal gleich null ist, denn dann wird die Schleife nach dem i-ten Durchlauf beendet.

Indem wir (2.2) wiederholt anwenden, erhalten wir

$$b^{(i)} - a^{(i)} \leq \frac{b^{(0)} - a^{(0)}}{2^i} = \frac{n}{2^i}$$

und stehen vor der Aufgabe, das kleinste i zu finden, für das die rechte Seite kleiner als eins wird, für das also $n < 2^i$ gilt.

Erinnerung 2.4 (Dyadischer Logarithmus) *Für jede Zahl $x \in \mathbb{R}_{>0}$ existiert genau eine Zahl $y \in \mathbb{R}$ derart, dass $x = 2^y$ gilt. Diese Zahl y nennen wir den* dyadischen Logarithmus *von x und bezeichnen sie mit $\log_2(x)$. Der Logarithmus erfüllt die Gleichungen*

$$\log_2(1) = 0, \qquad \log_2(2) = 1, \qquad \log_2(xy) = \log_2 x + \log_2 y \qquad \text{für alle } x, y \in \mathbb{R}_{>0}$$

und ist eine streng monoton wachsende Funktion.

Wir bezeichnen mit $m^{(i)} := b^{(i)} - a^{(i)}$ die nach der i-ten Iteration verbliebene Größe der zu durchsuchenden Menge. Dann folgt aus (2.2) die Abschätzung

$$m^{(i)} \leq \lfloor m^{(i-1)}/2 \rfloor.$$

Wenn wir nun mit $I(m)$ die Anzahl der Iterationen bezeichnen, die höchstens für eine Menge der Größe $m \in \mathbb{N}$ erforderlich ist, erhalten wir die Beziehung

$$I(m) \leq \begin{cases} 1 & \text{falls } m = 1, \\ 1 + I(\lfloor m/2 \rfloor) & \text{ansonsten} \end{cases} \qquad \text{für alle } m \in \mathbb{N}, \tag{2.3}$$

denn für eine einelementige Menge ist nur eine Iteration vonnöten, während wir mit jeder Iteration die Menge mindestens halbieren. Unsere Aufgabe besteht darin, $I(m)$ abzuschätzen. Dafür verwenden wir das folgende etwas allgemeinere Lemma:

Lemma 2.5 (Rekurrenz) *Seien $\alpha, \beta \in \mathbb{N}_0$ gegeben und sei $f : \mathbb{N} \to \mathbb{N}_0$ eine Abbildung mit*

$$f(n) \leq \begin{cases} \alpha & \textit{falls } n = 1, \\ \beta + f(\lfloor n/2 \rfloor) & \textit{ansonsten} \end{cases} \qquad \textit{für alle } n \in \mathbb{N}. \tag{2.4}$$

Dann gilt

$$f(n) \leq \alpha + \beta \lfloor \log_2 n \rfloor \qquad \textit{für alle } n \in \mathbb{N}. \tag{2.5}$$

Beweis. Da wir die Induktionsvoraussetzung auf $\lfloor n/2 \rfloor$ anwenden wollen, müssen wir die Induktionsbehauptung etwas allgemeiner formulieren: Wir werden per Induktion über $\ell \in \mathbb{N}$ beweisen, dass für alle $n \in \{1, \ldots, \ell\}$ die Abschätzung (2.5) gilt.

Induktionsanfang. Sei $\ell = 1$. Dann folgt $n = 1$ und wir haben

$$f(n) = f(1) \leq \alpha = \alpha + \beta\lfloor \log_2 1 \rfloor = \alpha + \beta\lfloor \log_2 n \rfloor.$$

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass (2.5) für alle $n \in \{1, \ldots, \ell\}$ gilt.

Induktionsschritt. Wir müssen die Aussage nur für $n = \ell + 1$ nachweisen, denn für $n \leq \ell$ gilt sie nach Voraussetzung schon. Wegen $n > 1$ folgt aus (2.4) die Ungleichung

$$f(n) \leq \beta + f(\lfloor n/2 \rfloor),$$

und dank

$$\lfloor n/2 \rfloor = \lfloor (\ell + 1)/2 \rfloor = \lfloor \ell/2 + 1/2 \rfloor \leq \lfloor \ell/2 + \ell/2 \rfloor = \ell$$

dürfen wir auf $f(\lfloor n/2 \rfloor)$ nach Induktionsvoraussetzung (2.5) anwenden, um

$$\begin{aligned} f(n) &\leq \beta + f(\lfloor n/2 \rfloor) \leq \beta + \alpha + \beta\lfloor \log_2 \lfloor n/2 \rfloor \rfloor \\ &\leq \beta + \alpha + \beta\lfloor \log_2 n/2 \rfloor = \alpha + \beta\lfloor 1 + \log_2 n/2 \rfloor \\ &= \alpha + \beta\lfloor \log_2 2 + \log_2 n/2 \rfloor = \alpha + \beta\lfloor \log_2 n \rfloor \end{aligned}$$

zu erhalten. Das ist die gewünschte Abschätzung. ∎

Indem wir dieses Lemma mit $\alpha = \beta = 1$ auf das in (2.3) definierte $I(m)$ anwenden, erhalten wir die Ungleichung $I(m) \leq 1 + \lfloor \log_2 m \rfloor$. Durch Abzählen der angewendeten Operatoren erhalten wir eine Aufwandsabschätzung:

Satz 2.6 (Rechenaufwand) *Der Algorithmus* `binary_search` *benötigt höchstens* $13 + 10\lfloor \log_2 n \rfloor$ *Operationen.*

Beweis. Die Schleife in den Zeilen 6 bis 14 benötigt insgesamt nicht mehr als 10 Operationen.

Ausdruck	Operationen
`a < b`	1
`k = (b + a) / 2`	3
`y == x[k]`	2
`y < x[k]`	2
`b = k` oder `a = k+1`	≤ 2

Hinzu kommen 3 Operationen für die Initialisierungen `a = 0` und `b = n` sowie die abschließende Prüfung der Bedingung `a < b`.

Da wir mit $a^{(0)} = 0$ und $b^{(0)} = n$ beginnen, werden nicht mehr als $I(n)$ Iterationen durchgeführt, so dass insgesamt nicht mehr als $3 + 10I(n)$ Operationen anfallen.

Mit Lemma 2.5 folgt, dass wir nicht mehr als

$$3 + 10I(n) \leq 3 + 10(1 + \lfloor \log_2 n \rfloor) = 13 + 10\lfloor \log_2 n \rfloor$$

Operationen für den gesamten Algorithmus benötigen. ∎

Damit arbeitet die binäre Suche *wesentlich* schneller als die lineare Suche: Falls sich die Anzahl n der zu durchsuchenden Elemente verdoppelt, verdoppelt sich auch der Aufwand der linearen Suche, während für die binäre Suche lediglich 10 Operationen hinzukommen.

Bei $n = 1\,000\,000$ Elementen benötigt die lineare Suche ungefähr $4\,000\,000$ Operationen, während für die binäre Suche $13 + 10\lfloor \log_2 n \rfloor = 203$ Operationen genügen.

2.3 Beispiel: Sortieren von Arrays

Nachdem wir nun festgestellt haben, dass ein sortiertes Array wesentliche Vorteile gegenüber einem unsortierten bietet, stellt sich die Frage, wie man aus einem unsortierten Array ein sortiertes machen könnte.

Falls wir viele Suchoperationen in *demselben* Array durchführen müssen, könnten wir dann nämlich das Array zunächst sortieren und anschließend die vielen Suchoperationen mit dem effizienten binären Suchverfahren sehr schnell durchführen.

Das Umsortieren eines Arrays lässt sich mathematisch durch eine Abbildung beschreiben, die jedem Index des neuen Arrays einen des alten Arrays zuordnet:

Definition 2.7 (Permutation) *Sei $n \in \mathbb{N}$. Eine bijektive Abbildung*

$$\pi : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$$

nennen wir eine (n-stellige) Permutation.

Die Voraussetzung der Bijektivität in dieser Definition stellt sicher, dass bei einer Permutation kein Index verloren geht und kein Index doppelt auftritt. Eine Permutation π ordnet jedem Index $i \in \{0, \ldots, n-1\}$ der neuen Anordnung der Folge den Index $\pi(i)$ der alten Anordnung zu, so dass die umsortierte Folge gerade $x_{\pi(0)}, x_{\pi(1)}, \ldots, x_{\pi(n-1)}$ ist.

Präzise formuliert sieht unsere Aufgabenstellung dann wie folgt aus:

> Gegeben sind $n \in \mathbb{N}$ und $x_0, \ldots, x_{n-1} \in \mathbb{Z}$.
> Gesucht ist eine n-stellige Permutation π mit $x_{\pi(0)} \leq x_{\pi(1)} \leq \ldots x_{\pi(n-1)}$.

In der Praxis sind wir in der Regel nur an dem sortierten Array interessiert, nicht an der Permutation π, aus der es entstanden ist, deshalb werden wir unsere Algorithmen so formulieren, dass sie das ursprüngliche Array direkt mit seinem sortierten Gegenstück überschreiben. Falls wir an π interessiert sind, können wir es konstruieren, indem wir jeden Schritt des Algorithmus' geeignet protokollieren.

Widmen wir uns nun einem konkreten Sortieralgorithmus. Ein erster Ansatz für das Sortieren eines Arrays beruht auf der einfachen Idee, ein sortiertes Array aufzubauen, indem man mit einem leeren Array beginnt und der Reihe nach die Elemente des ursprünglichen Arrays in das neue Array einsortiert.

Falls die Zahlen $x_0, \ldots, x_{j-1}$ bereits sortiert sind und wir ein neues Element y einfügen wollen, können wir das tun, indem wir y anhängen und die Folge

$$x_0, \ldots, x_{j-3}, x_{j-2}, x_{j-1}, y$$

erhalten. Falls $x_{j-1} \leq y$ gilt, sind wir schon fertig.

Anderenfalls, also falls $x_{j-1} > y$ gilt, ist die Folge nicht sortiert. Dieses Problem können wir beheben, indem wir y um eine Position nach links rücken lassen und

$$x_0, \ldots, x_{j-3}, x_{j-2}, y, x_{j-1}$$

erhalten. Nun ist $y \leq x_{j-1}$ sichergestellt. Falls auch $x_{j-2} \leq y$ gilt, sind wir wieder fertig.

Ansonsten lassen wir y wieder eine Position nach links rücken und gelangen zu

$$x_0, \ldots, x_{j-3}, y, x_{j-2}, x_{j-1}.$$

Nach Konstruktion gilt $y \leq x_{j-2} \leq x_{j-1}$. Falls $x_{j-3} \leq y$ gilt, sind wir fertig, ansonsten fahren wir wie bereits beschrieben fort, bis y an seinem angemessenen Platz angekommen ist. In der Programmiersprache C können wir diesen Algorithmus wie folgt umsetzen, wenn wir annehmen, dass das Array `x` mindestens `j+1` Elemente enthält:

```
1  i = j;
2  while((i > 0) && (x[i-1] > y)) {
3    x[i] = x[i-1];
4    i--;
5  }
6  x[i] = y;
```

In diesem Programmfragment gibt die Variable `i` jeweils die aktuelle Position von y an. Falls `x[i-1] > y` gilt, wird `x[i-1]` nach rechts geschoben und `i` heruntergezählt. Die Schleife endet, falls entweder y am Anfang des Arrays angekommen oder korrekt einsortiert ist. Bei der Initialisierung `i = j` ist zu beachten, dass in C Arrays ausgehend von 0 numeriert sind, so dass `x[j]` das $(j+1)$-te Element ist. In Zeile 2 nutzen wir in der Bedingung `i > 0 && x[i-1] > y` aus, dass der logische Und-Operator `&&` in C so definiert ist, dass der zweite Teilausdruck `x[i-1] > y` nur ausgewertet wird, falls der erste `i > 0` wahr ist. Wir brauchen also nicht zu befürchten, dass wir auf das undefinierte Element `x[-1]` des Arrays zugreifen.

Dieser Ansatz wird dadurch besonders elegant, dass wir ihn fast ohne zusätzlichen Speicher umsetzen können, indem wir schrittweise das ursprüngliche Array von links nach rechts durch das bisher sortierte Array überschreiben. Wir müssen dazu lediglich eine weitere Variable `j` einführen, die angibt, wie groß das sortierte Array bereits ist.

Der resultierende Algorithmus wird als *Sortieren durch Einfügen* bezeichnet, im Englischen als *insertion sort*, und ist in Abbildung 2.5 zusammengefasst.

Wir dürfen mit `j=1` beginnen, da ein Array mit nur einem Element bereits sortiert ist. Natürlich sind wir auch in diesem Fall daran interessiert, den Rechenaufwand des Verfahrens abzuschätzen.

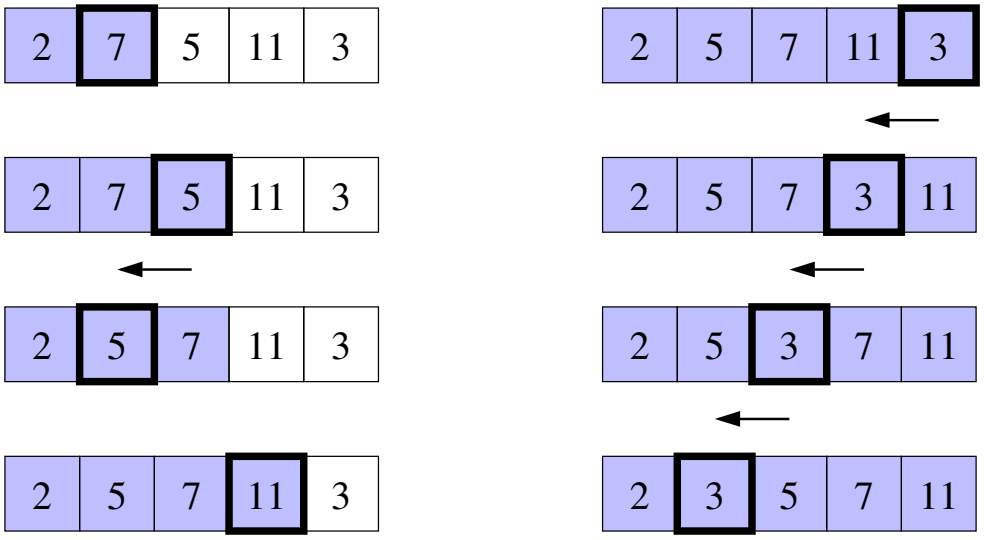


Abbildung 2.4: Sortieren durch Einfügen. Der jeweils aktive Teil des Arrays ist markiert, das einzusortierende Element ist umrandet.

```
1  void
2  insertionsort(int n, int *x)
3  {
4    int i, j;
5    int y;
6    for(j=1; j<n; j++) {
7      y = x[j];
8      i = j;
9      while((i > 0) && (x[i-1] > y)) {
10       x[i] = x[i-1];
11       i--;
12     }
13     x[i] = y;
14   }
15 }
```

Abbildung 2.5: Sortieren eines Arrays durch Einfügen

Lemma 2.8 (Rechenaufwand) *Der Algorithmus* `insertionsort` *benötigt höchstens* $5n^2 + 4n - 7$ *Operationen.*

Beweis. Wir untersuchen zunächst die innere Schleife in den Zeilen 9 bis 12. Eine Iteration

dieser Schleife benötigt 10 Operationen:

Ausdruck	Operationen
`(i > 0) && (x[i-1] > y)`	5
`x[i] = x[i-1]`	4
`i--`	1

Da die Variable `i` mit `i=j` beginnt und in jedem Schritt heruntergezählt wird, wird die Schleife höchstens j-mal durchlaufen. Die abschließende Überprüfung `(i > 0) && ...` erfordert nur zwei Operationen, da nur der linke Term ausgewertet wird. Also ist der Rechenaufwand für die Zeilen 9 bis 12 durch $10j + 2$ beschränkt.

Für die äußere Schleife in den Zeilen 6 bis 14 fallen in jeder Iteration zusätzliche 7 Operationen an:

Ausdruck	Operationen
`j<n`	1
`j++`	1
`y = x[j]`	2
`i = j`	1
`x[i] = y`	2

Hinzu kommen die Initialisierung `j=1` und die abschließende Überprüfung von `j<n`.

Da wir mit $j = 1$ anfangen, wird diese Schleife nur $(n-1)$-mal durchlaufen, so dass wir insgesamt nicht mehr als

$$\begin{aligned} 2 + \sum_{j=1}^{n-1}(10j + 2 + 7) &= 2 + 9(n-1) + 10\sum_{j=1}^{n-1} j = 2 + 9(n-1) + 10\frac{n(n-1)}{2} \\ &= 2 + 9(n-1) + 5n(n-1) = 2 + 9n - 9 + 5n^2 - 5n \\ &= 5n^2 + 4n - 7 \end{aligned}$$

Operationen benötigen. ∎

2.4 Mergesort

Ein Aufwand von ungefähr $5n^2$ Operationen für das Sortieren eines Arrays ist relativ hoch. Falls wir nur wenige Suchoperationen durchführen, kann der Gesamtaufwand für das Sortieren und das Suchen höher als der Aufwand der linearen Suche werden. Deshalb empfiehlt es sich, die Frage nach effizienteren Sortieralgorithmen zu behandeln.

Ein häufig erfolgreicher Ansatz ist das Prinzip „Teile und herrsche“ (lateinisch *divide et impera*, englisch *divide and conquer*), bei dem man ein großes Problem in kleinere Probleme zerlegt, die hoffentlich einfacher zu lösen sind, um dann aus deren Lösungen eine Lösung des Gesamtproblems zu konstruieren.

Wir können ein $m \in \{0, \ldots, n-1\}$ wählen und die zu sortierenden Zahlen

$$x_0, x_1, \ldots, x_{m-1}, x_m, \ldots, x_{n-1}$$

in zwei Teilfolgen

$$x_0, x_1, \ldots, x_{m-1} \qquad x_m, x_{m+1}, \ldots, x_{n-1}$$

zerlegen und beide separat sortieren. Wenn wir die dabei entstehenden sortierten Zahlenfolgen mit $y_0, \ldots, y_{m-1}$ und $z_0, \ldots, z_{n-m-1}$ bezeichnen, gilt also

$$\begin{aligned} y_0 \leq y_1 \leq \ldots \leq y_{m-1}, \qquad & \{x_0, x_1, \ldots, x_{m-1}\} = \{y_0, y_1, \ldots, y_{m-1}\}, \\ z_0 \leq z_1 \leq \ldots \leq z_{n-m-1}, \qquad & \{x_m, x_{m+1}, \ldots, x_{n-1}\} = \{z_0, z_1, \ldots, z_{n-m-1}\}. \end{aligned}$$

Unsere Aufgabe besteht nun darin, aus den beiden sortierten Folgen eine sortierte Gesamtfolge zusammenzusetzen. Das erste Element dieser Gesamtfolge $\hat{x}_0, \ldots, \hat{x}_{n-1}$ muss das Minimum aller Zahlen sein, es muss also

$$\begin{aligned} \hat{x}_0 &= \min\{x_0, \ldots, x_{m-1}, x_m, \ldots, x_{n-1}\} \\ &= \min\{y_0, \ldots, y_{m-1}, z_0, \ldots, z_{n-m-1}\} \\ &= \min\{\min\{y_0, \ldots, y_{m-1}\}, \min\{z_0, \ldots, z_{n-m-1}\}\} \\ &= \min\{y_0, z_0\} \end{aligned}$$

gelten. Im letzten Schritt nutzen wir aus, dass die beiden Teilmengen bereits sortiert sind, so dass y_0 und z_0 jeweils die minimalen Elemente sein müssen. Das Minimum der Gesamtmenge können wir also mit einem einzigen Vergleich ermitteln.

Wir übernehmen dieses Minimum in unser Ergebnis und streichen es aus der Folge, aus der es stammt. Das zweitkleinste Element der Menge, also den zweiten Eintrag des Ergebnisses, können wir bestimmen, indem wir das kleinste Element der verbliebenen Teilfolgen berechnen.

In dieser Weise können wir fortfahren, bis das Ergebnis vollständig konstruiert wurde. Diese Vorgehensweise lässt sich in dem folgenden C-Programmfragment zusammenfassen:

```
1  j = 0; k = 0;
2  for(i=0; i<n; i++)
3    if(j == m || (k+m < n && y[j] > z[k])) {
4      x[i] = z[k]; k++;
5    }
6    else {
7      x[i] = y[j]; j++;
8    }
```

Hier bezeichnet `m` die Länge des Arrays `y`. Die Variablen `j` und `k` geben die Position des ersten Elements des Rests der Arrays `y` und `z` an. Falls `y` keine weiteren Elemente mehr enthält, erkennbar an `j == m`, oder falls `y[j] > z[k]` gilt, ist das Minimum der Rest-Arrays `z[k]`, also kopieren wir es in das Ergebnis und zählen `k` weiter. Ansonsten kopieren wir das Element aus `y` und zählen `j` weiter.

Auch an dieser Stelle ist wieder wichtig, dass der logische Oder-Operator `||` sein zweites Argument nur auswertet, falls das erste Argument gleich null (im C-Sinn also logisch „falsch“) ist: Auf `y[j]` wird nur zugegriffen, solange `j` sinnvolle Werte aufweist.

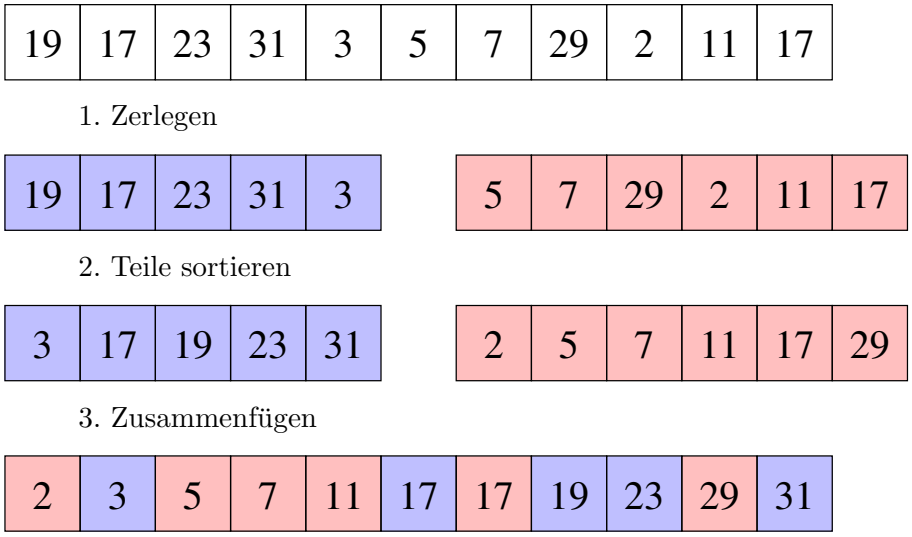


Abbildung 2.6: Prinzip des Mergesort-Algorithmus'.

Wir können also aus zwei bereits sortierten Arrays relativ einfach ein sortiertes Gesamtarray zusammensetzen. Es stellt sich nur die Frage, wie wir die beiden Teilarrays sortieren können. Ein einfacher Ansatz ist die *Rekursion*: Wir rufen die Sortierfunktion für die beiden Teilarrays auf. Solange wir sicherstellen, dass beide Teilarrays streng kleiner als das ursprüngliche Array sind, gelangen wir irgendwann zu einem Array der Länge eins und brauchen keinen weiteren Aufruf. Diese Forderung können wir leicht erfüllen, indem wir wie bei der binären Suche vorgehen und das Array jeweils halbieren, also $m = \lfloor n/2 \rfloor$ verwenden. Dieses Verfahren trägt den englischen Namen *Mergesort*, von *to merge*, dem englischen Wort für „zusammenfügen". Eine erste Fassung ist in Abbildung 2.7 zusammengefasst.

Diese Implementierung hat den Nachteil, dass fast das Doppelte des für das Array `x` erforderlichen Speichers zusätzlich als Hilfsspeicher angefordert werden muss: Als Beispiel untersuchen wir den besonders einfachen Fall $n = 2^p$ für ein $p \in \mathbb{N}_0$ und bezeichnen mit $M(n)$ die Anzahl an Array-Elementen, die bei einem Aufruf der Funktion `mergesort_naive` *insgesamt* zusätzlich benötigt werden. Das sind für $n > 1$ die n Elemente, die wir mit `malloc` für `y` und `z` anfordern, zuzüglich der $M(n/2)$, die in den beiden rekursiven Aufrufen angefordert werden. Da jeder Aufruf seinen Hilfsspeicher mit `free` auch wieder freigibt, sobald er ihn nicht mehr braucht, müssen wir $M(n/2)$ nicht doppelt zählen und erhalten

$$M(n) = \begin{cases} 0 & \text{falls } n = 1, \\ n + M(n/2) & \text{ansonsten} \end{cases} \qquad \text{für alle } n = 2^p,\ p \in \mathbb{N}_0.$$

Mit einer einfachen Induktion können wir nachprüfen, dass diese Rekurrenzformel gerade die Lösung $M(n) = 2(n-1)$ besitzt. Für allgemeine Werte von n ist die Menge des

```
1  void
2  mergesort_naive(int n, int *x)
3  {
4    int *y, *z;
5    int m;
6    int i, j, k;
7    if(n > 1) {
8      m = n / 2;
9      y = (int *) malloc(sizeof(int) * m);
10     z = (int *) malloc(sizeof(int) * (n-m));
11     for(j=0; j<m; j++) y[j] = x[j];
12     for(k=0; k<n-m; k++) z[k] = x[m+k];
13     mergesort_naive(m, y);
14     mergesort_naive(n-m, z);
15     j = 0; k = 0;
16     for(i=0; i<n; i++)
17       if(j == m || (k+m < n && y[j] > z[k])) {
18         x[i] = z[k]; k++;
19       }
20       else {
21         x[i] = y[j]; j++;
22       }
23     free(z); free(y);
24   }
25 }
```

Abbildung 2.7: Mergesort mit unnötig hohem Speicherbedarf

Hilfsspeichers durch eine ähnliche Formel gegeben:

Übungsaufgabe 2.9 (Speicherbedarf) *Der Bedarf an Hilfsspeicher erfüllt die Gleichung*

$$M(n) = \begin{cases} 0 & \text{falls } n = 1, \\ n + M(\lceil n/2 \rceil) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N}.$$

Beweisen Sie, dass

$$2(n-1) \leq M(n) \leq 2(n-1) + \lceil \log_2 n \rceil \qquad \text{für alle } n \in \mathbb{N}$$

gilt. Hinweis: Für alle $n \in \mathbb{N}$ *gilt* $\lceil \log_2(\lceil n/2 \rceil) \rceil = \lceil \log_2(n) \rceil - 1$.

Wir können den Algorithmus wesentlich eleganter gestalten, indem wir ausnutzen, dass die ursprünglichen Werte des Arrays `x` keine Rolle mehr spielen, sobald die Hilfsarrays

y und z gefüllt wurden. Wir können also x selber als Hilfsspeicher für die rekursiven Aufrufe „missbrauchen“. Dazu modifizieren wir die Funktion so, dass der zu verwendende Hilfsspeicher explizit als Parameter übergeben wird, und erhalten die in Abbildung 2.8 dargestellte Fassung.

```
1  void
2  mergesort(int n, int *x, int *h)
3  {
4    int *y, *z, *hy, *hz;
5    int m;
6    int i, j, k;
7    if(n > 1) {
8      m = n / 2;
9      y = h;   z = h + m;
10     hy = x;  hz = x + m;
11     for(i=0; i<n; i++) h[i] = x[i];
12     mergesort(m, y, hy);
13     mergesort(n-m, z, hz);
14     j = 0; k = 0;
15     for(i=0; i<n; i++)
16       if(j == m || (k+m < n && y[j] > z[k])) {
17         x[i] = z[k]; k++;
18       }
19       else {
20         x[i] = y[j]; j++;
21       }
22   }
23 }
```

Abbildung 2.8: Mergesort mit explizit angelegtem Hilfsspeicher

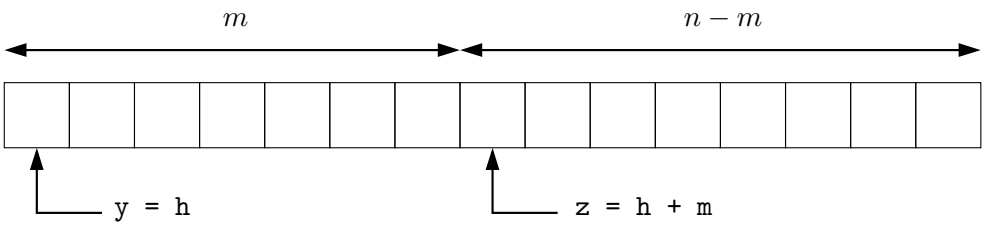


Abbildung 2.9: Aufteilung des Hilfsarrays h der Länge n in das Teilarray y der Länge m und das Teilarray z der Länge $n - m$ per Zeiger-Arithmetik.

Neben der Array-Länge `n` und dem zu sortierenden Array `x` erwartet diese Funktion auch ein Array `h` derselben Größe, das als Hilfsspeicher verwendet werden kann. Mit Hilfe der in C möglichen Zeiger-Arithmetik wird durch `y = h` und `z = h + m` das Hilfs-Array `h` in zwei Arrays der Größe `m` und `n-m` zerlegt: Das erste Array `y` beginnt an derselben Stelle wie `h`, das zweite `z` an der durch `h + m` gegebenen Position des Eintrags `h[m]`, der dann gerade `z[0]` entspricht. Die Zerlegung ist in Abbildung 2.9 illustriert.

Die Analyse eines rekursiven Algorithmus' ist etwas schwieriger als die eines iterativen: Bei einem iterativen Algorithmus genügt es, die Anzahl der Operationen pro Iteration zu zählen und eine Schranke für die maximale Anzahl der Iterationen anzugeben. Bei einem rekursiven Algorithmus müssen wir die Anzahl der Operationen berücksichtigen, die in den rekursiven Aufrufen auftreten, und diese rekursiven Aufrufe können weitere rekursive Aufrufe nach sich ziehen.

Allerdings lässt sich meistens relativ einfach eine *Rekurrenzformel* für den Rechenaufwand gewinnen, indem wir den Rechenaufwand als mathematische Funktion R aufassen, die einer Problemgröße n jeweils die Anzahl der Operationen $R(n)$ zuordnet, die für das Lösen eines Problems dieser Größe erforderlich sind. Solange für die Behandlung eines Problems der Größe n nur rekursive Aufrufe für Probleme kleinerer Größe auftreten, lässt sich dann die Formel mit Hilfe einer Induktion auflösen und so eine Abschätzung für den Aufwand $R(n)$ gewinnen.

Mit diesem Ansatz brauchen wir nur die Operationen zu zählen, die *mit Ausnahme der rekursiven Aufrufe* anfallen. Den Aufwand der rekursiven Aufrufe können wir mit Hilfe der Funktion R ausdrücken und so einen Ausdruck für den Gesamtaufwand erhalten, der sich mathematisch analysieren lässt. In unserem Beispiel führt dieser Ansatz zu der folgenden Aussage:

Lemma 2.10 (Rechenaufwand) *Sei $R(n)$ die Anzahl der Operationen, die die Funktion* `mergesort` *für ein Array der Länge n benötigt. Dann gilt*

$$R(n) \leq \begin{cases} 1 & \text{falls } n \leq 1, \\ 18 + 19n + R(\lfloor n/2 \rfloor) + R(n - \lfloor n/2 \rfloor) & \text{ansonsten} \end{cases} \quad \text{für alle } n \in \mathbb{N}_0. \quad (2.6)$$

Beweis. Für $n \leq 1$ wird lediglich die Bedingung `n > 1` ausgewertet, so dass

$$R(1) = 1$$

gilt. Für $n > 1$ summieren wir die Operationen der einzelnen Zeilen auf:

Zeile	Operationen
7	1
8	2
9	3
10	3
11	$2 + 5n$
12	$1 + R(m)$

Zeile	Operationen
13	$2 + R(n - m)$
14	2
15	$2 + 2n$
16	$8n$
17 oder 20	$4n$

In den Zeilen 12 und 13 ist zu beachten, dass auch die Funktionsaufrufe als Operationen gezählt werden. Insgesamt erhalten wir so

$$R(n) \leq 18 + 19n + R(m) + R(n-m),$$

wobei $m = \lfloor n/2 \rfloor$ gilt. Das ist die gewünschte Aussage. ∎

Die Rekurrenzformel (2.6) beschreibt den Rechenaufwand unseres Algorithmus' nur indirekt. Um eine explizite Formel zu erhalten, die sich beispielsweise mit der vergleichen lässt, die wir für `insertionsort` in Lemma 2.8 hergeleitet haben, müssen wir die Rekurrenzformel weiter analysieren. In unserem Fall lässt sich diese Aufgabe mit einem einfachen Induktionsbeweis lösen:

Lemma 2.11 (Rekurrenz) *Seien $\alpha, \beta, \gamma \in \mathbb{N}_0$ gegeben und sei $f : \mathbb{N} \to \mathbb{N}_0$ eine Abbildung mit*

$$f(n) \leq \begin{cases} \alpha & \text{falls } n = 1, \\ \beta + \gamma n + f(\lfloor n/2 \rfloor) + f(n - \lfloor n/2 \rfloor) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N}. \tag{2.7}$$

Dann gilt

$$f(n) \leq \alpha n + \beta(n-1) + \gamma n \lceil \log_2 n \rceil \qquad \text{für alle } n \in \mathbb{N}. \tag{2.8}$$

Beweis. Wie schon im Beweis des Lemmas 2.5 beweisen wir per Induktion über $\ell \in \mathbb{N}$, dass die Aussage (2.8) für alle $n \in \{1, \ldots, \ell\}$ gilt.

Induktionsanfang. Für $\ell = 1$ ist nur $n = 1$ zu untersuchen. Wegen $\log_2 1 = 0$ gilt bereits

$$f(n) = f(1) \leq \alpha = \alpha + \beta(1-1) + \gamma \lceil \log_2 1 \rceil = \alpha n + \beta(n-1) + \gamma n \lceil \log_2 n \rceil.$$

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass (2.8) für alle $n \in \{1, \ldots, \ell\}$ gilt.

Induktionsschritt. Wir müssen die Aussage für $n = \ell + 1$ nachweisen. Zur Abkürzung setzen wir

$$m := \lfloor n/2 \rfloor.$$

Wegen

$$\begin{aligned} m = \lfloor n/2 \rfloor = \lfloor \ell/2 + 1/2 \rfloor &\leq \lfloor \ell/2 + \ell/2 \rfloor = \ell, \\ n - m = n - \lfloor n/2 \rfloor = n - \lfloor \ell/2 + 1/2 \rfloor &\leq n - \lfloor 1/2 + 1/2 \rfloor = n - 1 = \ell \end{aligned}$$

dürfen wir die Induktionsvoraussetzung anwenden, um

$$\begin{aligned} f(m) &\leq \alpha m + \beta(m-1) + \gamma m \lceil \log_2(m) \rceil, \\ f(n-m) &\leq \alpha(n-m) + \beta(n-m-1) + \gamma(n-m) \lceil \log_2(n-m) \rceil \end{aligned}$$

zu erhalten. Mit (2.7) und $n = \ell + 1 > 1$ folgt daraus

$$f(n) \leq \beta + \gamma n + f(m) + f(n-m)$$

$$\begin{aligned}
&\leq \beta + \gamma n \\
&\quad + \alpha m + \beta(m-1) + \gamma m\lceil\log_2(m)\rceil \\
&\quad + \alpha(n-m) + \beta(n-m-1) + \gamma(n-m)\lceil\log_2(n-m)\rceil \\
&= \alpha n + \beta(n-1) + \gamma(n + m\lceil\log_2(m)\rceil + (n-m)\lceil\log_2(n-m)\rceil).
\end{aligned}$$

Damit wir die Terme in der letzten Klammer zusammenfassen können, wollen wir

$$\lceil\log_2(m)\rceil \leq \lceil\log_2(n)\rceil - 1, \qquad \lceil\log_2(n-m)\rceil \leq \lceil\log_2(n)\rceil - 1$$

beweisen. Die erste Ungleichung erhalten wir direkt aus der Definition

$$\lceil\log_2(m)\rceil \leq \lceil\log_2(n/2)\rceil = \lceil\log_2(n) - 1\rceil = \lceil\log_2(n)\rceil - 1.$$

Für die zweite Ungleichung gehen wir von

$$\lceil\log_2(n-m)\rceil = \lceil\log_2(n - \lfloor n/2\rfloor)\rceil$$

aus. Falls n eine gerade Zahl ist, folgt unmittelbar

$$\lceil\log_2(n-m)\rceil = \lceil\log_2(n - n/2)\rceil = \lceil\log_2(n/2)\rceil = \lceil\log_2(n)\rceil - 1.$$

Falls dagegen n eine ungerade Zahl ist, haben wir $m = (n-1)/2$ und deshalb

$$\lceil\log_2(n-m)\rceil = \lceil\log_2(n - n/2 + 1/2)\rceil = \lceil\log_2((n+1)/2)\rceil = \lceil\log_2(n+1)\rceil - 1.$$

Wir setzen $p := \lceil\log_2(n)\rceil$, so dass $2^{p-1} < n \leq 2^p$ gilt. Wegen $n = \ell + 1 \geq 2$ muss $p \geq 1$ gelten, so dass 2^p eine gerade Zahl ist. Da n ungerade ist, folgt $n < 2^p$, also $2^{p-1} < n + 1 \leq 2^p$ und somit $\lceil\log_2(n+1)\rceil = \lceil\log_2(n)\rceil$.

Insgesamt erhalten wir

$$\begin{aligned}
f(n) &\leq \alpha n + \beta(n-1) + \gamma(n + m\lceil\log_2(m)\rceil + (n-m)\lceil\log_2(n-m)\rceil) \\
&\leq \alpha n + \beta(n-1) + \gamma(n + m(\lceil\log_2(n)\rceil - 1) + (n-m)(\lceil\log_2(n)\rceil - 1)) \\
&= \alpha n + \beta(n-1) + \gamma(n + n(\lceil\log_2(n)\rceil - 1)) \\
&= \alpha n + \beta(n-1) + \gamma n\lceil\log_2(n)\rceil
\end{aligned}$$

und haben das gewünschte Ergebnis erreicht. ∎

Durch Kombination der in Lemma 2.10 gewonnenen Rekurrenzformel mit dem in Lemma 2.11 erhaltenen allgemeinen Resultat erhalten wir die gewünschte explizite Abschätzung des Rechenaufwands:

Satz 2.12 (Rechenaufwand) *Der Algorithmus* `mergesort` *benötigt höchstens* $19n + 19n\lceil\log_2 n\rceil$ *Operationen.*

Beweis. Nach Lemma 2.10 erfüllt der Rechenaufwand $R(n)$ die Bedingung (2.7) mit $\alpha = 1$, $\beta = 18$ und $\gamma = 19$, so dass wir mit Lemma 2.11 die Abschätzung

$$R(n) \leq n + 18(n-1) + 19n\lceil \log_2 n \rceil < 19n + 19n\lceil \log_2 n \rceil$$

erhalten. ∎

Für großes n ist der Mergesort-Algorithmus wesentlich effizienter als das Sortieren durch Einfügen, beispielsweise benötigt Mergesort für ein Array der Länge $n = 1\,000\,000$ nicht mehr als

$$19\,000\,000 + 19\,000\,000 \cdot 20 = 399\,000\,000 \text{ Operationen},$$

während wir bei Sortieren durch Einfügen mit ungefähr

$$5\,000\,000\,000\,000 \text{ Operationen}$$

rechnen müssen. Je nach Computer könnten wir also durch einen geschickt gewählten Algorithmus die Rechenzeit theoretisch um einen Faktor von ungefähr 12 000 verkürzen.

In der Praxis treten derartige Faktoren eher selten auf, weil beispielsweise Zugriffe auf den Hauptspeicher häufig dazu führen, dass der Prozessor nicht seine theoretische Spitzenleistung erreicht. Auf einem Intel Core™ i7-2600K beispielsweise benötigt `insertionsort` für $n = 1\,000\,000$ Elemente ungefähr 120 Sekunden, während `mergesort` nach ungefähr $0,04$ Sekunden fertig ist. Real wird das Sortieren also lediglich um einen Faktor von 3 000 beschleunigt.

2.5 Quicksort

Der Mergesort-Algorithmus benötigt zwar für große Arrays in der Regel wesentlich weniger Zeit als das Sortieren durch Einfügen, allerdings müssen wir ihm dafür auch zusätzlichen Speicher zur Verfügung stellen. Ideal wäre natürlich ein Sortierverfahren, das eine vergleichbar hohe Geschwindigkeit erreicht, aber ohne oder zumindest mit wesentlich weniger Hilfsspeicher auskommt.

Im Mergesort-Verfahren wird der Hilfsspeicher benötigt, um das Gesamtergebnis aus den Teilergebnissen zusammenzusetzen. Wenn wir auf den Hilfsspeicher verzichten wollen, könnten wir versuchen, diesen Schritt zu vermeiden, indem wir den Algorithmus so arrangieren, dass das Zusammensetzen entfallen kann. Das ist offenbar nur möglich, wenn alle Einträge des linken Teilarrays kleiner oder gleich den Einträgen des rechten Teilarrays sind, wenn also

$$x_i \leq x_j \qquad \text{für alle } i \leq m \leq j \tag{2.9}$$

gilt, wobei $m \in \{0, \ldots, n-1\}$ wieder die Länge des ersten Teilarrays angibt. Diese Voraussetzung bedeutet *nicht*, dass die Folge $x_0, x_1, \ldots, x_{n-1}$ bereits sortiert sein muss, denn beispielsweise für $m = 5$ wird keine Aussage darüber getroffen, ob $x_1 \leq x_3$ gilt.

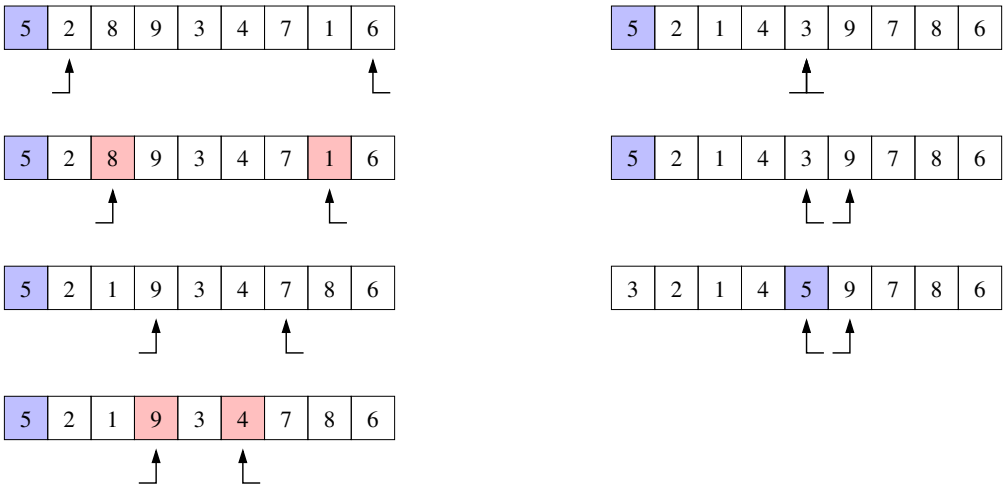


Abbildung 2.10: Umsortieren des Arrays im Quicksort-Algorithmus. Das Pivot-Element ist blau markiert, die Elemente, an denen die inneren `while`-Schleifen abbrechen, weil getauscht werden muss, sind rot hinterlegt.

Aufgrund dieser Tatsache lässt sich die Eigenschaft (2.9) wesentlich leichter als eine vollständige Sortierung des Arrays erreichen: Wir wählen ein Element $y \in \mathbb{Z}$, üblicherweise als *Pivot-Element* oder seltener als *Teiler* bezeichnet, und sortieren alle Elemente, die kleiner oder gleich y sind, an den Anfang des Arrays, während die restlichen Elemente an dessen Ende verschoben werden. Für dieses Umsortieren ist kaum Hilfsspeicher erforderlich, wenn wir die Teilarrays von links und rechts aufeinander zu wachsen lassen:

```
1   m = 0; j = n-1;
2   while(m <= j) {
3     while((m <= j) && (x[m] <= y))
4       m++;
5     while((m <= j) && (y < x[j]))
6       j--;
7     if(m < j) {
8       z = x[m]; x[m] = x[j]; x[j] = z;
9       m++; j--;
10    }
11  }
```

Bei der Behandlung der Indizes `m` und `j` empfiehlt es sich, Vorsicht walten zu lassen, weil ansonsten möglicherweise Sonderfälle (beispielsweise `x[0]` minimales oder maximales Element) nicht korrekt behandelt werden.

Dieses Programmfragment stellt sicher, dass in Zeile 7 jeweils

$$x_k \leq y \qquad \text{für alle } k \in \{0, \ldots, m-1\}, \tag{2.10a}$$
$$y < x_k \qquad \text{für alle } k \in \{j+1, \ldots, n-1\} \tag{2.10b}$$

gilt. Da die inneren `while`-Schleifen nur verlassen werden, wenn entweder $m > j$ oder $x_m > y$ beziehungsweise $y \geq x_j$ gilt, bleibt die obige Einschaft auch in Zeile 10 noch erhalten, nachdem x_m und x_j die Plätze getauscht haben.

Die äußere `while`-Schleife wird verlassen, sobald $m > j$ gilt. Dann haben wir auch $m \geq j + 1$ (bei genauerer Betrachtung sogar $m = j + 1$), und wegen (2.10) folgt bereits

$$x_k \leq y \qquad \text{für alle } k \in \{0, \ldots, m-1\},$$
$$y < x_k \qquad \text{für alle } k \in \{m, \ldots, n-1\}.$$

Damit ist unser Array in zwei Teilarrays der Längen m und $n - m$ zerlegt, die wir unabhängig voneinander sortieren können, um das sortierte Gesamtarray zu erhalten.

Es stellt sich die Frage nach dem Pivot-Element. Für den Augenblick wählen wir den einfachsten Ansatz und verwenden $y = x_0$. Nachdem die äußere `while`-Schleife abgeschlossen ist, wissen wir, dass genau m Elemente kleiner oder gleich y in unserem Array enthalten sind, also wissen wir auch, dass $x_0 = y$ in dem sortierten Array an der Stelle $m - 1$ stehen darf. Indem wir x_0 und x_{m-1} vertauschen, stellen wir sicher, dass zumindest das Pivot-Element an seiner endgültigen Stelle steht. Die restlichen $m - 1$ Elemente, die kleiner oder gleich y sind, und die $n - m$ Elemente, die echt größer sind, können wir nun durch rekursive Aufrufe unseres Algorithmus' behandeln. Da $m \geq 1$ durch unsere Wahl des Pivot-Elements sichergestellt ist, wird die Array-Größe mit jedem rekursiven Aufruf reduziert, so dass der resultierende Algorithmus sich nicht endlos selber aufrufen kann. Er trägt den Namen *Quicksort* und ist in Abbildung 2.11 dargestellt.

Für die Effizienz des Quicksort-Algorithmus' ist die Wahl des Pivot-Elements von zentraler Bedeutung, denn sie entscheidet darüber, wie groß die Teilarrays werden.

Besonders ungünstig ist der Fall, in dem das ursprüngliche Array *absteigend* sortiert ist, in dem also

$$x_0 > x_1 > x_2 > \ldots > x_{n-1}$$

gilt. Unser Algorithmus wählt $y = x_0$ und die erste innere `while`-Schleife in den Zeilen 10 und 11 läuft, bis `m>j` gilt. Es folgt $m = n$. Der Algorithmus befördert dann x_0 an die korrekte Stelle, nämlich in den letzten Eintrag des Arrays, und ruft sich in Zeile 20 rekursiv auf, um die verbliebenen $n - 1$ Einträge zu sortieren. Da wir x_0 und x_{n-1} vertauscht haben, steht jetzt das *kleinste* Element des Arrays an erster Stelle, so dass diesmal die zweite innere `while`-Schleife in den Zeilen 12 und 13 läuft, bis $j = 0$ gilt. Es folgt ein rekursiver Aufruf in Zeile 21, um die nun noch übrigens $n - 2$ Einträge zu sortieren. Diese Einträge sind wieder absteigend sortiert, so dass sich unser Argument wiederholen lässt.

Da die `while`-Schleifen mindestens $5n$ Operationen erfordern und jeder rekursive Aufruf die Arraygröße jeweils nur um eins reduziert, benötigt `quicksort` mindestens

$$\sum_{k=1}^{n} 5k = \frac{5}{2} n(n+1)$$

```
1  void
2  quicksort(int n, int *x)
3  {
4    int m, j;
5    int y, z;
6    if(n > 1) {
7      y = x[0];
8      m = 1; j = n-1;
9      while(m <= j) {
10       while((m <= j) && (x[m] <= y))
11         m++;
12       while((m <= j) && (y < x[j]))
13         j--;
14       if(m < j) {
15         z = x[m]; x[m] = x[j]; x[j] = z;
16         m++; j--;
17       }
18     }
19     z = x[0]; x[0] = x[m-1]; x[m-1] = z;
20     quicksort(m-1, x);
21     quicksort(n-m, x+m);
22   }
23 }
```

Abbildung 2.11: Grundlegender Quicksort-Algorithmus

Operationen, kann also *erheblich* langsamer als der Mergesort-Algorithmus sein, obwohl der Algorithmus ebenfalls auf dem Teile-und-herrsche-Ansatz beruht.

Immerhin können wir beweisen, dass der Rechenaufwand auch im schlimmsten Fall nicht schneller als quadratisch mit n wachsen kann:

Lemma 2.13 (Rechenaufwand) *Sei $R(n)$ die Anzahl der Operationen, die die Funktion* `quicksort` *für ein Array der Länge n höchstens benötigt. Wenn wir mit $\widehat{m}$ den Wert bezeichnen, den die Variable* `m` *in Zeile 19 aufweist, erhalten wir*

$$R(n) \leq \begin{cases} 1 & \text{falls } n \leq 1, \\ 16 + 15n + R(\widehat{m} - 1) + R(n - \widehat{m}) & \text{ansonsten} \end{cases} \quad \text{für alle } n \in \mathbb{N}_0. \tag{2.11}$$

Beweis. Falls $n \leq 1$ gilt, wird lediglich der Ausdruck `n > 1` in Zeile 6 ausgewertet und keine weitere Arbeit geleistet, also folgt in diesem Fall $R(n) = 1$.

Ansonsten müssen wir klären, wieviele Iterationen der äußeren Schleife in den Zeilen 9 bis 18 höchstens ausgeführt werden. Die Schleife läuft, solange $m \leq j$ gilt, und falls $m < j$ gelten sollte, werden in Zeile 16 m um eins erhöht und j um eins gesenkt. Falls

$m = j$ gelten sollte, muss entweder `x[m] <= y` oder `y < x[j]` gelten, so dass entweder in Zeile 11 oder in Zeile 13 dafür gesorgt wird, dass anschließend $m > j$ gilt und die Schleife endet.

Solange also $m < j$ gilt, wird in jeder Iteration die Differenz $j - m$ um mindestens zwei reduziert, für $m = j$ immerhin um eins. Bei Eintritt in die Schleife stellt Zeile 8 sicher, dass $j - m = n - 2$ gilt, also erfolgen höchstens

$$t = \lfloor (n-2)/2 \rfloor + 1 = \lfloor n/2 \rfloor \leq n/2$$

Iterationen.

In der Schleife in den Zeilen 10 und 11 wird m hochgezählt, bis die Schleifenbedingung verletzt ist. Wenn wir mit $\widehat{m}$ den endgültigen Wert bezeichnen, den m nach Verlassen der äußeren Schleife angenommen hat, folgt, dass die Zeile 11 höchstens $(\widehat{m} - 1)$-mal ausgeführt wurde, während Zeile 10 höchstens $(\widehat{m} - 1 + t)$-mal ausgeführt wurde, da die Schleifenbedingung in jeder Iteration der äußeren Schleife jeweils einmal nicht erfüllt gewesen sein muss. Damit ergibt sich ein Rechenaufwand von $4(\widehat{m} - 1 + t)$ für Zeile 10 und ein Aufwand von $\widehat{m} - 1$ für Zeile 11.

Entsprechend wird in den Zeilen 12 und 13 die Variable j heruntergezählt, bis die Schleifenbedingung verletzt ist. Wenn wir mit $\widehat{j}$ den endgültigen Wert bezeichnen, den j nach Verlassen der äußeren Schleife angenommen hat, folgt, dass die Zeile 13 höchstens $(n - 1 - \widehat{j})$-mal ausgeführt wurde, so dass $4(n - 1 - \widehat{j})$ Operationen anfallen, während die eine Operation in Zeile 12 höchstens $(n - 1 - \widehat{j} + t)$-mal ausgeführt wurde.

Die äußere Schleife wird beendet, sobald $m > j$ gilt. Da sich die Differenz der beiden Variablen nur in den Zeilen 11, 13 und 16 ändert und dabei in den ersten beiden Fällen $m \leq j$ und im letzten sogar $m < j$ sichergestellt sind, muss bei Verlassen der äußeren Schleife $m = j + 1$ gelten, also haben wir $\widehat{j} = \widehat{m} - 1$, so dass sich insgesamt für die Zeilen 12 und 13 höchstens $n - \widehat{m}$ Operationen für Zeile 13 und höchstens $4(n - \widehat{m} + t)$ Operationen für Zeile 12 ergeben.

Für die gesamte äußere Schleife erhalten wir so einen Aufwand von höchstens

$$\begin{aligned} 4(\widehat{m} - 1 + t) + \widehat{m} - 1 + 4(n - \widehat{m} + t) + n - \widehat{m} + 11t + 1 \\ = 5(n-1) + 19t + 1 \leq 15n - 4 \end{aligned}$$

Operationen, wobei neben den inneren Schleifen auch die Zeilen 9 und 14 bis 16 zu berücksichtigen sind.

In den Zeilen 6 bis 8 fallen nicht mehr als $1 + 2 + 3 = 6$ Operationen an, Zeile 19 benötigt 9, und die rekursiven Aufrufe in den Zeilen 20 und 21 erfordern $2 + R(\widehat{m} - 1)$ sowie $3 + R(n - \widehat{m})$ Operationen, so dass wir insgesamt die Rekurrenzformel (2.11) erhalten. ∎

Aus der Rekurrenzformel (2.11) können wir mit Hilfe des folgenden Lemmas wieder eine geschlossene Formel für eine obere Schranke des Aufwands gewinnen:

Lemma 2.14 (Rekurrenz) *Seien $\alpha, \beta \in \mathbb{N}_0$ gegeben und sei $f : \mathbb{N}_0 \to \mathbb{R}_{\geq 0}$ eine Abbildung mit*

$$f(n) \leq \begin{cases} \alpha & \text{falls } n = 0, \\ \alpha + \beta n + \max\{f(m-1) + f(n-m) \ : \ m \in \{1, ..., n\}\} & \text{ansonsten} \end{cases} \tag{2.12}$$

für alle $n \in \mathbb{N}_0$. Dann gilt

$$f(n) \leq \alpha(2n+1) + \frac{\beta}{2} n(n+1) \qquad \text{für alle } n \in \mathbb{N}_0. \tag{2.13}$$

Beweis. Wir beweisen für alle $\ell \in \mathbb{N}_0$, dass (2.13) für alle $n \in \{0, \ldots, \ell\}$ gilt.

Induktionsanfang. Für $\ell = 0$ folgt $n = \ell = 0$, also

$$f(0) \leq \alpha = \alpha(2n+1) + \frac{\beta}{2} n(n+1).$$

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gegeben, dass (2.13) für alle $n \in \{0, \ldots, \ell\}$ gilt.

Induktionsschritt. Wir müssen (2.13) für $n = \ell + 1$ nachweisen. Da $n \geq 1$ gilt, folgt aus (2.12)

$$f(n) \leq \alpha + \beta n + f(m-1) + f(n-m)$$

für ein $m \in \{1, \ldots, n\}$. Wegen $m - 1, n - m \in \{0, \ldots, n-1\} = \{0, \ldots, \ell\}$ dürfen wir die Induktionsvoraussetzung anwenden und erhalten

$$\begin{aligned} f(m-1) &\leq \alpha(2(m-1)+1) + \frac{\beta}{2}(m-1)m, \\ f(n-m) &\leq \alpha(2(n-m)+1) + \frac{\beta}{2}(n-m)(n-m+1). \end{aligned}$$

Insgesamt ergibt sich so

$$\begin{aligned} f(n) &\leq \alpha + \beta n + \alpha(2m-1) + \frac{\beta}{2}(m-1)m + \alpha(2(n-m)+1) + \frac{\beta}{2}(n-m)(n-m+1) \\ &= \alpha(2n+1) + \frac{\beta}{2}\left(2n + (m-1)m + (n-m)(n-m+1)\right) \\ &\leq \alpha(2n+1) + \frac{\beta}{2}\left(2n + (m-1)m + (n-m)(n-m+1) + 2(m-1)(n-m)\right) \\ &= \alpha(2n+1) + \frac{\beta}{2}\left(2n + (m-1)m + (m-1)(n-m)\right. \\ &\qquad \left. + (n-m)(n-m+1) + (n-m)(m-1)\right) \\ &= \alpha(2n+1) + \frac{\beta}{2}\left(2n + (m-1)n + (n-m)n\right) \\ &= \alpha(2n+1) + \frac{\beta}{2}\left(2n + (n-1)n\right) = \alpha(2n+1) + \frac{\beta}{2} n(n+1) \end{aligned}$$

und wir sind fertig. Der in der dritten Zeile eingefügte Term $2(m-1)(n-m)$ verschwindet genau dann, wenn $m = 1$ oder $m = n$ gilt. In diesem Fall erhalten wir also gerade die ungünstigste Abschätzung. ∎

Satz 2.15 (Rechenaufwand) *Der Algorithmus* `quicksort` *benötigt für ein Array der Länge n nicht mehr als* $16 + 32n + 8n(n+1)$ *Operationen.*

Beweis. Nach Lemma 2.13 gilt (2.12) mit $\alpha = 16$ und $\beta = 15$, so dass wir Lemma 2.14 anwenden können, um

$$R(n) \leq 16(2n+1) + \frac{15}{2}n(n+1) < 16 + 32n + 8n(n+1)$$

zu erhalten. ∎

Trotz des im ungünstigsten Fall quadratisch wachsenden Rechenaufwands erfreut sich der Quicksort-Algorithmus großer Beliebtheit, weil er sich sehr effizient implementieren lässt und „in den meisten Fällen“ schneller als andere Sortierverfahren arbeitet.

Die Effizienz des Algorithmus' steht und fällt mit der Qualität der gewählten Pivot-Elemente. Das bestmögliche Element wäre der *Median* der Werte des Arrays, also dasjenige Element, das größer als genau $\lfloor n/2 \rfloor$ Elemente ist, denn mit dieser Wahl würde das erste Teilarray genau $m = \lfloor n/2 \rfloor$ Elemente enthalten, so dass wir die Analyse des Verfahrens wie im Fall des Mergesort-Algorithmus' durchführen und zu einer vergleichbaren Effizienz kommen könnten.

Die Berechnung des Medians ist allerdings ebenfalls ein schwieriges Problem. Glücklicherweise würde es für unsere Zwecke auch schon ausreichen, ein Element zu finden, dass größer als $\lfloor \alpha n \rfloor$ und kleiner als $\lceil (1-\alpha)n \rceil$ Elemente des Arrays ist, wobei $\alpha \in (0,1)$ eine von n und natürlich dem Inhalt des Arrays unabhängige Konstante ist. Einen derartigen „approximativen Median“ kann man tatsächlich mit einem effizienten Algorithmus finden und so verhindern, dass der Quicksort-Algorithmus für ungünstige Arrays ineffizient wird. Allerdings wächst dadurch der Gesamtaufwand deutlich.

Ein einfacherer Ansatz besteht darin, das Pivot-Element *zufällig* zu wählen und zu untersuchen, wie hoch der Rechenaufwand im Mittel sein wird. Der resultierende *randomisierte Quicksort-Algorithmus* ist in Abbildung 2.12 zusammengefasst: In Zeile 7 wird mit der in der C-Standardbibliothek enthaltenen Funktion `rand` eine Zufallszahl k (in der Praxis meistens eher nur eine aus einer deterministischen Berechnung hervorgegangene Pseudo-Zufallszahl) zwischen 0 und $n-1$ ermittelt, in Zeile 8 werden dann die Array-Einträge x_0 und x_k getauscht, so dass das Pivot-Element anschließend wieder in x_0 steht. Gegenüber der in Lemma 2.13 gegebenen Abschätzung kommen dann 10 Operationen für die Berechnung von k und den Tausch hinzu, während 2 Operationen für das Setzen der Variablen `y` entfallen. Insgesamt erhöht sich die Anzahl der Operationen also um 8.

Für die Analyse des *randomisierten* Verfahrens müssen wir einerseits 8 Operationen für die Zeilen 7 und 8 hinzufügen (zwei weitere Operationen treten in leicht modifizierter Form bereits in Zeile 7 des ursprünglichen Algorithmus' auf) und andererseits vor allem Annahmen darüber treffen, wie wahrscheinlich die verschiedenen Werte von $\widehat{m}$ in Lemma 2.13 sind. Wenn wir davon ausgehen, dass alle Einträge des Arrays paarweise verschieden sind und `k` in Zeile 7 des Algorithmus' die Werte zwischen 0 und $n-1$ mit gleicher Wahrscheinlichkeit annimmt, erfüllt der *Erwartungswert* $E(n)$ des Rechenauf-

```
1  void
2  quicksort_randomized(int n, int *x)
3  {
4    int m, j, k;
5    int y, z;
6    if(n > 1) {
7      k = rand() % n;
8      y = x[k]; x[k] = x[0]; x[0] = y;
9      m = 1; j = n-1;
10     while(m <= j) {
11       while((m <= j) && (x[m] <= y))
12         m++;
13       while((m <= j) && (y < x[j]))
14         j--;
15       if(m < j) {
16         z = x[m]; x[m] = x[j]; x[j] = z;
17         m++; j--;
18       }
19     }
20     z = x[0]; x[0] = x[m-1]; x[m-1] = z;
21     quicksort_randomized(m-1, x);
22     quicksort_randomized(n-m, x+m);
23   }
24 }
```

Abbildung 2.12: Randomisierter Quicksort-Algorithmus

wands für ein Array der Länge $n \geq 2$ die durch

$$\begin{aligned} E(n) &= \frac{1}{n}\left(\sum_{m=1}^{n} 24 + 15n + E(m-1) + E(n-m)\right) \\ &= 24 + 15n + \frac{1}{n}\sum_{m=1}^{n}(E(m-1) + E(n-m)) \\ &= 24 + 15n + \frac{1}{n}\sum_{k=0}^{n-1} E(k) + \frac{1}{n}\sum_{k=0}^{n-1} E(k) \\ &= 24 + 15n + \frac{2}{n}\sum_{k=0}^{n-1} E(k) \\ &\leq 25 + 15n + \frac{2}{n}\sum_{k=1}^{n-1} E(k) \end{aligned} \tag{2.14}$$

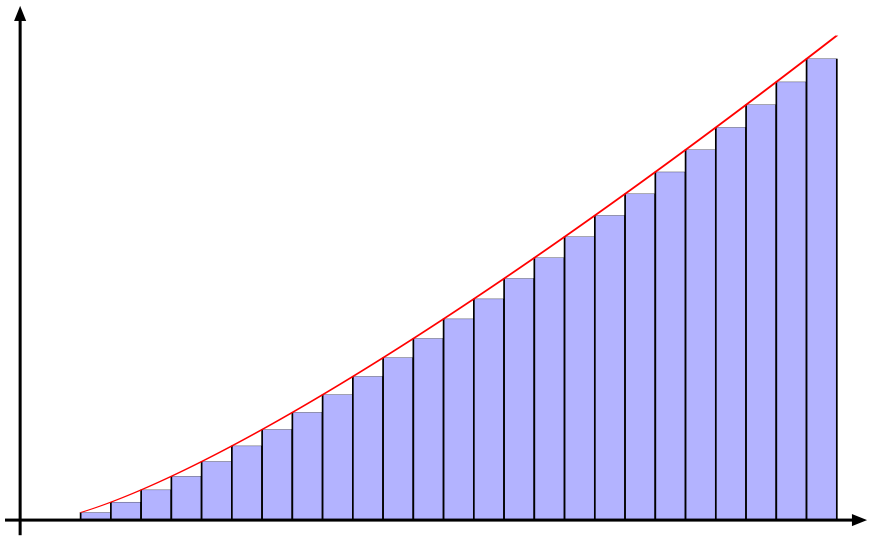


Abbildung 2.13: Abschätzung der Summe über $k \log(k)$ durch das Integral über $x \log(x)$. Die Summanden sind durch blaue Rechtecke der Breite eins dargestellt, die zu integrierende Funktion in rot.

gegebene Rekurrenzformel. Im letzten Schritt haben wir $E(0) = 1$ und $2/n \leq 1$ ausgenutzt.

Auch bei der Untersuchung dieser Formel erweist sich der Logarithmus als nützlich, allerdings diesmal nicht der dyadische, sondern der *natürliche Logarithmus*.

Erinnerung 2.16 (Natürlicher Logarithmus) *Mit* $e \approx 2,71828$ *bezeichnen wir die* Eulersche Zahl. *Die* Exponentialfunktion *ist durch* $x \mapsto e^x$ *gegeben, ihre Umkehrfunktion* $\ln : \mathbb{R}_{>0} \to \mathbb{R}$ *nennen wir den* natürlichen Logarithmus.

Wie der dyadische Logarithmus ist auch der natürliche Logarithmus eine streng monoton wachsende Funktion, für die die Rechenregeln

$$\ln(1) = 0, \qquad \ln(e) = 1, \qquad \ln(xy) = \ln(x) + \ln(y) \qquad \text{für alle } x, y \in \mathbb{R}_{>0}$$

gelten. Der natürliche Logarithmus ist stetig differenzierbar, seine Ableitung ist durch $\ln'(x) = 1/x$ *für alle* $x \in \mathbb{R}_{>0}$ *gegeben.*

Mit Hilfe des natürlichen Logarithmus' können wir die folgende Abschätzung für die Lösung unserer Rekurrenzformel gewinnen:

Lemma 2.17 (Rekurrenz) *Seien* $\alpha, \beta, \gamma \in \mathbb{N}_0$ *gegeben und sei* $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ *eine Abbildung mit*

$$f(n) \leq \begin{cases} \alpha & \text{falls } n = 1, \\ \beta + \gamma n + \frac{2}{n} \sum_{k=1}^{n-1} f(k) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N}. \tag{2.15}$$

Dann gilt

$$f(n) \leq \alpha n + \beta(n-1) + 2\gamma n \ln(n) \qquad \text{für alle } n \in \mathbb{N}. \tag{2.16}$$

Beweis. Auch in diesem Fall können wir per Induktion über $\ell \in \mathbb{N}$ beweisen, dass (2.16) für alle $n \in \{1, \ldots, \ell\}$ gilt.

Induktionsanfang. Für $\ell = 1$ folgt $n = \ell = 1$, und mit $\ln(1) = 0$ ergibt sich

$$f(n) = f(1) \leq \alpha = \alpha n + \beta(n-1) + 2\gamma n \ln(n).$$

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass (2.16) für alle $n \in \{1, \ldots, \ell\}$ gilt.

Induktionsschritt. Wir müssen die Aussage für $n = \ell + 1$ nachweisen. Wegen $n \geq 2$ erhalten wir aus (2.15) die Ungleichung

$$f(n) \leq \beta + \gamma n + \frac{2}{n} \sum_{k=1}^{n-1} f(k) = \beta + \gamma n + \frac{2}{n} \sum_{k=1}^{\ell} f(k),$$

und indem wir die Induktionsvoraussetzung auf $f(k)$ anwenden, erhalten wir

$$f(n) \leq \beta + \gamma n + \frac{2}{n} \sum_{k=1}^{n-1} (\alpha k + \beta(k-1) + 2\gamma k \ln(k)).$$

Mit der Gauß'schen Summenformel lassen sich die ersten beiden Summen einfach abschätzen, denn es gilt

$$\frac{2}{n} \sum_{k=1}^{n-1} \alpha k = \frac{2}{n} \alpha \sum_{k=1}^{n-1} k = \frac{2}{n} \alpha \frac{n(n-1)}{2} = \alpha(n-1) < \alpha n,$$
$$\frac{2}{n} \sum_{k=1}^{n-1} \beta(k-1) = \frac{2}{n} \beta \sum_{k=0}^{n-2} k = \frac{2}{n} \beta \frac{(n-2)(n-1)}{2} < \beta(n-2).$$

Für den dritten Term nutzen wir aus, dass die Funktion

$$g : \mathbb{R}_{>0} \to \mathbb{R}, \qquad x \mapsto x \ln(x),$$

monoton wachsend ist, so dass

$$k \ln(k) = \int_k^{k+1} k \ln(k)\, dx \leq \int_k^{k+1} x \ln(x)\, dx \qquad \text{für alle } k \in \mathbb{N}$$

gilt. Für unsere Summe ergibt sich wegen $\ln(1) = 0$ damit

$$\sum_{k=1}^{n-1} k \ln(k) = \sum_{k=2}^{n-1} k \ln(k) \leq \sum_{k=2}^{n-1} \int_k^{k+1} x \ln(x)\, dx = \int_2^n x \ln(x)\, dx = \int_2^n g(x)\, dx.$$

Diese Abschätzung der Summe durch das Integral ist in Abbilung 2.13 illustriert. Die Funktion g besitzt die Stammfunktion

$$G : \mathbb{R}_{>0} \to \mathbb{R}, \qquad x \mapsto \frac{x^2}{2}\ln(x) - \frac{x^2}{4},$$

so dass wir mit dem Hauptsatz der Integral- und Differentialrechnung das Integral ausrechnen können, um

$$\begin{aligned}\sum_{k=1}^{n-1} k\ln(k) &\leq \int_2^n g(x)\,dx = G(n) - G(2)\\ &= \frac{n^2}{2}\ln(n) - \frac{n^2}{4} - 2\ln(2) + 1 < \frac{n^2}{2}\ln(n) - \frac{n^2}{4}\end{aligned}$$

zu erhalten. Dabei haben wir im letzten Schritt ausgenutzt, dass $2\ln(2) = \ln(2^2) = \ln(4) > \ln(e) = 1$ gilt. Insgesamt haben wir also

$$\begin{aligned}f(n) &< \beta + \gamma n + \alpha n + \beta(n-2) + 2\gamma n\ln(n) - 2\gamma\frac{n}{2}\\ &= \alpha n + \beta(n-2+1) + 2\gamma n\ln(n) - \gamma n + \gamma n\\ &= \alpha n + \beta(n-1) + 2\gamma n\ln(n)\end{aligned}$$

bewiesen und sind fertig. ∎

Satz 2.18 (Rechenaufwand) *Im Erwartungswert benötigt der randomisierte Algorithmus* `quicksort_randomized` *nicht mehr als* $26n + 30n\ln(n)$ *Operationen für ein Array der Länge* n*.*

Beweis. Nach (2.14) erfüllt der erwartete Rechenaufwand $E(n)$ für ein Array der Länge n die Bedingung (2.15) mit $\alpha = 1$, $\beta = 25$ und $\gamma = 15$, so dass wir mit Lemma 2.17 die Abschätzung

$$E(n) \leq n + 25(n-1) + 30n\ln(n) < 26n + 30n\ln(n)$$

erhalten. ∎

Für einen praktischen Vergleich der bisher behandelten Verfahren ziehen wir wieder einen Prozessor des Typs Intel Core™ i7-2600K mit 3,4 GHz Taktfrequenz heran: Für ein Array mit $n = 1\,000\,000$ (Pseudo-) Zufallszahlen benötigt `quicksort` nur ungefähr $0,08$ Sekunden, während `mergesort` mit $0,04$ Sekunden auskommt.

Falls das Array absteigend sortiert ist, also der für `quicksort` ungünstigste Fall vorliegt, benötigt das Verfahren 221 Sekunden und muss außerdem etwas umgeschrieben werden, um zu verhindern, dass die n rekursiven Aufrufe den für Rekursionen vorgesehenen Speicher zum Überlaufen bringen. Sogar `insertionsort` ist mit 120 Sekunden deutlich schneller, während `mergesort` weiterhin nur $0,04$ Sekunden benötigt.

Die Situation für den Quicksort-Algorithmus verbessert sich, falls wir auf die randomisierte Variante `quicksort_randomized` zurückgreifen, die das absteigend sortierte Array in $0,03$ Sekunden sortiert, also sogar etwas schneller als `mergesort`.

2.6 Landau-Notation

Bisher haben wir den Aufwand eines Algorithmus' abgeschätzt, indem wir gezählt haben, wieviele Operatoren angewendet werden müssen, um ihn auszuführen. Diese Vorgehensweise ist nur eine grobe Schätzung der *Rechenzeit* des Algorithmus', denn unterschiedliche Operatoren benötigen in der Regel unterschiedlich viel Zeit, außerdem können bei modernen Prozessoren beispielsweise Cache-Speicher dazu führen, dass sogar ein und dieselbe Operation zu verschiedenen Zeiten unterschiedlich lange braucht.

Da also unsere Schätzung ohnehin ungenau ist, spricht nichts dagegen, sie noch ungenauer zu machen, falls es uns Arbeit erspart. Beispielsweise haben wir bei der Aufwandsabschätzung des Mergesort-Algorithmus' eine Schranke von $19n + 19n\lceil\log_2 n\rceil$ erhalten. Für große Werte von n, beispielsweise für $n = 1\,000\,000$, trägt der erste Term der Abschätzung lediglich ungefähr 5 Prozent zu dem Gesamtergebnis bei, so dass wir ihn weglassen könnten, ohne allzu viel an Genauigkeit einzubüßen.

Wir können also sagen, dass sich der Aufwand für großes n ungefähr wie $19n\lceil\log_2 n\rceil$ verhält. Wie bereits erwähnt ist das Zählen von Operationen ebenfalls nicht so exakt, wie es auf den ersten Blick erscheint, deshalb wird in der Praxis der Faktor 19 ebenfalls weggelassen, so dass man nur noch davon spricht, dass der Aufwand proportional zu $n \log_2 n$ wächst.

Diese Sprechweise wird durch die *Landau-Notation* (benannt nach Edmund Landau, vermutlich erstmals verwendet von Paul Bachmann) mathematisch präzise gefasst:

Definition 2.19 (Landau-Notation) *Seien $f, g : \mathbb{N} \to \mathbb{R}$ Abbildungen.*

Falls Konstanten $C \in \mathbb{R}_{>0}$ und $n_0 \in \mathbb{N}$ so existieren, dass

$$f(n) \leq Cg(n) \qquad \text{für alle } n \in \mathbb{N}_{\geq n_0}$$

gilt, schreiben wir $f \in \mathcal{O}(g)$ und sagen, dass die Funktion f asymptotisch nicht schneller als $\mathcal{O}(g)$ wächst*.*

Falls Konstanten $C \in \mathbb{R}_{>0}$ und $n_0 \in \mathbb{N}$ so existieren, dass

$$f(n) \geq Cg(n) \qquad \text{für alle } n \in \mathbb{N}_{\geq n_0}$$

gilt, schreiben wir $f \in \Omega(g)$ und sagen, dass die Funktion f asymptotisch nicht langsamer als $\Omega(g)$ wächst*.*

Falls sogar $C_1, C_2 \in \mathbb{R}_{>0}$ und $n_0 \in \mathbb{N}$ so existieren, dass

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \qquad \text{für alle } n \in \mathbb{N}_{\geq n_0}$$

gilt, schreiben wir $f \in \Theta(g)$ und sagen, dass die Funktion f asymptotisch wie $\Theta(g)$ wächst*.*

Mathematisch präzise können wir $\mathcal{O}(g)$, $\Omega(g)$ sowie $\Theta(g)$ als die folgenden Mengen von Abbildungen definieren:

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R} \ : \ \exists C \in \mathbb{R}_{>0}\, \exists n_0 \in \mathbb{N}\, \forall n \in \mathbb{N}_{\geq n_0} : f(n) \leq Cg(n)\},$$
$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R} \ : \ \exists C \in \mathbb{R}_{>0}\, \exists n_0 \in \mathbb{N}\, \forall n \in \mathbb{N}_{\geq n_0} : f(n) \geq Cg(n)\},$$
$$\Theta(g) := \{f : \mathbb{N} \to \mathbb{R} \ : \ \exists C_1, C_2 \in \mathbb{R}_{>0}\, \exists n_0 \in \mathbb{N}\, \forall n \in \mathbb{N}_{\geq n_0} : C_1 g(n) \leq f(n) \leq C_2 g(n)\}.$$

Da die Landau-Notation als Abkürzung gedacht ist, verzichtet man häufig darauf, die Funktionen f und g explizit zu definieren und schreibt stattdessen einen Term, der sie implizit festlegt. Beispielsweise ist es üblich,

$$f \in \mathcal{O}(n \log_2(n))$$

statt der präzisen Form

$$f \in \mathcal{O}(g) \text{ für } g : \mathbb{N} \to \mathbb{R},\ n \mapsto n \log_2(n)$$

zu verwenden, wenn klar ist, dass „$n \log_2(n)$“ als Abbildung zu verstehen ist, die $n \in \mathbb{N}$ den Wert $n \log_2(n)$ zuordnet.

Dieselbe Regel wird auch auf f angewendet, so dass man in der Literatur beispielsweise Formulierungen wie

$$\begin{aligned} m^2 + 7m &\in \Theta(m^2), & 5 \sin(k) &\in \mathcal{O}(1), \\ 13 + 10\lfloor \log_2 n \rfloor &\in \Theta(\log_2(n)), & 19n + 19n\lceil \log_2(n) \rceil &\in \Theta(n \log_2(n)) \end{aligned}$$

findet. Die Aussagen der ersten Zeile ergeben sich unmittelbar aus

$$\begin{aligned} m^2 \leq m^2 + 7m \leq m^2 + m^2 \leq 2m^2 \qquad &\text{für alle } m \in \mathbb{N}_{\geq 7}, \\ 5 \sin(k) \leq 5 = 5 \cdot 1 \qquad &\text{für alle } k \in \mathbb{N}. \end{aligned}$$

Wie man sieht, ist für die erste Zeile von entscheidender Bedeutung, dass wir $n_0 = 7$ wählen können, um $m \geq 7$ sicherzustellen, da sich nur dann der Term $7m$ durch m^2 abschätzen lässt.

Für den Nachweis der Aussagen der zweiten Zeile nutzen wir aus, dass für $n \in \mathbb{N}_{\geq 2}$ immer $\log_2(n) \geq \lfloor \log_2(n) \rfloor \geq 1$ und deshalb

$$\begin{aligned} \log_2(n) \leq \lfloor \log_2(n) \rfloor + 1 \leq 2\lfloor \log_2(n) \rfloor \leq 2 \log_2(n), \\ \log_2(n) \leq \lceil \log_2(n) \rceil \leq \log_2(n) + 1 \leq 2 \log_2(n) \end{aligned}$$

gelten, so dass wir im ersten Fall

$$5 \log_2(n) \leq 13 + 10\lfloor \log_2(n) \rfloor \leq (13 + 10) \log_2(n) = 23 \log_2(n)$$

und im zweiten

$$19n \log_2(n) \leq 19n + 19n\lceil \log_2(n) \rceil \leq (19 + 38) n \log_2(n) = 57n \log_2(n)$$

erhalten. Wie man sieht ist es in beiden Abschätzungen von entscheidender Bedeutung, dass wir eine Untergrenze für n festlegen dürfen, denn beispielsweise für $n = 1$ könnten wir 13 nicht durch ein Vielfaches von $\log_2(n) = 0$ nach oben beschränken.

Unsere bisherigen Aufwandsabschätzungen können wir mit Hilfe der neuen Notation kurz zusammenfassen:

- Der Rechenaufwand der linearen Suche wächst asymptotisch nicht schneller als $\mathcal{O}(n)$.

- Der Rechenaufwand der binären Suche wächst asymptotisch nicht schneller als $\mathcal{O}(\log_2(n))$.
- Der Rechenaufwand des Sortierens durch Einfügen wächst asymptotisch nicht schneller als $\mathcal{O}(n^2)$.
- Der Rechenaufwand des Mergesort-Algorithmus' wächst asymptotisch nicht schneller als $\mathcal{O}(n \log_2(n))$.
- Der Rechenaufwand des Quicksort-Algorithmus' wächst asymptotisch nicht schneller als $\mathcal{O}(n^2)$, für den randomisierten Algorithmus wächst der erwartete Aufwand nicht schneller als $\mathcal{O}(n \ln(n))$.

Die Landau-Notation ist ausgesprochen praktisch, wenn man nicht daran interessiert ist, einzelne Operationen zu zählen. Beispielsweise können wir bei dem Algorithmus `insertionsort` knapp argumentieren, dass die innere Schleife (Zeilen 9 bis 12) asymptotisch nicht mehr als $\mathcal{O}(n)$ Operationen benötigen kann. Damit benötigt auch eine vollständige Iteration der äußeren Schleife nicht mehr als $\mathcal{O}(n)$ Operationen.

Wir würden nun gerne aus dieser Aussage und der Tatsache, dass die äußere Schleife genau n-mal durchlaufen wird, darauf schließen, dass der Gesamtaufwand asymptotisch nicht schneller als $\mathcal{O}(n^2)$ wächst. Diese Aufgabe lässt sich mit den folgenden Rechenregeln einfach lösen:

Lemma 2.20 (Addition und Multiplikation) *Seien $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}_{\geq 0}$ gegeben. Es gelte $f_1 \in \mathcal{O}(g_1)$ und $f_2 \in \mathcal{O}(g_2)$. Dann folgen*

$$f_1 + f_2 \in \mathcal{O}(g_1 + g_2), \qquad f_1 f_2 \in \mathcal{O}(g_1 g_2).$$

Beweis. Nach Definition finden wir $C_1, C_2 \in \mathbb{R}_{>0}$ sowie $n_1, n_2 \in \mathbb{N}$ mit

$$\begin{aligned} f_1(n) &\leq C_1 g_1(n) && \text{für alle } n \in \mathbb{N}_{\geq n_1}, \\ f_2(n) &\leq C_2 g_2(n) && \text{für alle } n \in \mathbb{N}_{\geq n_2}. \end{aligned}$$

Für $n_0 := \max\{n_1, n_2\}$ folgen daraus

$$\begin{aligned} f_1(n) + f_2(n) &\leq C_1 g_1(n) + C_2 g_2(n) \leq (C_1 + C_2) g_1(n) + (C_1 + C_2) g_2(n) \\ &= (C_1 + C_2)(g_1 + g_2)(n) \qquad \text{und} \\ f_1(n) f_2(n) &\leq C_1 g_1(n) C_2 g_2(n) = (C_1 C_2)(g_1 g_2)(n) \qquad \text{für alle } n \in \mathbb{N}_{\geq n_0}, \end{aligned}$$

also $f_1 + f_2 \in \mathcal{O}(g_1 + g_2)$ sowie $f_1 f_2 \in \mathcal{O}(g_1 g_2)$. ∎

Für `insertionsort` ergibt sich, dass n Iterationen mit einem Aufwand von $\mathcal{O}(n)$ anfallen, so dass wir auf einen Gesamtaufwand von $\mathcal{O}(n^2)$ schließen dürfen.

Bei rekursiven Algorithmen müssen wir darauf achten, die Landau-Notation korrekt zu verwenden: Beispielsweise bei `mergesort` (siehe Abbildung 2.8) liegt die Anzahl der für das Kopieren und Zusammenfügen der Teil-Arrays erforderlichen Operationen in $\mathcal{O}(n)$.

Wir bezeichnen mit $R(n)$ eine obere Schranke für den Rechenaufwand der Funktion `mergesort` für $n \in \mathbb{N}$ Elemente.

Bemerkung 2.21 (Rekursion) *Mit einer (fehlerhaften!) Induktion könnten wir wie folgt „beweisen“, dass $R \in \mathcal{O}(n)$ gilt: Der Aufwand für Kopieren und Zusammenfügen liegt in $\mathcal{O}(n)$, und nach Induktionsvoraussetzung liegt auch der Aufwand für die rekursiven Aufrufe in $\mathcal{O}(\lfloor n/2 \rfloor) = \mathcal{O}(n)$ beziehungsweise $\mathcal{O}(n - \lfloor n/2 \rfloor) = \mathcal{O}(n)$, also muss nach Lemma 2.20 auch die Summe in $\mathcal{O}(n)$ liegen.*

Der Fehler dieses „Beweises“ besteht darin, dass sich bei der Addition in Lemma 2.20 die Konstante C der Landau-Notation verändert. Demzufolge wird bei jedem Schritt der Induktion die Konstante größer und ist damit nicht mehr von n unabhängig, wie es die Definition verlangt.

Um eine korrekte Aufwandsabschätzung trotz der durch die Landau-Notation gebotenen Vereinfachung zu erhalten, können wir uns mit dem folgenden Lemma die Arbeit erleichtern:

Lemma 2.22 (Vereinfachung) *Seien $f, g \in \mathbb{N} \to \mathbb{R}$ mit $f \in \mathcal{O}(g)$ derart gegeben, dass*

$$g(n) \neq 0 \qquad \text{für alle } n \in \mathbb{N}$$

gilt. Dann existiert ein $\widehat{C} \in \mathbb{R}_{>0}$ so, dass

$$f(n) \leq \widehat{C} g(n) \qquad \text{für alle } n \in \mathbb{N}$$

gilt, wir also für alle *n eine Abschätzung erhalten, nicht nur für hinreichend große.*

Beweis. Da $f \in \mathcal{O}(g)$ gilt, finden wir $C_0 \in \mathbb{R}_{>0}$ und $n_0 \in \mathbb{N}$ so, dass

$$f(n) \leq C_0 g(n) \qquad \text{für alle } n \in \mathbb{N}_{\geq n_0}$$

gilt, wir müssen also nur die ersten $n_0 - 1$ Zahlen behandeln.

Da die Menge $\{1, \ldots, n_0 - 1\}$ endlich und f/g auf dieser Menge wohldefiniert ist, existiert das Maximum

$$C_1 := \max\left\{ \frac{|f(n)|}{|g(n)|} \;:\; n \in \{1, \ldots, n_0 - 1\} \right\},$$

so dass wir

$$f(n) \leq C_1 g(n) \qquad \text{für alle } n \in \{1, \ldots, n_0 - 1\}$$

erhalten. Mit $\widehat{C} := \max\{C_0, C_1\}$ folgt die Behauptung. ∎

Jetzt können wir den Aufwand der Funktion `mergesort` korrekt abschätzen, indem wir die Konstanten explizit mitführen: Der Aufwand für Kopieren und Zusammenfügen ist in $\mathcal{O}(n)$, also existiert ein $\widehat{C} \in \mathbb{R}_{>0}$ so, dass für alle $n \in \mathbb{N}$ nicht mehr als $\widehat{C}n$ Operationen benötigt werden. Für den Gesamtaufwand erhalten wir dann die Rekurrenzformel

$$R(n) \leq \begin{cases} \widehat{C} & \text{falls } n = 1, \\ \widehat{C}n + R(\lfloor n/2 \rfloor) + R(n - \lfloor n/2 \rfloor) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N},$$

die wir wie zuvor mit Lemma 2.11 auflösen können, um einen Aufwand in $\mathcal{O}(n \log_2(n))$ zu erhalten. Das ist das korrekte Ergebnis.

2.7 Beispiel: Suchen in vollständig geordneten Mengen

Bisher haben wir uns darauf beschränkt, in Folgen von Zahlen zu suchen und Folgen von Zahlen zu sortieren. In der Praxis müssen wir häufig Folgen allgemeinerer Objekte verarbeiten, beispielsweise um einen Namen in einem Verzeichnis zu finden, also sind wir daran interessiert, unsere Algorithmen entsprechend zu verallgemeinern.

Dazu untersuchen wir, welche Eigenschaften wir benötigen, damit beispielsweise die binäre Suche funktioniert. Wenn wir davon ausgehen, dass die Objekte, unter denen wir suchen wollen, einer Menge M entstammen, brauchen wir eine Verallgemeinerung des Größenvergleichs zweier Zahlen, wir müssten also auch für $x, y \in M$ definieren, wann in einem geeigneten Sinn „$x \leq y$“ gelten soll.

Diese Aufgabe lässt sich mathematisch relativ einfach beschreiben:

Definition 2.23 (Relation) *Sei M eine Menge, und sei $\trianglelefteq \subseteq M \times M$. Dann nennen wir $\trianglelefteq$ eine* Relation auf M *und verwenden die Schreibweise*

$$a \trianglelefteq b :\iff (a, b) \in \trianglelefteq \qquad \text{für alle } a, b \in M.$$

Für unsere Zwecke soll $x \trianglelefteq y$ an die Stelle von $x \leq y$ treten, wir müssen also untersuchen, welche Eigenschaften die Relation $\trianglelefteq$ aufweisen muss, damit unser Algorithmus weiterhin funktioniert.

Angenommen, eine Menge M und eine Relation $\trianglelefteq$ sind gegeben. Wir wollen in einer Folge $x_0, x_1, \ldots, x_{n-1} \in M$ suchen und setzen voraus, dass sie in unserem verallgemeinerten Sinn sortiert ist, dass nämlich

$$x_0 \trianglelefteq x_1 \trianglelefteq \ldots \trianglelefteq x_{n-2} \trianglelefteq x_{n-1}$$

gilt. Um die Anwendbarkeit unseres Algorithmus' nicht unnötig einzuschränken, sollten wir zulassen, dass einzelne Elemente mehrfach in der Folge auftreten, und sie in diesem Fall in der üblichen Weise einsortieren: Zwei gleiche Elemente sind auch kleiner oder gleich.

Definition 2.24 (Reflexiv) *Eine Relation $\trianglelefteq$ auf einer Menge M heißt* reflexiv, *falls*

$$a \trianglelefteq a \qquad \text{für alle } a \in M \text{ gilt.} \tag{2.17}$$

Die binäre Suche beruht darauf, dass wir ein $k \in \{0, \ldots, n-1\}$ wählen und zunächst prüfen, ob $x_k = y$ gilt. Falls ja, sind wir fertig.

Falls nein, prüfen wir, ob $x_k \trianglelefteq y$ gilt. In diesem Fall sind wir bei unserem Algorithmus davon ausgegangen, dass y nur noch unter den Elementen $x_{k+1}, \ldots, x_{n-1}$ vorkommen kann, so dass es zulässig ist, unsere Suche auf diese Menge einzuschränken.

Wir wissen bereits, dass wir x_k ausschließen dürfen, denn wir haben bereits sichergestellt, dass $x_k \neq y$ gilt. Wenn wir x_{k-1} ebenfalls ausschließen möchten, müssten wir wissen, dass aus

$$x_{k-1} \trianglelefteq x_k, \qquad x_k \trianglelefteq y \qquad \text{und} \qquad x_k \neq y$$

bereits $x_{k-1} \neq y$ folgt. Dieses Ziel können wir erreichen, wenn wir eine zusätzliche Forderung an die Relation stellen:

Definition 2.25 (Antisymmetrisch) *Eine Relation $\trianglelefteq$ auf einer Menge M heißt* antisymmetrisch*, falls*

$$(a \trianglelefteq b) \wedge (b \trianglelefteq a) \Rightarrow a = b \qquad \text{für alle } a, b \in M \text{ gilt.} \tag{2.18}$$

Für eine antisymmetrische Relation $\trianglelefteq$ können wir wie folgt argumentieren: Falls $x_{k-1} = y$ gilt, folgt $x_k \trianglelefteq y = x_{k-1}$, und $x_{k-1} \trianglelefteq x_k$ zusammen mit der Antisymmetrie ergibt $x_k = x_{k-1} = y$. Per Kontraposition muss $x_k \neq y$ dann auch $x_{k-1} \neq y$ implizieren, so dass wir auch x_{k-1} ausschließen dürfen.

Leider können wir uns nicht ohne Weiteres induktiv weiter zu x_{k-2} vorarbeiten, weil wir für unser Argument auf $x_k \trianglelefteq y$ zurückgegriffen haben und jetzt $x_{k-1} \trianglelefteq y$ bräuchten. Wir wissen allerdings, dass $x_{k-1} \trianglelefteq x_k$ gilt, also wäre es günstig, wenn wir aus

$$x_{k-1} \trianglelefteq x_k \qquad \text{und} \qquad x_k \trianglelefteq y$$

bereits auf $x_{k-1} \trianglelefteq y$ schließen dürften.

Definition 2.26 (Transitiv) *Eine Relation $\trianglelefteq$ auf einer Menge M heißt* transitiv*, falls*

$$(a \trianglelefteq b) \wedge (b \trianglelefteq c) \Rightarrow a \trianglelefteq c \qquad \text{für alle } a, b, c \in M \text{ gilt.} \tag{2.19}$$

Falls $\trianglelefteq$ auch transitiv ist, können wir in der bereits beschriebenen Weise fortfahren und tatsächlich beweisen, dass $x_j \neq y$ für alle $j \in \{0, \ldots, k\}$ gelten muss.

Bei der binären Suche müssen wir, falls $x_k \not\trianglelefteq y$ gilt, eine Möglichkeit haben, die Indizes $\{k, \ldots, n-1\}$ auszuschließen. Ohne weitere Voraussetzungen an die Relation $\trianglelefteq$ fehlt uns dazu jegliche Grundlage, deshalb stellen wir noch eine letzte Forderung an die Relation $\trianglelefteq$.

Definition 2.27 (Total) *Eine Relation $\trianglelefteq$ auf einer Menge M heißt* total*, falls*

$$(a \trianglelefteq b) \vee (b \trianglelefteq a) \qquad \text{für alle } a, b \in M \text{ gilt.} \tag{2.20}$$

Mit Hilfe dieser Voraussetzungen können wir aus $x_k \not\trianglelefteq y$ bereits auf $y \trianglelefteq x_k$ schließen. Per Transitivität erhalten wir $y \trianglelefteq x_j$ und per Antisymmetrie $x_j \neq y$ für alle $j \in \{k, \ldots, n-1\}$.

Definition 2.28 (Ordnung) *Sei M eine Menge. Eine Relation $\trianglelefteq \subseteq M \times M$ auf M nennen wir eine* partielle Ordnung auf M *falls sie reflexiv, antisymmetrisch und transitiv ist, falls also (2.17), (2.18) und (2.19) gelten, nämlich*

$$\begin{aligned} a &\trianglelefteq a && \text{für alle } a \in M, \\ (a \trianglelefteq b) \wedge (b \trianglelefteq a) &\Rightarrow a = b && \text{für alle } a, b \in M, \\ (a \trianglelefteq b) \wedge (b \trianglelefteq c) &\Rightarrow a \trianglelefteq c && \text{für alle } a, b, c \in M. \end{aligned}$$

Wir nennen eine Relation $\trianglelefteq$ auf M eine totale Ordnung auf M*, falls sie eine partielle Ordnung und total ist, falls also zusätzlich (2.20) gilt, nämlich*

$$a \trianglelefteq b \vee b \trianglelefteq a \qquad \text{für alle } a, b \in M.$$

Falls $\trianglelefteq$ eine Ordnung auf einer Menge M ist, definieren wir in Anlehnung an die für die Ordnung $\leq$ auf den reellen Zahlen üblichen Konventionen die Abkürzungen

$$x \triangleleft y :\Longleftrightarrow (x \trianglelefteq y \wedge x \neq y), \qquad x \not\trianglelefteq y :\Longleftrightarrow \neg(x \trianglelefteq y) \qquad \text{für alle } x, y \in M.$$

Viele der für $\leq$ und $<$ bekannten Argumente übertragen sich unmittelbar auf $\trianglelefteq$ und $\triangleleft$. Beispielsweise gilt

$$x \triangleleft y \trianglelefteq z \Rightarrow x \triangleleft z \qquad \text{für alle } x, y, z \in M, \tag{2.21}$$

denn mit der Transitivität (2.19) folgt aus $x \trianglelefteq y$ und $y \trianglelefteq z$ bereits $x \trianglelefteq z$. Dank der Antisymmetrie (2.18) würde aus $x = z$ wegen $x \trianglelefteq y$ und $y \trianglelefteq z = x$ bereits $x = y$ folgen, und da das ausgeschlossen ist, erhalten wir auch $x \neq z$.

Wir können die für die binäre Suche erforderlichen Eigenschaften mit der bereits skizzierten Methode beweisen. Als Vorbereitung benötigen wir die folgende Aussage über die Anordnung der Elemente einer sortierten Folge:

Lemma 2.29 (Geordnete Folge) *Sei $\trianglelefteq$ eine totale Ordnung auf M. Seien $n \in \mathbb{N}$ und $x_0, \ldots, x_{n-1} \in M$ mit*

$$x_0 \trianglelefteq x_1 \trianglelefteq \ldots \trianglelefteq x_{n-2} \trianglelefteq x_{n-1} \tag{2.22}$$

gegeben. Dann gilt

$$i \leq j \Rightarrow x_i \trianglelefteq x_j \qquad \text{für alle } i, j \in \{0, \ldots, n-1\}. \tag{2.23}$$

Beweis. Wir beweisen für alle $\ell \in \mathbb{N}_0$ die Aussage

$$i \leq j \Rightarrow x_i \trianglelefteq x_j \qquad \text{für alle } i, j \in \{0, \ldots, n-1\} \text{ mit } j - i = \ell \tag{2.24}$$

per Induktion über ℓ. Damit ist dann auch (2.23) bewiesen.

Induktionsanfang. Seien $i, j \in \{0, \ldots, n-1\}$ mit $j - i = 0$ gegeben. Dann gilt offenbar $i = j$. Da $\trianglelefteq$ reflexiv ist, also (2.17) gilt, folgt $x_i \trianglelefteq x_i = x_j$.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gegeben, dass (2.24) für alle $i, j \in \{0, \ldots, n-1\}$ mit $j - i = \ell$ gilt.

Induktionsschritt. Wir müssen nachweisen, dass (2.24) für alle $i, j \in \{0, \ldots, n-1\}$ mit $j - i = \ell + 1$ gilt. Für $\ell + 1 \geq n$ gibt es keine solchen i und j, also ist die Aussage trivial erfüllt.

Wir brauchen also nur den Fall $\ell + 1 < n$ zu untersuchen. Seien $i, j \in \{0, \ldots, n-1\}$ mit $j - i = \ell + 1$ gegeben. Dann folgen $j - (i + 1) = \ell$ und damit auch $i + 1 \leq j \leq n - 1$, also können wir die Induktionsvoraussetzung anwenden, um $x_{i+1} \trianglelefteq x_j$ zu erhalten. Wegen (2.22) gilt auch $x_i \trianglelefteq x_{i+1}$, und da $\trianglelefteq$ transitiv ist, folgt mit (2.19) auch schon $x_i \trianglelefteq x_j$. ∎

Mit Hilfe dieses Lemmas können wir nun beweisen, dass der von uns beschriebene Algorithmus für die binäre Suche in einer sortierten Folge auch für allgemeine Mengen mit allgemeinen totalen Ordnungen korrekt arbeitet.

Lemma 2.30 (Binäre Suche) *Sei $\trianglelefteq$ eine totale Ordnung auf M. Seien $n \in \mathbb{N}$ und $x_0, \ldots, x_{n-1} \in M$ mit (2.22) gegeben. Sei $y \in M$, und sei $k \in \{0, \ldots, n-1\}$ gegeben. Falls $x_k \triangleleft y$ gilt, folgt*

$$x_j \neq y \qquad \text{für alle } j \in \{0, \ldots, k\}. \tag{2.25a}$$

Falls $y \triangleleft x_k$ gilt, folgt

$$x_j \neq y \qquad \text{für alle } j \in \{k, \ldots, n-1\}. \tag{2.25b}$$

Beweis. Mit Lemma 2.29 erhalten wir (2.23).

Wir untersuchen zunächst (2.25a). Gelte $x_k \trianglelefteq y$ und sei $j \in \{0, \ldots, k\}$ gegeben. Wir müssen beweisen, dass aus $x_k \neq y$ auch $x_j \neq y$ folgt. Per Kontraposition ist das äquivalent dazu, dass aus $x_j = y$ auch $x_k = y$ folgt.

Gelte also $x_j = y$. Nach Voraussetzung folgt daraus $x_k \trianglelefteq y = x_j$. Dank (2.23) haben wir auch $x_j \trianglelefteq x_k$, so dass mit der Antisymmetrie (2.18) bereits $x_k = x_j = y$ folgt.

Für den Beweis von (2.25b) gehen wir analog vor: Gelte $y \trianglelefteq x_k$ und sei $j \in \{k, \ldots, n-1\}$ gegeben. Wir berufen uns wieder auf ein Kontrapositionsargument, so dass wir nur beweisen müssen, dass aus $x_j = y$ bereits $x_k = y$ folgt.

Gelte also $x_j = y$. Nach Voraussetzung folgt daraus $x_j = y \trianglelefteq x_k$. Dank (2.23) haben wir auch $x_k \trianglelefteq x_j$, so dass mit der Antisymmetrie (2.18) bereits $x_k = x_j = y$ folgt. ∎

In ähnlicher Weise können wir auch für `insertionsort` und `mergesort` beweisen, dass die Algorithmen auch für allgemeine geordnete Mengen korrekt arbeiten.

Ein typisches Beispiel für eine Ordnung ist die, die in einem Lexikon zum Einsatz kommt, wenn eine Folge von Wörtern zu sortieren ist: Wir vergleichen zwei Wörter u und v. Falls der erste Buchstabe von u im Alphabet vor dem von v steht, ist u „kleiner“ als v. Falls der erste Buchstabe von u im Alphabet nach dem von v steht, ist v „kleiner“ als u. Falls die ersten Buchstaben von u und v gleich sind, wiederholen wir die Prozedur mit ihren jeweils nächsten Buchstaben. Falls wir dabei einen Punkt erreichen, an dem ein Wort keine Buchstaben mehr aufweist, ist es ebenfalls „kleiner“ als das andere.

Um die korrespondierende *lexikographische Ordnung* korrekt beschreiben zu können, müssen wir zunächst definieren, was wir überhaupt unter einem „Wort“ verstehen.

Definition 2.31 (Wörter) *Sei A eine Menge, und sei $\epsilon := \emptyset$. Für ein $k \in \mathbb{N}$ bezeichnen wir mit A^k die Menge der k-Tupel aus A und verwenden die Konvention $A^0 = \{\epsilon\}$. Die Menge*

$$A^* := \bigcup_{k=0}^{\infty} A^k$$

nennen wir die Menge der Wörter *über dem* Alphabet *A. ϵ nennen wir das* leere Wort.

Die *Länge* eines Worts definieren wir durch

$$|u| := \begin{cases} 0 & \text{falls } u = \epsilon, \\ k & \text{falls } u \in A^k \text{ für ein } k \in \mathbb{N} \end{cases} \qquad \text{für alle } u \in A^*.$$

Die einzelnen Buchstaben eines Worts u bezeichnen wir mit $u_0, u_1, \ldots, u_{|u|-1} \in A$.

Für die bereits umgangssprachlich beschriebene lexikographische Ordnung ist der erste Buchstabe relevant, in dem sich die beiden zu vergleichenden Wörter unterscheiden. Seine Position können wir mathematisch durch

$$p(u,v) := \max\{k \in \mathbb{N}_0 \ : \ k \leq |u|,\ k \leq |v|,\ u_i = v_i \text{ für alle } i < k\} \quad \text{für alle } u, v \in A^*$$

beschreiben und so die folgende Definition erhalten:

Definition 2.32 (Lexikographische Ordnung) *Sei A eine Menge und $\trianglelefteq$ eine Ordnung auf A. Die durch*

$$\begin{aligned} u \preceq v :\Longleftrightarrow\ & (p(u,v) = |u| \\ & \vee\ (p(u,v) < |u| \wedge p(u,v) < |v| \wedge u_{p(u,v)} \triangleleft v_{p(u,v)})) \qquad \textit{für alle } u, v \in A^* \end{aligned}$$

definierte Relation auf A^* nennen wir die zu $\trianglelefteq$ gehörende lexikographische Ordnung.

Die Definition entspricht unserer anschaulichen Beschreibung: Falls $p(u,v) = |u|$ gilt, folgt entweder $u = v$ oder v entsteht aus u durch Anhängen weiterer Elemente. Falls $p(u,v) < |u|$ und $p(u,v) < |v|$ gelten, unterscheiden sich die Wörter u und v in ihrer $p(u,v)$-ten Komponente, und diese Komponente entscheidet darüber, welches der beiden Wörter im Sinn der lexikographischen Ordnung „größer“ als das andere ist.

Wenn wir unsere Aussagen über das Suchen in und Sortieren von geordneten Mengen auf Mengen von Wörtern anwenden wollen, müssen wir nachweisen, dass die lexikographische Ordnung tatsächlich eine Ordnung ist.

Lemma 2.33 (Lexikographische Ordnung) *Sei A eine Menge und $\trianglelefteq$ eine Ordnung auf A. Dann ist die korrespondierende lexikographische Ordnung $\preceq$ auf A^* eine Ordnung.*

Beweis. Gemäß Definition 2.28 müssen wir nachprüfen, dass $\preceq$ reflexiv, antisymmetrisch, transitiv und total ist.

Reflexiv. Zunächst widmen wir uns der Reflexivität. Sei $u \in A^*$. Nach Definition gilt dann $p(u,u) = |u|$, also auch $u \preceq u$. Damit ist (2.17) erfüllt.

Antisymmetrisch. Als nächstes untersuchen wir die Antisymmetrie. Seien dazu $u, v \in A^*$ mit $u \preceq v$ gegeben. Wir wollen beweisen, dass aus $v \preceq u$ bereits $u = v$ folgt. Diesen Beweis führen wir per Kontraposition, wir beweisen also, dass wir aus $u \neq v$ auf $v \not\preceq u$ schließen dürfen. Dazu setzen wir $k := p(u,v) = p(v,u)$ und unterscheiden zwei Fälle:

Falls $k = |u|$ gilt, folgt wegen $u \neq v$ bereits $|v| > |u| = k$, also $p(v,u) \neq |v|$ sowie $p(v,u) \not< |u|$ und damit $v \not\preceq u$.

Falls $k < |u|$ gilt, müssen wegen $u \preceq v$ auch $k < |v|$ und $u_k \triangleleft v_k$ gelten. Da die Relation $\trianglelefteq$ antisymmetrisch ist, folgt $v_k \not\trianglelefteq u_k$ und damit wieder $v \not\preceq u$.

Also folgt aus $u \neq v$ auch $v \not\preceq u$, und damit ist (2.18) erfüllt.

Transitiv. Wenden wir uns nun dem Nachweis der Transitivität zu. Seien dazu $u, v, w \in A^*$ mit $u \preceq v$ und $v \preceq w$ gegeben. Sei $k := \min\{p(u,v), p(v,w)\}$. Dann folgt $u_i = v_i = w_i$ für alle $i < k$, also insbesondere $p(u,w) \geq k$.

1. Fall: Falls $k = |u|$ gilt, folgt mit

$$|u| = k \leq p(u, w) \leq |u|$$

bereits $p(u, w) = |u|$, also $u \preceq w$.

2. Fall: Falls $k = |v|$ gilt, folgt mit

$$|v| = k \leq p(u, v) \leq |v|$$

bereits $p(u, v) = |v|$, also $v \preceq u$ und mit der bereits bewiesenen Antisymmetrie also $u = v$ und somit $u = v \preceq w$.

3. Fall: Falls $k = |w|$ gilt, folgt mit

$$|w| = k \leq p(v, w) \leq |w|$$

bereits $p(v, w) = |w|$, also $w \preceq v$ und mit der bereits bewiesenen Antisymmetrie also $v = w$ und somit $u \preceq v = w$.

4. Fall: Ansonsten müssen $k < |u|$, $k < |v|$ und $k < |w|$ gelten. Falls $p(u, v) < p(v, w)$ gilt, erhalten wir $k = p(u, v)$ und wegen $u \preceq v$ bereits $u_k \lhd v_k = w_k$, also $u \preceq w$. Falls $p(v, w) < p(u, v)$ gilt, erhalten wir $k = p(v, w)$ und wegen $v \preceq w$ bereits $u_k = v_k \lhd w_k$, also $u \preceq w$. Falls $p(v, w) = p(u, v)$ gilt, folgt aus $u \preceq v$ und $v \preceq w$ bereits $u_k \lhd v_k$ sowie $v_k \lhd w_k$. Mit (2.21) schließen wir daraus auf $u_k \lhd w_k$, also $v \preceq w$.

Total. Schließlich müssen wir noch nachprüfen, dass wir nicht nur eine partielle, sondern eine totale Ordnung erhalten haben. Seien also $u, v \in A^*$, und sei $k := p(u, v)$. Falls $k = |u|$ gilt, folgt per Definition $u \preceq v$. Falls $k = |v|$ gilt, erhalten wir entsprechend $v \preceq u$. Anderenfalls muss $u_k \neq v_k$ gelten, und da $\trianglelefteq$ eine totale Ordnung ist, folgt daraus entweder $u_k \lhd v_k$ oder $v_k \lhd u_k$, also $u \preceq v$ oder $v \preceq u$. ■

In C werden Vergleichsoperationen häufig als Funktion implementiert, der die zu vergleichenden Objekte u und v als Parameter erhält und für $u \lhd v$ einen negativen Wert, für $v \lhd u$ einen positiven und für $u = v$ null zurück geben.

```
1  int
2  string_compare(const char *x, const char *y)
3  {
4    while(*x && *y && *x == *y) {
5      x++; y++;
6    }
7    return (*x) - (*y);
8  }
```

Abbildung 2.14: Lexikographischer Vergleich zweier Strings

In Abbildung 2.14 ist der lexikographische Vergleich zweier Strings als Beispiel angegeben. Die `while`-Schleife sucht nach dem ersten Zeichen, in dem sich beide Strings unterscheiden, und die Differenz dieser beiden Zeichen wird dann als Ergebnis zurückgegeben.

Der Algorithmus kann unerwartete Ergebnisse zurückgeben, falls `char` vorzeichenbehaftet ist: Beispielsweise wir der Buchstabe „a“ nach dem ISO/IEC-Standard 8859-1 durch die Zahl 95 dargestellt, der Buchstabe „ö“ dagegen durch die Zahl 246, die als vorzeichenbehaftete 8-Bit-Zahl interpretiert gerade -10 entspricht. Bei unserem Vergleich würde dann „ö“ vor „a“ einsortiert, und das entspricht sicherlich nicht den für Wörterbücher üblichen Regeln.

2.8 Korrektheit

Auch ein sehr schnell arbeitender Algorithmus ist relativ nutzlos, falls er nicht das korrekte Ergebnis findet. Neben der Effizienz ist deshalb auch die *Korrektheit* von Algorithmen eine wesentliche Eigenschaft, die vor allem bei der Entwicklung neuer Algorithmen eine entscheidende Rolle spielt.

Die Korrektheit eines Algorithmus' wird häufig beschrieben, indem man eine *Vorbedingung* und eine *Nachbedingung* angibt. Mit der Vorbedingung formulieren wir die Voraussetzungen, die wir an die dem Algorithmus übergebenen Daten stellen. Mit der Nachbedingung beschreiben wir die Eigenschaften, die das Ergebnis des Algorithmus' aufweisen muss.

Wir nennen den Algorithmus *partiell korrekt*, falls für alle Eingabedaten, die die Vorbedingung erfüllen, nach Abschluss des Algorithmus' die Nachbedingung erfüllt ist.

Unsere Definition der partiellen Korrektheit lässt es zu, dass der Algorithmus unendlich lange rechnet und niemals fertig wird, denn dann braucht er auch nicht die Nachbedingung zu erfüllen. In der Praxis wird ein derartiger Algorithmus meistens nutzlos sein, denn wir erhalten möglicherweise nie ein Ergebnis.

Deshalb bietet es sich an, zusätzlich zu fordern, dass der Algorithmus *terminiert*, dass er also für Eingabedaten, die die Vorbedingung erfüllen, auch nach endlich vielen Operationen fertig wird und die Nachbedingung erfüllt. Einen derartigen Algorithmus nennen wir *total korrekt*.

Beispiel: Binäre Suche. Als Beispiel untersuchen wir die binäre Suche, die in Abbildung 2.15 zusammengefasst ist. Damit der Algorithmus arbeiten kann, müssen die Elemente $x_0, x_1, \ldots, x_{n-1}$ aufsteigend sortiert sein, es muss also

$$x_0 \leq x_1 \leq x_2 \leq \ldots \leq x_{n-1}$$

gelten. Das ist eine sinnvolle Vorbedingung.

Der Rückgabewert k des Algorithmus' soll $k = -1$ erfüllen, falls y nicht Element der Menge $\{x_0, x_1, \ldots, x_{n-1}\}$ ist, oder anderenfalls $k \in \{0, \ldots, n-1\}$ mit $x_k = y$. Eine sinnvolle Nachbedingung lautet also

$$(k = -1 \wedge y \notin \{x_0, \ldots, x_{n-1}\}) \vee (k \in \{0, \ldots, n-1\} \wedge y = x_k). \tag{2.26}$$

Im Fall der binären Suche haben wir bereits im Rahmen der Analyse des Rechenaufwands gezeigt, dass der Algorithmus nur $\mathcal{O}(\log_2(n))$ Operationen erfordert, er terminiert also auf jeden Fall. Zu beweisen ist damit nur noch die partielle Korrektheit.

Für unsere Zwecke genügt ein relativ einfacher Ansatz: Wir formulieren Bedingungen, die an bestimmten Punkten des Algorithmus' gelten müssen, und beweisen, dass diese Bedingungen gültig bleiben, wenn wir die einzelnen Schritte des Algorithmus' ausführen.

```
1  int
2  binary_search(int n, int y, const int *x)
3  {
4    int a, b, k;
5    a = 0; b = n;
6    while(a < b) {
7      k = (b + a) / 2;
8      if(y == x[k])
9        return k;
10     else if(y < x[k])
11       b = k;
12     else
13       a = k + 1;
14   }
15   return -1;
16 }
```

Abbildung 2.15: Binäre Suche in einem Array

In unserem Beispiel besteht der Algorithmus im Wesentlichen aus einer Schleife, also genügt es, eine *Schleifeninvariante* zu formulieren, also eine Bedingung, die in jeder Iteration der Schleife erfüllt sein muss. In unserem Fall muss die Bedingung

$$y \in \{x_0, \dots, x_{n-1}\} \Rightarrow y \in \{x_a, \dots, x_{b-1}\}$$

in Zeile 6 des Algorithmus' gelten: Falls y überhaupt in dem Array vorkommt, muss es in dem Teilarray zwischen den Indizes a und $b-1$ vorkommen.

Unsere Aufgabe ist es nun, zu beweisen, dass diese Bedingung in jeder Iteration gültig ist. Wir bezeichnen wieder mit $a^{(\ell)}$ und $b^{(\ell)}$ die Werte der Variablen `a` und `b` nach dem Ende der ℓ-ten Iteration.

Lemma 2.34 (Schleifeninvariante) *Falls der Algorithmus* `binary_search` *mindestens $m \in \mathbb{N}_0$ Iterationen ausführt, gilt*

$$y \in \{x_0, \dots, x_{n-1}\} \Rightarrow y \in \{x_{a^{(m)}}, \dots, x_{b^{(m)}-1}\}. \tag{2.27}$$

Beweis. Wir führen den Beweis per Induktion.

Induktionsanfang. Gelte $m = 0$. Vor der ersten Iteration wurde in Zeile 5 dafür gesorgt, dass $a^{(0)} = 0$ und $b^{(0)} = n$ gelten, also ist die Schleifeninvariante für $a = a^{(0)}$ und $b = b^{(0)}$ trivial erfüllt.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gewählt, dass die Schleifeninvariante (2.27) für $m = \ell$ gilt.

Induktionsschritt. Wir müssen beweisen, dass die Invariante auch für $m = \ell + 1$ erfüllt ist, falls mindestens $\ell + 1$ Iterationen durchgeführt werden.

Falls $y = x_k$ erfüllt ist, wird der Algorithmus in Zeile 9 beendet und gibt das korrekte Ergebnis zurück, so dass die Nachbedingung (2.26) erfüllt ist. Dann gibt es keine $(\ell+1)$-te Iteration und wir sind bereits fertig.

Anderenfalls prüfen wir, ob $y < x_k$ gilt. Falls ja, muss auch $y < x_i$ für alle $i \geq k$ gelten, so dass y nur in $\{x_a, \ldots, x_{k-1}\}$ liegen kann. Wir setzen $a^{(\ell+1)} = a^{(\ell)}$ und $b^{(\ell+1)} = k$ und wissen, dass die Schleifeninvariante für $m = \ell + 1$ gültig bleibt.

Falls nein, muss $x_k < y$ gelten, so dass wir auch $x_i < y$ für alle $i \leq k$ erhalten, und damit kann y nur in $\{x_{k+1}, \ldots, x_{b-1}\}$ enthalten sein. Wir setzen $a^{(\ell+1)} = k + 1$ und $b^{(\ell+1)} = b^{(\ell)}$ und stellen fest, dass auch in diesem Fall die Schleifeninvariante für $m = \ell + 1$ gültig bleibt. ∎

Falls die Schleife abbricht, falls also $a \geq b$ gilt, ist die Menge $\{x_a, \ldots, x_{b-1}\}$ leer, somit haben wir

$$y \notin \emptyset = \{x_a, \ldots, x_{b-1}\}.$$

Per Kontraposition folgt aus (2.27), dass

$$y \notin \{x_0, \ldots, x_{n-1}\}$$

gelten muss, also ist es richtig, nach Verlassen der Schleife in Zeile 15 den Wert -1 zurückzugeben und so die Nachbedingung (2.26) zu erfüllen.

Damit haben wir die partielle Korrektheit des Algorithmus' bewiesen. Mit der Aufwandsabschätzung aus Satz 2.6 folgt, dass der Algorithmus auch total korrekt ist.

Beispiel: Teilarrays im Quicksort-Algorithmus. Als zweites Beispiel untersuchen wir die entscheidende Phase des Quicksort-Algorithmus', in der das Array $x_0, x_1, \ldots, x_{n-1}$ so umsortiert wird, dass das permutierte Array $x'_0, x'_1, \ldots, x'_{n-1}$ die Bedingung

$$x'_i \leq y < x'_j \qquad \text{für alle } 0 \leq i < m \leq j < n \tag{2.28}$$

erfüllt. Hier ist y wieder das Pivot-Element und m die Größe des ersten Teilarrays.

Der betreffende Abschnitt des Algorithmus' ist in Abbildung 2.16 dargestellt. Er soll für beliebige Arrays funktionieren, also stellen wir keine Vorbedingung, und er soll das Array so umsortieren, dass anschließend (2.28) gilt.

Der Algorithmus ist gerade so konstruiert, dass für alle $i \in \{0, \ldots, m-1\}$ die Eigenschaft $x'_i \leq y$ sichergestellt wird, während für alle $i \in \{j+1, \ldots, n-1\}$ die Eigenschaft $y < x'_i$ gilt. Als Schleifeninvariante formulieren wir also

$$(\forall i \in \{0, \ldots, m-1\} : x'_i \leq y) \wedge (\forall i \in \{j+1, \ldots, n-1\} : y < x'_i). \tag{2.29}$$

Diese Bedingung soll für alle Schleifen gelten, also sowohl in Zeile 2 als auch in Zeile 3 als auch in Zeile 5.

```
1   m = 0; j = n-1;
2   while(m <= j) {
3     while((m <= j) && (x[m] <= y))
4       m++;
5     while((m <= j) && (y < x[j]))
6       j--;
7     if(m < j) {
8       z = x[m]; x[m] = x[j]; x[j] = z;
9       m++; j--;
10    }
11  }
```

Abbildung 2.16: Teil des Quicksort-Algorithmus': Für ein Pivot-Element y wird das Array so umsortiert, dass die Elemente $x_0, \ldots, x_{m-1}$ kleiner oder gleich y sind, während $x_m, \ldots, x_{n-1}$ echt größer sind.

In den inneren Schleifen ist sie immer sicher gestellt, da m in Zeile 4 nur erhöht wird, falls $x'_m = x_m \leq y$ gilt, während j in Zeile 6 nur gesenkt wird, falls $y < x_j = x'_j$ gilt.

Bei Eintritt in die äußere Schleife ist die Invariante trivial erfüllt, da dank Zeile 1 sowohl $m = 0$ als auch $j = n - 1$ gelten und die Schleifeninvariante deshalb nur Bedingungen für leere Mengen aufweist. Da sie durch die inneren Schleifen nicht beeinträchtigt wird, gilt (2.29) auch in Zeile 7. In dieser Zeile gilt allerdings noch mehr: Da wir die erste `while`-Schleife verlassen haben, muss entweder $m > j$ oder $x_m > y$ gelten. Da wir die zweite `while`-Schleife verlassen haben, muss entweder $m > j$ oder $y \geq x_j$ gelten. Sollte die Bedingung $m < j$ in Zeile 7 also erfüllt sein, folgt $x_j \leq y < x_m$. In Zeile 8 tauschen wir x_j und x_m, setzen also $x'_m = x_j$ und $x'_j = x_m$, so dass anschließend

$$x'_m = x_j \leq y < x_m = x'_j$$

gilt. Damit dürfen wir in Zeile 9 m herauf- und j herunterzählen, ohne die Invariante zu verletzen.

Wir verlassen die äußere Schleife, sobald $m > j$ gilt, also $m \geq j + 1$. Damit folgt aus $i \geq m$ auch $i \geq j + 1$, also nach der Invarianten (2.29) bereits $y < x_i$. Damit ist die Nachbedingung (2.28) bewiesen.

Hoare-Kalkül. Wir haben unsere Korrektheitsbeweise relativ informell formuliert. Einen wesentlich präziseren Zugang bietet das auf C. A. R. Hoare zurückgehende *Hoare-Kalkül*, bei dem man ausgehend von Vor- und Nachbedingungen für einzelne Anweisungen induktiv Vor- und Nachbedingungen für größere Anweisungsblöcke konstruiert, bis man die gewünschten Vor- und Nachbedingungen für den gesamten Algorithmus erhalten hat.

Ein *Hoare-Tripel* besteht aus einer Vorbedingung P, einer Nachbedingung Q und einem Programmteil S und wird

$$\{P\}S\{Q\}$$

geschrieben. Wir interpretieren es als „wenn P vor der Ausführung des Programmteils S gilt, gilt Q nach der Ausführung“, also als einen logischen Ausdruck.

Wir können aus Hoare-Tripeln neue Hoare-Tripel zusammensetzen, falls beispielsweise

$$\{P\}S\{R\} \qquad \text{und} \qquad \{R\}T\{Q\}$$

gelten, dürfen wir für das zusammengesetzte Programmstück $S;T$ das Tripel

$$\{P\}S;T\{Q\}$$

formulieren. Eine Fallunterscheidung mit einer Bedingung B, die keine Seiteneffekte aufweist, also keine Variablen verändert, können wir durch Hoare-Tripel wie folgt modellieren: Für die beiden Zweige fordern wir

$$\{P \wedge B\}S\{Q\} \qquad \text{und} \qquad \{P \wedge \neg B\}T\{Q\}$$

und erhalten dann

$$\{P\}\texttt{if}(B)\ S\ \texttt{else}\ T\{Q\}.$$

Die von uns bereits verwendeten Schleifeninvarianten können wir ebenfalls im Hoare-Kalkül ausdrücken: Falls

$$\{P \wedge B\}S\{P\}$$

gilt, folgt

$$\{P\}\texttt{while}(B)\ S\{P \wedge \neg B\}.$$

Als Beispiel untersuchen wir das „ägyptische Potenzieren“, mit dem wir zu $x \in \mathbb{R}$ und $n \in \mathbb{N}_0$ die Potenz $z := x^n$ berechnen. Der Algorithmus ist in Abbildung 2.17 dargestellt und beruht auf der folgenden Idee: Wir stellen n im binären Stellenwertsystem dar, also als

$$n = n_p 2^p + n_{p-1}2^{p-1} + \ldots + n_1 2 + n_0, \qquad \text{mit Ziffern } n_0, \ldots, n_p \in \{0, 1\}.$$

Nach Potenzrechengesetz gilt dann

$$\begin{aligned} x^n &= x^{n_p 2^p + n_{p-1}2^{p-1} + \ldots + n_1 2 + n_0} \\ &= x^{n_p 2^p} x^{n_{p-1}2^{p-1}} \ldots x^{n_1 2} x^{n_0} \\ &= (x^{2^p})^{n_p} (x^{2^{p-1}})^{n_{p-1}} \ldots (x^2)^{n_1} x^{n_0}. \end{aligned}$$

Wir berechnen der Reihe nach x^{2^ℓ} durch wiederholtes Quadrieren der Zahl x und multiplizieren diejenigen Potenzen auf, für die $n_\ell = 1$ gilt.

Wir möchten nachweisen, dass die Schleifeninvariante $z = x^n y$ gilt. Dazu formulieren wir Hoare-Tripel für die einzelnen Anweisungen, die wir so wählen, dass sie zu der angestrebten Invariante passen:

$\{x^n y = z\}$

```
    if(n%2) y *= x;
```

```
1  double
2  power(double x, unsigned n)
3  {
4    double y = 1.0;
5    while(n > 0) {
6      if(n%2) y *= x;
7      x *= x;
8      n /= 2;
9    }
10   return y;
11 }
```

Abbildung 2.17: Ägyptischer Algorithmus für die Berechnung der Potenz $y = x^n$

$$\{(n \text{ gerade} \wedge x^n y = z) \vee (n \text{ ungerade} \wedge x^{n-1} y = z)\}$$

ist angemessen für die `if`-Anweisung,

$$\begin{array}{l}\{(n \text{ gerade} \wedge x^n y = z) \vee (n \text{ ungerade} \wedge x^{n-1} y = z)\} \\ \quad \texttt{x *= x;} \\ \{(n \text{ gerade} \wedge x^{n/2} y = z) \vee (n \text{ ungerade} \wedge x^{(n-1)/2} y = z)\}\end{array}$$

passt zu der Anweisung, die `x` quadriert, und

$$\begin{array}{l}\{(n \text{ gerade} \wedge x^{n/2} y = z) \vee (n \text{ ungerade} \wedge x^{(n-1)/2} y = z)\} \\ \quad \texttt{n /= 2;} \\ \{x^n y = z\}\end{array}$$

beschreibt das abgerundete Halbieren der Variable `n`.

Mit den Regeln des Hoare-Kalküls erfüllt die Hintereinanderausführung der drei Anweisungen deshalb

$$\begin{array}{l}\{x^n y = z\} \\ \quad \texttt{if(n\%2) y *= x;} \\ \quad \texttt{x *= x; n /= 2;} \\ \{x^n y = z\}.\end{array}$$

Mit der `while`-Regel folgt

$$\begin{array}{l}\{x^n y = z\} \\ \quad \texttt{while(n>0) \{} \\ \quad\quad \texttt{if(n\%2) y *= x;}\end{array}$$

```
        x *= x; n /= 2;
    }
```

$$\{(x^n y = z) \wedge n = 0\},$$

und die Nachbedingung impliziert $y = z$. Die Vorbedingung $x^n y = z$ ist erfüllt, da in Zeile 4 die Variable `y` auf den Wert 1 gesetzt wird.

2.9 Zusammenfassung

In diesem Kapitel haben wir anhand einiger einfacher Beispiele untersucht, welche Eigenschaften von Algorithmen für uns von Interesse sein könnten. Zu nennen sind

- Der Rechenaufwand, der im ungünstigsten Fall erforderlich wird,
- bei randomisierten Algorithmen der erwartete Rechenaufwand,
- der Speicherbedarf,
- die Flexibilität des Algorithmus', also ob er sich auf viele Aufgabenstellungen anwenden lässt, sowie
- die Korrektheit des Algorithmus'.

Den Rechenaufwand für den ungünstigsten Fall haben wir bei iterativen Verfahren abgeschätzt, indem wir die Anzahl der Iterationen abgeschätzt und dann die Anzahl der Operationen pro Iteration gezählt haben. Bei rekursiven Verfahren haben wir Rekurrenzformeln erhalten, aus denen wir mit den Lemmas 2.5, 2.11, 2.14 sowie 2.17 Aussagen über den Rechenaufwand gewinnen konnten.

Bei randomisierten Verfahren müssen wir von Annahmen über die Wahrscheinlichkeit der möglichen Parameterwerte ausgehen, um eine Aussage über den erwarteten Rechenaufwand zu erhalten. Im Fall des randomisierten Quicksort-Verfahrens haben wir so wieder eine Rekurrenzformel erhalten, die zu einer Abschätzung führte.

Der Bedarf an Hilfsspeicher ist entscheidend, wenn mit sehr großen Datenmengen gearbeitet werden soll und zu erwarten ist, dass der verfügbare Speicher knapp bemessen sein wird. Für rekursive Algorithmen wie `mergesort` lässt sich auch in diesem Fall eine Rekurrenzformel angeben, mit der sich der Speicherbedarf abschätzen lässt.

Ein Algorithmus ist natürlich um so nützlicher, je mehr Aufgaben sich mit ihm lösen lassen. Im Fall der in diesem Kapitel diskutierten Such- und Sortierverfahren haben wir gesehen, dass sie sich nicht nur für Mengen von Zahlen eignen, sondern sich mit geringem Aufwand auch auf allgemeine total geordnete Mengen übertragen lassen.

Schließlich ist natürlich von Interesse, ob der Algorithmus das richtige Ergebnis berechnet. Wir haben einen einfachen Ansatz kennen gelernt, mit dem sich die Korrektheit von Algorithmen beweisen lässt, indem man dem Algorithmus zusätzliche logische Bedingungen hinzufügt und untersucht, ob sie bei Ausführung des Algorithmus' gültig bleiben.

3 Grundlegende Datenstrukturen

Für die Effizienz eines Algorithmus' ist es von entscheidender Bedeutung, in welcher Form die zu verarbeitenden Daten vorliegen. Beispielsweise lässt sich das Einfügen eines neuen Datensatzes zu Beginn eines Arrays nicht bewerkstelligen, ohne das gesamte Array zu kopieren, während diese Aufgabe bei einer Liste in wenigen Operationen ausgeführt werden kann.

3.1 Listen

Ein Array weist in der Regel eine feste Größe auf, die bei der Anforderung des korrespondierenden Speicherbereichs festgelegt wurde. Datenstrukturen, deren Größe sich während der Laufzeit eines Algorithmus' nicht verändert, nennen wir *statisch*.

Dynamische Datenstrukturen dagegen können ihre Größe verändern und sich so den Bedürfnissen des Algorithmus' besser anpassen. Der Preis dafür ist in der Regel ein höherer Verwaltungsaufwand, der sich häufig auch in einem höheren Speicherbedarf niederschlägt.

Das einfachste Beispiel für eine dynamische Datenstruktur ist die *einfach verkettete Liste*, bei der wir einen Listeneintrag durch einen Zeiger auf die zu ihm gehörenden Daten und einen Zeiger auf das nächste Element der Liste darstellen. In der Sprache C können wir einen derartigen Datensatz mit einer Typdefinition der folgenden Bauart repräsentieren:

```
1 typedef struct _listelement listelement;
2 struct _listelement {
3   payload x;
4   listelement *next;
5 };
```

In diesem Fall bezeichnet `payload` den C-Typ der Daten, die wir in der Liste speichern wollen. In unseren Beispielen wird `payload` in der Regel dem vordefinierten Typ `int` entsprechen. `next` zeigt entweder auf das nächste Element der Liste oder ist gleich null, falls wir das Ende der Liste erreicht haben. Die unvollständige Definition in Zeile 1 erlaubt es uns, in Zeile 4 Zeiger auf den noch unvollständigen Datentyp `listelement` in ebendiesen Datentyp aufzunehmen.

Bemerkung 3.1 (Nutzdaten) *Bei der Definition des Typs* `payload` *können wir zwei Strategien verfolgen: Der Typ kann die zu speichernden Daten repräsentieren oder lediglich einen Zeiger auf diese Daten.*

Die erste Variante ist von Vorteil, falls die Liste nur relativ kleine Datensätze aufnehmen soll, die sich schnell kopieren lassen, denn dann liegen Daten und `next`*-Zeiger im Speicher nahe beieinander, so dass der Prozessor besonders schnell auf sie zugreifen kann.*

Die zweite Variante empfiehlt sich für große Datensätze, bei denen ein Kopiervorgang wesentlich länger als das Setzen eines Zeigers dauern würde.

Eine Liste können wir durch einen Zeiger auf ihr erstes Element, den *Kopf* der Liste, darstellen. Im einfachsten Fall wächst die Liste, indem wir vor dem aktuellen Kopf der Liste ein neues Element einfügen, das der neue Kopf wird und dessen `next`-Zeiger auf den alten Kopf verweist. Ein Beispiel ist in Abbildung 3.1 gegeben.

```
1  listelement *
2  new_listelement(payload x, listelement *next)
3  {
4    listelement *e;
5    e = (listelement *) malloc(sizeof(listelement));
6    e->x = x; e->next = next;
7    return e;
8  }
9  listelement *
10 del_listelement(listelement *e)
11 {
12   listelement *next;
13   next = e->next;
14   free(e);
15   return next;
16 }
17 int
18 main()
19 {
20   listelement *head;
21   head = new_listelement(1, 0);
22   head = new_listelement(2, head);
23   head = new_listelement(3, head);
24   while(head) head = del_listelement(head);
25   return 0;
26 }
```

Abbildung 3.1: Aufbau einer einfach verketteten Liste vom Kopf aus.

Die Funktion `new_listelement` fordert mit `malloc` Speicher für ein Listenelement an, setzt seine Felder auf geeignete Werte, und gibt einen Zeiger auf das Ergebnis zurück. Für derartige Funktionen ist die Bezeichnung *Konstruktor* üblich. Es ist häufig sehr

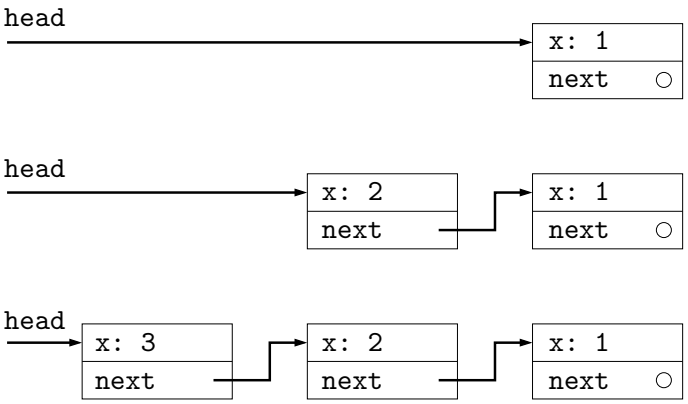


Abbildung 3.2: Aufbau einer einfach verketteten Liste vom Kopf aus. Nullzeiger sind durch Kreise dargestellt.

empfehlenswert, Elemente einer dynamischen Datenstruktur grundsätzlich über einen einzigen oder möglichst wenige Konstruktoren anzulegen, um die Wartung und Weiterentwicklung des resultierenden Programms zu vereinfachen.

Das Gegenstück eines Konstruktors ist ein *Destruktor*, nämlich eine Funktion, die den für ein Datenelement reservierten Speicher frei gibt und eventuelle sonstige Aufräumarbeiten erledigt. In unserem Fall gibt `del_listelement` den für ein `listelement` reservierten Speicher wieder frei. Um das Löschen einer vollständigen Liste zu vereinfachen, gibt `del_listelement` in unserem Fall auch den `next`-Zeiger des gelöschten Elements zurück, so dass wir mit der in Zeile 25 dargestellten Schleife elegant den gesamten von einer Liste belegten Speicher freigeben können. Dabei nutzen wir aus, dass die `while`-Schleife läuft, bis die Schleifenbedingung null ist: In unserem Fall passiert das genau dann, wenn wir den Nullzeiger erreichen, der nach unserer Konvention das Ende der Liste markiert.

Das in der Funktion `main` enthaltene Hauptprogramm erstellt unter Verwendung des Konstruktors eine einfach verkettete Liste mit den drei Werten 1, 2 und 3. Hierbei gehen wir davon aus, dass `payload` ein `int`-Typ ist. Die nach Ausführung der Zeilen 16, 17 und 18 entstandenen Listen sind in Abbildung 3.2 dargestellt. Es ist zu beachten, dass bei dieser Form der Listenkonstruktion die Elemente in einer Reihenfolge erscheinen, die genau entgegengesetzt zu der Reihenfolge verläuft, in der sie eingefügt wurden: In unserem Beispiel wurde `x3` als letztes eingefügt, befindet sich aber am Kopf der Liste, während das zuerst eingefügte Element `x1` am Ende der Liste steht. Je nach der geplanten Anwendung kann dieses Verhalten der vom Kopf aus wachsenden Liste ein Vor- oder ein Nachteil sein.

Beispiel: Suchen mit mehreren Ergebnissen. Eine einfache Anwendung einer einfach verketteten Liste ist eine Funktion, die mehrere Ergebnisse zurückgeben kann. Als Beispiel nehmen wir die lineare Suche in einem Array des Typs `payload`. Um einzelne Element dieses Typs miteinander vergleichen zu können, nehmen wir an, dass eine Funk-

tion

```
1 int
2 compare(payload x, payload y);
```

vorhanden ist, die (ähnlich den Standard-Bibliotheksfunktionen `strcmp` und `memcmp`) die beiden `payload`-Variablen `x` und `y` vergleicht, und einen negativen Wert zurückgibt, wenn der Wert der ersten kleiner als der der zweiten ist, einen positiven, falls er größer ist, und null, falls beide gleich sind. Falls beispielsweise `payload` ein `int`-Typ ist, könnte diese Funktion einfach `x-y` zurückgeben, um die übliche Ordnung auf den ganzen Zahlen zu beschreiben. Allgemeinere totale Ordnungen sind selbstverständlich auch zulässig.

Mit dem folgenden Programmfragment können wir dann in einer Liste mit dem Kopf `head` nach einem Schlüssel `y` suchen und die Ergebnisse in einer neuen Liste mit Kopf `results` zurückgeben:

```
1 listelement *e, *results;
2 results = 0;
3 for(e=head; e; e=e->next)
4   if(compare(e->x, y) == 0)
5     results = new_listelement(e->x, results);
```

Der Algorithmus entspricht der einfachen linearen Suche, die wir bereits kennen gelernt haben. In Zeile 3 nutzen wir aus, dass `e` genau dann gleich null ist, wenn wir das letzte Listenelement erreicht haben, so dass die Schleife beendet wird, weil der C-Konvention entsprechend eine Null als der Wahrheitswert „falsch“ interpretiert wird.

Am Ende wachsende Listen. Falls es wichtig ist, dass die Listenelemente in der Reihenfolge in der Liste stehen, in der sie eingefügt wurden, können wir neben dem Zeiger `head`, der auf den Kopf der Liste zeigt, noch einen zweiten Zeiger `tail` verwenden, der auf das *Ende* der Liste zeigt.

Ein neues Listenelement `e` können wir dann am Ende der Liste einfügen, indem wir `tail->next` auf `e` zeigen lassen und danach `tail` den neuen Wert `e` zuweisen. Natürlich müssen wir dabei darauf achten, ob `tail` überhaupt auf etwas zeigt, schließlich muss auch der Fall einer leeren Liste korrekt gehandhabt werden.

Während wir bei der am Kopf wachsenden Liste lediglich den Zeiger `head` benötigen, müssen wir bei der am Ende wachsenden Liste `head` und `tail` gleichzeitig verwalten, so dass es sich anbietet, neben den Listenelementen auch die Liste selber durch einen Datentyp darzustellen, der `head` und `tail` enthält.

Um sicherzustellen, dass unsere Listen jederzeit in einem definierten Zustand sind und unsere Algorithmen korrekt arbeiten, empfiehlt es sich, eine Funktion zu schreiben, die Speicherplatz für eine Variable des Typs `list` anlegt und die Felder `head` und `tail` korrekt initialisiert.

Selbstverständlich benötigen wir außerdem eine Möglichkeit, der Liste neue Listenelemente hinzuzufügen. Beide Funktionen sind in Abbildung 3.3 zusammengefasst.

Falls wir wieder drei Listenelemente für 1, 2 und 3 der Reihe nach einfügen und dazu diesmal die Funktion `addto_list` einsetzen, werden die Elemente in der Reihe in der

```
1  typedef struct _list list;
2  struct _list {
3    listelement *head;
4    listelement *tail;
5  };
6  list *
7  new_list()
8  {
9    list *li;
10   li = (list *) malloc(sizeof(list));
11   li->head = 0; li->tail = 0;
12   return li;
13 }
14 listelement *
15 addto_list(list *li, payload x)
16 {
17   listelement *e;
18   e = new_listelement(x, 0);
19   if(li->tail)
20     li->tail->next = e;
21   else
22     li->head = e;
23   li->tail = e;
24   return e;
25 }
```

Abbildung 3.3: Funktionen für das Erzeugen einer Datenstruktur für eine einfach verkettete Liste und das Einfügen von Listenelementen an deren Ende.

Liste aufgeführt, in der die Funktion aufgerufen wurde. Das Entstehen der Liste ist in Abbildung 3.4 illustriert. Wie geplant steht das zuletzt eingefügte Element 3 jetzt am Ende der Liste und das zuerst eingefügte 1 an deren Kopf.

Falls wir davon ausgehen, dass der Aufruf der Funktion `malloc` lediglich $\mathcal{O}(1)$ Operationen erfordert, dürfen wir festhalten, dass sowohl `new_listelement` als auch `addto_list` mit $\mathcal{O}(1)$ Operationen auskommen. Mit Lemma 2.20 folgt, dass wir eine Liste mit n Elementen in $\mathcal{O}(n)$ Operationen aufbauen können.

Beispiel: Bucketsort. Mit Hilfe mehrerer am Ende wachsender Listen können wir ein Sortierverfahren entwickeln, das Listen und Arrays sehr viel effizienter als die uns bisher bekannten Algorithmen sortieren kann, falls gewisse zusätzliche Eigenschaften erfüllt sind. Wir gehen dazu davon aus, dass jedes Element des Arrays eine Komponente enthält, nach der wir sortieren wollen. Diese Komponente bezeichnen wir als den *Schlüssel* des jeweiligen Array-Elements. Wir gehen davon aus, dass die Schlüssel nur Werte zwischen

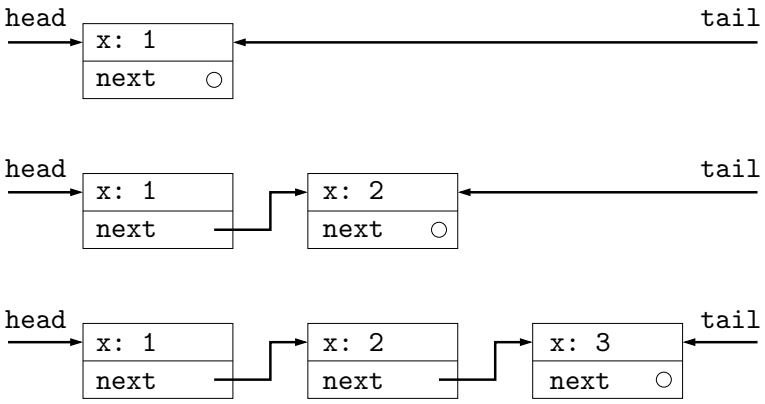


Abbildung 3.4: Aufbau einer einfach verketteten Liste vom Ende aus. Nullzeiger sind durch Kreise dargestellt.

0 und $k-1$ annehmen können, wobei k eine nicht allzu große Zahl ist.

Die Idee des *Bucketsort-Algorithmus'* besteht darin, für jeden der k Werte eine Liste anzulegen und dann die Listenelemente abhängig von ihren Schlüsselwerten zu einer dieser Listen hinzuzufügen. Wenn wir die Listen anschließend einfach aneinander hängen, ist das Ergebnis eine sortierte Liste. Anschaulich werden die Elemente der Liste je nach Schlüssel in „Körbe“ (engl. *buckets*) geworfen und aus den Inhalten dieser „Körbe“ entsteht dann die sortierte Liste.

Aus bestimmten Gründen ist es wünschenswert, dass Listenelemente mit *identischen* Schlüsseln in der sortierten Liste noch in derselben Reihenfolge wie in der ursprünglichen Liste auftreten. Einen Sortieralgorithmus mit dieser Eigenschaft nennen wir *stabil*. Dieses Ziel können wir am einfachsten erreichen, indem wir die „Körbe“ als am Ende wachsende Listen realisieren.

Insgesamt erhalten wir ein Sortierverfahren mit einem Rechenaufwand in $\mathcal{O}(n+k)$, das für hinreichend kleines k und hinreichend großes n wesentlich günstiger als die bisher diskutierten Sortierverfahren sein kann.

Beispiel: Radixsort. Der Nachteil des Bucketsort-Algorithmus' ist die Einschränkung auf nur wenige Schlüsselwerte. Dieser Nachteil lässt sich, zumindest für ganzzahlige Schlüssel, kompensieren, indem wir jeweils nur nach einer *Ziffer* der Schlüssel sortieren. In einem ersten Schritt sortieren wir nach der letzten Ziffer, also nach der mit dem geringsten Wert. Anschließend sortieren wir nach der zweitletzten Ziffer. Falls wir dabei ein stabiles Sortierverfahren wie den Bucketsort-Algorithmus verwenden, bleibt bei identischen zweitletzten Ziffern die im ersten Schritt gewonnene Anordnung erhalten. Wir wiederholen diese Methode, bis wir bei der Ziffer mit der höchsten Wertigkeit angekommen sind. Der so konstruierte Algorithmus trägt den Namen *Radixsort* und erreicht einen Rechenaufwand von $\mathcal{O}(m(n+k))$ für das Sortieren m-stelliger Zahlen.

Damit kann der Radixsort-Algorithmus wesentlich effizienter als alle bisher diskutierten Sortierverfahren arbeiten, falls k und m nicht zu groß werden. Insbesondere hängt

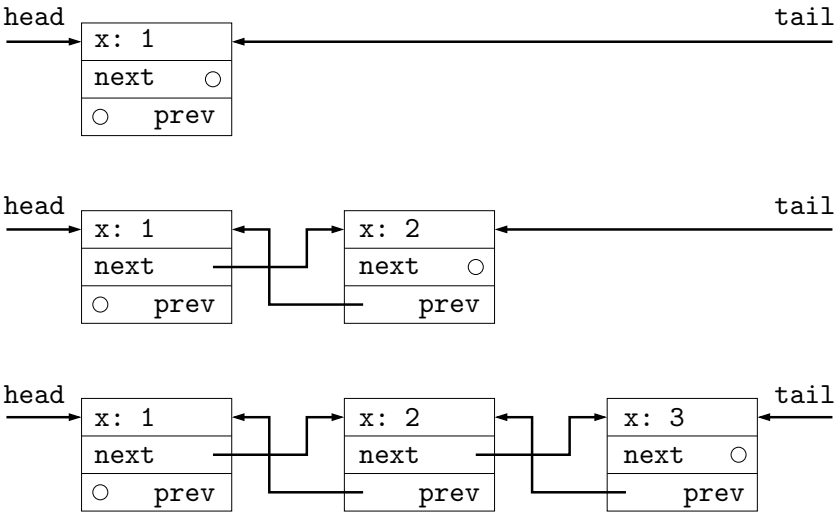


Abbildung 3.5: Aufbau einer doppelt verketteten Liste vom Ende aus. Nullzeiger sind durch Kreise dargestellt.

der Rechenaufwand lediglich *linear* von n ab, während wir bisher einen mindestens zu $n \log_2(n)$ proportionalen Aufwand erreichen konnten.

Der Preis für die höhere Effizienz ist die Einschränkung auf eine spezielle Problemklasse: Es reicht nicht aus, einzelne Elemente der zu sortierenden Datenmenge vergleichen zu können, wir benötigen den Zugriff auf einzelne Ziffern und die Ordnung muss von der Bauart der lexikografischen Ordnung sein.

Doppelt verkettete Listen. Wir haben bereits gesehen, dass es bei einer einfach verketteten Liste sehr einfach ist, das Element am Kopf der Liste zu löschen. Das Löschen eines anderen Elements erweist sich in der Regel als deutlich schwieriger: Wir müssten den `next`-Zeiger seines Vorgängers so anpassen, dass er auf den Nachfolger des zu löschenden Elements zeigt. Es gibt allerdings keine elegante Möglichkeit, diesen Vorgänger zu finden. Wenn wir beispielsweise das Element am Ende der Liste löschen wollen, müssen wir die gesamte Liste durchlaufen, um den Vorgänger zu finden. Bei langen Listen ist diese Vorgehensweise relativ aufwendig und deshalb unattraktiv.

Bei einer *doppelt verketteten Liste* nimmt man deshalb in jedem Listenelement einen weiteren Zeiger auf, der auf das ihm vorausgehende Element verweist. In unserem Fall können wir den Typ `listelement` um einen Zeiger `prev` (nach dem Englischen *previous*) auf das vorangehende Element erweitern. Der Aufbau einer Liste mit dieser erweiterten Zeigerstruktur ist in Abbildung 3.5 dargestellt.

Die Verwaltung des zusätzlichen Zeigers kostet natürlich etwas Zeit und Speicher, erlaubt es uns aber dafür, das Löschen eines Elements aus der Liste sehr effizient zu gestalten: Falls `e` auf ein zu löschendes Element zeigt, können wir es mit `e->next->prev = e->prev` und `e->prev->next=e->next` aus der Liste herausnehmen

und anschließend den zugehörigen Speicher freigeben. Natürlich müssen wir dabei darauf achten, die Sonderfälle eines am Kopf oder Ende der Liste stehenden Elements korrekt zu behandeln.

Die doppelte Verkettung bietet noch weitere Vorteile, beispielsweise können wir die Liste nicht nur vorwärts, sondern auch rückwärts durchlaufen und Elemente nicht nur an beliebiger Stelle löschen, sondern auch einfügen.

3.2 Keller

In vielen Algorithmen ist der konkrete Aufbau der Datenstruktur weniger wichtig als die Operationen, die sich effizient mit ihr durchführen lassen. Beispielsweise lassen sich bei einer am Kopf wachsenden Liste relativ einfach Elemente einfügen und die zuletzt eingefügten Elemente wieder zurückgeben, während der Zugriff auf am Ende der Liste liegende Elemente je nach Länge der Liste einen höheren Rechenaufwand bedeuten kann.

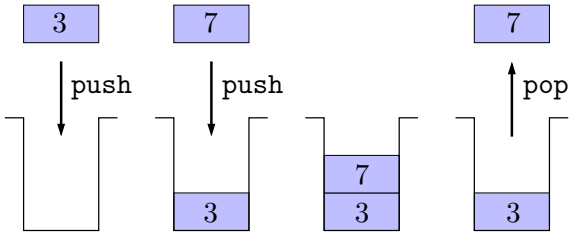


Abbildung 3.6: Darstellung der Push- und Pop-Operation eines Kellerspeicher.

Anschaulich kann man sich diese Datenstruktur als einen Stapel von Elementen vorstellen: Wir können effizient ein neues Element auf dem Stapel legen und das oberste Element von dem Stapel nehmen, aber der Zugriff auf andere Elemente ist aufwendiger. Derartige Strukturen werden als *Kellerspeicher* (im Englischen als *stack* bezeichnet) und müssen zwei grundlegende Operationen bieten: Das Ablegen eines Elements auf dem Stapel (im Englischen als *push* bezeichnet) und das Entfernen des obersten Elements des Stapels (im Englischen *pop*).

Da zuerst das Element zurückgegeben wird, das zuletzt in den Keller gekommen ist, hat sich für Kellerspeicher auch das Kürzel LIFO-Speicher etabliert: LIFO steht für *last in, first out*.

Ein Kellerspeicher kann in unterschiedlicher Weise implementiert werden. Naheliegend ist es natürlich, als grundlegende Datenstruktur eine am Kopf wachsende einfach verkettete Liste zu verwenden, da sie gerade die gewünschte Eigenschaft besitzt, dass die zuletzt hinzugefügten Elemente als erste in der Liste stehen. Eine entsprechende Implementierung ist in Abbildung 3.7 dargestellt. Um die Handhabung zu vereinfachen wurde dabei auch ein neuer Datentyp `stack` definiert, der lediglich den `head`-Zeiger der verwendeten Liste enthält.

Ein Anwender kann die Funktionen `push` und `pop` einsetzen, ohne etwas über den

```
typedef struct _stack stack;
struct _stack {
  listelement *head;
};
stack *
new_stack()
{
  stack *st;
  st = (stack *) malloc(sizeof(stack));
  st->head = 0;
  return st;
}
void
push(stack *st, payload x)
{
  st->head = new_listelement(x, st->head);
}
payload
pop(stack *st)
{
  payload x;
  if(st->head == 0) crash("Stack empty");
  x = st->head->x;
  st->head = del_listelement(st->head);
  return x;
}
payload
peek(stack *st)
{
  if(st->head == 0) crash("Stack empty");
  return st->head->x;
}
int
isempty(stack *st)
{
  return (st->head == 0);
}
```

Abbildung 3.7: Kellerspeicher mit Funktionen für das Hinzufügen und Zurückholen von Elementen.

internen Aufbau des Kellerspeichers wissen zu müssen. Der Ansatz, die konkrete Implementierung zu „verstecken“, bietet häufig Vorteile, weil er es dem Programmierer

ermöglicht, den intern verwendeten Datentyp umzustellen, ohne dass die den Kellerspeicher verwendenden Algorithmen angepasst werden müssen.

Etwas Vorsicht müssen wir bei der Behandlung eines leeren Kellers walten lassen: Falls der Keller leer ist, gibt die Funktion `pop` einen Nullzeiger zurück. Dieser Nullzeiger kann allerdings auch entstanden sein, indem wir ihn explizit per `push` in den Keller befördert haben. Wenn wir also verlässlich prüfen wollen, ob der Keller wirklich leer ist, sollten wir explizit nachsehen, ob der `head`-Zeiger gleich null ist. Alternativ können wir natürlich auch einfach verbieten, dass Nullzeiger mit `push` in den Keller gebracht werden.

Implizit haben wir Kellerspeicher bereits verwendet: Rekursive Funktionen werden durch den Compiler umgesetzt, indem die lokalen Variablen in einem Kellerspeicher in Sicherheit gebracht werden, wenn ein rekursiver Aufruf erfolgt. Nach der Rückkehr von diesem Aufruf können sie dann einfach rekonstruiert werden. Die meisten Rechnersysteme realisieren diesen Kellerspeicher allerdings nicht über eine Liste, sondern über eine Array einer gewissen Größe, bei dem ein Zeiger festhält, bis zu welchem Element es jeweils gefüllt ist. Bei einer Push-Operation werden dann beispielsweise lediglich die Daten in das Array geschrieben und der Zeiger heraufgezählt, bei einer Pop-Operation werden der Zeiger heruntergezählt und die Daten aus dem Array gelesen. Beides lässt sich sehr effizient umsetzen, bei vielen modernen Prozessoren genügt jeweils ein einziger Maschinenbefehl. Der Nachteil dieser Vorgehensweise ist, dass für das Array in der Regel nur eine feste Größe vorgesehen ist, so dass der Kellerspeicher „überlaufen“ kann (engl. *stack overflow*).

3.3 Warteschlange

Auch die Liste, in der wir neue Elemente am Ende einfügen, so dass die Reihenfolge der Elemente erhalten bleibt, lässt sich als Realisierung eines abstrakten Zugriffsmusters interpretieren: Der *Warteschlange* (im Englischen als *queue* bekannt), bei der neue Elemente am Ende eingefügt (im Englischen als *enqueue* bezeichnet) und am Anfang entnommen (im Englischen *dequeue*) werden können.

Im Gegensatz zu einem Kellerspeicher, bei dem wir jeweils nur das „jüngste“ Element aus dem Speicher entfernen können, ist es bei einer Warteschlange das „älteste“ Element.

Die Enqueue- und Dequeue-Operationen sind in Abbildung 3.8 grafisch dargestellt: Neue Elemente werden „von oben“ in die Warteschlange gesteckt und „von unten“ wieder entnommen. Damit sich die Warteschlangen gut zeichnen lassen, stellen wir uns vor, dass beim Entnehmen eines Elements die restlichen Elemente von oben „nachrutschen“.

Da zuerst das Element zurückgegeben wird, das als erstes in die Warteschlange eingereiht wurde, hat sich für Warteschlangen auch das Kürzel FIFO-Speicher etabliert. Hier steht FIFO für *first in, first out*.

Ähnlich einem Kellerspeicher kann auch die Warteschlange unterschiedlich implementiert werden, eine an ihrem Ende wachsende einfach verkettete Liste ist lediglich eine besonders naheliegende Variante, die in Abbildung 3.9 dargestellt ist. Bei dieser Umsetzung verwenden wir `qu->head`, um zu erkennen, ob die Warteschlange ein Element enthält. Diese Vorgehensweise hat den Vorteil, dass wir bei der Dequeue-Operation in

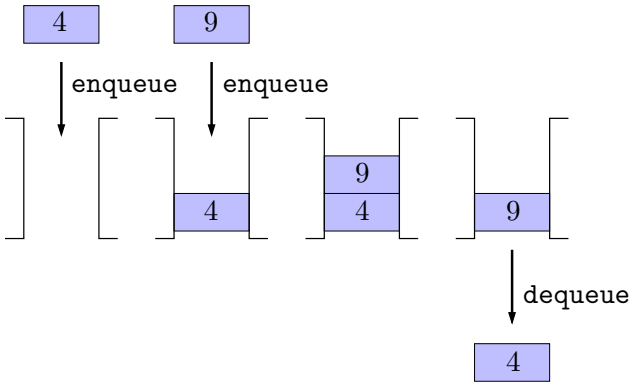


Abbildung 3.8: Darstellung der Enqueue- und Dequeue-Operation einer Warteschlange.

Zeile 31 nicht auch noch `qu->tail` auf null zu setzen brauchen, um die leere Liste zu kennzeichnen. Es kann also geschehen, dass `qu->tail` noch auf bereits ungültig gewordene Elemente verweist. Das ist allerdings kein Problem, da `qu->tail` nur verwendet wird, falls `qu->head` ungleich null, die Liste also nicht leer ist, und da bei dem Einfügen des ersten Elements in Zeile 22 dafür gesorgt wird, dass auch `qu->tail` wieder auf ein korrektes Element zeigt.

Auch in diesem Fall stellen wir dem Anwender lediglich Funktionen `enqueue` und `dequeue` zur Verfügung, damit wir uns die Freiheit bewahren, die konkrete Implementierung der Warteschlange bei Bedarf anpassen oder erweitern zu können.

3.4 Bäume

Listen eignen sich gut, um während der Laufzeit des Programms wachsende und schrumpfende Datenmengen handzuhaben, allerdings fehlt uns bei ihnen die Möglichkeit, effizient nach bestimmten Elementen zu suchen.

Für Arrays haben wir mit der binären Suche in Abschnitt 2.2 einen sehr effizienten Suchalgorithmus kennen gelernt, den wir nun auf den Fall dynamische Datenmengen zu übertragen versuchen werden. Die binäre Suche beruht auf der Idee, durch einen Vergleich mit einem Element in der Mitte des sortierten Arrays entscheiden zu können, in welcher Hälfte des Arrays die Suche fortgesetzt werden soll.

Das resultierende Zugriffsmuster ist in Abbildung 3.10 dargestellt: Der Algorithmus beginnt bei dem mittleren Element, auf das der schwarze Pfeil zeigt. Abhängig davon, ob $x_k \leq y$ gilt oder nicht wird dann entweder dem linken oder rechten Pfeil gefolgt, bis das gesuchte Element gefunden ist oder nicht weiter gesucht werden kann. Auf der rechten Seite der Abbildung sind die Elemente so angeordnet, dass alle, die in der i-ten Iteration geprüft werden, in der i-ten Zeile stehen.

Wenn wir das Verhalten einer binären Suche in einer dynamischen Datenstruktur reproduzieren wollen, müssen wir also eine Möglichkeit schaffen, je nach Ergebnis der

```
1  typedef struct _queue queue;
2  struct _queue {
3    listelement *head;
4    listelement *tail;
5  };
6  queue *
7  new_queue()
8  {
9    queue *qu;
10   qu = (queue *) malloc(sizeof(queue));
11   qu->head = 0; qu->tail = 0;
12   return qu;
13 }
14 void
15 enqueue(queue *qu, payload x)
16 {
17   if(qu->head) {
18     qu->tail->next = new_listelement(x, 0);
19     qu->tail = qu->tail->next;
20   }
21   else
22     qu->head = qu->tail = new_listelement(x, 0);
23 }
24 payload
25 dequeue(queue *qu)
26 {
27   payload x;
28   if(qu->head == 0) crash("Queue empty");
29   x = qu->head->x;
30   qu->head = del_listelement(qu->head);
31   return x;
32 }
33 int
34 isempty(queue *qu)
35 {
36   return (qu->head == 0);
37 }
```

Abbildung 3.9: Warteschlange mit Funktionen für das Hinzufügen und Entnehmen von Elementen.

Prüfung des jeweils aktuellen Elements zu dem mittleren Element der linken oder rechten Hälfte der Datenmenge zu wechseln.

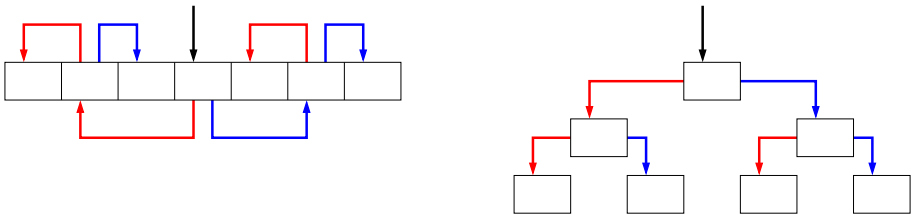


Abbildung 3.10: Zugriffsmuster der binären Suche auf Einträge des zu durchsuchenden Arrays. Rechts sind die Einträge nach der Iterationszahl eingeteilt, die erforderlich sind, um sie zu erreichen.

Anders als bei Listen, bei denen wir auf das erste oder letzte Element zugreifen können, müssten wir also eine Datenstruktur verwenden, bei der wir auf ein „mittleres“ Element zugreifen können. Außerdem müssen wir die Möglichkeit haben, nach einem Vergleich mit diesem mittleren Element ungefähr die Hälfte der zu durchsuchenden Datenmenge auszuschließen.

Diese Ziele können wir erreichen, wenn wir die bereits bekannte Listenstruktur so abändern, dass statt eines Zeigers auf das unmittelbar folgende Element zwei Zeiger auf die mittleren Elemente der „linken“ und „rechten“ Hälften der Datenmenge verwendet werden. Motiviert durch die auf der rechten Seite der Abbildung 3.10 dargestellte baumartig verzweigte Struktur bezeichnen wir diesen Datentyp als *Baum* (im Englischen *tree*):

```
1 typedef struct _tree tree;
2 struct _tree {
3   payload x;
4   tree *left;
5   tree *right;
6 };
```

Das Feld `x` enthält wie bisher die eigentlichen Daten, während die Zeiger `left` und `right` auf die „linke“ und „rechte Hälfte“ der gesamten Datenmenge zeigen. Der gesamte Baum wird beschrieben durch einen Zeiger auf das oberste Element, das wir als die *Wurzel* des Baums bezeichnen.

Suchbäume. Die entscheidende Idee der binären Suche besteht darin, dass man in einem *sortierten* Array sucht, so dass sich jeweils mit einem Vergleich eine Hälfte des Arrays von der weiteren Suche ausschließen lässt. Um dieses Verhalten für unsere neue `tree`-Struktur zu reproduzieren müssen wir fordern, dass alle Elemente, zu denen wir auf dem Weg über den `left`-Zeiger gelangen können, nicht echt größer als das aktuelle Element sind, während alle, die wir über den `right`-Zeiger erreichen können, nicht echt kleiner sind. Einen derartigen Baum bezeichnet man als *Suchbaum* für die gegebenen Elemente und die gegebene Ordnung. Die Suche nach einem Element `y` in einem Suchbaum

```
1  tree *
2  new_tree(payload x, tree *left, tree *right)
3  {
4    tree *t;
5    t = (tree *) malloc(sizeof(tree));
6    t->x = x; t->left = left; t->right = right;
7    return t;
8  }
9  tree *
10 addto_tree(tree *t, payload x)
11 {
12   if(t) {
13     if(compare(x, t->x) <= 0)
14       t->left = addto_tree(t->left, x);
15     else
16       t->right = addto_tree(t->right, x);
17   }
18   else
19     t = new_tree(x, 0, 0);
20   return t;
21 }
22 tree *
23 findin_tree(tree *t, payload y)
24 {
25   int cmp;
26   if(t) {
27     cmp = compare(y, t->x);
28     if(cmp == 0) return t;
29     else if(cmp < 0)
30       return findin_tree(t->left, y);
31     else
32       return findin_tree(t->right, y);
33   }
34   return 0;
35 }
```

Abbildung 3.11: Implementierung eines Suchbaums

mit Wurzel `t` gestaltet sich analog zur binären Suche in einem Array: Wir vergleichen das zu suchende Element `y` mit dem Element `t->x`. Falls beide gleich sind, sind wir fertig. Falls `y` kleiner als `t->x` ist, wiederholen wir die Prozedur für den Teilbaum mit der Wurzel `t->left`, ansonsten für den mit der Wurzel `t->right`.

Da wir eine dynamische Datenstruktur entwerfen, die in der Regel Element für Element

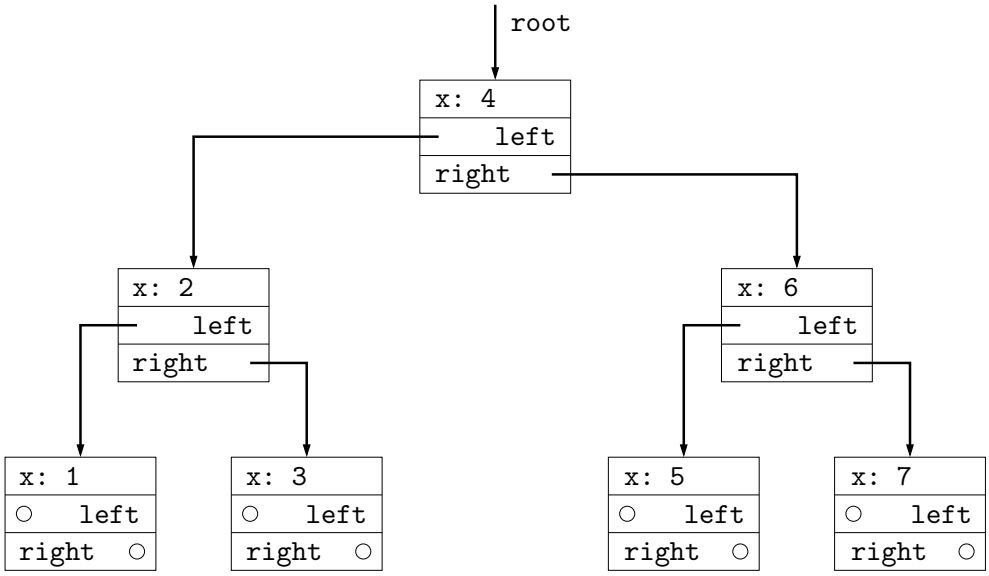


Abbildung 3.12: Aufbau eines binären Baums. Nullzeiger sind durch Kreise dargestellt.

aufgebaut wird, können wir bereits bei der Konstruktion der Struktur dafür sorgen, dass ein Suchbaum entsteht, indem wir die neuen Elemente an der richtigen Stelle einsortieren: Wenn ein Element `y` dem Baum hinzugefügt werden soll, auf den ein Zeiger `t` verweist, prüfen wir mit der Funktion `compare`, ob es echt größer als `t->x` ist. In diesem Fall gehen wir rekursiv vor und fügen es in den Teilbaum ein, auf den `t->right` verweist. Anderenfalls gehört das neue Element in den Teilbaum, auf den `t->left` zeigt. Die Rekursion endet, sobald wir den Nullzeiger erreichen, den wir dann durch einen Zeiger auf einen neuen Baum ersetzen, der nur das neue Element `x` enthält.

Für Elemente $1, 2, \ldots, 7$ mit der üblichen Ordnung ist ein möglicher Suchbaum in Abbildung 3.12 dargestellt. Er entsteht, indem man zunächst 4, dann 2 und 6, und schließlich die restlichen Elemente einfügt. Der Zeiger `root` ermöglicht es uns, auf die Wurzel der gesamten Baums zuzugreifen, er übernimmt also die Rolle, die der Zeiger `head` bei einer einfachen Liste spielt. Ein Nachteil der hier untersuchten einfachen Suchbäume besteht darin, dass die Reihenfolge der Einfügeoperationen eine große Rolle spielt: In dem hier dargestellten Fall sind nicht mehr als drei Vergleichsoperationen erforderlich, um ein Element zu finden oder festzustellen, dass es nicht in dem Baum enthalten ist. Wenn wir die Elemente dagegen in der Reihenfolge $1, 2, \ldots, 7$ einfügen, erhalten wir eine einfach verkettete Liste, bei der die `left`-Zeiger immer gleich null sind, denn jedes neue Element muss „rechts“ von allen vorangehenden eingefügt werden. Der Suchalgorithmus benötigt dann beispielsweise sieben Vergleiche, um das Element 7 zu finden.

Man kann zeigen, dass in einem optimaler Suchbaum mit n Elementen nicht mehr als $\lceil \log_2(n) \rceil$ Vergleiche benötigt werden, um ein Element zu finden oder festzustellen, dass es nicht im Baum liegt. Es gibt auch Beispiele (siehe Abbildung 3.14) für Suchbäume,

in denen n Vergleiche nötig werden. Allerdings können wir durch geeignete Modifikationen unserer Datenstruktur und unserer Algorithmen diese besonders ungünstigen Fälle ausschließen.

Um die relevanten Eigenschaften eines Baums präzise formulieren zu können, empfiehlt es sich, ihn mathematisch zu definieren. Dabei können wir uns an der Datenstruktur `tree` orientieren: Entsprechend dem Verbundtyp mit drei Feldern definieren wir den Baum als ein Tripel aus dem eigentlichen Datenelement und zwei weiteren Bäumen:

Definition 3.2 (Binärer Baum) *Sei X eine Menge. Wir definieren die Menge $\mathcal{T}_X$ als die kleinste Menge (bezüglich der Mengeninklusion), die die folgenden Bedingungen erfüllt:*

$$\emptyset \in \mathcal{T}_X, \tag{3.1a}$$

$$(t_1, t_2, x) \in \mathcal{T}_X \qquad \text{für alle } x \in X,\ t_1, t_2 \in \mathcal{T}_X. \tag{3.1b}$$

Die Elemente der Menge $\mathcal{T}_X$ nennen wir beschriftete binäre Bäume. *Den Baum $\emptyset$ nennen wir den* leeren Baum. *Für $t \in \mathcal{T}_X$ mit $t = (t_1, t_2, x)$ nennen wir x die* Beschriftung, *bezeichnet mit $\hat{t}$, und t_1 und t_2 den* linken *und* rechten Teilbaum.

Bemerkung 3.3 (Kleinste Menge) *Die Formulierung „die kleinste Menge (bezüglich der Mengeninklusion)“ in Definition 3.2 ist wie folgt zu verstehen: Falls zwei Mengen $\mathcal{T}_X^1$ und $\mathcal{T}_X^2$ gegeben sind, die beide die Bedingungen (3.1) erfüllen, lässt sich leicht nachprüfen, dass ihre Schnittmenge $\mathcal{T}_X^1 \cap \mathcal{T}_X^2$ diese Bedingungen ebenfalls erfüllt.*

Diese Eigenschaft gilt nicht nur für den Schnitt zweier Mengen, sondern auch für die Schnittmenge aller *Mengen, die die Bedingungen (3.1) erfüllen. Diese Schnittmenge ist offenbar eine Teilmenge jeder Menge, die (3.1) erfüllt, also bezüglich der Inklusionsrelation $\subseteq$ kleiner. Sie ist gemeint, wenn wir $\mathcal{T}_X$ schreiben.*

Dieses Konstruktionsprinzip lässt sich mit dem Lemma von Kuratowski-Zorn *auch für allgemeinere partielle Ordnungen $\trianglelefteq$ anstelle der Mengeninklusion $\subseteq$ verwenden, um maximale oder minimale Elemente zu konstruieren.*

Bäume sind ein ausgesprochen wichtiges Hilfsmittel bei der Konstruktion von Algorithmen: Wir können mit ihnen nicht nur, wie im vorliegenden Fall, Daten organisieren, wir können mit ihrer Hilfe auch die Abfolge von Funktionsaufrufen analysieren, die Übersetzung von Programmtexten beschreiben oder Abhängigkeiten zwischen einzelnen Rechenoperationen modellieren.

Um möglichst anschaulich mit ihnen umgehen zu können, verwenden wir die in Abbildung 3.13 illustrierte Darstellung: Für einen Baum $t = (t_1, t_2, x) \in \mathcal{T}_X$ stellen wir das Element x dar, setzen die Darstellungen der beiden Teilbäume t_1 und t_2 links und rechts darunter und verbinden sie mit x durch Linien (oder in diesem Fall Linienzüge).

Die Definition 3.2 spiegelt zwar unsere Datenstruktur gut wider, ist aber ansonsten etwas unhandlich. Deshalb führen wir die folgende alternative Charakterisierung ein:

Lemma 3.4 (Baumhöhe) *Sei X eine beliebige Menge. Wir definieren*

$$\mathcal{T}_{X,0} := \{\emptyset\},$$

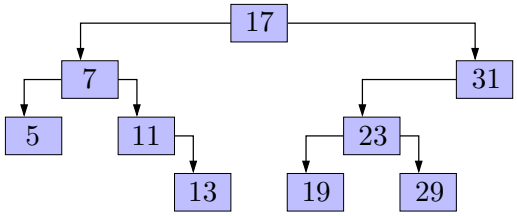


Abbildung 3.13: Darstellung eines binären Baums.

$$\begin{aligned}\mathcal{T}_{X,h} := \{(t_1,t_2,x) \;:\; x \in X,\ (t_1 \in \mathcal{T}_{X,h-1} \wedge t_2 \in \mathcal{T}_{X,k}) \text{ für ein } k \in \{0,\dots,h-1\}\}\\ \cup \{(t_1,t_2,x) \;:\; x \in X,\ (t_2 \in \mathcal{T}_{X,h-1} \wedge t_1 \in \mathcal{T}_{X,k}) \text{ für ein } k \in \{0,\dots,h-1\}\}\\ \text{für alle } h \in \mathbb{N}_0.\end{aligned}$$

Dann sind die Mengen $(\mathcal{T}_{X,h})_{h=0}^{\infty}$ *paarweise disjunkt und erfüllen*

$$\mathcal{T}_X = \bigcup_{h=0}^{\infty} \mathcal{T}_{X,h}. \tag{3.2}$$

Also existiert zu jedem Baum $t \in \mathcal{T}_X$ *genau ein* $h \in \mathbb{N}_0$ *mit* $t \in \mathcal{T}_{X,h}$*. Dieses* h *nennen wir die* Höhe *des Baums und schreiben es als* $\operatorname{height}(t)$*.*

Beweis. Wir setzen

$$\widetilde{\mathcal{T}}_X := \bigcup_{h=0}^{\infty} \mathcal{T}_{X,h}$$

und müssen $\mathcal{T}_X = \widetilde{\mathcal{T}}_X$ beweisen.

Dazu zeigen wir zunächst, dass $\widetilde{\mathcal{T}}_X \subseteq \mathcal{T}_X$ gilt. Diesen Beweis führen wir, indem wir per Induktion nachweisen, dass für jedes $\ell \in \mathbb{N}_0$

$$\mathcal{T}_{X,h} \subseteq \mathcal{T}_X \qquad \text{für alle } h \in \{0,\dots,\ell\} \tag{3.3}$$

gilt, denn daraus folgt, dass auch die Vereinigung aller $\mathcal{T}_{X,h}$ in $\mathcal{T}_X$ enthalten ist.

Induktionsanfang. Sei $\ell = 0$. Nach (3.1a) gilt $\mathcal{T}_{X,0} = \{\emptyset\} \subseteq \mathcal{T}_X$.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ derart gewählt, dass (3.3) gilt.

Induktionsschritt. Wir weisen nach, dass auch für $h := \ell + 1$ die Inklusion $\mathcal{T}_{X,h} \subseteq \mathcal{T}_X$ gilt. Sei also ein $t \in \mathcal{T}_{X,h}$ gegeben. Wegen $h = \ell + 1 > 0$ und der Definition der Menge $\mathcal{T}_{X,h}$ gilt $t = (t_1,t_2,x)$ mit $x \in X$ und $t_1 \in \mathcal{T}_{X,h-1}$ und $t_2 \in \mathcal{T}_{X,k}$ oder $t_1 \in \mathcal{T}_{X,k}$ und $t_2 \in \mathcal{T}_{X,h-1}$ für ein $k \in \{0,\dots,h-1\}$. Nach Induktionsvoraussetzung folgt wegen $k \leq h-1 = \ell$ bereits $t_1, t_2 \in \mathcal{T}_X$, und mit (3.1b) erhalten wir $t \in \mathcal{T}_X$.

Damit ist die Induktion vollständig und $\widetilde{\mathcal{T}}_X \subseteq \mathcal{T}_X$ bewiesen.

Um nachzuweisen, dass beide Mengen sogar gleich sind, können wir ausnutzen, dass $\mathcal{T}_X$ die *kleinste* Menge ist, die die Bedingungen (3.1) erfüllt: Wenn die Menge $\widetilde{\mathcal{T}}_X$ die Bedingungen (3.1) erfüllt, muss sie auch eine Obermenge dieser kleinsten Menge $\mathcal{T}_X$

sein, denn sonst ließe sich durch Schneiden beider Mengen eine noch kleinere Menge konstruieren.

Wir weisen also nach, dass $\widetilde{\mathcal{T}}$ die Bedingungen (3.1) erfüllt. Wegen $\emptyset \in \mathcal{T}_{X,0} \subseteq \widetilde{\mathcal{T}}_X$ gilt die Bedingung (3.1a).

Zum Nachweis der Bedingung (3.1b) seien nun $t_1, t_2 \in \widetilde{\mathcal{T}}_X$ und $x \in X$ fixiert. Dann existieren $h_1, h_2 \in \mathbb{N}_0$ mit $t_1 \in \mathcal{T}_{X,h_1}$ und $t_2 \in \mathcal{T}_{X,h_2}$. Ohne Beschränkung der Allgemeinheit sei $h_1 \geq h_2$. Wir setzen $h := h_1 + 1$ und stellen fest, dass damit $t_1 \in \mathcal{T}_{X,h-1} = \mathcal{T}_{X,h_1}$ und $t_2 \in \mathcal{T}_{X,h_2}$ mit $h_2 \leq h_1 = h - 1$ gilt. Nach Definition folgt $t := (t_1, t_2, x) \in \mathcal{T}_{X,h}$, also auch $t \in \widetilde{\mathcal{T}}_X$. Somit erfüllt die Menge $\widetilde{\mathcal{T}}_X$ auch (3.1b).

Da $\mathcal{T}_X$ als die *kleinste* Menge definiert ist, die (3.1) erfüllt, folgt $\mathcal{T}_X \subseteq \widetilde{\mathcal{T}}_X$, also insgesamt $\mathcal{T}_X = \widetilde{\mathcal{T}}_X$.

Es bleibt noch zu zeigen, dass die Mengen $\mathcal{T}_{X,h}$ paarweise disjunkt sind. Dazu beweisen wir per Induktion über alle $h \in \mathbb{N}$ die Gleichung

$$\mathcal{T}_{X,k} \cap \mathcal{T}_{X,h} = \emptyset \qquad \text{für alle } k \in \{0, \ldots, h-1\}. \tag{3.4}$$

Induktionsanfang. Sei $h = 1$. Da $\mathcal{T}_{X,0}$ lediglich die leere Menge $\emptyset$ enthält, die offenbar kein Tripel ist, folgt (3.4) direkt.

Induktionsvoraussetzung. Sei $h \in \mathbb{N}$ so gegeben, dass (3.4) gilt.

Induktionsschritt. Sei $t \in \mathcal{T}_{X,h+1}$ gegeben. Nach Definition gilt dann $t = (t_1, t_2, x)$ mit $x \in X$ und $t_1 \in \mathcal{T}_{X,h}$ oder $t_2 \in \mathcal{T}_{X,h}$.

Sei $k \in \{0, \ldots, h\}$ und $s \in \mathcal{T}_{X,k}$ gegeben. Für $k = 0$ folgt unmittelbar $s = \emptyset \neq t$. Für $k > 0$ dagegen erhalten wir $s = (s_1, s_2, y)$ mit $y \in X$ und $s_1 \in \mathcal{T}_{X,k_1}$ sowie $s_2 \in \mathcal{T}_{X,k_2}$ für $k_1, k_2 \in \{0, \ldots, h-1\}$.

Falls nun $t_1 \in \mathcal{T}_{X,h}$ gilt, folgt aus $k_1 < h$ mit der Induktionsvoraussetzung bereits $t_1 \neq s_1$. Anderenfalls gilt $t_2 \in \mathcal{T}_{X,h}$ und wir erhalten mit $k_2 < h$ und der Induktionsvoraussetzung $t_2 \neq s_2$. In beiden Fällen ist damit $t \neq s$ bewiesen. ∎

Wenn wir mathematisch mit Bäumen arbeiten wollen, wäre es nützlich, wenn uns induktive Definitionen und Induktionsbeweise zur Verfügung stehen würden. Die Baumhöhe bietet uns diese Möglichkeit. Wenn wir beispielsweise eine Funktion $f : \mathcal{T}_X \to \mathbb{R}$ definieren wollen, genügt es, ihren Wert $f(\emptyset)$ für den leeren Baum festzulegen und eine Formel des Typs

$$f(t) = g(f(t_1), f(t_2), x) \qquad \text{für alle } t = (t_1, t_2, x) \in \mathcal{T}_X \tag{3.5}$$

vorauszusetzen, wobei $g : \mathbb{R} \times \mathbb{R} \times X \to \mathbb{R}$ eine geeignete Funktion bezeichnet.

Die induktive Definition erfolgt dann analog zum Fall natürlicher Zahlen: Mit $f(\emptyset)$ ist f auf der Menge $\mathcal{T}_{X,0}$ definiert. Falls ein $\ell \in \mathbb{N}_0$ so gegeben ist, dass f auf $\mathcal{T}_{X,h}$ für alle $h \leq \ell$ definiert ist, können wir die Funktion mit (3.5) auch auf der Menge $\mathcal{T}_{X,\ell+1}$ definieren. Also ist sie für Bäume beliebiger Höhe definiert und somit für alle.

Beispielsweise können wir die Anzahl der Elemente eines Baums, also der nicht-leeren Teilbäume, wie folgt induktiv definieren: Es sei

$$\#t := \begin{cases} 1 + \#t_1 + \#t_2 & \text{falls } t = (t_1, t_2, x) \text{ mit } t_1, t_2 \in \mathcal{T}_X,\ x \in X, \\ 0 & \text{ansonsten, also falls } t = \emptyset \end{cases}$$

für alle $t \in \mathcal{T}_X$. Diese Definition spiegelt unmittelbar die Konstruktion unserer Datenstruktur `tree` wider: Falls ein Baum nicht leer ist, enthält er das Wurzelelement und alle Elemente in den beiden Teilbäumen t_1 und t_2.

Um Induktionsbeweise über Bäume führen zu können, müssen wir das Prinzip der Induktion über den natürlichen Zahlen verallgemeinern. Auch in diesem Fall hilft uns der Begriff der Höhe von Bäumen:

Lemma 3.5 (Strukturelle Induktion) *Sei $A : \mathcal{T}_X \to \{\mathit{wahr}, \mathit{falsch}\}$ eine Aussage auf Bäumen. Falls*

$$A(\emptyset) \text{ und} \tag{3.6a}$$

$$A(t_1) \wedge A(t_2) \Rightarrow A(t) \qquad \text{für alle } t = (t_1, t_2, x) \text{ mit } t_1, t_2 \in \mathcal{T}_X,\ x \in X \tag{3.6b}$$

gelten, gilt $A(t)$ für alle Bäume $t \in \mathcal{T}_X$.

Beweis. Wir führen den Beweis per Induktion über die Höhe der Bäume, indem wir für alle $\ell \in \mathbb{N}_0$ beweisen, dass $A(t)$ für alle $t \in \mathcal{T}_{X,h}$ mit $h \leq \ell$ gilt.

Induktionsanfang. Sei $\ell = 0$, also auch $h = 0$. Nach (3.6a) gilt $A(\emptyset)$, also wegen $\mathcal{T}_{X,0} = \{\emptyset\}$ auch die Behauptung.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gewählt, dass die Aussage $A(t)$ für alle $t \in \mathcal{T}_{X,h}$ mit $h \leq \ell$ gilt.

Induktionsschritt. Sei $t \in \mathcal{T}_{X,\ell+1}$. Nach Definition der Menge $\mathcal{T}_{X,\ell+1}$ gilt dann $t = (t_1, t_2, x)$ mit $t_1 \in \mathcal{T}_{x,h_1}$ und $t_2 \in \mathcal{T}_{x,h_2}$ für $h_1, h_2 < \ell + 1$, also $h_1, h_2 \leq \ell$. Mit der Induktionsvoraussetzung folgt, dass $A(t_1)$ und $A(t_2)$ gelten, so dass sich mit (3.6b) bereits $A(t)$ ergibt. ∎

Als Beispiel für einen strukturellen Induktionsbeweis leiten wir eine alternative Definition der Baumhöhe her, mit deren Hilfe wir die Baumhöhe praktisch berechnen können.

Lemma 3.6 (Baumhöhe als Maximum) *Für alle Bäume $t \in \mathcal{T}_X$ gilt*

$$\operatorname{height}(t) = \begin{cases} 1 + \max\{\operatorname{height}(t_1), \operatorname{height}(t_2)\} & \mathit{falls}\ t \neq \emptyset \\ 0 & \mathit{ansonsten.} \end{cases}$$

Beweis. Wir definieren die Funktion $m : \mathcal{T}_X \to \mathbb{N}_0$ induktiv durch

$$m(t) := \begin{cases} 1 + \max\{m(t_1), m(t_2)\} & \text{falls } t \neq \emptyset, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } t \in \mathcal{T}_X.$$

Wir werden per struktureller Induktion $\operatorname{height}(t) = m(t)$ für alle Bäume $t \in \mathcal{T}_X$ beweisen.

Induktionsanfang. Sei $t = \emptyset$. Dann gilt $t \in \mathcal{T}_{X,0}$, also $\operatorname{height}(t) = 0 = m(t)$.

Induktionsvoraussetzung. Seien $t_1, t_2 \in \mathcal{T}_X$ so gegeben, dass $\operatorname{height}(t_1) = m(t_1)$ und $\operatorname{height}(t_2) = m(t_2)$ gelten.

Induktionsschritt. Sei $t = (t_1, t_2, x)$ für ein $x \in X$. Zur Abkürzung setzen wir $h := \max\{\operatorname{height}(t_1), \operatorname{height}(t_2)\} + 1$ und stellen fest, dass nach Definition dann $t \in \mathcal{T}_{X,h}$ gelten

muss, also $\text{height}(t) = h$. Nach Induktionsvoraussetzung haben wir $\text{height}(t_1) = m(t_1)$ sowie $\text{height}(t_2) = m(t_2)$, also

$$\text{height}(t) = h = \max\{\text{height}(t_1), \text{height}(t_2)\} + 1 = \max\{m(t_1), m(t_2)\} + 1 = m(t).$$

■

Während wir bei der Induktion über natürliche Zahlen zeigen, dass aus der Gültigkeit der Behauptung für eine Zahl n auch die Gültigkeit für ihren Nachfolger $n + 1$ folgt, beweisen wir also bei der strukturellen Induktion, dass aus der Gültigkeit der Behauptung für zwei Bäume auch die Gültigkeit für den aus diesen Bäumen zusammengesetzten Baum folgt.

Der Ausgangspunkt unserer Definition eines Baums war die binäre Suche, die uns zu dem Begriff des Suchbaums führte, den wir nun mathematisch präzise definieren können. Damit wir mit einem Vergleich mit der Wurzel des Baums jeweils einen seiner beiden Teilbäume ausschließen können, müssen wir die Menge der in diesen Teilbäumen zu erwartenden Elemente kennen. Dazu definieren wir

$$\text{labels}(t) := \begin{cases} \{\hat{t}\} \cup \text{labels}(t_1) \cup \text{labels}(t_2) & \text{falls } t \neq \emptyset, \\ \emptyset & \text{ansonsten} \end{cases} \qquad \text{für alle } t \in \mathcal{T}_X.$$

Für die binäre Suche ist von entscheidender Bedeutung, dass die zu durchsuchenden Elemente total geordnet sind, wir gehen also im Folgenden davon aus, dass $\trianglelefteq$ eine totale Ordnung auf einer Menge M ist.

Die Idee des Suchbaums besteht darin, durch einen Vergleich mit dem Wurzelelement $\hat{t}$ des Baums entscheiden zu können, ob die Suche in dem linken Teilbaum t_1 oder in dem rechten t_2 fortgesetzt werden soll, wir brauchen also ein Gegenstück des Lemmas 2.30. Es bietet sich an, die relevante Eigenschaft direkt in der Definition eines Suchbaums zu verankern:

Definition 3.7 (Suchbaum) *Sei M eine Menge und $\trianglelefteq$ eine totale Ordnung auf M. Wie nennen einen Baum $t \in \mathcal{T}_M$* Suchbaum*, falls $t = \emptyset$ gilt oder für $t = (t_1, t_2, x)$ die Teilbäume t_1 und t_2 Suchbäume sind und die folgenden Bedingungen gelten:*

$$y \trianglelefteq x \qquad \text{für alle } y \in \text{labels}(t_1), \tag{3.7a}$$
$$x \triangleleft y \qquad \text{für alle } y \in \text{labels}(t_2). \tag{3.7b}$$

Zur Abkürzung führen wir die Notation ein, dass wir die Relationen $\trianglelefteq$ und $\triangleleft$ auch für Teilmengen $X, Y \subseteq M$ verwenden können:

$$X \trianglelefteq Y :\iff \forall x \in X,\ y \in Y : x \trianglelefteq y,$$
$$X \triangleleft Y :\iff \forall x \in X,\ y \in Y : x \triangleleft y.$$

Mit dieser Konvention können wir (3.7) kurz als

$$\text{labels}(t_1) \trianglelefteq \hat{t} \triangleleft \text{labels}(t_2)$$

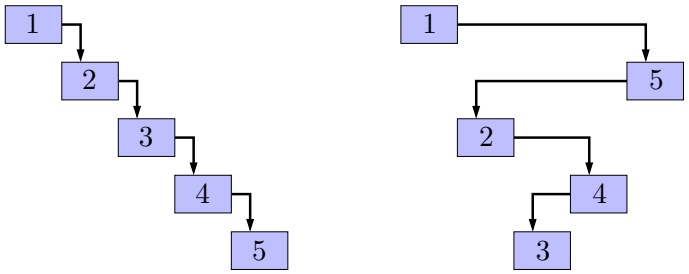


Abbildung 3.14: Ungünstige Suchbäume.

schreiben.

Aus der Definition 3.7 ergibt sich unmittelbar, dass die in Abbildung 3.11 dargestellten Algorithmen für das Einfügen eines Elements in einen Suchbaum und die Suche nach einem Element korrekt sind: Falls das einzufügende Element $y \in M$ die Bedingung $y \trianglelefteq \hat{t}$ erfüllt, muss es im linken Teilbaum untergebracht werden, sonst muss es der rechte sein. Falls das zu suchende Element die Bedingung $y \trianglelefteq \hat{t}$ erfüllt, muss es im linken Teilbaum gesucht werden, sonst kommt nur der rechte in Frage.

Ein Beispiel für einen Suchbaum für eine Teilmenge der natürlichen Zahlen findet sich in Abbildung 3.13.

Rechenaufwand. Der Rechenaufwand des Einfügens und Suchens (und auch vieler weiterer Operationen, die man auf Suchbäumen ausführen kann) hängt entscheidend von der Höhe des Baums ab, denn mit jedem rekursiven Schritt zu einem Teilbaum reduziert sich die Höhe um eins, so dass für einen Baum $t \in \mathcal{T}_M$ höchstens $\mathrm{height}(t)$ Schritte erforderlich werden.

Leider kann ein Baum für n Elemente eine Höhe von n erreichen, beispielsweise wenn die n Elemente in aufsteigend sortierter Reihenfolge eingefügt werden, so dass in jedem Schritt nur der rechte Teilbaum ergänzt wird und die listenartige Struktur entsteht, die auf der linken Seite der Abbildung 3.14 dargestellt ist. Die ähnlich ungünstige Struktur auf der rechten Seite der Abbildung ergibt sich, wenn wir abwechselnd das kleinste und größte verbliebene Element einsortieren. Die Schwierigkeiten ähneln denen, die wir bereits bei der Analyse des Quicksort-Algorithmus' beobachtet hatten, bei dem ebenfalls eine ungünstige Auswahl des Pivot-Elements zu sehr schlechten Ergebnissen führt.

Deshalb werden wir, wie schon in jenem Fall, nun untersuchen, welchen Aufwand wir *im statistischen Mittel* zu erwarten haben. Wir betrachten eine aufsteigend sortierte Folge

$$x_1 \lhd x_2 \lhd \ldots \lhd x_n$$

von paarweise verschiedenen Elementen, die wir in zufälliger Reihenfolge in den leeren Baum einfügen, um schließlich einen Suchbaum t zu erhalten. Dabei sollen in jedem Schritt alle Elemente mit derselben Wahrscheinlichkeit ausgewählt werden.

Wir wollen wissen, wieviele Vergleichsoperationen unser Suchalgorithmus im Mittel

benötigt, um ein Element x_i zu finden. Dazu bezeichnen wir mit $S(n)$ die Anzahl der Vergleichsoperationen, die nötig sind, um *jedes* Element einmal zu finden.

Bei n Elementen muss jeweils einmal mit dem Wurzelelement verglichen werden, so dass n Vergleiche anfallen. Anschließend wissen wir, ob wir die Suche im linken oder rechten Teilbaum fortsetzen müssen. Falls das m-te Element x_m in der Wurzel steht, enthält der linke Teilbaum $m-1$ Elemente, während der rechte $n-m$ Elemente enthält. Also fallen zusätzlich $S(m-1)+S(n-m)$ Vergleichsoperationen an, um in den beiden Teilbäumen zu suchen.

Da alle Elemente mit gleicher Wahrscheinlichkeit $1/n$ als Wurzel gewählt wurden, erhalten wir für den Erwartungswert die Rekurrenzformel

$$S(n) = \begin{cases} 0 & \text{falls } n = 0, \\ 1 & \text{falls } n = 1, \\ n + \frac{1}{n}\sum_{m=1}^{n}(S(m-1)+S(n-m)) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N}_0.$$

Um Lemma 2.17 anwenden zu können, halten wir fest, dass

$$\begin{aligned}\sum_{m=1}^{n}(S(m-1)+S(n-m)) &= \sum_{m=1}^{n} S(m-1) + \sum_{m=1}^{n} S(n-m) \\ &= \sum_{m=0}^{n-1} S(m) + \sum_{m=0}^{n-1} S(m) = 2\sum_{m=0}^{n-1} S(m) = 2\sum_{m=1}^{n-1} S(m)\end{aligned}$$

gilt, wobei wir $S(0) = 0$ ausnutzen. Damit lässt sich die Rekurrenzformel in der Form

$$S(n) = \begin{cases} 1 & \text{falls } n = 1, \\ n + \frac{2}{n}\sum_{m=1}^{n-1} S(m) & \text{ansonsten} \end{cases} \qquad \text{für alle } n \in \mathbb{N}$$

schreiben. Indem wir Lemma 2.17 auf $\alpha = 1$, $\beta = 0$ und $\gamma = 1$ anwenden, erhalten wir

$$S(n) \leq n + 2n\ln(n) \qquad \text{für alle } n \in \mathbb{N}.$$

Bei einem durch zufälliges Einfügen von n Elementen entstandenen Suchbaum benötigen wir also im Mittel

$$\frac{S(n)}{n} \leq 2\ln(n) + 1$$

Vergleichsoperationen, um ein Element zu finden.

3.5 Balancierte Bäume

Selbstverständlich würden wir es vorziehen, wenn wir garantieren könnten, dass ungünstige Fälle wie die in Abbildung 3.14 dargestellten nicht auftreten können. Theoretisch ist es möglich, einen idealen Suchbaum für n Elemente zu konstruieren, dessen Höhe lediglich $\lfloor \log_2(n) \rfloor + 1$ beträgt: Wir sortieren die Elemente und definieren den Baum so, dass er die binäre Suche widerspiegelt, für die nach Lemma 2.3 $\lfloor \log_2(n) \rfloor + 1$ Iterationen

genügen, so dass ein Baum dieser Tiefe entsteht. Dieser Zugang ist allerdings relativ aufwendig.

Wesentlich effizienter ist es, *AVL-Bäume* zu verwenden, die von G. M. Adelson-Velski und J. M. Landis entwickelt wurden: Wir weisen jedem Baum die *Balance*

$$\operatorname{bal}(t) := \begin{cases} \operatorname{height}(t_2) - \operatorname{height}(t_1) & \text{falls } t \neq \emptyset, \\ 0 & \text{ansonsten} \end{cases} \qquad \text{für alle } t \in \mathcal{T}_M \tag{3.8}$$

zu, die angibt, wie ungünstig er ist. Beispielsweise beträgt die Balance der in Abbildung 3.14 dargestellten Bäume jeweils 4, da der rechte Teilbaum eine Höhe von 4 aufweist, während der linke Teilbaum leer ist, also lediglich die Höhe 0 besitzt. Dagegen beträgt die Balance des günstigeren in Abbildung 3.13 dargestellten Baums null, da beide Teilbäume die Höhe 3 aufweisen.

Leider können wir nicht in jedem Fall sicherstellen, dass ein Baum eine Balance von null erreicht, beispielsweise hat jeder Baum mit zwei Elementen entweder die Balance 1 oder -1, abhängig davon, ob das zweite Element im rechten oder linken Teilbaum steht. Glücklicherweise ist es allerdings möglich, dafür zu sorgen, dass die Balance immer zwischen diesen Extremen liegt.

Definition 3.8 (AVL-Baum) *Sei M eine Menge. Wir nennen einen Baum $t \in \mathcal{T}_M$* AVL-Baum*, falls die beiden folgenden Bedingungen gelten:*

- *Die Balance erfüllt* $\operatorname{bal}(t) \in \{-1, 0, +1\}$ *und*
- *falls $t \neq \emptyset$ gilt, sind t_1 und t_2 AVL-Bäume.*

AVL-Bäume erlauben es uns, eine logarithmische Beziehung zwischen der Anzahl ihrer Elemente und ihrer Tiefe herzustellen, so dass beispielsweise für Suchoperationen lediglich logarithmischer Aufwand anfällt.

Satz 3.9 (AVL-Baum) *Wir bezeichnen mit $\varphi := (1 + \sqrt{5})/2$ den* goldenen Schnitt. *Für alle $t \in \mathcal{T}_M$ gilt*

$$t \text{ ist ein AVL-Baum} \Rightarrow \varphi^{\operatorname{height}(t)} - 1 \leq \#t \leq 2^{\operatorname{height}(t)} - 1 \tag{3.9}$$

Beweis. Wir halten zunächst fest, dass der goldene Schnitt φ die Gleichung

$$\varphi^2 = \frac{(1+\sqrt{5})^2}{4} = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{2 + 2\sqrt{5}}{4} + \frac{4}{4} = \frac{1+\sqrt{5}}{2} + 1 = \varphi + 1$$

erfüllt. Nach dieser Vorbetrachtung führen wir nun den eigentlichen Beweis mittels struktureller Induktion.

Induktionsanfang. Sei $t = \emptyset$. Dann gilt

$$\varphi^{\operatorname{height}(t)} - 1 = \varphi^0 - 1 = 0 = \#t = 0 = 2^0 - 1 = 2^{\operatorname{height}(t)} - 1.$$

Induktionsvoraussetzung. Seien $t_1, t_2 \in \mathcal{T}_M$ Bäume, für die (3.9) gilt.

Induktionsschritt. Sei $t = (t_1, t_2, x)$ für ein $x \in X$. Sei t ein AVL-Baum. Dann sind nach Definition 3.8 auch t_1 und t_2 AVL-Bäume. Wir setzen $h := \text{height}(t)$, $h_1 := \text{height}(t_1)$ und $h_2 := \text{height}(t_2)$.

Falls $\text{bal}(t) = -1$ gilt, folgen aus (3.8) die Gleichungen $h_2 - h_1 = -1$, also $h_1 = h_2 + 1$, und mit Lemma 3.6 $h = h_1 + 1 = h_2 + 2$. Mit der Induktionsvoraussetzung folgen

$$\begin{aligned}
\#t &= \#t_1 + \#t_2 + 1 \geq \varphi^{h_1} - 1 + \varphi^{h_2} - 1 + 1 = \varphi^{h_2+1} + \varphi^{h_2} - 1 \\
&= \varphi^{h_2}(\varphi + 1) - 1 = \varphi^{h_2}\varphi^2 - 1 = \varphi^{h_2+2} - 1 = \varphi^h - 1, \\
\#t &= \#t_1 + \#t_2 + 1 \leq 2^{h_1} - 1 + 2^{h_2} - 1 + 1 = 2^{h_2+1} + 2^{h_2} - 1 \\
&= 2^{h_2}(2 + 1) - 1 < 2^{h_2}2^2 - 1 = 2^{h_2+2} - 1 = 2^h - 1.
\end{aligned}$$

Falls $\text{bal}(t) = 0$ gilt, erhalten wir entsprechend $h_1 = h_2$ und $h = h_2 + 1$ sowie

$$\begin{aligned}
\#t &= \#t_1 + \#t_2 + 1 \geq \varphi^{h_1} - 1 + \varphi^{h_2} - 1 + 1 = \varphi^{h_2} + \varphi^{h_2} - 1 \\
&= \varphi^{h_2}2 - 1 \geq \varphi^{h_2}\varphi - 1 = \varphi^{h_2+1} - 1 = \varphi^h - 1, \\
\#t &= \#t_1 + \#t_2 + 1 \leq 2^{h_1} - 1 + 2^{h_2} - 1 + 1 = 2^{h_2} + 2^{h_2} - 1 \\
&= 2^{h_2}2 - 1 = 2^{h_2+1} - 1 = 2^h - 1.
\end{aligned}$$

Für $\text{bal}(t) = 1$ schließlich gelten $h_2 = h_1 + 1$ und $h = h_1 + 2$ sowie

$$\begin{aligned}
\#t &= \#t_1 + \#t_1 + 1 \geq \varphi^{h_1} - 1 + \varphi^{h_2} - 1 + 1 = \varphi^{h_1} + \varphi^{h_1+1} - 1 \\
&= \varphi^{h_1}(1 + \varphi) - 1 = \varphi^{h_1}\varphi^2 - 1 = \varphi^{h_1+2} - 1 = \varphi^h - 1, \\
\#t &= \#t_1 + \#t_2 + 1 \leq 2^{h_1} - 1 + 2^{h_2} - 1 + 1 = 2^{h_1} + 2^{h_1+1} - 1 \\
&= 2^{h_1}(1 + 2) - 1 < 2^{h_1}2^2 - 1 = 2^{h_1+2} - 1 = 2^h - 1.
\end{aligned}$$

Da t ein AVL-Baum ist, kann die Balance keine anderen Werte annehmen. ∎

Um eine Aussage über die Höhe des Baums zu erhalten, benötigen wir den Logarithmus zur Basis φ.

Erinnerung 3.10 (Logarithmus zu allgemeinen Basen) *Sei $\varphi \in \mathbb{R}_{>1}$.*

Für jede Zahl $x \in \mathbb{R}_{>0}$ existiert genau eine Zahl $y \in \mathbb{R}$ derart, dass $x = \varphi^y$ gilt. Diese Zahl nennen wir den Logarithmus zur Basis φ *von x und bezeichnen sie mit $\log_\varphi(x)$.*

Wir können eine Beziehung zu dem dyadischen Logarithmus (vgl. Erinnerung 2.4) herstellen, indem wir $2 = \varphi^{\log_\varphi(2)}$ ausnutzen: Sei $x \in \mathbb{R}_{>0}$ und $z := \log_2(x)$. Dann gilt

$$x = 2^z = (\varphi^{\log_\varphi(2)})^z = \varphi^{\log_\varphi(2)z},$$

also $\log_\varphi(x) = \log_\varphi(2)\log_2(x)$.

Damit ist auch der Logarithmus zur Basis φ eine streng monoton wachsende Funktion.

Folgerung 3.11 (Höhe eines AVL-Baums) *Sei $t \in \mathcal{T}_M$ ein AVL-Baum. Dann gilt*

$$\log_2(\#t + 1) \leq \text{height}(t) \leq \log_\varphi(\#t + 1) = \log_\varphi(2)\log_2(\#t + 1),$$

insbesondere liegt die Höhe eines AVL-Baums mit n Elementen in $\Theta(\log_2(n))$.

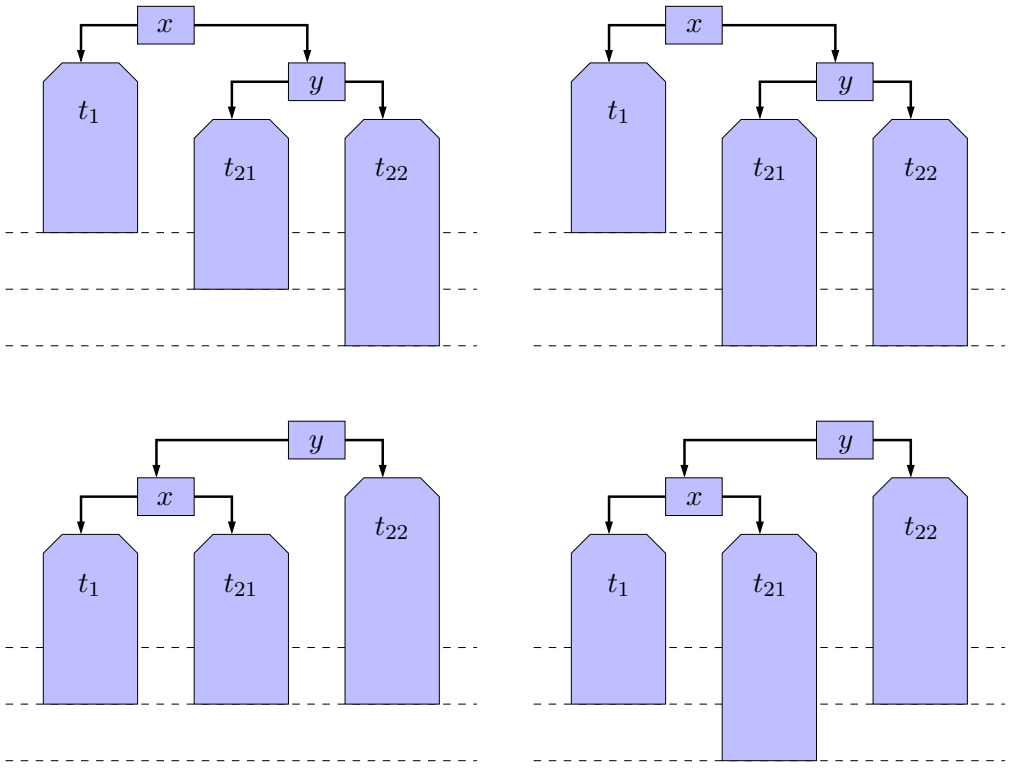


Abbildung 3.15: Links-Rotation in einem AVL-Baum t: Der jeweils unten dargestellte rotierte Baum $t' = ((t_1, t_{21}, x), t_{22}, y)$ ist besser balanciert als t, falls $\mathrm{bal}(t_2) = 1$ (links) oder $\mathrm{bal}(t_2) = 0$ (rechts) gilt.

Beweis. Sei $h := \mathrm{height}(t)$. Nach (3.9) gelten

$$\#t \leq 2^h - 1, \qquad \varphi^h - 1 \leq \#t,$$

also auch

$$\#t + 1 \leq 2^h, \qquad \varphi^h \leq \#t + 1,$$

und indem wir den dyadischen Logarithmus der linken Ungleichung den Logarithmus zur Basis φ von der rechten Ungleichung nehmen, folgt

$$\log_2(\#t + 1) \leq h, \qquad h \leq \log_\varphi(\#t + 1).$$

Mit Hilfe von Erinnerung 3.10 erhalten wir mit $\log_\varphi(\#t + 1) = \log_\varphi(2) \log_2(\#t + 1)$ die gewünschte Ungleichung. ∎

Da AVL-Bäume vorteilhafte Eigenschaften besitzen, wäre es also sinnvoll, wenn wir unsere Suchbäume so konstruieren könnten, dass sie AVL-Bäume werden. Diese Aufgabe

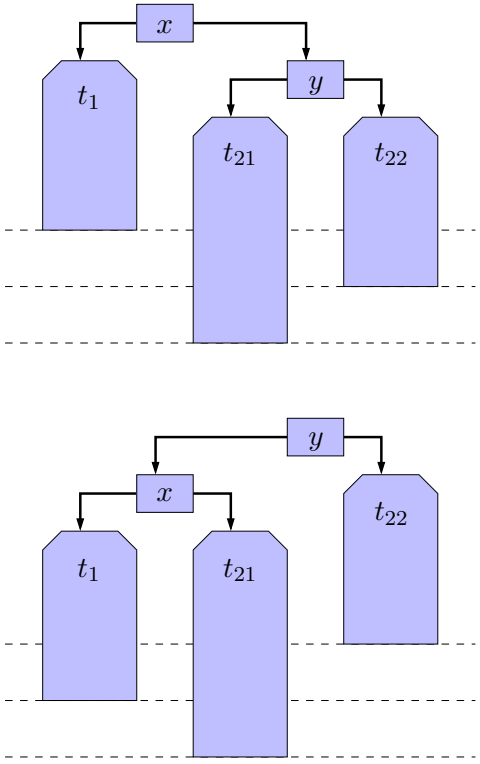


Abbildung 3.16: Für $\text{bal}(t_2) = -1$ bewirkt die Links-Rotation keine Verbesserung. Dieser Fall erfordert eine Doppelrotation.

lässt sich lösen, indem wir induktiv vorgehen: Der leere Baum ist immer ein AVL-Baum. Wenn wir zu einem AVL-Baum ein weiteres Element hinzufügen, kann die Bedingung $\text{bal}(t) \in \{-1, 0, 1\}$ in einigen Teilbäumen verloren gehen. Allerdings kann durch das Hinzufügen eines einzigen Elements die Balance offenbar im ungünstigen Fall die Werte -2 oder 2 annehmen, da der Baum schließlich vor dem Hinzufügen balanciert war. Wir brauchen also einen Algorithmus, um derartige Störungen zu korrigieren. Der Algorithmus muss dabei so beschaffen sein, dass er mit wenigen Operationen auskommt und sicherstellt, dass das Ergebnis nicht nur ein AVL-Baum, sondern auch weiterhin ein Suchbaum ist.

Die Wiederherstellung der AVL-Eigenschaft in einem Baum lässt sich besonders elegant durch *Baumrotationen* erreichen, bei denen wir die Wurzel eines Teilbaums verändern, ohne die für einen Suchbaum erforderliche Ordnung aufzugeben: Die *Links-Rotation* wirkt auf einen Baum $t \in \mathcal{T}_M$ mit $t = (t_1, t_2, x)$ und $t_2 = (t_{21}, t_{22}, y)$. Sie ersetzt ihn durch den Baum $t' := ((t_1, t_{21}, x), t_{22}, y)$, bei dem t_{22} näher an die Wurzel heran rückt und t_1 sich weiter von ihr entfernt.

Als Beispiel sind die Bäume t (oben) und t' (unten) in Abbildung 3.15 dargestellt. Im Interesse der Lesbarkeit werden die Teilbäume t_1, t_{21} und t_{22} dabei lediglich durch große

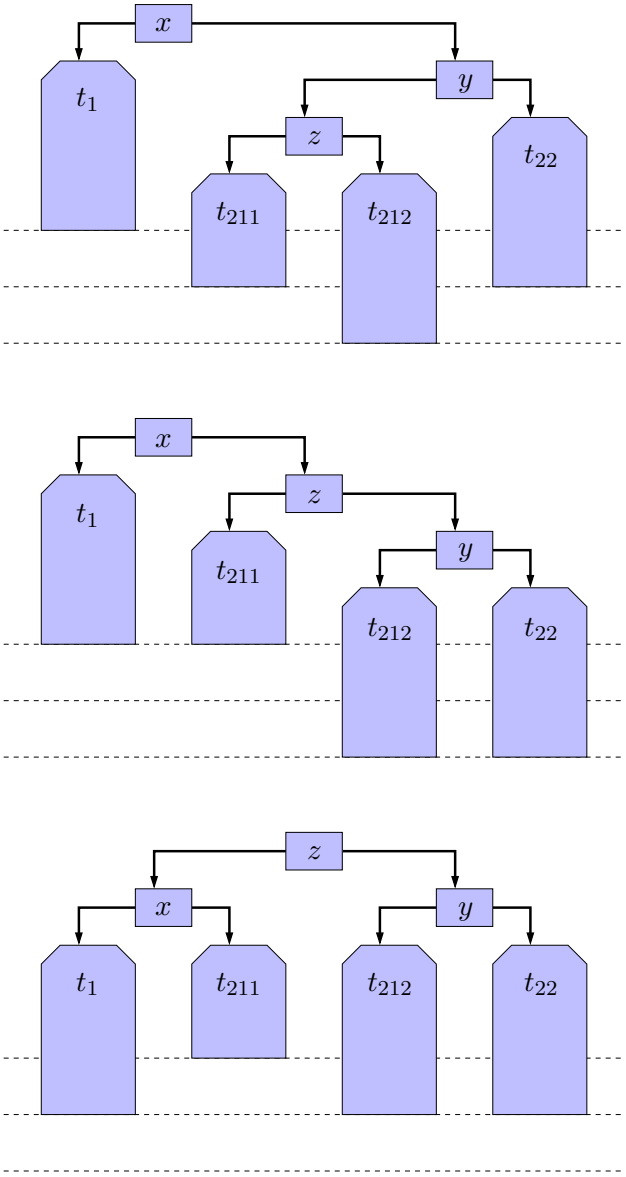


Abbildung 3.17: Rechts-Links-Doppelrotation in einem AVL-Baum t: Erst wird t_2 nach rechts rotiert, dann t nach links.

Blöcke repräsentiert, da ihr interner Aufbau von uns nicht von Interesse ist. Die Höhen der Teilbäume sind durch die Höhen der Blöcke angedeutet: Auf der linken Seite gelten $\text{height}(t_1) = \text{height}(t_{21}) = \text{height}(t_{22}) - 1$, also wegen $\text{height}(t_2) = \text{height}(t_{22}) + 1$ auch $\text{bal}(t) = 2$ und $\text{bal}(t_2) = 1$. Auf der rechten Seite haben wir $\text{height}(t_1) = \text{height}(t_{21}) - 1 = \text{height}(t_{22}) - 1$, also $\text{bal}(t) = 2$ und $\text{bal}(t_2) = 0$. In beiden Fällen spricht man von einer *Rechts-Rechts-Situation*, weil der rechte Teilbaum des Baums t und der rechte Teilbaum

des rechten Teilbaums t_2 jeweils höher (oder wenigstens nicht weniger hoch) als die korrespondierenden linken Teilbäume sind. Eine Balance von 2 ist ein Verstoß gegen die Anforderungen an AVL-Bäume, die sich durch die Links-Rotation beheben lässt: Für t' können wir an der Grafik $\text{bal}(t') = 0$ sowie $\text{bal}(t'_1) = 0$ ablesen. Wenn wir davon ausgehen, dass t_1, t_{21} und t_{22} bereits AVL-Bäume sind, sind damit nach der Linksrotation auch t' und t'_1 wieder AVL-Bäume.

Leider genügt eine Links-Rotation nicht immer, um eine Balance von $\text{bal}(t) = 2$ zu korrigieren: Falls $\text{bal}(t_2) = -1$ gelten sollte, liegt die in Abbildung 3.16 dargestellte *Rechts-Links-Situation* vor: Der rechte Teilbaum t_2 des Baums t ist höher, aber der linke Teilbaum t_{21} dieses Teilbaums t_2 ist höher als der rechte Teilbaum t_{22}. In diesem Fall sorgt die Links-Rotation dafür, dass die Balance des Baums t' den Wert -2 annimmt, der die AVL-Eigenschaft immer noch verletzt.

Diese Schwierigkeit können wir umgehen, indem wir zunächst eine analog zur Links-Rotation definierte *Rechts-Rotation* für den Teilbaum t_2 durchführen, die für eine positive Balance sorgt. Damit sind wir wieder in einer Rechts-Rechts-Situationen und können mit der bereits bekannten Links-Rotation fortfahren. Die resultierende *Rechts-Links-Doppelrotation* ist in Abbildung 3.17 dargestellt. Zumindest in der dort dargestellten Situation sorgt sie dafür, dass wir wieder einen AVL-Baum erhalten.

Lemma 3.12 (Rotationen) *Sei $t \in \mathcal{T}_M$ ein Suchbaum.*

Falls $\text{bal}(t) > 0$ *gilt, folgen $t \neq \emptyset$ sowie $t_2 \neq \emptyset$, so dass wir $t = (t_1, (t_{21}, t_{22}, y), x)$ für geeignete $x, y \in M$ und $t_1, t_{21}, t_{22} \in \mathcal{T}_M$ erhalten. Die* Linksrotation *dieses Baums ist durch*

$$t' := ((t_1, t_{21}, x), t_{22}, y)$$

definiert. t' ist wieder ein Suchbaum und erfüllt

$$h' = h + 1 - \begin{cases} \min\{\text{bal}(t), 1 + \text{bal}(t_2), 2\} & \textit{falls } \text{bal}(t_2) \geq 0, \\ \min\{\text{bal}(t), 2 - \text{bal}(t_2), 1\} & \textit{ansonsten}, \end{cases}$$

$$\text{bal}(t') = -2 + \begin{cases} \min\{\text{bal}(t), 1 + \text{bal}(t_2)\} & \textit{falls } \text{bal}(t_2) \geq 0, \\ \min\{\text{bal}(t), 1\} + \text{bal}(t_2) & \textit{ansonsten}, \end{cases}$$

$$\text{bal}(t'_1) = -1 + \text{bal}(t) - \max\{0, \text{bal}(t_2)\}.$$

Falls $\text{bal}(t) < 0$ *gilt, folgen $t \neq \emptyset$ sowie $t_1 \neq \emptyset$, so dass wir $t = ((t_{11}, t_{12}, x), t_2, y)$ für geeignete $x, y \in M$ und $t_{11}, t_{12}, t_2 \in \mathcal{T}_M$ erhalten. Die* Rechtsrotation *dieses Baums ist durch*

$$t' := (t_{11}, (t_{12}, t_2, y), x)$$

definiert. t' ist wieder ein Suchbaum und erfüllt

$$h' = h + 1 - \begin{cases} \min\{-\text{bal}(t), 1 - \text{bal}(t_1), 2\} & \textit{falls } \text{bal}(t_1) \leq 0, \\ \min\{-\text{bal}(t), 2 + \text{bal}(t_1), 1\} & \textit{ansonsten}, \end{cases}$$

$$\text{bal}(t') = 2 - \begin{cases} \min\{-\text{bal}(t), 1 - \text{bal}(t_1)\} & \textit{falls } \text{bal}(t_1) \leq 0, \\ \min\{-\text{bal}(t), 1\} + \text{bal}(t_1) & \textit{ansonsten}, \end{cases}$$

$$\mathrm{bal}(t_2') = 1 + \mathrm{bal}(t) + \max\{0, -\mathrm{bal}(t_1)\}.$$

Beweis. Gelte $\mathrm{bal}(t) > 0$. Dann kann t nicht der leere Baum sein, und t_2 kann ebenfalls nicht der leere Baum sein.

Da t ein Suchbaum ist, gelten nach Definition 3.7 und wegen $\mathrm{labels}(t_{21}), \mathrm{labels}(t_{22}) \subseteq \mathrm{labels}(t_2)$ die Beziehungen

$$\mathrm{labels}(t_1) \trianglelefteq x \lhd y, \qquad x \lhd \mathrm{labels}(t_{21}) \trianglelefteq y, \qquad x \lhd y \lhd \mathrm{labels}(t_{22}),$$

so dass auch $t_1' = (t_1, t_{21}, x)$ und $t' = (t_1', t_{22}, y)$ Suchbäume sind.

Etwas aufwendiger gestaltet sich die Berechnung der Höhe und der Balancen der neuen Bäume. Zur Abkürzung bezeichnen wir mit $h, h_1, h_2, h_{21}, h_{22}, h', h_1'$ die Höhen der Bäume $t, t_1, t_2, t_{21}, t_{22}, t', t_1'$.

Wegen $\mathrm{bal}(t) > 0$ erhalten wir mit Lemma 3.6

$$0 < \mathrm{bal}(t) = h_2 - h_1, \qquad h_1 < h_2, \qquad h = \max\{h_1, h_2\} + 1 = h_2 + 1,$$
$$h_2 = h - 1, \qquad h_1 = h - 1 - \mathrm{bal}(t),$$

können also h_1 und h_2 durch h und $\mathrm{bal}(t)$ ausdrücken.

Erster Fall: Rechts-Rechts-Situation. Falls $\mathrm{bal}(t_2) \geq 0$ gilt, folgt

$$0 \leq \mathrm{bal}(t_2) = h_{22} - h_{21}, \qquad h_{21} \leq h_{22}, \qquad h_2 = \max\{h_{21}, h_{22}\} + 1 = h_{22} + 1,$$
$$h_{22} = h_2 - 1 = h - 2, \qquad h_{21} = h_{22} - \mathrm{bal}(t_2) = h - 2 - \mathrm{bal}(t_2),$$

also können wir auch h_{21} und h_{22} durch h und $\mathrm{bal}(t_2)$ ausdrücken.

Aus $t_1' = (t_1, t_{21}, x)$ und $t' = (t_1', t_{22}, y)$ folgen

$$\begin{aligned}
h_1' &= \max\{h_1, h_{21}\} + 1 \\
&= \max\{h - 1 - \mathrm{bal}(t), h - 2 - \mathrm{bal}(t_2)\} + 1 \\
&= \max\{h - \mathrm{bal}(t), h - 1 - \mathrm{bal}(t_2)\} \\
&= h - \min\{\mathrm{bal}(t), 1 + \mathrm{bal}(t_2)\}, \\
h' &= \max\{h_1', h_{22}\} + 1 \\
&= \max\{h - \mathrm{bal}(t), h - 1 - \mathrm{bal}(t_2), h - 2\} + 1 \\
&= h + 1 - \min\{\mathrm{bal}(t), 1 + \mathrm{bal}(t_2), 2\}, \\
\mathrm{bal}(t_1') &= h_{21} - h_1 = h - 2 - \mathrm{bal}(t_2) - (h - 1 - \mathrm{bal}(t)) \\
&= -1 + \mathrm{bal}(t) - \mathrm{bal}(t_2), \\
\mathrm{bal}(t') &= h_{22} - h_1' = h - 2 - (h - \min\{\mathrm{bal}(t), 1 + \mathrm{bal}(t_2)\}) \\
&= \min\{\mathrm{bal}(t), 1 + \mathrm{bal}(t_2)\} - 2.
\end{aligned}$$

Zweiter Fall: Rechts-Links-Situation. Falls $\mathrm{bal}(t_2) \leq 0$ gilt, folgt

$$0 \geq \mathrm{bal}(t_2) = h_{22} - h_{21}, \qquad h_{22} \leq h_{21}, \qquad h_2 = \max\{h_{21}, h_{22}\} + 1 = h_{21} + 1,$$
$$h_{21} = h_2 - 1 = h - 2, \qquad h_{22} = h_{21} + \mathrm{bal}(t_2) = h - 2 + \mathrm{bal}(t_2),$$

```
1  tree *
2  rotateleft_tree(tree *t)
3  {
4    tree *t2;
5    t2 = t->right;
6    t->right = t2->left;
7    t2->left = t;
8    return t2;
9  }
10 tree *
11 rotateright_tree(tree *t)
12 {
13   tree *t1;
14   t1 = t->left;
15   t->left = t1->right;
16   t1->right = t;
17   return t1;
18 }
```

Abbildung 3.18: Baumrotationen

also können wir wieder h_{21} und h_{22} durch h und $\mathrm{bal}(t_2)$ ausdrücken.

Aus $t_1' = (t_1, t_{21}, x)$ und $t' = (t_1', t_{22}, y)$ folgen

$$\begin{aligned}
h_1' &= \max\{h_1, h_{21}\} + 1 \\
&= \max\{h - 1 - \mathrm{bal}(t), h - 2\} + 1 \\
&= \max\{h - \mathrm{bal}(t), h - 1\} + 1 \\
&= h - \min\{\mathrm{bal}(t), 1\}, \\
h' &= \max\{h_1', h_{22}\} + 1 \\
&= \max\{h - \mathrm{bal}(t), h - 1, h - 2 + \mathrm{bal}(t_2)\} + 1 \\
&= h + 1 - \min\{\mathrm{bal}(t), 1, 2 - \mathrm{bal}(t_2)\}, \\
\mathrm{bal}(t_1') &= h_{21} - h_1 = h - 2 - (h - 1 - \mathrm{bal}(t)) \\
&= -1 + \mathrm{bal}(t), \\
\mathrm{bal}(t') &= h_{22} - h_1' = h - 2 + \mathrm{bal}(t_2) - (h - \min\{\mathrm{bal}(t), 1\}) \\
&= \mathrm{bal}(t_2) + \min\{\mathrm{bal}(t), 1\} - 2.
\end{aligned}$$

Für die Rechtsrotation können wir entsprechend vorgehen. ■

Mit Hilfe dieses Lemmas können wir verifizieren, dass sich tatsächlich durch Rotationen die AVL-Eigenschaft eines Baums wiederherstellen lässt, nachdem ein Element eingefügt wurde.

Satz 3.13 (Konstruktion AVL-Baum) *Sei $t \in \mathcal{T}_M \setminus \{\emptyset\}$ ein nicht-leerer Suchbaum, und seien t_1 und t_2 AVL-Bäume.*

Falls $\mathrm{bal}(t) = 2$ *und* $\mathrm{bal}(t_2) \geq 0$ *(Rechts-Rechts-Situation) gelten, können wir mit einer Linksrotation t zu einem AVL-Baum machen.*

Falls $\mathrm{bal}(t) = 2$ *und* $\mathrm{bal}(t_2) < 0$ *(Rechts-Links-Situation) gelten, können wir dieses Ziel durch eine Rechtsrotation des Teilbaums t_2 und eine Linksrotation des resultierenden Gesamtbaums erreichen.*

Falls $\mathrm{bal}(t) = -2$ *und* $\mathrm{bal}(t_1) \leq 0$ *(Links-Links-Situation) gelten, können wir mit einer Rechtsrotation t zu einem AVL-Baum machen.*

Falls $\mathrm{bal}(t) = -2$ *und* $\mathrm{bal}(t_1) > 0$ *(Links-Rechts-Situation) gelten, können wir dieses Ziel durch eine Linksrotation des Teilbaums t_1 und eine Rechtsrotation des resultierenden Gesamtbaums erreichen.*

Beweis. Da t nicht der leere Baum ist, gilt $t = (t_1, t_2, x)$ für ein $x \in M$.

Wir behandeln zuerst den Fall $\mathrm{bal}(t) = 2$. Dann kann t_2 nicht der leere Baum sein, also muss auch $t_2 = (t_{21}, t_{22}, y)$ für ein $y \in M$ gelten.

1. Fall: Rechts-Rechts-Situation. Es gelte $\mathrm{bal}(t_2) \geq 0$. Da t_2 ein AVL-Baum ist, folgt $\mathrm{bal}(t_2) \in \{0, 1\}$. Die Linksrotation führt zu einem Baum $t' = ((t_1, t_{21}, x), t_{22}, y)$. Nach Lemma 3.12 gelten dann $\mathrm{bal}(t') \in \{-1, 0\}$ und $\mathrm{bal}(t'_1) \in \{0, 1\}$. Damit ist t' ein AVL-Baum.

2. Fall: Rechts-Links-Situation. Es gelte $\mathrm{bal}(t_2) < 0$. Da t_2 ein AVL-Baum ist, folgt $\mathrm{bal}(t_2) = -1$. Dann ist t_{21} nicht der leere Baum, es muss also $t_{21} = (t_{211}, t_{212}, z)$ für ein $z \in M$ gelten. Die Rechtsrotation des Teilbaums t_2 führt zu einem Baum $t'_2 = (t_{211}, (t_{212}, t_{22}, y), z)$. Nach Lemma 3.12 gelten dann $\mathrm{bal}(t'_2) \in \{0, 1\}$ und $\mathrm{bal}(t'_{22}) \in \{0, 1\}$, also ist t'_2 ein AVL-Baum. Dank $\mathrm{bal}(t'_2) \geq 0$ sind wir nun für $t' := (t_1, t'_2, x)$ in der Rechts-Rechts-Situation und können wie im ersten Fall fortfahren.

Wir behandeln nun den Fall $\mathrm{bal}(t) = -2$. Dann kann t_1 nicht der leere Baum sein, also muss auch $t_1 = (t_{11}, t_{12}, y)$ für ein $y \in M$ gelten.

3. Fall: Links-Links-Situation. Es gelte $\mathrm{bal}(t_1) \leq 0$. Da t_1 ein AVL-Baum ist, folgt $\mathrm{bal}(t_1) \in \{-1, 0\}$. Die Rechtsrotation führt zu einem Baum $t' = (t_{11}, (t_{12}, t_2, x), y)$. Nach Lemma 3.12 gelten dann $\mathrm{bal}(t') \in \{0, 1\}$ und $\mathrm{bal}(t'_2) \in \{-1.0\}$. Damit ist t' ein AVL-Baum.

4. Fall: Links-Rechts-Situation. Es gelte $\mathrm{bal}(t_1) > 0$. Da t_1 ein AVL-Baum ist, folgt $\mathrm{bal}(t_1) = 1$. Dann ist t_{12} nicht der leere Baum, es muss also $t_{12} = (t_{121}, t_{122}, z)$ für ein $z \in M$ gelten. Die Linksrotation des Teilbaums t_1 führt zu einem Baum $t'_1 = ((t_{11}, t_{121}, y), t_{122}, z)$. Nach Lemma 3.12 gelten dann $\mathrm{bal}(t'_1) \in \{-1, 0\}$ und $\mathrm{bal}(t'_{11}) \in \{-1, 0\}$, also ist t'_1 ein AVL-Baum. Dank $\mathrm{bal}(t'_1) \leq 0$ sind wir nun für $t' := (t'_1, t_2, x)$ in der Links-Links-Situation und können wie im dritten Fall fortfahren. ∎

Wir können die in Satz 3.13 skizzierte Vorgehensweise unmittelbar in einem Algorithmus umsetzen: Wenn wir ein neues Element in einen AVL-Baum einfügen, suchen wir zunächst ausgehend von der Wurzel die Position, an die das Element gehört. Anschließend können die Balancen nur in den dabei besuchten Teilbäumen verändert worden sein, so dass wir auch nur diese Teilbäume zu korrigieren brauchen.

```
1  tree *
2  addto_avl_tree(tree *t, payload x)
3  {
4    if(t) {
5      if(compare(x, t->x) <= 0) {
6        t->left = addto_avl_tree(t->left, x);
7        if(balance(t) == -2) {
8          if(balance(t->left) > 0)
9            t->left = rotateleft_tree(t->left);
10         t = rotateright_tree(t);
11       }
12     }
13     else {
14       t->right = addto_avl_tree(t->right, x);
15       if(balance(t) == 2) {
16         if(balance(t->right) < 0)
17           t->right = rotateright_tree(t->right);
18         t = rotateleft_tree(t);
19       }
20     }
21   }
22   else
23     t = new_tree(x, 0, 0);
24   return t;
25 }
```

Abbildung 3.19: Einfügen in einen AVL-Baum

Diese Aufgabe können wir sehr elegant lösen, indem wir die Korrekturen in unserem Algorithmus nach der Rückkehr aus dem rekursiven Aufruf ausführen, indem wir also zunächst den Teilbaum mit der geringsten Höhe behandeln und dann zu immer höheren übergehen, bis der gesamte Baum wieder die AVL-Eigenschaft aufweist.

Bei dieser Vorgehensweise ist sichergestellt, dass alle Teilbäume des aktuell zu behandelnden Baums bereits wieder AVL-Bäume sind, so dass sich Satz 3.13 anwenden lässt. Außerdem ist uns auch bekannt, in welchem Teilbaum das neue Element eingefügt wurde, so dass wir eine Fallunterscheidung einsparen können: Falls im linken Teilbaum eingefügt wurde, können nur eine Links-Links- oder eine Links-Rechts-Situation eintreten, anderenfalls nur eine Rechts-Rechts- oder eine Rechts-Links-Situation. Der resultierende Algorithmus ist in Abbildung 3.19 zusammengefasst.

Die Rotationen führen im ungünstigsten Fall zu $\Theta(1)$ zusätzlichen Operationen pro Aufruf der Funktion `addto_avl_tree`. Das Einfügen in einen Baum t erfordert höchstens $\text{height}(t)$ Aufrufe, und nach Folgerung 3.11 ist die Höhe für einen Baum mit n Elementen in $\Theta(\log_2(n))$, so dass wir mit Lemma 2.20 einen Gesamtaufwand von $\Theta(\log_2(n))$

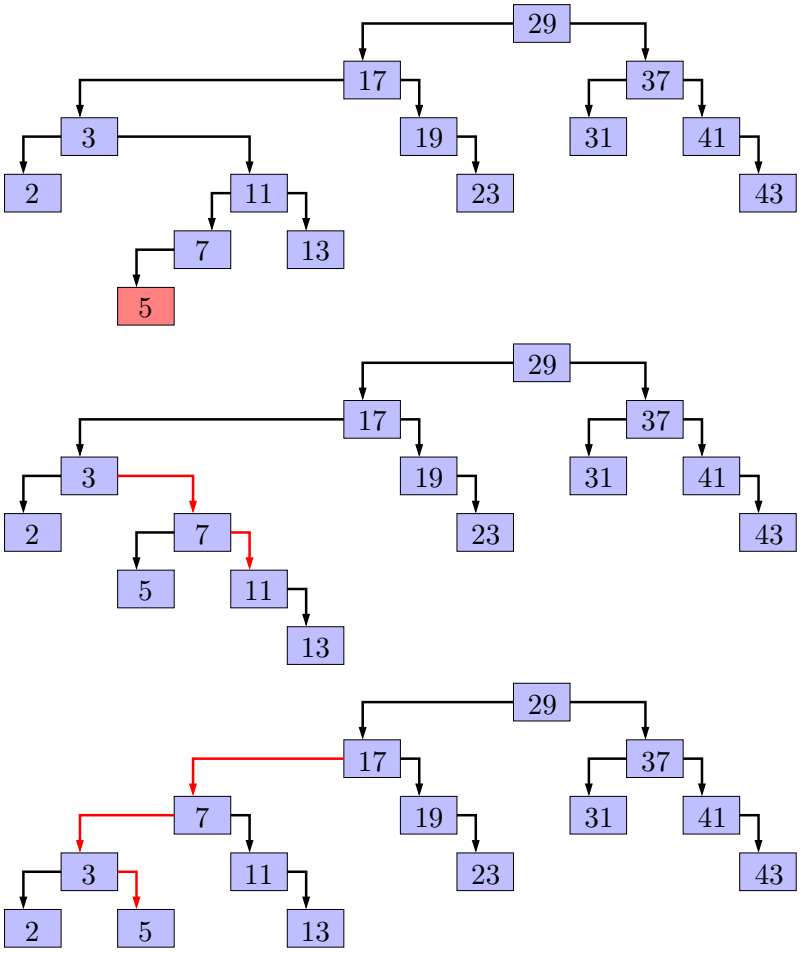


Abbildung 3.20: Wiederherstellen der AVL-Baum-Eigenschaft, nachdem das Element 5 eingefügt wurde. Die AVL-Bedingung ist zunächst im Element 3 verletzt, die Rechts-Links-Situation wird durch eine Doppelrotation korrigiert.

erhalten.

Natürlich gilt diese Abschätzung nur, falls wir bal(t) in $\mathcal{O}(1)$ Operationen berechnen können. Dieses Ziel lässt sich beispielsweise erreichen, indem wir den Verbundtyp `tree` um die Höhe oder (sparsamer, aber auch aufwendiger) die Balance des korrespondierenden Baums erweitern. Auf jeden Fall muss dabei die vollständige Neuberechnung der Höhe vermieden werden, denn sie erfordert $\Theta(n)$ Operationen. Glücklicherweise können wir auch in diesem Fall ausnutzen, dass das Einfügen eines Elements nur die Bäume beeinflusst, die unser Algorithmus durchläuft, so dass sich die Höhe elegant aktualisieren lässt.

Das Löschen eines Elements in einem AVL-Baum ist nur geringfügig schwieriger: Um

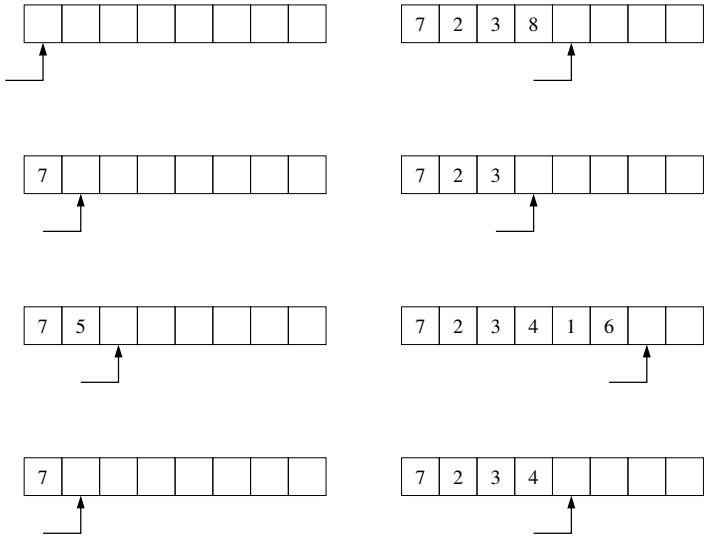


Abbildung 3.21: Darstellung eines Kellerspeichers durch ein Array. Der Pfeil markiert den Kopf des Kellers.

die Wurzel eines Baums t zu löschen, ersetzen wir sie durch das größte Element ihres linken Teilbaums t_1, falls das nicht der leere Baum ist. Anderenfalls können wir t unmittelbar durch t_2 ersetzen. Dabei verändern sich in den durchlaufenen Teilbäumen die Balancen, so dass wir mit passenden Rotationen die AVL-Eigenschaft wiederherstellen müssen.

3.6 Dynamische Datenstrukturen in Arrays

Falls wir wissen, dass eine Datenstruktur niemals mehr als n Elemente enthalten wird, können wir den relativ hohen Verwaltungsaufwand vermeiden, der mit dem Anlegen einer Liste oder eines Baums verbunden ist, indem wir die Datenstruktur durch ein Array darstellen.

Keller. Beispielsweise können wir einen Kellerspeicher durch ein Array `stack` der Länge `stacksize` und einen Zähler `head` darstellen, der jeweils angibt, wieviele Elemente der Keller bereits enthält. Abbildung 3.21 zeigt den Zustand des Arrays und des Zählers nach `push(7)`, `push(5)`, `pop()`, `push(2);push(3);push(8)`, `pop()`, `push(4);push(1);push(6)` und `pop();pop()`.

Die `push`- und `pop`-Operationen lassen sich in dieser Darstellung sehr einfach und effizient umsetzen: Bei der ersten schreiben wir das neue Element in den Eintrag `stack[head]` und zählen `head` herauf, bei der zweiten zählen wir `head` herunter und geben `stack[head]` zurück. Eine mögliche Implementierung findet sich in Abbildung 3.22.

```
1  void
2  push(payload x)
3  {
4    if(head == stacksize) crash("Stack overflow");
5    stack[head] = x;
6    head++;
7  }
8  payload
9  pop()
10 {
11   if(head == 0) crash("Stack empty");
12   --head;
13   return stack[head];
14 }
15 int
16 isempty()
17 {
18   return (head == 0);
19 }
```

Abbildung 3.22: Implementierung eines Kellerspeichers mit Hilfe eines Arrays.

Da in diesem Fall die Größe des Kellers beschränkt ist, müssen wir in der Funktion `push` darauf achten, dass nicht mehr Elemente in den Keller geschrieben werden, als er aufnehmen kann.

Sehr viele Rechnersysteme verwenden in dieser Weise implementierte Kellerspeicher, um die von Funktionen benötigten Parameter und lokalen Variablen zu verwalten. Dieser Ansatz vereinfacht beispielsweise die Umsetzung rekursiver Funktionen erheblich, hat aber den Nachteil, dass bei zu tief verschachtelten Aufrufen der für den Keller vorgesehene Speicherbereich ausgeschöpft ist und das Programm mit einem *stack overflow* beendet wird.

Warteschlange. Warteschlangen lassen sich ebenfalls durch Arrays repräsentieren: Wir verwenden ein Array `queue` der Größe `queuesize` und einen Zähler `tail`, der angibt, an der wievielten Stelle des Arrays das nächste neue Element eingefügt werden soll. Eine `enqueue`-Operation schreibt das neue Element in `queue[tail]` und zählt `tail` herauf, eine `dequeue`-Operation gibt das Element in `queue[0]` zurück und kopiert alle folgenden Element um eine Position nach vorne.

Offenbar ist dieser Zugang wenig effizient, weil jede `dequeue`-Operation einen Aufwand erfordert, der proportional zu der aktuellen Länge der Warteschlange ist.

Wesentlich eleganter lässt sich die Aufgabe lösen, indem wir einen weiteren Zähler `head` hinzu nehmen, der auf das älteste Element zeigt, das wir bei der nächsten `dequeue`-Operation erhalten werden. Dann müssen wir lediglich `head` um eins herauf zählen, um

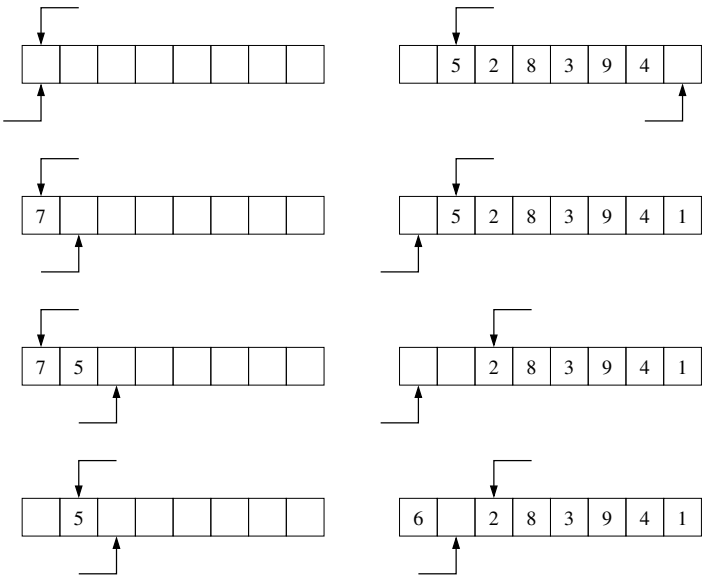


Abbildung 3.23: Darstellung einer Warteschlange durch ein Array. Der obere Pfeil markiert jeweils den Kopf der Warteschlange, der untere deren Ende.

das nächste Element der Warteschlange zu erhalten.

Leider wäre in diesem Fall die Datenstruktur nach `queuesize` `enqueue`-Operationen ausgeschöpft, selbst wenn die Warteschlange niemals mehr als ein Element enthalten haben sollte: `tail` wird niemals heruntergezählt.

Dieses Problem lässt sich elegant lösen, indem wir das Array *zyklisch* verwenden: Sobald `head` oder `tail` den Wert `queuesize` erreichen, werden sie wieder auf null gesetzt. Dadurch kann die Warteschlange im Prinzip beliebig viele `enqueue`-Operationen durchführen, solange niemals mehr als `queuesize` Elemente *gleichzeitig* verwaltet werden müssen.

Als Beispiel ist in Abbildung 3.23 eine Warteschlange dargestellt, die der Reihe nach die Operationen `enqueue(7)`, `enqueue(5)`, `dequeue()` ausführt, dann mittels `enqueue(2);enqueue(8);enqueue(3);enqueue(9);enqueue(4)` das Array füllt, mit `enqueue(1)` das Zurücksetzen des `tail`-Zählers bewirkt und schließlich `dequeue()` und `enqueue(6)` ausführt.

Allerdings wird es bei dieser Vorgehensweise schwierig, zu unterscheiden, ob die Warteschlange kein Element oder `queuesize` Elemente enthält. Es gibt verschiedene Möglichkeiten, dieses Problem zu lösen, beispielsweise indem man zählt, wie oft `head` und `tail` jeweils auf null zurückgesetzt wurden. Falls einer der beiden Zähler ungerade und der andere gerade ist, ist die Warteschlange voll, ansonsten leer.

Sehr viel einfacher ist es allerdings, jeweils den Eintrag vor `head` (zyklisch gerechnet)

```
1  void
2  enqueue(payload x)
3  {
4    int next_tail = (tail+1) % queuesize;
5    if(next_tail == head) crash("Queue overflow");
6    queue[tail] = x;
7    tail = next_tail;
8  }
9  payload
10 dequeue()
11 {
12   payload x;
13   if(head == tail) crash("Queue empty");
14   x = queue[head];
15   head = (head + 1) % queuesize;
16   return x;
17 }
18 int
19 isempty()
20 {
21   return (head == tail);
22 }
```

Abbildung 3.24: Implementierung einer Warteschlange mit Hilfe eines Arrays.

ungenutzt zu lassen, denn dann können `head` und `tail` nur übereinstimmen, falls die Warteschlange leer ist. Eine mögliche Implementierung ist in Abbildung 3.24 dargestellt.

In dieser Implementierung verwenden wir den Modulo-Operator `%`, um dafür zu sorgen, dass die beiden Zähler bei Erreichen der Grenze `queuesize` wieder auf null zurückgesetzt werden. Im Interesse einer hohen Ausführungsgeschwindigkeit empfiehlt es sich häufig, eine Zweierpotenz für `queuesize` zu verwenden, so dass sich die Modulo-Operation durch eine einfach bitweise Und-Verknüpfung sehr effizient ausführen lässt.

Baum. Allgemeine Bäume durch ein Array darzustellen ist etwas komplizierter, allerdings können wir *vollständige binäre Bäume* relativ einfach verarbeiten. Einen binären Baum $t \in \mathcal{T}_M$ nennen wir *vollständig*, wenn entweder $t = \emptyset$ gilt oder t_1 und t_2 dieselbe Höhe aufweisen und ebenfalls vollständig sind. Ein Blick auf den Beweis zu Satz 3.9 zeigt, dass dann $\#t = 2^{\text{height}(t)} - 1$ gelten muss. Vollständige binäre Bäume zeichnen sich gerade dadurch aus, dass sie die maximale Anzahl von Elementen enthalten, die bei ihrer Höhe möglich ist. Da die Höhe für den Rechenaufwand der meisten Baum-Algorithmen den Ausschlag gibt, sind vollständige Bäume häufig besonders effizient.

Um einen vollständigen binären Baum durch ein Array darzustellen, müssen wir jedem Element des Baums einen Ort in dem Array zuordnen. Das sollte natürlich in einer Weise

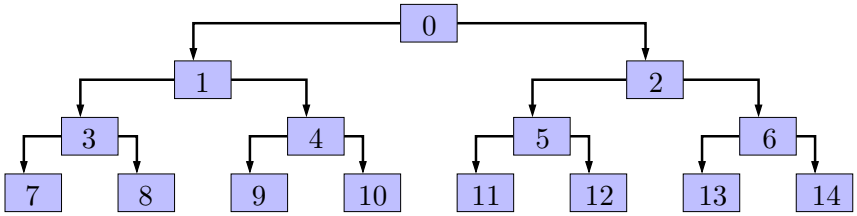


Abbildung 3.25: Stufenweise Numerierung der $15 = 2^4 - 1$ Elemente eines vollständigen binären Baums der Höhe 4.

geschehen, die es uns ermöglicht, typische Operationen wie den Wechsel zu einem linken oder rechten Teilbaum effizient auszuführen.

Ein einfacher Zugang besteht darin, die einzelnen Elemente des Baums nach ihrer „Entfernung“ von der Wurzel sortiert aufzuzählen, also zuerst die Wurzel, dann die Wurzeln ihrer Teilbäume, dann die Wurzeln der Teilbäume der Teilbäume und so weiter. Alle Elemente mit derselben Entfernung fassen wir zu einer *Stufe* des Baums zusammen, die Elemente einer Stufe speichern wir von links nach rechts sortiert in einem Array `x`.

Das Wurzelelement $\hat{t}$ wird damit in `x[0]` gespeichert, die Wurzelelemente $\hat{t}_1$ und $\hat{t}_2$ des linken und rechten Teilbaums in `x[1]` und `x[2]`, die Wurzelelemente $\hat{t}_{11}$ und $\hat{t}_{12}$ des linken und rechten Teilbaums des linken Teilbaums in `x[3]` und `x[4]` sowie $\hat{t}_{21}$ und $\hat{t}_{22}$ in `x[5]` und `x[6]`.

Da sich die Anzahl der Elemente auf jeder Stufe verdoppelt, gehört zu dem am weitesten links stehenden Element auf Stufe ℓ gerade der Index

$$\sum_{k=0}^{\ell-1} 2^k = 2^\ell - 1,$$

so dass wir durch Multiplikation mit zwei und Addition von eins gerade den Index auf der nächsten Stufe $\ell + 1$ erhalten. Der Index des rechten Teilbaums eines Baums ergibt sich aus dem des linken Teilbaums durch Addition von eins, so dass sich die folgende Numerierung anbietet:

- Das Wurzelelement erhält die Nummer 0.
- Falls das Wurzelement eines Teilbaums die Nummer k trägt, erhält das Wurzelelement des linken Teilbaums die Nummer $2k + 1$.
- Das des rechten Teilbaums erhält die Nummer $2k + 2$.

Die Numerierung für einen vollständigen binären Baum der Höhe 4 ist in Abbildung 3.25 dargestellt.

Diese Darstellung des Baums hat den Vorteil, dass sie ohne jegliche Verwaltungsinformationen auskommt, schließlich werden nur die eigentlichen Daten gespeichert, keinerlei zusätzliche Zeiger. Der Preis für diese Effizienz ist die erheblich eingeschränkte Flexibilität: Wir können nur Bäume darstellen, die genau $2^h - 1$ Elemente für ein $h \in \mathbb{N}_0$

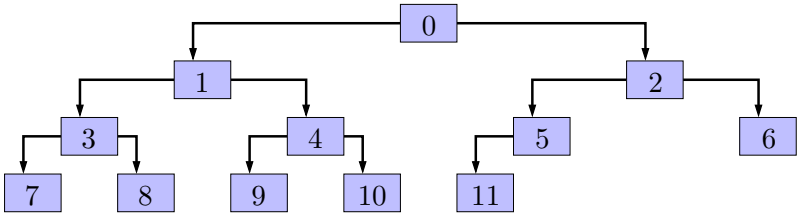


Abbildung 3.26: Stufenweise Numerierung der $n = 12$ Elemente eines fast vollständigen binären Baums der Höhe 4.

enthalten. Damit ist es beispielsweise nicht möglich, einzelne Elemente hinzuzufügen oder zu löschen.

3.7 Heapsort

Mit Hilfe der im vorangehenden Abschnitt eingeführten Darstellung eines binären Baums durch ein Array lässt sich ein sehr effizientes Sortierverfahren entwickeln, das, ähnlich wie der Mergesort-Algorithmus, lediglich $\mathcal{O}(n \log_2(n))$ Operationen für ein Array der Länge n benötigt, aber auch, ähnlich wie der Quicksort-Algorithmus, nur $\mathcal{O}(1)$ zusätzliche Speicherplätze verwendet. Es verbindet also die Vorteile beider Verfahren und vermeidet deren Nachteile.

Die grundlegende Idee besteht darin, eine effiziente Methode zu entwickeln, mit der sich das größte Element des Arrays finden lässt. Wir verschieben es an das Ende des Arrays, also an seine endgültige Position, und wiederholen die Prozedur für den noch unsortierten Teil des Arrays, bis nur noch ein Array der Länge eins übrig bleibt.

Sei also `x` ein Array der Länge n, das wir aufsteigend sortieren wollen. Wenn $n = 2^h - 1$ gelten würde, also $h = \log_2(n+1)$, könnten wir `x` in der im vorigen Abschnitt diskutierten Weise als vollständigen binären Baum interpretieren. Anderenfalls verwenden wir einen *fast vollständigen binären Baum* der Höhe $h = \lceil \log_2(n+1) \rceil$, der sich von einem vollständigen Baum nur dadurch unterscheidet, dass die Elemente mit den Nummern $n, n+1, \ldots, 2^h - 2$ fehlen. Ein Beispiel für $n = 12$ ist in Abbildung 3.26 dargestellt.

Wie bereits erwähnt besteht unser Ziel darin, das größte Element des Arrays zu ermitteln. Dazu wäre es am günstigsten, wenn dieses Element in der Wurzel des Baums stehen würde, denn dann könnten wir es unmittelbar ablesen, es sollte also

$$\text{labels}(t) \trianglelefteq \hat{t}$$

gelten. Um diese Eigenschaft sicher zu stellen, müssten wir allerdings bei einer naiven Vorgehensweise das Element mit allen anderen Elementen des Baums vergleichen.

Wesentlich eleganter ist es, zu fordern, dass die größten Elemente der *Teilbäume* t_1 und t_2 auch in deren Wurzeln stehen, dass also

$$\text{labels}(t_1) \trianglelefteq \hat{t}_1, \qquad \text{labels}(t_2) \trianglelefteq \hat{t}_2$$

gilt. Dann würde es nämlich genügen,

$$(\hat{t}_1 \trianglelefteq \hat{t}) \wedge (\hat{t}_2 \trianglelefteq \hat{t})$$

zu überprüfen und die Transitivität (2.19) der Ordnung auszunutzen.

Dieselbe Argumentation können wir auch auf die Teilbäume anwenden und gelangen so zu der folgenden Eigenschaft:

Definition 3.14 (Halde) *Wir nennen einen Baum $t \in \mathcal{T}_M$* Halde *(engl.* heap*), falls $t = \emptyset$ gilt oder seine Teilbäume t_1 und t_2 Halden sind und die Bedingung*

$$t_1 \neq \emptyset \Rightarrow \hat{t}_1 \trianglelefteq \hat{t}, \qquad t_2 \neq \emptyset \Rightarrow \hat{t}_2 \trianglelefteq \hat{t}. \tag{3.10}$$

erfüllen.

Aus dieser „lokalen“ Eigenschaft des Baums folgt bereits die gewünschte „globale“ Maximalitätsaussage:

Lemma 3.15 (Halde) *Sei $t \in \mathcal{T}_M$ eine Halde. Dann gilt*

$$t \neq \emptyset \Rightarrow \operatorname{labels}(t) \trianglelefteq \hat{t}, \tag{3.11}$$

das Wurzelelement ist also das Maximum aller Elemente.

Beweis. Wir zeigen per struktureller Induktion für alle $t \in \mathcal{T}_M$ die Aussage

$$t \text{ ist eine Halde mit } t \neq \emptyset \Rightarrow \operatorname{labels}(t) \trianglelefteq \hat{t}. \tag{3.12}$$

Induktionsanfang. Für den leeren Baum $t = \emptyset$ ist nichts zu beweisen.
Induktionsvoraussetzung. Seien $t_1, t_2 \in \mathcal{T}_M$ Bäume, für die (3.12) gilt.
Induktionsschritt. Sei $x \in X$ und $t = (t_1, t_2, x)$. Sei t eine Halde.
Nach Definition sind dann auch t_1 und t_2 Halden.
Falls $t_1 \neq \emptyset$ gilt, folgt aus (3.10) $\hat{t}_1 \trianglelefteq \hat{t}$. Da t_1 nach Induktionsvoraussetzung eine Halde ist, gilt mit (3.12)

$$\operatorname{labels}(t_1) \trianglelefteq \hat{t}_1,$$

und mit der Transitivität der Ordnung (2.19) folgt aus $\hat{t}_1 \trianglelefteq \hat{t}$ bereits

$$\operatorname{labels}(t_1) \trianglelefteq \hat{t}.$$

Falls $t_2 \neq \emptyset$ gilt, können wir entsprechend vorgehen, um $\operatorname{labels}(t_2) \trianglelefteq \hat{t}$ zu erhalten.
Da die Ordnung auch reflexiv ist, gilt $\hat{t} \trianglelefteq \hat{t}$, also insgesamt

$$\operatorname{labels}(t) = \operatorname{labels}(t_1) \cup \operatorname{labels}(t_2) \cup \{\hat{t}\} \trianglelefteq \hat{t}.$$

Das ist die Induktionsbehauptung. ∎

Natürlich nutzt uns diese Eigenschaft eines Baums nur etwas, wenn wir auch die Möglichkeit haben, sie herzustellen. Dazu können wir induktiv vorgehen: Falls $t =$

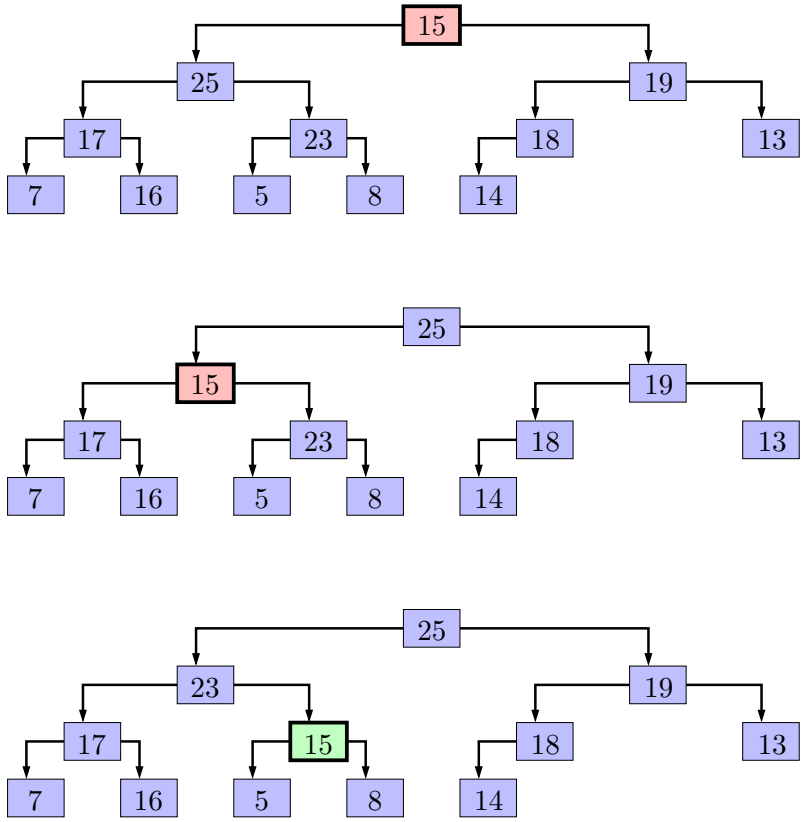


Abbildung 3.27: Wiederherstellung der Halden-Eigenschaft, die in der Wurzel des Baums gestört ist: Das Wurzelelement wird jeweils in den Teilbaum getauscht, dessen Wurzel größer ist.

(t_1, t_2, x) ein Baum ist und die beiden Teilbäume t_1 und t_2 bereits Halden sind, müssen wir nach Definition 3.14 lediglich dafür sorgen, dass $\hat{t}_1 \trianglelefteq x$ sowie $\hat{t}_2 \trianglelefteq x$ gelten. Falls diese Eigenschaft verletzt ist, können wir sie wiederherstellen, indem wir x mit $\hat{t}_1$ oder $\hat{t}_2$ vertauschen, je nachdem, welches der beiden Elemente größer ist.

Falls beispielsweise $\hat{t}_2 \trianglelefteq \hat{t}_1$ und $x \triangleleft \hat{t}_1$ gelten, tauschen wir x gegen $\hat{t}_1$, indem wir $t_1' := (t_{11}, t_{12}, x)$ und $t' := (t_1', t_2, \hat{t}_1)$ setzen. Da t_1' eine kleinere Wurzel als t_1 hat, kann die Halden-Eigenschaft in diesem neuen Teilbaum verletzt sein, so dass wir die Prozedur für ihn wiederholen müssen.

In Abbildung 3.28 ist der resultierende Algorithmus dargestellt. Die Funktion `sink` erhält das Array `x`, dessen Länge `n` und den Index `i` desjenigen Elements, in dem die Halden-Eigenschaft verletzt sein könnte. Wir gehen davon aus, dass die potentiell in `2*i+1` und `2*i+2` wurzelnden Teilbäume bereits Halden sind.

In Zeile 7 wird der Index `j` des linken Sohns berechnet. Die in Zeile 8 beginnende Schleife läuft so lange, wie `i` mindestens einen Sohn besitzt, so lange also die Möglichkeit besteht, dass die Halden-Eigenschaft verletzt ist. In Zeile 9 wird geprüft, ob auch ein

```
1  void
2  sink(int i, int n, int *x)
3  {
4    int xi;
5    int j;
6    xi = x[i];
7    j = 2*i+1;
8    while(j < n) {
9      if((j+1 < n) && (x[j] < x[j+1]))
10       j++;
11     if(xi < x[j]) {
12       x[i] = x[j]; i = j;
13       j = 2*i+1;
14     }
15     else break;
16   }
17   x[i] = xi;
18 }
```

Abbildung 3.28: Wiederherstellung der Halden-Eigenschaft in einem Teilbaum, in dem sie lediglich in der Wurzel verletzt sein darf.

rechter Sohn `j+1` existiert und ob das zugehörige Element größer als das des linken Sohns ist. In diesem Fall wechseln wir in Zeile 10 zu dem rechten Sohn.

Also wird in Zeile 11 das Wurzelelement mit dem Maximum der beiden Sohnelemente verglichen. Falls die Halden-Eigenschaft verletzt ist, werden `x[i]` und `x[j]` getauscht. Da nun die Halden-Eigenschaft in dem in `j` wurzelnden Teilbaum verletzt sein kann, wird `i` angepasst und die Schleife erneut durchlaufen.

Anderenfalls ist die Halden-Eigenschaft erfüllt und die Schleife kann beendet werden.

Um dafür zu sorgen, dass der *gesamte* Baum eine Halde ist, stellen wir in der bereits beschriebenen Weise die gewünschte Eigenschaft zunächst für die Teilbäume der Höhe 1 her, dann für die der Höhe 2, und so weiter, bis der gesamte Baum erreicht ist. Da sich ein Baum der Höhe h nach Definition nur aus Teilbäumen geringerer Höhe zusammensetzen kann, ist bei dieser Vorgehensweise sicher gestellt, dass die Teilbäume jeweils bereits Halden sind, so dass sich die Funktion `sink` verwenden lässt, um den gesamten Baum zu korrigieren. Da die Teilbäume eines größeren Baums immer höhere Indizes als die Wurzelelement aufweisen, stellt eine einfache herabzählende Schleife sicher, dass alle Teilbäume der Höhe nach geordnet durchlaufen werden.

Teilbäume der Höhe 1 sind immer Halden, also können wir unsere Schleife bei dem ersten Index beginnen lassen, für den mindestens ein Teilbaum existiert. Ein Index $m \in \mathbb{N}_0$ besitzt genau dann einen Teilbaum, wenn

$$2m + 1 < n \quad \Longleftrightarrow \quad 2m < n - 1 \quad \Longleftrightarrow \quad m < \frac{n-1}{2}$$

$$\iff \quad m < \begin{cases} (n-1)/2 = \lfloor n/2 \rfloor & \text{falls } n \text{ ungerade,} \\ n/2 - 1/2 = \lfloor n/2 \rfloor - 1/2 & \text{ansonsten} \end{cases}$$

gilt. Da m eine ganze Zahl ist, sind $m < \lfloor n/2 \rfloor - 1/2$ und $m < \lfloor n/2 \rfloor$ äquivalent, also genügt es, die Schleife bei $\lfloor n/2 \rfloor - 1$ beginnen zu lassen, so dass das folgende Programmfragment die Halden-Eigenschaft für den gesamten Baum sicherstellt:

```
1 for(i=n/2-1; i>=0; i--)
2   sink(i, n, x);
```

Wie üblich ist hier zu beachten, dass in der Programmiersprache C bei der Division ganzer Zahlen immer abgerundet wird, so dass die Schleife wie gewünscht bei $\lfloor n/2 \rfloor - 1$ beginnt.

Bemerkung 3.16 (Vorzeichenlose Schleifenvariable) *Falls wir für* `i` *eine vorzeichenlose Variable (beispielsweise* `unsigned int`*) verwenden, ist die Schleifenbedingung* `i>=0` *immer erfüllt und die Schleife arbeitet nicht korrekt. In diesem Fall können wir den Postfix-Dekrement-Operator gewinnbringend einsetzen: Der Ausdruck* `i-->0` *reduziert* `i` *um eins und prüft, ob der* alte *Wert der Variablen* `i` *größer als null war. Da* `i` *jetzt* vor *Betreten des Schleifenrumpfs reduziert wird, müssen wir die Schleife statt mit* $n/2 - 1$ *mit dem um eins höheren Wert* $n/2$ *beginnen lassen. Es ergibt sich die folgende sicherere Variante:*

```
1 for(i=n/2; i-->0; )
2   sink(i, n, x);
```

Nachdem wir eine Halde konstruiert haben, können wir das größte Element des Arrays an der Wurzel finden. In einem sortierten Array sollte dieses Element an der letzten Stelle stehen, also bietet es sich an, es mit dem aktuellen letzten Element zu tauschen, die Halden-Eigenschaft wieder herzustellen und dann die Prozedur für das um dieses Element verkürzte Array zu wiederholen. Durch das Verkürzen des Arrays bleibt bei unserer Numerierung die Halden-Eigenschaft erhalten, so dass kein zusätzlicher Aufwand anfällt.

Der resultierende Algorithmus ist in Abbildung 3.29 dargestellt, er trägt den Namen *Heapsort*. In den Zeilen 6 und 7 wird in der bereits beschriebenen Weise die Halden-Eigenschaft hergestellt, in den Zeilen 8 bis 12 wird jeweils das in der Wurzel der Halde stehende Element mit dem Element am Ende des Arrays getauscht, die Länge des Arrays um eins reduziert und mit `sink(0, n, x)` dafür gesorgt, dass die dabei möglicherweise verletzte Halden-Eigenschaft wieder hergestellt wird.

Natürlich wollen wir auch für dieses Sortierverfahren feststellen, wie hoch der Rechenaufwand im ungünstigsten Fall werden kann. Dazu untersuchen wir zunächst die Funktion `sink`, die für den Algorithmus von zentraler Bedeutung ist.

Lemma 3.17 (Rechenaufwand) *Falls i die Wurzel eines Baums der Höhe h bezeichnet, benötigt die Funktion* `sink` *höchstens* $8 + 18h$ *Operationen.*

```
1  void
2  heapsort(int n, int *x)
3  {
4    int z;
5    int i;
6    for(i=n/2-1; i>=0; i--)
7      sink(i, n, x);
8    while(n > 1) {
9      n--;
10     z = x[n]; x[n] = x[0]; x[0] = z;
11     sink(0, n, x);
12   }
13 }
```

Abbildung 3.29: Heapsort-Algorithmus.

Beweis. In Zeile 6 fallen 2 Operationen an, in Zeile 7 sind es 3 Operationen und in Zeile 17 weitere 2 Operationen. In den Zeilen der Schleife benötigen wir

Zeile	Operationen
8	1
9	7
10	1
11	2
12	4
13	3

Also benötigt jede Iteration höchstens 18 Operationen, und da bei jeder Iteration zu einem der beiden Teilbäume übergegangen wird, können höchstens h Iterationen erforderlich werden. Nach der letzten Iteration muss noch einmal in Zeile 7 die Schleifenbedingung geprüft werden, so dass insgesamt nicht mehr als $8 + 18h$ Operationen benötigt werden. ∎

Mit Hilfe dieses Lemmas können wir eine erste Abschätzung für den Rechenaufwand des Heapsort-Verfahrens gewinnen.

Satz 3.18 (Rechenaufwand) *Der Algorithmus* `heapsort` *benötigt nicht mehr als* $22n + 27n\lceil\log_2(n+1)\rceil$ *Operationen.*

Beweis. Wir haben bereits gesehen, dass wir ein Array der Länge n als einen fast vollständigen binären Baum der Höhe $h = \lceil\log_2(n+1)\rceil$ interpretieren können. Demzufolge hat jeder der höchstens $n/2$ Teilbäume, für die wir in Zeile 7 die Funktion `sink` aufrufen, höchstens die Höhe h, so dass nach Lemma 3.17 nicht mehr als $8 + 18h$ Operationen benötigt werden.

Die Schleife in den Zeilen 6 und 7 benötigt 3 Operationen für das Setzen der Variable i sowie in jeder Iteration je 2 Operationen, um die Schleifenbedingung zu prüfen und i herunter zu zählen. Schließlich muss nach der letzten Iteration noch einmal die Schleifenbedingung mit einer Operation geprüft werden.

In jeder Iteration wird `sink` für einen Teilbaum aufgerufen, dessen Höhe h nicht überschreiten kann, so dass nach Lemma 3.17 für die Zeilen 6 und 7 nicht mehr als

$$4 + \frac{n}{2}(2 + 8 + 18h) = 4 + 5n + 9nh \tag{3.13}$$

Operationen anfallen.

Die Schleife in den Zeilen 8 bis 12 wird $(n-1)$-mal durchlaufen. In jedem Durchlauf benötigt das Überprüfen der Schleifenbedingung eine Operation, das Herunterzählen der Variablen n eine weitere und das Tauschen von `x[n]` und `x[0]` insgesamt 7 Operationen, so dass wir inklusive des Aufrufs der Funktion `sink` mit nicht mehr als

$$(n-1)(1+1+7+8+18h) = (n-1)(17+18h) \leq 17n + 18nh - 17$$

Operationen auskommen.

Die Gesamtzahl der Rechenoperationen ist damit durch

$$4 + 5n + 9nh + 17n + 18nh - 17 = 22n + 27nh - 13 < 22n + 27n\lceil \log_2(n+1) \rceil$$

beschränkt. ∎

Die erste Phase des Algorithmus', also das Herstellen der Halden-Eigenschaft in den Zeilen 6 und 7, lässt sich noch etwas präziser abschätzen.

Bemerkung 3.19 (Verbesserte Abschätzung) *In dem vorangehenden Beweis sind wir in (3.13) davon ausgegangen, dass jeder Aufruf der Funktion* `sink` *für einen Teilbaum der Höhe höchstens h erfolgt. Das ist relativ pessimistisch: Lediglich der gesamte Baum weist die Höhe h auf, seine beiden Teilbäume nur noch eine Höhe von $h-1$, deren insgesamt vier Teilbäume nur noch eine Höhe von $h-2$. Insgesamt treten höchstens $2^{h-\ell}$ Teilbäume der Tiefe ℓ auf, so dass wir (3.13) durch*

$$4 + \frac{n}{2}8 + \sum_{\ell=0}^{h} 18\ell 2^{h-\ell} = 4 + 4n + 18\,2^h \sum_{\ell=1}^{h} \ell 2^{-\ell} < 4 + 2n + 36(n+1)\sum_{\ell=1}^{h} \ell 2^{-\ell}$$

ersetzen können. Dabei haben wir im letzten Schritt $h = \lceil \log_2(n+1) \rceil < \log_2(n+1) + 1$ ausgenutzt. Unsere Aufgabe ist es nun, die Summe

$$S(h) := \sum_{\ell=1}^{h} \ell 2^{-\ell}$$

abzuschätzen. Dazu betrachten wir

$$\frac{1}{2}S(h) = \frac{1}{2}\sum_{\ell=1}^{h} \ell 2^{-\ell} = \sum_{\ell=1}^{h} \ell 2^{-(\ell+1)} = \sum_{\ell=2}^{h+1} (\ell-1)2^{-\ell} = \sum_{\ell=1}^{h+1} (\ell-1)2^{-\ell}$$

und erhalten

$$\frac{1}{2}S(h) = \left(1 - \frac{1}{2}\right) S(h) = S(h) - \frac{1}{2}S(h) = \sum_{\ell=1}^{h} \ell 2^{-\ell} - \sum_{\ell=1}^{h+1} (\ell - 1)2^{-\ell}$$
$$= \sum_{\ell=1}^{h} (\ell - \ell + 1)2^{-\ell} - h2^{-(h+1)} = \sum_{\ell=1}^{h} 2^{-\ell} - h2^{-(h+1)} \leq \sum_{\ell=1}^{h} 2^{-\ell}.$$

Die geometrische Reihe können wir dank

$$\frac{1}{2}\sum_{\ell=1}^{h} 2^{-\ell} = \left(1 - \frac{1}{2}\right) \sum_{\ell=1}^{h} 2^{-\ell} = \sum_{\ell=1}^{h} 2^{-\ell} - \frac{1}{2}\sum_{\ell=1}^{h} 2^{-\ell} = \sum_{\ell=1}^{h} 2^{-\ell} - \sum_{\ell=1}^{h} 2^{-(\ell+1)}$$
$$= \sum_{\ell=1}^{h} 2^{-\ell} - \sum_{\ell=2}^{h+1} 2^{-\ell} = 2^{-1} - 2^{-(h+1)} = \frac{2^h - 1}{2^{h+1}}$$

durch

$$\sum_{\ell=1}^{h} 2^{-\ell} = 2\frac{2^h - 1}{2^{h+1}} = \frac{2^h - 1}{2^h} = 1 - 2^{-h} < 1$$

abschätzen, so dass wir insgesamt

$$S(h) \leq 2\sum_{\ell=1}^{h} 2^{-\ell} < 2$$

erhalten. Demnach können wir (3.13) durch

$$4 + 4n + 2\ 36(n + 1) = 4 + 4n + 72(n + 1) = 76 + 76n$$

ersetzen. Das Herstellen der Halden-Eigenschaft erfordert also lediglich $\Theta(n)$ *Operationen statt* $\mathcal{O}(n \log_2(n))$.

Theoretisch bietet der Heapsort-Algorithmus erhebliche Vorteile gegenüber den Mergesort- und Quicksort-Algorithmen, da er ohne größeren Hilfsspeicher auskommt und auch im ungünstigsten Fall nicht mehr als $\mathcal{O}(n \log_2(n))$ Operationen benötigt. In der Praxis ist die Reihenfolge der bei `heapsort` auftretenden Speicherzugriffe für moderne Prozessoren eher ungünstig, so dass insgesamt länger gerechnet wird: Während `mergesort` für ein Array mit $n = 10\,000\,000$ Elementen lediglich 1, 13 Sekunden benötigt, braucht `heapsort` mit 1, 79 Sekunden fast 60% mehr Zeit.

Prioritätswarteschlange. Mit Hilfe einer Halde lässt sich auch eine Variante der bereits bekannten Warteschlange konstruieren, bei der die einzelnen Elemente der Warteschlange um eine *Priorität* ergänzt werden und wir bei der Dequeue-Operation nicht das älteste Element erhalten, sondern das mit der höchsten Priorität.

Dazu verwalten wir die Elemente in einer durch ein Array dargestellte Halde, bei der die Ordnung durch die Prioritäten der Elemente festgelegt ist, so dass das Element mit

```
1  void
2  rise(int j, int *x)
3  {
4    int xj;
5    int i;
6    xj = x[j];
7    while(j > 0) {
8      i = (j-1)/2;
9      if(x[i] < xj) {
10       x[j] = x[i]; j = i;
11     }
12     else break;
13   }
14   x[j] = xj;
15 }
```

Abbildung 3.30: Wiederherstellung der Halden-Eigenschaft in einem Teilbaum, in dem sie in einem Teilbaum verletzt sein darf.

der höchsten Priorität an der Wurzel zu finden ist. Dieses Element geben wir bei der Dequeue-Operation zurück und ersetzen es, wie schon im Heapsort-Algorithmus, durch das letzte Element des Arrays.

Bei der Enqueue-Operation wird ein weiteres Element am Ende des Arrays hinzugefügt, das die Halden-Eigenschaft stören kann, falls es größer als die Wurzel des übergeordneten Baums ist. Dieses Problem können wir lösen, indem wir das Element aufsteigen lassen, es also mit der Wurzel des nächsthöheren Baums vertauschen. Dadurch kann die Halden-Eigenschaft auf der nächsten Stufe verloren gehen, so dass wir die Prozedur wiederholen müssen.

Da bei dieser Vorgehensweise jeweils die Wurzeln der Teilbäume nur vergrößert werden, kann die Halden-Eigenschaft nur im jeweils übergeordneten Baum verletzt sein, so dass es wieder genügt, auf jeder Stufe des Baums eine Korrektur vorzunehmen. Der resultierende Algorithmus ist in Abbildung 3.30 dargestellt. Er berechnet zunächst durch $i = \lfloor (j-1)/2 \rfloor$ den Index des übergeordneten Baums und prüft, ob die Halden-Eigenschaft durch das neue Element verletzt wurde. In diesem Fall tauschen die alte Wurzel und das neue Element die Plätze und der Algorithmus prüft, ob das neue Element an seiner neuen Position immer noch die Halden-Eigenschaft verletzt. Enqueue- und Dequeue-Funktionen benötigen so höchstens $\mathcal{O}(\log_2(n+1))$ Operationen.

4 Graphen

In vielen Anwendungen treten mathematische Objekte auf, die in Beziehung zueinander stehen: Die Orte auf einer Landkarte sind durch Straßen verbunden, der Zustand eines Computers wird durch einen Befehl in einen anderen Zustand überführt, Daten werden von einem Knoten eines Rechnernetzes durch eine Leitung zu einem anderen übertragen.

Derartige Zusammenhänge lassen sich mathematisch einheitlich durch *Graphen* beschreiben und algorithmisch analysieren.

4.1 Definition eines Graphen

Definition 4.1 (Graph) *Sei V eine endliche Menge und $E \subseteq V \times V$. Das Paar $G = (V, E)$ nennen wir dann einen* (gerichteten) Graphen *(engl.* (directed) graph *oder* digraph*).*

Die Menge V bezeichnen wir als die Menge der Knoten *(engl.* vertices*) des Graphen, sie wird gelegentlich als $V(G)$ notiert. Die Menge E bezeichnen wir als die Menge der* Kanten *(engl.* edges*) des Graphen, für sie ist die Notation $E(G)$ üblich.*

Ein Graph wird häufig analog zu einer Landkarte interpretiert: Die Knoten entsprechen Orten auf der Landkarte, die Kanten den Straßenstücke, die diese Orte miteinander verbinden. Da aus $(u, v) \in E$ nicht unbedingt $(v, u) \in E$ folgt, können dabei durchaus Einbahnstraßen auftreten.

Falls dieser Sonderfall nicht auftritt, sprechen wir von einem *ungerichteten* Graphen.

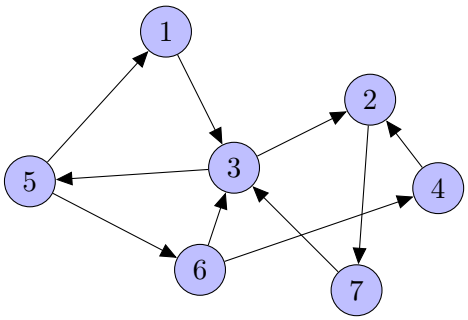
Definition 4.2 (Ungerichteter Graph) *Sei $G = (V, E)$ ein Graph. Wir nennen G einen* ungerichteten *Graphen, falls*

$$(u, v) \in E \iff (v, u) \in E \qquad \text{für alle } u, v \in V$$

gilt, falls also zu jeder Kante auch die entgegengerichtete Kante im Graphen vorkommt.

Mit Graphen lassen sich viele verschiedene Aufgabenstellungen beschreiben:

- Routenfindung: Beispielsweise in Navigationssystemen für Fahrzeuge: Die einzelnen Orte sind die Knoten des Graphen, die Straßenstücke zwischen ihnen die Kanten. Gesucht ist in diesem Kontext beispielsweise die kürzeste Verbindung zwischen zwei Orten.
- Verkehrsplanung: Knoten und Kanten werden wieder als Orte und Straßen interpretiert, aber diesmal stellt sich die Frage, wie möglichst viele Verkehrsteilnehmer *gleichzeitig* reisen können, ohne die Straßen zu überlasten.



$$V = \{1, 2, 3, 4, 5, 6, 7\},$$
$$E = \{(1, 3), (2, 7), (3, 2), (3, 5), (4, 2), (5, 1), (5, 6), (6, 3), (6, 4), (7, 3)\}$$

Abbildung 4.1: Gerichteter Graph

- Simulation elektrischer Schaltungen: Die einzelnen Bauelemente sind die Kanten des Graphen, die in Knoten miteinander in Kontakt stehen. Suchen könnte man in diesem Fall die Spannungen, die in den Knoten vorliegen, oder die Ströme, die durch die Bauelemente fließen.
- Transportplanung: Die Knoten des Graphen sind Hersteller, Zwischenlager und Verbraucher, die Kanten sind Transportwege. Von Interesse könnte in diesem Fall sein, wie man möglichst günstig die Verbraucher versorgt.
- Ressourcenplanung: Es gibt zwei Sorten von Knoten: Ressourcen, beispielsweise Maschinen, und Aufgaben, beispielsweise Produktionsschritte. Kanten geben an, welche Aufgaben welche Ressourcen benötigen. Gesucht ist eine Zuordnung, bei der möglichst viele Aufgaben die nötigen Ressourcen erhalten.
- Strategiespiele: Bei manchen Spielen lassen sich die Konfiguration des Spielfelds als Knoten und die Spielzüge als Kanten interpretieren, durch die die einzelnen Konfigurationen ineinander überführt werden. Gefragt wäre eine Strategie, mit der man eine Konfiguration erreicht, mit der man das Spiel gewinnt.
- Aufgabenplanung: Größere Projekte zerfallen oft in kleinere Aufgaben, die voneinander abhängig sind. Wenn man die Aufgaben als Knoten und ihre Abhängigkeiten als Kanten beschreibt, kann man nach einer Reihenfolge suchen, in der für jede Aufgabe sichergestellt ist, dass die von ihrer Fertigstellung abhängigen Aufgaben erst nach ihr bearbeitet werden.

Die meisten unserer Algorithmen bewegen sich bei der Suche nach der Lösung entlang der Kanten von einem Knoten zum nächsten, so dass es sinnvoll erscheint, eine Datenstruktur zu verwenden, bei der sich diese Operation effizient durchführen lässt.

Die Knoten eines Graphen stellen wir der Einfachheit halber durch ganze Zahlen dar. Zu jedem Knoten führen wir eine einfach verkettete Liste mit Elementen des Typs `edge`, die die von ihm ausgehenden Kanten beschreibt. Zeiger auf die Köpfe dieser Listen speichern wir in einem Array `edgelist`, die Anzahl der Knoten in einem Feld `vertices`.

```
typedef struct _edge edge;
struct _edge {
  int to;
  edge *next;
};
edge *
new_edge(int to, edge *next)
{
  edge *e;
  e = (edge *) malloc(sizeof(edge));
  e->to = to; e->next = next;
  return e;
}
typedef struct _graph graph;
struct _graph {
  edge **edgelist;
  int vertices;
};
graph *
new_graph(int vertices)
{
  graph *g;
  int i;
  g = (graph *) malloc(sizeof(graph));
  g->edgelist = (edge **) malloc(sizeof(edge *) * vertices);
  g->vertices = vertices;
  for(i=0; i<vertices; i++) g->edgelist[i] = 0;
  return g;
}
```

Abbildung 4.2: Darstellung eines Graphen.

Beides fassen wir in einem neuen Typ `graph` zusammen, der uns als Grundlage unserer Algorithmen dienen wird.

Der Typ `edge` enthält neben dem Index `to` des Endknoten der durch ihn dargestellten Kante auch den für einfach verkettete Listen üblichen `next`-Zeiger.

Beispielsweise können wir bei dieser Datenstruktur alle von einem Knoten v aus über eine Kante $(v, w) \in E$ erreichbaren Knoten w mit der folgenden Schleife durchlaufen:

```
for(e=edgelist[v]; e; e=e->next) {
  w = e->to;
  printf("%d -> %d\n", v, w);
}
```

Festzustellen, ob v und w durch eine Kante verbunden sind, erfordert bei dieser Datenstruktur (im Gegensatz zu anderen) eine zusätzliche Schleife:

```
1 for(e=edgelist[v]; e && (e->to != w); e=e->next)
2   ;
3 if(e) printf("%d reachable from %d\n", w, v);
```

Glücklicherweise lassen sich viele Algorithmen so formulieren, dass es genügt, die Nachbarn eines Knoten durchlaufen zu können, und für diese Operation eignet sich unsere Datenstruktur sehr gut.

4.2 Breitensuche

Wenn wir uns einen Graph als Landkarte vorstellen, ist eine naheliegende Frage, wie wir auf dieser Karte von einem Ort zu einem anderen gelangen können, indem wir den eingezeichneten Straßen folgen.

Definition 4.3 (Kantenzug) *Sei $G = (V, E)$ ein Graph. Ein Tupel $(v_0, v_1, \ldots, v_n)$ von Knoten nennen wir einen* Kantenzug *(engl.* walk*), falls*

$$(v_{i-1}, v_i) \in E \qquad \text{für alle } i \in \{1, \ldots, n\}$$

gilt. In diesem Fall nennen wir n die Länge *des Kantenzugs, v_0 nennen wir den* Anfangsknoten *und v_n den* Endknoten *des Kantenzugs. Wir bezeichnen $(v_0, v_1, \ldots, v_n)$ auch als* Kantenzug von v_0 zu v_n.

Für alle Knoten $v \in V$ bezeichnen wir mit (v) den Kantenzug der Länge null von v zu sich selbst.

Häufig sind wir daran interessiert, Kantenzüge möglichst geringer Länge zu konstruieren. Bei derartigen Kantenzügen sollte kein Knoten doppelt vorkommen, denn sonst könnte man ihre Länge einfach reduzieren:

Lemma 4.4 (Kantenzug kürzen) *Sei $G = (V, E)$ ein Graph und $(v_0, v_1, \ldots, v_n)$ ein Kantenzug in G. Falls $v_i = v_j$ mit $i < j$ gelten sollte, existiert ein Kantenzug der Länge $n - (j - i) < n$.*

Beweis. Gelte $v_i = v_j$ für $i < j$. Dann ist $(v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_n)$ wegen $(v_i, v_{j+1}) = (v_j, v_{j+1}) \in E$ wieder ein Kantenzug, in dem allerdings die $j - i$ Elemente $v_{i+1}, \ldots, v_j$ gegenüber dem ursprünglichen fehlen. Also beträgt seine Länge lediglich $n - (j - i)$, und aus $i < j$ folgt $j - i > 0$, also $n - (j - i) < n$. ∎

Die besondere Bedeutung von Kantenzügen ohne doppelte Knoten würdigen wir durch eine separate Bezeichnung.

Definition 4.5 (Pfad) *Sei $G = (V, E)$ ein Graph. Einen Kantenzug $(v_0, \ldots, v_n)$ nennen wir einen* Pfad *(engl.* path*), falls*

$$v_i \neq v_j \qquad \text{für alle } i, j \in \{0, \ldots, n\} \text{ mit } i \neq j$$

gilt, falls also kein Knoten in dem Kantenzug doppelt vorkommt.

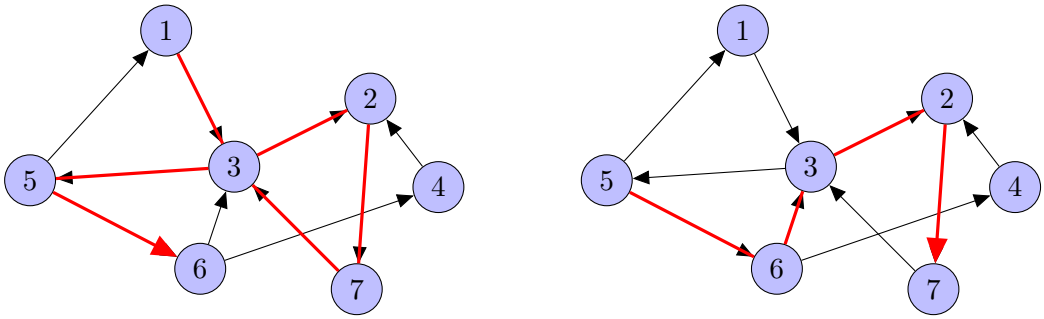


Abbildung 4.3: Kantenzug $(1, 3, 2, 7, 3, 5, 6)$ (links) und Pfad $(5, 6, 3, 2, 7)$ (rechts) in einem Graphen

Von besonderem Interesse ist natürlich die Frage, ob es überhaupt möglich ist, von jedem Knoten eines Graphen zu jedem anderen zu gelangen, indem man einem Pfad oder Kantenzug folgt.

Definition 4.6 (Zusammenhängender Graph) *Sei $G = (V, E)$ ein Graph. Wir nennen G* (stark) zusammenhängend *(engl.* (strongly) connected*), falls für alle $v, w \in V$ ein Kantenzug von v zu w existiert.*

Falls ein Graph nicht zusammenhängend ist, können wir ihn immerhin in zusammenhängende Graphen zerlegen. Das ist nützlich, da viele graphentheoretische Algorithmen nur für zusammenhängende Graphen korrekt arbeiten und häufig bei nicht zusammenhängenden Graphen nur einen Teil des Graphen behandeln.

Definition 4.7 (Teilgraph) *Seien $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ Graphen. Falls $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$ gelten, bezeichnen wir G_1 als* Teilgraphen *von G_2.*

Falls $E_1 = E_2 \cap (V_1 \times V_1)$ gilt, nennen wir G_1 den durch V_1 induzierten Teilgraphen *von G_2 und schreiben ihn $G_1 = G_2[V_1]$.*

Definition 4.8 (Zusammenhangskomponente) *Sei $G = (V, E)$ ein Graph und $U \subseteq V$. Falls der durch U induzierte Teilgraph $G[U]$ zusammenhängend ist und für jede Menge $W \subseteq V$ mit $U \subsetneq W$ der induzierte Teilgraph $G[W]$ nicht zusammenhängend ist, nennen wir U eine* Zusammenhangskomponente *von G.*

Für zwei beliebige Knoten in einer Zusammenhangskomponente können wir also einen Kantenzug finden, der sie verbindet.

Eine typische Aufgabe besteht darin, einen möglichst kurzen Pfad von einem Knoten $v \in V$ zu einem anderen Knoten $w \in V$ zu finden. Wie bereits diskutiert wollen wir dabei möglichst darauf verzichten, uns im Kreis zu bewegen.

Definition 4.9 (Kreis) *Sei $G = (V, E)$ ein Graph. Einen Pfad $(v_0, \ldots, v_n)$ nennen wir* Kreis *(engl.* cycle*), falls $n \geq 2$ und $(v_n, v_0) \in E$ gelten.*

Falls in einem Graphen keine Kreise existieren, nennen wir ihn kreisfrei *(engl.* acyclic*).*

Ein einfacher Ansatz für die Lösung der Aufgabe, einen möglichst kurzen Pfad von einem Knoten v zu einem Knoten w zu finden, besteht darin, von dem Knoten v ausgehend zunächst alle Knoten zu suchen, die sich mit einem Pfad der Länge 1 erreichen lassen, dann alle, für die Pfad der Länge 2 genügt, und so weiter, bis zum ersten Mal w besucht wird. Da immer *alle* Knoten einer gewissen Entfernung zu v gleichzeitig behandelt werden, ist damit der kürzeste Pfad gefunden.

Präzise beschreiben wir die Vorgehensweise durch die Mengen

$$D_0 := \{v\},$$
$$D_\ell := \{u \in V \ : \ (\forall k < \ell : u \notin D_k) \wedge (\exists z \in D_{\ell-1} : (z,u) \in E)\} \qquad \text{für alle } \ell \in \mathbb{N}.$$

Wir können mathematisch nachweisen, dass wir mit diesen Mengen in der Tat kürzeste Pfade konstruieren können.

Satz 4.10 (Kürzester Pfad) *Sei $\ell \in \mathbb{N}_0$. Es gilt $w \in D_\ell$ genau dann, wenn ein Pfad der Länge ℓ von v nach w existiert, aber kein kürzerer.*

Beweis. Nach Lemma 4.4 muss ein Kantenzug minimaler Länge von v nach w bereits ein Pfad sein, also genügt es, per Induktion über $\ell \in \mathbb{N}_0$ zu beweisen, dass für alle $w \in V$ genau dann ein *Kantenzug* minimaler Länge ℓ existiert, falls $w \in D_\ell$ gilt.

Induktionsanfang. Sei $\ell = 0$. Der einzige Pfad der Länge ℓ mit Anfangsknoten v ist (v), also ist v auch der einzige Knoten, der mit einem Kantenzug dieser Länge erreicht werden kann. Kürzere Kantenzüge gibt es offenbar nicht.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gegeben, dass für alle $k \in \{0, \ldots, \ell\}$ und alle Knoten $w \in V$ genau dann ein Kantenzug minimaler Länge k existiert, falls $w \in D_k$ gilt.

Induktionsschritt. Sei zunächst $w \in D_{\ell+1}$. Da $\ell + 1 > 0$ gilt, existiert nach Definition ein $z \in D_\ell$ mit $(z,w) \in E$. Nach Induktionsvoraussetzung muss dann ein Kantenzug $(v_0, \ldots, v_\ell)$ der Länge ℓ von $v = v_0$ zu $z = v_\ell$ existieren. Also ist $(v_0, \ldots, v_\ell, w)$ ein Kantenzug der Länge $\ell + 1$ von $v = v_0$ zu w.

Für alle $k < \ell + 1$ gilt nach Definition der Menge $D_{\ell+1}$ insbesondere $w \notin D_k$, also existiert nach Induktionsvoraussetzung kein Kantenzug der Länge k von v zu w.

Sei nun $w \in V$ so gegeben, dass ein Kantenzug $(v_0, \ldots, v_{\ell+1})$ von v zu w existiert, aber kein kürzerer. Dann ist $(v_0, \ldots, v_\ell)$ ein Kantenzug von $v = v_0$ zu $z := v_\ell$. Es ist auch ein Kantenzug minimaler Länge, denn aus jedem Kantenzug $(v'_0, \ldots, v'_k)$ von v zu z können wir wegen $(z,w) = (v_\ell, v_{\ell+1}) \in E$ einen Kantenzug $(v'_0, \ldots, v'_k, w)$ von v zu w konstruieren, der nach Voraussetzung mindestens die Länge $\ell + 1$ aufweisen muss.

Nach Induktionsvoraussetzung muss dann $z \in D_\ell$ gelten. Für alle $k < \ell + 1$ kann nach Voraussetzung kein Kantenzug der Länge k von v zu w existieren, also gilt wieder mit der Induktionsvoraussetzung $w \notin D_k$. Aus $z \in D_\ell$, $w \notin D_k$ für alle $k < \ell + 1$ und $(z,w) \in E$ folgt $w \in D_{\ell+1}$. ∎

Wir könnten also den kürzesten Pfad zwischen zwei Knoten v und w berechnen, indem wir der Reihe nach $D_0, D_1, D_2, \ldots$ konstruieren, bis w in einer dieser Mengen enthalten

ist. Die Verwaltung dieser Mengen wird allerdings in der Regel einen relativ hohen Aufwand nach sich ziehen.

Meistens ist die konkrete Pfadlänge ℓ gar nicht von Interesse, wir suchen lediglich den kürzesten Pfad. Das hat zur Folge, dass wir lediglich dafür zu sorgen haben, dass unser Algorithmus zuerst alle Knoten der Menge D_0 durchläuft, dann alle der Menge D_1, dann alle der Menge D_2, und so weiter, bis w gefunden ist. Diese Aufgabe lässt sich elegant mit den in Abschnitt 3.3 eingeführten Warteschlangen lösen: Die Warteschlange nimmt diejenigen Knoten auf, die noch zu prüfen sind. Zu Beginn des Algorithmus enthält sie lediglich v, dann werden für jeden zu prüfenden Knoten alle Knoten zu der Warteschlange hinzugefügt, die noch nicht geprüft wurden. Durch die Struktur der Warteschlange ist dann sicher gestellt, dass die Elemente der Mengen $D_0, D_1, D_2, \ldots$ in der richtigen Reihenfolge durchlaufen werden, obwohl die Mengen selbst nicht mehr explizit auftreten.

```
1  void
2  bfs(const graph *g, int src, int *pred)
3  {
4    queue *qu;
5    int *visited;
6    edge *e;
7    int v, w;
8    qu = new_queue();
9    visited = (int *) malloc(sizeof(int) * g->vertices);
10   for(v=0; v<g->vertices; v++) visited[v] = 0;
11   for(v=0; v<g->vertices; v++) pred[v] = -1;
12   enqueue(qu, src); visited[src] = 1;
13   while(!isempty(qu)) {
14     v = dequeue(qu);
15     for(e=g->edgelist[v]; e; e=e->next) {
16       w = e->to;
17       if(visited[w] == 0) {
18         enqueue(qu, w); visited[w] = 1;
19         pred[w] = v;
20       }
21     }
22     visited[v] = 2;
23   }
24   free(visited); del_queue(qu);
25 }
```

Abbildung 4.4: Breitensuche in einem Graphen.

Der resultierende Algorithmus wird als *Breitensuche* (engl. *breadth-first search* oder kurz *BFS*) bezeichnet. In Abbildung 4.4 ist eine mögliche Implementierung dargestellt,

die die in Abbildung 3.9 beschriebenen Funktionen `enqueue`, `dequeue` und `isempty` verwendet, um Knoten der Warteschlange hinzu zu fügen und zu entnehmen.

Der Algorithmus verwendet ein Hilfsarray `visited`, um festzuhalten, welche Knoten bereits besucht wurden. Jedem Knoten ist eine Zahl zugeordnet, die angibt, wie weit seine Verarbeitung vorangeschritten ist:

- Der Wert 0 bedeutet, dass der Knoten noch nicht entdeckt wurde,
- der Wert 1, dass er entdeckt wurde, aber noch nicht besucht,
- der Wert 2, dass er besucht wurde und die von ihm aus erreichbaren Knoten in die Warteschlange eingetragen wurden.

Wenn in der inneren Schleife der Zeilen 15 bis 21 alle von dem Knoten `i` ausgehenden Kanten verarbeitet werden, wird jeweils geprüft, ob der Endknoten `j` bereits als besucht markiert wurde, um zu verhindern, dass wir uns endlos in einem Kreis durch den Graphen bewegen.

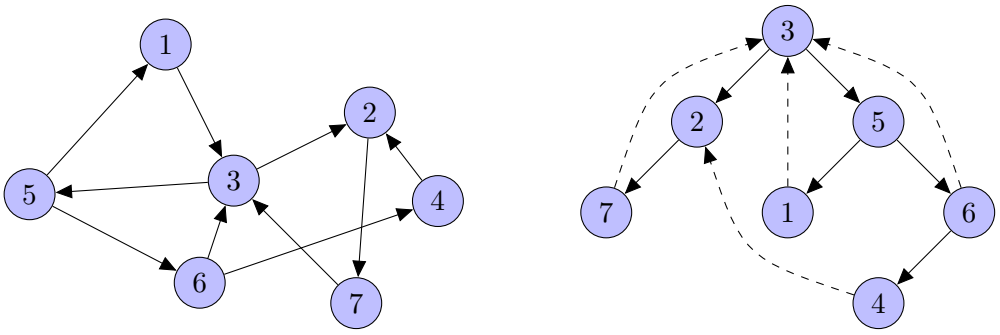


Abbildung 4.5: Breitensuche in Baumdarstellung: Die Knoten wurden rechts so umsortiert, dass sich der kürzeste Pfad von dem Anfangsknoten 3 zu jedem Knoten unmittelbar ablesen lässt, nicht verfolgte Kanten sind gestrichelt.

Unsere Implementierung gibt nicht einfach den kürzesten Pfad von dem als `src` übergebenen Anfangsknoten zu einem gegebenen Endknoten `trg` zurück, sondern füllt stattdessen ein Array `pred` mit den Nummern der Knoten, von denen aus ein Knoten besucht wurde. Aus dieser Information lässt sich einfach der kürzeste Pfad zwischen `src` und einem *beliebigen* Knoten rekonstruieren: Der kürzeste Pfad endet in `trg`. `trg` wurde von dem Knoten `pred[trg]` aus besucht, also muss `pred[trg]` der Vorgänger des Knotens `trg` im kürzesten Pfad sein. Entsprechend muss `pred[pred[trg]]` der Vorgänger des Vorgängers sein, so dass wir durch die folgende Schleife den Pfad rekonstruieren können:

```
1  w = trg;
2  if(pred[w] < 0)
3    printf("No path to %d\n", w);
```

```
4   else
5     while(w != src) {
6       printf("%d reached from %d\n", w, pred[w]);
7       w = pred[w];
8     }
```

Da unser Algorithmus die Elemente des Arrays `pred` in Zeile 11 mit `-1` füllt, bevor die eigentliche Breitensuche beginnt und diese Elemente mit den Vorgängerinformationen füllt, bedeutet `pred[w] < 0` gerade, dass der Knoten `i` nie besucht wurde. Nach Lemma 4.10 kann das allerdings nur passieren, falls er von dem Anfangsknoten `src` aus nicht erreichbar ist, also der Graph nicht zusammenhängend ist.

Mit Hilfe der Breitensuche können wir weitere Informationen über den Graphen gewinnen: Falls beispielsweise in Zeile 16 ein Knoten `w` auftritt, der schon besucht wurde, könnten wir einen Kreis in dem Graphen gefunden haben. Falls `pred[v] != w` gilt, ist `w` nicht der unmittelbare Vorgänger von `v`, so dass tatsächlich ein Kreis vorliegt.

Nach der Ausführung der Breitensuche sind in dem Array `visited` die Einträge der Knoten gesetzt, die in derselben Zusammenhangskomponente wie der Anfangsknoten `src` liegen. Sollte `visited[v] == 0` für ein `v` gelten, können wir eine weitere Breitensuche mit Anfangsknoten `v` durchführen, um dessen Zusammenhangskomponente zu konstruieren. In dieser Weise können wir fortfahren, bis alle Zusammenhangskomponenten konstruiert und markiert sind.

Bemerkung 4.11 (Rechenaufwand) *Durch die Verwendung der Markierungen in* `visited` *ist sicher gestellt, dass jeder Knoten höchstens einmal in die Warteschlange gelangt, also kann die äußere Schleife in den Zeilen 13 bis 23 höchstens $|V|$ Iterationen ausführen, wobei $|V|$ die Mächtigkeit der Knotenmenge V bezeichnet.*

Die innere Schleife in den Zeilen 15 bis 21 durchläuft alle von dem Knoten `i` *ausgehenden Kanten, und da jeder Knoten nur höchstens einmal in der Warteschlange vorkommt, wird insgesamt jede Kante nur höchstens einmal behandelt.*

Da für jede Iteration der inneren Schleife lediglich $\mathcal{O}(1)$ Operationen benötigt werden, genügen für die vollständige Breitensuche nach Lemma 2.20 insgesamt $\mathcal{O}(|V| + |E|)$ Operationen.

4.3 Tiefensuche

Als nächstes Beispiel beschäftigen wir uns mit der Frage, wie sich einzelne voneinander abhängende Aufgaben in eine Reihenfolge bringen lassen, in der eine einzelne Aufgabe erst dann in Angriff genommen wird, wenn alle, von denen sie abhängt, bereits gelöst wurden.

Dazu konstruieren wir einen Graphen, in dem die einzelnen Aufgaben durch die Knoten repräsentiert sind und eine Kante von einem Knoten v zu einem Knoten w existiert, falls die Aufgabe w vor der Aufgabe v gelöst werden muss.

Mathematisch lässt sich die Aufgabe wie folgt formulieren:

Gegeben seien ein Graph $G = (V, E)$ und ein Knoten $v \in V$. Gesucht ist eine Folge $v_1, \ldots, v_n \in V$ derart, dass $v = v_n$ gilt und

$$(v_i, w) \in E \Rightarrow \exists j < i : w = v_j \qquad \text{für alle } w \in V,\ i \in \{1, \ldots, n\}.$$

Falls also die Aufgabe w vor der Aufgabe v_i gelöst werden muss, ist w in der Folge enthalten und vor v_i einsortiert. Diese Aufgabe nennt man *topologisches Sortieren*.

Als einfaches Beispiel können wir die Reihenfolge betrachten, in der man sich verschiedene Kleidungsstücke anziehen kann:

- Die Schuhe kommen nach den Socken.
- Die Schuhe kommen nach der Hose.
- Die Hose kommt nach der Unterhose.
- Der Pullover kommt nach dem Hemd.
- Fertig angezogen sind wir, wenn wir alle Kleidungsstücke tragen.

Der korrespondierende Graph (mit passenden Abkürzungen für die Kleidungsstücke) ist in Abbildung 4.6 dargestellt.

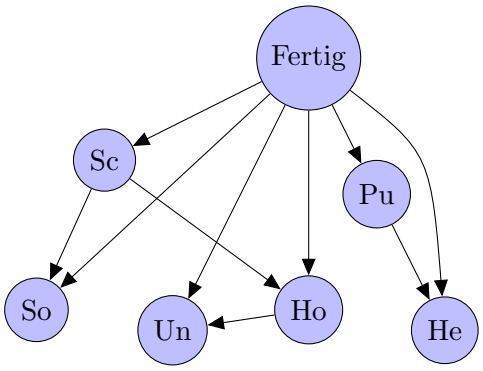


Abbildung 4.6: Darstellung der Abhängigkeiten zwischen den einzelnen Aufgaben durch einen Graphen.

Eine Lösung können wir systematisch konstruieren, indem wir zuerst feststellen, dass wir den Pullover oder die Schuhe als letztes anziehen können. Vor dem Pullover muss das Hemd kommen, vor dem Hemd das Unterhemd. Vor den Schuhen müssen die Socken und die Hose angezogen werden, vor der Hose die Unterhose. Indem wir die Reihenfolge umkehren, erhalten wir „Unterhose, Hose, Socken, Schuhe, Unterhemd, Hemd, Pullover“.

Falls eine Lösung existiert, können wir sie mit einem relativ einfachen Algorithmus konstruieren: Den „Endknoten“ $v \in V$ setzen wir an das Ende der Folge, dann konstruieren wir rekursiv für alle Knoten $w \in V$ mit $(v, w) \in E$ Folgen, die wir dann zu der Gesamtfolge zusammensetzen.

```
1  void
2  visit(const graph *g, int v, int *visited)
3  {
4    edge *e;
5    int w;
6    visited[v] = 2;
7    for(e=g->edgelist[v]; e ; e=e->next) {
8      w = e->to;
9      if(visited[w] == 0)
10       visit(g, w, visited);
11   }
12   visited[v] = 3;
13 }
14 void
15 dfs(const graph *g, int src)
16 {
17   int *visited;
18   int v;
19   visited = (int *) malloc(sizeof(int) * g->vertices);
20   for(v=0; v<g->vertices; v++) visited[v] = 0;
21   visit(g, src, visited);
22   free(visited);
23 }
```

Abbildung 4.7: Rekursive Tiefensuche in einem Graphen.

Der resultierende Algorithmus trägt den Namen *Tiefensuche* (engl. *depth-first search* oder kurz *DFS*), eine Implementierung ist in Abbildung 4.7 dargestellt. Das Array `visited` dient wie zuvor dem Zweck, zu vermeiden, dass wir Knoten doppelt besuchen. Wenn wir in Zeile 12 ein `printf("%d\n", v)` einfügen, gibt der Algorithmus die gewünschte Folge aus. Das Array `visited` bietet uns auch die Möglichkeit, zu erkennen, ob der Graph Kreise aufweist: Falls in Zeile 9 `visited[w] == 2` gilt, existiert ein Pfad von `w` nach `v` und wegen Zeile 8 auch eine Kante von `v` zu `w`. Für das Beispiel des topologischen Sortierens würde das bedeuten, dass es keine Folge mit den gewünschten Eigenschaften gibt.

Nicht-rekursive Implementierung. Da jeder rekursive Funktionsaufruf einen gewissen Verwaltungsaufwand bedeutet, kann es sinnvoll sein, die rekursive Struktur der Tiefensuche mit Hilfe der in Abschnitt 3.2 eingeführten Kellerspeicher nachzubilden: Eine Implementierung ist in Abbildung 4.8 dargestellt. Die Werte des Arrays `visited` beschreiben wieder die verschiedenen Phasen, die ein Knoten durchläuft:

- Der Wert 0 bedeutet, dass der Knoten noch nicht entdeckt wurde,

- der Wert 1, dass er entdeckt, aber noch nicht besucht wurde,
- der Wert 2, dass er besucht wurde, aber noch nicht alle „Nachbarknoten“, also die von ihm aus erreichbaren Knoten,
- der Wert 3 schließlich, dass er und alle Nachbarknoten besucht wurden, wir mit diesem Knoten also fertig sind.

Wir untersuchen jeweils das jüngste Element `v` im Kellerspeicher und prüfen, ob wir es zum ersten Mal besuchen. In diesem Fall gilt `visited[v] < 2` und wir fügen alle Knoten `w`, die zu denen eine Kante von `v` führt, in den Keller ein. Anderenfalls gilt `visited[v] == 2` und wir entfernen `v` aus dem Keller und markieren es durch `visited[v] = 3` als vollständig verarbeitet.

Sie verwendet die in Abbildung 3.7 zu findende Implementierung eines Kellerspeichers: Die Funktionen `push` und `pop` fügen ein Element dem Keller hinzu oder entnehmen es, die Funktion `isempty` prüft, ob noch Elemente vorhanden sind, und die Funktion `peek` gibt das jüngste Element zurück, ohne es aus dem Keller zu entfernen. Letztere Funktion ist für unsere Zwecke nützlich, weil jedes Element im Keller zweimal bearbeitet werden muss und es ineffizient wäre, es zwischendurch aus dem Keller zu entfernen.

Bemerkung 4.12 (Knoten mehrfach im Keller) *Bei dieser Implementierung ist zu beachten, dass ein Knoten w mehrfach im Kellerspeicher auftreten kann, falls Knoten z_1 und z_2 auf dem Pfad vom Ausgangsknoten zu ihm besucht werden, die* beide *eine Kante zu w besitzen.*

Trotzdem sind nie mehr als $\#E$ Elemente im Kellerspeicher: Sobald alle ausgehenden Kanten eines Knotens v verarbeitet wurden, wird `visited[v]` *auf* 2 *gesetzt, so dass diese Kanten nie wieder berücksichtigt werden. Also wird jede Kante höchstens einmal verarbeitet, also treten auch höchstens $\#E$ Aufrufe der Funktion* `push` *auf.*

Ankunfts- und Abschiedszeiten. Die Tiefensuche protokolliert ihren Fortschritt so, dass uns anschließend nützliche Informationen über die Struktur des Graphen zur Verfügung stehen: Wir halten fest, in welchem Schritt der Suche ein Knoten das erste und das letzte Mal besucht wurde. Dazu führen wir einen Zähler `t` mit, der die „Zeit“ seit dem Anfang der Suche beschreibt, sowie zwei Arrays `d` und `f`. Das Array `d` beschreibt die „Ankunftszeit“ (der Variablenname ist durch das Englische Wort *discovered* motiviert), das Array `f` die „Abschiedszeit“ (entsprechend durch *finished* motiviert).

Wenn ein Knoten `v` zum ersten Mal besucht wird, zählen wir `t` hoch und tragen den neuen Wert in `d[v]` ein. Wenn der Knoten zum letzten Mal besucht wird, zählen wir `t` wieder hoch und tragen das Ergebnis in `f[v]` ein.

Satz 4.13 (Ankunfts- und Abschiedszeiten) *Seien $(d_v)_{v\in V}$ und $(f_v)_{v\in V}$ die im Rahmen der Tiefensuche berechneten Ankunfts- und Abschiedszeiten.*

Falls für zwei Knoten $v, w \in V$ die Ungleichung

$$d_v \leq d_w < f_v \tag{4.1}$$

```
1  void
2  dfs(const graph *g, int src, int *d, int *f)
3  {
4    stack *st;
5    int *visited;
6    edge *e;
7    int v, w;
8    int t;
9    st = new_stack();
10   visited = (int *) malloc(sizeof(int) * g->vertices);
11   for(v=0; v<g->vertices; v++) visited[v] = 0;
12   for(v=0; v<g->vertices; v++) d[v] = 0;
13   t = 0;
14   push(st, src); visited[src] = 1; t++; d[src] = t;
15   while(!isempty(st)) {
16     v = peek(st);
17     if(visited[v] < 2) {
18       visited[v] = 2; t++; d[v] = t;
19       for(e=g->edgelist[v]; e; e=e->next) {
20         w = e->to;
21         if(visited[w] < 2) {
22           push(st, w); visited[w] = 1;
23         }
24       }
25     }
26     else if(visited[v] == 2) {
27       pop(st); visited[v] = 3; t++; f[v] = t;
28     }
29     else pop(st);
30   }
31   free(visited); del_stack(st);
32 }
```

Abbildung 4.8: Tiefensuche in einem Graphen mit Ankunfts- und Abschiedszeiten.

gilt, hat die Tiefensuche einen Pfad von v zu w verfolgt.

Die umgekehrte Implikation erhalten wir sogar in etwas stärkerer Form: Falls die Tiefensuche einen Pfad von v zu w verfolgt hat, gilt die Ungleichung

$$d_v \leq d_w < f_w \leq f_v. \tag{4.2}$$

Beweis. Wir beweisen zunächst per Induktion über $f_v - d_v \in \mathbb{N}$, dass aus (4.1) die Existenz eines Pfads folgt.

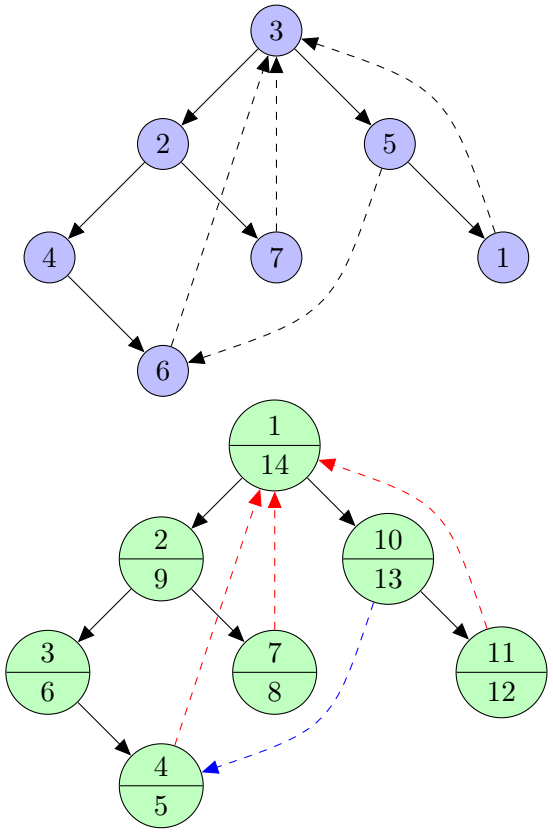


Abbildung 4.9: Tiefensuche mit Ankunfts- und Abschiedszeiten. Im unteren Graphen ist für jeden Knoten jeweils in der oberen Hälfte die Ankunfts- und in der unteren die Abschiedszeit angegeben.

Induktionsanfang. Sei $\ell = 1$. Seien $v, w \in V$ mit (4.1) und $f_v - d_v \leq \ell$ gegeben. Nach Konstruktion der Tiefensuche gilt

$$d_v + 1 \leq f_v \leq d_v + \ell = d_v + 1,$$

also $f_v = d_v + 1$. Aus (4.1) folgt dann

$$d_v \leq d_w < f_v = d_v + 1,$$

also $d_v = d_w$ und damit $v = w$. Demnach erfüllt der Pfad (v) der Länge null unsere Anforderungen.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass für alle $v, w \in V$ mit (4.1) und $f_v - d_v \leq \ell$ ein von der Tiefensuche verfolgter Pfad von v zu w existiert.

Induktionsschritt. Seien $v, w \in V$ mit (4.1) und $f_v - d_v \leq \ell + 1$ gegeben. Für $f_v - d_v \leq \ell$ folgt die Existenz eines Pfads bereits aus der Induktionsvoraussetzung. Gelte also nun $f_v - d_v = \ell + 1$. Da $f_v = d_v + \ell + 1 > d_v + 1$ gilt, muss die Tiefensuche ausgehend von

v andere Knoten besucht haben, zu denen Kanten von v führen. Seien $z_1, \ldots, z_k \in V$ diese Knoten, und seien sie in der Reihenfolge angeordnet, in der die Tiefensuche sie besucht hat. Da wir die Knoten der Reihe nach besuchen, erfüllen ihre Ankunfts- und Abschiedszeiten die Gleichungen

$$d_{z_1} = d_v + 1, \qquad d_{z_{i+1}} = f_{z_i} + 1, \qquad f_v = f_{z_k} + 1 \qquad \text{für alle } i \in \{1, \ldots, k\}.$$

Falls $d_v = d_w$ gelten sollte, folgt bereits $v = w$ und wir können wieder den Pfad (v) der Länge null verwenden.

Anderenfalls gilt $d_v < d_w < f_v$, also $d_{z_1} \leq d_w$ und $d_w \leq f_{z_k}$. Da eine Ankunftszeit nach unserer Konstruktion niemals gleich einer Abschiedszeit (denn es wird jedesmal hochgezählt) sein kann, muss sogar $d_w < f_{z_k}$ gelten.

Also existiert ein $j \in \{1, \ldots, k\}$ mit $d_{z_j} \leq d_w < f_{z_j}$. Wir setzen $z := z_j$ und wenden die Induktionsvoraussetzung an, um einen Pfad $(v_0, \ldots, v_n)$ von z zu w zu erhalten, der von der Tiefensuche verfolgt wurde. Da $(v, v_0) = (v, z) = (v, z_j) \in E$ gilt, ist dann $(v, v_0, \ldots, v_n)$ ein Kantenzug von v zu w, der von der Tiefensuche verfolgt wurde. Da die Tiefensuche keinen Knoten zweimal besucht, ist es auch ein Pfad.

Damit ist der erste Teil des Beweises abgeschlossen. Wir beweisen nun, dass (4.2) gilt, falls ein Pfad von v zu w von der Tiefensuche verfolgt wurde. Dazu verwenden wir eine Induktion über die Pfadlänge.

Induktionsanfang. Sei $\ell = 0$. Für jeden Pfad der Länge null stimmen Anfangs- und Endknoten überein, also gilt auch (4.2).

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass für alle $v, w \in V$ die Ungleichung (4.2) gilt, falls sie durch einen von der Tiefensuche verfolgten Pfad der Länge ℓ verbunden sind.

Induktionsschritt. Seien $v, w \in V$ so gegeben, dass ein von der Tiefensuche verfolgter Pfad der Länge $\ell+1$ von v zu w existiert. Wir bezeichnen diesen Pfad mit $(v_0, \ldots, v_{\ell+1})$.

Dann ist offenbar $(v_1, \ldots, v_{\ell+1})$ ein Pfad der Länge ℓ von $z := v_1$ zu $w = v_{\ell+1}$, der von der Tiefensuche verfolgt wurde, also muss nach Induktionsvoraussetzung

$$d_z \leq d_w < f_w \leq f_z$$

gelten. Da die Tiefensuche auch die Kante $(v_0, v_1) = (v, z)$ verfolgt hat, müssen auch $d_v < d_z$ und $f_z < f_v$ gelten, so dass wir

$$d_v < d_z \leq d_w < f_w \leq f_z < f_v$$

erhalten, also insbesondere (4.2). ∎

Nach Abschluss des Algorithmus' können wir also durch Prüfen der Ankunfts- und Abschiedszeiten feststellen, ob ein Knoten w von einem Knoten v aus erreicht wurde.

Wir können auch feststellen, ob eine Kante $e = (v, w) \in E$ zu einem Kreis gehört: Falls $d_w \leq d_v < f_w$ gilt, muss nach Satz 4.13 ein Pfad von w zu v existieren, den wir mit der Kante e zu einem Kreis schließen können.

In der Regel werden die Kanten des Graphen abhängig von den Ankunfts- und Abschiedszeiten in drei Kategorien gegliedert:

- $e = (v, w) \in E$ ist eine *Vorwärtskante*, falls $d_v \leq d_w < f_v$ gilt, falls also w von einem Pfad von v aus erreicht wurde.

  Ein Sonderfall ist die *Baumkante*, die sich dadurch auszeichnet, dass kein $z \in V$ mit $d_v < d_z < d_w < f_z < f_v$ existiert, dass also auf dem Pfad von v zu w kein anderer Knoten besucht wurde.

- $e = (v, w) \in E$ ist eine *Rückwärtskante*, falls $d_w \leq d_v < f_w$ gilt, falls also v von einem Pfad von w aus erreicht wurde.

- $e = (v, w) \in E$ ist eine *Querkante*, falls $f_v < d_w$ oder $f_w < d_v$ gilt, falls also die Verarbeitung des einen Knotens schon abgeschlossen war, bevor mit der des anderen begonnen wurde.

In Abbildung 4.9 ist oben der Graph und unten der Graph mit eingetragenen Ankunfts- und Abschiedszeiten dargestellt. Rückwärtskanten sind rot eingefärbt, Querkanten blau, alle Vorwärtskanten sind in diesem Beispiel auch Baumkanten.

4.4 Optimale Pfade in gewichteten Graphen

Mit Hilfe der Breitensuche können wir die kürzesten Pfade zwischen zwei Knoten in einem Graphen berechnen. Dabei ist die Pfadlänge gegeben durch die Anzahl der Kanten, denen wir folgen müssen.

In der Praxis sind häufig nicht alle Kanten „gleich lang“, beispielsweise weisen Straßensegmente in der Datenbank eines Navigationssystems in der Regel unterschiedliche Längen auf. In diesem Fall ist anzunehmen, dass die Breitensuche keine nützlichen Ergebnisse berechnen wird.

Die unterschiedliche Länge der Kanten beschreiben wir durch eine *Gewichtsfunktion*

$$d : E \to \mathbb{R},$$

die jeder Kante $(v, w) \in E$ einen Wert $d(v, w)$ zuordnet, den wir als Länge der Kante interpretieren.

Die gewichtete Länge eines Kantenzugs $(v_0, \ldots, v_n)$ ist dann die reelle Zahl

$$\sum_{i=1}^{n} d(v_{i-1}, v_i).$$

Wir suchen einen Algorithmus, der für zwei Knoten $v, w \in E$ einen Kantenzug von v mit w findet, dessen *gewichtete* Länge minimal ist.

Diese Aufgabe lässt sich besonders elegant lösen, falls die Gewichte $d(v, w)$ nicht negativ sind: Diese Eigenschaft erlaubt es uns, das Lemma 4.4 zu verallgemeinern.

Lemma 4.14 (Gewichteten Kantenzug kürzen) *Sei $G = (V, E)$ ein Graph, $d : E \to \mathbb{R}_{\geq 0}$ eine nicht-negative Gewichtsfunktion und $(v_0, v_1, \ldots, v_n)$ ein Kantenzug. Falls $v_i = v_j$ für ein $i < j$ gelten sollte, existiert ein Kantenzug der Länge $n - (j - i) < n$, dessen* gewichtete *Länge nicht größer als die des ursprünglichen ist.*

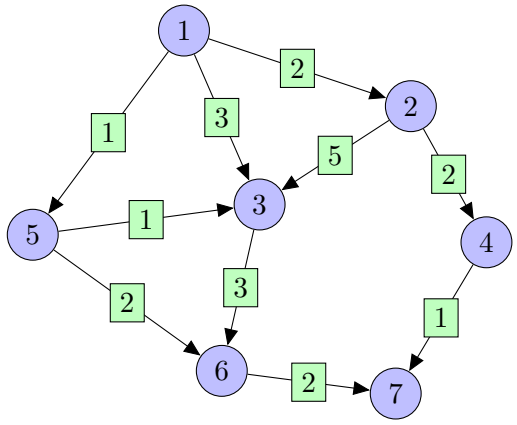


Abbildung 4.10: Graph mit Kantengewichten.

Beweis. Gelte $v_i = v_j$ für $i < j$. Dann ist $(v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_n)$ wegen $(v_i, v_{j+1}) = (v_j, v_{j+1})$ ein Kantenzug, und zwar einer der Länge $n - (j - i)$.

Seine gewichtete Länge beträgt

$$\begin{aligned}&\sum_{k=1}^{i} d(v_{k-1}, v_k) + \sum_{k=j+1}^{n} d(v_{k-1}, v_k)\\ &\quad\leq \sum_{k=1}^{i} d(v_{k-1}, v_k) + \underbrace{\sum_{k=i+1}^{j} d(v_{k-1}, v_k)}_{\geq 0} + \sum_{k=j+1}^{n} d(v_{k-1}, v_k) = \sum_{k=1}^{n} d(v_{k-1}, v_k),\end{aligned}$$

ist also nicht größer als die des ursprünglichen Kantenzugs. ∎

Zur Vereinfachung werden wir für den Rest dieses Abschnitts abkürzend von der Länge eines Kantenzugs sprechen, wenn die gewichtete Länge gemeint ist.

Aus unserem Lemma folgt insbesondere, dass wir aus einem Kantenzug minimaler Länge auch einen Pfad minimaler Länge konstruieren können, indem wir doppelt auftretende Knoten eliminieren.

Wir dürfen uns also darauf beschränken, einen *Pfad* minimaler Länge zu suchen.

Der nach E. Dijkstra benannte *Dijkstra-Algorithmus* beruht auf der Idee, auszunutzen, dass sich ein Pfad minimaler Länge aus Teilpfaden minimaler Länge zusammensetzen lassen muss, denn wenn ein Teilpfad länger als nötig wäre, ließe sich der Gesamtpfad verkürzen, indem man den Teilpfad verkürzt. Mathematisch präzise fassen wir diese Aussage in dem folgenden Lemma.

Lemma 4.15 (Optimalität) *Sei $d : E \to \mathbb{R}_{\geq 0}$ eine nicht-negative Gewichtsfunktion. Falls $(v_0, \ldots, v_n)$ ein Pfad minimaler Länge von v zu w ist, muss für jedes $i \in \{0, \ldots, n\}$ der Pfad $(v_0, \ldots, v_i)$ ein Pfad minimaler Länge von v zu v_i sein.*

Beweis. Sei $i \in \{0, \dots, n\}$, und sei $(v'_0, \dots, v'_k)$ ein Pfad von $v = v'_0$ zu $v_i = v'_k$. Dann ist $(v'_0, \dots, v'_k, v_{i+1}, \dots, v_n)$ wegen $(v'_k, v_{i+1}) = (v_i, v_{i+1}) \in E$ ein Kantenzug von v zu w. Seine Länge beträgt

$$\sum_{\ell=1}^{k} d(v'_{\ell-1}, v'_\ell) + \sum_{\ell=i+1}^{n} d(v_{\ell-1}, v_\ell).$$

Falls $(v_0, \dots, v_n)$ die minimale Länge aufweist, muss demnach

$$\sum_{\ell=1}^{k} d(v'_{\ell-1}, v'_\ell) + \sum_{\ell=i+1}^{n} d(v_{\ell-1}, v_\ell) \geq \sum_{\ell=1}^{n} d(v_{\ell-1}, v_\ell)$$

gelten, und indem wir den zweiten Summanden der linken Seite von beiden Seiten subtrahieren folgt

$$\sum_{\ell=1}^{k} d(v'_{\ell-1}, v'_\ell) \geq \sum_{\ell=1}^{i} d(v_{\ell-1}, v_\ell),$$

also kann die Länge des Pfads $(v'_0, \dots, v'_k)$ nicht echt kleiner als die des Pfads $(v_0, \dots, v_i)$ sein. ∎

Der Dijkstra-Algorithmus beruht auf der Idee, Pfade minimaler Länge schrittweise um weitere Kanten zu verlängern, um so Pfade minimaler Länge zu weiteren Knoten zu konstruieren. Durch eine geschickte Auswahl dieser neuen Zielknoten lässt sich mit geringem Aufwand garantieren, dass tatsächlich Pfade minimaler Länge entstehen.

Dazu wird für jeden Schritt eine Menge $M_\ell \subseteq V$ von Knoten konstruiert, für die Pfade minimaler Länge bekannt sind. In jedem Schritt wird ein weiterer Knoten zu M_ℓ hinzu genommen, bis schließlich alle Knoten in dieser Menge enthalten sind. Dann sind Pfade minimaler Länge für alle Knoten gefunden.

Diese Pfade können wir verlängern, indem wir einen Knoten $w \in V \setminus M_\ell$ wählen, zu dem eine Kante $(z, w) \in E$ von einem Knoten $z \in M_\ell$ führt. Derjenige dieser Knoten, für den der kürzeste Pfad entsteht, wird dann der Menge M_ℓ hinzugefügt, um $M_{\ell+1}$ zu erhalten.

Zur Abkürzung bezeichnen wir für alle $w \in V$ die Länge eines Pfads mit minimaler Länge von v zu w mit δ_w.

Die Länge der Pfade, die unser Algorithmus von v zu w konkret konstruiert, bezeichnen wir mit $\hat{\delta}_w$, um anschließend zu beweisen, dass $\hat{\delta}_w = \delta_w$ gilt, dass wir also tatsächlich die Pfade minimaler Länge gefunden haben.

Um nun ein $w \in V \setminus M_\ell$ zu finden, zu dem ein kürzester Pfad führt, genügt es nach Lemma 4.14, alle $z \in M_\ell$ mit $(z, w) \in E$ zu untersuchen und die Länge $\delta_z + d(z, w)$ des Kantenzugs zu berücksichtigen, der entstehen würde, wenn wir den *kürzesten* Kantenzug von v zu z um die Kante von z zu w verlängern.

Insgesamt erhalten wir die folgende Vorgehensweise:

1. Wir beginnen mit

   $$M_1 := \{v\}, \qquad \hat{\delta}_v := 0, \qquad \ell := 1,$$

   denn einen Pfad von v zu v der minimalen Länge null kennen wir bereits.

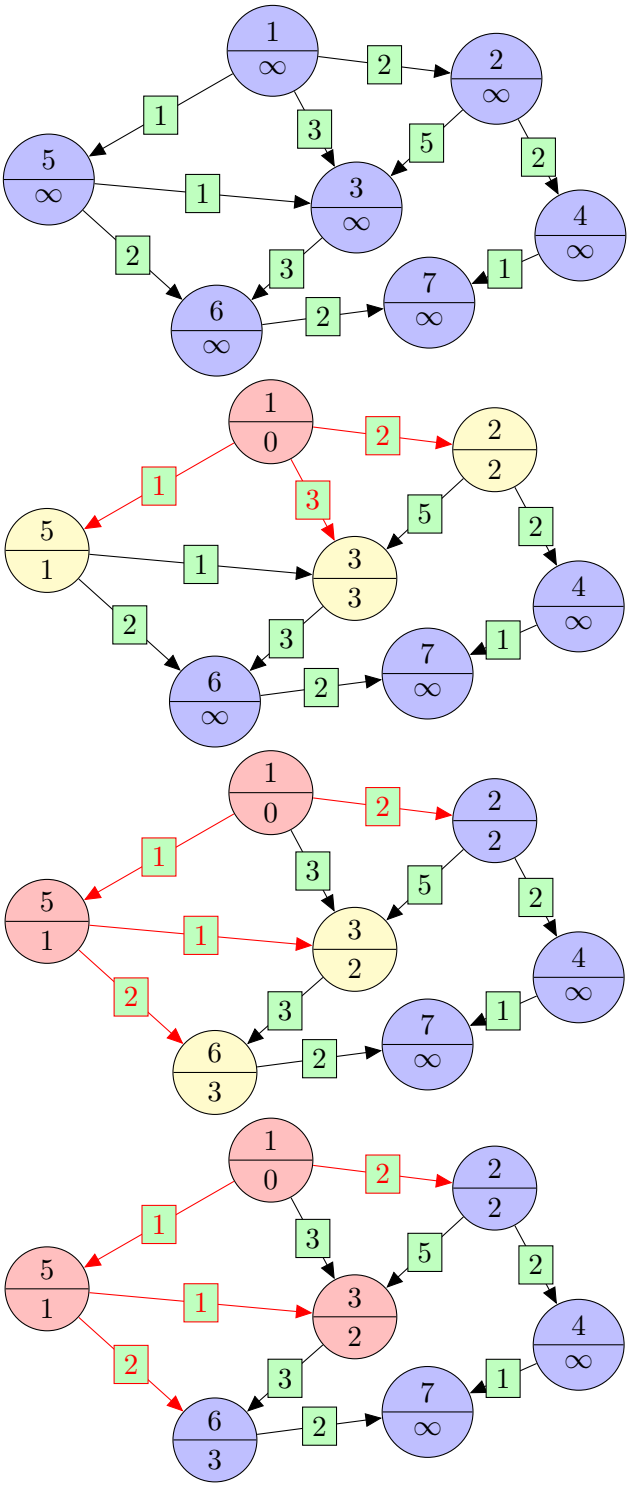


Abbildung 4.11: Dijkstra-Algorithmus, Teil 1.

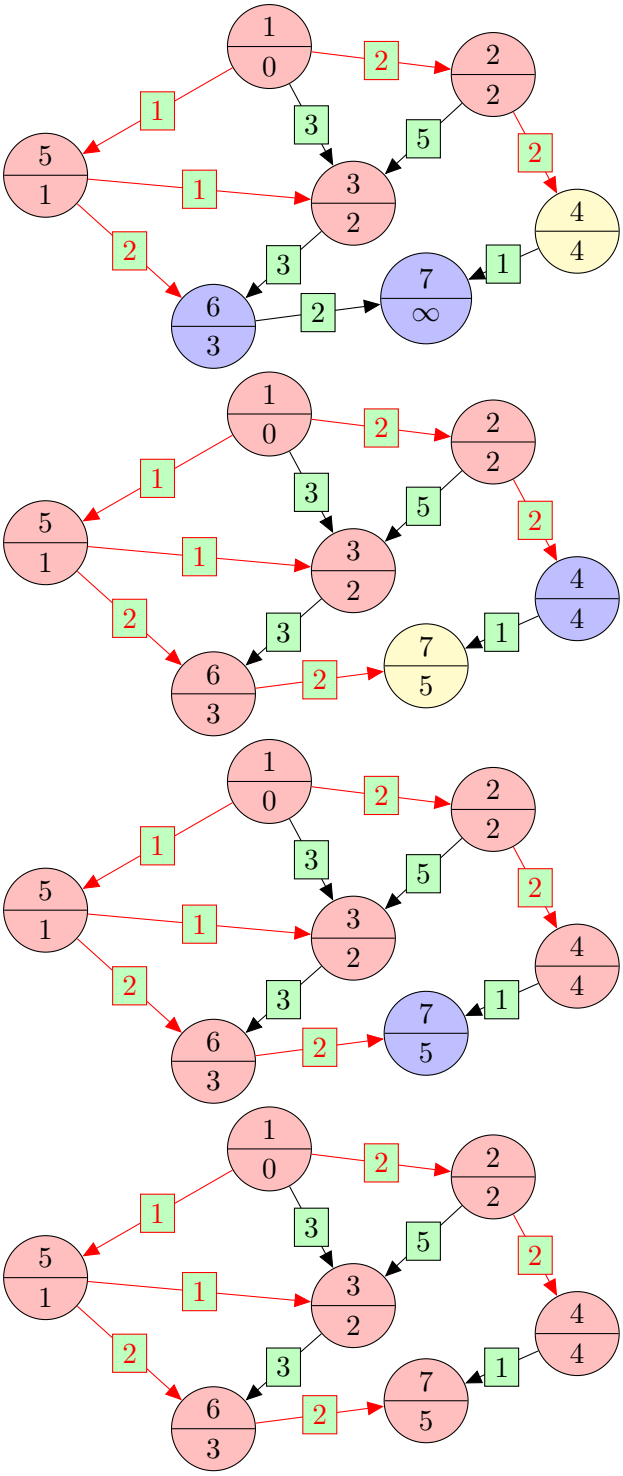


Abbildung 4.12: Dijkstra-Algorithmus, Teil 2.

2. Falls $V = M_\ell$ gilt, sind wir fertig.

3. Sonst finden wir ein $w \in V \setminus M_\ell$ und ein $z \in M_\ell$ derart, dass $(z, w) \in E$ gilt und $\hat{\delta}_z + d(z, w)$ minimal ist.

4. Wir setzen

$$M_{\ell+1} := M_\ell \cup \{w\}, \qquad \hat{\delta}_w := \hat{\delta}_z + d(z, w), \qquad \ell \leftarrow \ell + 1$$

und fahren in Schritt 2 fort, solange $M_\ell \neq V$ gilt.

Da sich die Zugehörigkeit eines Knotens zu der Menge M_ℓ einfach mit Hilfe von Markierungen und die Auswahl eines geeigneten Knotens $w \in V \setminus M_\ell$ mit einer Halde elegant lösen lassen, erhalten wir eine praktisch durchführbare Konstruktion.

Satz 4.16 (Dijkstra-Algorithmus) *Sei $d : E \to \mathbb{R}_{\geq 0}$ eine nicht-negative Gewichtsfunktion. Für alle $w \in V$ gilt $\hat{\delta}_w = \delta_w$.*

Beweis. Wir beweisen die Aussage

$$(\ell \leq |V| \wedge z \in M_\ell) \Rightarrow \hat{\delta}_z = \delta_z \qquad \text{für alle } \ell \in \mathbb{N},\ z \in V \tag{4.3}$$

per Induktion über $\ell \in \mathbb{N}$.

Induktionsanfang. Für $\ell = 1$ gilt $M_1 = \{v\}$, und der kürzeste Pfad von v zu v hat die Länge $\delta_v = 0 = \hat{\delta}_v$. Nach Konstruktion gilt $\hat{\delta}_u \geq 0$ für alle $u \in V$, also auch $\hat{\delta}_v \leq \hat{\delta}_u$.

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}$ so gegeben, dass (4.3) gilt.

Induktionsschritt. Falls $\ell + 1 > |V|$ gilt, ist nichts zu beweisen.

Ansonsten existieren nach Konstruktion ein $w \in V \setminus M_\ell$ und ein $z \in M_\ell$ so, dass $M_{\ell+1} = M_\ell \cup \{w\}$ und $(z, w) \in E$ gelten und die Minimalitätsbedingung

$$\hat{\delta}_z + d(z, w) \leq \hat{\delta}_{z'} + d(z', w') \qquad \text{für alle } z' \in M_\ell,\ w' \in V \setminus M_\ell,\ (z', w') \in E \tag{4.4}$$

erfüllt ist. Da der Unterschied zwischen $M_{\ell+1}$ und M_ℓ nur das Element w ist, brauchen wir auch nur zu beweisen, dass $\hat{\delta}_w = \delta_w$ gilt.

Zunächst zeigen wir, dass $\hat{\delta}_w \geq \delta_w$ gilt. Nach Definition finden wir einen Pfad $(v_0, \ldots, v_n)$ der Länge δ_z von v zu z. Dann ist $(v_0, \ldots, v_n, w)$ wegen $(v_n, w) = (z, w) \in E$ ein Kantenzug der Länge $\delta_z + d(z, w)$. Wir können die Induktionsvoraussetzung auf $z \in M_\ell$ anwenden, um $\hat{\delta}_z = \delta_z$ zu erhalten, also hat unser Kantenzug die Länge $\hat{\delta}_z + d(z, w) = \hat{\delta}_w$. Da δ_w die minimale Länge aller Kantenzüge von v zu w ist, folgt daraus bereits $\hat{\delta}_w \geq \delta_w$.

Nun müssen wir $\hat{\delta}_w \leq \delta_w$ zeigen. Dazu wählen wir einen Pfad $(v_0, \ldots, v_n)$ minimaler Länge von v zu w und setzen

$$m := \max\{i \in \{0, \ldots, n\} \ :\ v_i \in M_\ell\}.$$

Wir stellen fest, dass wegen $v_n = w \notin M_\ell$ und $v_0 = v \in M_\ell$ die Ungleichungen $0 \leq m < n$ gelten müssen.

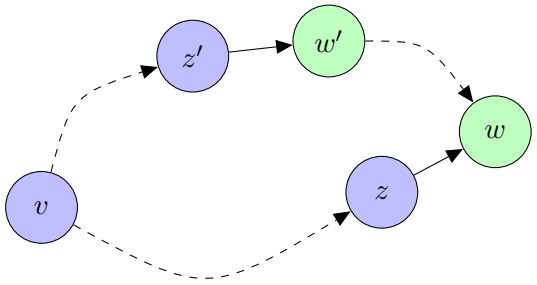


Abbildung 4.13: Der kürzeste Pfad (oben) zu w verläuft über z' und w'. Pfade sind gestrichelt dargestellt. Knoten in M_ℓ sind blau markiert, der Rest grün.

Wir setzen $z' := v_m \in M_\ell$ und $w' := v_{m+1}$ (siehe Abbildung 4.13). Aus der Wahl von m folgt $w' \in V \setminus M_\ell$.

Nach Lemma 4.15 ist $(v_0, \ldots, v_m)$ ein Pfad minimaler Länge von $v = v_0$ zu $z' = v_m$, also gilt

$$\delta_{z'} = \sum_{i=1}^{m} d(v_{i-1}, v_i).$$

Mit der Induktionsvoraussetzung folgt aus $z' \in M_\ell$ auch $\delta_{z'} = \hat{\delta}_{z'}$, also

$$\hat{\delta}_{z'} = \delta_{z'} = \sum_{i=1}^{m} d(v_{i-1}, v_i).$$

Aufgrund der Minimalitätsbedingung (4.4) und der Nicht-Negativität der Kantengewichte folgt

$$\begin{aligned}\delta_w &= \sum_{i=1}^{n} d(v_{i-1}, v_i) \geq \sum_{i=1}^{m+1} d(v_{i-1}, v_i) \\ &= \hat{\delta}_{z'} + d(v_m, v_{m+1}) = \hat{\delta}_{z'} + d(z', w') \geq \hat{\delta}_z + d(z, w) = \hat{\delta}_w.\end{aligned}$$

Da wir $\delta_w \leq \hat{\delta}_w$ bereits bewiesen haben, erhalten wir also $\delta_w = \hat{\delta}_w$. ∎

Um nicht in jedem Schritt des Algorithmus' *alle* $z \in M_\ell$ untersuchen zu müssen, bietet es sich an, den Algorithmus so zu arrangieren, dass die relevanten Größen mit geringem Aufwand aktualisiert werden können, sobald M_ℓ vergrößert wird.

Dazu verwenden wir ein Array `delta`, das die Werte δ_w für alle Knoten $w \in V$ aufnehmen soll. Im ℓ-ten Schritt des Algorithmus' können wir die Komponenten des Arrays, die zu Knoten *außerhalb* der Menge M_ℓ gehören, allerdings verwenden, um die für den Algorithmus benötigten Größen

$$\tilde{\delta}_w^{(\ell)} := \min\{\delta_z + d(z, w) \ : \ z \in M_\ell,\ (z, w) \in E\} \qquad \text{für alle } w \in V \setminus M_\ell$$

aufzunehmen. Für theoretische Zwecke können wir dabei von $\min \emptyset = \infty$ ausgehen, ein Knoten w, zu dem keine Kante aus der Menge M_ℓ führt, hat also einen „unendlich großen Abstand“ zu v.

Falls nun M_ℓ durch Hinzunahme eines neuen Knotens z zu $M_{\ell+1} = M_\ell \cup \{z\}$ wird, brauchen wir lediglich für alle Knoten $w \in V \setminus M_{\ell+1}$ mit $(z, w) \in E$ zu prüfen, ob sich $\tilde{\delta}_w^{(\ell+1)}$ gegenüber $\tilde{\delta}_w^{(\ell)}$ reduziert.

```
1  void
2  shortest_path(const graph *g, int src, float *delta, int *pred)
3  {
4    int *visited;
5    edge *e;
6    int z, w;
7    visited = (int *) malloc(sizeof(int) * g->vertices);
8    for(w=0; w<g->vertices; w++) visited[w] = 0;
9    for(w=0; w<g->vertices; w++) pred[w] = -1;
10   z = src;
11   delta[z] = 0.0; visited[z] = 1;
12   while(z >= 0) {
13     visited[z] = 2;
14     for(e=g->edgelist[z]; e; e=e->next) {
15       w = e->to;
16       if(visited[w] < 1) {
17         delta[w] = delta[z] + e->weight; pred[w] = z;
18         visited[w] = 1;
19       }
20       else if(delta[w] > delta[z] + e->weight) {
21         delta[w] = delta[z] + e->weight; pred[w] = z;
22       }
23     }
24     z = -1;
25     for(w=0; w<g->vertices; w++)
26       if(visited[w] == 1)
27         if((z < 0) || (delta[w] < delta[z]))
28           z = w;
29   }
30   free(visited);
31 }
```

Abbildung 4.14: Einfache Fassung des Dijkstra-Algorithmus’ für die Berechnung kürzester Pfade in einem gewichteten Graphen.

Die resultierende Fassung des Dijkstra-Algorithmus ist in Abbildung 4.14 dargestellt.

Wie schon bei der Breitensuche werden in dem Array `pred` die Vorgänger der Knoten auf dem jeweils kürzesten Pfad zu dem Ausgangsknoten `src` gespeichert, so dass sich diese Pfade einfach rekonstruieren lassen. Das Array `visited` verwenden wir, um festzustellen, ob ein Knoten bereits einmal besucht worden ist. Wenn `w` das erste Mal besucht wird, setzen wir `visited[w]=1`, um zu signalisieren, dass `delta[w]` die Länge eines Pfads von `src` zu `w` enthält. Sobald der Knoten `z` in die Menge M_ℓ aufgenommen wird, setzen wir `visited[z]=2`, um festzuhalten, dass er bei zukünftigen Suchen nach dem Knoten mit dem geringsten Abstand zu M_ℓ nicht mehr berücksichtigt werden soll.

Bemerkung 4.17 (Bellman-Optimalitätsprinzip) *Der Dijkstra-Algorithmus kann als eine Anwendung des allgemeineren* Optimalitätsprinzips von Bellman *gesehen werden, das auf R. Bellman zurückgeht und besagt, dass eine optimale Lösung mancher Probleme aus optimalen Lösungen von Teilproblemen zusammengesetzt werden kann. Demzufolge können wir uns der optimalen Lösung nähern, indem wir optimale Lösungen von Teilproblemen konstruieren.*

Im Fall des Dijkstra-Algorithmus' ist dieses Prinzip in Lemma 4.15 formuliert: Ein Pfad minimaler Länge von v zu w besteht aus Pfaden minimaler Länge von v zu Zwischenpunkten z, also genügt es, wenn wir bereits gefundene Pfade minimaler Länge nach und nach um einzelne Kanten erweitern, bis wir schließlich alle Pfade konstruiert haben.

Bemerkung 4.18 (Rechenaufwand) *Da in jeder Iteration der äußeren Schleife in den Zeilen 12 bis 29 jeweils ein bisher nicht mit* 2 *markierter Knoten markiert, also der Menge M_ℓ hinzugefügt, wird, kann die äußere Schleife höchstens $|V|$-mal durchlaufen werden. Die erste innere Schleife in den Zeilen 14 bis 23 durchläuft alle von z ausgehenden Kanten, und da jedes $z \in V$ höchstens einmal auftritt, tritt auch jede Kante höchstens einmal auf, so dass sich für* alle *Durchläufe der ersten inneren Schleife ein Aufwand von $\mathcal{O}(|E|)$ ergibt. Die zweite innere Schleife in den Zeilen 24 bis 28 dient der Suche nach dem Knoten mit dem kleinsten Abstand zu M_ℓ, kann also höchstens $|V|-1$ Iterationen erfordern. Da die äußere Schleife höchstens $|V|$-mal durchlaufen wird, erhalten wir für die zweite innere Schleife einen Gesamtaufwand von $\mathcal{O}(|V|^2)$. Für den vollständigen Algorithmus ergibt sich so ein Aufwand von $\mathcal{O}(|V|^2 + |E|)$.*

4.5 Verbesserte Fassung des Dijkstra-Algorithmus'

Eine *quadratische* Abhängigkeit des Rechenaufwands von der Knotenanzahl $|V|$ ist relativ unattraktiv. Dieser Aufwand wird von der Suche nach einem z mit minimaler Pfadlänge $\tilde{\delta}_z$ in den Zeilen 24 bis 28 des Algorithmus' verursacht, also sind wir daran interessiert, diesen Teil zu beschleunigen.

Auf M. L. Fredman und R. E. Tarjan geht die Idee zurück, diese Aufgabe mit Hilfe einer Halde zu lösen. Wir werden hier statt der von ihnen vorgeschlagenen *Fibonacci-Halde* die bereits in Abschnitt 3.7 besprochene Umsetzung einer Halde durch ein Array so modifizieren, dass sie sich für unsere Zwecke eignet.

Für den Dijkstra-Algorithmus benötigen wir eine Datenstruktur, mit der wir schnell das *minimale* Element einer Menge finden und aus der Menge entfernen sowie neue

Elemente hinzufügen können. Außerdem brauchen wir für die Aktualisierung der Längen $\tilde{\delta}_w$ eine Möglichkeit, den Wert eines in der Menge enthaltenen Elements zu reduzieren.

Die ersten beiden Aufgaben lassen sich mit einer Halde erfüllen, bei der wir schlicht die Ordnung $\leq$ durch $\geq$ ersetzen. Die dritte Aufgabe erfordert zusätzlichen Aufwand: Wir müssen herausfinden können, welcher Index zu einem bereits in die Halde aufgenommenen Knoten gehört. Das lässt sich einfach durch ein Array bewerkstelligen, das jedem Knoten seinen Platz in der Halde zuordnet.

Da die Halde nicht nur die Werte $\tilde{\delta}_w$, sondern auch die zugehörigen Knoten w speichern soll, verwenden wir den Datentyp `heapelement`, der wie folgt definiert ist:

```
1  typedef struct _heapelement heapelement;
2  struct _heapelement {
3    float delta;
4    int vertex;
5  };
```

Die eigentliche Halde wird wieder durch ein Array beschrieben, dem wir ein weiteres Array zur Seite stellen, das angibt, wo ein bestimmter Knoten in der Halde zu finden ist. Außerdem bietet es sich an, den aktuellen „Füllstand“ der Halde zu speichern, so dass sich die folgende Datenstruktur ergibt:

```
1  typedef struct _heap heap;
2  struct _heap {
3    heapelement *data;
4    int *inheap;
5    int size;
6    int maxsize;
7  };
```

Wir werden die Halde so organisieren, dass das *kleinste* Element, also das mit dem geringsten `delta`-Eintrag, in der Wurzel steht. Wie schon im Heapsort-Algorithmus müssen wir dazu in der Lage sein, dieses Element effizient aus der Halde zu entfernen, und wie schon in besagtem Algorithmus lösen wir diese Aufgabe, indem wir das letzte Element des Arrays in die Wurzel wechseln und dann absinken lassen, bis die Halden-Eigenschaft wiederhergestellt ist.

Das Auswählen und Entfernen desjenigen $w \in V \setminus M_\ell$, für das $\tilde{\delta}_w$ minimal ist, können wir mit einer derartige organisierten Halde einfach bewerkstelligen: Das gewünschte Element steht infolge der Halden-Eigenschaft in der Wurzel der Halde, ist also sehr leicht zu finden. Um es zu entfernen, ersetzen wir es durch das letzte Element des Arrays, das die Halde repräsentiert und stellen die Halden-Eigenschaft wieder her, indem wir dieses Element in der Halde absinken lassen. Eine Implementierung findet sich in Abbildung 4.15.

Wenn M_ℓ gewachsen ist, wenn also $M_{\ell+1} = M_\ell \cup \{z\}$ gilt, müssen wir $\tilde{\delta}_w$ für alle $w \in V \setminus M_{\ell+1}$ aktualisieren. Nach Konstruktion des Dijkstra-Algorithmus kann $\tilde{\delta}_w$ dabei nur kleiner werden, es kann also sein, dass die Halden-Eigenschaft nach der Aktualisierung verletzt ist. In diesem Fall können wir den zu w gehörenden Eintrag in der Halde aufsteigen lassen, bis die Eigenschaft wieder gilt.

```
1  void
2  sink(heap *hp, int i)
3  {
4    heapelement *x = hp->data;
5    int *inh = hp->inheap;
6    int n = hp->size;
7    heapelement xi;
8    int j;
9    xi = x[i];
10   j = 2*i+1;
11   while(j < n) {
12     if((j+1 < n) && (x[j]->delta > x[j+1]->delta))
13       j++;
14     if(xi->delta > x[j]->delta) {
15       x[i] = x[j]; inh[x[i]->vertex] = i; i = j;
16       j = 2*i+1;
17     }
18     else break;
19   }
20   x[i] = xi; inh[xi->vertex] = i;
21 }
22 int
23 removemin(heap *hp)
24 {
25   int z;
26   if(hp->size == 0) crash("Heap empty");
27   z = hp->data[0]->vertex;
28   hp->size--;
29   hp->data[0] = hp->data[hp->size];
30   sink(hp, 0);
31   return z;
32 }
```

Abbildung 4.15: Entfernen des minimalen Elements aus der Halde

Es kann auch sein, dass von dem neuen Knoten z aus ein Knoten $w \in V$ erreichbar ist, der vorher noch nicht untersucht wurde. Diesen neuen Knoten fügen wir dann am Ende des die Halde repräsentierenden Arrays ein und lassen ihn aufsteigen, bis die Halden-Eigenschaft wieder hergestellt ist. Eine mögliche Implementierung findet sich in Abbildung 4.16.

Der Einsatz der Halde führt wie gewünscht dazu, dass sich der Dijkstra-Algorithmus wesentlich effizienter ausführen lässt: Die aufwendige Suche nach dem Minimum erfordert nun nicht mehr $\mathcal{O}(|V|)$ Operationen, sondern begnügt sich mit $\mathcal{O}(\log_2(|V|+1))$.

```
1  void
2  rise(heap *hp, int j)
3  {
4    heapelement *x = hp->data;
5    int *inh = hp->inheap;
6    int n = hp->size;
7    heapelement xj;
8    int i;
9    xj = x[j];
10   while(j > 0) {
11     i = (j-1)/2;
12     if(x[i]->delta > xj) {
13       x[j] = x[i]; inh[x[j]->vertex] = j; j = i;
14     }
15     else break;
16   }
17   x[j] = xj; inh[xj->vertex] = j;
18 }
19 void
20 addkey(heap *hp, int vertex, float delta)
21 {
22   if(hp->size >= hp->maxsize) crash("Heap overflow");
23   hp->data[hp->size]->vertex = vertex;
24   hp->data[hp->size]->delta = delta;
25   hp->size++;
26   rise(hp, hp->size-1);
27 }
28 void
29 decreasekey(heap *hp, int vertex, float delta)
30 {
31   int j;
32   j = hp->inheap[vertex];
33   if(delta > hp->data[j]->delta) crash("Delta increased");
34   hp->data[j]->delta = delta;
35   rise(hp, j);
36 }
```

Abbildung 4.16: Einfügen eines Elements und Reduzieren des δ_z-Werts

Bemerkung 4.19 (Rechenaufwand) *Jeder Knoten wird höchstens einmal in die Halde aufgenommen und höchstens einmal aus ihr entfernt. Die Halde hat eine maximale Tiefe von* $\lceil \log_2(|V|+1) \rceil$*, also fallen für das Einfügen und Entnehmen nicht mehr als* $\mathcal{O}(|V| \log_2(|V|+1))$ *Operationen an. Bei jeder neuen Kante kann ein Knoten in der*

```
1  void
2  shortest_path(const graph *g, int src, float *delta, int *pred)
3  {
4    heap *hp;
5    edge *e;
6    int z, w;
7    hp = new_heap(g->vertices);
8    for(w=0; w<g->vertices; w++) pred[w] = -1;
9    z = src; delta[z] = 0.0; pred[z] = z;
10   addkey(hp, z, delta[z]);
11   while(!isempty(hp)) {
12     z = removemin(hp);
13     for(e=g->edgelist[z]; e; e=e->next) {
14       w = e->to;
15       if(pred[w] < 0) {
16         delta[w] = delta[z] + e->weight; pred[w] = z;
17         addkey(hp, w, delta[w]);
18       }
19       else if(delta[w] > delta[z] + e->weight) {
20         delta[w] = delta[z] + e->weight; pred[w] = z;
21         decreasekey(hp, w, delta[w]);
22       }
23     }
24   }
25   del_heap(hp);
26 }
```

Abbildung 4.17: Effiziente Implementierung des Dijkstra-Algorithmus', bei der die verbleibenden Knoten in einer nach ihren $\tilde{\delta}_w$-Werten sortierten Halde verwaltet werden.

Halde aufsteigen, allerdings höchstens $\lceil \log_2(|V|+1) \rceil$*-mal für jede Kante, so dass ein Aufwand von* $\mathcal{O}(|E| \log_2(|V|+1))$ *hinzu kommt.*

Insgesamt erhalten wir damit einen Aufwand von $\mathcal{O}((|E|+|V|) \log_2(|V|+1))$ *Operationen für den verbesserten Dijkstra-Algorithmus.*

Mit dem von Fredman und Tarjan verwendeten Fibonacci-Heap kann man den Rechenaufwand weiter auf $\mathcal{O}(|E| + |V| \log_2(|V|))$ *reduzieren.*

4.6 Optimale Pfade zwischen allen Knoten

Falls wir sehr häufig optimale Pfade zwischen beliebigen Knoten eines Graphs finden müssen, kann es sich lohnen, solche Pfade für *alle* Kombinationen von Anfangs- und

Endknoten zu berechnen und in einer geeigneten Tabelle zu speichern.

Diese Aufgabe können wir mit dem *Floyd-Warshall-Algorithmus* lösen, der nach R. Floyd und S. Warshall benannt ist. Er berechnet für einen Graphen $G = (V, E)$ mit Kantengewichten d die kürzesten Pfade zwischen allen Knoten $v, w \in V$ und kann, anders als der Dijkstra-Algorithmus, auch mit negativen Kantengewichten umgehen.

Allerdings gilt das nur, solange dadurch nicht die Existenz kürzester Pfade verloren geht: Falls beispielsweise $d(v, w) + d(w, v) < 0$ gelten würde, könnten wir durch $(v, w, v, w, v, w, \ldots, v, w)$ einen Kantenzug von v zu w beliebig kleiner Länge konstruieren, es gäbe also keinen Kantenzug minimaler Länge.

Um die Lösbarkeit der Aufgabe sicher zu stellen, müssen wir derartige Situationen verbieten: Es gelte

$$d(v_n, v_0) + \sum_{i=1}^{n} d(v_{i-1}, v_i) \geq 0 \qquad \text{für alle Kreise } (v_0, \ldots, v_n) \text{ in } G, \tag{4.5}$$

es sind also keine Kreise negativer Länge zugelassen. Mit Hilfe dieser Voraussetzung können wir die folgende verallgemeinerte Fassung des Lemmas 4.14 beweisen:

Lemma 4.20 (Kantenzug kürzen) *Sei $(v_0, \ldots, v_n)$ ein Kantenzug von v zu w. Dann können wir durch Wegstreichen von Knoten daraus einen Pfad von v zu w konstruieren, der nicht länger als der ursprüngliche Kantenzug ist.*

Beweis. Wir führen den Beweis per Induktion über die Anzahl der in dem Kantenzug doppelt vorkommenden Knoten, beweisen also

$$|\{(i, j) \ : \ 0 \leq i < j \leq n, \ v_i = v_j\}| \leq \ell \Rightarrow \text{Pfad kann konstruiert werden}$$

für alle $\ell \in \mathbb{N}_0$.

Induktionsanfang. Sei $\ell = 0$. Falls in dem Kantenzug kein Knoten doppelt vorkommt, ist er bereits ein Pfad und wir sind fertig.

Induktionsschritt. Sei $\ell \in \mathbb{N}_0$ so gegeben, dass für alle Kantenzüge $(v_0, \ldots, v_n)$ von v zu w mit

$$|\{(i, j) \ : \ 0 \leq i < j \leq n, \ v_i = v_j\}| \leq \ell$$

ein Pfad konstruiert werden kann, der nicht länger als der Kantenzug ist.

Induktionsvoraussetzung. Sei $(v_0, \ldots, v_n)$ ein Kantenzug von v zu w mit

$$|\{(i, j) \ : \ 0 \leq i < j \leq n \, v_i = v_j\}| \leq \ell + 1.$$

Die Menge ist offenbar nicht leer, also können wir aus ihr ein Paar (i, j) so wählen, dass $j - i$ minimal ist. Aufgrund dieser Minimalität muss dann $(v_i, \ldots, v_{j-1})$ ein Pfad und wegen $(v_{j-1}, v_i) = (v_{j-1}, v_j) \in E$ auch ein Kreis sein. Mit (4.5) folgt

$$\sum_{k=i+1}^{j} d(v_{k-1}, v_k) = d(v_{j-1}, v_j) + \sum_{k=i+1}^{j-1} d(v_{k-1}, v_k) = d(v_{j-1}, v_i) + \sum_{k=i+1}^{j-1} d(v_{k-1}, v_k) \geq 0.$$

Wegen $(v_i, v_{j+1}) = (v_j, v_{j+1}) \in E$ ist damit $(v'_0, \ldots, v'_m) := (v_0, \ldots, v_i, v_{j+1}, \ldots, v_n)$ ein Kantenzug von v zu w mit $m = n - (j - i)$, dessen (gewichtete) Länge wegen

$$\begin{aligned}
&\sum_{k=1}^{i} d(v_{k-1}, v_k) + \sum_{k=j+1}^{n} d(v_{k-1}, v_k) \\
&\quad \leq \sum_{k=1}^{i} d(v_{k-1}, v_k) + \sum_{k=i+1}^{j} d(v_{k-1}, v_k) + \sum_{k=j+1}^{n} d(v_{k-1}, v_k) \\
&\quad = \sum_{k=1}^{n} d(v_{k-1}, v_k)
\end{aligned}$$

nicht größer als die des ursprünglichen Kantenzugs $(v_0, \ldots, v_n)$ sein kann.

Da $(v'_0, \ldots, v'_m)$ aus diesem Kantenzug durch das Streichen der Elemente $v_{i+1}, \ldots, v_j$ entsteht, gilt

$$\begin{aligned}
&|\{(i', j') \ : \ 0 \leq i' < j' \leq m, \ v'_{i'} = v'_{j'}\}| \\
&\qquad = |\{(i', j') \ : \ 0 \leq i' < j' \leq n, \ v_{i'} = v_{j'}, \ i', j' \notin \{i+1, \ldots, j\}\}| \\
&\qquad \leq |\{(i', j') \ : \ 0 \leq i' < j' \leq n, \ v_{i'} = v_{j'}\}| - 1 = \ell,
\end{aligned}$$

also können wir mit der Induktionsvoraussetzung daraus einen Pfad konstruieren, ohne die Länge zu vergrößern. ∎

Ähnlich dem Dijkstra-Algorithmus basiert auch der Floyd-Warshall-Algorithmus auf der Idee, die optimale Lösung eines Problems aus optimalen Lösungen von Teilproblemen zu konstruieren. Er beruht auf der Idee der *dynamischen Programmierung*, bei der Lösungen der Teilprobleme in einer Tabelle aufgeführt und aktualisiert werden.

Im konkreten Fall bestehen die Teilprobleme darin, kürzeste Pfade zwischen zwei Knoten $v, w \in V$ zu finden, bei denen nur Zwischenknoten aus einer Teilmenge der Knotenmenge V verwendet werden dürfen. In jedem Schritt wird diese Teilmenge dann um einen Knoten vergrößert und mit Hilfe der tabellierten Pfade geprüft, ob sich mit Hilfe des neuen Knotens Pfade verkürzen lassen.

Zur Vereinfachung gehen wir davon aus, dass die Knotenmenge durchnumeriert ist, dass also $V = \{0, \ldots, n-1\}$ gilt. Wir suchen Pfade minimaler Länge, deren Zwischenknoten jeweils echt kleiner als ein gegebenes $k \in \mathbb{N}_0$ sind.

Definition 4.21 (k-Pfad) *Sei $k \in \mathbb{N}_0$. Wir nennen einen Pfad $(v_0, \ldots, v_n)$ einen k-Pfad, falls*

$$v_i < k \qquad \text{für alle } i \in \{1, \ldots, n-1\}$$

gilt, falls also jeder Zwischenknoten echt kleiner als k ist.

Der Floyd-Warshall-Algorithmus konstruiert zunächst 0-Pfade, auf denen keine Zwischenknoten auftreten können, so dass sie nur aus keiner oder einer einzelnen Kante

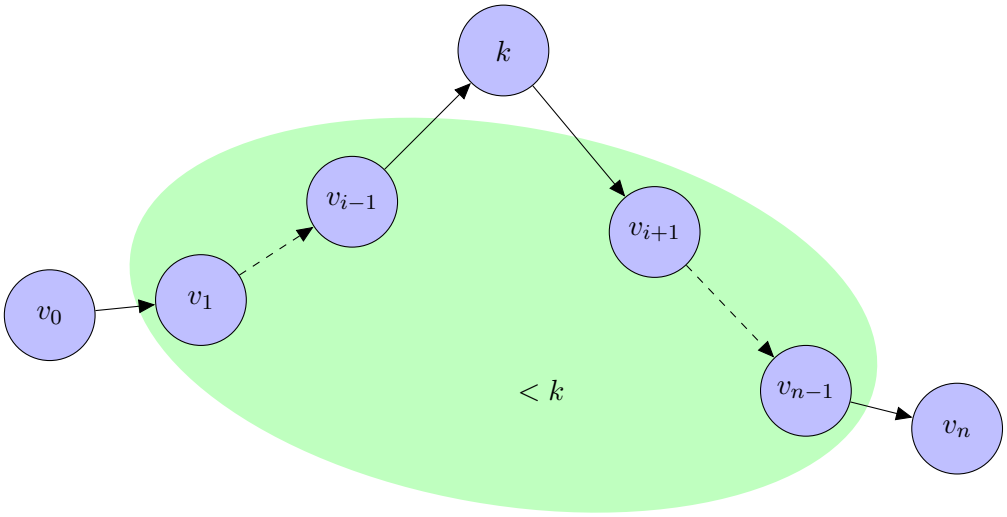


Abbildung 4.18: Ein $(k+1)$-Pfad kann den Knoten k nur höchstens einmal besuchen, alle anderen Zwischenknoten müssen echt kleiner als k sein.

bestehen können. Anschließend werden für $k \in \{0, \ldots, n-1\}$ jeweils aus k-Pfaden $(k+1)$-Pfade konstruiert, bis schließlich n-Pfade minimaler Länge vorliegen. Wegen $V = \{0, \ldots, n-1\}$ ist jeder Pfad auch ein n-Pfad, so dass der kürzeste unter den n-Pfaden auch der kürzeste Pfad unter allen Pfaden ist.

Die Längen der Pfade speichern wir für alle Anfangs- und Endknoten in Matrizen $D^{(k)} \in \mathbb{R}^{n \times n}$, die durch

$$d_{vw}^{(k)} := \min \left\{ \sum_{i=1}^{n} d(v_{i-1}, v_i) \;:\; (v_0, \ldots, v_n) \text{ ist } k\text{-Pfad von } v \text{ zu } w \right\}$$

definiert sind. Die Konstruktion beginnt mit $k = 0$: Da kein Knoten echt kleiner als null ist, kommen nur Pfade ohne Zwischenknoten in Frage, also nur solche der Länge null oder eins. Pfade der Länge null kommen nur für $v = w$ in Frage, Pfade der Länge eins entsprechen Kanten. Also erhalten wir

$$d_{vw}^{(0)} := \begin{cases} 0 & \text{falls } v = w, \\ d(v,w) & \text{falls } v \neq w \text{ und } (v,w) \in E, \\ \infty & \text{ansonsten} \end{cases} \qquad \text{für alle } v, w \in \{0, \ldots, n-1\}.$$

Um aus der Matrix $D^{(k)}$ die Matrix $D^{(k+1)}$ zu konstruieren, müssen wir über eine Möglichkeit verfügen, aus k-Pfaden $(k+1)$-Pfade zu konstruieren.

Die Idee der Konstruktion ist einfach: Ein $(k+1)$-Pfad von v zu w darf, da er insbesondere ein Pfad ist, den Knoten k höchstens einmal besuchen. Tut er es nicht, ist er auch ein k-Pfad, also aus dem vorigen Schritt schon bekannt. Besucht er dagegen den Knoten k, müssen ein k-Pfad von v zu k und ein k-Pfad von k zu w existieren, also

können wir dank Lemma 4.20 aus diesen beiden Pfaden einen $(k+1)$-Pfad von v zu w zusammensetzen, dessen Länge $d_{vk}^{(k)} + d_{kw}^{(k)}$ nicht überschreitet (vgl. Abbildung 4.18). Zusammengefasst ergibt sich die folgende Rekurrenzformel:

Satz 4.22 (Floyd-Warshall-Algorithmus) *Es gilt*

$$d_{vw}^{(k+1)} = \min\{d_{vw}^{(k)}, d_{vk}^{(k)} + d_{kw}^{(k)}\} \qquad \text{für alle } v, w, k \in \{0, \ldots, n-1\}. \tag{4.6}$$

Beweis. Seien $v, w, k \in \{1, \ldots, n\}$ gegeben.

Wir stellen zunächst fest, dass nach Definition jeder k-Pfad auch ein $(k+1)$-Pfad ist, so dass $d_{vw}^{(k+1)} \leq d_{vw}^{(k)}$ trivial gilt. Sei $(v_0, \ldots, v_n)$ ein k-Pfad minimaler Länge von v zu k. Sei $(w_0, \ldots, w_m)$ ein k-Pfad minimaler Länge von k zu w. Dann gilt $v_n = k = w_0$ und wir erhalten mit $(v_0, \ldots, v_n, w_1, \ldots, w_m)$ einen Kantenzug von v zu w, dessen Knoten echt kleiner als $k+1$ sind und dessen Länge $d_{vk}^{(k)} + d_{kw}^{(k)}$ beträgt. Mit Lemma 4.20 können wir daraus einen $(k+1)$-Pfad konstruieren, dessen Länge $d_{vk}^{(k)} + d_{kw}^{(k)}$ nicht überschreitet. Also folgt

$$d_{vw}^{(k+1)} \leq \min\{d_{vw}^{(k)}, d_{vk}^{(k)} + d_{kw}^{(k)}\}.$$

Zum Nachweis der entgegengesetzten Ungleichung wählen wir einen $(k+1)$-Pfad minimaler Länge $(v_0, \ldots, v_n)$ von v zu w. Da es sich um einen Pfad handelt, kann der Knoten k höchstens einmal als Zwischenknoten vorkommen. Falls er überhaupt nicht vorkommt, liegt ein k-Pfad vor. In diesem Fall gilt $d_{vw}^{(k+1)} = d_{vw}^{(k)}$.

Anderenfalls fixieren wir den Index $i \in \{1, \ldots, n-1\}$, für den $v_i = k$ gilt. Da kein anderer Zwischenknoten des $(k+1)$-Pfads gleich k sein kann, müssen $(v_0, \ldots, v_i)$ und $(v_i, \ldots, w)$ jeweils k-Pfade von v zu k und von k zu w sein. Ihre Längen können nicht geringer als die minimalen Längen $d_{vk}^{(k)}$ und $d_{kw}^{(k)}$ sein, also folgt $d_{vw}^{(k+1)} \geq d_{vk}^{(k)} + d_{kw}^{(k)}$.

Insgesamt erhalten wir

$$d_{vw}^{(k+1)} \geq \min\{d_{vw}^{(k)}, d_{vk}^{(k)} + d_{kw}^{(k)}\},$$

und damit auch die Gleichung (4.6). ∎

Natürlich wäre es von Interesse, auch die kürzesten Pfade zwischen zwei beliebigen Knoten konkret angeben zu können. Glücklicherweise lässt sich diese Aufgabe sehr elegant lösen, indem wir uns lediglich für jeden Eintrag $d_{vw}^{(k)}$ in einem Eintrag $z_{vw}^{(k)}$ merken, wie er entstanden ist: Wir setzen

$$z_{vw}^{(0)} = \begin{cases} -1 & \text{falls } (v,w) \in E, \\ -2 & \text{falls } v = w, \\ -3 & \text{ansonsten} \end{cases} \qquad \text{für alle } v, w \in V,$$

$$z_{vw}^{(k+1)} = \begin{cases} z_{vw}^{(k)} & \text{falls } d_{vw}^{(k+1)} = d_{vw}^{(k)}, \\ k & \text{falls } d_{vw}^{(k+1)} = d_{vk}^{(k)} + d_{kw}^{(k)} \end{cases} \qquad \text{für alle } v, w \in V.$$

Die erste Gleichung sorgt dafür, dass Pfade mit einer Kante durch -1 markiert werden, Pfade der Länge null mit -2, während aller anderen mit -3 als unbekannt gekennzeichnet

sind. Die zweite Gleichung stellt sicher, dass $z_{vw}^{(k+1)}$ jeweils die Nummer des größten Zwischenpunkts eines kürzesten Pfads von v zu w ist.

Bei dieser Vorgehensweise können wir die Pfade minimaler Länge rekursiv rekonstruieren: Der Pfad von v zu w hat keine Zwischenknoten, falls $z_{vw}^{(n)} < 0$ gilt. Ansonsten setzt er sich aus dem Pfad von v zu $z_{vw}^{(n)}$ und dem Pfad von $z_{vw}^{(n)}$ zu w zusammen. Wir brauchen also nur neben den Matrizen $D^{(k)}$ auch die Matrizen $Z^{(k)}$ zu speichern, um Pfadlängen und Pfade zu erhalten.

Der Algorithmus kann die Matrix $D^{(k)}$ jeweils unmittelbar mit der Matrix $D^{(k+1)}$ überschreiben, da mit $d_{kk}^{(k)} = 0$ auch

$$\begin{aligned} d_{vk}^{(k+1)} &= \min\{d_{vk}^{(k)}, d_{vk}^{(k)} + d_{kk}^{(k)}\} = d_{vk}^{(k)}, \\ d_{kw}^{(k+1)} &= \min\{d_{kw}^{(k)}, d_{kk}^{(k)} + d_{kw}^{(k)}\} = d_{kw}^{(k)} \qquad \text{für alle } v, w \in V \end{aligned}$$

gelten, so dass sich die k-te Zeile und Spalte der Matrix im k-ten Schritt nicht ändern.

Bemerkung 4.23 (Row-major-Darstellung) *In der Programmiersprache C ist es üblich, zweidimensionale Arrays wie die Matrizen $D^{(k)}$ und $Z^{(k)}$ in* row-major order *abzuspeichern, also jeweils die Zeilen fortlaufend in den Speicher zu schreiben. Beispielsweise wird eine Matrix $A \in \mathbb{R}^{3\times 4}$ durch ein Array* `a` *der Länge* 12 *dargestellt, dessen Einträge die Matrix wie folgt beschreiben:*

$$A = \begin{pmatrix} \texttt{a[0]} & \texttt{a[1]} & \texttt{a[2]} & \texttt{a[3]} \\ \texttt{a[4]} & \texttt{a[5]} & \texttt{a[6]} & \texttt{a[7]} \\ \texttt{a[8]} & \texttt{a[9]} & \texttt{a[10]} & \texttt{a[11]} \end{pmatrix}$$

Der Eintrag a_{vw} der Matrix $A \in \mathbb{R}^{n\times m}$ findet sich dann in dem Eintrag `a[v*m+w]` *des das Array darstellenden Arrays* `a` *der Länge nm.*

Die in Abbildung 4.19 dargestellte Implementierung des Floyd-Warshall-Algorithmus' füllt die Arrays `d` und `z` mit den Einträgen der Matrizen $D^{(n)}$ und $Z^{(n)}$, so dass sich anschließend die Längen der kürzesten Pfade leicht ablesen und diese Pfade in der beschriebenen Weise leicht konstruieren lassen.

Die bereits skizzierte Vorgehensweise für die Rekonstruktion der kürzesten Pfade aus der Matrix Z ist in Abbildung 4.20 illustriert: Wenn wir den Pfad von v zu w konstruieren wollen, können vier Situationen auftreten: Erstens kann es passieren, dass kein Pfad existiert. Dann gilt $z_{vw} = -3$.

Zweitens kann der kürzeste Pfad für $v = w$ gerade (v) sein. Dann gilt $z_{vw} = -2$.

Drittens kann der kürzeste Pfad eine Kante $(v, w) \in E$ sein. Dann haben wir $z_{vw} = -1$ und geben den Endknoten w aus.

Viertens kann ein Zwischenknoten $k := z_{vw}$ verwendet werden. Dann müssen wir zunächst den Pfad von v zu k und anschließend den von k zu w ausgeben. Diese Aufgabe lösen wir am einfachsten per Rekursion.

Dabei besteht nicht die Gefahr, dass es zu einer endlosen Rekursion kommt: Nach Konstruktion ist z_{vw} immer der Zwischenknoten maximaler Größe. Da wir einen Pfad

```
1  void
2  shortest_paths(const graph *g, float *d, int *z)
3  {
4    edge *e;
5    float dk;
6    int v, k, w;
7    int n;
8    n = g->vertices;
9    for(v=0; v<n; v++)
10     for(w=0; w<n; w++) {
11       d[v*n+w] = INFINITY; z[v*n+w] = -3;
12     }
13   for(v=0; v<n; v++) {
14     d[v*n+v] = 0.0; z[v*n+v] = -2;
15     for(e=g->edgelist[v]; e; e=e->next) {
16       w = e->to; dk = e->weight;
17       if(dk < d[v*n+w]) {
18         d[v*n+w] = dk; z[v*n+w] = -1;
19       }
20     }
21   }
22   for(k=0; k<n; k++)
23     for(v=0; v<n; v++)
24       for(w=0; w<n; w++) {
25         dk = d[v*n+k] + d[k*n+w];
26         if(dk < d[v*n+w]) {
27           d[v*n+w] = dk; z[v*n+w] = k;
28         }
29       }
30 }
```

Abbildung 4.19: Floyd-Warshall-Algorithmus für die Berechnung sämtlicher kürzester Pfade in einem gewichteten Graphen.

konstruieren, kann er auch nur einmal besucht werden, also müssen alle anderen Zwischenknoten echt kleiner als z_{vw} sein. Da es in V keine endlos echt absteigende Folge gibt, kann auch die Rekursion nicht endlos aufgerufen werden.

Unsere Implementierung ist so geschrieben, dass `print_path_recursion` alle Knoten mit Ausnahme des ersten ausgibt. Für dessen Ausgabe ist `print_path` zuständig. Dank dieser Vorgehensweise können die Knoten in der üblichen Form einer durch Kommas getrennten Folge ausgegeben werden.

Bemerkung 4.24 (Rechenaufwand) *Entscheidend für den Rechenaufwand sind die*

```
1  static void
2  print_path_recursion(int ldz, const int *z, int v, int w)
3  {
4    int k;
5    k = z[v*ldz+w];
6    if(k < -2)
7      crash("No path");
8    else if(k < -1)
9      ;
10   else if(k < 0)
11     printf(", %d", w);
12   else {
13     print_path_recursion(ldz, z, v, k);
14     print_path_recursion(ldz, z, k, w);
15   }
16 }
17 void
18 print_path(const graph *g, const int *z, int v, int w)
19 {
20   printf("%d", v);
21   print_path_recursion(g->vertices, z, v, w);
22   printf("\n");
23 }
```

Abbildung 4.20: Rekonstruktion der kürzesten Pfade aus der durch den Floyd-Warshall-Algorithmus konstruierten Matrix Z.

drei geschachtelten Schleifen in den Zeilen 19 bis 26, die jeweils über alle Knoten laufen, also $\Theta(|V|^3)$ *Operationen erfordern.*

Die geschachtelten Schleifen in den Zeilen 9 bis 12 benötigen nur $\mathcal{O}(|V|^2)$ *Operationen, die in den Zeilen 13 bis 18 nur* $\mathcal{O}(|E| + |V|)$*, sind also wegen* $|E| \leq |V|^2$ *gegenüber den erstgenannten vernachlässigbar.*

Damit erfordert der Gesamtalgorithmus $\Theta(|V|^3)$ *Operationen.*

5 Konstruktionsprinzipien

In den vorangehenden Kapiteln haben wir eine Reihe von Algorithmen für wichtige Problemklassen kennen gelernt. Diese Algorithmen beruhen auf allgemeinen Konstruktionsprinzipien, die uns dabei helfen können, auch für neue Aufgaben neue Algorithmen zu konstruieren. In diesem Kapitel werden einige Beispiele für diese Prinzipien vorgestellt.

5.1 Teile und herrsche: Karatsuba-Multiplikation

Sowohl der Mergesort- als auch der Quicksort-Algorithmus beruhen auf dem Prinzip „teile und herrsche“: Eine zu lösende Aufgabe wird in kleinere Teilaufgaben zerlegt, und die Teilaufgaben werden rekursiv gelöst. Aus den Lösungen der Teilaufgaben wird dann die Lösung der Gesamtaufgabe rekonstruiert.

Ein weiteres Beispiel für diese Vorgehensweise ist der *Karatsuba-Algorithmus* für die Multiplikation n-stelliger Zahlen: Für kleine Werte von n können wir die Multiplikation direkt ausführen, interessant ist für uns nur der Fall $n \geq 2$. Wir bezeichnen die Basis der Zahldarstellung mit $b \in \mathbb{N}_{\geq 2}$ und gehen davon aus, dass zwei Zahlen $x, y \in \mathbb{N}_0$ durch ihre Ziffern $x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1} \in \{0, \ldots, b-1\}$ nach dem üblichen Stellenwertsystem dargestellt sind:

$$x = x_{n-1}b^{n-1} + x_{n-2}b^{n-1} + \ldots + x_1 b + x_0,$$
$$y = y_{n-1}b^{n-1} + y_{n-2}b^{n-2} + \ldots + y_1 b + y_0.$$

Wir setzen $m := \lceil n/2 \rceil$ und zerlegen x und y in die ersten $n-m$ und die letzten m Stellen:

$$x_{\mathrm{lo}} := x_{m-1}b^{m-1} + \ldots + x_1 b + x_0, \qquad x_{\mathrm{hi}} := x_{n-1}b^{n-m-1} + \ldots + x_{m+1}b + x_m,$$
$$y_{\mathrm{lo}} := y_{m-1}b^{m-1} + \ldots + y_1 b + y_0, \qquad y_{\mathrm{hi}} := y_{n-1}b^{n-m-1} + \ldots + y_{m+1}b + y_m.$$

Dann gelten die Gleichungen

$$x = x_{\mathrm{hi}}b^m + x_{\mathrm{lo}}, \qquad y = y_{\mathrm{hi}}b^m + y_{\mathrm{lo}},$$

so dass das Produkt der beiden Zahlen durch

$$z := xy = (x_{\mathrm{hi}}b^m + x_{\mathrm{lo}})(y_{\mathrm{hi}}b^m + y_{\mathrm{lo}}) = x_{\mathrm{hi}}y_{\mathrm{hi}}b^{2m} + (x_{\mathrm{hi}}y_{\mathrm{lo}} + x_{\mathrm{lo}}y_{\mathrm{hi}})b^m + x_{\mathrm{lo}}y_{\mathrm{lo}}$$

gegeben ist. Wir müssen also lediglich die drei Zahlen

$$\alpha := x_{\mathrm{hi}}y_{\mathrm{hi}}, \qquad \beta := x_{\mathrm{hi}}y_{\mathrm{lo}} + x_{\mathrm{lo}}y_{\mathrm{hi}}, \qquad \gamma := x_{\mathrm{lo}}y_{\mathrm{lo}}$$

berechnen und geeignet aufaddieren. Auf den ersten Blick benötigen wir dafür vier Multiplikationen von höchstens m-stelligen Zahlen sowie eine Addition.

Auf den zweiten Blick sehen wir, dass

$$\alpha + \beta + \gamma = x_{\text{hi}}y_{\text{hi}} + x_{\text{hi}}y_{\text{lo}} + x_{\text{lo}}y_{\text{hi}} + x_{\text{lo}}y_{\text{lo}} = (x_{\text{hi}} + x_{\text{lo}})(y_{\text{hi}} + y_{\text{lo}})$$

gilt, so dass wir β auch mit nur einer einzigen Multiplikation, zwei Additionen und zwei Subtraktionen berechnen können:

$$\beta = (x_{\text{hi}} + x_{\text{lo}})(y_{\text{hi}} + y_{\text{lo}}) - \alpha - \gamma.$$

Insgesamt erhalten wir also drei Multiplikationen, vier Additionen und zwei Subtraktionen, nämlich

$$\alpha := x_{\text{hi}}y_{\text{hi}}, \qquad \gamma := x_{\text{lo}}y_{\text{lo}}, \qquad \beta := (x_{\text{hi}} + x_{\text{lo}})(y_{\text{lo}} + y_{\text{hi}}) - \alpha - \gamma, \tag{5.1a}$$

$$z := \alpha b^{2m} + \beta b^m + \gamma. \tag{5.1b}$$

Die Multiplikationen mit b^{2m} und b^m zählen wir nicht mit, da sie lediglich einer Verschiebung der Ziffern um $2m$ beziehungsweise m Stellen nach links entsprechen.

Die in (5.1) auftretenden drei Multiplikationen von nur noch m-stelligen Zahlen können wir wieder per Rekursion behandeln und gelangen so zu einem Algorithmus für die Multiplikation n-stelliger Zahlen. Dieser Algorithmus wird als die *Karatsuba-Multiplikation* (nach A. A. Karatsuba) bezeichnet. Wir werden nun nachweisen, dass er für große Zahlen wesentlich schneller als die konventionelle Multiplikation sein kann.

Wir untersuchen den resultierenden Rechenaufwand lediglich für den Fall, dass $n = 2^p$ gilt, dass n also eine Zweierpotenz ist. Für Additionen und Subtraktionen genügen $\mathcal{O}(n)$ Operationen, so dass wir mit Lemma 2.22 eine Konstante $\widehat{C} \in \mathbb{R}_{>0}$ so finden, dass der Rechenaufwand $R(n)$ für die Karatsuba-Multiplikation zweier n-stelliger Zahlen durch die Rekurrenzformel

$$R(n) \leq \begin{cases} \widehat{C} & \text{falls } n = 1, \\ \widehat{C}n + 3R(n/2) & \text{ansonsten} \end{cases} \qquad \text{für alle } n = 2^p \text{ mit } p \in \mathbb{N}_0$$

gegeben ist. Diese Rekurrenzformel können wir wie folgt auflösen:

Lemma 5.1 (Rekurrenz) *Seien $\alpha, \beta \in \mathbb{N}_0$ und $\gamma \in \mathbb{R}_{>2}$ gegeben und sei $f : \mathbb{N} \to \mathbb{N}_0$ eine Abbildung mit*

$$f(n) \leq \begin{cases} \alpha & \text{falls } n = 1, \\ \beta n + \gamma f(n/2) & \text{ansonsten} \end{cases} \qquad \text{für alle } n = 2^\ell \text{ mit } \ell \in \mathbb{N}_0. \tag{5.2}$$

Dann gilt mit

$$b := \frac{2\beta}{\gamma - 2}, \qquad a := \alpha + b$$

die Abschätzung

$$f(n) \leq an^{\log_2(\gamma)} - bn \qquad \text{für alle } n = 2^\ell \text{ mit } \ell \in \mathbb{N}_0. \tag{5.3}$$

Beweis. Wir beweisen die Aussage per Induktion über $\ell \in \mathbb{N}_0$.

Induktionsanfang. Sei $\ell = 0$. Dann gilt $n = 2^\ell = 1$ und nach (5.2) folgt

$$f(1) \leq \alpha = a - b = an^{\log_2(\gamma)} - bn.$$

Induktionsvoraussetzung. Sei $\ell \in \mathbb{N}_0$ so gegeben, dass (5.3) für $n = 2^\ell$ gilt.

Induktionsschritt. Sei $n = 2^{\ell+1}$. Nach (5.2) gilt wegen $n \geq 2$ dann

$$f(n) \leq \beta n + \gamma f(n/2)$$

und mit der Induktionsvoraussetzung folgt

$$\begin{aligned} f(n) &\leq \beta n + \gamma(an^{\log_2(\gamma)} - bn) = \beta n + \gamma a(n/2)^{\log_2(\gamma)} - \gamma b(n/2) \\ &= \gamma a(n/2)^{\log_2(\gamma)} - (\gamma b/2 - \beta)n. \end{aligned}$$

Für den ersten Term nutzen wir

$$\gamma(n/2)^{\log_2(\gamma)} = \gamma(1/2)^{\log_2(\gamma)} n^{\log_2(\gamma)} = \gamma \frac{1}{2^{\log_2(\gamma)}} n^{\log_2(\gamma)} = \gamma \frac{1}{\gamma} n^{\log_2(\gamma)} = n^{\log_2(\gamma)}.$$

Für den zweiten Term ergibt sich

$$\gamma b/2 - \beta = \frac{\beta\gamma}{\gamma - 2} - \beta = \frac{\beta\gamma}{\gamma - 2} - \frac{\beta(\gamma - 2)}{\gamma - 2} = \frac{\beta\gamma - \beta\gamma + 2\beta}{\gamma - 2} = \frac{2\beta}{\gamma - 2} = b,$$

so dass wir insgesamt

$$f(n) \leq an^{\log_2(\gamma)} - bn$$

bewiesen haben, und damit die Induktionsbehauptung. ∎

Damit lässt sich der Rechenaufwand für die Karatsuba-Multiplikation durch $R(n) \leq \widehat{C}(3n^{\log_2(3)} - 2n)$ beschränken. Bemerkenswert an dieser Abschätzung ist, dass der Exponent $\log_2(3) \approx 1,58496$ deutlich kleiner als 2 ist, so dass der rekursive Algorithmus für große Zahlen wesentlich effizienter als der aus der Schule bekannte arbeiten kann.

5.2 Tiefensuche: Sudoku

Viele Aufgaben lassen sich durch Graphen ausdrücken und dann mit graphentheoretischen Algorithmen lösen. Als Beispiel untersuchen wir das *Sudoku-Puzzle*. Wir bezeichnen mit $D := \{1, \ldots, 9\}$ die Menge der natürlichen Zahlen von 1 bis 9 und suchen nach einer 9×9-Matrix $S \in D^{9\times 9}$ derart, dass in jeder Zeile, jeder Spalte und jeder der in

$$S = \left(\begin{array}{ccc|ccc|ccc} s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} & s_{17} & s_{18} & s_{19} \\ s_{21} & s_{22} & s_{23} & s_{24} & s_{25} & s_{26} & s_{27} & s_{28} & s_{29} \\ s_{31} & s_{32} & s_{33} & s_{34} & s_{35} & s_{36} & s_{37} & s_{38} & s_{39} \\ \hline s_{41} & s_{42} & s_{43} & s_{44} & s_{45} & s_{46} & s_{47} & s_{48} & s_{49} \\ s_{51} & s_{52} & s_{53} & s_{54} & s_{55} & s_{56} & s_{57} & s_{58} & s_{59} \\ s_{61} & s_{62} & s_{63} & s_{64} & s_{65} & s_{66} & s_{67} & s_{68} & s_{69} \\ \hline s_{71} & s_{72} & s_{73} & s_{74} & s_{75} & s_{76} & s_{77} & s_{78} & s_{79} \\ s_{81} & s_{82} & s_{83} & s_{84} & s_{85} & s_{86} & s_{87} & s_{88} & s_{89} \\ s_{91} & s_{92} & s_{93} & s_{94} & s_{95} & s_{96} & s_{97} & s_{98} & s_{99} \end{array}\right)$$

markierten 3×3-Teilmatrizen jede Zahl genau einmal auftritt.

Mathematisch lässt sich diese Bedingung kompakt durch die Mengen beschreiben, in denen die Zahlen nur einmal auftreten dürfen:

$$\begin{aligned}
M_1 &:= \{(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9)\},\\
M_2 &:= \{(2,1),(2,2),(2,3),(2,4),(2,5),(2,6),(2,7),(2,8),(2,9)\},\\
&\vdots\\
M_9 &:= \{(9,1),(9,2),(9,3),(9,4),(9,5),(9,6),(9,7),(9,8),(9,9)\},\\
M_{10} &:= \{(1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(7,1),(8,1),(9,1)\},\\
&\vdots\\
M_{18} &:= \{(1,9),(2,9),(3,9),(4,9),(5,9),(6,9),(7,9),(8,9),(9,9)\},\\
M_{19} &:= \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\},\\
&\vdots\\
M_{27} &:= \{(7,7),(7,8),(7,9),(8,7),(8,8),(8,9),(9,7),(9,8),(9,9)\}.
\end{aligned}$$

Dabei beschreiben die Mengen $M_1, \ldots, M_9$ die neun Zeilen, die Mengen $M_{10}, \ldots, M_{18}$ die neun Spalten und die Mengen $M_{19}, \ldots, M_{27}$ die neun 3×3-Blöcke.

Eine Matrix $S \in D^{9\times 9}$ ist eine korrekte Lösung, falls

$$|\{(i,j) \in M_p \ : \ s_{ij} = a\}| = 1 \qquad \text{für alle } p \in \{1,\ldots,27\},\ a \in D$$

gilt. Die Aufgabenstellung eines Sudoku-Puzzles besteht aus einer „unvollständigen“ Matrix, in der einige Eintäge fehlen, die durch den Spieler ergänzt werden müssen. Wenn wir fehlende Einträge durch die Null darstellen, ist eine solche unvollständige Matrix ein Element der Menge $D_0^{9\times 9}$ mit $D_0 := D \cup \{0\}$. Damit sie sich ergänzen lässt, muss mindestens

$$|\{(i,j) \in M_p \ : \ s_{ij} = a\}| \leq 1 \qquad \text{für alle } p \in \{1,\ldots,27\},\ a \in D \tag{5.4}$$

gelten. Ein Spielzug besteht darin, einen der Nulleinträge durch eine Zahl aus D zu ersetzen: Die neue Matrix $S' \in D_0^{9\times 9}$ geht aus der alten hervor, wenn $k, \ell \in \{1,\ldots,9\}$ so existieren, dass

$$s_{k\ell} = 0, \qquad s'_{k\ell} \neq 0, \tag{5.5a}$$

$$s_{ij} = s'_{ij} \qquad \text{für alle } i,j \in \{1,\ldots,9\} \text{ mit } (i,j) \neq (k,\ell) \tag{5.5b}$$

gelten, dass also genau eine Null durch etwas anderes ersetzt wurde und alle sonstigen Einträge unverändert geblieben sind.

Wir können unsere Aufgabe mit Hilfe eines Graphen formulieren: Jeder Zustand der Matrix ist ein Knoten, jeder mögliche Spielzug ist eine Kante. Um ein gegebenes Puzzle zu lösen, müssen wir lediglich einen Kantenzug von dem Knoten, der den Ausgangszustand repräsentiert, zu einem Knoten finden, der einer Matrix ohne Nulleinträge entspricht. Da die einzelnen Kanten Spielzüge darstellen, entspricht nämlich ein Kantenzug einer Folge von Spielzügen, die uns von der Ausgangsmatrix zu dem Ergebnis führen.

Mathematisch präzise können wir den Graphen wie folgt beschreiben:

$$V := \{S \in D_0^{9\times 9} \ : \ S \text{ erfüllt } (5.4)\},$$
$$E := \{(S, S') \ : \ S, S' \in D_0^{9\times 9} \text{ und es existieren } k, \ell \in \{1, \ldots, 9\} \text{ mit } (5.5)\},$$

denn dann leistet $G := (V, E)$ gerade das Gewünschte.

Wenn uns ein *sehr* großer Computer zur Verfügung stünde, der die sehr vielen Knoten und Kanten dieses Graphen speichern kann, könnten wir die bereits beschriebenen Algorithmen verwenden, beispielsweise Breiten- oder Tiefensuche, um die relevanten Kantenzüge zu finden.

In der Praxis ist es eher ratsam, den Graphen nur *implizit* zu verwenden: Wir verwenden die Beschreibung (5.5), um herauszufinden, zwischen welchen Knoten Kanten bestehen, und wir speichern nur diejenigen Knoten, die der Suchalgorithmus jeweils benötigt.

Auf die Arrays `visited` und `pred` müssen wir bei diesem Zugang verzichten, da nicht genug Speicher für sie zur Verfügung steht. Bei `pred` ist das unproblematisch, weil wir den Kantenzug anders rekonstruieren können. Bei `visited` dagegen könnten Schwierigkeiten auftreten, weil wir dieses Array für die Erkennung bereits besuchter Knoten verwenden, also für die Vermeidung von Endlosschleifen.

In unserem konkreten Beispiel werden Endlosschleifen allerdings durch die Aufgabenstellung ausgeschlossen: Mit jedem Spielzug reduziert sich die Anzahl der Nulleinträge der Matrix um eins, also kann in keinem Kantenzug ein Knoten doppelt auftreten. Damit ist jeder Kantenzug ein Pfad, und da eine Matrix höchstens 81 Nulleinträge aufweisen kann, gibt es auch keine Pfade mit mehr als 81 Knoten.

Diese Eigenschaft legt es nahe, die Tiefensuche zur Konstruktion der Lösung zu verwenden: Da kein Pfad mit mehr als 81 Knoten existiert, sollte ein Kellerspeicher mit 81 Elementen ausreichen, und der Speicherbedarf von 81 Matrizen mit 9×9 Einträgen ist relativ gering.

5.3 Dynamisches Programmieren: Rucksack-Problem

Eine berühmte Fragestellung aus dem Bereich der Optimierung ist das *Rucksack-Problem*, das wir hier in einer besonders einfachen Variante untersuchen: Gegeben sind eine Anzahl von Gegenständen, die jeweils ein Gewicht und einen Wert haben. Unsere Aufgabe besteht darin, einen Rucksack so mit Gegenständen zu füllen, dass sein Inhalt einen möglichst hohen Gesamtwert hat, aber ein gegebenes Maximalgewicht nicht übersteigt.

Wir übersetzen die Aufgabenstellung in die Sprache der Mathematik, indem wir mit n die Anzahl der Gegenstände bezeichnen und die Gegenstände von 0 bis $n-1$ durchnumerieren. Dann soll $w_i \in \mathbb{N}_0$ das Gewicht des Gegenstands i bezeichnen und $c_i \in \mathbb{R}$ seinen Wert. Das maximale Gewicht, das der Rucksack aushalten kann, nennen wir $g \in \mathbb{N}$

Eine Auswahl von Gegenständen, die in den Rucksack gesteckt werden können, beschreiben wir durch eine Menge $R \subseteq \{0, \ldots, n-1\}$. $i \in R$ soll bedeuten, dass das Objekt

i im Rucksack ist. Gesamtwert und Gesamtgewicht sind dann durch

$$C(R) := \sum_{i \in R} c_i, \qquad W(R) := \sum_{i \in R} w_i$$

gegeben. Gesucht ist ein $R \subseteq \{0, \ldots, n-1\}$, für das der Wert maximal wird, ohne das maximale Gewicht zu überschreiten, es sollen also die Ungleichungen

$$\begin{aligned} &W(R) \leq g, \\ &C(R) \geq C(R') \qquad \text{für alle } R' \subseteq \{0, \ldots, n-1\} \text{ mit } W(R') \leq g \end{aligned}$$

gelten.

Für die Lösung dieser Aufgabe verwenden wir den Ansatz der dynamischen Programmierung: Wir bezeichnen mit $a_m^{(k)}$ den maximalen Wert, den ein Rucksack annehmen kann, der nur Gegenstände mit Nummern echt kleiner als k enthalten und dessen Gewicht m nicht überschreiten darf:

$$\begin{aligned} a_m^{(k)} := \max\{C(R) \;:\; R \subseteq \{0, \ldots, k-1\},\ W(R) \leq m\} \\ \text{für alle } k \in \{0, \ldots, n\},\ m \in \{0, \ldots, g\}, \end{aligned}$$

wobei für $k = 0$ auf der rechten Seite nur $R = \emptyset$ auftreten kann. Dann ist $a_g^{(n)}$ der Wert der Lösung unseres Optimierungsproblems.

Die Werte $a_m^{(k)}$ konstruieren wir induktiv:

Lemma 5.2 (Rucksack) *Es gelten*

$$\begin{aligned} a_m^{(0)} &= 0 && \text{für alle } m \in \{0, \ldots, g\}, \\ a_m^{(k+1)} &= \max\{a_m^{(k)}, c_k + a_{m-w_k}^{(k)}\} && \text{für alle } k \in \{0, \ldots, n-1\},\ m \in \{0, \ldots, g\}. \end{aligned}$$

Beweis. Da es keine Gegenstände $i \in \{0, \ldots, n-1\}$ mit $i < 0$ gilt, folgt $a_m^{(0)} = 0$ für alle $m \in \{0, \ldots, g\}$.

Seien nun $k \in \{0, \ldots, n-1\}$ und $m \in \{0, \ldots, g\}$ gegeben. Wir beweisen zunächst

$$a_m^{(k+1)} \geq \max\{a_m^{(k)}, c_k + a_{m-w_k}^{(k)}\}. \tag{5.6}$$

Wegen $\{0, \ldots, k-1\} \subseteq \{0, \ldots, k\}$ folgt unmittelbar $a_m^{(k+1)} \geq a_m^{(k)}$. Sei $R \subseteq \{0, \ldots, k-1\}$ mir $C(R) = a_{m-w_k}^{(k)}$ und $W(R) \leq m - w_k$ gegeben. Dann erfüllt $R_+ := R \cup \{k\}$ sowohl $C(R_+) = c_k + a_{m-w_k}^{(k)}$ als auch $W(R_+) \leq m - w_k + w_k = m$. Also folgt $a_m^{(k+1)} \geq c_k + a_{m-w_k}^{(k)}$ und wir haben (5.6) bewiesen.

Nun beweisen wir

$$a_m^{(k+1)} \leq \max\{a_m^{(k)}, c_k + a_{m-w_k}^{(k)}\}. \tag{5.7}$$

Sei dazu $R \subseteq \{0, \ldots, k\}$ mit $C(R) = a_m^{(k+1)}$ und $W(R) \leq m$ gegeben.

Falls $k \notin R$ gilt, folgt unmittelbar $C(R) \leq a_m^{(k)}$.

Anderenfalls können wir $R_- := R \setminus \{k\}$ untersuchen. Es gelten $W(R_-) \leq m - w_k$ und $R' \subseteq \{0, \ldots, k-1\}$, also folgt $C(R_-) \leq a_{m-w_k}^{(k)}$. Damit erhalten wir $C(R) = c_k + C(R_-) \leq c_k + a_{m-w_k}^{(k)}$.

Also ist (5.7) bewiesen, und damit auch das Lemma. ∎

Index

Algorithmus, 5
AVL-Baum, 75
    Definition, 75

Baum
    als Array, 89
    AVL-Baum, 75
    binär, 68
    Höhe, 68
    Halde, 92
    Rotation, 78
    Suchbaum, 72
Breitensuche, 107
Bucketsort, 58

Datenstruktur
    Baum, 65
    doppelt verkettete Liste, 59
    einfach verkettete Liste, 53
    Keller, 60
    Warteschlange, 62
Destruktor, 55
Dijkstra-Algorithmus, 117
dynamisches Programmieren, 130

FIFO, 62
Floyd-Warshall-Algorithmus, 129

Gauß-Klammer, 9
Graph, 101
    Breitensuche, 107
    Dijkstra-Algorithmus, 117
    Floyd-Warshall-Algorithmus, 129
    Kantenzug, 104
    Kreis, 105
    Pfad, 104
    Teilgraph, 105
    Tiefensuche, 110
    ungerichtet, 101
    zusammenhängend, 105

Halde, 92
Heap, 92
Heapsort, 95

Insertionsort, 15
Iteration, 11

Kanten, 101
Kantenzug, 104
Karatsuba-Multiplikation, 137
Keller, 60
    als Array, 86
Knoten, 101
Konstruktor, 54
Korrektheit
    partiell, 46
    total, 46
Kreis, 105

Landau-Notation, 36
lexikographische Ordnung, 44
LIFO, 60
Liste
    doppelt verkettet, 59
    einfach verkettet, 53
    Kopf, 54
Logarithmus
    dyadisch, 12
    natürlich, 33

Mergesort, 19

Nachbedingung, 46
Natürlicher Logarithmus, 33

Ordnung, 41

partielle Korrektheit, 46
partielle Ordnung, 41
Permutation, 14
Pfad, 104

Queue, 62
Quicksort, 27

Radixsort, 58
Rekursion, 19
Relation, 40
    antisymmetrisch, 41
    reflexiv, 40
    total, 41
    transitiv, 41

Sortieren
    Bucketsort, 58
    Heapsort, 95
    Insertionsort, 15
    Mergesort, 19
    Quicksort, 27
    Radixsort, 58
    stabil, 58
Stack, 60
Strukturelle Induktion, 71
Suchbaum, 65, 72
Suche
    binär, 9
    linear, 7
Sudoku, 139

Teilgraph, 105
Tiefensuche, 110
Topologisches Sortieren, 109
totale Korrektheit, 46

Ungerichteter Graph, 101

Vorbedingung, 46

Warteschlange, 62
    als Array, 87

Zusammenhängender Graph, 105
Zusammenhangskomponente, 105