Technische Universität Braunschweig
Institut für Betriebssysteme und Rechnerverbund
Abteilung Algorithmik

Skript

Algorithmen und Datenstrukturen

Braunschweig, den 25. Februar 2021

Inhaltsverzeichnis

Vorwort

Die Vorlesung *Algorithmen und Datenstrukturen* wendet sich an Studierende am Beginn des Studiums. Für die Fächer Informatik und Wirtschaftsinformatik ist sie eine Pflichtveranstaltung im ersten Semester, für einige andere Fächer kommt sie meist nicht viel später. Entsprechend haben wir uns bemüht, den Inhalt möglichst vielfältig zu gestalten, so dass auch bei unterschiedlichen Voraussetzungen verschiedenste Anregungen geboten werden. Dabei geschieht der Zugang jeweils problemorientiert: Ziel ist es, dass man versteht, warum eine Frage, ein Thema oder eine Struktur interessant ist, bevor die Dinge genauer formalisiert werden, und ein Thema jeweils schrittweise weiterentwickelt, zum Teil mit passenden Exkursen. Dazu gehört oft auch ein Blick auf die Personen, auf die Ideen zurückgehen – Informatik ist keineswegs eine abstrakte und unpersönliche Wissenschaft.

Dieses Skript soll bei der Teilnahme lediglich eine Hilfestellung geben. Es kann immer nur ein sehr schlechter Ersatz für die Teilnahme an den Präsenzveranstaltungen sein, die durch die aktive Interaktion mit den Anwesenden ein wesentlich lebendigeres und differenzierteres Bild vermitteln kann. Entsprechend fehlen oft die besonders wichtigen freien und spontanen Zwischentexte, Querverbindungen, Akzente, die in Person ganz anders vermittelt werden können. Eine Vorlesung kann nur Anregungen und Ansatzpunkte zum Studium eines Themas liefern, das erst durch eigene Beschäftigung lebendig wird. Genauso kann ein Skript immer nur Ausschnitte und Eckpunkte einer Vorlesung liefern.

Der Inhalt der Vorlesung wurde über sieben Wintersemester hinweg entwickelt und angepasst. Trotzdem sind einige Aspekte immer im Fluss. Das gilt auch für dieses Skript, das sich im Verlaufe des Semesters und nicht zuletzt durch Feedback von den Teilnehmern stetig entwickeln wird; insofern haben wir weder den Anspruch auf Vollständigkeit, noch auf präzise Übereinstimmung mit den tatsächlich präsentierten Inhalten.

Besonderer Dank gilt Ilonka Pingel, die viel Arbeit investiert hat, um die bislang verschiedenen Medienbestandteile in einem einheitlichen Schriftstück zusammenzuführen.

Viel Spaß und Erfolg!

Sándor Fekete, Wintersemester 2014/15

1 Einleitung

Bevor wir uns im folgenden Kapitel mit den ersten konkreten algorithmischen Fragstellungen beschäftigen, einige einführende Worte zum Begriff „*Algorithmus*“!

1.1 Was ist ein Algorithmus?

Betrachten wir zunächst die Definition des Begriffs *Algorithmus* frei nach Wikipedia.

> Ein *Algorithmus* ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen [4].

Beispiele für solche Algorithmen sind Kochrezepte und Bedienungsanleitungen sowie Notenblätter oder ein Programmablaufplan. Obige Definition bedient sich jedoch der Begriffe *Problem*, *Klasse eines Problems* und *Lösung eines Problems*. Die Bedeutungen dieser Ausdrücke sind jedoch vage und müssen daher präzisiert werden.

Beispiel 1.1 (GGT)**.** *Gegeben ist das* konkrete Problem*:*

> *„Bestimme den größten gemeinsamen Teiler von 144 und 729.“*

Die Lösung *hierzu ist* 9*.*
Verallgemeinert man das Problem, so ergibt sich die Klasse des Problems*:*

> *„Bestimme den größten gemeinsamen Teiler von x und y.“*

Um diese Aufgabe zu lösen, bedarf es einer Anleitung, wie dafür vorzugehen ist. Diese Anleitung liefert der Euklidische Algorithmus.

1.2 Wie formalisiert man einen Algorithmus?

Einen Algorithmus zu formulieren, bedeutet im Allgemeinen, dass beliebige Berechnungen zu abstrakten Formalisierungen werden. Hierfür sind mathematische Modelle sehr hilfreich. Eines der wichtigsten mathematischen Rechnermodelle der Theoretischen Informatik ist die Turingmaschine, die nach dem Mathematiker Alan Turing benannt ist und von ihm 1936 eingeführt wurde [6].[1]

Ein Algorithmus oder ein Programm kann durch eine *Turingmaschine* repräsentiert werden. Dies macht Algorithmen und Programme einfach analysierbar, da es sich bei der

[1] Nach Alan Turing ist auch der Zugang auf der östlichen Seite des Informatikzentrums der TU Braunschweig benannt: die Alan-Turing-Allee.

zugehörigen Turingmaschine um ein mathematisches Objekt handelt, welches also mit mathematischen Methoden untersucht werden kann. Eine Turingmaschine besteht aus unterschiedlichen Komponenten. Es gibt das unendlich lange *(Speicher-)Band*, das aus unendlich vielen Feldern besteht. Diese Felder können lediglich sequentiell angesprochen werden und speichern jeweils genau ein Zeichen. Ein Zeichen muss aus dem sogenannten *Eingabealphabet* kommen, da die Turingmaschine nur Zeichen dieses Alphabetes verarbeiten kann. Eine Ausnahme hiervon bildet das Leersymbol, das einfach einem leeren Feld entspricht und nicht zum Eingabealphabet gehört. Weiterhin gibt es einen Lese- und Schreibkopf, der zur Ansteuerung der Felder benutzt wird und diese auslesen und verändern kann. Die Bewegungen des Lese-/Schreibkopfes und die Veränderungen, die auf den angesteuerten Feldern vorgenommen werden, werden vom Programm bzw. Algorithmus vorgegeben, der durch diese Turingmaschine modelliert wird [6].

Man kann mehrere aufeinanderfolgende Symbole unterschiedlich interpretieren, z. B. als Zahlen. Mithilfe solcher Interpretationen kann man Turingmaschinen also als Funktionen verstehen, die als Eingabe eine Zeichenkette auf dem Band erhalten und als Ausgabe eine (neue) Zeichenkette auf demselben Band produzieren. Die Funktionen, die durch ebensolche Turingmaschinen beschreibbar sind, nennt man auch *Turing-berechenbar* [6].

Algorithmen, die durch Turingmaschinen modellierbar sind, erfüllen damit die zentralen Eigenschaften aus dem nachfolgenden Abschnitt 1.3.

1.3 Eigenschaften von Algorithmen

Ein Algorithmus besitzt die folgenden Eigenschaften.

(1) *Finitheit*: Das Verfahren muss in einem endlichen Text eindeutig beschreibbar sein.

(2) *Ausführbarkeit*: Jeder Schritt des Verfahrens muss tatsächlich ausführbar sein.

(3) *Dynamische Finitheit*: Das Verfahren darf zu jedem Zeitpunkt nur endlich viel Speicherplatz benötigen (s. Platzkomplexität).

(4) *Terminierung*: Das Verfahren darf nur endlich viele Schritte benötigen (s. auch Zeitkomplexität).

Determiniertheit: Gleicher Input liefert gleiches Ergebnis.
Determinismus: Alle Schritte sind genau festgelegt.
Randomisierter Algorithmus: Man darf „würfeln“.

1.4 Datenstrukturen

Eine Datenstruktur erlaubt es, die für eine Aufgabe notwendigen Informationen

- geeignet zu repräsentieren

- sowie den Zugriff und die Verwaltung während der Bearbeitung in effizienter Weise zu ermöglichen.

1.5 Ausblick

Die Vorlesung wird verschiedene Aspekte von Algorithmen behandeln. Dazu gehören oft auch Analyse und Verständnis der zugrunde liegenden mathematischen Strukturen. Gerade letzteres macht oft den eigentlich Witz aus!
Dieses Verständnis ist wichtig, um

- ein Gefühl für die Besonderheiten eines Problems zu bekommen.
- ein Gefühl für das Funktionieren einer Lösungsmethode zu bekommen.
- eine Lösungsmethode zu entwickeln.
- zu zeigen, dass eine Lösungsmethoden korrekt ist.
- zu zeigen, dass es keine Lösung gibt.
- zu zeigen, dass es keine Lösungsmethode gibt.
- **Spaß dabei zu haben!**

2 Graphen

2.1 Historie

Graphen tauchen in den unterschiedlichsten Lebenslagen und Anwendungsbereichen auf, in denen es darum geht, Beziehungen zwischen Objekten auszudrücken. In der Algorithmik sind sie von besonderer Bedeutung, weil Computer prinzipiell mit diskreten Daten und Strukturen arbeiten, so das Graphen nicht nur als zu bearbeitende Objekte auftauchen, sondern auch bei der Beschreibung von Lösungsmethoden.

Bemerkenswerterweise kann man den historischen Ausgangspunkt von Graphen in der wissenschaftlichen Literatur sehr genau datieren: Im Jahre 1735 präsentierte Leonhard Euler (damals 28 Jahre) der Petersburger Akademie der Wissenschaften seinen Artikel "Solutio problematis ad geometriam situs pertinentis", in dem er sich mit dem Königsberger Brückenproblem beschäftigt: Ist es möglich, alle sieben Brücken Königsbergs in einem durchgehenden Spaziergang zu benutzen, ohne dass eine doppelt überschritten oder andere Mittel zum Überschreiten von Gewässern zur Verfügung stehen? Damit gleicht dieses Problem dem wohlbekannten "Haus des Nikolaus".

In diesem Kapitel werden wir uns genauer mit Graphen auseinandersetzen - und dabei nicht nur verstehen, warum Knoten mit ungerade vielen Kanten für die Existenz von Eulerwegen eine besondere Rolle spielen, sondern auch, wann und wie man Eulerwege auch tatsächlich finden kann.

2.2 Formale Graphenbegriffe

Bevor wir uns näher mit den algorithmischen Aspekten von Wegen in Graphen beschäftigen, müssen wir einen genaueren Blick darauf werfen, wie man Graphen formal sauber definieren kann. (Schließlich kann man einem Computerprogramm auch nicht einfach ein Bild zeigen und mit den Händen wedeln.)

Abbildung 2.1a zeigt ein Beispiel eines Graphen, der aus Knoten (in diesem Falle: $v_1, v_2, v_3, v_4, v_5, v_6$) und Kanten (hier: $e_{1,2}, e_{1,3}, e_{1,4}, e_{2,3}, e_{3,4}, e_{3,5}, e_{4,5}, e_{5,6}$) besteht. Dabei ist das Bild nur eine Veranschaulichung, die für Menschen gut fassbar ist. Zugleich ist die Benennung der Kanten suggestiv: Sie zeigt, welche Knoten durch eine Kante verbunden sind. Tatsächlich kann man den Graphen als eine abstrakte Struktur unabhängig von einem Bild beschreiben.

Die Bezeichnung im Beispiel suggeriert, dass es zwischen zwei Kanten nur eine Beziehung oder keine geben kann – also eine Kante immer eine Menge aus zwei Knoten ist, die verbunden sind. Tatsächlich gibt es oft Situationen, in den mehrere Beziehungen oder

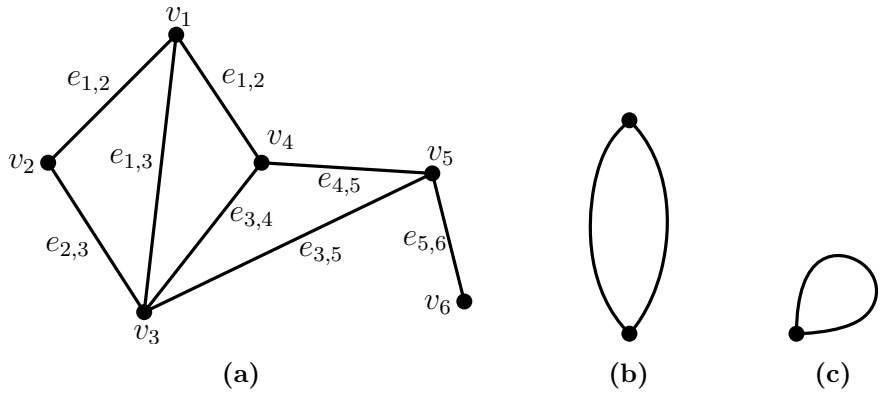


Abbildung 2.1: Beispiele für Graphen: (a) Graph msit sechs Knoten und acht Kanten. (b) Graph mit zwei parallelen Kanten. (c) Graph mit einer Schleife.

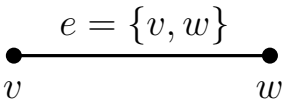


Abbildung 2.2: Ein Beispiel für eine Kante eines Graphen, interpretiert als Menge aus zwei (Knoten-) Objekten.

Verbindungen existieren – symbolisch dargestellt in Abbildung 2.1b. Außerdem kann eine Kante einen Knoten mit sich selbst verbinden – siehe Abbildung 2.1c.

Entsprechend sorgfältig wollen wir bei der formalen Definition eines Graphen vorgehen: Zunächst einmal lassen wir auch parallele Kanten und Schleifen zu.

Definition 2.1 (Ungerichteter Graph)**.**

(1)(i) Ein ungerichteter Graph G ist ein Tripel (V, E, Ψ), für das

(a) V und E endlichen Mengen sind und

(b) Ψ eine Funktion mit

$$\Psi : E \to \{X \subseteq V \mid 1 \leq |X| \leq 2\} \tag{2.1}$$

ist. D. h. jede Kante enthält einen Knoten (Schleife) oder zwei.

(ii) V ist die Knotenmenge.

(iii) E ist die Kantenmenge.

(2)(i) Zwei Kanten e, e' sind parallel, wenn $\Psi(e) = \Psi(e')$.

(ii) Eine Kante e ist eine Schleife, falls $|\Psi(e)| = 1$ gilt.

(iii) Ein Graph ohne parallele Kanten und Schleifen heißt einfach*. Man schreibt dann $G = (V, E)$ mit $E(G)$ der Kantenmenge von G.*

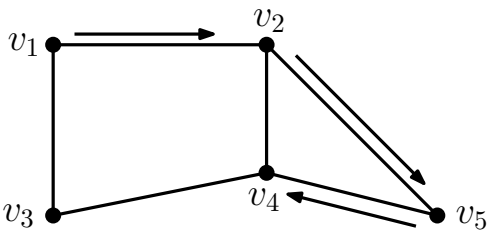


Abbildung 2.3: Ein Pfad v_1, v_2, v_5, v_4 in einem Graphen.

(iv) In einem einfachen Graphen kann man $\{v_i, v_j\}$ für eine Kante $e_{i,j}$ zwischen v_i und v_j schreiben.

(v) $|E|$ ist die Zahl der Kanten von G.

(vi) Oft verwendet man die Buchstaben

n für $|V|$, die Anzahl der Knoten,

m für $|E|$, die Anzahl der Kanten.

Damit haben wir grundlegende Begriffe und Beschreibungen, um damit zu beschreiben, wie wir uns in einem Graphen bewegen können – siehe Abbildung 2.3.

Definition 2.2 (Nachbarschaft in Graphen)**.**

(1) (i) Eine Kante $e = \{v, w\}$ in einem einfachen Graphen verbindet *zwei Knoten v und w. Der Knoten v ist dann Nachbar von w, d. h. v und w sind* adjazent *(„benachbart“).*

(ii) Außerdem ist sowohl v als auch w inzident *(„zusammentreffend mit“) zu e.*

(2) (i) Ein Teilgraph $H = (V(H), E(H))$ eines Graphen $G = (V(G), E(G))$ ist ein Graph mit

$$V(H) \subseteq V(G) \tag{2.2}$$
$$E(H) \subseteq E(G). \tag{2.3}$$

(ii) H ist aufspannend*, wenn $V(H) = V(G)$ gilt.*

(3) (i) Eine Kantenfolge *W in einem Graphen $G = (V, E)$ ist eine Folge*

$$v, e, v_2, e_2, v_3, ..., e_k, v_{k+1} \tag{2.4}$$

mit $k \geq 0$, $e_i = \{v_i, v_{i+1}\} \in E$ und $v_1, ..., v_{k+1} \in V$.

(ii) Wiederholt sich keine Kante in einer Kantenfolge, dann spricht man von einem Weg*.*

(iii) Wiederholt sich kein Knoten in einer Kantenfolge, dann spricht man von einem Pfad*.*

(iv) *Ein* geschlossener Weg *(auch* Tour*) kehrt am Ende zum Startknoten zurück.*

(v) *Ein* Kreis *ist ein geschlossener Pfad, d. h. der Pfad kehrt zum Startknoten zurück.*

(vi) *Ein* Eulerweg *ist ein Weg, der alle Kanten eines Graphen benutzt. Eine* Eulertour *kehrt außerdem zum Startknoten zurück.*

(vii) *Ein* Hamiltonpfad *ist ein Pfad, der alle Knoten eines Graphen besucht.*

(viii) *Ein* Hamiltonkreis *(auch* Hamiltontour*) kehrt außerdem zum Startknoten zurück.*

(4) *Ein Graph ist* zusammenhängend*, wenn es zwischen je zwei Knoten einen Weg gibt.*

(5) *Der* Grad *eines Knotens v ist die Anzahl der inzidenten Kanten und wird bezeichnet mit $\delta(v)$.*

2.3 Eulerwege

Damit lässt sich nun das *Problem von Euler* sauber festhalten.

Problem 2.3 (Eulerweg)**.** Gegeben *ist ein Graph $G = (V, E)$.*
Gesucht *ist ein Eulerweg W in G – oder ein Argument, dass kein Eulerweg existiert.*

Satz 2.4 (Euler)**.** *Ein Graph $G = (V, E)$ kann nur dann einen Eulerweg haben, wenn es höchstens zwei Knoten mit ungeradem Grad gibt.*
G kann nur dann einen geschlossenen Eulerweg haben, wenn alle Knoten geraden Grad haben.

Beweis. Seien $G = (V, E)$ ein Graph und $v \in V$ ein Knoten mit ungeradem Grad $\delta(v)$. Dann kann die Zahl der in einem Eulerweg W zu v hinführenden Kanten nicht gleich der von v wegführenden Kanten sein. Also muss W in v beginnen oder enden. Damit:

(1) Es gibt in einem Eulerweg nur einen Start- und Endknoten.

(2) Bei einem geschlossenen Eulerweg gibt es für den Start- und Endknoten w gleich viele hin- und wegführende Kanten. Also ist auch $\delta(w)$ gerade. □

Mit diesem Satz stellen sich nun folgende Fragen: (I) Was ist mit Graphen, in denen nur ein Knoten ungeraden Grad besitzt? (II) Die Bedingungen in Satz 2.4 sind *notwendig*, das heißt, sie müssen auf jeden Fall erfüllt werden, wenn es eine Chance auf einen Eulerweg geben soll. Sind sie aber auch *hinreichend*, das heißt, gibt es bei Erfüllung auch wirklich einen Weg? (III) Wie findet man einen Eulerweg? (III') Wie sieht ein Algorithmus zum Finden eines Eulerweges aus? Die Antwort auf (I) können wir etwas allgemeiner geben:

Lemma 2.5 (Handshake-Lemma)**.** *Für jeden beliebigen einfachen Graphen ist die Zahl der Knoten mit ungeradem Grad eine gerade Zahl.*

Eingabe: Graph G

Ausgabe: Ein Weg in G

1: starte in einem Knoten v_0
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: $i \leftarrow 0$
3: **while** es gibt eine zu v_i inzidente, unbenutzte Kante **do**
4: wähle eine zu v_i inzidente, unbenutzte Kante $\{v_i, v_j\}$
5: laufe zum Nachbarknoten v_j
6: lösche $\{v_i, v_j\}$ aus der Menge der unbenutzten Kanten
7: $v_{i+1} \leftarrow v_j$
8: $i \leftarrow i + 1$

Algorithmus 2.7: Algorithmus zum Finden eines Weges in einem Graphen

Eingabe: Zusammenhängender Graph G mit höchstens 2 ungeraden Knoten

Ausgabe: Ein Eulerweg, bzw. eine Eulertour in G

1: starte in einem Knoten v
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: verwende Algorithmus 2.7, um einen Weg W von v aus zu bestimmen
3: **while** es existieren unbenutzte Kanten **do**
4: wähle einen Knoten w aus W mit positivem Grad im Restgraphen
5: verwende Algorithmus 2.7, um einen Weg W' von w aus zu bestimmen
6: verschmelze W und W'

Algorithmus 2.8: Hierholzers Algorithmus zum Finden eines Eulerweges oder einer Eulertour

Den Beweis lassen wir zu Übungszwecken aus. Durch das Lemma haben wir nun die Antwort auf die Frage (I): Es gibt keine Graphen, die nur einen Knoten mit ungeradem Grad besitzen.

Betrachten wir nun den Algorithmus 2.7, der in einem gegeben Graphen einen Weg bestimmt. Wir starten dazu in einem beliebigen Knoten v_0 und benutzen sukzessiv unbenutzte Kanten. Gibt es einen Knoten mit ungerade Grad, dann starten wir dort.

Satz 2.9. *Für den Algorithmus 2.7 gelten:*

(1) Das Verfahren stoppt in endlicher Zeit, ist also tatsächlich ein Algorithmus.

(2) Der Algorithmus liefert einen Weg.

(3) Ist v_0 ungerade, stoppt der Algorithmus im zweiten ungeraden Knoten.

(4) Ist G eulersch (d. h. haben alle Knoten geraden Grad), stoppt der Algorithmus in v_0, liefert also einen geschlossenen Weg.

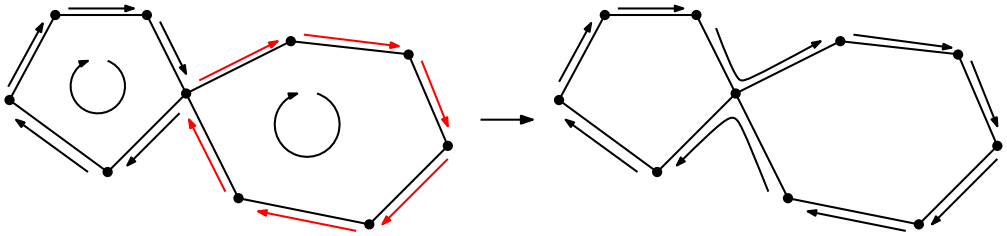


Abbildung 2.4: Verbinde geschlossene Wege mit gemeinsamen Knoten zu einem geschlossenen Weg.

Beweis.

(1) In Algorithmus 2.7 wird bei jedem Durchlauf der Schleife (Zeilen 3-8) in Zeile 6 eine Kante entfernt. Dies kann nur endlich oft passieren.

(2) Nach Konstruktion erhalten wir eine Kantenfolge; keine Kante wird doppelt verwendet.

(3) Ein gerader Knoten wird genauso oft betreten wie verlassen, also kann der Algorithmus weder im Startknoten, noch in einem ungeraden Knoten stoppen, wenn der Startknoten gerade ist. Es bleibt nur der andere ungerade Knoten.

(4) Wie in (3) kann der Algorithmus in keinem geraden Knoten stoppen, der nicht Anfangsknoten ist, also bleibt nur der Startknoten. □

Es lassen sich nun zwei Beobachtungen anstellen. Die schlechte ist, dass noch unbenutzte Kanten übrig bleiben, wenn der Algorithmus stoppt. Die gute ist in folgendem Satz festgehalten.

Satz 2.10. *Wenn der Algorithmus 2.7 stoppt, bleibt ein eulerscher Graph zurück, d. h. ein Graph mit lauter Knoten geraden Grades.*

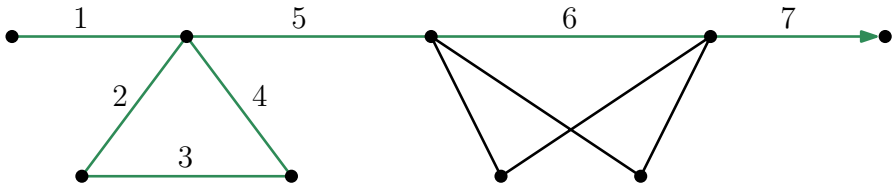
Beweis. Durch Entfernen des Weges wird für jeden geraden Knoten der Grad um eine gerade Zahl geändert, bleibt also gerade. Für einen der ggf. vorhandenen ungeraden Knoten wird der Graph um eine ungerade Zahl geändert, wird also gerade. Wähle also einen neuen Knoten mit positivem Grad auf dem erhaltenen Weg, durchlaufe Algorithmus 2.7, verschmelze die Wege. Das liefert uns Algorithmus 2.8. □

Korollar 2.11.

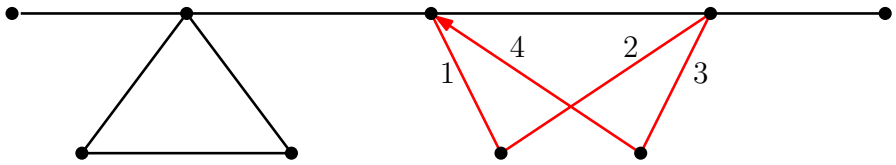
(1) Zwei geschlossene Wege mit einem gemeinsamen Knoten kann man in einen geschlossenen Weg verwandeln. Beispielhaft ist dies in Abbildung 2.4 zu sehen.

(2) Man kann aus allen Wegen einen *Weg machen, wenn der Graph zusammenhängend ist, indem man immer wieder (1) anwendet.*

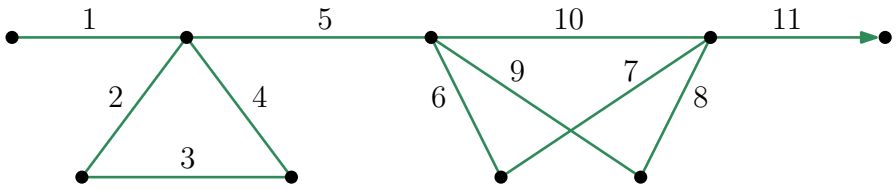
Satz 2.12. *Für den Algorithmus 2.8 (Algorithmus von Hierholzer) gelten:*



(a) Ein Weg in einem Graphen, der beide ungeraden Knoten verbindet, aber Kanten zurücklässt.



(b) Eine Tour im Graphen, der nur zurückgelassene Kanten verwendet.



(c) Ein Eulerweg in einem Graphen, der Weg aus (a) und Tour aus (b) zu einem Weg verschmilzt.

Abbildung 2.5: Beispiele für die sukzessive Konstruktion eines Eulerweges in einem Graphen.

(1) Das Verfahren ist endlich.

(2) Alle Schritte lassen sich korrekt ausführen.

(3) Der Algorithmus liefert einen Eulerweg bzw. eine Eulertour.

Beweis.

(1) Da es nur endlich viele Kanten gibt, muss das Verfahren irgendwann terminieren.

(2) Solange es noch unbenutzte Kanten gibt, kann man diese auch von den benutzten Kanten aus besuchen: Weil G zusammenhängend ist, muss es einen Knoten geben, der sowohl zu einer benutzten als auch zu einer unbenutzten Kante inzident ist.

(3) Am Ende erhält man einen Weg und es gibt keine unbenutzte Kanten mehr. □

Damit haben wir also einen Algorithmus, der einen Eulerweg bzw. eine Eulertour in dem gegebenen Graphen ausgibt. Dass man einen Eulerweg auch anders finden kann, zeigte Fleury 1883: anstatt eine beliebige unbenutze Kante zu benutzen und Wege zu verschmelzen, dürfen nur Kanten benutzt werden, die den Restgraphen zusammenhängend lassen (siehe Algorithmus 2.13).

Satz 2.14. *Für den Algorithmus 2.13 gelten:*

Eingabe: Zusammenhängender Graph G mit höchstens 2 ungeraden Knoten

Ausgabe: Ein Eulerweg, bzw. eine Eulertour in G

1: starte in einem Knoten v_0
   (wenn einer mit ungeradem Grad existiert, dort, sonst beliebig)
2: $i \leftarrow 0$
3: **while** es gibt eine zu v_i inzidente, unbenutzte Kante **do**
4: wähle eine dieser Kanten $\{v_i, v_j\}$, *die den Restgraph zshgd. lässt*
5: laufe zum Nachbarknoten v_j
6: markiere $\{v_i, v_j\}$ als benutzt
7: $v_{i+1} \leftarrow v_j$
8: $i \leftarrow i + 1$

Algorithmus 2.13: Algorithmus von Fleury zum Finden eines Eulerweges oder einer Eulertour

(1) Das Verfahren terminiert.

(2) Alle Schritte lassen sich korrekt ausführen.

(3) Der Algorithmus liefert einen Eulerweg bzw. eine Eulertour.

Beweis.

(1) Da es nur endlich viele Kanten gibt, muss das Verfahren irgendwann terminieren.

(2) Wenn es nur eine Kante gibt, um den aktuellen Knoten zu verlassen, bleibt der Restgraph nach deren Benutzung zusammenhängend. Wenn es mehr als eine Kante gibt, um den aktuellen Knoten zu verlassen, von denen eine bei Entfernen den Graphen unzusammenhängend macht, sind alle anderen Kanten in geschlossenen Wegen enthalten. Also wählt man eine der anderen.

(3) Nach Konstruktion erhält man einen Weg. Man hat immer einen zusammenhängenden Graphen, kann also keinen Kanten zurücklassen. □

2.4 Schlusswort

Von William Tutte (1917-2002), einem der bedeutendsten Graphentheoretiker des 20. Jahrhunderts, gibt es ein zu diesem Kapitel passendes Gedicht:

Some citizens of Koenigsberg
Were walking on the strand
Beside the river Pregel
With its seven bridges spanned.

Oh, Euler come and walk with us
Those burghers did beseech
We'll walk the seven bridges o'er
And pass but once by each.

"It can't be done" then Euler cried
"Here comes the Q. E. D.
Your islands are but vertices,
And all of odd degree."

In der Vorlesung wird eine moderne Interpretation geboten – frei nach MC Hammer!

3 Suche in Graphen

3.1 Vorspann

In diesem Kapitel beschäftigen wir uns als Grundfrage mit der Suche in Graphen. Dabei entdecken und entwickeln wir nach und nach die zugehörigen Aspekte von Algorithmen und Datenstrukturen.

Im Zeitalter von Facebook kennt jeder soziale Netze und direkte oder indirekte Verbindungen darin: Zwei miteinander bekannte Personen haben Abstand 1; zwei nicht miteinander bekannte Personen mit einem gemeinsamen Bekannten haben Abstand 2, und so weiter. Sehr bekannt sind in solchen Netzwerken auch die "Six degrees of separation": Es ist fast unmöglich, zwei Personen zu finden, die man nicht in höchstens sechs Schritten verbinden kann.

Lange vor sozialen Medien haben sich Wissenschaftler bereits mit Abständen in derartigen Netzwerken beschäftigt. Eine zentrale Figur dabei war (und ist) der ungarische Mathematiker Paul Erdős, der nicht nur unglaublich produktiv, sondern auch sehr kooperativ war. Das motivierte den Begriff der *Erdős-Zahl:* Erdős selber hat die Erdő-Zahl 0; jeder seiner Koautoren (der also mit ihm gemeinsam einen wissenschaftlichen Fachartikel publiziert hat) hat Erdős-Zahl 1; deren Koautoren, die nicht mit Erdős publiziert haben, Erdős-Zahl 2, und so weiter. Auch hier geht es also um den Abstand in einem Graphen: Knoten sind Wissenschaftler, Kanten entsprechen Verbindungen durch gemeinsame Publikationen. Kleine Erdős-Zahlen sind durchaus prestigeträchtig – so dass es sogar Menschen gibt, die ihre Erdős-Zahlen auf Webseiten, in Lebensläufe oder auf Autokennzeichen schreiben.

Eine etwas neuere Variante der Erdős-Zahlen gibt es im Kontext von Filmen. Hier ist der amerikanische Schauspieler Kevin Bacon die Bezugsgröße; für Kevin-Bacon-Zahlen gibt es einschlägige Orakel im Netz, mit einer guten Datenbasis durch die Internet Movie Database (IMDb).

In allen diesen Kontexten geht es um ähnliche Fragen: Gegeben ist ein (meistens implizit beschriebener und sehr großer) Graph, in dem es gilt, Verbindungen zwischen Knoten zu finden. Dabei kann es darum gehen, irgendwelche oder auch möglichst kurze Verbindungen zu finden. Indirekt stellen sich zahlreiche Fragen zu Codierung der dabei auftauchenden Strukturen, zur Art und Größe der zugehörigen Datenstrukturen und der Laufzeit von Algorithmen.

3.2 Problemdefinitionen

Problem 3.1 (s-t-Weg)**.** Gegeben *ist der Graph* $G = (V, E)$*, der Startknoten* s *und der Zielknoten* t*.*
Gesucht *ist der Weg von* s *nach* t*, sofern einer existiert.*

Allgemeiner also:

Problem 3.2 (Zusammenhangskomponente)**.** Gegeben *ist der Graph* $G = (V, E)$ *und der Startknoten* s*.*
Gesucht *ist die Menge aller von* s *aus erreichbaren Knoten, die sogenannte* Zusammenhangskomponente *von* s*. Also die Wege, die die Erreichbarkeit sichern.*

Satz 3.3. *Wenn ein Weg zwischen zwei Knoten* s *und* t *in einem Graphen existiert, dann existiert auch ein Pfad.*

Beweis. Sei $W = s, e_1, v_1, ..., v_m, e_{m+1}, t$ ein Weg von s nach t.
Idee: Kreise auf dem Weg W eliminieren.
Technische Umsetzung:
Betrachte unter allen Wegen einen W' mit möglichst wenigen Kanten, d. h. W' ist der kürzeste Weg und damit ein Pfad von s nach t.
Annahme: W' hat einen doppelt besuchten Knoten v_i mit

$$W' = s, e_1, v_1, ..., v_i, e_{i+1}, ..., v_i, e_k, ..., t. \tag{3.1}$$

Dann gibt es einen noch kürzeren Weg W'' mit

$$W'' = s, e_1, v_1, ..., v_i, e_k, ..., t. \tag{3.2}$$

Das widerspricht jedoch der Annahme, dass W' so kurz wie möglich ist. Also hat W' keinen doppelt besuchten Knoten, ist also ein Pfad. □

Daraus ergibt sich bereits eine Konsequenz.

Korollar 3.4. *Für Problem 3.2 gibt es als Erreichbarkeit sichernde Menge von Kanten immer eine Menge, die keinen Kreis enthält.*

Aus dem Beweis ergibt sich sogar eine Verschärfung.

Korollar 3.5. *Für Problem 3.2 gibt es immer eine kreisfreie Menge von Kanten, die die am kürzesten mögliche Erreichbarkeit sichert.*

Das motiviert die folgende Definition.

Definition 3.6 (Wald und Baum)**.**

(1) Ein Wald *ist ein kreisfreier Graph.*

(2) Ein Baum *ist eine Zusammenhangskomponente in einem Wald, also ein kreisfreier zusammenhängender Graph.*

(3) Ein aufspannender Baum *(auch* Spannbaum*, engl.* spanning tree*) ist ein Baum, der alle Knoten verbindet.*

Eingabe: Graph $G = (V, E)$, Knoten $s \in V$

Ausgabe:

(1) Knotenmenge $Y \subseteq V$, die von s aus erreichbar ist (die Zusammenhangskomponente von s)

(2) Kantenmenge $T \subseteq E$, die die Erreichbarkeit sicherstellt (Spannbaum der Zusammenhangskomponente)

```
1:  R ← {s}
2:  Y ← {s}
3:  T ← ∅

4:  while R ≠ ∅ do
5:      v ← wähle Knoten aus R

6:      if es gibt kein w ∈ V \ Y mit {v, w} ∈ E then
7:          R ← R \ {v}
8:      else
9:          w ← wähle w ∈ V \ Y mit e = {v, w} ∈ E

10:         R ← R ∪ {w}
11:         Y ← Y ∪ {w}
12:         T ← T ∪ {e}
```

Algorithmus 3.7: Bestimmung der Zusammenhangskomponente eines Startknotens s eines Graphens sowie des zugehörigen Spannbaums.

3.3 Zusammenhangskomponenten

Um nun eine Zusammenhangskomponente zu finden, starten wir bei einem Knoten und konstruieren von dort aus einen Baum mit den Kanten des gegebenen Graphen. Eine detailierte Beschreibung kann Algorithmus 3.7 entnommen werden.

Satz 3.8. *Der Algorithmus 3.7 ist*

(1) endlich

(2) korrekt.

Beweis.

(1) Bei jedem Durchlauf der Schleife (Zeilen 4-12) wird entweder in Zeile 5 ein Element aus R entfernt oder in den Zeilen 10-12 ein Element zu R und Y hinzugefügt. Damit können die Zeilen 10-12 nur $(n-1)$-mal durchlaufen, also auch R nur $(n-1)$-mal erweitert und somit R höchstens n-mal verkleinert werden. Die Schleife (Zeilen 4-12) kann also höchstens $(2n-1)$-mal durchlaufen werden.

(2) Zu jedem Zeitpunkt ist (Y, T) ein s enthaltender Baum, denn

(a) alle Knoten in G sind von s aus erreichbar (und umgekehrt) und

(b) neu eingefügte Kanten verbinden die bisherigen Knoten aus Y nur mit bislang nicht erreichbaren Knoten, können also keinen Kreis schließen.

Nun bleibt zu zeigen, dass alle erreichbaren Knoten auch korrekt identifiziert werden. Dazu nehmen wir an, dass es am Ende einen Knoten $w \in V \setminus Y$ gibt, der in G von s aus erreichbar ist.

Sei P ein s-w-Pfad in G und sei x, y eine Kante von P mit $x \in Y$, $y \notin Y$. Da x zu Y gehört, wurde x auch zu R hinzugefügt. Der Algorithmus stoppt aber nicht, bevor x aus R entfernt wird. Das wird in Zeile 7 nur vorgenommen, wenn es in Zeile 6 keine Kante x, y mit $y \notin Y$ gibt. Das ist ein Widerspruch zur Annahme. □

3.4 Warteschlange und Stapel

Im Graphenscan-Algorithmus 3.7 sind einige Dinge offengelassen: Er funktioniert ganz unabhängig davon, wie man die Menge R abarbeitet – also zum Beispiel auch, wenn man R relativ wild und ohne besondere Reihenfolge verwaltet. Trotzdem ist es wichtig, sich Gedanken zu machen, welche systematischen Reihenfolgen sich dafür besonders anbieten – und welche Datenstrukturen dem entsprechen. In jedem Falle gehören dazu zwei Operationen: das *Hinzufügen* zur verwalteten Menge und das *Abrufen* aus der Menge.

Eine naheliegende Art der Bearbeitung ist in Form einer *Warteschlange*; dies entspricht dem Prinzip *First In – First Out*, kurz FIFO. Man kennt Warteschlangen aus den unterschiedlichsten Kontexten; sie werden aus guten Gründen gerne eingesetzt, wenn es um das Abarbeiten von Einzelaufgaben geht. Man kann Warteschlangen auch für das Verwalten von Daten in Arrayform einsetzen; die entsprechenden Operationen zum Einfügen und Abrufen sind als Pseudocode 3.1 und 3.2 gezeigt.

```
1: function ENQUEUE(Q, x)
2:     Q[end[Q]] ← x
3:     if end[Q] = length[Q] then
4:         end[Q] ← 1
5:     else
6:         end[Q] ← end[Q] + 1
```

Pseudocode 3.1: Das Hinzufügen eines Elementes x zu einer Warteschlange Q.

Eine andere natürliche Art der Bearbeitung ist der *Stapel*, manchmal auch *Keller* genannt, der dem Prinzip *Last In – First Out*, entspricht, kurz LIFO. Stapel werden ebenfalls oft eingesetzt – nicht nur, wenn man Dinge auf dem Schreibtisch stapelt (und nur von oben hinzufügt oder entfernt), sondern auch in ganz anderen Zusammenhängen. (Seit einigen Jahren zählen Radiosender die folgenden Musiktitel in umgekehrter Reihenfolge auf, so dass der letztgenannte zuerst gespielt wird.) Genau wie Warteschlangen

```
1: function DEQUEUE(Q)
2:     x ← Q[head[Q]]
3:     if head[Q] = length[Q] then
4:         head[Q] ← 1
5:     else
6:         head[Q] ← head[Q] + 1
7:     return x
```

Pseudocode 3.2: Das vorderste Element x der Warteschlange Q abrufen.

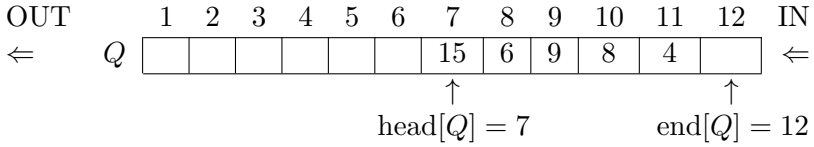


Abbildung 3.1: Beispiel für eine Warteschlange, die mithilfe eines Arrays umgesetzt wird.

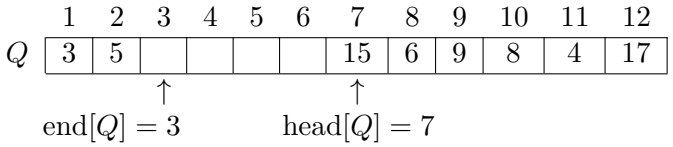


Abbildung 3.2: Sequenzielles hinzufügen von 17, 3 und 5 zur Warteschlange Q aus Abbildung 3.1 gemäß Algorithmus 3.1.

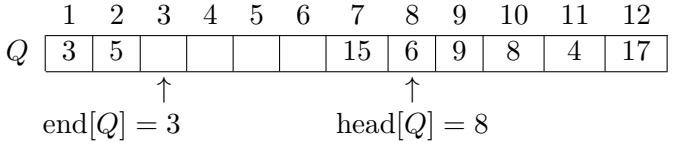


Abbildung 3.3: Entfernen des ersten Elementes der Warteschlange Q aus Abbildung 3.2 gemäß Algorithmus 3.2.

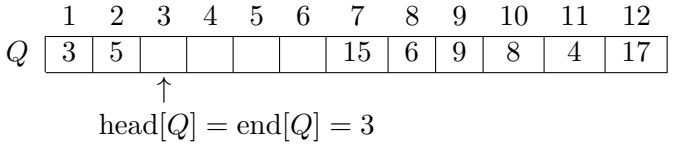


Abbildung 3.4: Nach acht suksezziven Aufrufen von DEQUEUE auf die Warteschlange aus Abbildung 3.3 ist die Warteschlange leer.

kann man Arrays für das Verwalten von Daten in Arrayform einsetzen; die entsprechenden Operationen zum Einfügen und Abrufen sind als Pseudocode 3.3, 3.4 und 3.5 gezeigt.

```
1: function STACK-EMPTY(S)
2:     if top[S] = 0 then
3:         return WAHR
4:     else
5:         return FALSCH
```

Pseudocode 3.3: Prüfen, ob ein Stapel S leer ist.

```
1: function PUSH(S, x)
2:     top[S] ← top[S] + 1
3:     S[top[S]] ← x
```

Pseudocode 3.4: Ein Element x auf den Stapel S legen.

```
1: function POP(S)
2:     if STACK-EMPTY(S) then
3:         error „Unterlauf“
4:     else
5:         top[S] ← top[S] − 1
6:         return S[top[S] + 1]
```

Pseudocode 3.5: Das oberste Element des Stapels S abrufen.

3.5 Tiefensuche und Breitensuche

Die Verwendung von Warteschlange oder Stapel als Datenstruktur zum Verwalten der Menge R im Kontext des Graphenscan-Algorithmus zeigen sehr schön das Zusammenspiel von Algorithmen und Datenstrukturen! Obwohl man im Prinzip nur verschiedene Varianten desselben Algorithmus erhält, läuft die Suche sehr unterschiedlich ab.

Setzt man eine Warteschlange ein, so geht die Suche *in die Breite*; man spricht auch von *Breitensuche* oder *Breadth-First Search*, kurz BFS. Wir werden im Abschnitt 3.9 sehen, dass dies besonders interessante Eigenschaften liefert, inbesondere kürzeste Wege von einer Quelle zu allen anderen erreichbaren Knoten eines Graphen: Die Breitensuche breitet sich wie eine Welle gleichmäßig von der Quelle in alle Richtungen aus. Allerdings ist bei der Umsetzung in der realen Welt erforderlich, dass man für so eine Suche entweder jeweils zum aktuellen Knoten läuft (was unter Umständen viel Lauferei erfordern kann) oder viele Personen gleichzeitig beteiligt sind. Insofern kann man sich die Suche hier auch als einen parallelen Prozess vorstellen, an dem viele kooperierende Sammler beteiligt sind.

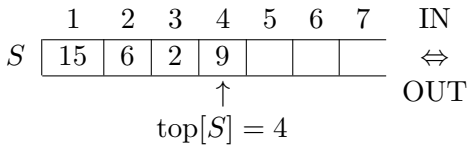


Abbildung 3.5: Beispiel für einen Stapel, der mithilfe eines Arrays umgesetzt wird.

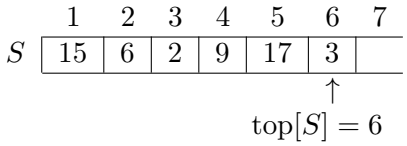


Abbildung 3.6: Sequenzielles hinzufügen von 17 und 3 zum Stapel S aus Abbildung 3.5 gemäß Algorithmus 3.4.

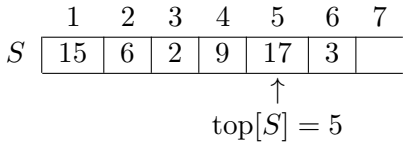


Abbildung 3.7: Entfernen des obersten Elementes des Stapels S aus Abbildung 3.6 gemäß Algorithmus 3.5.

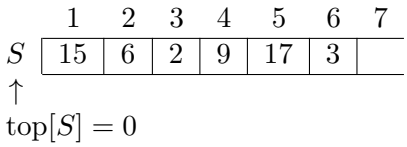
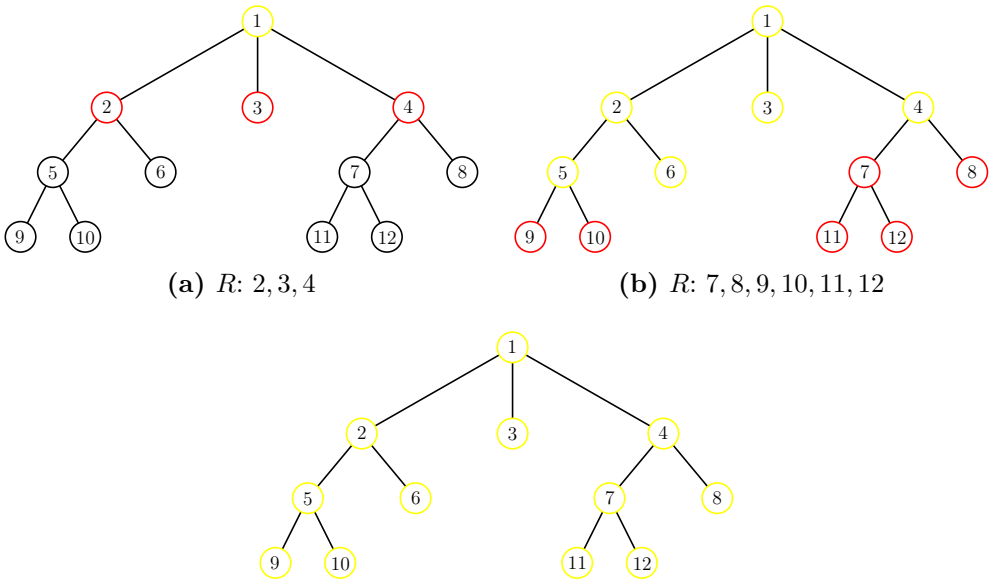


Abbildung 3.8: Nach fünf suksezziven Aufrufen von pop auf den Stapel aus Abbildung 3.7 ist der Stapel S leer.



(a) R: $2, 3, 4$ **(b)** R: $7, 8, 9, 10, 11, 12$

(c) R: $\emptyset \Rightarrow$ Stop!

Abbildung 3.9: Ablauf der Breitensuche, also des Graphenscanalgorithmus' 3.7 bei Verwendung einer Warteschlange als Datenstruktur für R.

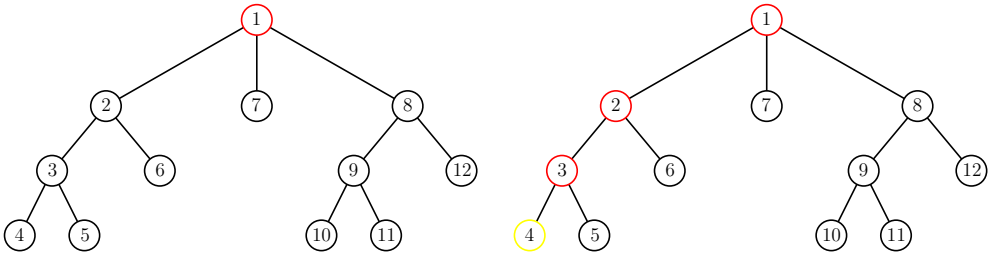
Das Gegenstück zur Breitensuche ist die *Tiefensuche* oder *Depth-First Search*, kurz DFS. Sie entsteht als Variante des Graphenscan-Algorithmus, wenn man einen Stapel als Datenstruktur zum Verwalten von R einsetzt. Ein Vorteil der Tiefensuche ist, dass man sie tatsächlich als Einzelner zum Durchlaufen eines Graphen verwenden kann; sie eignet sich also für „einsame Jäger". Zugleich kann es aber sein, dass man bis zum Erreichen eines bestimmten Knotens einen beachtlichen Weg zurücklegen kann.

3.6 Datenstrukturen für Graphen

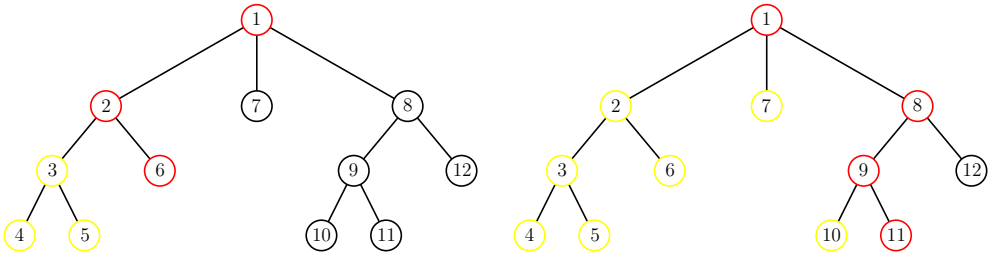
Bevor wir weiter auf die Eigenschaften von Breiten- und Tiefensuche eingehen können, müssen wir uns zuerst noch mit einer sehr handfesten Fragen bei der Umsetzung auseinandersetzen, die wiederum zu Datenstrukturen führt: Wie beschreibt man einen Graphen?

3.6.1 Inzidenzmatrix

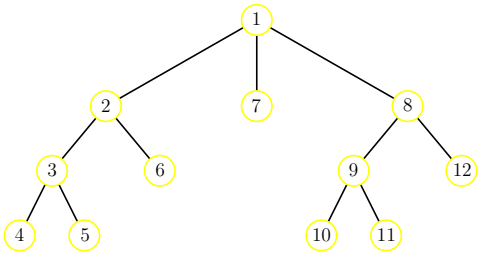
Eine Inzidenzmatrix eines Graphen $G = (V, E)$ beschreibt, welche Knote $v_i \in V$ zu welchen Kanten $e_j \in E$ inzident sind. Daher hat diese Matrix die Dimension $n \times m$, wobei $n = |V|$ und $m = |E|$ gilt. Dies bedeutet für große Graphen natürlich, dass die Matrix tendenziell viele Nullen enthält. Die zum Graphen aus Abbildung 3.11 gehörende



(a) R: 1 **(b)** R: 1, 2, 3



(c) R: 1, 2, 6 **(d)** R: 1, 8, 9, 11



(e) R: $\emptyset \Rightarrow$ Stop!

Abbildung 3.10: Ablauf der Tiefensuche, also des Graphenscanalgorithmus' 3.7 bei Verwendung eines Stapels als Datenstruktur für R.

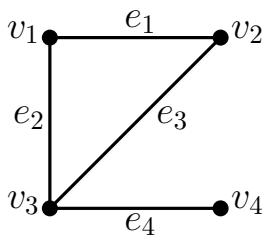


Abbildung 3.11: Ein einfacher Graph G.

Inzidenzmatrix A sieht dann wie folgt aus.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{3.3}$$

Es gilt also $A \in \{0,1\}^{n \times m}$ mit

$$a_{v,e} \coloneqq \begin{cases} 1 & \text{für } v \in e \\ 0 & \text{sonst} \end{cases} \tag{3.4}$$

3.6.2 Adjazenzmatrix

Eine Adjazenzmatrix eines Graphen $G = (V, E)$ beschreibt, welche Knoten v_i zu welchen Knoten v_j adjazent sind ($v_i, v_j \in V$). Daher hat diese Matrix die Dimension $n \times n$, wobei $n = |V|$ gilt. Dies bedeutet, dass eine Adjazenzmatrix immer quadratisch ist. Für einfache Graphen enthält die Diagonale ausschließlich Nullen. Die zum Graphen aus Abbildung 3.11 gehörende Adjazenzmatrix A sieht dann wie folgt aus.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{3.5}$$

Es gilt also $A \in \{0,1\}^{n \times n}$ mit

$$a_{v,w} \coloneqq \begin{cases} 1 & \text{für } \{v, w\} \in E \\ 0 & \text{sonst} \end{cases} \tag{3.6}$$

3.6.3 Kantenliste

Eine Kantenliste eines Graphen $G = (V, E)$ ist eine Liste aus zweielementigen Knotenmengen $\{v_i, v_j\}$, die eine Kante von v_i zu v_j beschreiben ($v_i, v_j \in V$). Die zum Graphen aus Abbildung 3.11 gehörende Kantenliste sieht dann wie folgt aus.

$$[\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}] \tag{3.7}$$

Benötigt wird nun eine Kantennummerierung. Hierfür betrachtet man jeden Index als Binärzahl der Länge $\log_2(n+1)$ bzw. als Dezimalzahl mit $\log_{10}(n+1)$ Stellen, wobei wieder $n = |V|$ gilt. Die beiden Varianten unterscheiden sich lediglich um den Faktor $\log_2(10) = 3,3219...$ in der Codierungslänge, denn es gilt $\log_2(n) = \log_2(10) \cdot \log_{10}(n)$.

Für einen Graphen mit m Kanten und n Knoten ergibt sich in (ausführlicher) Codierung somit ein Platzbedarf von

$$(6m - 1) + 2m \log_{10}(n + 1). \tag{3.8}$$

Dabei kann man an ein paar Stellen sparen (z. B. „v“ oder „{“, „}“ weglassen), aber auch mehr Platz investieren (z. B. in ASCII codieren bzw. binär statt dezimal codieren). So wäre auch ein Platzbedarf von

$$(2m - 1) + 2m \log_2(n + 1) \tag{3.9}$$

denkbar.
Was ist wirklich wichtig dabei?

(1) Die Kantenliste ist sparsamer als die Inzidenzmatrix, denn wenn n nicht zu klein ist ($n \geq 8$), dann gilt

$$n \geq 2 + 2 \log_2(n + 1). \tag{3.10}$$

Also ist auch

$$mn > (2m - 1) + 2m \log_2(n + 1). \tag{3.11}$$

(2) Unabhängig von der Codierung ist für die Größe des Speicherplatzes der zweite Ausdruck wichtig, denn es gilt

$$\begin{aligned} 2m \log_2(n + 1) &\leq (2m - 1) + 2m \log_2(n + 1) && (3.12)\\ &\leq 4m \log_2(n + 1) \qquad \text{für } n \geq 2 && (3.13) \end{aligned}$$

(3) Letztlich kommt es also gar nicht so sehr auf die Vorfaktoren an (die sind codierungsabhängig!), sondern auf den Ausdruck

$$m \log(n). \tag{3.14}$$

(4) In $m \log(n)$ steckt das Wesen der Kantenliste: Zähle für alle m Kanten die Nummern der beteiligten Knoten auf!
Wie man in Abschnitt 3.7 sehen wird, lohnt sich dafür eine eigene Notation:

Die Kantenliste benötigt $\Theta(m \log(n))$ Speicherplatz.

3.6.4 Adjazenzliste

Eine Adjazenzliste eines Graphen $G = (V, E)$ gibt zu jedem Knoten v_i die benachbarten Knoten v_j an ($v_i, v_j \in V$). Das ist etwas praktischer als die Kantenliste, wenn man für Graphenalgorithmen direkten Zugriff auf die Nachbarn einen Knotens benötigt. Man

muss nicht die Nachbarn erst mühsam aus einer Liste heraussuchen. Die zum Graphen aus Abbildung 3.11 gehörende Adjazenzliste sieht dann wie folgt aus.

$$v_1 : v_2, v_3; \tag{3.15}$$
$$v_2 : v_1, v_3; \tag{3.16}$$
$$v_3 : v_1, v_2, v_4; \tag{3.17}$$
$$v_4 : v_3; \tag{3.18}$$

Auch hier betrachte man wieder den Speicherbedarf. Dafür benötigt man die Länge einer solchen Adjazenzliste. Da jede Kante doppelt auftaucht, nämlich einmal für jeden Knoten, gilt

$$2n + 4m + n\log_{10}(n+1) + 2m\log_{10}(n+1). \tag{3.19}$$

D. h. $\Theta(n\log(n) + m\log(n))$.
Im Allgemeinen sind Graphen mit vielen isolierten Knoten (ohne Kanten!) uninteressant, d. h. z. B. $m \geq \frac{n}{2}$, $m \geq n$ o. ä. Also wieder $\Theta(m\log(n))$.
Nun möchte man jedoch, zur Verbesserung der Zugriffszeit, den direkten Zugriff auf die Nachbarn eines Knotens, denn das Durchsuchen der Liste nach Semikolons dauert. Dafür werden Zeiger (*engl.* Pointer) auf jeden Knoten (nach dem Semikolon) gesetzt.

$$\begin{array}{cccc} \downarrow & \downarrow & \downarrow & \downarrow \\ v_1\colon v_2, v_3; & v_2\colon v_1, v_3; & v_3 : v_1, v_2, v_4; & v_4 : v_3; \end{array} \tag{3.20}$$

Man verwendet hier also das Prinzip:

> „Man muss nicht alles wissen, man muss nur wissen, wo's steht."

Betrachtet man nun wieder den Speicherbedarf, so erkennt man, dass für jeden Knoten ein Zeiger benötigt wird, also n Stück. Jeder dieser Zeiger codiert eine Speicherzelle, d. h. die Nummer eines Bits in der Liste. Ein solcher Zeiger benötigt selbst also

$$\log_2(2n + 4m + n\log_2(n+1)) + 2m\log_2(n+1) \tag{3.21}$$
$$\leq \log_2(9m + \log_2(n+1) + 1) \qquad \text{für } n \geq 10,\ m \geq n \tag{3.22}$$
$$\leq \log_2(9) + \log_2(m) + \log_2(\log_2(n+1)) + 1 \tag{3.23}$$
$$\leq 2\log_2(m) \tag{3.24}$$

Bits. Insgesamt sind das also $\Theta(n\log_2(m))$ Bits.
Bekanntermaßen gilt für einfache Graphen, dass $m \leq n^2$, d. h.

$$\log_2(m) \leq \log_2(n^2) = 2\log_2(n). \tag{3.25}$$

Somit gilt

$$n\log_2(m) \leq 2n\log_2(n), \tag{3.26}$$

und damit liegt der Speicherverbrauch insgesamt bei

$$\Theta(n\log_2(m) + m\log_2(n)) = \Theta(m\log_2(n)). \tag{3.27}$$

3.7 Wachstum von Funktionen

Im letzten Abschnitt wurden Funktionen abgeschätzt und auf die „wesentlichen“ Bestandteile reduziert, um Größenordnungen und Wachstumsverhalten zu beschreiben. Dies wird nun mit folgender Definition formalisiert.

Definition 3.9 (Θ-Notation)**.** *Seien $f, g : \mathbb{N} \to \mathbb{R}$, dann gilt*

$$f \in \Theta(g) :\Leftrightarrow \exists c_1, c_2 \in \mathbb{R}_{>0} : \exists n_0 \in \mathbb{N}_{>0} : \forall n \in \mathbb{N}_{\geq n_0} : \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{3.28}$$

Man sagt, f wächst asymptotisch in derselben Größenordnung wie g.

Beispiele:

(1) $2n^2 - 1 \in \Theta(n^2)$

(2) $\frac{n^3}{1000} + n^2 + n\log(n) \in \Theta(n^3)$

Manchmal hat man aber auch nur untere oder obere Abschätzungen, sodass sich die folgenden Definitionen ergeben.

Definition 3.10 (O-Notation)**.** *Seien $f, g : \mathbb{N} \to \mathbb{R}$, dann gilt*

$$f \in O(g) :\Leftrightarrow \exists c \in \mathbb{R}_{>0} : \exists n_0 \in \mathbb{N}_{>0} : \forall n \in \mathbb{N}_{\geq n_0} : \\ 0 \leq f(n) \leq cg(n) \tag{3.29}$$

Man sagt, f wächst höchstens in derselben Größenordnung wie g.

Beispiele:

(1) $2n^2 - 1 \in O(n^2)$

(2) $2n^2 - 1 \in O(n^3)$

(3) $n\log(n) \in O(n^2)$

Definition 3.11 (Ω-Notation)**.** *Seien $f, g : \mathbb{N} \to \mathbb{R}$, dann gilt*

$$f \in \Omega(g) :\Leftrightarrow \exists c \in \mathbb{R}_{>0} : \exists n_0 \in \mathbb{N}_{>0} : \forall n \in \mathbb{N}_{\geq n_0} : \\ 0 \leq cg(n) \leq f(n) \tag{3.30}$$

Beispiele:

(1) $2n^2 - 1 \in \Omega(n^2)$

(2) $2n^3 - 1 \in \Omega(n^2)$

Zu den soeben definierten Notationen lassen sich einige Eigenschaften festhalten.

Satz 3.12. *Seien $f, g : \mathbb{N} \to \mathbb{R}$, dann gelten*

(1) $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$

(2) $f \in \Theta(g) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$

(3) $f \in O(g) \Leftrightarrow g \in \Omega(f)$

Beweis. Übung! □

3.8 Laufzeit von BFS und DFS

Durchsucht man einen Graphen, so sollten alle Knoten und Kanten dabei betrachtet werden. Daher lässt sich eine untere Schranke von $\Omega(n+m)$ nicht unterbieten. Tatsächlich kann man dies auch als Oberschranke erreichen.

Satz 3.13 (Laufzeit Graphenscan-Algorithmus)**.** *Der Graphenscan-Algorithmus 3.7 lässt sich so implementieren, dass die Laufzeit $O(m+n)$ beträgt.*

Beweis. Annahme: G ist durch eine Adjazenzliste gegeben.
Für jeden Knoten verwendet man einen Zeiger, der auf die „aktuelle“Kante für diesen Knoten in der Liste zeigt (d. h. auf den „aktuellen“Nachbarn). Anfangs zeigt $akt(x)$ auf das erste Element in der Liste. In Zeile 9 wird der aktuelle Nachbar ausgewählt und der Zeiger weiter bewegt. Wird das Listenende erreicht, wird x aus R entfernt und nicht mehr eingeführt.
Also ergibt sich eine Gesamtlaufzeit von $O(m+n)$. □

Korollar 3.14. *Mit Algorithmus 3.7 kann man alle Zusammenhangskomponenten eines Graphen berechnen.*

Beweis. Wende den Algorithmus an und überprüfe, ob $Y = V$ ist. Falls ja, ist der Graph zusammenhängend. Andernfalls wurde *eine* Zusammenhangskomponente berechnet. Nun startet man den Algorithmus erneut für einen Knoten $s' \in V \setminus Y$ usw. Wieder wird keine Kante von einem Knoten aus doppelt angefasst, also bleibt die Laufzeit Linear, d. h. $O(n+m)$. □

Korollar 3.15. *BFS und DFS haben die Laufzeit $O(n+m)$.*

Beweis. Einfügen in R (Warteschlange oder Stapel) lässt sich jeweils in konstanter Zeit vornehmen, der Rest überträgt sich von Satz 3.13. □

3.9 Besondere Eigenschaften von BFS und DFS

Einfach gesagt ist

- DFS eine bestmögliche, individuelle Suchstrategie mit lokaler Information.
- BFS eine bestmögliche, kooperative Suchstrategie mit globaler Information.

Konkret heißt das, dass

- DFS gut für die Suche nach einem Ausweg aus einem Labyrinth geeignet ist.
- BFS gut für die Suche nach kürzesten Wegen geeignet ist.

Satz 3.16 (Lokale Suche mit DFS)**.** *DFS ist eine optimale lokale Suchstrategie im folgenden Sinne.*

Eingabe: Graph $G = (V, E)$, Knoten $s \in V$

Ausgabe:

(1) Knotenmenge $Y \subseteq V$, die von s aus erreichbar ist (die Zusammenhangskomponente von s)

(2) für jeden Knoten $v \in Y$ die Länge $d(s, v)$ eines kürzesten s-t-Weges

(3) Kantenmenge $T \subseteq E$, die die Erreichbarkeit sicherstellt (Spannbaum der Zusammenhangskomponente)

```
1:  ENQUEUE(R, s)
2:  Y ← {s}
3:  T ← ∅

4:  while head[R] ≠ end[R] do                ▷ Solange die Queue nicht leer ist
5:      v ← wähle erstes Element aus R

6:      if es gibt kein w ∈ V \ Y mit {v, w} ∈ E then
7:          DEQUEUE(R)
8:      else
9:          w ← wähle w ∈ V \ Y mit e = {v, w} ∈ E

10:         ENQUEUE(R, w)
11:         Y ← Y ∪ {w}
12:         T ← T ∪ {e}
13:         l(w) ← l(v) + 1
```

Algorithmus 3.17: Bestimmung der Zusammenhangskomponente eines Startknotens s eines Graphens sowie des zugehörigen Spannbaums. Zusätzlich Bestimmung der Länge des kürzesten s-t-Weges für jeden von s aus erreichbaren Knoten.

(1) DFS findet in jedem zusammenhängendem Graphen mit n Knoten einen Weg der Länge höchstens $2n - 1$, der alle Knoten besucht.

(2) Für jede lokale Suchstrategie gibt es einen Graphen mit n Knoten, sodass der letzte Knoten erst nach einer Weglänge von $2n - 1$ besucht wird.

Beweis. Übung! □

Satz 3.18. *Für den Algorithmus 3.17 gelten:*

(1) Das Verfahren ist endlich.

(2) Die Laufzeit ist $O(n + m)$.

(3) Am Ende ist für jeden erreichbaren Knoten $v \in Y$ die Länge eines kürzesten Weges von s nach v im Baum (Y, T) *durch $l(v)$ gegeben.*

(4) Am Ende ist für jeden erreichbaren Knoten $v \in Y$ die Länge eines kürzesten Weges von s nach v im Graphen (V, E) *durch l(v) gegeben.*

Beweis.

(1) Wie für Algorithmus 3.7 gelten alle Eigenschaften. Zusätzlich ist für jeden Knoten $v \in Y$ per Induktion der Wert l(v) tatsächlich definiert.

(2) Die Laufzeit bleibt von Algorithmus 3.7 erhalten.

(3) Sei $d_{(Y,T)}$ die Länge eines kürzesten Weges von s nach v in (Y, T). Dann zeigt man durch Induktion über $d_{(Y,T)}(s, v)$, dass für alle Knoten $d_{(Y,T)}(s, v) = l(v)$ gilt.
Induktionsanfang:
$d_{(Y,T)}(s, v) = 0$ gilt für $v = s$ und $l(s) = 0$.
Induktionsannahme:
Sei $d_{(Y,T)}(s, v) = l(v)$ für alle $v \in V$ mit $d_{(Y,T)}(s, v) \leq k - 1$.
Induktionsschritt:
Sei $w \in V$ ein Knoten mit $d_{(Y,T)}(s, w) = k$. Dann gibt es im Baum (Y, T) einen eindeutigen Weg von s zu w. Sei v der Vorgänger von w in diesem Weg, also $v, w \in T$. Nach Induktionsannahme gilt

$$\begin{array}{lllll} & d_{(Y,T)}(s,v) & = & l(v), & \\ \text{außerdem ist} & d_{(Y,T)}(s,w) & = & d_{(Y,T)}(s,v)+1 & \text{(Abstand im Baum)} \\ \text{und} & l(w) & = & l(v)+1. & \text{(wird in Alg. 3.7 gesetzt)} \\ \text{Also auch} & d_{(Y,T)}(s,w) & = & l(w). & \end{array}$$

(4) Man benötigt zunächst die Eigenschaft, dass zu jedem Zeitpunkt für die Warteschlange $R : v_i, v_{i+1}, ..., v_k$ gilt

$$l(v_i) \leq ... \leq l(v_k) \leq l(v_i) + 1. \tag{3.31}$$

Dies beweist man per Induktion über die Zahl z der aufgenommen Kanten.
Induktionsanfang:
Sei $z = 0$, dann besteht die Warteschlange R nur aus s und somit gilt 3.31.
Induktionsannahme:
Die Aussage gelte nach $(z - 1)$ Kanten. Sei $R : v_i, v_{i+1}, ..., v_k$ und $l(v_i) \leq l(v_{i+1} \leq ... \leq l(v_k) \leq l(v_i) + 1$.
Induktionsschritt:
Man fügt eine weitere Kante ein. Werden dafür zunächst Knoten aus R gelöscht, weil sie keinen neuen Nachbarn haben, ändert das nichts an 3.31. Sei v_j danach der erste Knoten, für den eine neue Kante $e = \{v_j, v_{k+1}\}$ eingefügt wird. Dann bekommt man $R : v_j, v_{j+1}, ..., v_k, v_{k+1}$, mit $l(v_i) \leq ... \leq l(v_j) \leq ... \leq l(v_k) \leq l(v_i) + 1$. Wegen $l(v_i) \leq l(v_j)$ gilt auch $l(v_i) + 1 \leq l(v_j) + 1 = l(v_{k+1})$. Damit erhält man

$$l(v_j) \leq ... \leq l(v_k) \leq l(v_i) + 1 \leq l(v_j) + 1 = l(v_{k+1}) \tag{3.32}$$

und 3.31 gilt weiterhin.

Jetzt nimmt man an, dass es am Ende des Algorithmus 3.17 einen Knoten $w \in V$ gibt, für den kein kürzester Weg gefunden wurde, also $d(s,w) < d_{(Y,T)}(s,w) = l(w)$. Unter den Knoten mit dieser Eigenschaft wählen wir einen mit minimalem Abstand von d in G.
Sei P ein kürzester s-w-Weg in G und sei $e = \{v,w\}$ die letzte Kante in P. D. h. $d(s,w) = d(s,v) + 1$. Damit erhält man $d(s,v) = d_{(Y,T)}(s,v)$, aber $d(s,w) < d_{(Y,T)}(s,w)$, also gehört e nicht zu T. Außerdem ist

$$\begin{aligned} l(w) &= d_{(Y,T)}(s,w) && (3.33)\\ &> d(s,w) && (3.34)\\ &= d(s,v) + 1 && (3.35)\\ &= d_{(Y,T)}(s,v) + 1 && (3.36)\\ &= l(v) + 1. && (3.37) \end{aligned}$$

Wäre $w \in Y$, hätte man wegen $l(v) > l(v) + 1$, also einen Widerspruch zur Eigenschaft 3.31. Also wäre $w \notin Y$. Dann wäre die Kante $e = \{v,w\}$ zum Zeitpunkt der Entfernung von v aus R aber eine Verbindung von v mit einem Knoten $v \notin Y$, im Widerspruch zur Abfrage in Zeile 6. □

3.10 Schlusswort

Zur Melodie von “Longest Time” von Billy Joel:

Wo-oh-oh-oh ist ein kurzer Weg?
Wo-oh-oh-oh ist ein kurzer Weg?
Wo-oh-oh-oh ist ein kurzer ...

Facebook, Bacon und die Erdős-Zahl
Graphenscan ist einfach und genial.
Er schafft Verbindung
und er dient der Knotenfindung
wenn auch nicht notwendig auf ’nem kurzen Weg.

Wo-oh-oh-oh ist ein kurzer Weg?
Wo-oh-oh-oh ist ein kurzer Weg?

Stapeleinsatz macht die Suche tief,
was sehr gut wär, wenn allein man lief’:
kartenlos jagen
ohne nach dem Weg zu fragen
denn mit DFS erreicht man jedes Ziel.

Adjazenzliste Datenstruktur
zweimal muss man nur
jede Kante betrachten!

Warteschlange macht die Suche breit
löst das Problem in linearer Zeit.
Ganz auf die Schnelle,
so wie eine Floodingwelle
liefert BFS uns einen kurzen Weg!

So-oh-oh-oh findet man den Weg,
so-oh-oh-oh findet man den Weg,
so-oh-oh-oh findet man den Weg!

4 Dynamische Datenstrukturen

In diesem Kapitel widmen wir uns Datenstrukturen, für die die Menge von Objekten dynamisch ist. Dazu betrachten wir zunächst allgemeine Operationen an, die bei diesen Datenstrukturen Anwendung finden. Danach geben wir verschiedene Datenstrukturen an, die diese Grundoperationen umsetzen.

4.1 Grundoperationen

Ganz allgemein muss eine dynamische Datenstruktur eine Menge S von Objekten verwalten und verschiedene Operationen ausführen können. Dazu sei nun im Folgenden:

S Menge von Objekten

k Wert eines Elements („Schlüssel“)

x Zeiger auf ein Element

NIL spezieller, „leerer“ Zeiger

SEARCH(S, k): „Suche in S nach k“

Beschreibung: Durchsuche die Menge S nach einem Element vom Wert k.

Ausgabe: Zeiger x, falls x existent. NIL, falls kein Element den Wert k hat.

INSERT(S, x): „Füge x in S ein“

Beschreibung: Erweitere S um das Element, das unter der Adresse x steht.

DELETE(S, x): „Entferne x aus S“

Beschreibung: Lösche das unter der Adresse x stehende Element aus S.

MINIMUM(S): „Suche das Minimum in S“

Beschreibung: Finde in S ein Element von kleinstem Wert. (*Annahme:* Die Werte lassen sich vollständig vergleichen!)

Ausgabe: Zeiger x auf solche ein Element.

MAXIMUM(S): „Suche das Maximum in S“

Beschreibung: Finde in S ein Element von größtem Wert. (*Annahme:* Die Werte lassen sich vollständig vergleichen!)

Ausgabe: Zeiger x auf solch ein Element.

PREDECESSOR(S, x): „Finde das nächstkleinere Element“

Beschreibung: Für ein in x stehendes Element in S, bestimme ein Element von nächstkleinerem Wert in S.

Ausgabe: Zeiger y auf Element. NIL, falls x Minimum von S angibt.

SUCCESSOR(S, x): „Finde das nächstgrößere Element“

Beschreibung: Für ein in x stehendes Element in S, bestimme ein Element von nächstgrößerem Wert in S.

Ausgabe: Zeiger y auf Element. NIL, falls x Maximum von S angibt.

Wie nimmt man das vor und wie lange dauert das in Abhängigkeit von der Größe von S? Da es sich um unsortierte Unterlagen handelt, muss immer jedes Element angeschaut werden, also: $O(n)$. Mit sortierten Unterlagen ginge dies schneller. Zum Vergleich: $O(n)$ ist lineare Zeit und somit für unsere Zwecke eher langsam; man muss jedes Element betrachten. Im Gegensatz dazu is $O(1)$ sehr schnell. Wir benötigen immer gleich lange, egal wie viele Objekte wir besitzen. Eine Zwischenstufe erreicht man durch wiederholtes Halbieren. Die dadurch erhaltene logarithmische Laufzeit ($O(\log n)$) ist immer noch schnell.

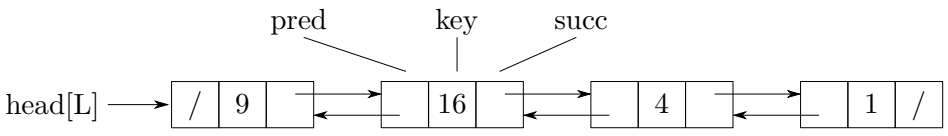
4.2 Stapel und Warteschlange

Bereits in Abschnitt 3.4 wurden Stapel und Warteschlange als Datenstruktur vorgestellt. Wir möchten an dieser Stelle an die Basisoperationen erinnern:

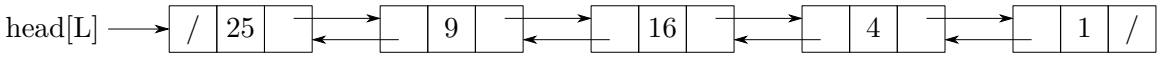
- ENQUEUE(Q, x): Hängt Element x an Schlange Q.
- DEQUEUE(Q): Entfernt das vorderste Element aus Q und gibt es aus.
- STACK-EMPTY(S): Prüft, ob der gegebene Stapel S leer ist.
- PUSH(S, x): Legt Element x auf den Stapel S
- POP(S): Entfernt oberstes Element vom Stapel S und gibt es aus.

4.3 Verkettete Listen

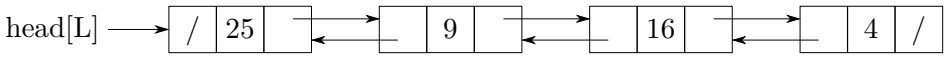
Die Idee hinter verketteten Listen ist, dass die Objekte nicht explizit in aufeinanderfolgende Speicherzellen angeordnet sind, sondern mittels Zeigern von anderen Objekten erreichbar sind. Speziell bei doppelt verketteten Listen besitzt ein Objekt einen Schlüssel (*emph*) und jeweils einen Zeiger (*Pointer*) auf den Vorgänger (*Predecessor*) und Nachfolger (*Successor*). Des weiteren gibt es noch einen Zeiger auf das Objekt, welches keinen Vorgänger besitzt; der Kopf (*head*) der Liste. Ein Beispiel für eine doppelt verkettete Liste ist in Abbildung 4.1a angegeben.



(a) Beispiel für eine doppelt verkettete Liste



(b) Füge der Liste aus Abb. 4.1a vorne das Element mit Schlüssel 25 ein.



(c) Finde in der Liste aus Abb. 4.1b ein Element mit dem Schlüssel 1 und lösche es.

Abbildung 4.1: Struktur einer doppelt verketteten Liste.

Möchte man ein Objekt x einfügen, so wird der Nachfolger von x der Kopf der Liste, der Vorgänger vom Kopf der Liste wird auf x gesetzt und schließlich der neue Kopf auf x gesetzt. Der zu dieser Operation gehörige Pseudocode ist in Algorithmus 4.1 zu finden. Ein Beispiel einer Durchführung von List-Insert ist in Abbildung 4.2b zu sehen.

Sucht man nun einen Schlüssel x in einer Liste, so kann man über die Nachfolger der Objekte das gewünschte Element suchen (siehe auch Pseudocode 4.2). Möchte man dieses Element nun auch löschen, so müssen nur der Nachfolger vom Vorgänger und der Vorgänger vom Nachfolger neu gesetzt werden (siehe dazu Pseudocode 4.3). Ein Beispiel für das Löschen eines Elements ist in Abbildung 4.2c zu sehen.

```
1: function LIST-INSERT(L, x)
2:     succ[x] ← head[L]
3:     if head[L] ≠ NIL then
4:         pred[head[L]] ← x
5:     head[L] ← x
6:     pred[x] ← NIL
```

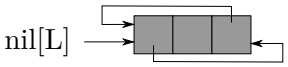
Pseudocode 4.1: Einfügen eines Elementes x in eine doppelt verkettete Liste L. Die Laufzeit beträgt $O(1)$.

```
1: function LIST-SEARCH(L, k)
2:     x ← head[L]
3:     while x ≠ NIL und key[x] ≠ k do
4:         x ← succ[x]
5:     return x
```

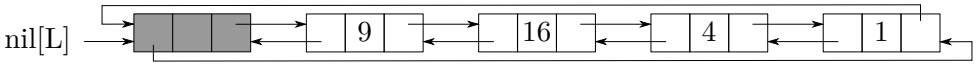
Pseudocode 4.2: Suchen eines Schlüssels k in einer doppelt verketteten Liste L. Die Laufzeit beträgt $O(n)$.

```
1: function LISTE-DELETE(L, x)
2:     if pred[x] ≠ NIL then
3:         succ[pred[x]] ← succ[x]
4:     else
5:         head[L] ← succ[x]
6:     if succ[x] ≠ NIL then
7:         pred[succ[x]] ← pred[x]
```

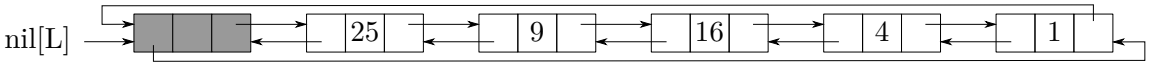
Pseudocode 4.3: Entfernen eines Elementes x aus einer doppelt verketteten Liste L. Die Laufzeit beträgt $O(1)$.



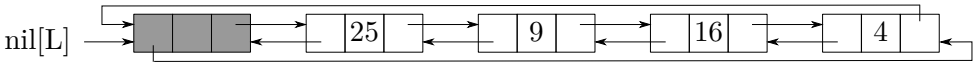
(a) Beispiel für eine leere, doppelt verkettete Liste



(b) Beispiel für eine doppelt verkettete Liste mit zyklischer Struktur und Wächter



(c) Füge der Liste aus Abb. 4.2b vorne das Element mit Schlüssel 25 ein.



(d) Finde in der Liste aus Abb. 4.2c ein Element mit dem Schlüssel 1 und lösche es.

Abbildung 4.2: Alternative: Zyklische Struktur einer doppelt verketteten Liste mit „Wächter“ nil[L].

Eine Variation doppelt verketteter Listen ist die zyklische doppelt verkettete Liste. In Abbildung 4.2 ist eine solche Struktur mit einem *Wächter* abgebildet. Bei dieser Struktur entfällt beispielsweise beim Einfügen in die Liste der Test, ob die Liste leer ist (siehe Pseudocode 4.4). Da jedes Element sowohl Vorgänger als auch Nachfolger besitzt, kann auch das Löschen aus der Liste deutlich einfach erfolgen (siehe Pseudocode 4.6).

```
1: function LIST-INSERT'(L, x)
2:     succ[x] ← succ[nil[L]]
3:     pred[succ[nil[L]]] ← x
4:     succ[nil[L]] ← x
5:     pred[x] ← nil[L]
```

Pseudocode 4.4: Einfügen eines Elementes x in eine doppelt verkettete Liste L mit zyklischer Struktur und „Wächter“ nil[L].

```
1: function LIST-SEARCH'(L, k)
2:     x ← succ[nil[L]]
3:     while x ≠ nil[L] und key[x] ≠ k do
4:         x ← succ[x]
5:     return x
```

Pseudocode 4.5: Suchen eines Schlüssels k in einer doppelt verketteten Liste L mit zyklischer Struktur und „Wächter“ nil[L].

```
1: function LIST-DELETE'(L, x)
2:     succ[pred[x]] ← succ[x]
3:     pred[succ[x]] ← pred[x]
```

Pseudocode 4.6: Entfernen eines Elementes x aus einer doppelt verketteten Liste L mit zyklischer Struktur und „Wächter“ nil[L].

Wie bereits erwähnt, müssen die Elemente im Speicher nicht aufeinanderfolgen, sondern dürfen verteilt sein. Die Vorgänger und Nachfolger zeigen also als Pointer auf die Stelle des vorherigen beziehungsweise nachfolgenden Elements. Ein Beispiel wie das aussehen kann ist in Abbildung 4.3 gegeben. (Randnotiz: Mit Hilfe doppel-verketteter Listen lässt sich der Hierholzer-Algorithmus zum Finden von Eulertouren mit Laufzeit $O(n+m)$ implementieren. Dies zu beweisen stellt eine gute Übung dar.)

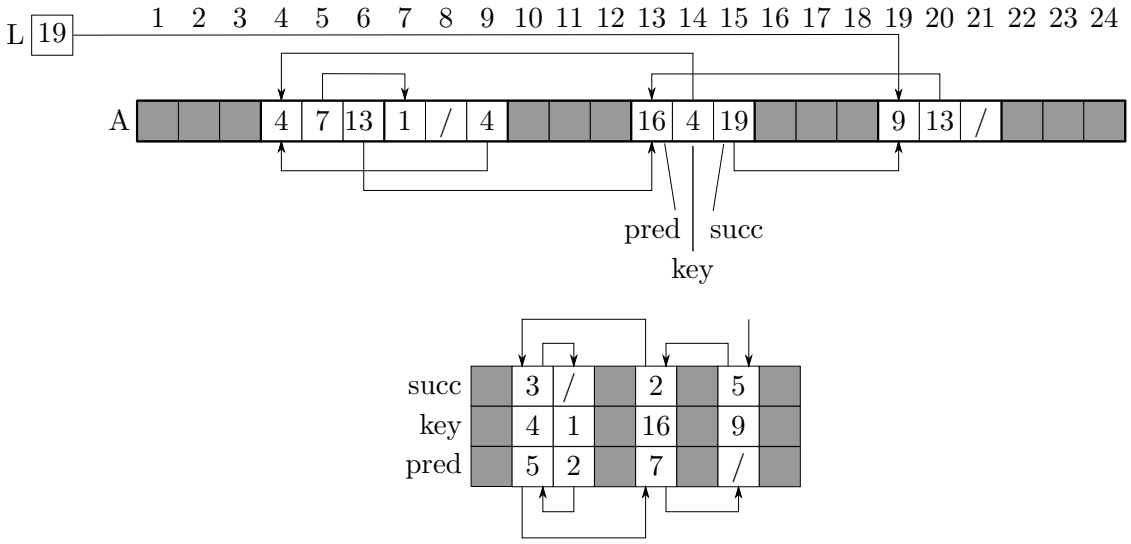


Abbildung 4.3: Speicherung kann irgendwo erfolgen.

4.4 Binäre Suche

Schauen wir uns folgendes Spiel an: Spieler A denkt sich eine Zahl zwischen 65 und 128 aus. Spieler B soll nun mit möglichst wenigen Versuchen diese Zahl zu erraten. Nach

Eingabe: Sortiertes Array S mit Einträgen $S[I]$, Suchwert x, linke Randposition left und rechte Randposition right.

Ausgabe: Position von x zwischen Arraypositionen left und right, falls existent.

```
1:  function BINARY-SEARCH(S, x, left, right)
2:      while left ≤ right do
3:          middle ← ⌊(left+right)/2⌋
4:          if S[middle] = x then
5:              return middle
6:          else if S[middle] > x then
7:              right ← middle − 1
8:          else
9:              left ← middle + 1
10:     return „x nicht gefunden!“
```

Algorithmus 4.1: Binäre Suche eines Wertes x in einem sortierten Array S zwischen den Positionen left und right.

einem erfolglosen Rateversuch verrät A, ob die gesuchte Zahl kleiner oder größer ist. Eine einfache Strategie dieses Spiel zu meistern ist, nach jedem Versuch die Anzahl möglicher Zahlen zu halbieren. Ein Algorithmus, der genau diese Strategie verfolgt, ist in Algorithmus 4.1 gegeben. Den nachfolgenden Satz geben wir zu Übungszwecken ohne Beweis an.

Satz 4.2. *Die binäre Suche (s. Alg. 4.1) terminiert in* $O(\log(\mathit{left} - \mathit{right}))$ *Schritten (für* $\mathit{right} > \mathit{left}$*).*

4.5 Binäre Suchbäume

Mit der Idee der binären Suche lässt sich nun eine Datenstruktur entwickeln, die darauf abzielt logaithmische Laufzeiten für das Finden und Löschen von Daten zu erhalten. Diese Datnstruktur verhält sich ähnlich zu einer Liste, wobei nun jedes Element zwei „Nachfolger“ (*Kinder*) bekommt: ein linkes Kind mit kleineren Schlüsselwerten und ein rechtes Kind mit größeren Schlüsselwerten. Die formale Definition für binäre Suchbäume ist wie folgt.

Definition 4.3 (Gerichteter Baum)**.**

(1) Ein gerichteter Graph $D = (V, A)$ besteht aus einer endlichen Menge V von Knoten v und einer endlichen Menge A von gerichteten Kanten $a = (v, w)$ (v ist Vorgänger von w).

(2) Ein gerichteter Baum $B = (V, T)$ hat folgende Eigenschaften:

(i) Es gibt einen eindeutigen Knoten w ohne Vorgänger.

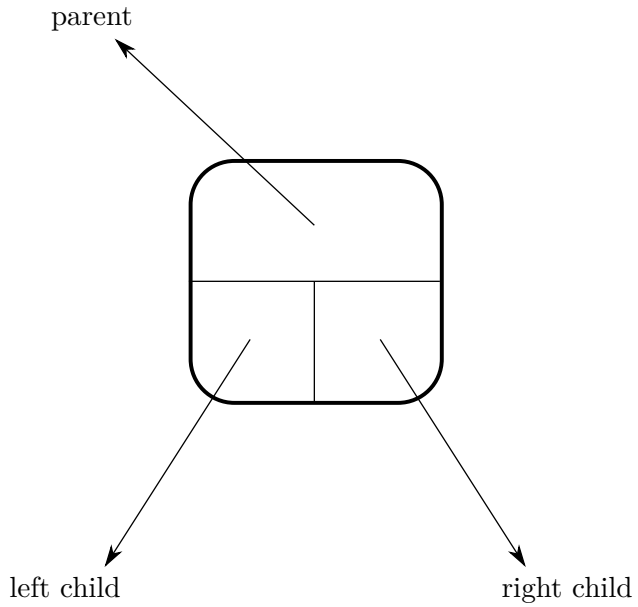


Abbildung 4.4: Struktur eines Elementes eines binären Suchbaumes

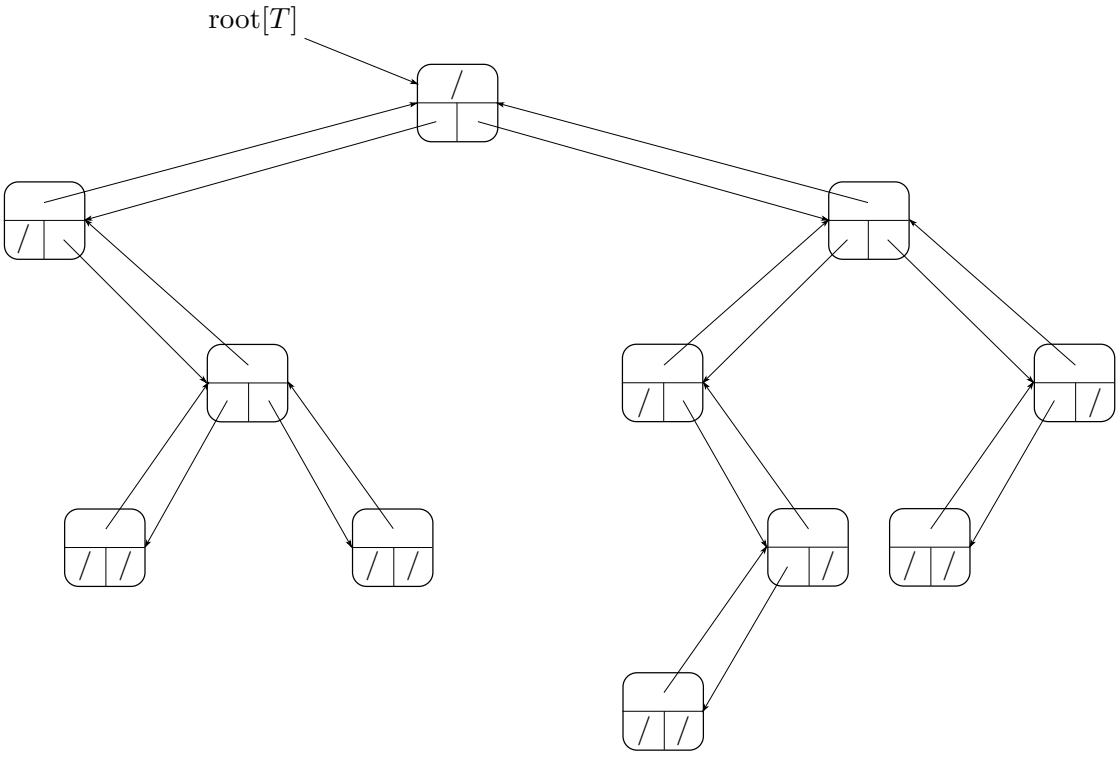
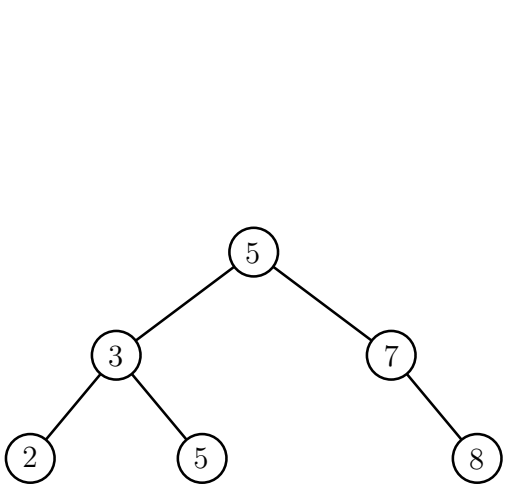
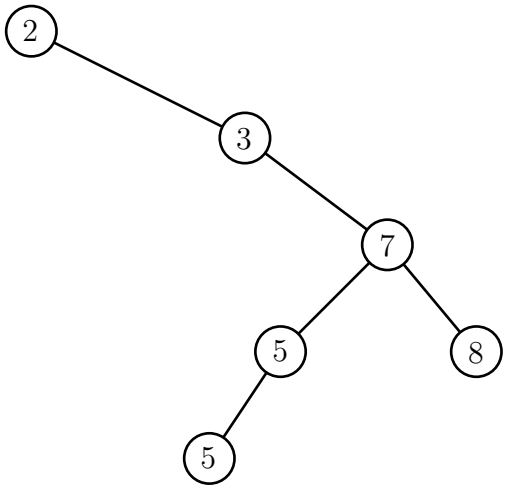


Abbildung 4.5: Struktur eines binären Suchbaumes



(a) Beispiel für einen balancierten binären Suchbaum



(b) Beispiel für einen nicht balancierten binären Suchbaum

Abbildung 4.6: Ordnungsstruktur auf binären Suchbäumen: Kleinere (bzw. nicht größere) Zahlen befinden sich im linken, größere Zahlen im rechten Teilbaum.

*(ii) Jeder Knoten, außer w, ist auf einem eindeutigen Weg von w aus erreichbar. D. h. insbesondere, dass $v \in V$ einen eindeutigen Vorgänger (*auch *Vaterknoten) hat.*

(3) Die Höhe eines gerichteten Baumes ist die maximale Länge eines gerichteten Weges von der Wurzel.

*(4) Ein binärer Baum ist ein gerichteter Baum, in dem jeder Knoten höchstens zwei Nachfolger (*Kindknoten*) hat. Einer ist der „linke“ $l[v]$, der andere der „rechte“ $r[v]$.*

(5) Ein binärer Baum heißt voll*, wenn jeder Knoten zwei oder keinen Kindknoten hat.*

(6) Ein binärer Baum heißt vollständig*, wenn zusätzlich jedes Blatt den gleichen Abstand zu Wurzel besitzt.*

(7) Ein Knoten ohne Kindknoten heißt Blatt*.*

(8) Der Teilbaum eines Knotens v ist durch die Menge der von v aus erreichbaren Knoten und der dabei verwendeten Kanten definiert. Der linke Teilbaum ist der Teilbaum von $l[v]$, der Rechte der von $r[v]$.

(9) In einem binären Suchbaum hat jeder Knoten v einen Schlüsselwert $S[v]$ und es gilt:

- *$S[u] \leq S[v]$ für Knoten u im linken Teilbaum von v.*
- *$S[u] > S[v]$ für Knoten u im rechten Teilbaum von v.*

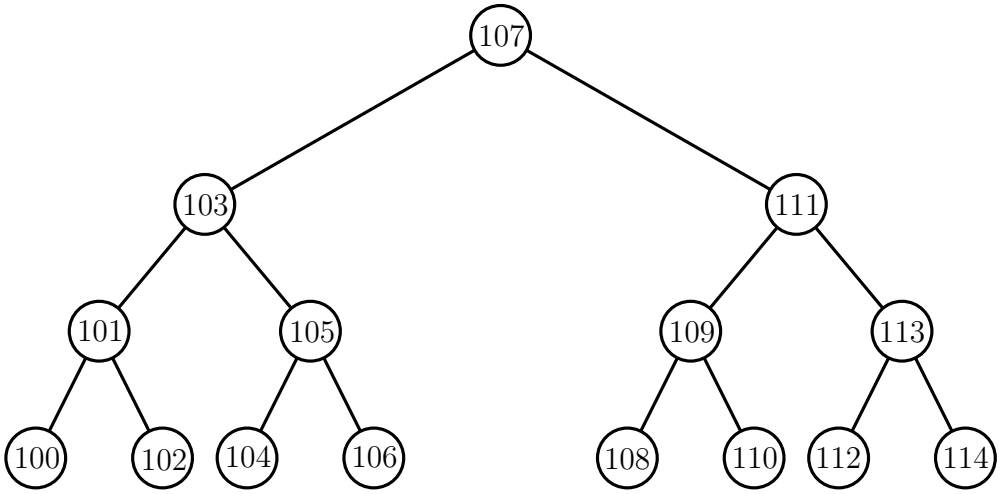


Abbildung 4.7: Beispiel für einen vollständigen binären Suchbaum.

Ein Beispiel für einen binären Suchbaum ist in Abbildung 4.7. In einer solchen Struktur lassen sich nun Elemente über einfache Algorithmen finden. Um das *Minimum* zu finden, gehen wir so lange in den linken Teilbaum bis wir kein linkes Kind mehr besitzen (vgl. Pseudocode 4.7). Analog dazu können wir das *Maximum* finden, indem wir iterativ in den rechten Teilbaum gehen, bis kein rechtes Kind mehr existiert (vgl. Pseudocode 4.8). Um nach einem *Schlüssel x zu suchen*, vergleicht man x mit dem aktuellen Knoten und wiederholt die Prozedur im linken (rechten) Teilbaum, wenn x kleiner (größer) als der aktuelle Knoten ist (vgl. Pseudocode 4.9). Abseits der bereits genannten Funktionen können im Baum auch einfach der Vorgänger (*Predecessor*) und Nachfolger (*Successor*) gefunden werden. Dazu sucht man das Maximum im linken Teilbaum für den Vorgänger und das Minimum im rechten Teilbaum für den Nachfolger. Der Pseudocode zum Finden des Nachfolgers ist in Pseudocode 4.10 gegeben.

```
1: function TREE-MINIMUM(x)
2:     while left[x] ≠ NIL do
3:         x ← left[x]
4:     return x
```

Pseudocode 4.7: Suchen des Minimums in einem binären Suchbaum, der durch den Wurzelknoten x gegeben ist.

Satz 4.4. *Suchen, Minimum, Maximum, Nachfolger und Vorgänger können in einem binären Suchbaum der Höhe h in $O(h)$ durchlaufen werden.*

Beweis. Klar, der Baum wird nur vertikal abwärts durchlaufen! □

Bisher haben wir nur Operationen betrachtet, die nur auf bestehenden Elementen operieren. Nun wollen wir noch Elemente einfügen und entfernen.

```
1: function TREE-MAXIMUM(x)
2:     while right[x] ≠ NIL do
3:         x ← right[x]
4:     return x
```

Pseudocode 4.8: Suchen des Maximums in einem binären Suchbaum, der durch den Wurzelknoten x gegeben ist.

```
1: function ITERATIVE-TREE-SEARCH(x, k)
2:     while x ≠ NIL und k ≠ key[x] do
3:         if k < key[x] then
4:             x ← left[x]
5:         else
6:             x ← right[x]
7:     return x
```

Pseudocode 4.9: Suchen eines Schlüssels k in einem binären Suchbaum, der durch den Wurzelknoten x gegeben ist.

```
1: function TREE-SUCCESSOR(x)
2:     if right[x] ≠ NIL then
3:         return TREE-MINIMUM(right[x])
4:     y ← parent[x]
5:     while y ≠ NIL und x = right[y] do
6:         x ← y
7:         y ← parent[y]
8:     return y
```

Pseudocode 4.10: Finden des Nachfolger-Schlüssels vom Schlüssel vom Knoten x.

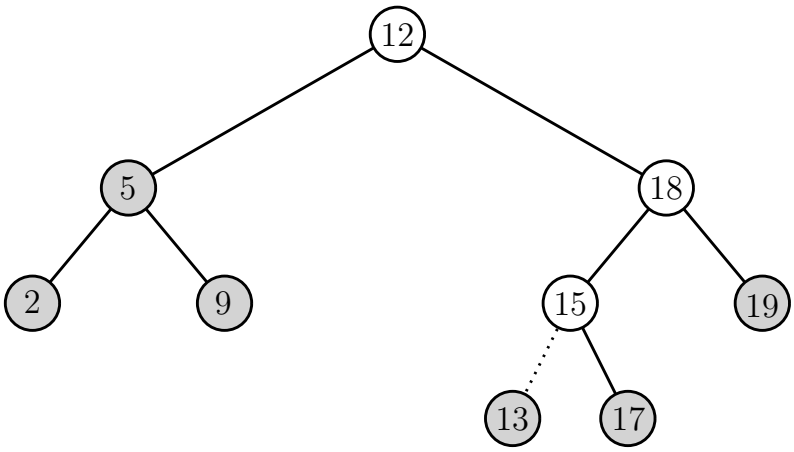


Abbildung 4.8: Einfügen der 13 in einen Suchbaum. Die weißen Knoten werden bei der Suche nach der korrekten Stelle abgelaufen.

```
 1: function TREE-INSERT(T, z)
 2:     y ← NIL
 3:     x ← root[T]
 4:     while x ≠ NIL do
 5:         y ← x
 6:         if key[z] < key[x] then
 7:             x ← left[x]
 8:         else
 9:             x ← right[x]
10:     parent[z] ← y
11:     if y = NIL then
12:         root[T] ← z
13:     else if key[z] < key[y] then
14:         left[y] ← z
15:     else
16:         right[y] ← z
```

Pseudocode 4.11: Einfügen eines Knotens z in einen binären Suchbaum T.

Zum Einfügen eines Elements mit Schlüssel x gehen wir iterativ (wie bei einer Suche) in den linken Teilbaum, falls x kleiner ist und in den rechten Teilbaum, falls x größer ist. Ist man an dem Punkt angelangt, bei dem man nicht in den linken bzw. rechten Teilbaum gehen kann, so fügt man das Element an dieser freien Stelle ein. Eine formale Beschreibung ist in Pseudocode 4.11 zu finden.

Satz 4.5. *Einfügen in einen binären Suchbaum der Höhe h benötigt $O(h)$.*

Beweis. Klar, der Baum wird nur vertikal abwärts durchlaufen! □

Um ein Element zu löschen, müssen wir es zunächst finden. Sind wir bei dem gesuchten Element mit Schlüssel z, so können wir es nicht einfach entfernen. Es werden drei Fälle unterschieden: (1) Hat z keine Kinder, kann z einfach entfernt werden (siehe Abbildung 4.9). (2) Hat z genau ein Kind, dann rutscht das Kind an die Stelle von z (siehe ABbildung 4.10). (3) Hat z zwei Kinder, dann suchen wir den Nachfolger y von z, kopieren y an die Stelle von z und löschen die alte Position von y (ggf. rutschen hierbei das Kind von y nach oben; siehe Abbildung 4.11). Eine formale Beschreibung zum Löschen eines Elements ist in Pseudocode 4.12 zu finden.

Satz 4.6. *Löschen aus einem binären Suchbaum der Höhe h benötigt $O(h)$.*

Beweis. Klar, der Baum wird im Wesentlichen nur einmal durchlaufen! □

Wie man sieht hängen alle Operationen auf Bäumen von der Höhe des Baumes ab. Das kann im schlimmsten Fall $O(n)$ bedeuten, wie Abbildung 4.12 zeigt. Optimal wäre eine Laufzeit von $O(\log n)$. Die Frage ist also, wie man die Tiefe des Baumes auf $O(\log n)$ beschränken kann.

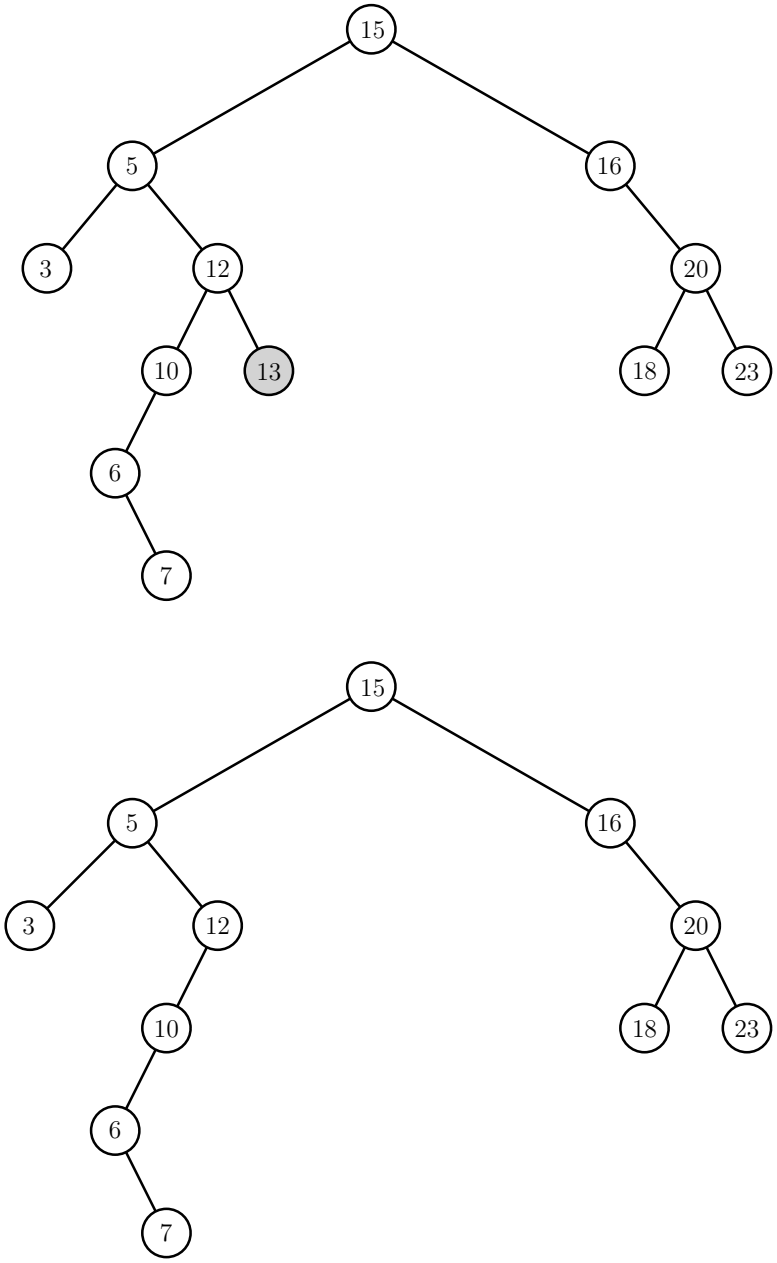


Abbildung 4.9: Entfernen eines Elementes ohne Kinder aus einem binären Suchbaum

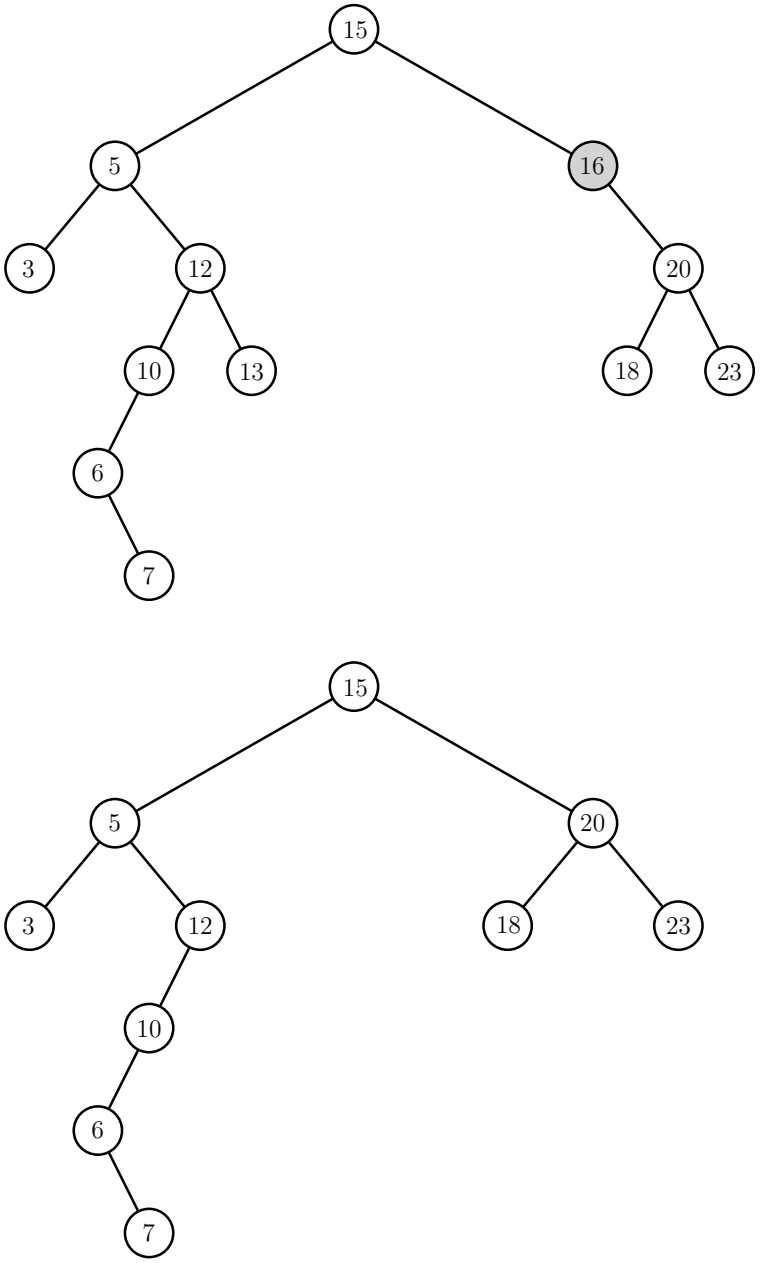


Abbildung 4.10: Entfernen eines Elementes mit einem Kind aus einem binären Suchbaum

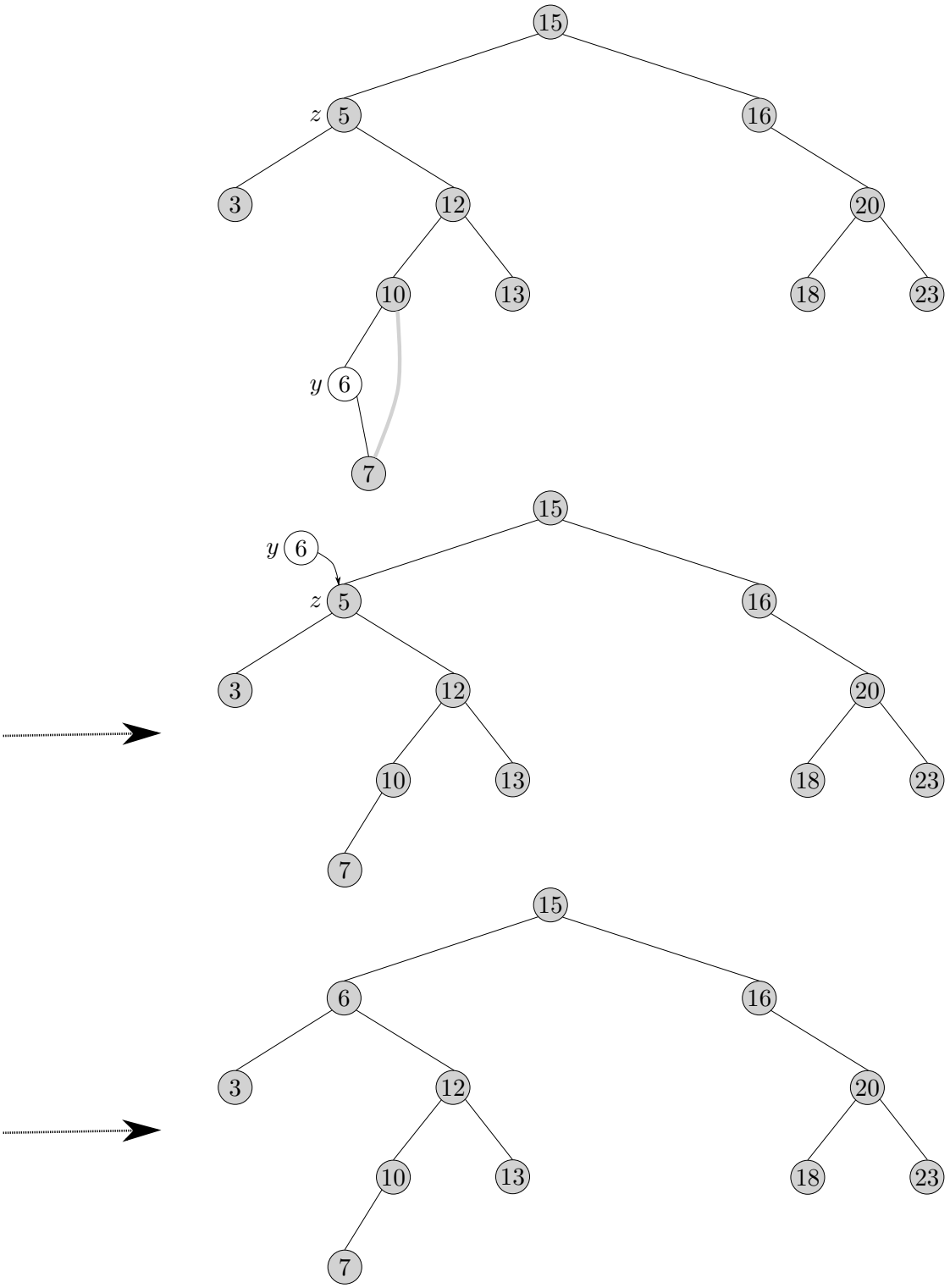


Abbildung 4.11: Entfernen eines Elementes mit zwei Kindern aus einem binären Suchbaum

```
 1: function TREE-DELETE(T, z)
 2:     if left[z] = NIL oder right[z] = NIL then
 3:         y ← z
 4:     else
 5:         y ← TREE-SUCCESSOR(z)
 6:     if left[y] ≠ NIL then
 7:         x ← left[y]
 8:     else
 9:         x ← right[y]
10:     if x ≠ NIL then
11:         parent[x] ← parent[y]
12:     if parent[y] = NIL then
13:         root[T] ← x
14:     else if y = left[parent[y]] then
15:         left[parent[y]] ← x
16:     else
17:         right[parent[y]] ← x
18:     if y ≠ z then
19:         key[z] ← key[y]
20:         kopiere die Nutzdaten von y in z
21:     return y
```

Pseudocode 4.12: Entfernen eines Knotens z aus einem binären Suchbaum T.

4.6 AVL-Bäume

Bei binären Suchbäumen ist für die Laufzeit von Algorithmen auf selbigem die Höhe h entscheidend, da bspw. Einfügen und Entfernen von Elementen in bzw. aus einem binären Suchbaum in $O(h)$ läuft.

Wie kann die Höhe möglichst klein gehalten werden? Eine Idee ist, den Baum „*balanciert*“ aufzubauen, d.h. der linke und rechte Teilbaum soll gleich groß sein. Dies ist natürlich nicht immer möglich. Wir wollen also versuchen, den Baum „annähernd balanciert“ zu halten. Ein weiteres Problem ist, dass das Gewicht, d.h. die Anzahl an Knoten in einem Teilbaum, eine globale Eigenschaft ist. Über diese Eigenschaft den Baum balanciert zu halten, könnte also schwierig werden. Auf der anderen Seite ist das Gewicht für die Laufzeiten nicht wichtig, sondern die Tiefe bzw. Höhe des Baumes. Diese Idee führt zur folgenden Definition 4.7.

Definition 4.7 (AVL-Baum nach Adel'son-Vel'skij und Landis (1962))**.**

(1) Ein binärer Suchbaum ist höhenbalanciert*, wenn sich für jeden inneren Knoten v die Höhe der beiden Kinder von v um höchstens* 1 *unterscheidet.*

(2) Ein höhenbalancierter Suchbaum heißt auch AVL-Baum.

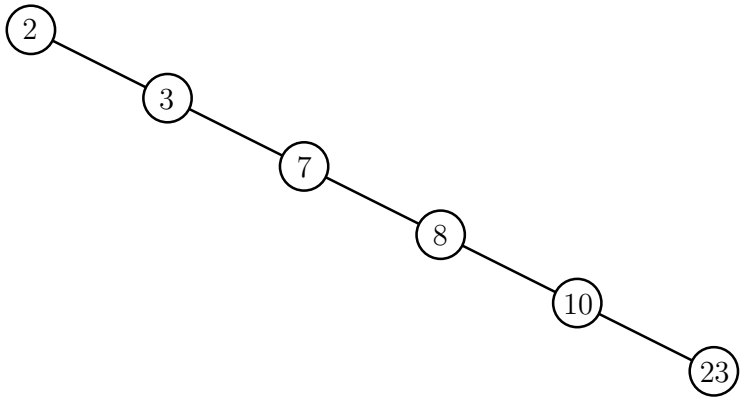
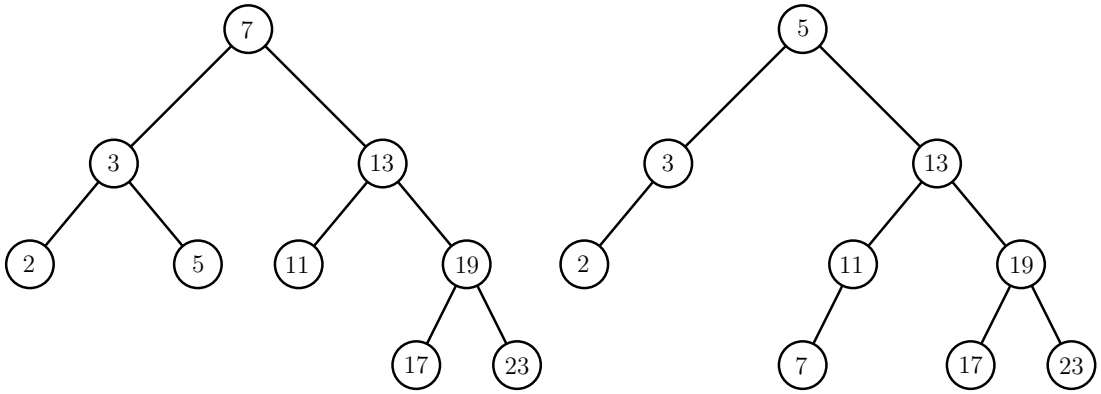


Abbildung 4.12: Schlechte Struktur, falls der binäre Suchbaum maximale Höhe hat.



(a) Gute Struktur, falls der binäre Suchbaum balanciert ist.

(b) Ebenfalls eine gute Struktur.

Abbildung 4.13: Gute Strukturen in Bezug auf die Höhe von binären Suchbäumen.

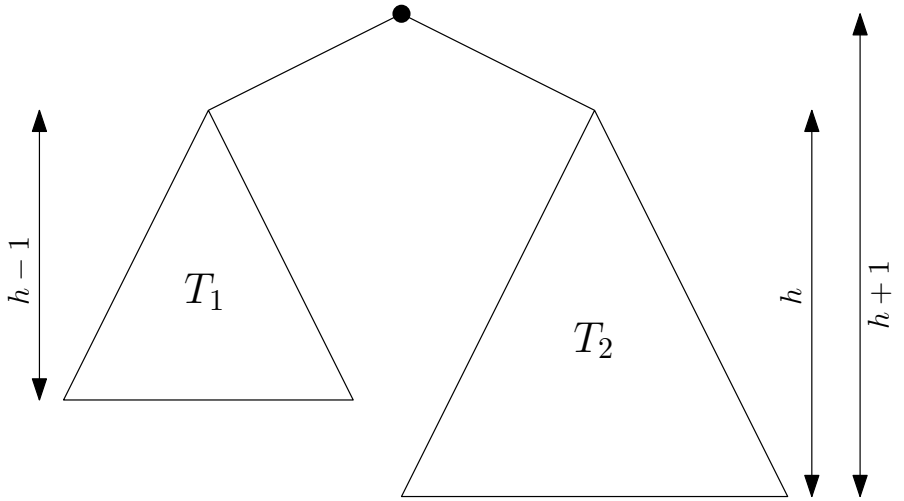


Abbildung 4.14: Ein allgemeiner höhenbalancierter Suchbaum

Abbildung 4.13 zeigt Beispiele für höhenbalancierte Bäume. Reicht die Eigenschaft der Höhenbalanciertheit aus, um logarithmische Höhe zu garantieren? Und wie kann man dafür Sorgen, dass die Eigenschaft beim Einfügen und Löschen von Elementen erhalten bleibt? Wir beobachten, dass, wenn ein Baum höhenbalanciert ist, alle seine Teilbäume höhenbalanciert sind. Die Höhenbalanciertheit ist somit eine rekursive Eigenschaft. Schauen wir uns zunächst folgende Fragen an: Wieviele Knoten braucht man mindestens für einen AVL-Baum der Höhe h? Das heißt, man untersucht nicht die Höhe in Abhängigkeit von der Knotenzahl, sondern die Knotenzahl in Abhängigkeit von der Höhe. Wenn n mindestens exponentiell in h wächst, dann wächst h höchstens logarithmisch in n.

Betrachte allgemein Abbildung 4.14. Mit

$$\begin{aligned} n(1) &= 1 \\ n(2) &= 2 \\ n(h+1) &= 1 + n(h) + n(h-1) \end{aligned}$$

erhält man die Funktionswerte, die in Tabelle 4.1 zu sehen sind. Dies ist sehr ähnlich zu einer Funktion f mit

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(h+1) &= f(h) + f(h-1), \end{aligned}$$

welche die Funktionswerte liefert, die in Tabelle 4.2 zu sehen sind.

Satz 4.8. *Die Höhe eines AVL-Baumes mit n Knoten ist $O(\log(n))$.*

Beweis. Statt einer oberen Schranke für die Höhe eines AVL-Baumes zeigt man zunächst eine untere Schranke für die Zahl der Knoten eines AVL-Baumes! Sei $n(h)$ die kleinste

h	$n(h)$
1	1
2	2
3	4
4	7
5	12
6	20
7	33
8	54
...	...

Tabelle 4.1: Mindestanzahl benötigter Knoten, $n(h)$, für einen AVL-Baum der Höhe h.

h	$f(h)$
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21
...	...

Tabelle 4.2: Fibonacci-Zahlen

Zahl von Knoten eines AVL-Baumes der Höhe h. Man beobachtet

$$n(0) = 1 \qquad \text{(ein Knoten erforderlich)}$$
$$n(1) = 2 \qquad \text{(zwei Knoten erforderlich wegen Höhe)}$$

Noch einmal gilt

$$\underset{}{n(h)} = \underset{\substack{\uparrow \\ \text{Wurzelknoten}}}{1} + \underset{\substack{\uparrow \\ \text{Teilbaum}}}{n(h-1)} + \underset{\substack{\uparrow \\ \text{Teilbaum}}}{n(h-2)}.$$

Man zeigt per Induktion, dass

$$n(h) \geq 2^{\frac{h}{2}-1}$$

gilt.

Induktionsanfang:
Sei $h = 1$, dann gilt

$$n(h) = n(1) = 1 \geq 2^{-\frac{1}{2}} = 2^{\frac{h}{2}-1}.$$

Sei $h = 2$, dann gilt

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

Induktionsannahme:
Es gelte die Behauptung für alle $h \in \{0, ..., k\}$.

Induktionsschritt:
Es ist z. z.: Die Behauptung gilt für $h = k + 1$. Betrachte also

$$n(k+1) = 1 + n(k) + n(k-1).$$

Da $n(h)$ monoton wächst (für größere Höhe braucht man mindestens so viele Knoten wir für niedrigere), gilt

$$n(k) \geq n(k-1).$$

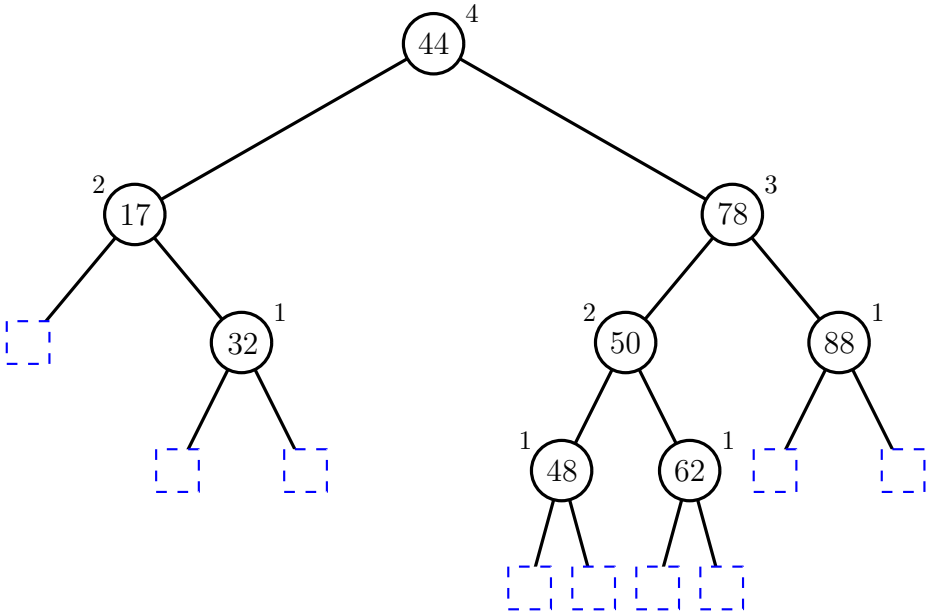


Abbildung 4.15: Beispiel für einen höhenbalancierten AVL-Baum.

Also ist

$$\begin{aligned} n(k+1) &\geq 1 + 2 \cdot n(k-1) \\ &\geq 1 + 2 \cdot 2^{\frac{k-1}{2}-1} && \text{(Induktionsannahme)} \\ &= 1 + 2^{\frac{k+1}{2}-1} \end{aligned}$$

Damit gilt die Behauptung auch für $h = k + 1$.

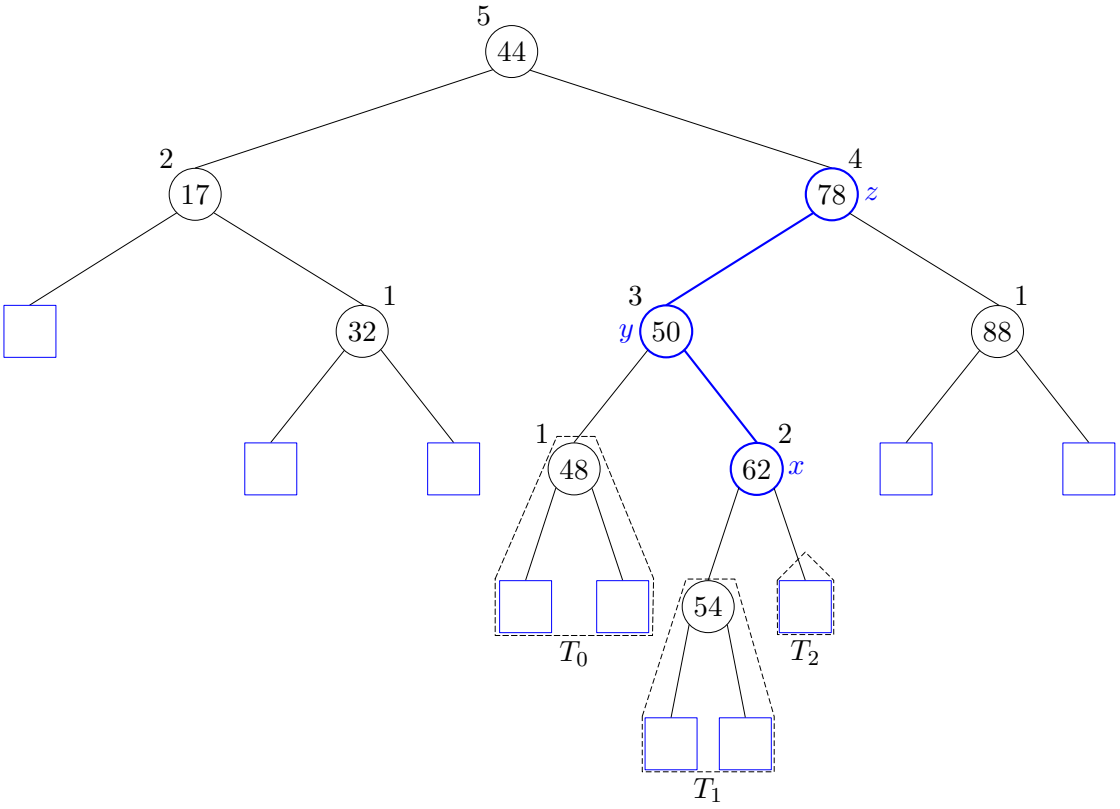
Induktionsschluss:
Für alle $h \in \mathbb{N}$ gilt $n(h) \geq 2^{\frac{h}{2}-1}$! Damit ist auch

$$\log_2(n(h)) \geq \frac{h}{2} - 1 \qquad \text{oder}$$
$$h \leq 2(\log_2(n) + 1),$$

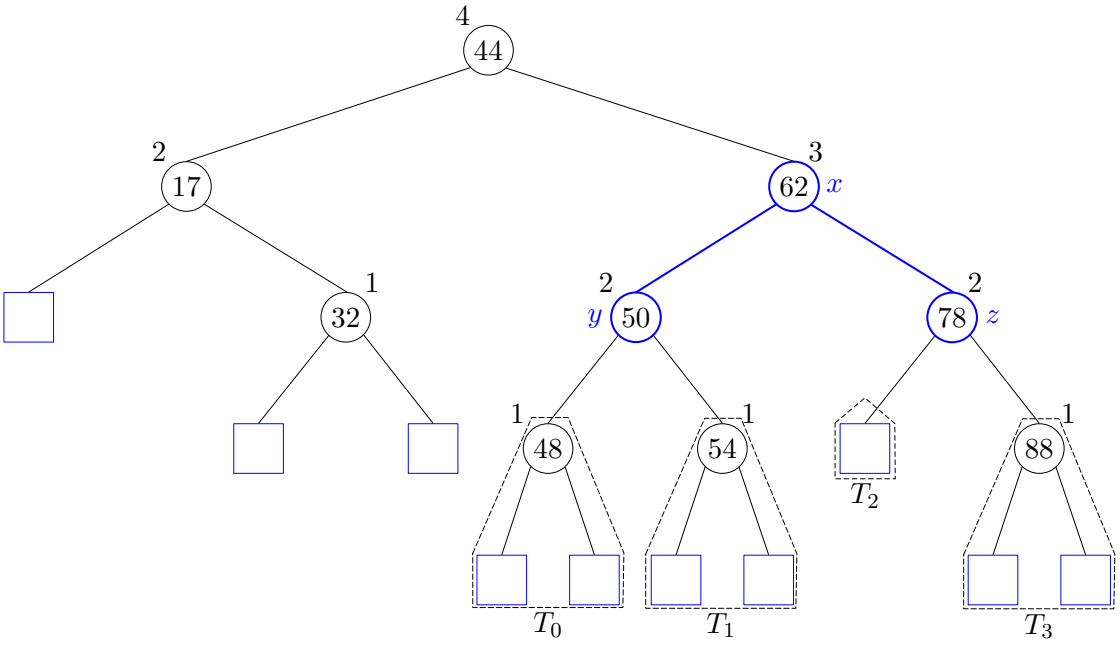
denn $n(h)$ ist die kleinstmögliche Knotenzahl bei Höhe h.
Damit hat ein AVL-Baum mit n Knoten höchstens die Höhe $2(\log_2(n)+1)$, also $O(\log(n))$. □

Es bleibt ein Problem offen: Wie bewahrt man Höhenbalanciertheit beim Einfügen und Löschen in bzw. aus einem binären Suchbaum? Wir beobachten: Höhenbalanciertheit ändert sich beim Einfügen einzelner Elemente nur wenig - und lokal!

Satz 4.10. *Mithilfe von Algorithmus 4.9 kann man einen AVL-Baum auch nach einer Einfüge-Operation höhenbalanciert halten. Die Zeit dafür ist $O(1)$.*



(a) Unbalancierter AVL-Baum aus Abbildung 4.15 nach dem Einfügen der 54.



(b) Höhenbalancierter AVL-Baum nach der Rotation.

Abbildung 4.16: Das Einfügen eines neuen Knotens in einen AVL-Baum mit einer Rotation zur Wiederherstellung der Höhenbalanciertheit.

Eingabe: Knoten x eines binären Suchbaumes T, Vaterknoten y und Großvaterknoten z

Ausgabe: Binärer Suchbaum T nach Umstrukturierung mit x, y und z

1: **function** RESTRUCTURE(x, y, z)
2: Sei (a, b, c) die Größensortierung der Knoten x, y und z.
3: Sei (T_0, T_1, T_2, T_3) die Größensortierung der vier Teilbäume unter x, y und z, die nicht Wurzeln x, y oder z haben.
4: Ersetze den Teilbaum mit Wurzel z durch einen neuen Teilbaum mit Wurzel b.
5: Setze a als linkes Kind von b, mit T_0 und T_1 als linken und rechten Teilbaum unter a.
6: Setze c als rechtes Kind von b, mit T_2 und T_3 als linken und rechten Teilbaum unter c.
7: **return**

Algorithmus 4.9: Algorithmus, der für einen Knoten x eines binären Suchbaumes, seinen Vaterknoten y und seinen Großvaterknoten z eine Rotation durchführt, sodass der Teilbaum z wieder höhenbalanciert ist.

Beweis. Annahme: Durch Hinzufügen eines Knotens v ist der Baum unbalanciert geworden. Sei z der nach dem Einfügen niedrigste unbalancierte Vorfahre von v. Sei y das Kind von z, das Vorfahre von v ist; y muss zwei höher sein, als das andere Kind von z. Sei x das Kind von y, das im selben Teilbaum wie v liegt.

Jetzt ersetzt man die Teilstruktur z, y, x (drei Knoten untereinander) durch eine Teilstruktur mit zwei Knoten unter einem. Es ist z. z.: Danach ist der Baum ein AVL-Baum! Hierfür betrachte man die möglichen Anordnungen von x, y und z:

(1) $x \leq y \leq z$ (s. Abb. 4.17)

(2) $x \leq z \leq y$

(3) $y \leq x \leq z$ (s. Abb. 4.18)

(4) $y \leq z \leq x$

(5) $z \leq x \leq y$ (s. Abb. 4.19)

(6) $z \leq y \leq x$ (s. Abb. 4.20)

zu (1) Der Baum ist, wie in Abbildung 4.21 zu sehen, wieder höhenbalanciert.

zu (2) Dieser Fall kann nicht auftreten!

zu (3) Der Baum ist, wie in Abbildung 4.22 zu sehen, wieder höhenbalanciert.

zu (4) Dieser Fall kann nicht auftreten!

zu (5) Der Baum ist, wie in Abbildung 4.23 zu sehen, wieder höhenbalanciert.

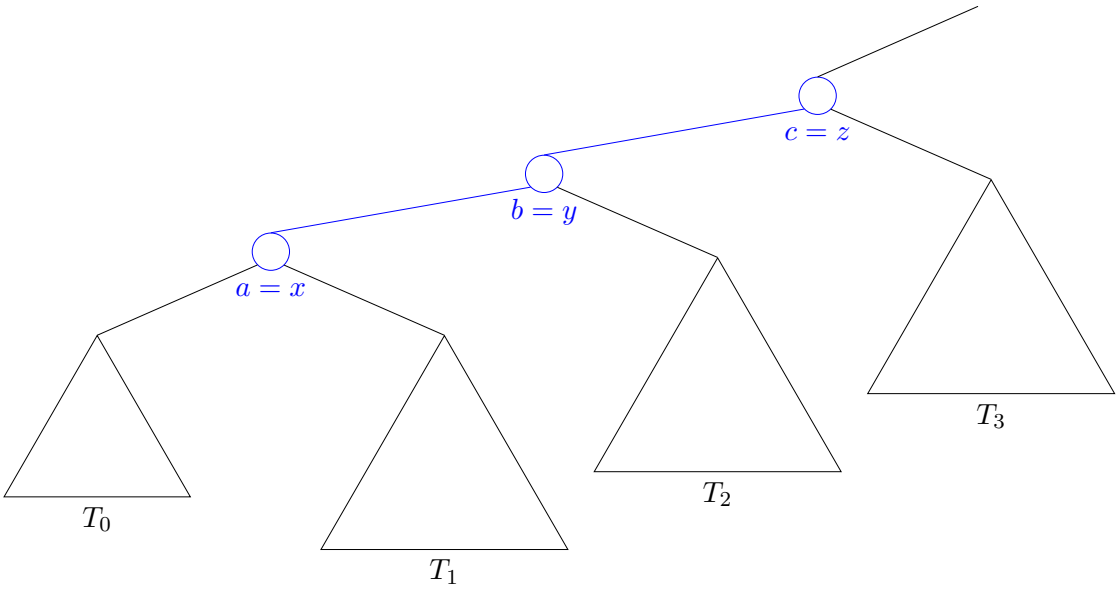


Abbildung 4.17: AVL-Baum-Anordnung zu (1) des Beweises von Satz 4.10.

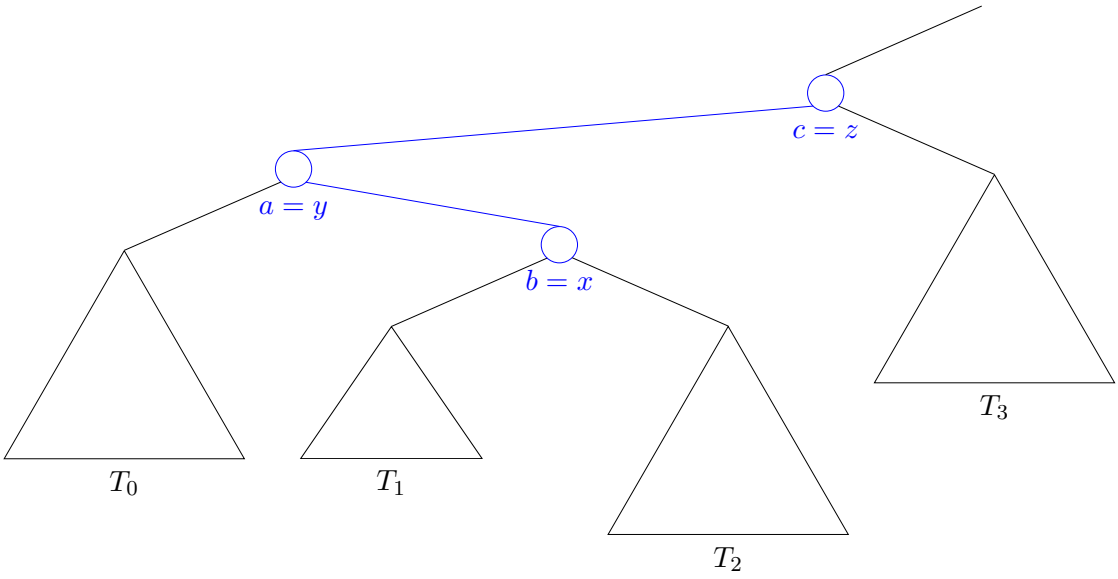


Abbildung 4.18: AVL-Baum-Anordnung zu (3) des Beweises von Satz 4.10.

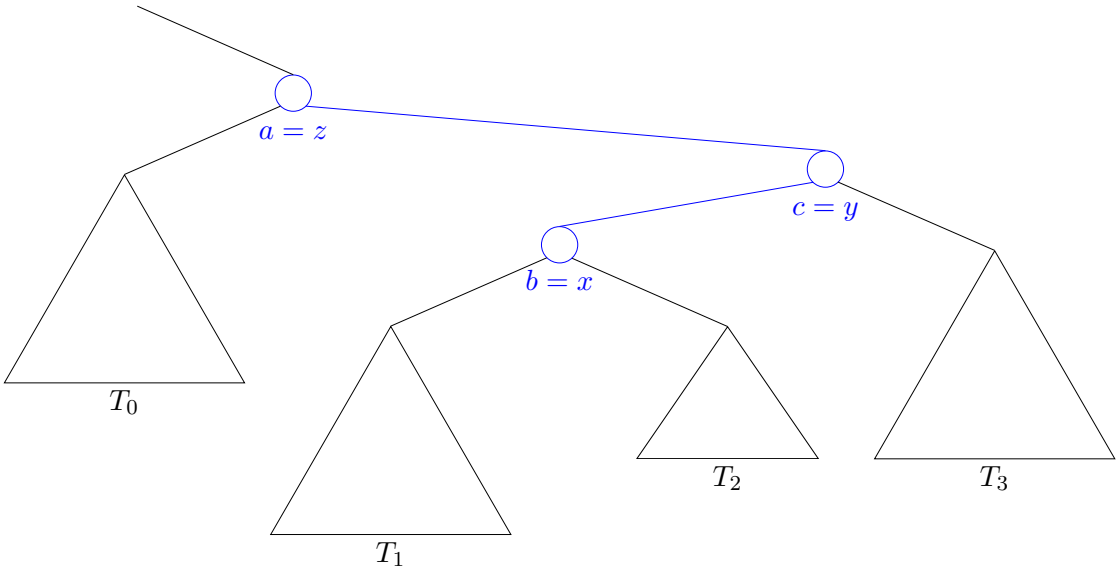


Abbildung 4.19: AVL-Baum-Anordnung zu (5) des Beweises von Satz 4.10.

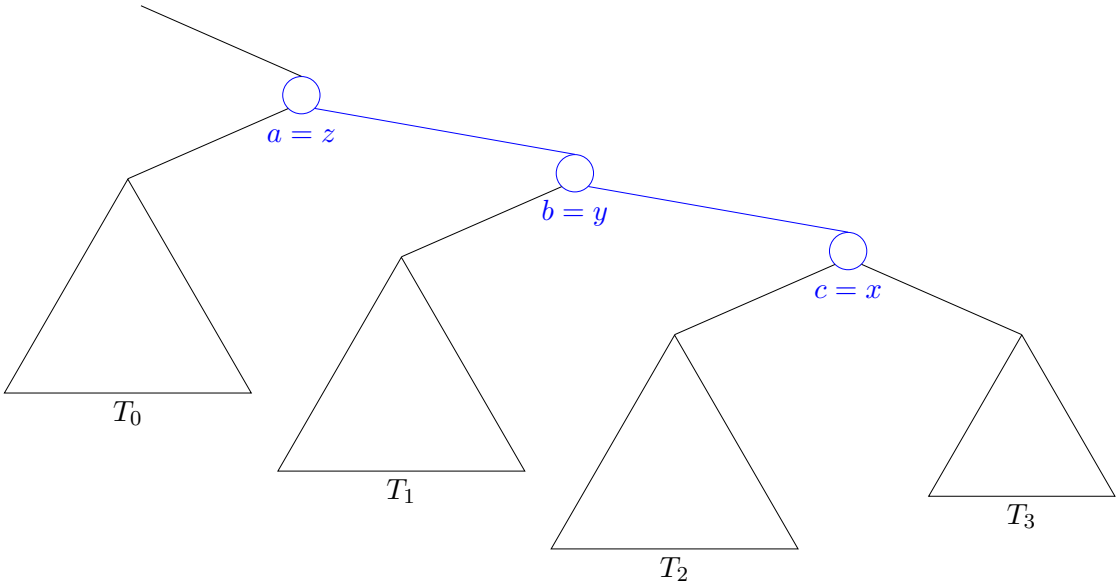


Abbildung 4.20: AVL-Baum-Anordnung zu (6) des Beweises von Satz 4.10.

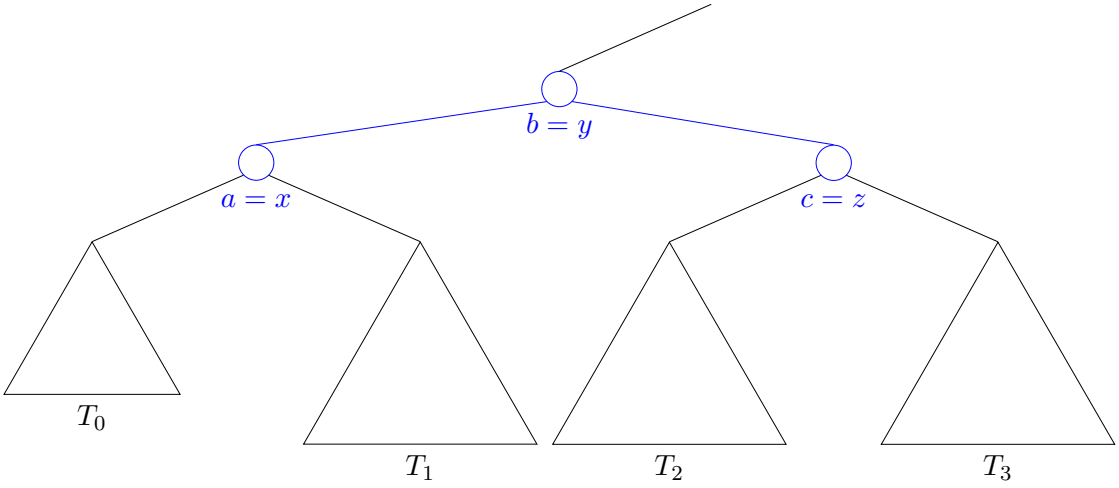


Abbildung 4.21: Höhenbalancierte AVL-Baum-Anordnung zu (1) des Beweises von Satz 4.10.

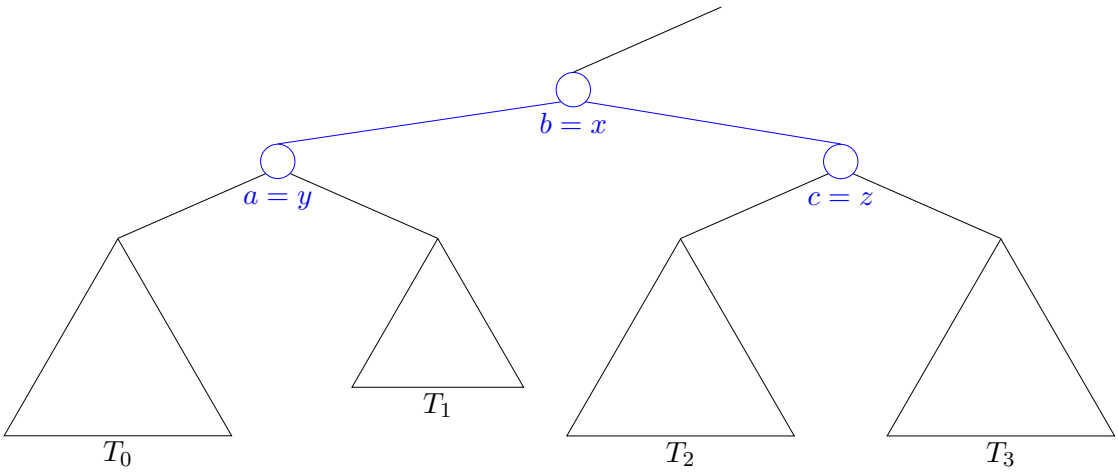


Abbildung 4.22: Höhenbalancierte AVL-Baum-Anordnung zu (3) des Beweises von Satz 4.10.

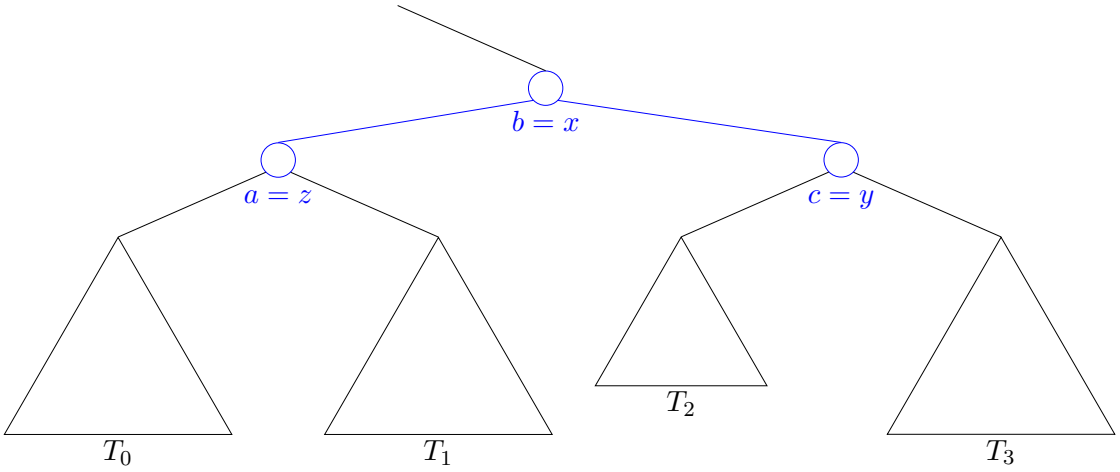


Abbildung 4.23: Höhenbalancierte AVL-Baum-Anordnung zu (5) des Beweises von Satz 4.10.

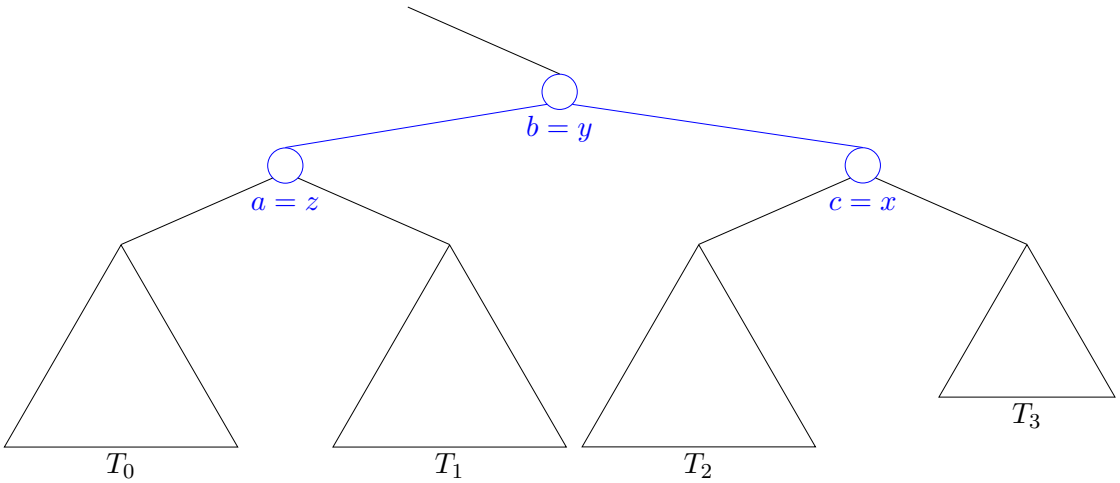


Abbildung 4.24: Höhenbalancierte AVL-Baum-Anordnung zu (6) des Beweises von Satz 4.10.

zu (6) Der Baum ist, wie in Abbildung 4.24 zu sehen, wieder höhenbalanciert.

Alle Schritte erfordern nur konstant viele Rechenoperationen. □

In dem Beispiel aus Abbildung 4.25 reicht es aus nach dem Löschen eines Knotens ein einziges Mal Restructure (s. Algorithmus 4.9) zu verwenden. Allerdings kann im Allgemeinen ein neues Problem auftauchen: Der neue höhenbalancierte Teilbaum von b kann niedriger sein, als der vorher von z. Dadurch wird eventuell der Vater von z unbalanciert! D.h. man verwendet erneut Restructure und zwar so lange, bis die AVL-Eigenschaft (also Höhenbalanciertheit) wiederhergestellt ist. Aufgrund der Höhe von höchstens $\log(n)$ müssen auch höchstens $\log(n)$ Restructure-Operationen gemacht werden, welche jeweils in $O(1)$ laufen.

Satz 4.11. *Mithilfe von Restructure (s. Algorithmus 4.9) kann man einen AVL-Baum auch nach einer Löschen-Operation höhenbalanciert halten. Die Zeit dafür ist* $O(\log(n))$.

4.7 Fibonacci-Zahlen

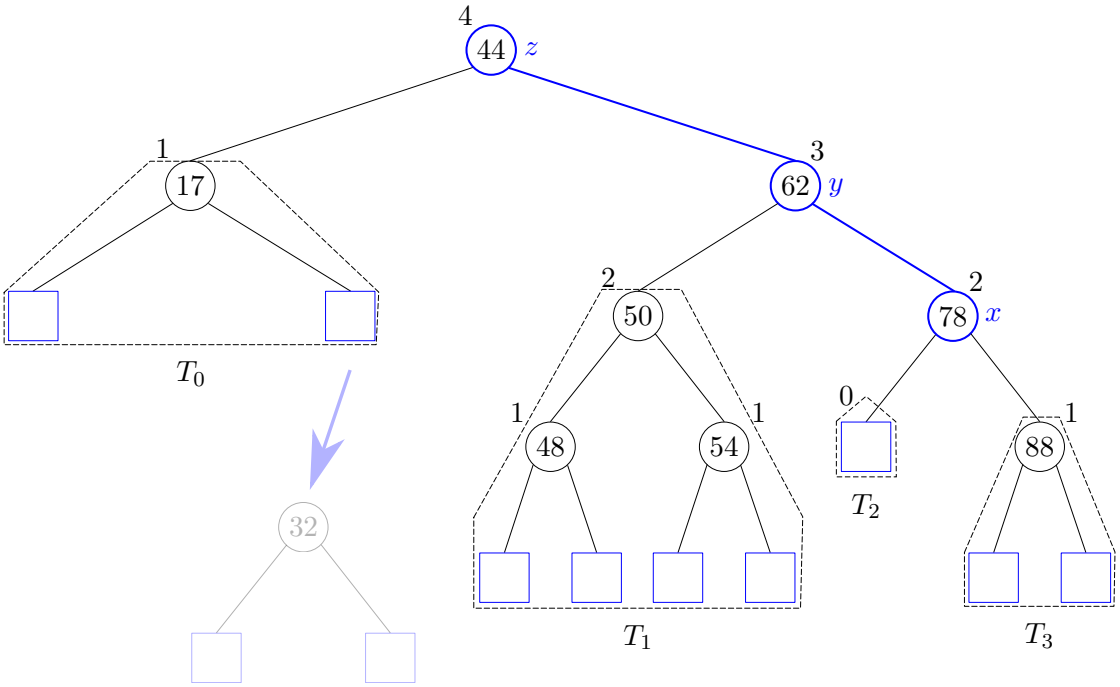
Die Fibonacci-Zahl $f(n)$, benannt nach Leonardo da Pisa (gen. Fibonacci, 1180 - 1241), berechnet sich durch die Summe des Vorgängers und des Vorvorgängers. Also: $f(n) = f(n-1) + f(n-2)$, wobei $f(1) = 1$ und $f(2) = 1$. Man erhält dadurch die Zahlen 1, 1, 2, 3, 5, 8, 13, 21, ...

Die Fibonacci-Zahlen kommen immer wieder in der Natur vor. Hier ein paar Beispiele (Pflanzenarten):

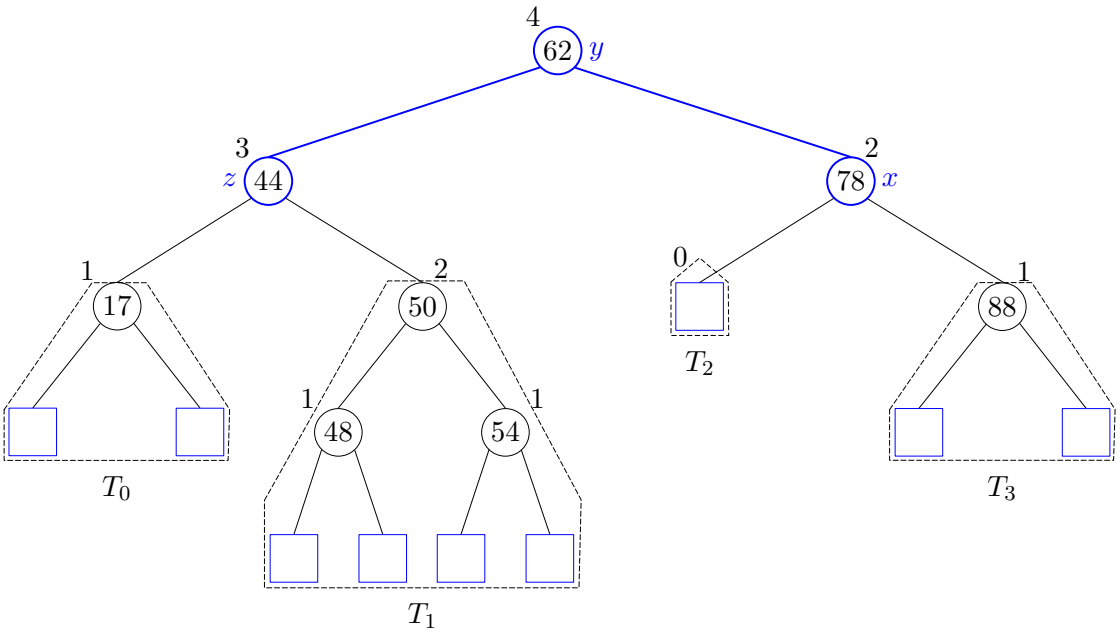
- Calla, besitzt 1 Blatt
- Christusdorn, besitzt 2 Blätter
- Waldlilie, besitzt 3 Blätter
- Akelei, besitzt 5 Blätter
- Blutwurz, besitzt 8 Blätter
- Rudbecki, besitzt 13 Blätter
- Gänseblümchen, besitzt (je nach Art) 21 bzw. 34 Blätter

4.8 Rot-Schwarz-Bäume

Rot-Schwarz-Bäume sind eine andere Methode, Daten strukturiert zu speichern und Daten in logarithmischer Zeit einzufügen, zu löschen und zu suchen. Die Idee ist es die Knoten rot und schwarz zu färben und dabei Invarianten einzuhalten. Ein Vorteil gegenüber AVL-Bäumen ist, dass man nicht die globale Eigenschaft der Höhe an jedem Knoten speichern muss, sondern nur ein Bit; nämlich ob der Knoten rot oder schwarz



(a) Unbalancierter AVL-Baum aus Abbildung 4.16b nach dem Löschen der 32.



(b) Höhenbalancierter AVL-Baum nach der Rotation.

Abbildung 4.25: Das Löschen eines neuen Knotens aus einem AVL-Baum mit einer Rotation zur Wiederherstellung der Höhenbalanciertheit.

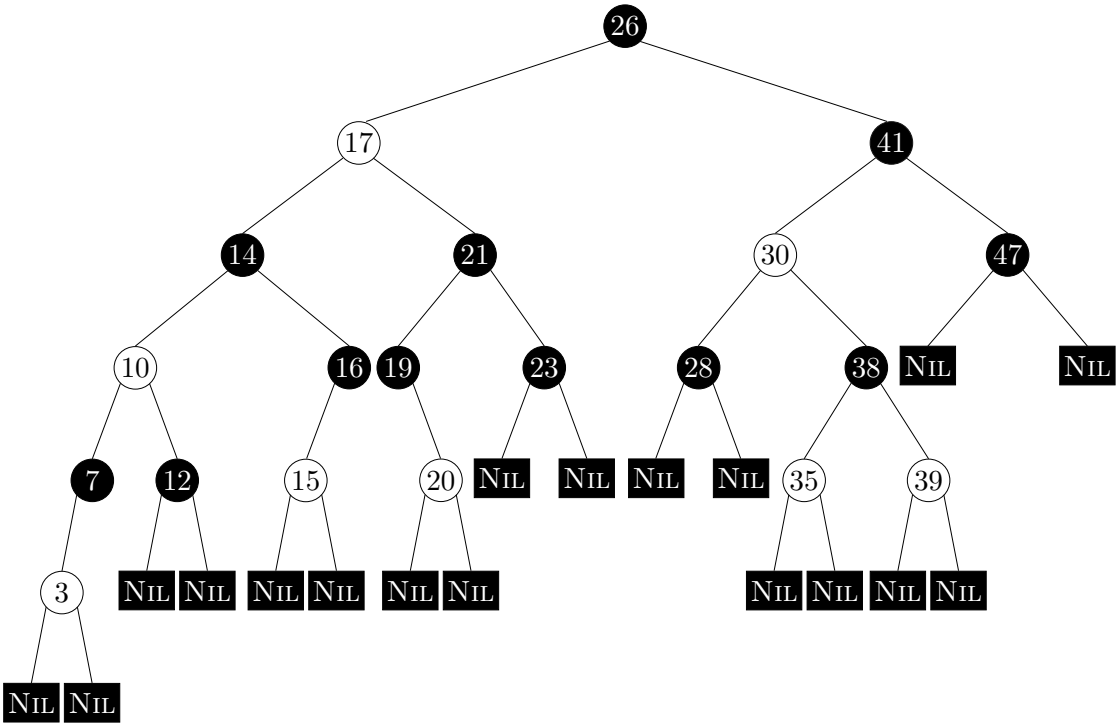


Abbildung 4.26: Ein Beispiel für einen Rot-Schwarz-Baum.

ist. Dadurch lassen sich Rot-Schwarz-Bäume in der Praxis viel leichter implementieren. Allerdings ist ein Nachteil, dass Rot-Schwarz-Bäume gegenüber AVL-Bäumen schlechter balanciert sind.

Definition 4.12 (Rot-Schwarz-Baum)**.** *Ein binärer Suchbaum heißt Rot-Schwarz-Baum, wenn er folgende Eigenschaften erfüllt.*

- *Jeder Knoten ist rot oder schwarz.*
- *Die Wurzel ist schwarz.*
- *Jedes Blatt (Nil) ist schwarz.*
- *Wenn ein Knoten rot ist, sind seine beiden Kinder schwarz.*
- *Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem Blatt des Teilbaumes dieses Knotens enden, die gleiche Anzahl schwarzer Knoten.*

Satz 4.13. *Ein Rot-Schwarz-Baum mit n Knoten hat Höhe $O(\log(n))$.*

Beweis. Sei s die Anzahl an schwarzen Knoten auf jedem Wurzel-Blatt-Pfad. Wenn die Höhe minimal ist, also s, erkennen wir direkt, dass die Höhe logarithmisch ist. Wir betrachten also den Worst-Case für einen Rot-Schwarz-Baum: Wir konstruieren uns einen Rot-Schwarz-Baum, dessen Höhe $2s$ ist und möglichst wenig Knoten besitzt (einen

höheren Baum können wir nicht besitzen, wenn wir s fixieren). Wir zeigen nun, dass jeder Rot-Schwarz-Baum mit Höhe $2s$ mindestens $2^{s+1} - 2$ Knoten besitzt.

Betrachte einen Rot-Schwarz-Baum, der über die folgende rekursive Vorschrift konstruiert wird: Wir erstellen einen schwarzen Knoten als Wurzel und sein linkes Kind ist rot. Die rechten Teilbäume von der Wurzel und dem roten Knoten werden vollständige Rot-Schwarz-Bäume, die keinen roten Knoten besitzen. Das sind $2^{s-1} - 1$ viele Knoten pro Teilbaum. Für den linken Teilbaum wiederholen wir die Prozedur mit $s := s - 1$.

Dieser Rot-Schwarz-Baum der Höhe s mindestens $N(s)$ viele Knoten, wobei

$$N(s) = 2 + 2 \cdot (2^{s-1} - 1) + N(s-1) = \sum_{i=1}^{s} 2 \cdot (2 + 2^{i-1} - 1) = 2^{s+1} - 2.$$

Stellen wir das nach s um erhalten wir $s = \log(N(s)+2) - 1$. Da die Höhe h des Baumes maximal $2s$ sein kann, folgt $h \leq 2s = 2(\log(N(s) + 2) - 1)$, also $h \in O(\log n)$. □

Satz 4.14. *Ein Rot-Schwarz-Baum benötigt $O(\log(n))$ Zeit für dynamische Operationen auf Datenmengen mit n Objekten - d. h. genauso lange wie ein AVL-Baum.*

Um dieses Theorem zu beweisen, spalten wir das Theorem in zwei Teile auf; nämlich in das Einfügen und Löschen in den Rot-Schwarz-Baum.

Lemma 4.15. *Ein Rot-Schwarz-Baum benötigt $O(\log(n))$ Zeit für das Einfügen eines Elements.*

Beweis. Sei x der Knoten, der eingefügt werden soll. Dann bekommt x die Farbe *rot* und wird wie bei binären Suchbäumen üblich bzgl. dem Wert x an die richtige Stelle gesetzt. Sei N der eingefügte Knoten, $P = \text{parent}[x]$, $G = \text{parent}[\text{parent}[x]]$ und U das zweite Kind von G. Es können nun sechs Fälle auftreten:

(1) N ist die Wurzel.

(2) P ist schwarz.

(3) P ist rot und die Wurzel.

(4) P ist rot, G und U sind schwarz und N und P haben nicht die gleiche Kindesrichtung.

(5) P ist rot, G und U sind schwarz und N und P haben nicht die gleiche Kindesrichtung.

(6) P ist rot, G ist schwarz und U ist rot.

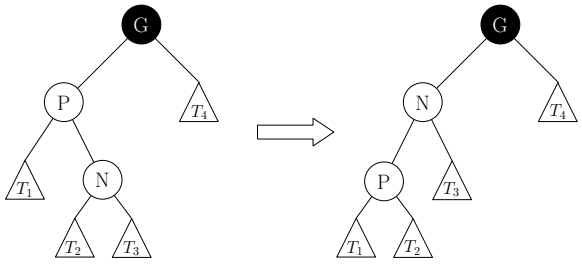
Die Fälle können nun folgendermaßen aufgelöst werden:

(1) Da N die Wurzel ist, können wir N schwarz färben.

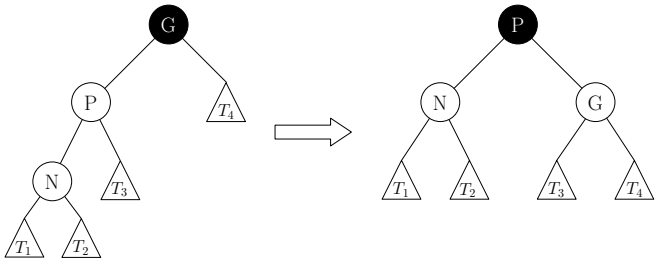
(2) Der Baum erfüllt immer noch alle Eigenschaften eines Rot-Schwarz-Baumes.

(3) Da P die Wurzel ist, können wir P schwarz färben, sodass keine zwei roten Knoten hintereinander folgen. Außerdem erhöht sich die Anzahl der schwarzen Knoten auf allen Pfaden um eins. Somit sind alle Eigenschaften eines Rot-Schwarz-Baumes erfüllt.

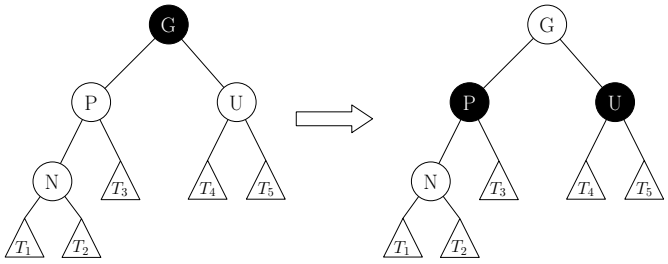
(4) Wir tauschen die Position von P und N, d.h. N wird ein Kind von G und P wird ein Kind von G. Dabei ist zu beachten, dass auch die Teilbäume von N und P umgesetzt werden müssen. Ein Beispiel dieses Falles ist in Abbildung 4.27a gegeben. Da N und P nun die gleiche Kindesrichtung besitzen, wählen wir $N := P$ und fahren mit Punkt 5 fort.



(a) Fall 4



(b) Fall 5



(c) Fall 6

Abbildung 4.27: Verschiedene Fälle, die beim Restrukturieren nach einer Einfügeoperation auftreten. Weiße Knoten zählen als rote, schwarze als schwarze Knoten. Die trivialen Fälle (1)-(3) sind nicht abgebildet. Symmetrische Fälle, z.B. P als rechtes Kind von G, funktionieren analog.

(5) Wir setzen N und G als Kinder von P und setzen dabei auch wieder die Teilbäume um (siehe Abbildung 4.27b). Damit gelten nun wieder alle Rot-Schwarz-Baum-Eigenschaften. Zusätzlich färben wir G rot und P schwarz.

(6) Ist der Onkelknoten U auch rot, so kann einfach G rot und sowohl P als auch U schwarz gefärbt werden (siehe Abbildung 4.27c). Somit folgen in diesem Teilbaum keine zwei roten Knoten aufeinander und die Zahl schwarzer Knoten bleibt auf jedem Pfad gleich. Da G nun rot ist, müssen wir von dort aus nach oben auf weitere Konflikte prüfen. □

Lemma 4.16. *Ein Rot-Schwarz-Baum benötigt $O(\log(n))$ Zeit für das Löschen eines Elements.*

Beweis. Sei x das zu löschende Element. Wenn x zwei Kinder hat, suchen wir, wie bei binären Suchbäumen, den Nachfolger von x und setzen ihn an die Stelle von x. Effektiv müssen wir also nur den Fall betrachten, wenn x nur ein oder kein Kind besitzt.

Betrachten wir zunächst den Fall, dass x ein Kind N besitzt. Dann muss N rot sein, da ansonsten die Anzahl schwarzer Knoten nicht auf jedem Wurzel-Blatt-Pfad gleich ist. Wir setzen N an die Stelle von x und färben den Knoten schwarz.

Besitzt x kein Kind und ist rot, so können wir diesen Knoten einfach löschen.

Nun bleibt der Fall über, dass x keine Kinder hat und die Farbe schwarz hat. Wir ersetzen x durch einen NIL-Zeiger N. Dadurch haben alle Wurzel-Blatt-Pfade durch N einen schwarzen Knoten weniger. Sei nun $P = \text{parent}[N]$, S das zweite Kind von P, C das kleinere und D das größere Kind von S. Es können folgende Fälle auftreten:

(1) N ist die Wurzel.

(2) N, P, S, C und D sind schwarz.

(3) N, P, C und D sind schwarz; S ist rot.

(4) N und S sind schwarz; C ist rot.

(5) N und S sind schwarz; D ist rot.

Die Fälle können nun folgendermaßen aufgelöst werden:

(1) Da alle Pfade durch N führen, besitzen alle Wurzel-Blatt-Pfade gleich viele schwarze Knoten.

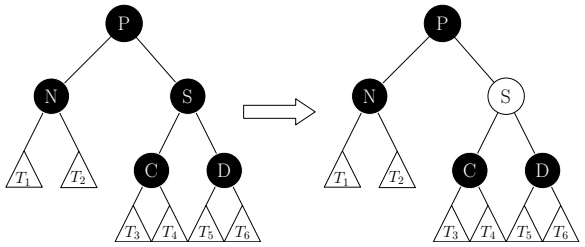
(2) Da C und D schwarz sind, dürfen wir S rot färben (siehe Abbildung 4.28a). Dadurch enthalten alle Pfade durch P einen schwarzen Knoten weniger. Wir führen die Restrukturierung mit P als neues N weiter fort.

(3) Wir rotieren über S und P, d.h. C wird ein Kind von P und S wird Vater von P (siehe Abbildung 4.28b). Wir führen die Restrukturierung weiterhin mit N fort.

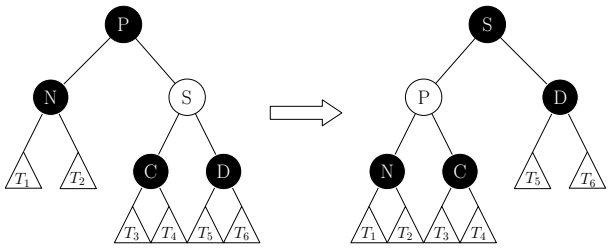
(4) Wie in Fall (2), färben wir P und S um (siehe Abbildung 4.28c). Dadurch erhöht sich die Anzahl an schwarzen Knoten auf Pfaden durch N um eins, sodass alle Wurzel-Blatt-Pfade wieder gleich viele schwarze Knoten enthalten.

(5) Wir rotieren über C und S, d.h. C wird ein Kind von P und S wird ein Kind von C (zusätzlich werden auch die Teilbäume von C und S umgehängt). Außerdem werden C und S umgefärbt (siehe Abbildung 4.28d). Wir führen dann die Restrukturierung durch Fall 6 weiterhin mit N fort.

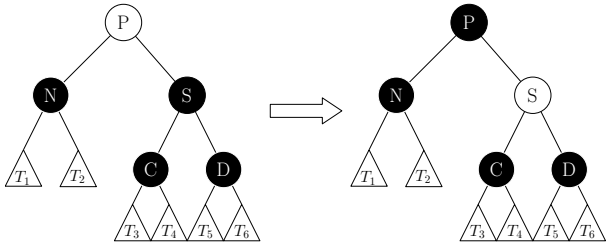
(6) Wir rotieren über P und S, d.h. P wir ein Kind von S und die dazugehörigen Teilbäume werden entsprechend umgehängt. S erhält dabei die Farbe von P und sowohl D als auch P werden schwarz gefärbt (siehe Abbildung 4.28e). Dadurch erhöht sich die Anzahl an schwarzen Knoten auf Pfaden durch N um eins, sodass alle Wurzel-Blatt-Pfade wieder gleich viele schwarze Knoten enthalten. □



(a) Fall 2

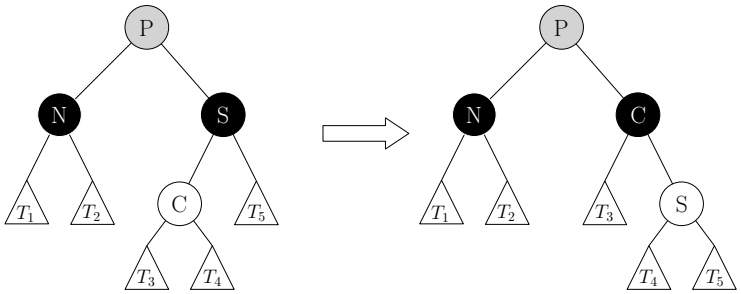


(b) Fall 3

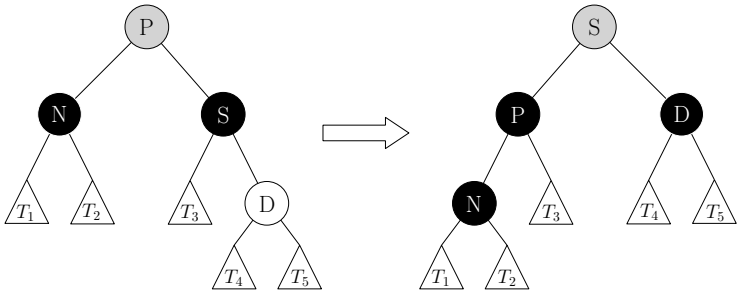


(c) Fall 4

Abbildung 4.28: Verschiedene Fälle, die beim Restrukturieren nach einer Löschoperation auftreten. Weiße Knoten zählen als rote, schwarze als schwarze Knoten. Symmetrische Fälle, z.B. N als ein rechtes Kind von P, funktionieren analog. Ist ein Knoten grau, spielt die Farbe des Knotens keine Rolle für diesen Fall.



(d) Fall 5



(e) Fall 6

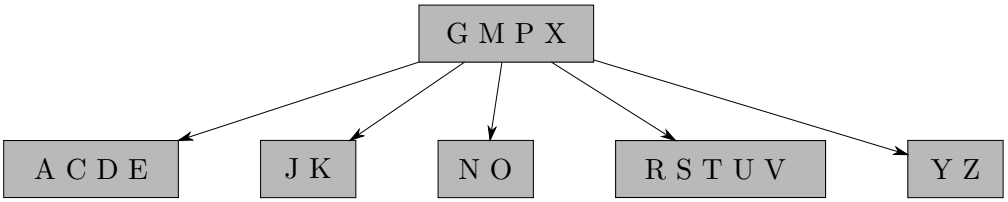
Abbildung 4.28: Fortsetzung von Abbildung 4.28

4.9 B-Bäume

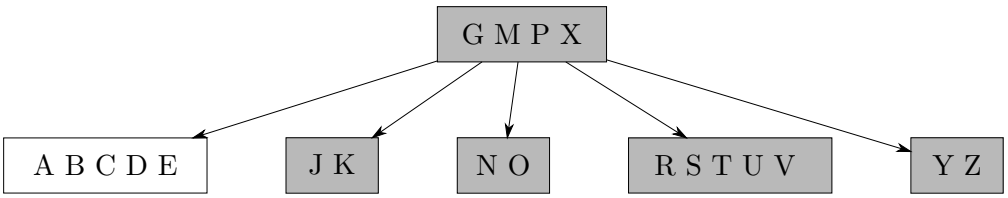
Für den Kontext betrachte man Speicherhierarchien mit unterschiedlich schnellen Zugriffszeiten.

Definition 4.13 (B-Baum)**.** *Ein B-Baum ist ein gerichteter Baum, der die folgenden Eigenschaft hat.*

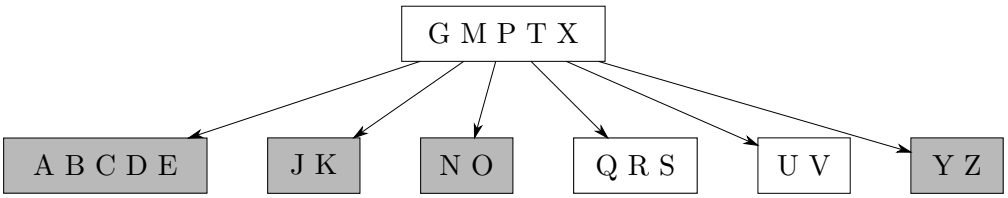
- *Jeder Knoten hat die folgenden Attribute:*
  - *die Anzahl $n[x]$ der in x gespeicherten Schlüssel,*
  - *die sortiert gespeicherten Schlüssel,*
  - *ein boolescher Wert, der anzeigt, ob x ein Blatt ist.*
- *Jeder innere Knoten enthält $(n[x] + 1)$ Zeiger auf seine Kinder.*
- *Die Schlüssel unterteilen die darunter stehenden Teilbäume nach Größe.*
- *Alle Blätter haben gleiche Tiefe.*
- *Jeder Knoten hat Mindest- und Maximalanzahlen von Schlüssel.*
  - *Jeder Knoten, außer der Wurzelknoten, hat mindestens $(t-1)$ Schlüssel. Also hat jeder innere Knoten mindestens t Kinder.*
  - *Jeder Knoten hat höchstens $(2t-1)$ Schlüssel, also höchstens $2t$ Kinder.*



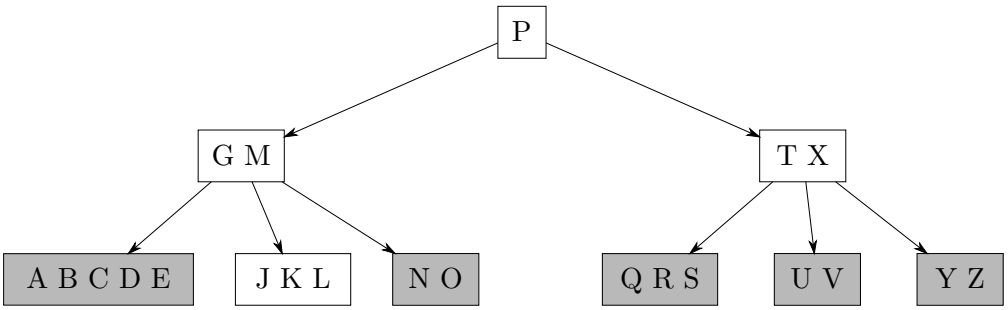
(a) Ursprünglicher Baum



(b) B eingefügt



(c) Q eingefügt



(d) L eingefügt

Abbildung 4.29: Einige Einfügeoperationen mit Aufteilen auf einem B-Baum mit $t = 3$.

Satz 4.14. *Ein B-Baum der Höhe h benötigt zur dynamischen Datenverwaltung im schlimmsten Falle $O(h)$ Plattenoperationen und $O(t \cdot \log_t(n))$ CPU-Zeit.*

Beweis. Im Wesentlichen muss der B-Baum nur einmal durchlaufen werden, wodurch h Knoten besucht werden. An jedem Knoten muss nun der richtige Zweig gesucht werden, was $O(1)$ Plattenoperationen und $O(t)$ CPU-Zeit benötigt. Da jeder Knoten (außer der Wurzel) mindestens t Kinder besitzen, folgt direkt die logarithmische Höhe, also $h \in O(\log_t(n))$. Es bleibt nun zu zeigen, dass wir den B-Baum durch Einfüge- und Löschoperation balanciert halten können.

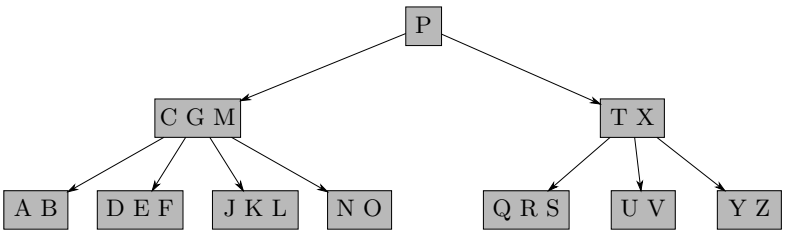
Einfügen: Beim Durchlaufen des B-Baumes, um ein Element k einzufügen, prüfen wir an jedem Knoten x, ob x voll ist, d.h. $2t-1$ Schlüssel enthält. Ist das der Fall, teilen wir den Knoten wie in Abbildung 4.29c und 4.29d gezeigt. Durch das Aufteilen erhalten wir zwei Knoten (drei Knoten, falls die Wurzel geteilt wurde), die beide mindestens $t-1$ Schlüssel besitzen.

Löschen: Analog zum Einfügen wird bei jedem Knoten x_p während des Durchlaufens geprüft, ob der nächste Knoten x ausreichend Schlüssel (mindestens t viele) besitzt. Gibt es nur $t-1$, so folgt eine *Verschiebung* oder eine *Verschmelzung* statt.

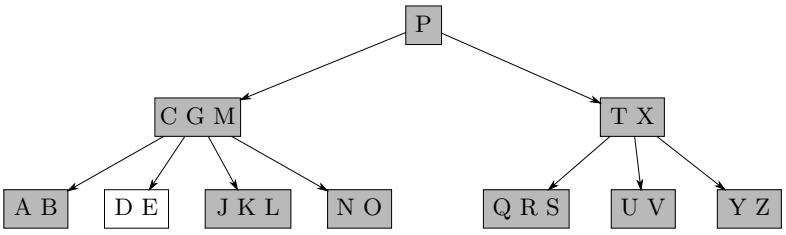
Besitzt der vorangehende oder nachfolgende Geschwisterknoten von x mindestens t Schlüssel, findet eine Verschiebung statt. O.B.d.A. habe der nachfolgende Knoten x_r mindestens t Schlüssel, k_g sei der kleinste Schlüssel und k_p sei der Schlüssel in x_p, der x und x_g trennt. Dann verschieben wir k_p nach x und k_g an die alte Stelle von k_p. Die alte linke Verzweigung von k_g wird nun die neue rechte Verzweigung von k_p.

Haben beide Geschwisterknoten nicht genügend Knoten, so findet eine Verschmelzung statt: Sei x_g ein Geschwisterknoten und k_p der Schlüssel in x_p, der x_g und x trennt. Dann erstellen wir einen Knoten mit den Schlüsseln aus dem Knoten x_g, aus dem Knoten x und k_p. Dadurch wird k_p aus x_p entfernt. Ist x_p der Wurzelknoten, so kann dieser nun leer sein. In diesem Fall können wir den B-Baum schrumpfen, indem wir den Pointer der Wurzel auf das Kind der aktuellen Wurzel setzen (siehe Abbildung 4.30e und 4.30f).

Wenn nun ein Schlüssel k entfernt werden soll, kann es einfach gelöscht werden, falls es sich in einem Blatt befindet. Liegt k allerdings in einem inneren Knoten, so ersetzen wir k mit dem Vorgänger oder Nachfolger, wie bereits bei binären Suchbäumen bekannt. Da der Vorgänger bzw. Nachfolger in einem Blatt liegt, kann das Ersetzen einfach erfolgen. □

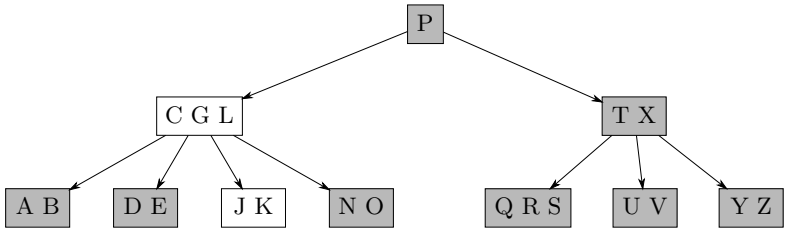


(a) Ursprünglicher Baum

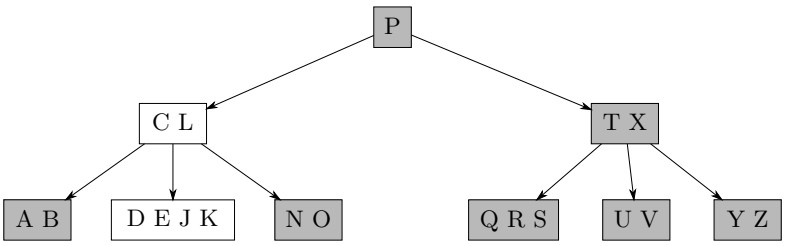


(b) F gelöscht

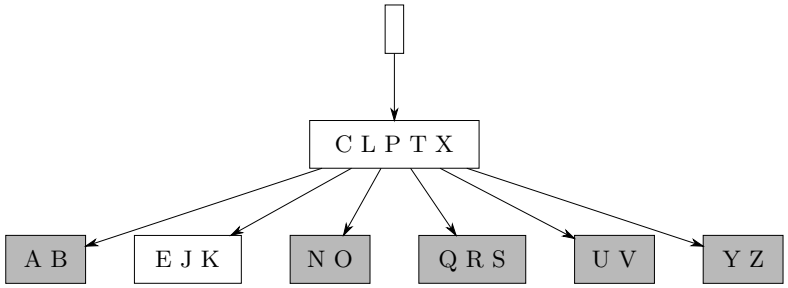
Abbildung 4.30: Einige Löschoperationen auf einem B-Baum mit $t = 3$.



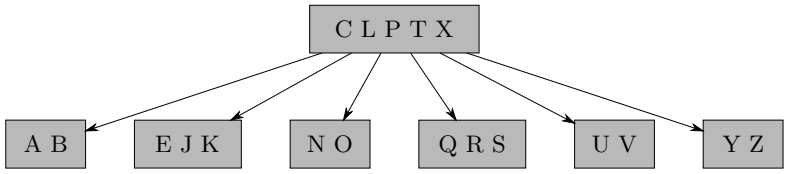
(c) M gelöscht



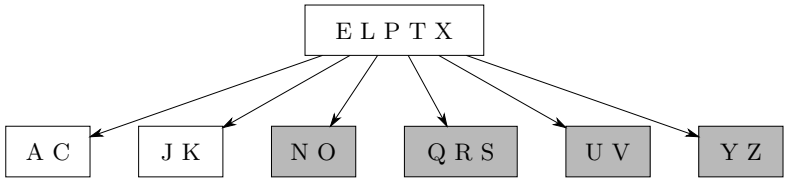
(d) G gelöscht



(e) D gelöscht



(f) Baum schrumpft



(g) B gelöscht

Abbildung 4.30: Einige Löschoperationen auf einem B-Baum mit $t = 3$ (Forts.).

4.10 Heaps

Definition 4.14 (Heap)**.** *Ein gerichteter binärer Baum heißt* binärer Max-Heap*, wenn er folgende Eigenschaften besitzt.*

- *Jeder Knoten hat einen Schlüssel.*
- *Ist h die maximale Distanz von der Wurzel, dann haben alle Ebenen $i < h$ genau zwei Knoten.*
- *Auf Ebene h sind die linken $(n - 2^h + 1)$ Positionen besetzt.*
- *Der Schlüssel jedes Knotens ist mindestens so groß wie die seiner Kinder.*

In einem Min-Heap *sind die Schlüssel höchstens so große wie die von Kinderknoten.*

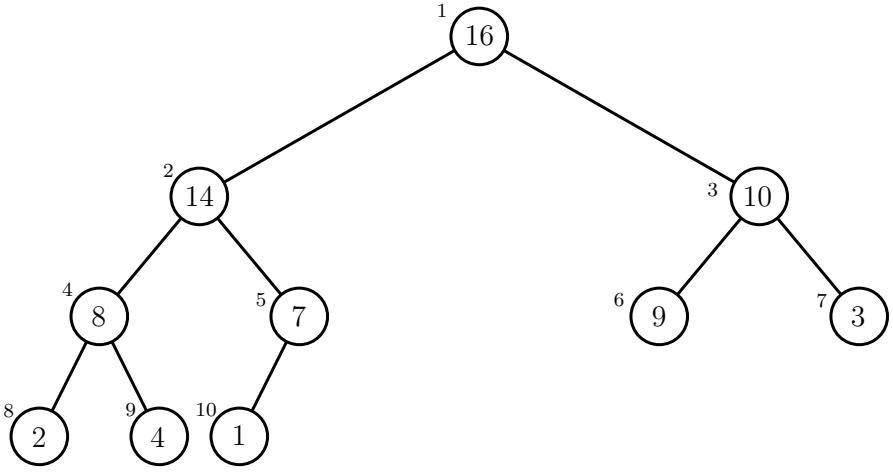


Abbildung 4.31: Ein Beispiel für einen Max-Heap.

Max-Heaps werden in der Regel als Array gespeichert. Über das Array kann nun die Baumstruktur hergeleitet werden: Ein Knoten an Position i im Array besitzt als Kinder die Positionen $2i$ und $2i + 1$. Da nicht jedes Array ein Max-Heap ist, benötigen wir eine Methode, wie wir das Array umstrukturieren können, damit es ein Max-Heap wird. Betrachten wir dazu den Algorithmus Build-Max-Heap (siehe Algorithmus 4.16). Beginnend bei der Hälfte des Arrays, rufen wir auf das i-te Element einmal Max-Heapify auf, dass das Element in der Heap-Struktur rekursiv nach „unten“tauscht, falls eines seiner aktuellen Kinder größer sein sollte. Dass diese Methode tatsächlich einen Max-Heap konstruiert lassen wir an dieser Stelle aus.

Da man zum Bauen eines Heaps $O(\log(n))$ Operationen pro Ebene benötigt, ist klar, dass ein Max-Heap in $O(n \log(n))$ gebaut werden kann. Da aber mehr Knoten auf niedrigeren Ebenen sind, was weniger Arbeit erfordert, kann man diese Abschätzung noch verbessern.

Satz 4.14. *Ein Max-Heap mit n Knoten kann in $O(n)$ gebaut werden.*

```
1: function MAX-HEAPIFY(A, i)
2:     l ← left(i)
3:     r ← right(i)
4:     if l ≤ heap-size[A] und A[l] > A[i] then
5:         maximum ← l
6:     else
7:         maximum ← i
8:     if r ≤ heap-size[A] und A[r] > A[maximum] then
9:         maximum ← r
10:    if maximum ≠ i then
11:        vertausche A[i] ↔ A[maximum]
12:        MAX-HEAPIFY(A, maximum)
```

Algorithmus 4.15: Algorithmus Max-Heapify, der für einen Baum A ab Knoten i (Richtung Wurzel) die Max-Heap-Eigenschaft herstellt.

```
1: function BUILD-MAX-HEAP(A)
2:     heap-size[A] ← länge[A]
3:     for i ← ⌊länge[A]/2⌋ downto 1 do
4:         MAX-HEAPIFY(A, i)
```

Algorithmus 4.16: Algorithmus, der aus einem Baum A einen Max-Heap macht.

Beweis. Sei h die letzte Höhenebene, die 2^{h-1} Knoten besitzt und mindestens einen Knoten enthält, bei dem wir MAX-HEAPIFY anwenden. Man sieht, dass für einen Knoten auf Ebene i maximal $h-i$ Tauschoperationen durchgeführt werden müssen, bis wir den nächsten Knoten betrachten. Für BUILD-MAX-HEAP erhalten wir also folgende Laufzeit:

$$\sum_{i=1}^{h} 2^{i-1} \cdot (h-i) \leq \sum_{i=1}^{h} \frac{2^{h-1}}{2^{h-i}} \cdot (h-i)$$

Da $2^{h-1} \leq n$ gilt, folgt:

$$\sum_{i=0}^{h} \frac{2^{h-1}}{2^{h-i}} \cdot (h-i) \leq n \cdot \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = n \cdot \sum_{i=0}^{h} \frac{i}{2^i} \in \Theta(n) \qquad \square$$

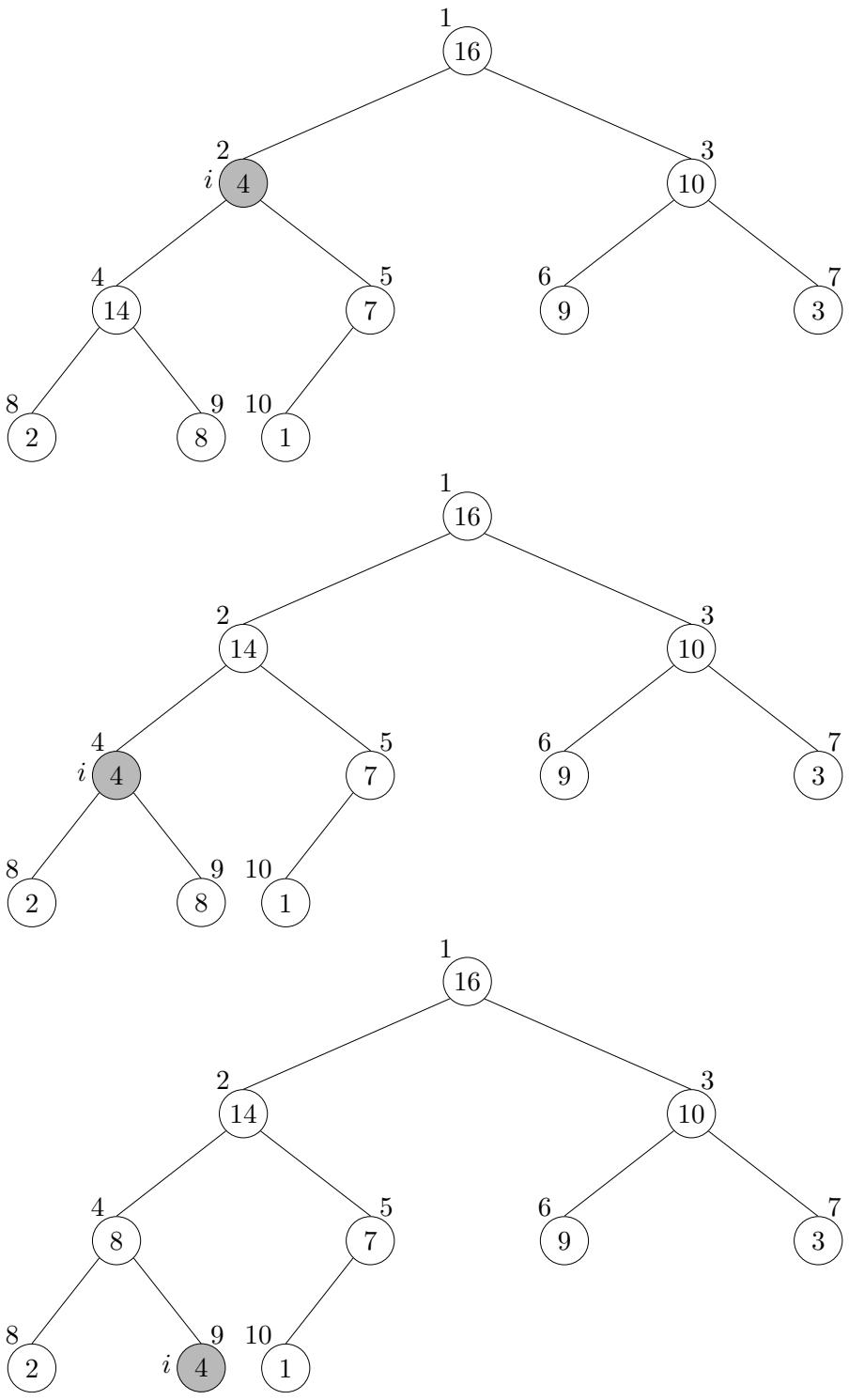


Abbildung 4.32: Veranschaulichung der Operation Heapify.

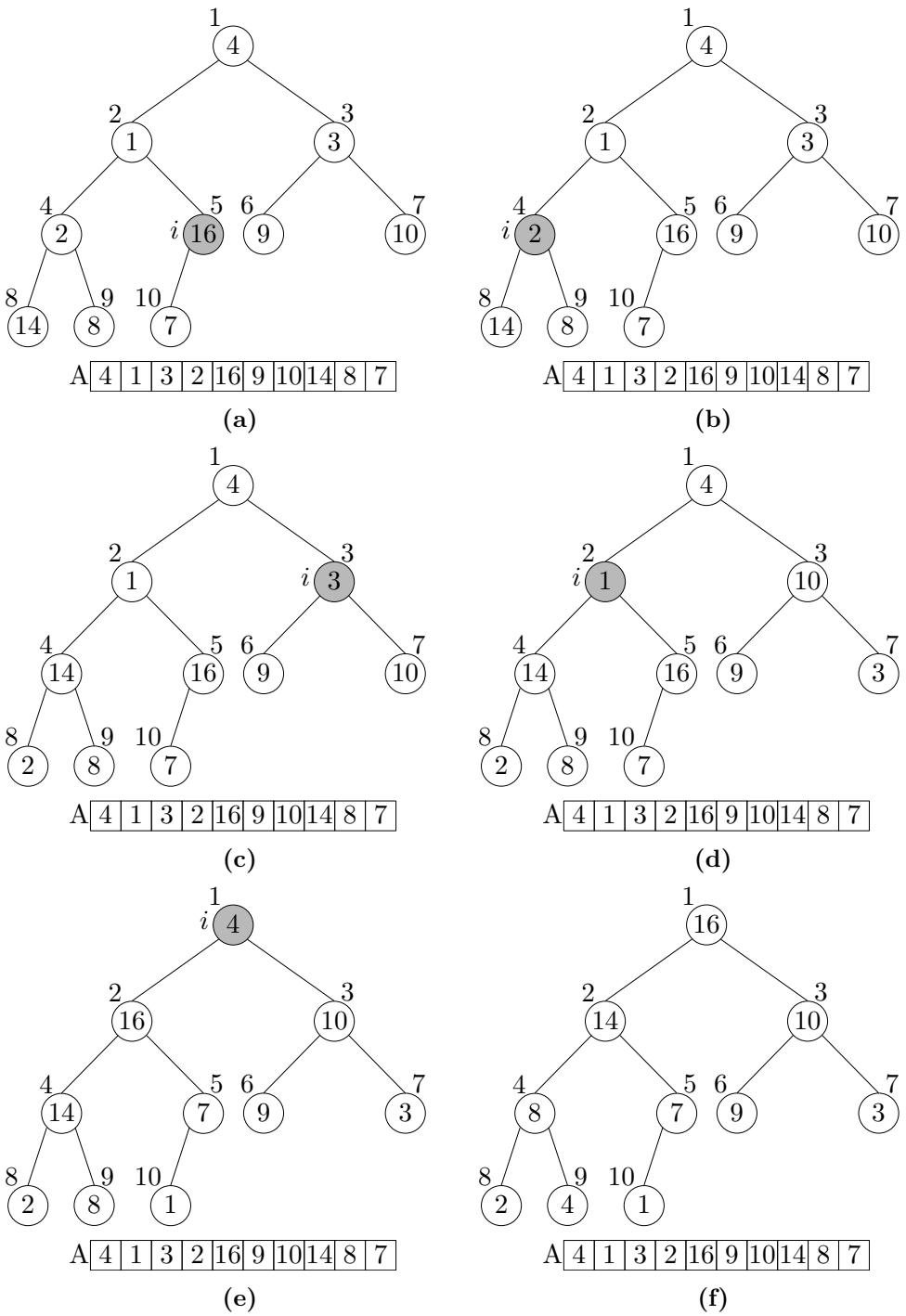


Abbildung 4.33: Funktionsweise des Algorithmus Build-Max-Heap, der aus einem Baum einen Max-Heap macht.

4.11 Andere Strukturen

4.11.1 Fibonacci-Heaps

Die Fibonacci-Heaps sind eine Heap-Struktur mit sehr schneller *amortisierter* Zugriffszeit (s. Tabelle 4.3) und werden als Prioritätenwarteschlange benutzt, d.h. jeder Knoten besitzt eine *Priorität* (auch Schlüssel), sodass Knoten mit hoher (bzw. niedriger) Priorität schnell entnommen werden können. Genauer besteht ein Fibonacci-Heap aus einer Liste von Bäumen mit den Eigenschaften von Min-Heaps. Jeder Knoten im Heap besteht aus einem Schlüssel und einem weiteren Bit, welches markiert, ob für diesen Knoten bereits ein Kind entfernt wurde. Zusätzlich werden, wie für die Liste von Bäumen, für die Kinder jedes Knotens eine Doppelt-Verkettete-Liste verwendet. Der Zeiger für den Startknoten des Fibonacci-Heaps zeigt immer auf die kleinste Wurzel aller Bäume.

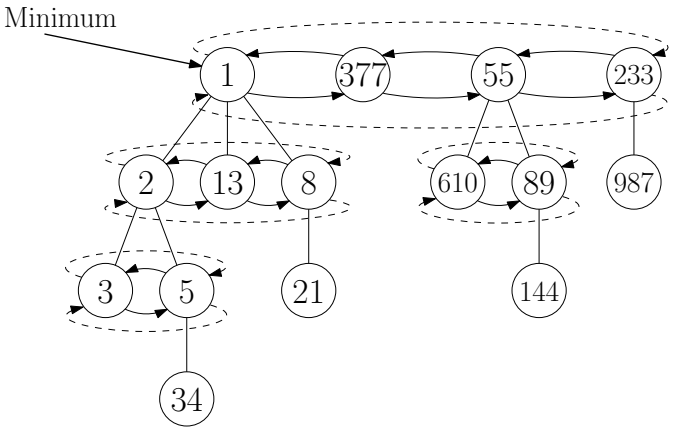


Abbildung 4.34: Ein Beispiel für einen Fibonacci-Heap.

Um ein Element x *einzufügen*, starten wir einen neuen Baum mit x als Wurzel und fügen diesen in den Fibonacci-Heap ein. Dabei muss der Zeiger auf das kleinste Element ggf. aktualisiert werden. Trivialerweise kostet diese Operation nur $O(1)$ Zeit.

Um ein Element x in der Priorität hochzustufen, wird die DECREASEKEY Operation durchgeführt. Hierbei wird der gesamte Knoten mitsamt anhängender Teilbäume aus dem aktuellen Baum entfernt und als neuer Baum mit aktualisiertem Schlüssel in den Fibonacci-Heap eingetragen. War x bereits das zweite Kind vom alten Vaterknoten, das entfernt wurde, so wird auch diese samt Teilbäume als neuer Baum hinzugefügt (dieser Test wird rekursiv auch auf seinen Vater angewendet). Werden Kinder von einer Wurzel entfernt, brauchen wir nichts weiter durchführen.

Möchten wir nun das Minimum aus dem Fibonacci-Heap *entfernen*, fügen wir jedes Kind des Minimums als neuen Baum in den Fibonacci-Heap ein und löschen das Minimum. Bevor wir das neue Minimum suchen, führen wir einen *cleanup* durch: Sei A ein Array mit $1 + 2\log n$ Feldern $A[0], \ldots, A[2\log n]$. Für jeden Baum B im Fibonacci-Heap speichern mit B in Feld $A[d]$, wobei d der Grad der Wurzel von B ist. Gibt es bereits einen Baum B' in diesem Feld, verknüpfen wir diese beiden Bäume, indem der Baum

mit der größeren Wurzel an den Baum mit der kleineren Wurzel gehängt wird. Der neue Baum wird dann in $A[d+1]$ gespeichert und wiederholt auf einen Konflikt geprüft. Den Beweis, dass mit dieser Konfliktbehebung am Ende kein Baum ausgelassen wurde, lassen wir an diese Stelle aus. Man kann außerdem zeigen, dass diese Operationen amortisiert konstante Zeit benötigen.

Soll ein Element x *gelöscht* werden, verringert man den Wert von x mittels DECREASEKEY auf einen Wert kleiner als das Minimum und entfernt dieses dann.

Common Operations	**insert**	**findMin**	**deleteMin**	**delete**	**merge**
Unsorted Linked List	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$
AVL-tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m+n)$
Max-Heap	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(m+n)$
Fibonacci-Heap	$O(1)$	$O(1)$	$O(\log n)$ (*)	$O(\log n)$ (*)	$O(1)$ (*)

Tabelle 4.3: Die Laufzeiten von unterschiedlichen Operationen auf Fibonacci-Heaps frei nach Wikipedia [5]. (*): Amortisierte Laufzeit

4.11.2 Cache-Oblivious B-Trees

Für einen Cache-Oblivious Algrithmus ist die Cachegröße des Systems unbekannt. Dies hat den Vorteil, dass diese Algorithmen auf verschiedenen Systemen die gleiche Leistung bezüglich Eingabemenge und Cachegröße erbringen. Bender, Demaine und Farach-Colton [1] haben den Cache-Oblivious B-Tree entwickelt, eine auf B-Bäumen basierte Datenstruktur, die zum Suchen eines Elements $\Theta(\log_B N)$ und für Einfügen und Löschen amortisiert $\Theta(\log_B N)$ Speicherübertragungen benötigt. Dabei ist B die Größe eines Blocks, welcher zwischen den Speicherhierarchien übertragen wird. Mehr zu Cache-Oblivious B-Trees findet man zum Beispiel in [1].

5 Sortieren

In diesem Kapitel betrachten wir verschiedene Sortierverfahren zum Sortieren von n Zahlen (bzw. vergleichbaren Objekten) und analysieren ihre Laufzeit. Dazu werden Techniken benötigt, um rekursive Gleichungen aufzulösen oder abzuschätzen. Weiter stellt sich die Frage, welche Laufzeitschranken existieren; gibt es Algorithmen die optimal bezüglich der asymptotischen Laufzeit sind?

5.1 Mergesort

Was ist ein Sortierproblem?

Gegeben: n Objekte unterschiedlicher Größe

Gesucht: Eine Sortierung nach Größe

Betrachte beispielsweise die folgende Zahlenreihe, die aufsteigend sortiert werden soll:

$$23, 17, 13, 19, 33, 28, 15$$

Mittels eines naiven Ansatzes - nämlich immer das kleinste Element suchen und an die erste Stelle setzen - liefert einen Algorithmus mit Laufzeit $O(n^2)$. Die sortierte Reihe ist:

$$13, 15, 17, 19, 23, 28, 33$$

Nun stellt sich die Frage: geht das schneller? Ein Prinzip, welches sich hier anbieten, ist „Divide and Conquer“, (*dt.* „Teile und herrsche“, *lat.* divide et impera). Wir wollen also nicht das ganze Problem betrachten, sondern teilen dieses auf in kleine Teilprobleme und lösen diese. Sind die Teilprobleme gelöst, können sie benutzt werden, um die Lösung für das Gesamtproblem zu berechnen. Man spricht auch von divide, conquer, combine (*dt.* teile, herrsche, füge zusammen).

Ein Algorithmus, der dieses Prinzip nutzt, ist *Mergesort*. Dabei wird ein Teilarray rekursiv in der Mitte geteilt, bis ein Array entsteht, welches ein einziges Element enthält.

8		4		9		6		3		11		7		12			
8		4		9		6		3		11		7		12		↓	Divide
8		4		9		6		3		11		7		12			
8		4		9		6		3		11		7		12	←		Conquer
4		8		6		9		3		11		7		12			
4		6		8		9		3		7		11		12		↓	Combine
3		4		6		7		8		9		11		12			

Abbildung 5.1: Beispiel für den parallelen Ablauf von Mergesort

```
1: function MERGESORT(A, p, r)
2:     if p < r then
3:         q ← ⌊(p+r)/2⌋
4:         MERGESORT(A, p, q)
5:         MERGESORT(A, q + 1, r)
6:         MERGE(A, p, q, r)
```

Algorithmus 5.1: Algorithmus Mergesort, der ein Array A zwischen Index p und r (also das Subarray $A[p, ..., r]$)sortiert.

Danach werden die Teilarrays nach und nach zusammengefügt (*Merge-Schritt*), bis das gesamte Array sortiert ist. Die Details sind in Algorithmus 5.1 und 5.2 gegeben. Ein Beispiel mit einer parallelen Ausführung der Divide- und Mergeschritte ist in Abbildung 5.1 gegeben. Eine genaue Durchführung eines Mergeschrittes ist in Abbildung 5.2

5.1.1 Laufzeit von Mergesort

Wie viele Schritte benötigt Mergesort für ein Array der Länge n?

Satz 5.3 (Komplexität von Mergesort)**.** *Für ein n-elementiges Array A hat Mergesort eine Laufzeit von $O(n \log(n))$.*

Beweis. Zunächst runden wir n auf die nächste Zweierpotenz (dabei ändert sich n maximal um einen Faktor 2). Wir erhalten also jeweils zwei Subarrays der Größe $\frac{n}{2}$.

Jetzt betrachten wir $T(n)$, die Laufzeit von Mergesort für einen n-elementigen Array A. Dann haben wir:

Divide: $O(1)$

Conquer: $2 \cdot T(\frac{n}{2})$

Combine: $O(n)$

Damit ergibt sich folgende Rekursionsgleichung:

$$T(n) = \begin{cases} O(1), & n = 1 \\ O(1) + 2 \cdot T(\frac{n}{2}) + O(n) = 2 \cdot T(\frac{n}{2}) + O(n) & \text{für } n \geq 2 \end{cases}$$

Das ist eine „Rekursionsgleichung“. (Mehr dazu später.) Für geeignete Konstanten c, d können wir auch schreiben:

$$T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n, \quad n \geq 2, c \in \mathbb{R}$$
$$T(1) = O(1) = d$$

Nun müssen wir zeigen, dass $T(n) \leq g \cdot n \log n$. Das zeigen wir per Induktion.
Induktionsanfang:

$$n = 2 : T(2) = 2 \cdot T(1) + c \cdot 2 = 2 \cdot d + 2 \cdot c = 2(d + c)$$

```
 1: function MERGE(A, p, q, r)
 2:     n_1 ← q − p + 1
 3:     n_2 ← r − q
 4:     erzeuge die Felder L[1, ..., n_1 + 1] und R[1, ..., n_2 + 1]
 5:     for i ← 1 to n_1 do
 6:         L[i] ← A[p + i − 1]
 7:     for j ← 1 to n_2 do
 8:         R[j] ← A[q + j]
 9:     L[n_1 + 1] ← ∞
10:     R[n_2 + 1] ← ∞
11:     i ← 1
12:     j ← 1
13:     for k ← p to r do
14:         if L[i] ≤ R[j] then
15:             A[k] ← L[i]
16:             i ← i + 1
17:         else
18:             A[k] ← R[j]
19:             j ← j + 1
```

Algorithmus 5.2: Algorithmus Merge, der zwei sortierte Subarrays $A[p, ..., q]$ und $A[q+1, ..., r]$ zu einem sortierten Subarray $A[p, ..., r]$ verschmilzt.

Außerdem ist $g \cdot n \log n = 2 \cdot g \overset{!}{\leq} 2 \cdot (c + d)$. Wenn wir $g \geq (d + c)$ wählen (was $g \geq c$ impliziert), dann ist das erfüllt.

Induktionsannahme:
Die Behauptung gelte für $\frac{n}{2} = 2^{k-1}$, d.h.

$$T(n) \leq g \cdot \frac{n}{2} \cdot \log \frac{n}{2}$$

Induktionsschritt: $k - 1 \to k$, d.h. von $\frac{n}{2} \to n$:

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n}{2}) + c \cdot n \\ &\leq 2 \cdot (g \cdot \frac{n}{2} \cdot \log \frac{n}{2}) + c \cdot n \\ &= g \cdot n \cdot \log n \underbrace{-g \cdot n + c \cdot n}_{\leq 0, \text{ da } g \geq c} \\ &\leq g \cdot n \log \end{aligned}$$

Also ist $T(n) \in O(n \log n)$, wie behauptet. □

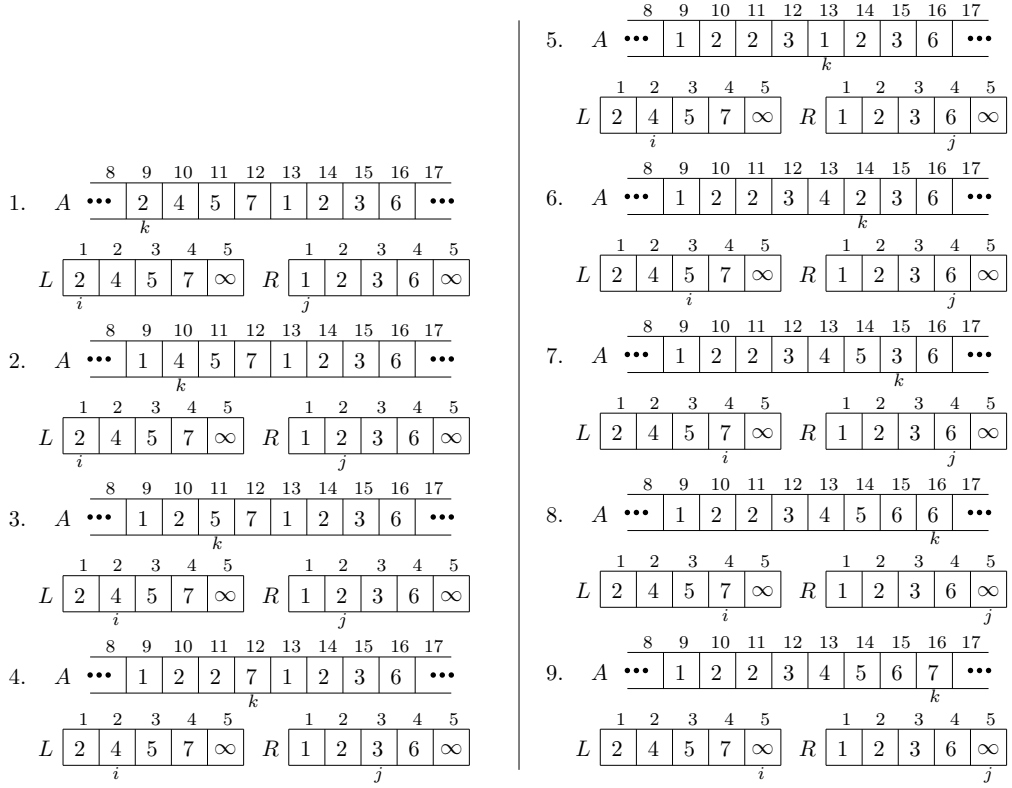


Abbildung 5.2: Beispiel für ein Mergeschritt. Das Teilarray von A wird in Arrays L und R aufgeteilt. In Schritten 2 bis 9 wird das Teilarray von A mit den sortierten Werten überschrieben.

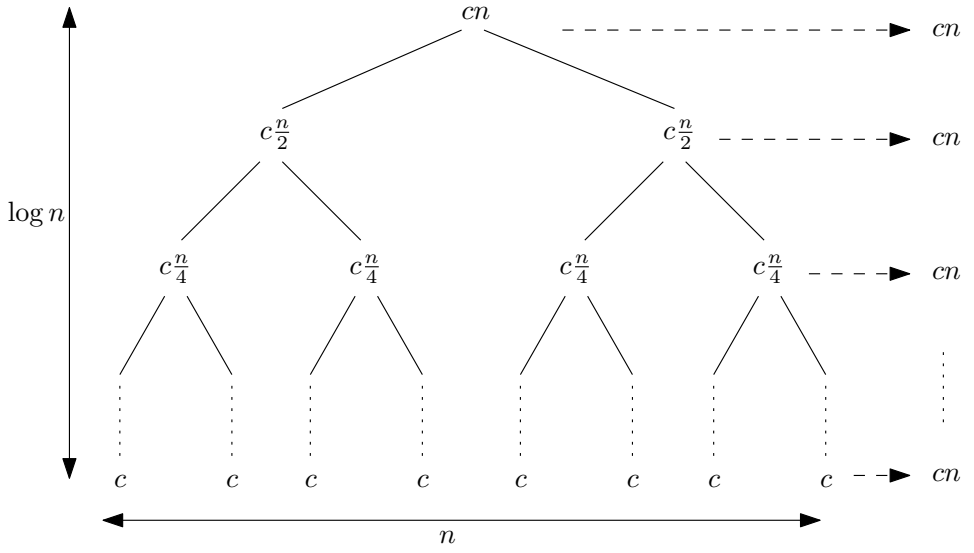


Abbildung 5.3: Veranschaulichung der Laufzeiten aller Subroutinen von Mergesort. Pro Ebene wird insgesamt eine Laufzeit von cn benötigt; bei einer logarithmischen Tiefe ergibt das insgesamt $O(n \log n)$ Schritte.

5.2 Schranken für das Sortieren

Schranken für Probleme teilen sich oft in zwei Teile auf: obere und untere Schranken. Die Schranken, beispielsweise für die Laufzeit oder auch für den Speicherdedarf, werden wie üblich durch asymptotische Abschätzungen in der $\mathcal{O}$-Notation angegeben. Obere Schranken erlangt man durch Angabe eines Algorithmus, der das gegebene Problem korrekt löst. Eine obere Schranke für das Sortierproblem erhalten wir somit durch Mergesort: $\mathcal{O}(n \log n)$. Ein weiteres Beispiel ist die Berechnung von Zusammenhangskomponenten. Eine obere Schranke für die Laufzeit ist $\mathcal{O}(n)$, da wir mit Breiten- und Tiefensuche einen Algorithmus für das Problem besitzen.

Untere Schranken für ein Problem zu finden, kann eine Herausforderung sein, da diese Schranke für jeden Algorithmus gelten muss. Formal lässt sich das folgendermaßen darstellen:

$$\min_{\text{Algorithmus } A} \left(\max_{\text{Eingabe } E} (T_A(E)) \right)$$

dabei ist $T_A(E)$ die Zeit, die Algorithmus A für die Eingabe E benötigt. Der nachfolgenden Tabelle können untere und obere Schranken für bisher kennengelernte Probleme entnommen werden.

	minimale Laufzeit best case	maximale Laufzeit worst case
DFS	$\Omega(n)$	$O(n)$
BFS	$\Omega(n)$	$O(n)$
Insert bin. Suchbaum	$\Omega(1)$	$O(n)$
Insert AVL-Baum	$\Omega(\log n)$	$O(\log n)$
bin. Suche	$\Omega(1)$	$O(\log n)$

5.2.1 Untere Schranke für das Problem Sortieren

Definition 5.4 (Permutation)**.** *Eine Permutation π ist eine Umsortierung von n Objekten, d. h. eine bijektive Abbildung $\pi : \{1, ..., n\} \to \{1, ..., n\}$ mit $i \longmapsto \pi(i)$. Man schreibt π auch in folgender Weise*

$$\begin{pmatrix} 1 & \dots & n \\ \pi(1) & \dots & \pi(n) \end{pmatrix}, \tag{5.1}$$

d. h. als Wertetabelle.

Beobachtung 5.5. *Es gibt genau $n!$ Permutationen von n Objekten.*

Wie lange benötigt man zum Sortieren mithilfe von nicht mehr als paarweisem Vergleichen?

Satz 5.6. *Für n Objekte $x_1, ..., x_n$ benötigt man zum Sortieren mindestens $\Omega(n \log(n))$, wenn man die Objekte nur paarweise vergleichen kann.*

Beweis.

(1) Zunächst hat man $n! = n \cdot (n-1) \cdot ... \cdot 2 \cdot 1$ viele mögliche Permutationen.

(2) Jeder Vergleich teilt die Menge der verbleibenden möglichen Permutationen in zwei Teilmengen.

(3) Im schlechtesten Falle bleibt jeweils die größere Teilmenge übrig.

(4) Man kann also bestenfalls eine Halbierung der verbleibenden Menge von Permutationen erzwingen.

(5) Bis man eine eindeutige Permutation sicher identifiziert hat, braucht man also mindestens $\log_2(n!)$ Vergleiche.

(6)

$$\begin{aligned}
\log_2(n!) &= \sum_{i=1}^{n} \log_2(i) \\
&\geq \sum_{i=\frac{n}{2}}^{n} \log_2(i) \\
&\geq \frac{n}{2} \log_2\left(\frac{n}{2}\right) \\
&= \frac{n}{2} \left(\log_2 n - 1\right) \\
&\in \Omega(n \log(n))
\end{aligned}$$

□

5.2.2 Weitere Sortieralgorithmen

Neben Mergesort gibt es noch zahlreiche andere Sortierverfahren. Wir wollen an dieser Stelle zwei weitere vorstellen.

Die Idee von ***Bubblesort*** ist, dass eine Zahl wie eine Blase aufsteigen will, sie allerdings von großen Blasen zurückgedrängt wird. Wir betrachten also zwei benachbarte Felder und tauschen sie, falls die linke Zahl größer als die rechte ist. Ein detailierter Algorithmus ist in Pseudocode 5.1 gegeben. Man kann zeigen, dass Bubblesort ein korrektes und stabiles Sortierverfahren mit Laufzeit $O(n^2)$ ist. Der Beweis wird zu Übungszwecken ausgelassen.

Definition 5.7. *Ein Sortierverfahren heißt* stabil*, wenn gleiche Werte nach dem Algorithmusdurchlauf in der gleichen Reihenfolge sind wie vorher.*

Betrachten wir folgendes Beispiel:

[7]	[1]	8	2	3	5
[1]	[7]	8	2	3	5
1	[7]	[8]	2	3	5
1	7	[8]	[2]	3	5
1	7	[2]	[8]	3	5
1	7	2	[8]	[3]	5
1	7	2	[3]	[8]	5
1	7	2	3	[8]	[5]
1	7	2	3	[5]	[8]
[1]	[7]	2	3	5	8
1	[7]	[2]	3	5	8
1	[2]	[7]	3	5	8
1	2	[7]	[3]	5	8
1	2	[3]	[7]	5	8
1	2	3	[7]	[5]	8
1	2	3	[5]	[7]	8

Danach gibt es keine Tauschoperationen mehr.

Eingabe: Array $A[1] \dots A[2]$

Ausgabe: Sortiertes Array

```
1: for i = 0 to n − 1 do
2:     for j = 1 to n − i − 1 do
3:         if A[j] > A[j + 1] then
4:             Vertausche A[j] und A[j + 1]1
```

Pseudocode 5.1: Ein formaler Bubblesort-Algorithmus

Ein weiterer Sortieralgorithmus ist ***Selectionsort***. Die Idee ist bei einer Sequenz von Zahlen in einem Array A das Minimum zu suchen und an die linkeste Stelle zu tauschen. Das wiederholt man auf dem Bereich ohne dem Minimum bis das Array sortiert ist. Ein Beispiel:

[7]	[1]	8	2	3	5
1	[7]	8	[2]	3	5
1	2	[8]	7	[3]	5
1	2	3	[7]	8	[5]
1	2	3	5	[8]	[7]
1	2	3	5	7	[8]
1	2	3	5	7	8

5.3 Behandeln von Rekursionen

Welche Möglichkeiten gibt es, Rekursionen zu lösen?

5.3.1 Substitutionsmethode

Wie bereits gesehen, können wir eine Lösung raten und die Korrektheit per Vollständiger Induktion beweisen. Das haben wir im Beweis von Satz 5.3 angewendet. Eine Schwierigkeit bei diesem Ansatz ist es gute Lösungen zu finden. Da man in der Regel an einer möglichst genauen Lösung interessiert ist, kann das schwer bis unmöglich sein. Oft kann man aber Abschätzungen gewinnen, z.B.

$$T(n) \in \Omega(n)$$
$$T(n) \in O(n^2)$$

5.3.2 Erzeugende Funktionen

Man kann Rekursionen auch oft lösen, indem man sie in einen abstrakt-formalen Kontext einbettet und die dafür bekannten Rechenregeln anwendet. Dafür betrachten wir

Definition 5.8. *(Erzeugende Funktion)*
Sei $(a_n)_{n\in\mathbb{N}}$ eine Zahlenfolge mit $a_n \in \mathbb{R}$. Dann heißt

$$f(x) = \sum_{n=0}^{\infty} a_n \cdot x^n$$

die (gewöhnliche) erzeugende Funktion von $(a_n)_{n\in\mathbb{N}}$

Beispiel:
$a_n = 2a_{n-1}$, $n = 0, 1, 2, \ldots$
$a_0 = 1$
$a_1 = 2$, $a_2 = 4$, $a_3 = 8$, $\ldots$
$f(x) = 1x^0 + 2x^1 + 4x^2 + 8x^3 + \ldots$

Wir suchen eine „geschlossene Form“ für a_n, also einen expliziten Ausdruck, um a_n direkt aus n (ohne Rekursion) ausrechnen zu können.

Wir können aufgrund der Rekursion $f(x)$ umformen:

$$\begin{aligned} f(x) &= \sum_{n=0}^{\infty} a_n x^n \\ &= a_0 x^0 + \sum_{n=1}^{\infty} a_n x^n = a_0 + \sum_{n=1}^{\infty} 2 \cdot a_{n-1} x^n \\ &= a_0 + 2x \cdot \sum_{n=0}^{\infty} a_n x^n = a_0 + 2x \cdot f(x) \end{aligned}$$

Auflösen nach $f(x)$:

$$\begin{aligned} & f(x) = 1 + 2xcdotf(x) \\ \Leftrightarrow & f(x) \cdot (1 - 2x) = 1, \quad \text{d.h. } f(x) = \frac{1}{1-2x} \end{aligned}$$

Das kann man auch als Reihe schreiben, indem man

$$\sum_{n=0}^{\infty} p^n = \frac{1}{1-p}$$

nutzt:

$$\sum_{n=0}^{\infty} a_n x^n = f(x) = \frac{1}{1-2x} = \sum_{n=0}^{\infty} (2x)^n = \sum_{n=0}^{\infty} 2^n x^n$$

Damit Gleichheit gilt, müssen die Koeffizienten gleich sein

$$\begin{aligned} a_0 &= 2^0 \\ a_1 &= 2^1 \\ &\vdots \\ a_n &= 2^n \end{aligned}$$

Das ist eine geschlossene Form! Das geht natürlich auch für kompliziertere Fälle, die schwerer zu lösen sind. Oft setzt man dafür auch ein breiteres Arsenal an Umformungen ein, Tabellen von Funktionsentwicklungen, etc.

Betrachte nun ein weiteres Beispiel für Rekursionsgleichungen: die Fibonacci-Zahlen.

$$F_0 = 1 \tag{5.2}$$
$$F_1 = 1 \tag{5.3}$$
$$F_n = F_{n-1} + F_{n-2} \qquad \text{für } n \geq 2 \tag{5.4}$$

Dafür wird eine geschlossene Form gesucht! Gewöhnliche erzeugende Funktion:

$$F(x) = \sum_{n=0}^{\infty} F_n x^n \tag{5.5}$$

$$= F_0 x^0 + F_1 x^1 + \sum_{n=2}^{\infty} F_n x^n \tag{5.6}$$

$$= 0 + x + \sum_{n=2}^{\infty} (F_{n-1} + F_{n-2}) x^n \tag{5.7}$$

$$= x + \sum_{n=2}^{\infty} F_{n-1} x^n + \sum_{n=2}^{\infty} F_{n-2} x^n \tag{5.8}$$

$$= x + x \sum_{n=1}^{\infty} F_n x^n + x^2 \sum_{n=0}^{\infty} F_{n-2} x^n \tag{5.9}$$

$$= x + (x + x^2) \sum_{n=0}^{\infty} F_n x^n \tag{5.10}$$

$$= x + (x + x^2) \cdot F(x) \tag{5.11}$$

$$\Leftrightarrow F(x) = \frac{x}{1 - x - x^2} \tag{5.12}$$

Das kann man weiter zerlegen (Partialbruchzerlegung):

$$\frac{x}{1 - x - x^2} \stackrel{!}{=} \frac{A}{1 - ax} - \frac{B}{1 - bx} = \frac{A(1 - bx) - B(1 - ax)}{(1 - ax)(1 - bx)} \tag{5.13}$$

Daraus rechnet man für den Nenner:

$$(1 - ax)(1 - bx) \stackrel{!}{=} 1 - x - x^2 \tag{5.14}$$

$$1 - (a + b)x + abx^2 \stackrel{!}{=} 1 - x - x^2 \tag{5.15}$$

Also gilt

$$a + b = 1 \qquad \text{und} \tag{5.16}$$

$$ab = -1. \tag{5.17}$$

Somit folgt

$$b = 1 - a \qquad \text{und} \tag{5.18}$$

$$a(a - 1) = -1 \tag{5.19}$$

und somit erhält man

$$a = \frac{1 + \sqrt{5}}{2},\ b = \frac{1 - \sqrt{5}}{2}. \tag{5.20}$$

Außerdem gilt

$$A - B + (-Ab + Ba)x = x, \tag{5.21}$$

d. h. für den Zähler

$$A = B \text{ und } A = \frac{1}{\sqrt{5}} = B. \tag{5.22}$$

Damit hat man

$$F(x) = \frac{1}{\sqrt{5}(1-ax)} - \frac{1}{\sqrt{5}(1-bx)} \text{ mit } a = \frac{1+\sqrt{5}}{2},\ b = \frac{1-\sqrt{5}}{2}. \tag{5.23}$$

Wieder mit geometrischen Reihen ist

$$\frac{1}{1-ax} = \sum_{n=0}^{\infty} a^n x^n \quad \text{und} \quad \frac{1}{1-bx} = \sum_{n=0}^{\infty} b^n x^n, \tag{5.24}$$

also

$$F_n = \frac{1}{\sqrt{5}}(a^n - b^n) \tag{5.25}$$

Mit den gebräuchlichen Abkürzungen

$$a = \frac{1+\sqrt{5}}{2} =: \varphi \qquad (\text{„Goldener Schnitt“}) \tag{5.26}$$

$$\text{und } b = \frac{1-\sqrt{5}}{2} =: \overline{\varphi} \tag{5.27}$$

ergibt sich also

$$F_n = \frac{1}{\sqrt{5}}(\varphi^n - \overline{\varphi}^n) \tag{5.28}$$

bzw. ganz explizit

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)\right). \tag{5.29}$$

Erstaunlicherweise liefert das immer eine ganze Zahl. Außerdem sieht man, dass die n-te Fibonacci-Zahl der n-ten Potenz des Goldenen Schnittes entspricht - mit dem Korrekturterm $\overline{\varphi}^n$ und um $\frac{1}{\sqrt{5}}$ normiert. Da $|\overline{\varphi}| < 1$, ist $\left|\frac{\overline{\varphi}^n}{\sqrt{5}}\right| < \frac{1}{2}$, d. h. F_n ist die ganze Zahl, die am nächstens an $\frac{1}{\sqrt{5}}\varphi^n$ ist. Das erklärt auch, warum das Verhältnis aufeinanderfolgender Fibonacci-Zahlen gegen φ konvergiert.

5.3.3 Mastertheorem

Man betrachte allgemein:

$$T(n) \quad = \quad \underbrace{\sum_{i=1}^{m} T(\alpha_i \cdot n)}_{\uparrow} \quad + \quad \underbrace{\Theta(n^k)}_{\uparrow}$$

Aufteilung in Teilprobleme — Kosten der Aufteilung + Zusammenfügen

Dafür ist eine allgemeine Formel gesucht!

Satz 5.9 (Mastertheorem)**.** *Sei* $T : \mathbb{N} \to \mathbb{R}$ *mit*

$$T(n) = \sum_{i=1}^{m} T(\alpha_i \cdot n) + \Theta(n^k), \tag{5.30}$$

wobei $\alpha_i \in \mathbb{R}$ *mit* $0 < \alpha_i < 1$, $m \in \mathbb{N}$ *und* $k \in \mathbb{R}$*. Dann gilt*

$$T(n) \in \begin{cases} \Theta(n^k) & \text{für } \sum_{i=1}^{m} \alpha_i^k < 1 \\ \Theta(n^k \log(n)) & \text{für } \sum_{i=1}^{m} \alpha_i^k = 1 \\ \Theta(n^c) & \text{mit } \sum_{i=1}^{m} \alpha_i^c = 1 \text{ für } \sum_{i=1}^{m} \alpha_i^k > 1 \end{cases} \tag{5.31}$$

Beweis. Nicht hier (s. z. B. Kapitel 4.4 in [2])! □

Beispiel 5.10.

(1) $U(n) = 8U\left(\frac{n}{3}\right) + n^2$
(Ausprobieren mit $U(1) = 1$*:*

$$\begin{aligned} U(3) &= 8 + 9 &&= 17 && (5.32)\\ U(9) &= 8 \cdot 17 + 81 &&= 217 && (5.33)\\ U(27) &= 8 \cdot 217 + 27^2 &&= 2465 && (5.34)\\ U(81) &= 8 \cdot 2465 + 81^2 &&= 26281 && (5.35)\\ U(243) &= 8 \cdot 26281 + 243^2 &&= 269297 && (5.36)\\ &\vdots && \quad) && (5.37) \end{aligned}$$

Im Mastertheorem:
Es sind $\alpha_1 = ... = \alpha_8 = \frac{1}{3}$, $m = 8$ *und* $k = 2$*. Dann gilt*

$$\sum_{i=1}^{8} \left(\frac{1}{3}\right)^2 < 1, \tag{5.38}$$

also erster Fall: $U(n) \in \Theta(n^2)$*!*

(2) $V(n) = 9V\left(\frac{n}{3}\right) + n^2$
(Ausprobieren mit $V(1) = 1$*:*

$$\begin{aligned}
V(1) &= 1 &&= 1 \cdot 3^0 &&= 1 \cdot 1^2 && (5.39)\\
V(3) &= 9 \cdot 1 + 3^2 &&= 2 \cdot 3^2 &&= 2 \cdot 3^2 && (5.40)\\
V(9) &= 9 \cdot (2 \cdot 3^2) + 9^2 &&= 3 \cdot 3^4 &&= 3 \cdot 9^2 && (5.41)\\
V(27) &= 9 \cdot (3 \cdot 9^2) + 27^2 &&= 4 \cdot 3^6 &&= 4 \cdot 27^2 && (5.42)\\
V(81) &= 9 \cdot (4 \cdot 27^2) + 81^2 &&= 5 \cdot 3^8 &&= 5 \cdot 81^2 && (5.43)\\
&\vdots && && &&)\ (5.44)
\end{aligned}$$

Im Mastertheorem:
Es sind $\alpha_1 = ... = \alpha_9 = \frac{1}{3}$*,* $m = 9$ *und* $k = 2$*. Dann gilt*

$$\sum_{i=1}^{9} \left(\frac{1}{3}\right)^2 = 1, \tag{5.45}$$

also zweiter Fall: $V(n) \in \Theta(n^2 \log(n))$*!*

(3) $W(n) = 10W\left(\frac{n}{3}\right) + n^2$
(Ausprobieren mit $V(1) = 1$*:*

$$\begin{aligned}
W(1) &= 1 && && && (5.46)\\
W(3) &= 10 \cdot 1 + 3^2 &&= 19 && && (5.47)\\
W(9) &= 10 \cdot 19 + 9^2 &&= 271 && && (5.48)\\
W(27) &= 10 \cdot 271 + 27^2 &&= 3439 && && (5.49)\\
W(81) &= 10 \cdot 3439 + 81^2 &&= 40951 && && (5.50)\\
W(243) &= 10 \cdot 40951 + 243^2 &&= 468559 && && (5.51)\\
&\vdots && && &&)\ (5.52)
\end{aligned}$$

Im Mastertheorem:
Es sind $\alpha_1 = ... = \alpha_{10} = \frac{1}{3}$*,* $m = 10$ *und* $k = 2$*. Dann gilt*

$$\sum_{i=1}^{10} \left(\frac{1}{3}\right)^2 > 1, \tag{5.53}$$

also dritter Fall: gesucht wird c *mit* $\sum_{i=1}^{10} \alpha_i^c = 1$*, d. h.*

$$\begin{aligned}
& 10 \cdot \left(\frac{1}{3}\right)^c &&= 1 && && (5.54)\\
\Leftrightarrow\ & \left(\frac{1}{3}\right)^c &&= \frac{1}{10} && && (5.55)\\
\Leftrightarrow\ & c &&= \log_3(10) &&\approx 2,096. && (5.56)
\end{aligned}$$

Das liefert $W(n) \in \Theta(n^{2,096})$*!*

5.4 Nichtlineare Rekursion

Die Welt ist nicht linear! Nichtlinearität birgt viele Überraschungen.

5.4.1 Logistische Rekursion

Betrachte eine Funktion, die proportional zu einer Größe wächst:

$$x_{n+1} = (1+q)x_n$$

Dies liefert

$$x_n = (1+q)^n x_0,$$

also ein exponentielles Wachstum. Betrachtet man das Bevölkerungswachstum, merkt man, dass dies nicht beliebig lange weiter geht. Hier gibt es unter anderem:

Zuwachs durch Fruchbarkeit

$$x_{n+1} = q_f x_n$$

Schwund durch Verhungerung (u.Ä.)

$$x_{n+1} = q_v(G - x_n)$$

Das ergibt zusammen

$$x_{n+1} = q_f q_v(G - x_n),$$

bzw. normiert ergibt dies

$$x_{n+1} = r x_n(1 - x_n)$$

Bildlich liefert das Abbildung 5.4. Weitere Informationen und Bilder gibt es unter [3] zu finden.

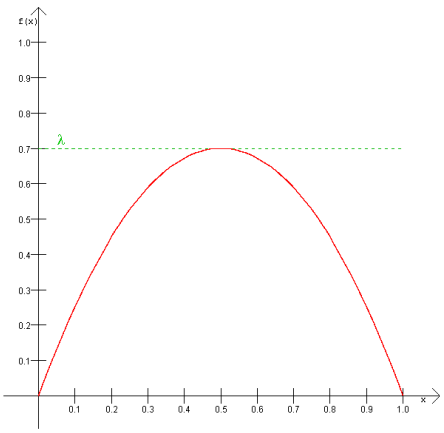


Abbildung 5.4: Logistische Parabel (siehe auch Kapitel 2 in [3])

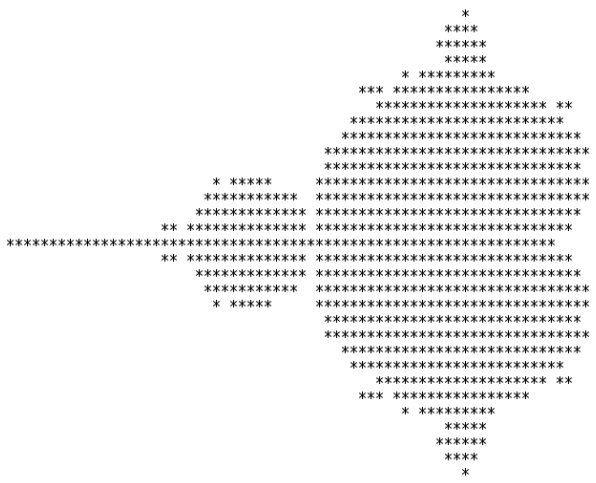


Abbildung 5.5: Mandelbrot Menge

5.4.2 Mandelbrotmenge

Wie sieht es aus mit quadratischen Rekursionen:

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

Für welche Werte c bleibt das beschränkt?

$$c = -2 : 0, -2, 2, 2, \ldots$$

$$c = 1/4 : 0, 0.25, 0.3125, 0.3476, 0.3708, \ldots$$

Das kann man auf logistische Iteration abbilden.

Aber: Für welche *komplexen* Werte c bleibt das beschränkt? Betrachte dafür Abbildungen 5.5 und 5.6 [7]. Es kann außerdem ein Zusammenhang zwischen einer Mandelbrotmenge und einer logistischen Iteration beobachtet werden (siehe Abbildung 5.7).

5.4.3 Fraktale

Betrachten wir zunächst die Koch-Kurve (siehe Abbildung 5.8), benannt nach Niels Fabian Helge Hartmut von Koch (1870 - 1924). Sie wird unter anderem auch Schneeflockenkurve genannt. Pro Iteration wächst die Länge um einen Faktor von 4/3. Die Fläche wird niemals den Kreis um das Dreieck überschreiten. Die Hausdorff-Dimension der Randes beträgt $\log(4)/\log(3) = 1.2618595\ldots$, wobei die Hausdorff-Dimension mit folgender Frage beschrieben werden kann: *Wie wächst das Gesamtmaß in Abhängigkeit von der Größe?*

Betrachten wir das Sierpinski-Dreieck (siehe Abbildung 5.9), können wir folgendes festhalten: Die Fläche schrumpft pro Iteration um den Faktor 3/4 und die Hausdorff-Dimension der Fläche beträgt $\log(3)/\log(2) = 1.5849625$.

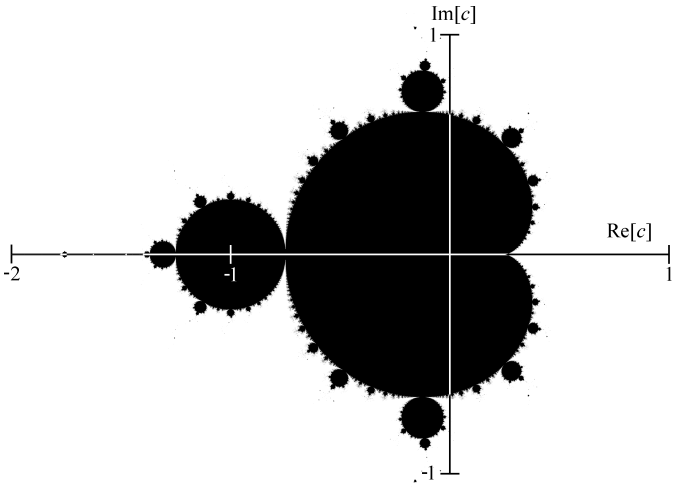


Abbildung 5.6: Besser aufgelöste Mandelbrot Menge

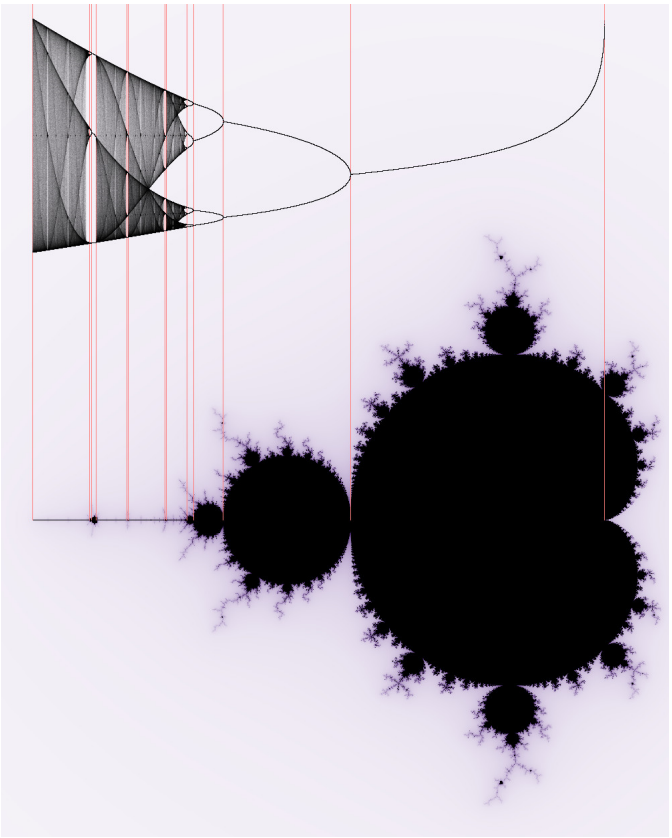


Abbildung 5.7: Abbildung des Realteils auf die logistische Iteration

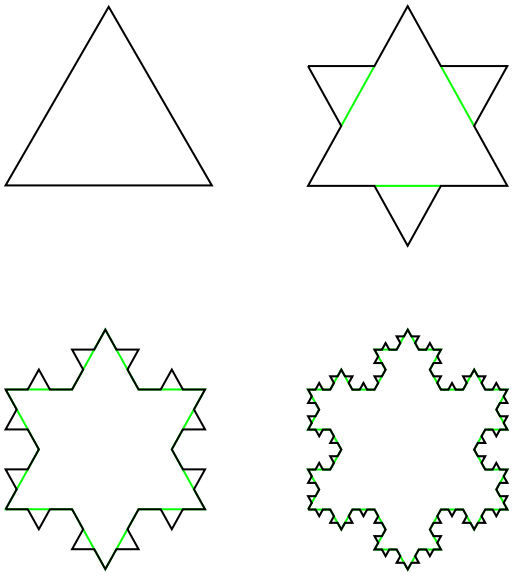


Abbildung 5.8: Die ersten vier Iterationen der Koch-Kurve

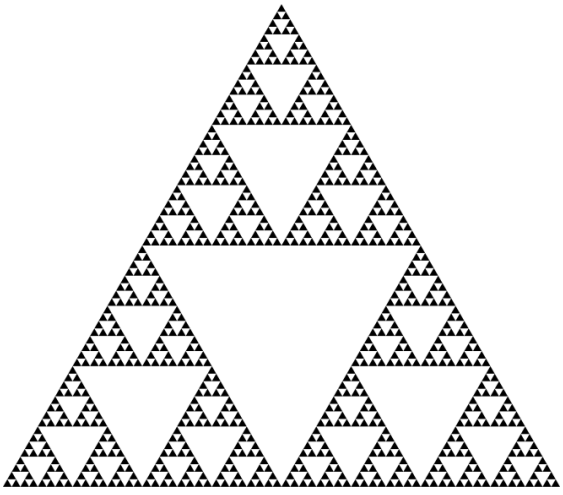


Abbildung 5.9: Sierpinski Dreieck

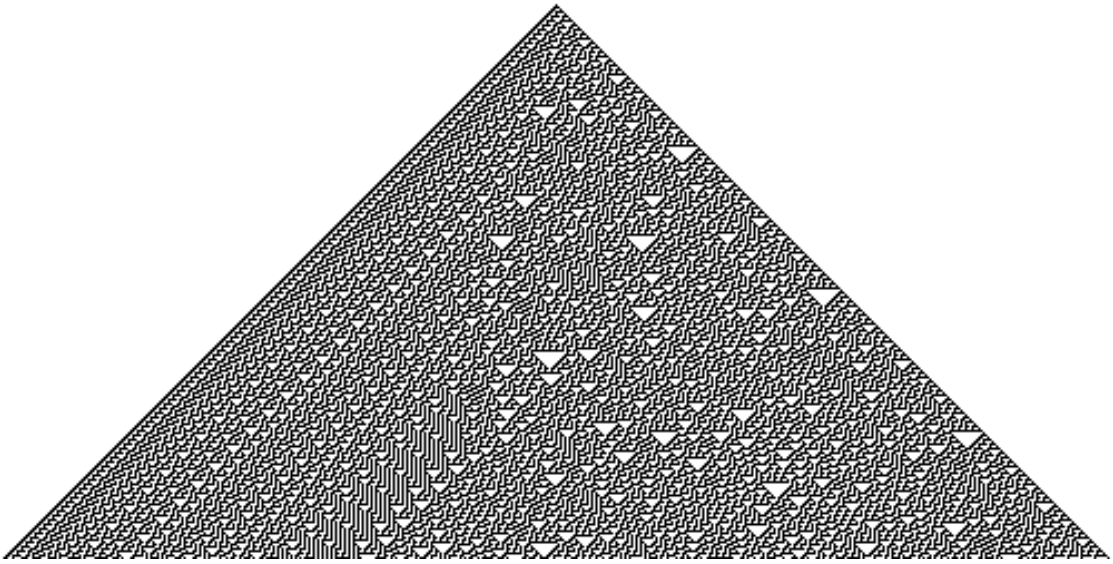


Abbildung 5.10: Anwendung von Rule 30

Fraktale können auch in der Natur beobachtet werden:

- Romanesco
- Eiskristalle
- Adhäsionsmuster
- Muster elektrischer Ladung
- Farn

5.4.4 Zelluläre Automaten

In einem zellulären Automaten definiert sich der Zustand einer Zelle z zum Zeitpunkt $t+1$ über die Zustände von z und von einer festen vorgegeben Nachbarschaft von z zum Zeitpunkt t.

Betrachten wir *Rule 30*. Die Zustände des eindimensionalen Automaten lassen sich über folgende Tabelle definieren.

Aktueller Zustand von z und seinen beiden Nachbarn	111	110	101	100	011	010	001	000
Neuer Zustand von z	0	0	0	1	1	1	1	0

Da 00011110 binär die Zahl 30 im Dezimalsystem ist, erhält Rule 30 seinen Namen. Wendet man Rule 30 an, erhält man die in Abbildung 5.10 gezeigte Figur.

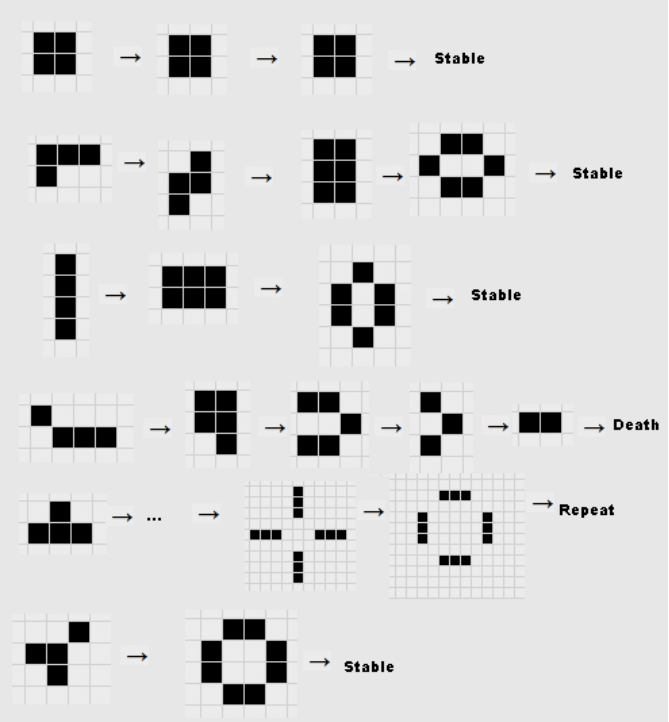


Abbildung 5.11: Anwendung vom Game of Life auf verschiedene Ausgangssituationen

In einem zweidimensionalen Automaten (oder auch *Game of Life*) besitzt jede Zelle 8 Nachbarn und ist entweder lebend oder tot. Es können folgende Zustandsübergänge definiert werden:

(1) Eine lebende Zelle mit weniger als zwei lebenden Nachbarn stirbt

(2) Eine lebende Zelle mit mehr als drei Nachbarn stirbt

(3) Eine lebende Zelle mit zwei oder drei Nachbarn lebt weiter

(4) Eine tote Zelle mit genau drei lebenden Nachbarn wird lebend

Eine Anwendung dieser Regeln kann in Abbildung 5.11 gesehen werden.

5.5 Quicksort

In diesem Abschnitt betrachten wir *Quicksort*. Wie Mergesort ist auch bei Quicksort Divide and Conquer ein Hauptbestandteil. Wir wollen also das Array in zwei Teilarrays aufteilen und dort rekursiv weiter sortieren. Die Besonderheit bei Quicksort ist, dass wir keinen Merge-Schritt benötigen, sondern das Array anhand einens *Pivot*-Elements so aufteilen, dass die Menge in kleinere und größere Elemente aufgeteilt wird. Ein Problem ist, dass die Balance der Aufteilung im vorneherein nicht absehbar ist.

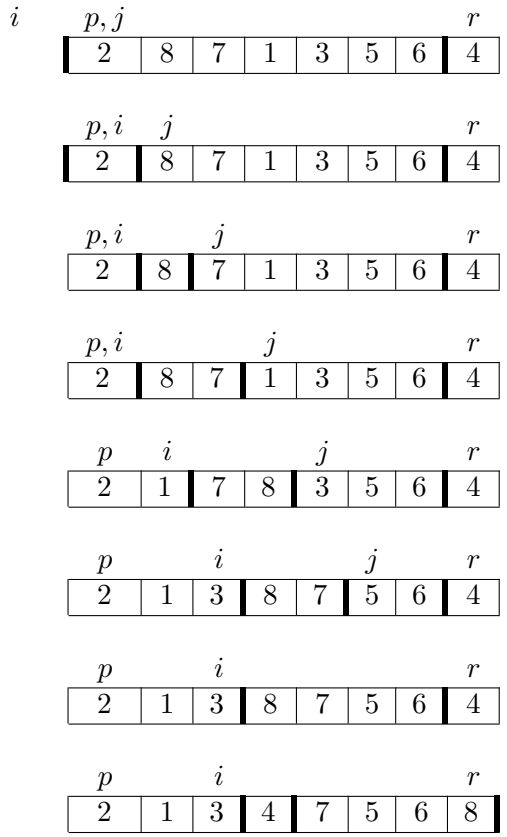
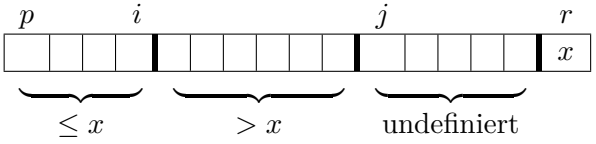


Abbildung 5.12: Ablauf von Quicksort anhand eines Beispiel-Arrays.

5.5.1 Ablauf Quicksort

Der Ablauf von Quicksort ist in Abbildung 5.12 zu sehen. Für einen Zwischenzustand von Quicksort gilt:



5.5.2 Algorithmische Beschreibung

In Algorithmus 5.11 wird zunächst das Array über Partition (Algorithmus 5.12) so modifiziert, sodass links von q (dem Pivotelement) nur kleinere und rechts von q nur größere Elemente existieren. Dann wird rekursiv auf die beiden von q induzierten Teilarrays Quicksort aufgerufen.

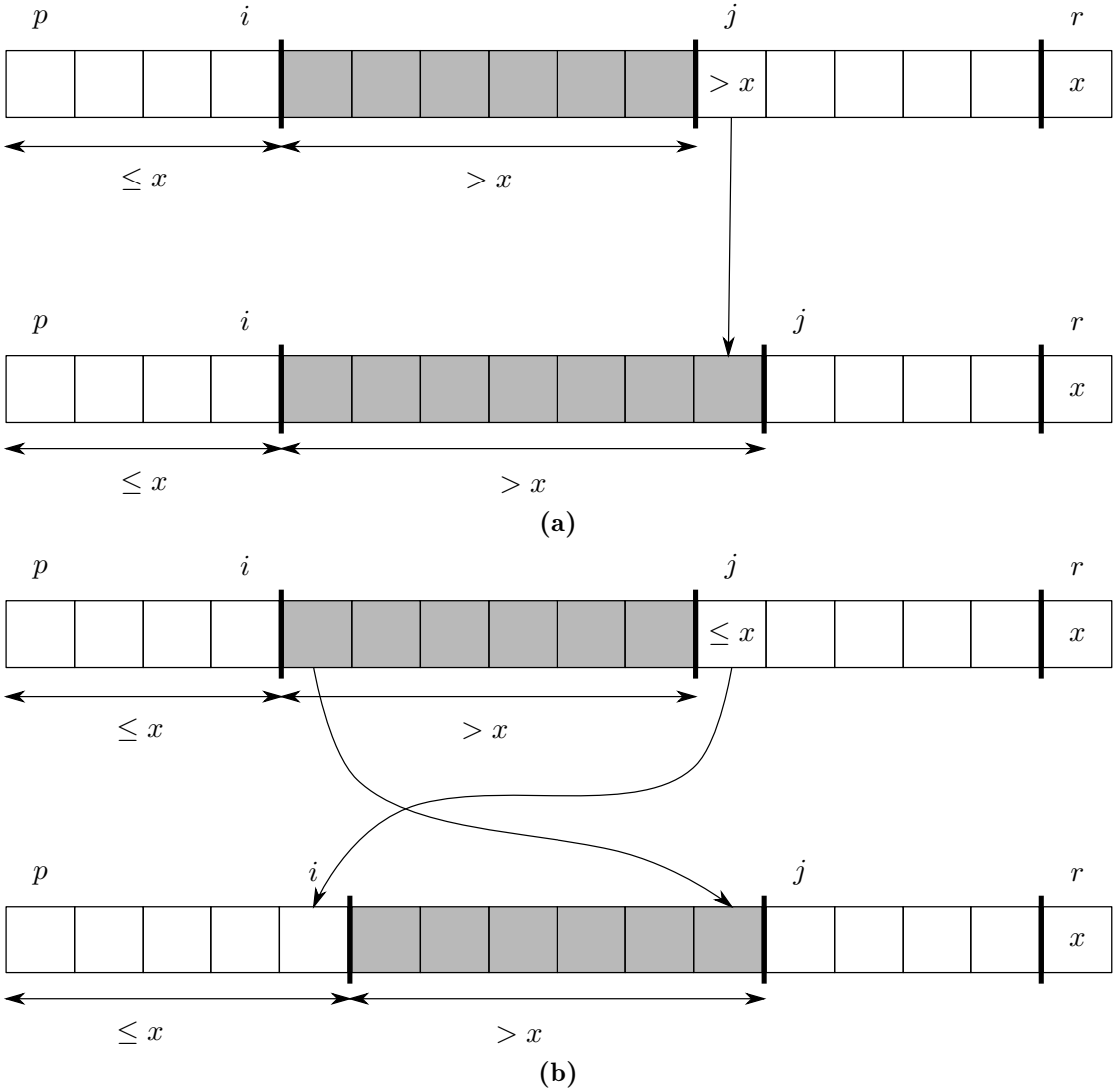


Abbildung 5.13: Allgemeiner Ablauf von Quicksort

Eingabe: Subarray $A = [1, ..., n]$, das bei Index p beginnt und bei Index r endet, also $A[p, ..., r]$

Ausgabe: Sortiertes Subarray

```
1: function QUICKSORT(A, p, r)
2:     if p < r then
3:         q ← PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
```

Algorithmus 5.11: Algorithmus Quicksort, der ein Subarray $A[p, ..., r]$ sortiert.

Eingabe: Subarray $A = [1, ..., n]$, das bei Index p beginnt und bei Index r endet, also $A[p, ..., r]$

Ausgabe: Zwei Subarrays $A[p, ..., q-1]$ und $A[q+1, ..., r]$ mit $A[i] \leq A[q]$ und $A[q] < A[j]$ für $i = p, ..., q - 1$ und $j = q + 1, ..., r$

```
1: function PARTITION(A, p, r)
2:     x ← A[r]
3:     i ← p − 1
4:     for j ← p to r − 1 do
5:         if A[j] ≤ x then
6:             i ← i + 1
7:             vertausche A[i] ↔ A[j]
8:     vertausche A[i + 1] ↔ A[r]
9:     return i + 1
```

Algorithmus 5.12: Algorithmus Partition, der ein Subarray $A[p, ..., r]$ in zwei Subarrays aufteilt, wovon eines nur Elemente enthält, die kleiner als ein gewähltes Pivotelement aus $A[p, ...r]$ sind, und das andere nur Elemente enthält, die größer als das gewählte Pivotelement sind.

5.5.3 Laufzeit von Quicksort

Wie viele Schritte benötigt Quicksort für ein Array der Länge n?
Unterscheidung:

- Worst Case
- Best Case
- Average Case

Quicksort: Worst Case

Der schlechteste Fall tritt ein, wenn das Pivotelement extremal liegt, d. h. nach Partition (s. Alg. 5.12) ist ein Subarray $(n-1)$ lang, das andere hat Länge 0.

$$T(n) = T(n-1) + T(0) + \Theta(n) \tag{5.57}$$
$$= T(n-1) + \Theta(n) \tag{5.58}$$

Man sieht (und beweist leicht), dass

$$\sum_{i=0}^{n} \Theta(n-i) = \Theta(n^2) \tag{5.59}$$

gilt, also liegt Quicksort im Worst Case in $\Theta(n^2)$.

Quicksort: Best Case

Im besten Fall liegt das Pivotelement genau in der Mitte des Arrays, d. h. nach Partition haben beide Subarrays im Wesentlichen die Länge $\frac{n}{2}$.

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n) \tag{5.60}$$

Man sieht z. B. mit dem Mastertheorem, dass Quicksort daher im Best Case in $\Theta(n \log(n))$ liegt. Das funktioniert auch noch, wenn Partition nur ein *annähernd* balanciertes Ergebnis liefert.

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn \tag{5.61}$$

Man sieht wieder mit dem Mastertheorem, dass Quicksort auch für diesen Sonderfall noch in der Best Case Laufzeit von $\Theta(n \log(n))$ liegt.

Quicksort: Average Case

Satz 5.13 (Erwartete Laufzeit von Quicksort)**.** *Für ein n-elementiges Array A hat Quicksort eine erwartete Laufzeit von $O(n \log(n))$.*

Beweis. Man betrachte eine zufällige Eingabepermutation, d. h. alle $n!$ Permutationen sind gleich wahrscheinlich mit jeweils Wahrscheinlichkeit $\frac{1}{n!}$.
Zwei Elemente s_i und s_j werden

- höchstens einmal verglichen.
- genau dann verglichen, wenn eines das Referenzelement (Pivot-Element) ist.

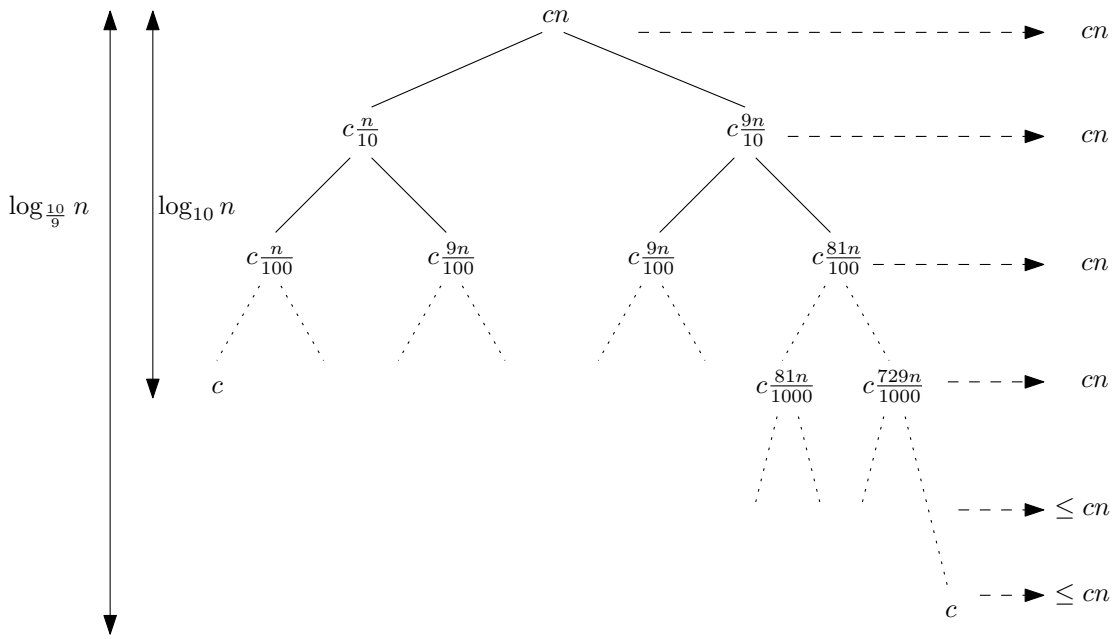
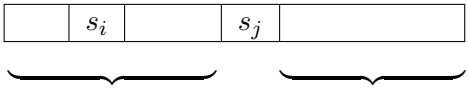


Abbildung 5.14: Veranschaulichung der Laufzeiten aller Subroutinen von Quicksort für den Best Case.

Danach sind beide in verschiedenen Subarrays!



Man betrachte eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } s_i \text{ und } s_j \text{ verglichen werden} \\ 0 & \text{sonst} \end{cases}. \tag{5.62}$$

Dann ist

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \tag{5.63}$$

die gesamte Anzahl der Vergleiche. Damit ergibt sich

$$\begin{aligned} E[x] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}], \end{aligned} \tag{5.64}$$

wobei $E[X_{ij}] = 1 \cdot P(X_{ij} = 1) + 0 \cdot P(X_{ij} = 0)$. Man muss also die Wahrscheinlichkeit bestimmen, dass s_i und s_j verglichen werden. Dafür betrachte man

$$S_{ij} \coloneqq \{s_i, ..., s_j\}, \tag{5.65}$$

also: $\boxed{s_i \mid \ldots \mid s_j}$. Dann ist

$$P(s_i \text{ wird mit } s_j \text{ verglichen}) = P(s_i \text{ oder } s_j \text{ ist erster Pivot in } S_{ij}). \tag{5.66}$$

Denn wird s_i oder s_j gewählt, wird es mit allen anderen verglichen, wird aber zuerst ein Pivot dazwischen gewählt, werden s_i und s_j getrennt und nicht mehr verglichen! Da nur einer erster Pivot sein kann, ist

$$\begin{aligned} P(s_i \text{ oder } s_j \text{ erster Pivot in } S_{ij}) &= P(s_i \text{ erster Pivot in } S_{ij}) + P(s_j \text{ erster Pivot in } S_{ij}) \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}, \end{aligned} \tag{5.67}$$

denn die Pivots werden zufällig sowie unabhängig gewählt und die Menge S_{ij} hat $(j-i+1)$ Elemente.
Jetzt kombiniert man 5.64 und 5.67 und erhält

$$E[x] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-1+i} \tag{5.68}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \qquad (k \coloneqq j-i) \tag{5.69}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}. \tag{5.70}$$

Jetzt schätzt man die harmonische Reihe ab. Es ergibt sich also

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + ... + \frac{1}{n} \tag{5.71}$$

$$\leq \underbrace{1}_{1} + \underbrace{\frac{1}{2} + \frac{1}{2}}_{1} + \underbrace{\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}}_{1} + ... + \frac{1}{2^{\log_2(n)}}. \tag{5.72}$$

Das sind $(1 + \log_2(n))$ Einsen, also

$$\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \log_2(n). \tag{5.73}$$

Somit gilt

$$\sum_{k=1}^{n} \frac{2}{k} \in \Theta(\log(n)) \tag{5.74}$$

und damit $E[x] \in O(n \log(n))$. □

Warum ist das interessant?

Eingabe: Array $A[1], ..., A[n]$ mit Schlüsselwerten aus $\{1, ..., k\}$

Ausgabe: Sortierte Kopie $B[1], ..., B[n]$ von $A[1], ..., A[n]$

```
1: function COUNTINGSORT(A, B, k)
2:     for i ← 1 to k do
3:         C[i] ← 0                                  ▷ Zählerarray initialisieren
4:     for j ← 1 to n do
5:         C[A[j]] ← C[A[j]] + 1        ▷ Anzahl Elemente mit Wert A[j] um 1 erhöhen
6:     for i ← 1 to k do
7:         C[i] ← C[i] + C[i − 1]       ▷ Anzahl Elemente mit Wert höchstens i zählen
8:     for j ← n downto 1 do
9:         B[C[A[j]]] ← A[j]          ▷ Element A[j] an die richtige Stelle in B schreiben
10:        C[A[j]] ← C[A[j]] − 1
```

Algorithmus 5.14: Algorithmus Countingsort, der n-elementiges Array A nimmt und eine sortierte Kopie B von A zurückgibt.

5.6 Sortieren in linearer Zeit

In Satz 5.6 wurde gezeigt, dass das Sortieren von n Objekten nicht schneller als in $\Omega(n \log(n))$ möglich ist - vorausgesetzt man kann dafür keine Zusatzinformationen einsetzen und darf Objekte nur paarweise vergleichen (und ggf. vertauschen). In vielen Situationen hat man aber zusätzliche Informationen! Dies soll im Folgenden analysiert werden.

5.6.1 Countingsort

Im einfachsten Fall sind die n Objekte Zahlen, aus dem Bereich $1, ..., k$. Wenn nun k klein ist (insbesondere $k \in O(n)$), dann gibt es einen einfachen Ansatz, der ganz ohne Vergleiche auskommt:

- Lege ein „Zählarray“ $C[0], ..., C[k]$ an.
- Gehe das zu sortierende Array $A[1], ..., A[n]$ durch. Für jedes Objekt $A[j]$ erhöhe den entsprechenden Eintrag in C, d. h. inkrementiere $C[A[j]]$ um 1.
- Für $j = 1, ..., k$ schreibe jeweils die Zahl j in die nächsten $C[A[j]]$ Einträge von $B[i]$. So liefert am Ende $B[1], ..., B[n]$ eine sortierte Form von $A[1], ..., A[n]$.

Im Detail ist dies in Algorithmus 5.14 zu sehen.

Satz 5.15. *Für n Objekte mit Schlüsseln aus $\{1, ..., k\}$ benötigt Countingsort (s. Alg. 5.14) die Zeit $O(n + k)$. Insbesondere benötigt man für $k \in O(n)$ die Zeit von $\Theta(n)$.*

5.6.2 Radixsort

Idee:
Sortiere nicht gleich nach den Schlüsselwerten, sondern nach den *Ziffern* der Schlüsselwerte! Dies wurde frühere sehr häufig bei Lochkarten eingesetzt.

Naiver Ansatz:
Sortiere nach größter Ziffer, dann nach zweiter Ziffer, ...

Problem: Entweder benötigt man viele temporäre Zwischenmengen oder die Vorsortierung geht kaputt!

Besserer Ansatz:
Sortiere nach letzter Ziffer, dann nach vorletzter Ziffer, ...

Ein Beispiel:

329		720		720		329
457		355		329		355
657		436		436		436
839	$\rightarrow$	457	$\rightarrow$	839	$\rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839

Wichtig: Jeweilige Sortierung darf Reihenfolge gleichwertiger Ziffern nicht verändern!

Definition 5.16 (Stabilität)**.** *Ein Sortierverfahren heißt* stabil*, wenn gleiche Werte nach dem Algorithmusdurchlauf in der gleichen Reihenfolge sind wie vorher.*

Satz 5.17 (Stabilität)**.** *Mergesort und Countingsort sind stabil, Quicksort je nach Umsetzung der Pivot-Regel.*

Beweis. Übung! □

Satz 5.19. *Radixsort (s. Alg. 5.18) liefert ein korrektes Ergebnis in der Zeit* $\Theta(d \cdot (n+k))$.

Beweis. Für je zwei Zahlen muss am Ende die Reihenfolge stimmen. Diese entspricht der Ordnung der höchsten Ziffern, die verschieden sind. Die Sortierung nach diesen Ziffern liefert also die richtige Reihenfolge, die sich wegen der Stabilität (von Countingsort) nicht mehr ändert. Die Laufzeit folgt aus Satz 5.15. □

Eingabe: n Zahlen mit je d Ziffern, die k verschiedene Werte annehmen können, $A[i], ..., A[n]$

Ausgabe: Sortiertes Array

```
1: function RADIXSORT(A, d)
2:     erzeuge (mit 0 initialisiertes) Array B
3:     for i ← 1 to d do
4:         COUNTINGSORT(A, B, i)
```

Algorithmus 5.18: Algorithmus Radixsort, der ein Array mit n Zahlen aus je d Ziffern, die k verschiedene Werte annehmen können, sortiert.

Eingabe: n Zahlen $A[i], ..., A[n]$

Ausgabe: Sortiertes Array

```
1: function SPAGHETTISORT(A, n)
2:     for i ← 1 to n do
3:         schneide eine Spaghetti der Länge A[i] zurecht
4:     Stoße alle Spaghetti auf den Tisch
5:     for i ← 1 to n do
6:         wähle die längste verbleibende Nudel
```

Algorithmus 5.20: Algorithmus Spaghettisort, der ein Array mit n Zahlen sortiert.

5.6.3 Spaghettisort

Nicht ganz ernst gemeint! Bei Algorithmus 5.20 handelt es sich um einen „analogen Algorithmus“- nicht digital!

Satz 5.21. *Spaghettisort (s. Alg. 5.20) sortiert in linearer Zeit!*

Beweis. Scheint klar! □

Problematisch: Rechnermodell, Rechenoperationen, Speicherplatz, Ausführbarkeit (und Genauigkeit!) für große n...

5.7 Suchen

In diesem Kapitel betrachten wir eine Menge $X = \{x_1, \ldots, x_n\}$ von n paarweise verschiedenen Zahlen.

Definition 5.22. *(Rank k Element) Sei $m \in X$. Dann besitzt m den Rank k (oder ist das k-te Element), wenn*

(1) $|\{x \in X \mid x \leq m\}| \geq k$ *und*

(2) $|\{x \in X \mid x \geq m\}| \geq n - k + 1$.

Speziell heißt m Median*, wenn er den Rank $\lceil \frac{n}{2} \rceil$ besitzt.*

Wenn X und k bekannt sind, wie schnell kann man das k-te Element berechnen? Ein erster Ansatz ist schnell gefunden:

- Sortiere X aufsteigend
- finde k-tes Element mit linearem Durchlauf

Die Korrektheit ist schnell zu sehen. An Zeit benötigen wir $O(n \log n) + O(k) = O(n \log n)$. Das geht schneller, z.B. mit dem Select-Algorithmus (siehe Algorithmus 5.23). Beachte, dass es in dem Select-Algorithmus maximal zwei rekursive Aufrufe gibt: Zeile 9 und entweder Zeile 14, Zeile 18 oder keinen weiteren.

Beispiel 5.24. *Sei $n = 25$ und $k = 23$, sowie*

$$X = \{1, 22, 10, 13, 24, 6, 18, 21, 4, 25, 11, 16, 2, 20, 8, 17, 5, 12, 19, 14, 3, 9, 15, 7, 23\}$$

Aufteilen in $t = \lceil \frac{n}{5} \rceil = 5$ Blöcke:

1	6	11	17	3
22	18	16	5	9
10	21	2	12	15
13	4	20	19	7
24	25	8	14	23

$\longrightarrow$ *Gruppen sortieren*

1	4	2	5	3
10	6	8	12	7
13	18	11	14	9
22	21	16	17	15
24	25	20	19	23

Eingabe: Menge von Zahlen $X = \{x_1, \ldots, x_n\}$ und eine Zahl $k \leq n$

Ausgabe: k-tes Element

```
1:  function SELECT(X,k)
2:      if |X| ≤ 5 then
3:          Bestimme k-tes Element m mit erstem Ansatz
4:          return m
5:      else
6:          Unterteile X in fünfer Blöcke B_1, ..., B_t mit t := ⌈n/5⌉
7:          Bestimme in jedem B_i das mittlere Element m_i (bzgl. Sortierung in B_i)
8:          Setze M' := ⋃_{i=1}^{t} {m_i}
9:          m ← SELECT(M', ⌈t/2⌉)
10:         X_1 = {x ∈ X | x < m}
11:         X_2 = {x ∈ X | x = m}
12:         X_3 = {x ∈ X | x > m}
13:         if |X_1| ≥ k then
14:             return SELECT(X_1, k)
15:         else if |X_1| + |X_2| ≥ k then
16:             return m
17:         else
18:             return SELECT(X_3, k − (|X_1| + |X_2|))
```

Algorithmus 5.23: Select Algorithmus zum finden des k-ten Elements

Wir erhalten $M' = \{m_1, \ldots, m_t\} = \{13, 18, 11, 14, 9\}$. *Durch den rekursiven Aufruf* SELECT(M',$\lceil \frac{t}{2} \rceil$) = SELECT($\{13, 18, 11, 14, 9\}$, *3) erhalten wir* $m = 13$ *(Da* $|M'| \leq 5$ *ist, brauchen wir nur den ersten Ansatz benutzen).*

Wir berechnen nun:

- $X_1 = \{1, 10, 6, 4, 11, 2, 8, 5, 12, 3, 9, 7\}$
- $X_2 = \{13\}$
- $X_3 = \{22, 24, 18, 21, 25, 16, 20, 17, 19, 14, 15, 23\}$

Es gilt: $|X_1| = 12 < k = 23 \Rightarrow$ *nicht Zeile 14;* $|X_1| + |X_2| = 13 < k = 23 \Rightarrow$ *nicht Zeile 16. Wir kommen also in Zeile 18 und rufen* SELECT(X_3, $k - (|X_1| + |X_2|)$) = SELECT($\{22, 24, 18, 21, 25, 16, 20, 17, 19, 14, 15, 23\}$, *10) auf.*

Also: $X = \{22, 24, 18, 21, 25, 16, 20, 17, 19, 14, 15, 23\}$, $k = 10$
Aufteilen in fünfer Blöcke liefert:

22	*16*	*15*
24	*20*	*23*
18	*17*	
21	*19*	
25	*14*	

$\longrightarrow$ *Gruppen sortieren*

18	*14*	*15*
21	*16*	*23*
22	*17*	
24	*19*	
25	*20*	

Wir erhalten $M' = \{m_1, \ldots, m_t\} = \{22, 17, 23\}$. *Rekursiver Aufruf* SELECT(M', *2) liefert* $m = 22$ *(wieder war nur der erste Ansatz nötig, da* $|M'| \leq 5$ *ist).*

Wir berechnen:

- $X_1 = \{18, 21, 16, 20, 17, 19, 14, 15\}$
- $X_2 = \{22\}$
- $X_3 = \{24, 25, 23\}$

Es gilt: $|X_1| = 8 < k = 10 \Rightarrow$ *nicht Zeile 14;* $|X_1| + |X_2| = 9 < k = 10 \Rightarrow$ *nicht Zeile 16. Wir kommen also wieder in Zeile 18 und rufen* SELECT(X_3, $k - (|X_1| + |X_2|)$) = SELECT($\{24, 25, 23\}$, *1) auf. Da* $|X_3| \leq 5$ *genügt der erste Ansatz und wir erhalten* $m = 23$. *Also ist 23 das 23-te Element von*

$$\{1, 22, 10, 13, 24, 6, 18, 21, 4, 25, 11, 16, 2, 20, 8, 17, 5, 12, 19, 14, 3, 9, 15, 7, 23\}$$

Das Konzept der Partitionierung wurde bereits bei Quicksort (siehe Kapitel 5.5) gesehen.

Satz 5.25. *Select (Algorithmus 5.23) ist korrekt, d.h. er bestimmt das k-te Element x*

Beweis. Wir beweisen den Satz mittels vollständiger Induktion über $n = |X|$. Für $n \leq 5$ ist es klar, dass das korrekte Element zurückgegeben wird (erster Ansatz). Nehmen wir nun an, dass Select für Mengen mit Größe kleiner n das k'-te Element bestimmt. Betrachten wir eine Menge mit n Elementen. Es treten drei Fälle auf:

(1) $x \in X_1$

(2) $x \in X_2$

(3) $x \in X_3$

Alle drei Mengen besitzen weniger als n Elemente, d.h. nach Induktionsvoraussetzung können wir das k-te Element in jedem Fall korrekt bestimmen. □

Wie schnell ist Select? Wir können die rekursive Gleichung aufstellen:

$$T(n) = \underbrace{T(\frac{n}{5})}_{\text{Rekursiver Aufruf für } M'} + \underbrace{T(\max\{|X_1|, |X_3|\})}_{\text{Rekursiver Aufruf für } X_1 \text{ oder } X_3} + \underbrace{O(n)}_{\text{Unterteilung in und Sortierung der fünfer Blöcke}}$$

Wie groß können X_2 und X_3 maximal sein? Wir beantworten diese Frage durch das Ausschlussprinzip: Welche Elemente sind nicht in X_1 bzw. X_3? Betrachte ein Beispiel. Wir haben sortierte Blöcke:

1	4	2	5	3
10	6	8	12	7
13	18	11	14	9
22	21	16	17	15
24	25	20	19	23

Betrachte sortierte Anordnung der Blöcke bzgl. m $\longrightarrow$ nur für Analyse nicht im Algorithmus

3	2	1	5	4
7	8	10	12	6
9	11	13	14	18
15	16	22	17	21
23	20	24	19	25

Beobachte: In dem 3×3 Block links von m nur Elemente $\leq m$, gehören also nicht zu X_3. Dies sind etwa $\frac{1}{4}|X|$ Elemente, also enthält X_3 maximal $\frac{3}{4}|X|$ Elemente. Analog kann man das für $|X_1| \leq \frac{3}{4}|X|$ argumentieren.

Lemma 5.26. *$|X_1|, |X_3| \leq \frac{3}{4}n$ für $n \geq 50$.*

Beweis. Seien $m_1 < m_2 < \cdots < m_t$. Schreibe die Blöcke nebeneinander:

		...		...		
		...		...		
m_1	m_2	...	m	...	m_{t-1}	m_t
		...		...		
		...		...		

Es gilt $m_1, \ldots, m_{\lceil t/2 \rceil} \leq m$. In jedem Block B_i mit $i \leq \lceil t/2 \rceil$ gibt es drei Elemente $\leq m$, die nicht unterhalb m_i liegen. Es gilt also:

$$|\{x \in X | x \leq m\}| \geq 3 \cdot \lceil \frac{t}{2} \rceil^{-1} = 3(\lceil \frac{n}{10} \rceil - 1)$$

$$\Rightarrow |X_3| \leq n - 3(\lceil \frac{n}{10} \rceil - 1)$$

Analog verläuft man für X_1. Also: $|X_1| \leq n - 3(\lceil \frac{n}{10} \rceil - 1)$

Es gilt $n - 3(\lceil \frac{n}{10} \rceil - 1) \leq \frac{3}{4}n$ für $n \geq 50$:

$$\begin{aligned} n - 3(\lceil \frac{n}{10} \rceil - 1) &\leq n - \frac{3}{10}n + 1 \\ &\leq \frac{7n}{10} + \frac{n}{50} = \frac{36n}{50} \leq \frac{3}{4}n \end{aligned}$$

□

Satz 5.27. *Select hat eine Laufzeit von $O(n)$.*

Beweis.

$$T(n) = \underbrace{T(\frac{n}{5})}_{\substack{\text{Rekursiver Aufruf} \\ \text{auf } M'}} + \underbrace{T(\frac{3}{4}n)}_{\substack{\text{Rekursiver Aufruf} \\ \text{auf } X_1 \text{ oder } X_3}} + \underbrace{O(n)}_{\substack{\text{Unterteilung in und} \\ \text{Anordnung der Blöcke}}} + \underbrace{O(n)}_{\substack{\text{Berechnung von} \\ X_1, X_2 \text{ und } X_3}}$$

Mit dem Mastertheorem erhalten wir $m = 2, \alpha_1 = \frac{1}{5}, \alpha_2 = \frac{3}{4}, k = 1$. Es gilt $\sum_{i=1}^{m} \alpha_i^k = \frac{1}{5} + \frac{3}{4} = \frac{19}{20} < 1$. Also $T(n) \in \Theta(n^k) = \Theta(n)$ □

Satz 5.28. *Das k-te Element kann nicht schneller als $O(n)$ bestimmt werden.*

Beweis. Wir zeigen, dass mindestens $n - 1$ Vergleiche nötig sind. Nehmen wir an, es reichten $n - 2$ Vergleiche in einem beliebigen Algorithmus aus. Sei x das k-te Element. Betrachte den Graphen $G = (V, E)$ mit $V \widehat{=} X$ und $\{x_i, x_j\} \in E$ falls x_i mit x_j verglichen wird. G besitzt $n-2$ Kanten und kann demnach nicht zusammenhängend sein. Es gibt also ein Element $y \in X$, sodass y und x in unterschiedlichen Zusammenhangskomponenten sind. Dann hängt der Algorithmus nicht vom Wert y ab. Widerspruch. □

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Pseudocodeverzeichnis

Literaturverzeichnis

[1] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[2] Thomas H Cormen, Charles E Leiserson, Ronald Rivest, and Clifford Stein. *Algorithmen-Eine Einführung*. Oldenbourg Verlag, 2010.

[3] Nikolai Krützmann. Logistische Parabel http://www.mp3-kolleg.de/fab-inet, February 2015.

[4] Wikipedia. Wikipedia – Algorithmus, March 2014.

[5] Wikipedia. Wikipedia – Fibonacci Heap, May 2014.

[6] Wikipedia. Wikipedia – Turingmaschine, October 2014.

[7] Wikipedia. Wikipedia – Mandelbrot set, February 2016.