Rheinische Friedrich-Wilhelms-Universität Bonn
Mathematisch-Naturwissenschaftliche Fakultät

Algorithmen und Berechnungskomplexität I

Skript WS 2015/2016

Nach Aufzeichnungen von Rolf Klein, Elmar Langetepe und Heiko Röglin

Bonn, Oktober 2015

Inhaltsverzeichnis

Kapitel 1

Einführung

In dieser Vorlesung wird die Komplexität von Berechnungsproblemen untersucht und wir werden die dafür notwendigen theoretischen Konzepte entwickeln. Für viele Problemstellungen werden Algorithmen (Lösungspläne) vorgestellt und analysiert.

1.1 Algorithmen

Formal gesehen ist ein *Algorithmus* ein Lösungsplan, der für eine *Probleminstanz* eines definierten Berechnungsproblems die enstprechende *Ausgabe* liefert.

Beispiel: **Sortierproblem**

Eingabe: Eine Folge von n reellen Zahlen $(a_1, a_2, \ldots, a_n)$.

Ausgabe: Eine Permutation π, so dass $(a_{\pi(1)}, a_{\pi(2)}, \ldots a_{\pi(n)})$ eine sortierte Folge der Eingabe ergibt, d.h., $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$.

Konkrete Eingabefolgen werden *Instanzen* genannt.

Bezüglich der Algorithmen und der Problemstellung werden wir uns insgesamt zunächst mit drei Begriffen auseinandersetzen:

- Korrektheit: Wir wollen für die Algorithmen formal beweisen, dass zu jeder Instanz das richtige Ergebnis berechnet wird.
- Laufzeitabschätzung: Wie ist die Laufzeit des Algorithmus in Bezug auf die Eingabegröße n und wie effizient ist der Algorithmus im Vergleich zu anderen möglichen Algorithmen?
- Speicherplatzabschätzung: Wieviel Speicher in Bezug auf die Eingabegröße n benötigt der Algorithmus und wie effizient ist das im Vergleich zu anderen möglichen Algorithmen?

Die hier betrachteten Algorithmen sollen mithilfe eines Computers ihre Aufgabe erfüllen, deshalb werden wir uns an gängigen Programmiersprachen orientieren und dazu *Pseudocode* verwenden.

1.1.1 Anwendungsgebiete

Algorithmen spielen in verschiedenen Anwendungsgebieten eine große Rolle. Die Problemstellungen werden durch mathematische Modellierung so formuliert, dass sie durch einen Rechner bearbeitet werden können.

- Human Genome Project: Datenanalyse, DNA-Vergleiche, Matchingalgorithmen, ...
- Internet: Routing, Datenhaltung, Suchmaschinen, Routenplaner, Verschlüsselung, ...
- Industrie: Beladen von Containern, Optimieren von Arbeitsabläufen, ...
- Infrastruktur: Optimieren von Netzen, Umwege minimieren, Kürzeste Wege berechnen/approximieren, ...
- ...

Ein Beispiel aus unserer aktuellen algorithmischen Forschung in Zusammenarbeit mit dem FKIE Wachtberg.

Wir betrachten ein Gebiet, in dem es einen Industrie-Unfall gegeben hat und das von einer Menge von Robotern beobachtet werden soll. Das Gebiet ist durch den Menschen nicht mehr betretbar. Die Roboter haben einen beschränkten Senderadius und sollen den Kontakt untereinander und zur Basisstation halten. Von der Basisstation aus sollen die Roboter zu vorher festgelegten Positionen gefahren werden. Die Positionen und die Wege der einzelnen Positionen zueinander konnten berechnet werden. Damit die Roboter den Kontakt untereinander nicht verlieren, ist es manchmal erforderlich, mehrere Roboter von Position A zu Position B zu bewegen. Für eine Bewegung von A nach B sind somit zum Beispiel mindestens 10 Roboter notwendig.

Die Frage lautet, wieviel Roboter brauchen wir, um von einer Basisstation alle gegebenen Positionen zu besetzen und bei der Bewegung zu gewährleisten, dass die Agenten alle in Kontakt bleiben?

Modellbildung durch einen Graphen $G = (V, E)$ mit ganzzahligen Knotengewichten w_v für $v \in V$ und ganzzahligen Kantengewichten w_e für $e \in E$, ein Startknoten v_s; siehe Abbildung 1.1.

Folgende Bedingungen:

- Eine Kante e darf nur mit $k \geq w_e$ Agenten traversiert werden.
- Ein Knoten v darf nur mit $k \geq w_v$ Agenten besucht werden.
- Beim ersten Besuch eines Knotens v bleiben w_v Agenten am Knoten zurück.

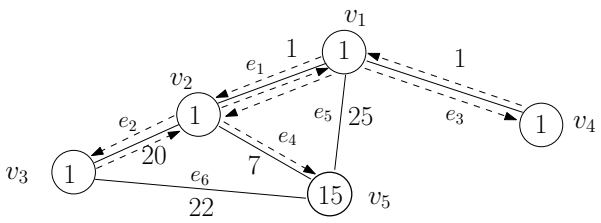

Abbildung 1.1: Falls die Agenten am Knoten v_1 starten, kann eine Tour mit einer minimalen Anzahl an Agenten wie folgt beschrieben werden. 23 Agenten in einer einzelnen Gruppe besuchen Knoten in der Reihenfolge v_1,v_2,v_3,v_4 and v_5. Die Kanten e_5 and e_6 werden nicht besucht.

Frage: Wieviele Agenten werden benötigt, um alle Knoten v mit w_v Agenten zu füllen? Welchen Weg können die Agenten dabei zurücklegen? Lässt sich ein effizienter Algorithmus formulieren, der dieses Problem für beliebige Instanzen löst?

Die Abbildung zeigt ein Beispiel: Vom festgelegten Basisknoten v_1 aus kommt man mit 23 Agenten aus. Beginnend am Knoten v_1 bleibt zunächst ein Agent zurück und 22 Agenten bewegen sich über die Kante e_1 nach v_2. Hier wird ein Agent abgelegt und mit 21 Agenten geht es zum Knoten v_3 über die Kante e_2. Nachdem wir dort einen Agenten abgelegt haben, können wir mit den verbleibenden 20 Agenten gerade noch über die Kante zurück. Hier stellt sich quasi heraus, dass es mit weniger als 23 gar nicht gehen kann!

Mit den 20 Agenten besuchen wir v_4 (über e_1 und e_3) und lassen einen Agenten dort. Dann geht es mit den verbleibenden über e_3, e_1 nnd e_4 zu v_5. Hier werden 15 Agenten benötigt und wir sind fertig.

Es stellt sich heraus, dass dieses Problem für allgemeine Graphen wirklich *schwer* zu lösen ist. Für einfache Graphen (Bäume) gibt es effiziente Algorithmen, durch die die optimale Lösung in best-möglicher Laufzeit (auch durch die Verwendung effizienter Datenstrukturen (Heaps)) ermittelt werden kann. Daraus resultiert auch eine Approximationslösung für allgemeine Graphen. Aus dem Graphen wird ein Baum mit guter Verbindungseigenschaft generiert und darauf der Baumalgorithmus angewendet.

Ein formaler Beweis zeigt jeweils die Güte der Approximation, die Optimalität des Baumalgorithmus und die Schwere des Problems im Allgemeinen.

Dieses Beispiel enthält somit viele Aspekte, die in der Vorlesung vermittelt werden sollen.

1.1.2 Maschinenmodell

Die Laufzeitabschätzungen der Algorithmen soll unabhängig von der jeweils aktuellen Hardware-Rechnerausstattung (z.B. Taktfrequenz, Verarbeitungsbreite etc.) sein, deshalb zählen wir die Anzahl der ausgeführten *Elementaroperationen* in unserem jeweiligen Maschinenmodell. Der Einfachheit halber veranschlagen wir für jede Operation in etwa die gleichen Kosten.

Wir verwenden das Modell einer sogenannten **REAL RAM**, eine *Random Access Maschine*, die mit reellen Zahlen rechnet. In diesem Modell werden Anweisungen nacheinander (sequentiell) ausgeführt. Das REAL RAM Modell orientiert sich an realen Rechnern und hat folgende Spezifikation:

- Abzählbar unendlich viele Speicherzellen, die mit den natürlichen Zahlen adressiert werden.
- Jede Speicherzelle kann eine beliebige reelle oder natürliche Zahl enthalten. Direkter oder indirekter Zugriff ist möglich.
- Elementare Rechenoperationen: Addieren, Subtrahieren, Multiplizieren, Dividieren, Restbilden, Abrunden, Aufrunden.
- Elementare Relationen: kleiner gleich, größer gleich, gleich, $\vee$, $\wedge$.
- Kontrollierende Befehle: Verzweigung, Aufruf von Unterroutinen, Rückgabe.
- Datenbewegende Befehle: Laden, Speichern, Kopieren.

Alle angegebenen Operationen können in konstanter Zeit ausgeführt werden.

Wir nehmen an, dass Integer-Zahlen der Größe m mit $c \log m$ Bits für eine Konstante $c \geq 1$ dargestellt werden können. Reelle Zahlen werden gemäß des IEEE Standards im Rechner durch Binärzahlen approximiert. Innerhalb des Darstellungsbereiches können arithmetische Operationen mit reellen Zahlen im Rechner sehr effizient durchgeführt werden.

1.1.3 Algorithmische Paradigmen

Algorithmen lassen sich häufig bezüglich ihrer Heransgehensweise an das jeweilige Problem kategorisieren. Zunächst seien hier einige davon kurz erwähnt. Es geht hier im Wesentlichen darum, wie das Problem gelöst wird.

- Brute-Force (Naive Methode)
- Inkrementelle Konstruktion (Schrittweise Vergrößerung der Eingabe)
- Divide-and-Conquer (Aufteilen und Zusammenfügen)
- Greedy (*gierig*, schnelle Verbesserungen)
- Dynamische Programmierung (Tabellarische Auflistung und Verwertung von Teillösungen)
- Sweep (geometrische Probleme, Dimensionsreduktion)
- ...

Die Methoden können auch kombiniert auftreten, zum Beispiel ein Divide-and-Conquer Algorithmus, der im Merge-Schritt einen Sweep verwendet. Auf die einzelnen Paradigmen werden wir später gezielt für konkrete Problemstellungen eingehen.

1.1.4 Analysemethode

Sei P eine Probleminstanz eines Problems Π und sei A ein Algorithmus mit RAM-Anweisungen, der für jede Instanz $P \in \Pi$ eine Lösung liefert. Im Sprachgebrauch wird eine Instanz P auch oft als Problem bezeichnet, wenn keine Verwechslung zu befürchten ist.

Die Anzahl der Elementaroperationen eines Algorithmus A bei einer Eingabegröße $|P| = n \in \mathbb{N}$ wird durch eine *Kostenfunktion* $T_A : \mathbb{N} \mapsto \mathbb{R}^+$ beschrieben. Die Instanz P könnte beispielsweise eine Menge von n Zahlen sein.

Die exakte Bestimmung dieser Funktion kann vernachlässigt werden, insbesondere, da auf unterschiedlichen Rechnern verschiedene Konstanten für die Elementaroperationen beachtet werden müssten. Also sind wir bei der Analyse nur an der Größenordung der Funktion T_A in Abhängigkeit der Eingabegröße $|P| = n$ interessiert. Ebenso verhält es sich mit der Eingabegröße selbst, ob wir es mit n Punkten in der Ebene zu tun haben oder mit $2n$ Koordinaten, soll beispielsweise keine Rolle bei der Beschreibung der Größenordnung spielen.

Die Anzahl der Elementaroperationen eines Algorithmus mit Eingabegröße $n \in \mathbb{N}$ kann auch von der Zusammenstellung der Eingabe selbst abhängen. Deshalb können wir verschiedene Kennzahlen verwenden:

- Worst-case: Maximal mögliche Anzahl an Elementaroperationen

- Average-case: Mittlere Anzahl der Elementaroperationen, gemittelt über alle möglichen Eingaben der entsprechenden Größenordnung

Die Analyse kleiner Eingabegrößen ist generell wenig aussagekräftig, da wir in diesem Fall die Summe aller Elementaroperationen unter einer großen Konstante subsumieren können. Deshalb untersuchen wir das *asymptotische Verhalten* der Kostenfunktion.

Das führt zu folgenden Notationen für Klassen von Funktionen:

Definition 1 (O-,Ω-,Θ-Notation)

$$\begin{aligned} g \in O(f) &:\Leftrightarrow \text{ es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\quad \text{so dass } g(n) \leq C\, f(n) \text{ für alle } n \geq n_0. \end{aligned}$$

$$\begin{aligned} g \in \Omega(f) &:\Leftrightarrow f \in O(g) \\ &:\Leftrightarrow \text{ es existiert ein } n_0 \geq 0 \text{ und ein } C > 0, \\ &\quad \text{so dass } f(n) \leq C\, g(n) \text{ für alle } n \geq n_0. \end{aligned}$$

$$g \in \Theta(f) :\Leftrightarrow g \in O(f) \text{ und } g \in \Omega(f).$$

Wir erlauben also mit $n \geq n_0$ endliche viele Ausnahmestellen.

Die obige Notation wird in dem Sinne gebraucht, dass $f \in O(g)$ eine *obere Schranke* an die Funktion f durch g liefert, währenddessen $f \in \Omega(g)$, eine *untere Schranke* für f durch g darstellt.

Beispiele:

$3n+2 \in O(n)$, da für alle $n \geq 2$ und für $C = 4$ gilt: $3n+2 \leq 4n$. Desgleichen gilt $3n + 2 \in \Omega(n)$, da für alle $n \geq 0$ bereits $n \leq 3n + 2$ gilt. Somit gilt $3n + 2 \in \Theta(n)$.

Für $g(n) = 2n^3 - 18n^2 - \sin(n) + 16n + 3$ führt zunächst die grobe Abschätzung

$$\begin{aligned} g(n) &\leq 2n^3 + 16n + 3 \\ &\leq (2 + 16 + 3)n^3 \text{ für alle } n \geq 1 \end{aligned}$$

zur Aussage $g \in O(n^3)$. Es gilt aber auch

$$\begin{aligned} g(n) &\geq 2n^3 - 18n^2 \\ &= n^3 + (n - 18)n^2 \\ &\geq n^3 \text{ für alle } n \geq 15 \end{aligned}$$

und deshalb gilt hier ebenfalls $g \in \Omega(n^3)$ und $g \in \Theta(n^3)$. Wir sagen auch, dass die Funktion g in der *Größenordnung* n^3 wächst.

Eine weitere hilfreiche Notationskonvention ist, dass wir die obigen Symbole der Einfachheit halber bei einer asymptotischen Betrachtung auch in Formeln verwenden möchten. So drückt $2n^2+3n+1 = 2n^2+\Theta(n)$ aus, dass wir uns um die Funktion $3n+1$ in der Größenordnung $\Theta(n)$ nicht genau kümmern möchten. Diese Konvention kann hilfreich sein, wenn beispielsweise eine Laufzeitabschätzung *rekursiv* durch eine Formel $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ beschrieben werden darf. Wahlweise kann hier auch $T(n) = 2T\left(\frac{n}{2}\right) + C \cdot n$ für ein nicht näher bestimmtes C verwendet werden.

Außerdem kann es sinnvoll sein, die Analyse der Laufzeiten *ausgabesensitiv* vorzunehmen, um die Ausgabekomplexität nicht der algorithmischen Komplexität anzulasten. Falls wir beispielsweise die Menge aller Schnittpunkte von n Liniensegmenten suchen, so ist klar, dass es im worst-case $\binom{n}{2}$ viele Schnittpunkte geben kann, jeder Algorithmus also mindestens diese Laufzeit im worst-case benötigt. Eine ausgabesensitive Analyse könnte zu einem Ergebnis von $O((n+k)\log n)$ führen, wobei k die Anzahl der tatächlichen Schnittpunkte angibt.

1.2 Ein einfaches Beispiel: Insertionsort

Die obigen Begriffe wollen wir zunächst auf das einfache Beispiel des Sortierens anwenden. Wir nehmen an, die n zu sortierenden Zahlen sind in einem *Array* A gespeichert.

Eingabe: n reelle Zahlen $A[1], A[2], \ldots, A[n]$.

Ausgabe: Ein Array B der Zahlen aus A mit $B[1] \leq B[2] \leq \cdots \leq B[n]$.

Beachte, dass wir hier eine etwas andere Ausgabe als zu Beginn erwarten (Permutation π). Die Aufgabenstellung ist aber äquivalent.

Eine *Brute-Force* Vorgehensweise wäre, das kleinste Element des Arrays zu ermitteln, dieses zu entfernen und mit dem restlichen Array fortzufahren.

Wir betrachten einen *inkrementellen* Ansatz. Sukzessive wird ein immer größerer Teil des Arrays A in B sortiert. Anders ausgedrückt, falls wir die ersten $j-1$ Elemente des Arrays A in B bereits sortiert haben, nehmen wir uns danach das j-te Element vor und sortieren es in B ein.

Die Korrektheit des Algorithmus wird durch eine *Schleifeninvariante* gezeigt. Wir zeigen formal, dass für jeden Beginn der FOR-Schleife die fogende Invariante gilt: *Für den aktuellen Wert j gilt, dass B bis zum Index $j-1$ eine sortierte Folge der ersten $j-1$ Einträge von A ist und dass A und B ab dem Index j übereinstimmen*

Formal gesehen handelt es sich hier um eine Induktion, die bei einem be-

Algorithmus 1 InsertionSort(A)
Array A (nichtleer) sortiert in Array B ausgeben

```
1: B[1] := A[1];
2: for j = 2 to length(A) do
3:    key := A[j]; i := j − 1;
4:    while i > 0 und B[i] > key do
5:       B[i + 1] := B[i]; i := i − 1;
6:    end while
7:    B[i + 1] := key;
8: end for
9: RETURN B
```

stimmten Parameter terminiert.

Induktionsanfang: $j = 2$. Die Eigenschaft ist erfüllt, durch die erste Anweisung.

Induktionsschritt: Wir nehmen an, dass die Aussage bereits für $j - 1 \geq 2$ gilt. Jetzt werden in der WHILE-Schleife im Array B solange die Werte um einen Index nach hinten geschoben, bis das Element $key := A[j]$ kleiner gleich dem i-ten Element von B ist. Dann wird key als $(i+1)$-tes Element in B eingetragen. Falls das bis $i = 0$ nicht eintritt, wird key als 1-tes Element von B eingetragen. Insgesamt ist B danach bis zum Index j ein sortiertes Array der ersten j-Elemente von A.

Terminierung: Falls $j = length(A) + 1$ wird, gilt zunächst die Invariante, dass für den aktuellen Wert j gilt, dass B bis zum Index $j-1 = length(A)$ eine sortierte Folge der ersten $length(A)$ Einträge von A ist. Die FOR-Schleife bricht ab und das Ergebnis wird korrekt in B ausgegeben.

Übungsaufgabe: Streng genommen muss die WHILE-Schleife genauso analysiert werden. Welche Schleifeninvariante muss hier gewählt werden?

Für eine Laufzeitabschätzung analysieren wir die einzelnen Programmschritte. Dabei sei t_j die Anzahl der Durchläufe der WHILE-Schleife wenn die FOR-Schleife für j aufgerufen wird. Sei $length(A) = n$.

Es ergibt sich die folgende Tabelle:

Anweisungen in Algorithmus 1	Kosten	Häufigkeit
1:	c_1	1
3:	c_2	$n - 1$
5:	c_3	$\sum_{j=2}^{n} t_j$
7:	c_4	$n - 1$

Insgesamt ergibt sich nach unseren Konventionen folgende Laufzeit:

$$T(n) = c_3 \sum_{j=2}^{n} t_j + O(n)$$

Die Laufzeit hängt also von der Größe von t_j ab. Im schlechtesten Fall ist das Array A absteigend sortiert, alle Elemente aus B müssen verschoben werden und das j-te Element wird vorne eingefügt. In diesem Fall gilt $t_j = j$ und wir haben.

$$T(n) \leq c_3 \sum_{j=1}^{n} j + O(n) = c_3 \frac{n(n+1)}{2} + O(n) \in O(n^2)$$

Die worst-case Analyse ist im allgemeinen sinnvoll, weil ...

- ... wir eine Laufzeit für jede beliebige Eingabe garantieren.
- ... der worst-case in vielen Anwendungsfeldern relativ häufig vorkommen kann.
- ... die Analyse der mittleren Laufzeit nicht unbedingt bessere Ergebnisse erzielt.

Beispielsweise können wir bei der Analyse einer mittleren Laufzeit die Frage stellen, an welcher Stelle im Mittel das j-te Element eingeordnet werden muss. Im Mittel ist die Hälfte der Elemente größer als $A[j]$ und die Hälfte der Elemente kleiner als $A[j]$ und somit könnten wir $t_j = \frac{j}{2}$ betrachten. Das führt asymptotisch gesehen exakt zur gleichen Laufzeit.

Anders kann es aussehen, wenn wir aus der Eingabe durch *Würfeln* zufällig gleichverteilt die Elemente aus A ziehen.

Solche *probabilistische Analysen* werden wir später für sogenannte *randomisierte* Algorithmen durchführen. Im Gegensatz dazu haben wir hier zunächst einen *deterministischen* Algorithmus betrachtet. Der Lösungsplan ist von vorneherein vollständig festgelegt und nicht vom Zufall abhängig.

Kapitel 2

Divide-and-Conquer

In diesem Kapitel behandeln wird die Entwurfsmethode *Divide-and-Conquer* (übersetzt: Teile und herrsche) und eine zugehörige allgemeine Analysemethode. Gleichzeitig lernen wir dadurch das Prinzip der *Rekursion* kennen, das heißt wir rufen bestimmte Routinen unseres Algorithmus immer wieder auf, um Teilprobleme zu lösen. Am Ende werden die Lösungen der Teilprobleme zur Lösung des Gesamtproblems zusammengefügt bzw. zusammengemischt.

Das allgemeine Schema der Methode für ein Problem P der Größe n funktioniert wie folgt:

1. Divide	$\begin{cases} n > c & : \text{ Teile das Problem in } k \geq 2 \text{ Teilprobleme} \\ & \quad \text{der Größen } n_1, n_2, \ldots, n_k, \text{ gehe zu 2.} \\ n \leq c & : \text{ Löse das Problem direkt} \end{cases}$
2. Conquer	Löse die k Teilprobleme auf dieselbe Art (rekursiv)
3. Merge	Füge die k berechneten Teillösungen zu einer Gesamtlösung zusammen

2.1 Sortieren von Zahlenketten, Mergesort

Hier betrachten wir $k = 2$ und $n_1, n_2 \approx \frac{n}{2}$. Für eine konkrete Zahlenfolge ergibt sich das folgende Aufrufschema eines Divide-and-Conquer Algorithmus:

Eingabe	:	$(3,2,1,7,3,4,9,2)$
Divide I.	:	$(3,2,1,7)(3,4,9,2)$
(Conquer)Divide II.	:	$(3,2)(1,7)(3,4),(9,2)$
(Conquer)Divide III.	:	$(3)(2)(1)(7)(3)(4)(9)(2)$
Merge III.	:	(2,3)(1,7)(3,4)(2,9)
Merge II.	:	(1,2,3,7)(2,3,4,9)
Merge I.	:	(1,2,2,3,3,4,7,9)

Im Pseudocode und unter Verwendung von Arrays könnte eine Implementierung wie folgt aussehen. Wir verwenden ein nichtleeres Array A von Index 1 bis $n = \text{length}(A)$ und rufen zur vollständigen Sortierung MergeSort$(A, 1, n)$ auf.

Algorithmus 2 MergeSort(A,p,r)
Array A wird zwischen Index p und r sortiert

```
1: if p < r then
2:     q := ⌊(p + r)/2⌋;
3:     MergeSort(A, p, q);
4:     MergeSort(A, q + 1, r);
5:     Merge(A, p, q, r);
6: end if
```

Die eigentliche Arbeit (bis auf die Anzahl der rekursiven Aufrufe) wird im Merge-Schritt vollzogen. Dafür kopieren wir die bereits von Index p bis q und von Index $q+1$ bis r sortierten Abschnitte in zwei Arrays L und R und fügen diese dann wieder gemeinsam sortiert in A von Index p bis r ein. Für ein Abbruchkriterium des Merge-Schrittes fügen wir einen Dummywert ∞ am Ende von L und R ein.

Danach werden die beiden Teilarrays gemischt, indem in einem gleichzeitigen Durchlauf jeweils das kleinste momentan vorderste Element von L und R in A eingefügt wird.

2.1.1 Laufzeitanalyse

Für einen einzelnen Merge-Schritt entsteht offensichtlich ein Aufwand von $O(|L| + |R|)$; siehe Prozedur 3. In jedem Schritt der finalen FOR-Schleife wird ein Element von L oder R nicht mehr betrachtet. Deshalb kann die Schleife nicht mehr als $(|L| + |R|)$ Mal aufgerufen werden.

Zur Laufzeitanalyse verwenden wir eine Rekursionsgleichung. Eine klassische Vereinfachung ist dabei, dass wir annehmen, dass n eine Zeierpotenz ist, also

Prozedur 3 Merge(A,p,q,r)
Array A vorsortiert zwischen Index p und q und Index $q+1$ und r wird zwischen p und r sortiert

```
1: n_1 := q - p + 1; n_2 := r - q;
2: for i = 1 to n_1 do
3:    L[i] := A[p + i - 1];
4: end for
5: for j = 1 to n_2 do
6:    R[j] := A[q + j];
7: end for   // erzeuge Arrays L[1...(n_1 + 1)] und R[1...(n_2 + 1)]
8: L[n_1 + 1] := ∞; R[n_2 + 1] := ∞;
9: i := 1; j := 1;
10: for k = p to r do
11:    if L[i] ≤ R[j] then
12:       A[k] := L[i]; i := i + 1;
13:    else
14:       A[k] := R[j]; j := j + 1;
15:    end if
16: end for
```

$n = 2^k$. Wir können die Kostenfunktion dann für den obigen Algorithmus wie folgt beschreiben.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.1}$$

In der obigen Gleichung haben wir eine einzelne Konstante c verwendet. Das ist prinzipiell erlaubt, da wir für die Konstanten für das Problem der Größe 1 und den Merge-Schritt einfach die größte der beiden Konstanten wählen können. Die asymptotische Laufzeit ändert sich nicht.

Man beachte, dass in den Kosten cn auch das Aufteilen der Aufgabenstellung in zwei Teilprobleme subsumiert ist. Hier musste lediglich ein Index in der Mitte in konstanter Zeit berechnet werden.

Die Rekursionsgleichung läßt sich nun leicht inbesondere auch wegen der Annahme $n = 2^k$ durch einen Rekursionsbaum abschätzen, siehe Abbildung 2.1. Die allerunterste Ebene des Baumes enthält exakt n Aufrufe für die Arrays der Größe 1. In jeder darüberliegenden Ebene werden jeweils zwei Mengen der darunterliegenden Ebene im Merge-Schritt vereinigt. Die Kosten sind jeweils *linear* zur Summe der Größen der darunterliegenden Arrays. Somit sind die Kosten auf jeder Ebene linear zur Gesamtzahl der Elemente überhaupt. Der Baum hat $k = \log_2 n$ viele Ebenen, da sich die Anzahl der Knoten von unten nach oben jeweils halbiert. Wir können also $T(n) = O(n \log n)$ und sogar $T(n) = \Theta(n \log n)$ folgern. Streng genommen hätten wir die Rekursi-

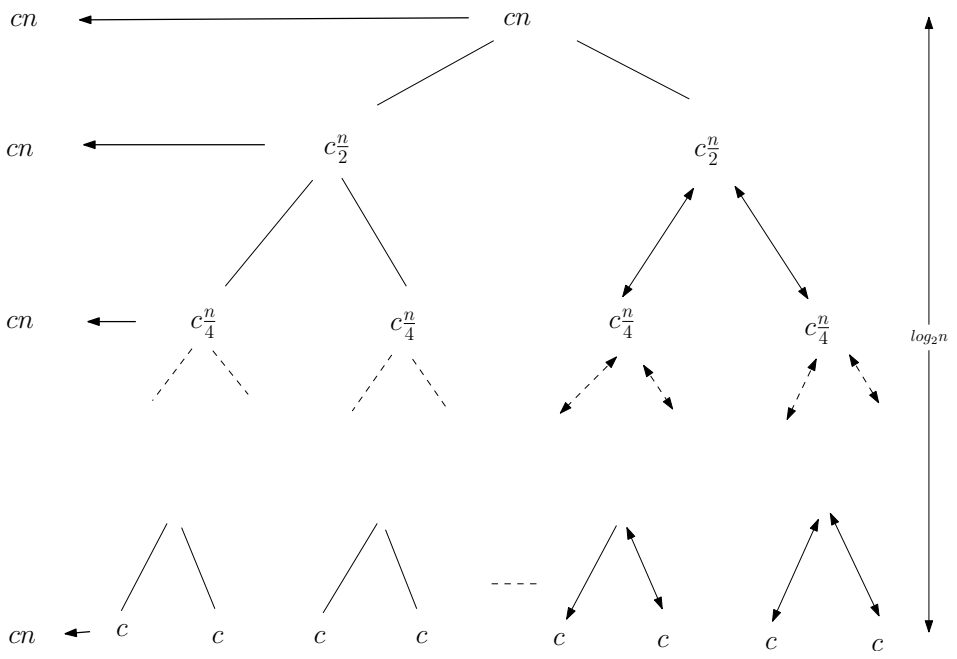

Abbildung 2.1: Nach $k = \log_2 n$ Ebenen sind nur noch einzelne Elemente vorhanden. Auf jeder Ebene des Rekursionsbaumes fällt ein Aufwand von cn für die einzelnen Merge-Schritte an.

onsgleichnung (2.1) wie folgt formulieren müssen.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn & \text{falls } n > 1 \end{cases} \tag{2.2}$$

In jedem Fall müssen wir uns beim Aufstellen und Lösen der Rekursionsformeln Gedanken darüber machen, ob die Einschränkung $n = 2^k$ nicht das asysmptotischen Verhalten der Laufzeit beinflusst. Im obigen Fall ist das offensichtlich nicht der Fall, der zugehörige Rekursionsbaum wäre zwar nicht so ausgeglichen, wie in Abbildung 2.1. Für jedes n könnten wir aber beispielsweise die nächste Zweierpotenz oberhalb von n nehmen, um eine obere Schranke zu erzielen, sowie die nächste Zweierpotenz unterhalb von n für eine untere Schranke der Laufzeit. Der Algorithmus hat dann entsprechend mehr oder entsprechend weniger Aufrufe zu absolvieren. Darüber müssen wir uns aber Gedanken machen.

Beispielsweise kann eine Rekursionsformel der Form

$$T(n) = \begin{cases} n & \text{falls } n \text{ gerade} \\ n^2 & \text{falls } n \text{ ungerade} \end{cases} \tag{2.3}$$

nicht mit dem Ansatz $n = 2^k$ betrachtet werden.

2.1.2 Korrektheit

Die Korrektheit des obigen Divide-and-Conquer Algorithmus ergibt sich direkt aus der Korrektheit der Merge-Prozedur. Sobald diese gemäß ihrer Spezifikation funktioniert, erhalten wir die richtigen Zwischenergebnisse rekursiv für weitere Aufrufe.

Zunächst ist klar, dass die Merge-Prozedur durch die ersten beiden FOR-Schleifen die Einträge des Arrays A von Index p bis Index q und von Index $q + 1$ bis r in die Kopien L und R der Größen $n_1 := q - p + 1$ $n_2 := r - q$ einträgt. Streng genommen müssen auch diese Behauptungen durch eine Schleifeninvariante bewiesen werden.

Wir beschränken uns hier auf die dritte FOR-Schleife und formulieren eine geeignete Schleifeninvariante.

Zu Beginn jeder Iteration der FOR-Schleife enthält das Array $A[p \ldots k-1]$ die $(k-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. $L[i]$ und $R[j]$ sind die kleinsten Einträge der jeweiligen Arrays, die noch nicht in A zurückkopiert wurden.

Induktionsanfang: $k = p$. Das Array $A[p \ldots k-1]$ enthält die $k - p = 0$ kleinsten Elemente von L und R. $L[1]$ und $R[1]$ sind die kleinsten Elemente der Arrays, die noch nicht zurückkopiert wurden.

Induktionsschritt: Wir nehmen an, die Aussage gelte bereits für $k = p+l$ und wollen zeigen, dass die Aussage nach dem Durchlauf der FOR-Schleife für $k' = p+l+1$ erhalten bleibt. Die Schleife wird mit $k = p+l$ ausgeführt. Es werden die aktuellen Köpfe der beiden Arrays L und R verglichen und das kleinste der beiden Elemente als $k = (p+l)$-tes Element in A eingetragen. Also besteht danach $A[p \dots k'-1]$ für $k' = p+l+1$ aus den $(k'-p)$ kleinsten Einträge aus L und R in sortierter Reihenfolge. Je nachdem, ob $L[i] \leq R[j]$ oder $L[i] > R[j]$ gilt, wird i oder j um eins erhöht und die Aussage gilt auch für $k' = p+l+1$ und das neue j oder i.

Terminierung: Beim Abbruch gilt $k = r+1$ und $A[p \dots r]$ enthält die $r+1-p$ kleinsten Elemente von L und R in sortierter Reihenfolge. Zusammen enthalten L und R insgesamt $n_1 + n_2 + 2 = r - p + 3$ Elemente. Also muss sowohl i am Index $n_1 + 1$ und j am Index $n_2 + 1$ angekommen sein. Alle Werte außer den Dummywerten ∞ sind in A eingetragen worden.

2.1.3 Untere Schranke von Sortieralgorithmen

Mergesort kann also n Zahlen $(a_1, a_2, \dots, a_n)$ in Zeit $O(n \log n)$ sortieren. Dabei greift der Algorithmus auf die Elemente der Eingabefolge nur durch *paarweise Vergleiche* zu (siehe Zeile 11 der Merge-Prozedur): Es wird getestet, ob zum Beispiel $a_i < a_j$ gilt, und der Ausgang dieses Tests entscheidet, was der Algorithmus als nächstes macht. "Gerechnet" wird mit den a_i aber nicht.

Mit solchen vergleichsbasierten Sortierverfahren kann man deswegen nicht nur Zahlen sortieren, sondern Objekte aus beliebigen vollständig geordneten Mengen, vorausgesetzt, es steht ein Größenvergleichstest zur Verfügung.

Interessanterweise ist Mergesort sogar optimal.

Theorem 2 *Jeder vergleichsbasierte Sortieralgorithmus benötigt im worst-case $\Omega(n \log n)$ viele Vergleiche, um n Objekte zu sortieren.*

Beweis. Sei A ein vergleichsbasierter Sortieralgorithmus. Wir nehmen der Einfachheit halber an, dass A deterministisch ist; das Theorem gilt aber auch für randomisierte Verfahren. Außerdem setzen wir voraus, dass die a_i paarweise verschieden sind. Weil A den Input $(a_1, a_2, \dots, a_n)$ zunächst nicht kennt, wird stets derselbe Test $a_i < a_j$? als erster ausgeführt. In Abhängigkeit vom Ergebnis verzweigt der Algorithmus. Für alle Eingaben, bei denen $a_i < a_j$ gilt, wird jetzt stets derselbe Test $a_v < a_w$? als nächster ausgeführt. Ebenso kommt für alle Inputs mit $a_i > a_j$ als nächstes stets derselbe Test $a_r < a_s$? an die Reihe, und so fort. So entsteht der Entscheidungsbaum B_A von Algorithmus A, ein Binärbaum, der mindestens so viele Blätter enthält,

wie es Permutationen gibt; denn ein korrekter Sortieralgorithmus darf für unterschiedlich permutierte Eingabefolgen nicht zum selben Ergebnis kommen. Also gilt

$$\begin{aligned} \text{Höhe}(B_A) &\geq \log_2(n!) \geq \log_2\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log_2\left(\frac{n}{2}\right) \\ &\geq \frac{1}{3}n\log_2(n) \end{aligned}$$

für alle $n \geq 8$. Dabei gilt die erste Ungleichung, weil es in $n!$ mindestens $n/2$ viele Faktoren gibt, die mindestens so groß sind wie $n/2$. Es gibt also im Baum B_A mindestens einen Pfad der Länge $\Omega(n \log n)$, und wenn die Eingabe so permutiert ist, wie es den Testergebnissen längs dieses Pfades entspricht, muss Algorithmus A alle diese Vergleiche ausführen. □

Wenn die zu sortierenden Objekte aber Zahlen sind, mit denen man "rechnen" darf, gilt die untere Schranke von Theorem 2 nicht. Zum Beispiel geht der Algorithmus Radixsort folgendermaßen vor, um n k-stellige Zahlen zu sortieren: Zunächst werden leere 10 Listen $L_0, L_1, \ldots, L_9$ angelegt, je eine für jede Ziffer. Nun werden abwechselnd k Verteilungs- und Sammelphasen ausgeführt. Man beginnt mit der letzten der k Stellen, durchläuft die Eingabereihenfolge und hängt die Zahl a_i an diejenige Liste an, die zur letzten Ziffer von a_i gehört. Anschließend werden die Zahlen aus den Listen wieder eingesammelt, in der Reihenfolge $L_0, L_1, \ldots, L_9$ und unter Beibehalt der Reihenfolgen innerhalb der Listen. Mit der neuen Zahlenfolge verfährt man ebenso, verwendet aber nun die vorletzte Stelle, und so fort. Wenn man schließlich die Zahlen nach der ersten Stelle verteilt und wieder eingesammelt hat, sind sie sortiert (!).

Offensichtlich führt Radixsort nur $O(k\,n)$ viele Schritte aus. Wenn die Stellenzahl k als konstant betrachtet wird (was bei Standard-Zahlentypen meist der Fall ist), so ergibt sich die Laufzeit $O(n)$.

2.2 Closest Pair von n Punkten

Wir betrachten die folgende geometrische Aufgabenstellung. Für einer Menge S von n Punkten $p_1, p_2, \ldots, p_n$ in der Ebene, soll das Punktepaar mit kleinstem Euklidischen Abstand $d(\cdot, \cdot)$ berechnet werden.

Wir suchen also den Wert

$$d_S = \min_{1 \leq i,j \leq n, i \neq j} d(p_i, p_j).$$

Der Brute-Force Ansatz für dieses Problem vergleicht systematisch alle $O(n^2)$ vielen Distanzen und benötigt eine Laufzeit von $\Theta(n^2)$.

Für einen Divide-and-Conquer Algorithmus sortieren wir die Punkte zunächst nach X-Koordinaten und teilen die Punktmenge rekursiv in zwei gleichgroße Mengen anhand der X-Koordinaten auf.

1. Divide	Teile das Problem in Teilmengen S_r und S_l gleicher Größe anhand der X-Koordinaten auf.
2. Conquer	Bestimme rekursiv d_{S_l} und d_{S_r}.
3. Merge	Berechne d_S durch die Verwendung von d_{S_l} und d_{S_r}.

Wir wollen im Merge-Schritt wiederum möglichst wenig Aufwand für die Bestimmung von d_S verwenden. Sei $d := \min\{d_{S_l}, d_{S_r}\}$. Wir brauchen für den Merge nur Punkte betrachten die von S_l zu Punkten aus S_r einen Abstand $< d$ haben. Nur die Punkte innerhalb des Streifens der Breite $2d$ symmetrisch entlang der *Splitvertikalen* können einen solchen Abstand $< d$ zueinander haben; siehe Abbildung 2.2. Im Merge-Schritt wird folgendes Verfahren an-

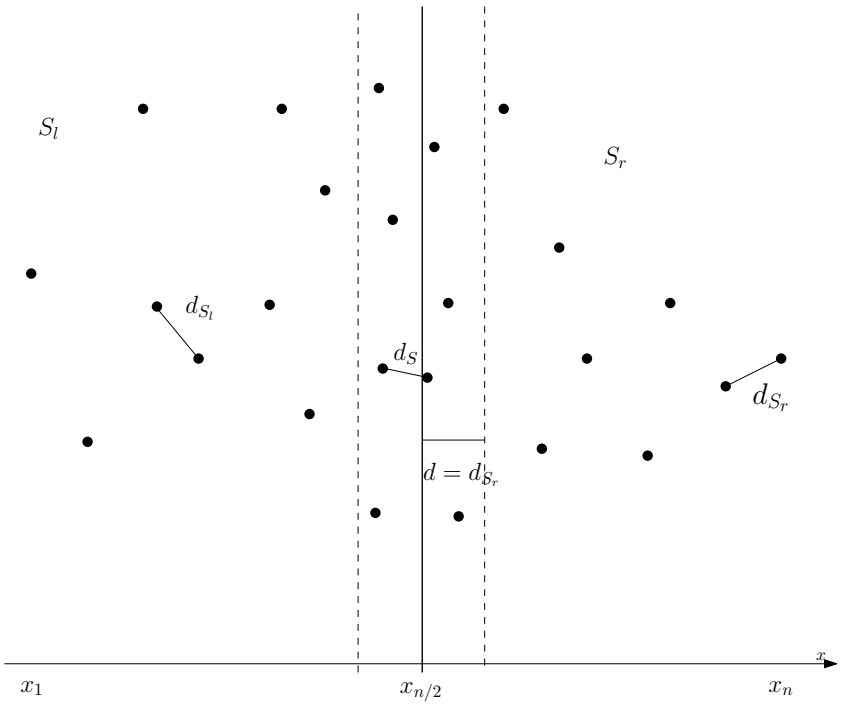

Abbildung 2.2: Für den Merge-Schritt müssen nur die Punkte der beiden Teilstreifen der Breite d entlang der Splitvertikalen betrachtet werden.

gewendet:

- Bestimme die Punkte in den zugehörigen rechten und linken Teilstrei-

fen der Breite d und sortiere diese jeweils nach Y-Koordinaten;

- gehe die Punkte im linken Teilstreifen nach fallenden Y-Koordinaten durch
- für jeden solchen Punkt p betrachte alle Punkte aus dem rechten Teilstreifen deren; Y-Koordinaten im Bereich $[p_y - d, p_y + d]$ liegen
- der zugehörige Punktbereich auf der rechten Seite wandert sukzessive nach *unten*.

Behauptung: Zu jedem Zeitpunkt sind nicht mehr als 10 Punkte im zugehörigen Bereich.

Beweis. (Behauptung) Nehmen wir an, dass p und q ein dichtestes Punktepaar mit Abstand $< d$ sind. Dann können wir oBdA annehmen, dass p im linken und q im rechten Teilstreifen liegt. Der Punkt q kann nicht außerhalb des Rechtecks $R(p)$ der Breite d und Höhe $2d$ im rechten Teilstreifen liegen; siehe Abbildung 2.3. Außerdem haben alle Punkte in $R(p)$ einen Abstand von mindestens d zueinander. Dann sind die Kreisumgebungen $U_{\frac{d}{2}}(q)$ in $R(p)$ disjunkt. Für jeden Punkt q in $R(p)$ ist mindestens ein Viertel dieser Kreisfäche in $R(p)$ enthalten. $R(p)$ kann dann höchstens

$$\frac{2d^2}{\frac{\pi}{4}\left(\frac{d}{2}\right)^2} = \frac{32}{\pi} < 11$$

viele Punkte enthalten. □

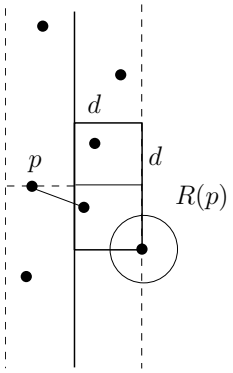

Abbildung 2.3: Für jeden Punkt p des linken Teilstreifens müssen nur konstant viele Punkte im rechten Teilstreifen getestet werden. Die zu testenden Punkte haben Y-Koordinaten, die im Bereich $[p_y - d, p_y + d]$ liegen.

Zur Analyse der Laufzeit betrachten wir nun die Einzelschritte des beschriebenen Algorithmus.

- In einem ersten Schritt werden die Punkte nach X-Koordinaten sortiert. Das kostet $O(n \log n)$ Zeit.
- Der Divide-Schritt kostet dann stets lineare Zeit.
- Für den Merge Schritt ergibt sich folgender Aufwand:
  1. Es müssen die Punkte mit X-Koordinaten aus den Teilstreifen ermittelt werden, das kostet $O(n)$ Zeit.
  2. Danach werden die Punkte in den jeweiligen Bereichen nach Y-Koordinate sortiert. Das kostet $O(n \log n)$ Zeit.
  3. In einem linearen Durchlauf wird für jeden Punkt p zu einer konstanten Anzahl (≤ 10) von Punkten q der Abstand ermittelt und der aktuell kleinste gespeichert. Insgesamt liegt der Aufwand in $O(n)$ für diesen Schritt.

Insgesamt erstellen wir daraus die folgende Rekursionsgleichung.

$$T(n) = \begin{cases} c & \text{falls } n = 1 \\ 2T\left(\frac{n}{2}\right) + c \cdot n \log n & \text{falls } n > 1 \end{cases} \tag{2.4}$$

Wie wir solche Rekursionsgleichungen im Allgemeinen lösen ist Gegenstand des nächsten Kapitels. Eine Übungsaufgabe besteht dann darin, die obige Gleichung abzuschätzen.

Kapitel 3

Lösen von Rekursionsgleichungen

Für die Analyse von Algorithmen, die das Prinzip der Rekursion verwenden, kann häufig eine Rekursionsgleichung aufgestellt werden.

Zunächst erwähnen wir einige Konventionen. Obwohl unsere Kostenfunktionen auf den natürlichen Zahlen definiert sind ignorieren wir diesen Sachverhalt und nehmen an, dass Funktionen wie $T(n)$ zwischen den ganzzahligen Argumenten monoton interpoliert werden. Deshalb können wir zum Beispiel auch für ungerade Zahlen n eine Formel wie $T(\frac{n}{2})$ hinschreiben. Außerdem können wir statt der genauen Laufzeit $U(n)$ eines Algorithmus, die manchmal nur mit umständlichen Fallunterscheidungen beschrieben werden kann, eine obere Schranke $T(n) \geq U(n)$ verwenden. Wenn wir dann eine Rekursionsgleichung für $T(n)$ aufstellen und lösen, haben wir auch eine Abschätzung von $U(n)$ nach oben. Schließlich lassen wir die Abschätzungen für die Anfangswerte von n oft weg, wenn es sich um Konstanten handelt, und konzentrieren uns auf die Rekursionsformel.

Wir wollen nun einige Beispiele für Rekursionsgleichungen betrachten.

Bei der *binären Suche* ist ein aufsteigend sortiertes Array $A = (a_1, a_2, \ldots, a_n)$ von Objekten a_i aus einer vollständig geordneten Menge M gegeben. Es soll festgestellt werden, ob ein Objekt $x \in M$ in diesem Array vorkommt. Zu diesem Zweck wird x zunächst mit dem mittleren Element a_{mitte} von A verglichen, wobei

$$\text{mitte} \;=\; \left\lfloor \frac{1+n}{2} \right\rfloor$$

gewählt wird. Gilt $x = a_{\text{mitte}}$, so hat man x gefunden und ist fertig. Ist $x < a_{\text{mitte}}$, so braucht man die Objekte rechts von a_{mitte} nicht mehr zu betrachten, denn das Array A ist ja aufsteigend sortiert. Man setzt die binäre Suche also rekursiv auf dem linken Teilstück $(a_1, a_2, \ldots, a_{\text{mitte}\,-1})$ von A fort.

Ganz analog wird die binäre Suche auf dem rechten Teil $(a_{\text{mitte}+1}, \ldots, a_n)$ fortgesetzt, falls $x > a_{\text{mitte}}$ gilt. Natürlich endet die Rekursion mit der Meldung "nicht gefunden", sobald das verbleibende Array keine Objekte mehr enthält.

Egal, ob n gerade oder ungerade ist, das verbleibende Array hat höchstens die Länge $\frac{n}{2}$. Deshalb erhalten wir als Laufzeitabschätzung für binäres Suchen die Rekursionsgleichung

$$T(n) = T\left(\frac{n}{2}\right) + c, \tag{3.1}$$

wobei die Konstante c den Aufwand für die Berechnung von a_{mitte} und den Vergleich mit x abschätzt.

Für das Sortierverfahren *Mergesort* hatten wir in Kapitel 2.1 schon die Formel

$$T(n) = 2\,T\left(\frac{n}{2}\right) + c\,n \tag{3.2}$$

aufgestellt, während wir für den Divide-and-Conquer Algorithmus für das Closest-Pair-Problem in Kapitel 2.2 die Gleichung

$$T(n) = 2\,T\left(\frac{n}{2}\right) + c\,n\log n \tag{3.3}$$

erhalten hatten.

Eine wichtige und immer wieder auftretende Aufgabe ist die *Multiplikation von Matrizen.* Berechnet man bei der Multiplikation

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{n1} & \cdots & c_{nn} \end{pmatrix}$$

nach der Schulmethode alle Koeffizienten $c_{i,j}$ einzeln nach der Formel

$$c_{i,j} \;=\; \sum_{k=1}^{n} a_{ik}\, b_{kj},$$

so ergeben sich n^3 viele Multiplikationen von Zahlen. Ob es mit Divide-and-Conquer schneller geht?

Für gerade Zahlen n können wir die Matrizen durch jeweils vier Blockmatrizen der Größe $\frac{n}{2} \times \frac{n}{2}$ darstellen:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Jede der vier Blockmatrizen C_{ij} ergibt sich wegen

$$C_{ij} \;=\; A_{i1} \cdot B_{1j} \;+\; A_{i2} \cdot B_{2j}$$

durch Multiplikation von zwei Matrizen der Größe $\frac{n}{2} \times \frac{n}{2}$. Also erhalten wir die Rekursionsgleichung

$$T(n) = 8\,T\left(\frac{n}{2}\right) + c\,n^2, \tag{3.4}$$

bei der $c\,n^2$ die Kosten für die insgesamt vier Additionen von $\frac{n}{2} \times \frac{n}{2}$ - Matrizen abschätzt. *V. Strassen* entdeckte 1969 eine Möglichkeit, eine der acht Matrixmultiplikationen einzusparen. Er definierte sieben Hilfsmatrizen

$$\begin{aligned}
M_1 &:= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
M_2 &:= (A_{21} + A_{22}) \cdot B_{11} \\
M_3 &:= A_{11} \cdot (B_{12} - B_{22}) \\
M_4 &:= A_{22} \cdot (B_{21} - B_{11}) \\
M_5 &:= (A_{11} + A_{12}) \cdot B_{22} \\
M_6 &:= (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) \\
M_7 &:= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}),
\end{aligned}$$

deren Berechnung jeweils eine Multiplikation von zwei $\frac{n}{2} \times \frac{n}{2}$ - Matrizen erfordert, und konnte nun die C_{ij} folgendermaßen daraus gewinnen:

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6.
\end{aligned}$$

Bei diesem Ansatz werden also in der Rekursion nur 7 Multiplikationen benötigt, allerdings 18 Additionen und Subtraktionen. Somit ergibt sich die Rekursionsgleichung

$$T(n) = 7\,T\left(\frac{n}{2}\right) + c\,n^2, \tag{3.5}$$

mit einer etwas größeren Konstante c als in (3.4).

Die Beispiele (3.1) bis (3.5) sind Spezialfälle der allgemeinen Rekursionsgleichung

$$T(n) \;=\; a\,T\left(\frac{n}{b}\right) \;+\; f(n). \tag{3.6}$$

Die Konstante a beschreibt die Anzahl der Teile, in die das Problem zerlegt wird, und $\frac{n}{b}$ die Größe der Teilprobleme. Die Funktion $f(n)$ gibt den Gesamtaufwand beim Teilen und Zusammensetzen eines Problems der Größe n

an. Durch iterierte Substitution ergibt sich

$$\begin{aligned} T(n) &= a\,T\left(\frac{n}{b}\right) + f(n) \\ &= a\left(a\,T\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) \\ &= a^2\,T\left(\frac{n}{b^2}\right) + a\,f\left(\frac{n}{b}\right) + f(n) \\ &= a^2\left(a\,T\left(\frac{n}{b^3}\right) + f\left(\frac{n}{b^2}\right)\right) + a\,f\left(\frac{n}{b}\right) + f(n) \\ &= a^3\,T\left(\frac{n}{b^3}\right) + a^2\,f\left(\frac{n}{b^2}\right) + a\,f\left(\frac{n}{b}\right) + f(n), \end{aligned}$$

und nun errät man leicht, dass in der i-ten Substitution

$$T(n) \;=\; a^i\,T\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j\,f\left(\frac{n}{b^j}\right)$$

herauskommt. Strenggenommen muss man diese Gleichung durch vollständige Induktion über i beweisen; der Beweis ist aber so einfach, dass wir ihn weglassen. Nun nehmen wir an, dass $n = b^k$ eine Potenz von b ist und setzen $i = k$ ein. Wegen

$$a^k \;=\; a^{\log_b n} \;=\; n^{\log_b a}$$

ergibt sich

$$T(n) \;=\; n^{\log_b a}\,T(1) + \sum_{j=0}^{\log_b n-1} a^j\,f\left(\frac{n}{b^j}\right). \tag{3.7}$$

Mit dieser Formel können wir nun die Rekursionsgleichungen lösen, die wir oben aufgestellt hatten.

Beim *binären Suchen* (3.1) wird in Formel (3.7) $a = 1, b = 2$ und $f(n) = c$ gesetzt, und wir erhalten

$$T(n) \;=\; T(1) + \sum_{j=0}^{\log n-1} c \;\;\in\; \Theta(\log n),$$

wobei $\log n$ stets $\log_2 n$ bedeutet. Für das Sortierverfahren *Mergesort* (3.2) wird in (3.7) $a = b = 2$ und $f(n) = cn$ gesetzt, und es ergibt sich

$$T(n) \;=\; n\,T(1) + \sum_{j=0}^{\log n-1} 2^j\,c\frac{n}{2^j} \;\;\in\; \Theta(n \log n),$$

was wir auch Betrachtung des Rekursionsbaums schon herausgefunden hatten. Beim *Closest-Pair-Problem* (3.3) ist ebenfalls $a = b = 2$ aber $f(n) =$

$cn \log n$. Hier erhalten wir

$$\begin{aligned} T(n) &= n\,T(1) + \sum_{j=0}^{\log n-1} 2^j\, c\frac{n}{2^j} \log\left(\frac{n}{2^j}\right) \\ &= n\,T(1) + cn\left(\log^2 n - \frac{(\log n - 1)\log n}{2}\right) \in \Theta(n \log^2 n), \end{aligned}$$

wobei wir die Formel $\sum_{j=0}^{m} j = \frac{m(m+1)}{2}$ für $m = \log n - 1$ benutzt haben. Wir werden aber später sehen, dass das Problem *Closest Pair* sich noch effizienter lösen lässt.

Betrachten wir abschließend die *Matrixmultiplikation*. Zur Lösung der Rekursion (3.4) setzen wir in Formel (3.7) die Werte $a = 8, b = 2$ und $f(n) = cn^2$ ein und bekommen

$$\begin{aligned} T(n) &= n^3\,T(1) + \sum_{j=0}^{\log n-1} 8^j\, c\left(\frac{n}{2^j}\right)^2 \\ &= n^3\,T(1) + cn^2 \sum_{j=0}^{\log n-1} 2^j \in \Theta(n^3), \end{aligned}$$

denn nach der Formel $\sum_{j=0}^{m} q^j = \frac{q^{m+1}-1}{q-1}$ ist der Wert der Summe in $\Theta(n)$. Die (naive) Rekursion auf 8 Produkte von Teilmatrizen ist also nicht besser als die Schulmethode. Verwendet man aber den Ansatz von Strassen (3.5), so hat a nur den Wert 7, und es folgt aus (3.7)

$$\begin{aligned} T(n) &= n^{\log 7}\,T(1) + \sum_{j=0}^{\log n-1} 7^j\, c\left(\frac{n}{2^j}\right)^2 \\ &= n^{\log 7}\,T(1) + cn^2 \sum_{j=0}^{\log n-1} \left(\frac{7}{4}\right)^j \in \Theta(n^{2.807\ldots}), \end{aligned}$$

denn es ist ja $\left(\frac{7}{4}\right)^{\log n} = \frac{n^{\log 7}}{n^2}$ und $\log 7 = 2.8073549\ldots$. Dieses Verfahren ist also deutlich schneller als die Schulmethode. Bis heute ist nicht bekannt, was der kleinstmögliche Exponent ω ist, für den zwei $n \times n$ - Matrizen sich in Zeit $O(n^\omega)$ multiplizieren lassen. Der Rekord liegt gegenwärtig bei $\omega = 2.37\ldots$.

Man hätte die Lösungen der obigen Rekursionsgleichungen auch durch Untersuchung der Rekursionsbäume erhalten können, wie anfangs am Beispiel von Mergesort vorgeführt. Welchem Verfahren man den Vorzug gibt, mag vom Geschmack abhängen; die Rekursionsbäume sind recht anschaulich, die Substitutionsmethode ist mathematisch präziser.

In der Literatur findet sich eine Diskussion der Formel (3.7) unter der Bezeichnung "Mastertheorem". Zum Beweis verwendet man ganz ähnliche Argumente wie wir sie gerade benutzt haben; siehe z.B. [1].

Theorem 3 *Seien $a \geq 1$ und $b > 1$ Konstanten und $f(n)$ eine nichtnegative Funktion. Dann gilt für die Rekursionsgleichung (3.6):*

1. *Wenn $f(n) \in O\left(n^{\log_b a - \epsilon}\right)$ für $\epsilon > 0$, dann gilt $T(n) \in \Theta\left(n^{\log_b a}\right)$.*
2. *Wenn $f(n) \in \Theta\left(n^{\log_b a}\right)$, dann gilt $T(n) \in \Theta\left(n^{\log_b a} \log n\right)$.*
3. *Wenn $f(n) \in \Omega\left(n^{\log_b a + \epsilon}\right)$ für $\epsilon > 0$ und $a\ f(n/b) \leq c \cdot f(n)$ für $c < 1$ und für hinreichend große n, dann gilt $T(n) \in \Theta\left(f(n)\right)$.*

3.1 Typische Schwierigkeiten

Bei der Lösung von Rekursionsgleichungen können u.a. typischerweise die folgenden Probleme auftreten:

1. Der Induktionsbeweis ist aufgrund der Konstanten fehlerhaft.
2. Die Lösungsform ist zu ungenau.
3. Die Lösungsform erfordert eine Variablentransformation.

Wir geben hier Beispiele für die obigen Probleme an.

Korrekte Konstanten: Für

$$T(n) \quad = \quad 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

könnten wir

$$\begin{aligned} T(n) &\leq \left(c \left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq c \cdot n + n \\ &= O(n) \qquad \text{Falsch!!} \end{aligned}$$

folgern, da c eine Konstante ist. Der Fehler liegt darin, dass wir nicht exakt $T(n) \leq cn$ bewiesen haben. Wir müssen die Konstante genau angeben.

Ein ganz ähnliches Problem kann auftreten, wenn man die O - Notation leichtfertig zur Behandlung von Summen nicht-konstanter Länge verwendet. Sind zum Beispiel alle Funktionen $f_i(n)$ in $O(g(n))$, so folgt daraus im allgemeinen *nicht*, dass die Funktion $F(n) = \sum_{i=0}^{n} f_i(n)$ in $O(ng(n))$ liegt. Beispiel: Alle Funktionen $f_i(n) = i$ sind in $O(1)$, aber $F(n)$ ist quadratisch und nicht linear.

Lösungsform ist zu ungenau: Für

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 1 & \text{falls } n > 1 \end{cases}$$

vermuten wir $T(n) \leq cn$. Die Induktionsannahme ist für $c > 1$ erfüllt. Leider führt

$$\begin{aligned} T(n) &\leq c\left\lfloor \frac{n}{2} \right\rfloor + c\left\lceil \frac{n}{2} \right\rceil + 1 \\ &= cn + 1 > cn \end{aligned}$$

im Induktionsschluss durch Substitution nicht zum Ziel. Wir müssen hier $T(n) \leq cn - 1$ wählen und dann für die Induktionsannahme $c > 2$.

Lösung durch Variablentransformation:

Für

$$T(n) = \begin{cases} 1 & \text{falls } n = 1 \\ 2T\left(\lfloor \sqrt{n} \rfloor\right) + \log n & \text{falls } n > 1 \end{cases}$$

können wir eine geeignete Umformung vornehmen. Durch $m = \log n$ gelangen wir vereinfacht zu

$T(2^m) = 2T\left(2^{m/2}\right) + m$ und erhalten durch Umbenennung die Gleichung $S(m) = 2S(m/2) + m$. Da, wie bereits gezeigt $S(m) = O(m \log m)$ gilt erhalten wir

$$T(n) = T(2^m) = S(m) \in O(m \log m) \in O(\log n \log \log n).$$

Kapitel 4

Dynamische Programmierung

Das Prinzip der dynamischen Programmierung kann dann angewendet werden, wenn eine rekursive Problemzerlegung an vielen Stellen die gleiche Teillösung verlangt. In diesem Fall ist es ratsam, die Lösungen dieser Teilprobleme zu speichern und gegebenenfalls darauf zurückzugreifen.

Falls die Teillösungen immer wieder neu berechnet werden, kann der Rechenaufwand exponentiell steigen. Grundsätzlich besteht die dynamische Programmierung aus einem rekursiven Aufruf für Teilprobleme und aus einer Tabellierung von Zwischenergebnissen die, wiederverwendet werden sollen.

Wir betrachten zunächst ein einfaches klassisches Beispiel, das diese wesentlichen Bestandteile bereits charakterisiert. Generell werden durch die dynamischen Programmierung meistens Optimierungsprobleme gelöst. In solche Fällen wird dann versucht, zu jedem Teilproblem eine optimale Lösung zu finden.

Dynamische Programmierung ist dann sinnvoll, wenn das sogenannte *Optimalitätsprinzip von Bellman* vorliegt:

Eine optimale Lösung hat stets eine Zerlegung in optimale Teilprobleme.

Ein solches Beispiel wird in Abschnitt 4.2 vorgestellt.

4.1 Fibonacci Zahlen

Wir betrachten die Fibonacci-Zahlen, die rekursiv wie folgt definiert werden.

$$\begin{aligned} \text{fib}(0) &= 1 && (1)\\ \text{fib}(1) &= 1 && (2)\\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) && (3) \end{aligned}$$

Wollen wir nun beispielsweise fib(5) berechnen, so ergibt sich folgender Rekursionsbaum für die Aufrufe von fib; siehe Abbildung 4.1(I). Dieser Baum

hat für fib(n) eine Größe von $\Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$, wie eine Übungsaufgabe zeigt.

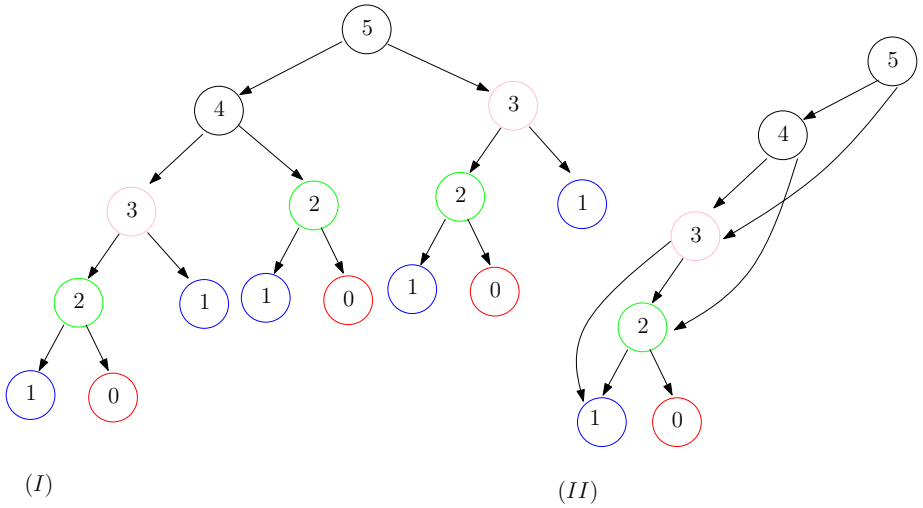

Abbildung 4.1: (I) Der vollständige Rekursionsbaum für fib(5) enthält einige Redundanzen. (II) Die rekursiven Abhängigkeiten können einfach dargestellt werden.

Offensichtlich ist es ratsam, einige der Ergebnisse zu speichern, die rekursiven Abhängigkeiten können einfacher dargestellt werden, wie in Abbildung 4.1(II). Wir wollen zum Beispiel fib(2) nicht dreimal wieder aus den Vorgängern neu berechnen, deshalb speichern wir die Werte ab. Es ist hier so, dass sich Teilprobleme überlappen und diese effizient gespeichert werden sollen. Der Abhängigkeitsgraph in Abbildung 4.1(II) hat nur eine Größe von $\Theta(n)$. Das Prinzip der Speicherung von Zwischenergebnissen wird auch *Memoisation* genannt.

Wenn wir nun die Abhängigkeiten nutzen wollen und von oben nach unten (Top-Down) die benötigten Werte fib(i) bestimmen wollen, könnten wir algorithmisch wie in Algorithmus 4 vorgehen, wobei FibMemo die memoisierende Prozedur 5 ist.

Algorithmus 4 FibTopDown(n)

```
1: F[1] := 1; F[0] := 1;
2: for i = 2 to n do
3:     F[i] := 0;
4: end for
5: RETURN FibMemo(n, F)
```

Hierbei ist FibMemo die folgende memoisierende Prozedur.

Prozedur 5 FibMemo(n,F)

1: **if** $F[n] > 0$ **then**
2: **RETURN** $F[n]$
3: **end if**
4: $F[n] :=$ FibMemo$(n-1, F)$ + FibMemo$(n-2, F)$;
5: **RETURN** $F[n]$

In Algorithmus 4 wird das Feld F zunächst mit Nullen initialisiert, und dann beginnt den Aufruf der rekursiven Prozedur 5. Falls der Wert von $F[n]$ bekannt ist, geben wir diesen hier zurück, sonst wird er rekursiv gemäß der Rekursionsformel berechnet.

Laufzeitabschätzung: Wir benötigen im Algorithmus 4 $O(n)$ Zeit für die Initialisierung. Danach wird FibMemo(n, F) aufgerufen. Jeder Aufruf von FibMemo(m, F) in Prozedur 5 generiert jeden der Aufrufe FibMemo$(m-1, F)$ und FibMemo$(m-2, F)$ höchstens einmal. Für jedes $l < n$ wird somit FibMemo(l, F) höchstens zweimal aufgerufen. Ohne die rekursiven Aufrufe wird nur $O(1)$ Zeit in Prozedur 5 benötigt. Insgesamt gilt für die Laufzeit $T(n) \in O(n)$.

Wir machen nun die folgende Beobachtung. Obwohl im obigen Algorithmus $F[n]$ in einer Top-Down Vorgehensweise errechnet wird, werden die eigentlichen Werte von unten nach oben berechnet. Erst wenn die Rekursion bei $F[1]$ und $F[0]$ angekommen ist, werden neue Felder belegt und die Memoisation setzt ein. Wenn wir beispielsweise $F[5]$ berechnen, rufen wir nacheinander FibMemo$(5, F)$, FibMemo$(4, F)$, FibMemo$(3, F)$, FibMemo$(2, F)$ und FibMemo$(1, F)$ auf bis hier zum ersten Mal ein Wert $F[1]$ zurückgegeben wird. Dann wird wieder FibMemo$(0, F)$ aufgerufen und $F[2]$ belegt. Dann wird FibMemo$(1, F)$ nochmal aufgerufen und $F[3]$ belegt. Danach rufen wir FibMemo$(2, F)$ auf und belegen $F[4]$. Dann FibMemo$(3, F)$ und $F[5]$ wird belegt und ausgegeben.

Deshalb ist es offensichtlich sinnvoller gleich die Werte von unten nach oben (Bottom-Up) zu berechnen. Da für die Berechnung von $F[n]$ nur zwei Werte aus $F[0], \ldots, F[n-1]$ benötigt werden, sparen wir außerdem Speicherplatz. Die obige Memoisation und die Top-Down vorgehensweise ist hier bezüglich des Speicherplatzes nicht effizient.

Insgesamt erhalten wir folgende effiziente dynamische Programmierungslösung.

- Es müssen insgesamt nur $\Theta(n)$ viele Berechnungen durchgeführt werden.
- Die Bottom-Up Berechnung stellt sicher, dass die benötigten Werte stets vorliegen.

Algorithmus 6 FibDynProg(n)

1: $F_{\text{letzt}} := 1$; $F_{\text{vorletzt}} := 1$; $F := 1$;
2: **for** $i = 2$ to n **do**
3: $\quad F := F_{\text{vorletzt}} + F_{\text{letzt}}$;
4: $\quad F_{\text{vorletzt}} := F_{\text{letzt}}$; $F_{\text{letzt}} := F$;
5: **end for**
6: **RETURN** F

- Der Speicherplatzbedarf liegt in $O(1)$.

Beachte: Manchmal kann eine Top-Down Vorgehensweise auch sinnvoller sein, wenn zum Beispiel gar nicht alle Werte verwendet werden, die wir Bottom-Up berechnen würden. Die Prinzipien sind also nicht *dogmatisch* zu interpretieren. Es kann auch sinnvoll sein, die Zwischenergebnisse für mehrfache Aufrufe des Algorithmus hintereinander zu speichern.

Grundprinzip der dynamischen Programmierung:

- Formuliere das Problem insgesamt rekursiv. Stelle fest, dass sich Teillösungen überlappen und rekursiv voneinander abhängen.
- Stelle eine Rekursionsgleichnung (Top-Down) auf.
- Löse das Problem (Bottom-Up) durch das Ausfüllen von Tabellen mit Teillösungen (Memoisation).
- Ziel: Die Zahl der Tabelleneinträge soll klein gehalten werden.
- Anwendung: Bei einer direkten Ausführung der Rekursion ist mit vielen Mehrfachberechnungen zu rechnen.

Die Korrektheit der dynamischen Programmierung ergibt sich in der Regel dadurch, dass die rekursive Problembeschreibung korrekt durchgeführt wurde. Die Laufzeitanalyse ist in der Regel nicht schwer, da die Algorithmen typischerweise aus geschachtelten FOR-Schleifen bestehen. Das eigentliche Problem der dynamischen Programmierung ist es, eine geeignete rekursive Formulierung des Problems zu finden. Wir werden uns deshalb ein signifikantes Optimierungsbeispiel ansehen in dem auch das *Optimalitätsprinzip von Bellman* zum tragen kommt.

4.2 Matrixmultiplikation

Wir betrachten das Multiplizieren von reellwertigen Matrizen. Seien $A \in \mathbb{R}^{i \times j}$ und $B \in \mathbb{R}^{j \times k}$ Matrizen mit i Zeilen und j Spalten bzw. j Zeilen und

k Spalten, dann beschreibt $A \cdot B$ die Multiplikation dieser beiden Matrizen. Insgesamt sind zur Berechnung von $A \cdot B = C \in \mathbb{R}^{i \times k}$ $i \cdot k \cdot j$ Floating-Point Berechnungen notwendig. Genauer: Die Matrixeinträge von C werden durch

$$c_{st} = \sum_{l=1}^{j} a_{sl} \cdot b_{lt}$$

für alle $s = 1, \ldots i$ und alle $t = 1, \ldots, k$ berechnet, also $i \cdot j \cdot k$ Multiplikationen und Additionen.

Falls wir nun mehrere Matrizen multiplizieren wollen, also $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ dann müssen auch hier die entsprechenden Matrizen gemäß der Größen *zusammenpassen*. Die Anzahl der Spalten von A_i muss der Anzahl der Zeilen von A_{i+1} entsprechen für $i = 1, \ldots, n-1$.

Die Matrixmultiplikation ist *assoziativ*, für drei Matrizen $A \in \mathbb{R}^{i \times j}$, $A \in \mathbb{R}^{j \times k}$ und $C \in \mathbb{R}^{k \times l}$ gilt also $(A \cdot B) \cdot C = A \cdot (B \cdot C) \in \mathbb{R}^{i \times l}$. Deshalb lassen sich auch allgemeine (zusammenpassende) Ketten beliebig Klammern. Diese Klammerung hat Einfluss auf die Anzahl der Floating-Point Operationen.

Beispiel: Für $A \in \mathbb{R}^{10 \times 50}$, $B \in \mathbb{R}^{50 \times 10}$ und $C \in \mathbb{R}^{10 \times 50}$, benötigt $(A \cdot B) \cdot C$ zuerst $10 \times 50 \times 10$ Operationen für $(A \cdot B)$ und danach $10 \times 10 \times 50$ Operationen für die Multiplikation des Ergebnisses mit C also insgesamt 10000 Operationen. Dahingegen benötigt $A \cdot (B \cdot C)$ zuerst $50 \times 10 \times 50$ Operationen für $(B \cdot C)$ und danach $10 \times 50 \times 50$ Operationen für die Multiplikation des Ergebnisses mit A also insgesamt 50000 Operationen. Die Klammerung $(A \cdot B) \cdot C$ ist also deutlich günstiger.

Allgemeine Problembeschreibung: Bestimme für die Berechnung von $E = A_1 \cdot A_2 \cdot \ldots \cdot A_n$ mit Dimensionen $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$ die optimale Klammerung für die minimale Anzahl an Operationen.

Dieses Problem wollen wir durch dynamische Programmierung lösen. Deshalb muss zunächst die Lösung rekursiv durch Teillösungen angegeben werden. Falls $M(n)$ alle Möglichkeiten beschreibt, wie man n Matrizen klammert, dann ist $M(n) = \sum_{k=1}^{n-1} M(k) \cdot M(n-k)$ mit $M(1) = 1$. Diese Rekursionsgleichung liegt in $\Omega(2^n)$. Es ist also nicht zweckmäßig, alle Möglichkeiten auszuprobieren.

Wir wollen zunächst herausfinden, wie groß die Anzahl $M(n)$ der Klammerungen von n Faktoren ist, und dabei auch unsere Kenntnisse über die Lösung von Rekursionen erweitern. Ein Trick besteht darin, die unbekannten Zahlen $M(n)$ als Koeffizienten einer formalen Potenzreihe

$$f(X) \;=\; \sum_{n=0}^{\infty} M(n)\, X^n$$

anzusehen und aus der Rekursionsgleichung für die $M(n)$ eine algebraische Gleichung für die sogenannte *erzeugende Funktion* $f(X)$ zu gewinnen. Der

Bequemlichkeit halber setzen wir $M(0) := 0$ und erhalten damit für alle $n \geq 2$ die Rekursionsgleichung

$$M(n) = \sum_{k=0}^{n} M(k)\, M(n-k).$$

Genau solche Summen (Faltungen oder Cauchy-Produkte genannt) treten als Koeffizienten von X^n auf, wenn man die Reihe $f(X)$ quadriert:

$$\begin{aligned} f(X)^2 &= \sum_{n=0}^{\infty} \Big(\sum_{k=0}^{n} M(k)\, M(n-k) \Big) X^n && (4.1) \\ &= 0 \cdot X^0 + 0 \cdot X^1 + \sum_{n=2}^{\infty} M(n)\, X^n && (4.2) \\ &= f(X) - 1 \cdot X^1. && (4.3) \end{aligned}$$

Die Summation in Gleichung (4.2) darf erst ab $n = 2$ beginnen, denn erst ab da gilt die Rekursionsgleichung. Für $n = 0$ und $n = 1$ hat die innere Summe in (4.1) den Wert 0 wegen $M(0) = 0$.

Lösen der quadratischen Gleichung

$$f(X)^2 - f(X) + X = 0$$

ergibt

$$f(X) = \frac{1 \pm \sqrt{1 - 4X}}{2}, \tag{4.4}$$

und die Wurzel hat die Reihendarstellung

$$\sqrt{1 - 4X} = \sum_{n=0}^{\infty} \binom{2n}{n} \frac{1}{1 - 2n} X^n; \tag{4.5}$$

man kommt darauf, indem man zum Beispiel in der Taylorreihenentwicklung für $\sqrt{1 + X}$ die Variable X durch $-4X$ ersetzt. Der Nenner in (4.5) wird für $n \geq 1$ negativ. Deshalb muss in (4.4) das Minuszeichen verwendet werden, und man erhält

$$\begin{aligned} f(X) &= \frac{1}{2} - \frac{1}{2} \sum_{n=0}^{\infty} \binom{2n}{n} \frac{1}{1 - 2n} X^n \\ &= \frac{1}{2} \sum_{n=1}^{\infty} \binom{2n}{n} \frac{1}{2n - 1} X^n, \end{aligned}$$

weil die Summe für $n = 0$ den Koeffizienten 1 hat. Durch Koeffizientenvergleich ergibt sich jetzt

$$M(n) = \frac{1}{2} \binom{2n}{n} \frac{1}{2n - 1}.$$

Zum Beispiel kommt für $n = 4$ der Wert $M(4) = 5$ heraus, und tatsächlich gibt es genau 5 Arten, um 4 Faktoren zu beklammern, nämlich

$$((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd)).$$

Die Zahl $C_{n-1} := M(n)$ wird die $n-1$-te *Catalan'sche Zahl* genannt. Um ihre Größe abzuschätzen, können wir die *Stirling-Formel*

$$n! \sim \sqrt{2\pi n}\, \frac{n^n}{e^n}$$

als asymptotische Näherung für die Fakultät anwenden (was bedeutet, dass der Quotient der beiden Seiten gegen 1 konvergiert,) und erhalten

$$\binom{2n}{n} \sim \sqrt{4\pi n}\, \frac{(2n)^{2n}}{e^{2n}}\, \frac{1}{2\pi n}\, \frac{(e^n)^2}{(n^n)^2} \quad \in \quad \Theta\left(\frac{2^{2n}}{\sqrt{n}}\right).$$

Also ist $M(n) \in \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$ eine exponentiell wachsende Größe.

Wir wollen nun optimale Teillösungen wiederverwenden. Sei nun $K(l,k)$ der minimale Aufwand bzw. die optimale Klammerung für $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$. Zum Beispiel zerlegt sich zu Beginn ein erster Multiplikationsschritt für die Stelle s, d.h. der minimale Aufwand für $(A_1 \cdot A_2 \cdot \cdots \cdot A_s) \cdot (A_{s+1} \cdot A_{s+2} \cdot \cdots \cdot A_n)$, rekursiv in die Kosten

$$K(1,s) + K(s+1,n) + d_0 \cdot d_s \cdot d_n \tag{4.6}$$

Das Ziel ist nun, das optimale s zu finden, und das wollen wir rekursiv beschreiben.

- Offensichtlich ist $K(l,l) = 0$ für $l = 1, \ldots, n$
- Die Dimension eine Teilkette $A_l \cdot A_{l+1} \cdot \cdots \cdot A_k$ ist $d_{l-1} \times d_k$.
- Die Kosten für ein Teilen bei s sind $K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k)$

Da wir $K(1,n)$ minimieren wollen, erhalten wir insgesamt:

$$K(l,k) = \begin{cases} 0 & \text{falls } l = k \\ \min_{l \le s \le k-1} K(l,s) + d_{l-1} \cdot d_s \cdot d_k + K(s+1,k) & \text{falls } l < k \end{cases}. \tag{4.7}$$

Wir stellen fest, dass dynamische Programmierung geeignet ist:

- Wiederverwendung: Hier werden viele Teilprobleme mehrfach benutzt, zum Beispiel $K(1,2)$ wird verwendet von $K(1,3), K(1,4), \ldots, K(1,n)$.

- Beschränkte Anzahl: Für alle $1 \leq l < k \leq n$ gibt es genau ein Teilproblem, also insgesamt $\Theta(n^2)$ viele.
- Bearbeitungsreihenfolge: Falls alle Werte $K(l, s)$ und $K(s+1, k)$ bekannt sind, kann $K(l, k)$ in Zeit $O(k-l) \in O(n)$ berechnet werden.

Die geschickte Auswertung läßt sich sehr gut durch eine Matrix beschreiben. Wir verwenden ein Array $K[1..n, 1..n]$.

Zuerst werden die Diagonalen eingetragen, dann berechnen wir sukzessive die Nebendiagonalen von rechts unten nach links oben, zu jedem Zeitpunkt sind für $K(l, k)$ alle Werte $K(l, s)$ und alle Werte $K(s+1, k)$ für $l \leq s$ und $s+1 \leq k$ berechnet worden; siehe Abbildung 4.2.

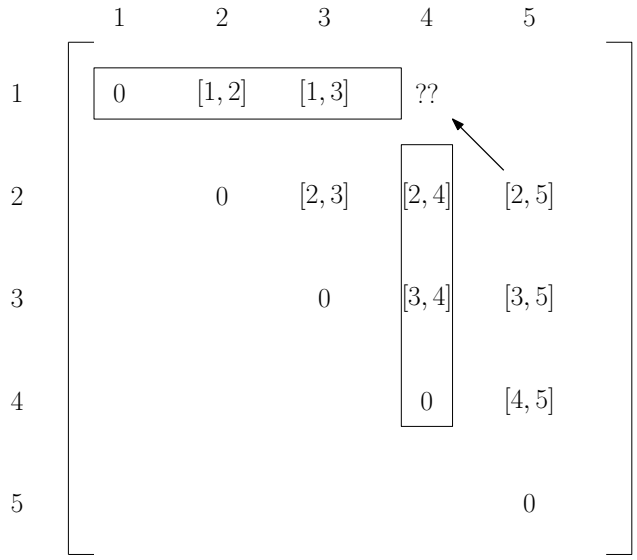

Abbildung 4.2: Die Nebendiagonalen können von unten nach oben berechnet werden. Nachdem zwei Nebendiagonalen bereits berechnet wurden beginnen wir die Berechnung der dritten Nebendiagonalen. Die benötigten Werte liegen bereits vor.

Korrektheit: Am Ende der Auswertung wird der Wert $[1, n]$ die minimale Anzahl an Operationen enthalten, wir haben den Wert rekursiv genauso festgelegt. Wir wollen uns natürlich auch die optimale Klammerung merken. Dazu müssen wir uns nur für jeden berechneten Matrixeintrag den jeweils optimalen Index $s[l, k]$ merken. Falls beispielsweise der optimale Index für die Berechnung von $[1, 5]$ bei $s = 2$ liegt, ist die Klammerung $(A_1A_2)(A_3A_4A_5)$ rekursiv die beste und wir speichern den Wert $s[1, 5] = 2$. Die optimale Klammerung von (A_1A_2) zur Berechnung von $[1, 2]$ ist trivialerweise $s = 1$ also $s[1, 2] = 1$. Rekursiv haben wir auch die optimale Klammerung aus

$[3,5]$ gespeichert, diese kann $s=3$ oder $s=4$ sein, somit ist $s[3,5] \in \{3,4\}$. Insgesamt erhalten wir so die optimale Klammerung.

Laufzeitanalyse und Speicherplatz: Insgesamt wird Speicherplatz von $O(n^2)$ für beide Matrizen (Einträge $K[l,k]$ und $s[l,k]$) verwendet. Für die Berechnung eines Eintrages $K[l,k]$ und der Klammerung $s[l,k]$ wird ein Aufwand von $O(k-l) \in O(n)$ benötigt. Da wir $O(n^2)$ viele Aufrufe haben, ergibt sich ein Gesamtaufwand von $O(n^3)$.

Übungsaufgabe: Formulieren Sie einen Pseudocode, der den obigen Algorithmus beschreibt.

4.3 Rucksackproblem

Wir betrachten das Problem des optimalen Befüllens eines Rucksackes mit Gesamtgewichtskapazität G. Dazu sei eine Menge von n Gegenständen a_i mit Gewicht $g_i > 0$ und Wert w_i gegeben. Wir wollen den Rucksack *optimal* füllen.

Aufgabenstellung: Bestimme eine Teilmenge $\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\}$ der Elemente aus $\{a_1, \ldots, a_n\}$, so dass $\sum_{j=1}^{k} g_{i_j} \leq G$ gilt und $\sum_{j=1}^{k} w_{i_j}$ maximal ist. Bestimme also den Wert:

$$w_{\max} = \max_{\{a_{i_1}, a_{i_2}, \ldots, a_{i_k}\} \subseteq \{a_1, \ldots, a_n\}} \left\{ \sum_{j=1}^{k} w_{i_j} \,\middle|\, \sum_{j=1}^{k} g_{i_j} \leq G \right\}.$$

Wenn wir dieses Problem rekursiv lösen wollen, können wir naiv einen Rekursionsbaum angeben, der in jeder Ebene i die Entscheidung für den i-ten Gegenstand berücksichtigt; siehe Abbildung 4.3. Dadurch werden alle möglichen Fälle beschrieben. Das ist aber nicht effizient, da der Baum eine exponentielle Größe hat.

Wir betrachten deshalb die folgende Vereinfachung und wollen Teilergebnisse verwenden. Sei dazu $W(i,j)$ der optimale Wert falls nur die ersten i Gegenstände betrachten werden und der Rucksack die Kapazität j hat. Am Ende soll $w_{\max} = W(n,G)$ berechnet werden.

Wir können $W(i,j)$ rekursiv wie folgt angeben. Die Frage stellt sich, ob bei Lösungen mit den ersten $i-1$ Gegenständen der i-te Gegenstand hinzugenommen werden soll oder nicht. Falls nicht, bleibt der Wert bei $W(i-1,j)$ stehen. Falls ja, kommt der Wert w_i hinzu. Allerdings müssen wir zur Berechnung von $W(i,j)$ dann rekursiv auf $W(i-1, j-g_i)$ zurückgreifen, $W(i-1, j-g_i) + w_i$ ist dann die Lösung. Der maximale Wert aus diesen beiden Fällen ergibt den Wert $W(i,j)$.

Falls j negativ wird, ist das Ergebnis ungültig. Das wird durch $W(i,j) = -\infty$

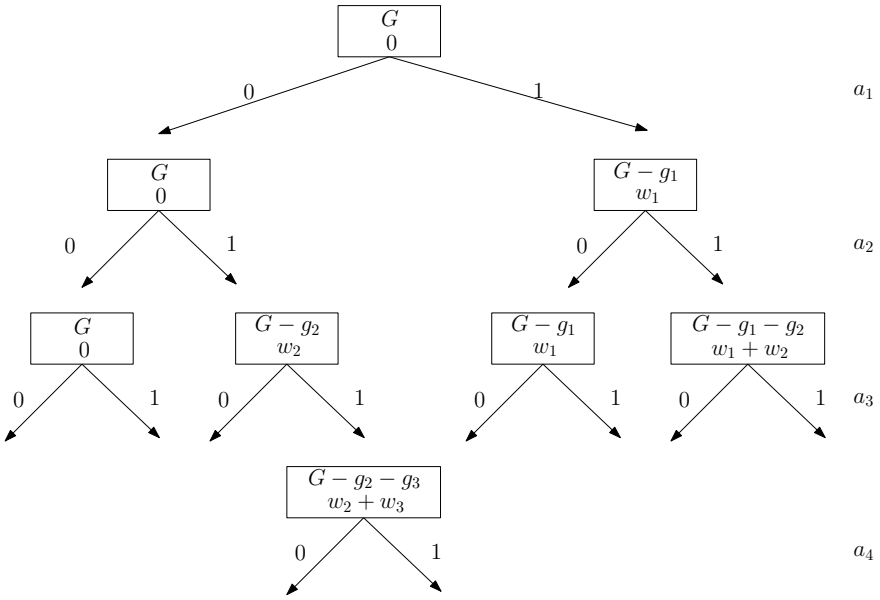

Abbildung 4.3: In Ebene i wird die Entscheidung für den i-ten Gegenstand getroffen. Der Baum hat exponentielle Größe.

ausgedrückt. Falls i null wird, setzen wir den Wert $W(i,j)$ auf null, da kein Gegenstand verwendet wird.

Wir kommen zu folgender Rekursionsgleichung, die wir danach Bottom-Up lösen wollen.

$$W(i,j) = \begin{cases} 0 & \text{falls } i = 0 \\ -\infty & \text{falls } j < 0 \\ \max\{W(i-1,j), W(i-1,j-g_i) + w_i\} & \text{sonst} \end{cases} \quad (4.8)$$

Nun wollen wir wiederum die Werte $W(i,j)$ in einem Array $W[0..n, 0..G]$ berechnen und initialisieren $W[0,j]$ und $W[i,0]$ mit 0. Aus (4.8) ergibt sich offensichtlich, dass wir die Werte am besten Zeilenweise von links nach rechts und mit steigender Zeilenzahl berechnen. Zur Berechnung von $W(i,j)$ muss $W(i-1,j)$ und $W(i-1,j-g_i)$ vorliegen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6,4)$ $a_2 = (1,2)$, $a_3 = (2,3)$, $a_4 = (5,8)$ und $a_5 = (9,10)$ mit $G = 12$.

Wir beschreiben einen Teilausschnitt der zugehörigen Matrix während der Berechnung. Zur Berechnung von $W(2,7)$ wird $W(1,7) = 4$ (ohne a_2) und $W(1,7-g_2) + w_2 = W(1,6) + 2 = 4 + 2 = 6$ (mit a_2) verglichen. $W(1,7)$ ist

also gleich 6. Zeilenweise wird die Matrix von links nach rechts gefüllt.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 0 & 2 & 2 & 2 & 2 & 2 & 4 & W(2,7)? & x & x & x & x & x \\ 0 & x & x & x & x & x & x & x & x & x & x & x & x \\ 0 & x & x & x & x & x & x & x & x & x & x & x & x \\ 0 & x & x & x & x & x & x & x & x & x & x & x & x \end{pmatrix} \end{array}$$

Auch für diesen Fall wollen wir uns am Ende die optimale Lösung generieren können. Dazu merken wir uns in einer zweiten Matrix $L[0..n, 0..G]$, woraus $W(i,j)$ entstanden ist. Zum Beispiel merken wir uns für $W(1,7)$ das a_2 benutzt wird und der Plan aus $L(1,6)$. Diese Information wird in $L(2,7)$ gespeichert, zum Beispiel durch $L[2,7] := (a_2, [1,6])$. In $L(1,6)$ steht dann wiederum, dass a_1 benutzt wird.

Korrektheit: Offensichtlich läßt sich nach Berechnung von $W(n,G)$ und $L(n,G)$ der Lösungsplan rekursiv angeben, und das optimale Packen ist berechnet worden. Die Lösung ist rekursiv genauso formuliert worden.

Wir geben hier nochmal explizit die Formulierung im Pseudocode in Algorithmus 7 an und analysieren dann die Korrektheit und die Laufzeit.

Algorithmus 7 RucksackDynProg($a_i = (g_i, w_i), i = 1, \ldots n$, Gesamtgew.: G)

```
1:  for i = 0 to G do
2:     W[0, i] := 0;
3:  end for
4:  for j = 0 to n do
5:     W[j, 0] := 0;
6:  end for
7:  for i = 1 to n do
8:     for j = 1 to G do
9:        if (j − g_i) < 0 then
10:          W[i, j] := W[i − 1, j];    // a_i passt gar nicht rein
11:       else
12:          W[i, j] := max{W[i − 1, j], W[i − 1, j − g_i] + w_i};
13:       end if
14:    end for
15: end for
16: RETURN W[n, G]
```

Laufzeit und Speicherplatz: In Algorithmus 7 werden nach der Initialisierung der Werte $W[i,0]$ und $W[0,j]$ zwei *FOR*-Schleifen geschachtelt, wobei

die Äußere von 1 bis n läuft und die Innere von 1 bis G. Im Inneren ist der Aufwand $\Theta(1)$. Also ist der Aufwand insgesamt in $O(n \cdot G)$ und für das Array benötigen wir $O(n \cdot G)$ Speicherplatz.

Übungsaufgabe: Füllen Sie die Matrizen $W[0..5, 0..12]$ und $L[0..5, 0..12]$ für das obige Beispiel. Formulieren Sie die algorithmische Lösung im Pseudocode auch für die Belegung der Matrix L zur Rekonstruktion der optimalen Lösung.

Bemerkungen: Der obige Algorithmus ist für ganzzahlige Gewichte entworfen worden, und seine Laufzeit hängt polynomiell von der Anzahl n der Gegenstände und der Größe G des Rucksacks ab. Während n linear in der Größe der Probleminstanz ist, gilt dies für G nicht, denn die Zahl G wird ja in der Eingabe durch $\log_2 G$ viele Bits dargestellt. Deshalb ist der oben vorgestellte Algorithmus *keine* polynomielle Lösung für das Rucksackproblem, und man vermutet, dass es keine gibt. Darauf werden wir im kommenden Semester noch ausführlicher eingehen.

Das Rucksackproblem wird deutlich einfacher, wenn die Gegenstände beliebig teilbar sind (Zucker, Mehl, ...). Dazu mehr im nächsten Kapitel.

Kapitel 5

Greedy Algorithmen

Neben den bereits behandelten klassischen Entwurfsmethoden der Algorithmik gibt es eine weitere Methode, die sich auf die intuitive Idee stütz, dass lokal optimale Berechnungsschritte auch global optimal sind. Es geht also auch in diesem Kapitel um Optimierungsprobleme. Der lokal optimale Berechnungsschritt soll dabei möglichst nicht wieder rückgängig gemacht werden. Außerdem werden keine Alternativen ausprobiert. In jedem Schritt wird der nächste lokal optimale Schritt berechnet und tatsächlich ausgeführt. In diesem Sinne nennt man diese Vorgehensweise auch *iterativ*. Die Kosten für einen *Iterationsschritt* sollen klein gehalten werden. Das ist meistens möglich, da wir zunächst nicht auf die Gesamtlösung achten. Zur Bestimmung des nächsten Schrittes wird allerdings häufig eine Liste von Kandidaten für den nächsten Schritt vorgehalten. Diese Liste wird nach dem Schritt aktualisiert. Greedy-Verfahren sind dennoch in der Regel einfach zu beschreiben und benötigen wenig Rechenaufwand.

Dieses *Greedy*-Prinzip wird leider nicht immer die optimale Lösung erzielen, drei Szenarien sind denkbar:

1. Wir erhalten die optimale Lösung.
2. Wir erhalten nicht die optimale Lösung, aber wir können zumindest beweisen, dass die Lösung im Vergleich zum Optimum nicht beliebig schlecht ist.
3. Die Lösung ist im Vergleich zum Optimum beliebig schlecht.

Zusammengefasst ergibt sich folgendes Grundprinzip der Greedy-Vorgehensweise:

- Lokal optimale Schritte werden iterativ ausgeführt.
- Eine Kandidatenliste wird dafür bereithalten und ggf. aktualisiert.

- Es werden keine Alternativen ausprobiert oder Zwischergebnisse tabellarisch abgespeichert.
- Die Kosten für den nächsten Schritt sind nicht sehr hoch.
- Die Lösung kann je nach Aufgabenstellung optimal, beliebig schlecht oder approximativ sein.

5.1 Rucksackproblem

Das bereits bekannte Rucksackproblem aus dem letzten Abschnitt läßt sich leicht in Greedy-Manier formulieren. Lokal optimal erscheint dabei, dass wir den Gegenstand mit dem jeweiligen besten Preis/Gewichtverhältnis als erstes Einfügen.

Beispiel: Für $n = 5$ notieren $a_i = (g_i, w_i)$ mit $a_1 = (6, 4)$ $a_2 = (1, 2)$, $a_3 = (2, 3)$, $a_4 = (5, 8)$ und $a_5 = (9, 10)$ mit $G = 12$. Die optimale Lösung ist hier offensichtlich (a_5, a_2, a_3) mit einem Wert von 15.

Zur Erinnerung: die Gegenstände haben ein Gewicht von g_i und einen Wert von w_i. Deshalb bestimmen wir zunächst eine Kandidatenliste gemäß des Verhältnisses $v_i = \frac{w_i}{g_i}$. Wir haben $v_1 = \frac{4}{6} = \frac{2}{3}$, $v_2 = \frac{2}{1} = 2$, $v_3 = \frac{3}{2} = 1.5$, $v_4 = \frac{8}{5}$ und $v_5 = \frac{10}{9}$. Das ergibt folgende Sortierung $v_2 > v_4 > v_3 > v_5 > v_1$. Ein Greedy Algorithmus wählt sukzessive den aktuell besten Kandidaten und revidiert diese Entscheidung nicht. Für den Algorithmus 8 nehmen wir zur einfachen Beschreibung an, dass die Gegenstände nach dem Preis/Gewichtsverhältnis sortiert vorliegen, also $v_1 \geq v_2 \geq \ldots \geq v_n$.

Algorithmus 8 RucksackGreedy($v_i = \frac{w_i}{g_i}$ nach Größe sortiert, Gewicht: G)

```
1: for i = 1 to n do
2:    if g_i ≤ G then
3:       λ_i := 1; G := G − g_i;
4:    else
5:       λ_i := 0;
6:    end if
7: end for
8: RETURN Σ_{i1}^{n} λ_i w_i
```

Mit einer entsprechenden Umbenennung bedeutet das in unserem Beispiel, dass der Algorithmus sukzessive a_2 und a_4 und a_3 auswählt und dann die Kapazität $12 - 1 - 5 - 2 = 4$ verbleibt. Nun kann a_5 und a_1 nicht mehr verwendet werden. Insgesamt ergibt sich ein Gesamtwert von $2 + 8 + 3 = 13$. Der Greedy-Algorithmus liefert hier also keine optimale Lösung.

Schlimmer noch, wir können ein ganz einfaches Beispiel angeben, indem der Greedy-Algorithmus beliebig schlecht wird.

Worst-Case Beispiel: Sei $a_1 = (1, 1)$ und $a_2 = (G, G-1)$. Dann ist $v_1 = 1 > \frac{G-1}{G} = v_1$. Der Greedy Algorithmus legt $\lambda_1 = 1$ und $\lambda_2 = 0$ fest und das Ergebnis ist 1. Optimal wäre $\lambda_1 = 0$ und $\lambda_2 = 1$ mit Ergebnis $G-1$. Da G beliebig groß werden kann, wird das Verhältnis zwischen dem Wert der optimalen Lösung und dem Wert der Greedy-Lösung, $\frac{G-1}{1}$, beliebig groß.

Die Situation ändert sich grundlegend, wenn die zu packenden Gegenstände beliebig teilbar sind. Dann können wir von jedem $a_i = (g_i, w_i)$ einen beliebigen Anteil $\lambda_i \in [0, 1]$ in den Rucksack packen, der $\lambda_i \cdot g_i$ wiegt und den Wert $\lambda_i \cdot w_i$ beiträgt. In Algorithmus 8 wird dann in Zeile 5 die Zahl λ_i nicht auf null gesetzt sondern auf G/g_i. Die Gegenstände $a_1, \ldots, a_{i-1}$ werden also "ganz" eingepackt, von a_i aber nur so viel, wie gerade noch in den Rucksack hineinpasst.

Bei obigem Beispiel wird also zunächst $a_1 = (1, 1)$ ganz verpackt und von $a_2 = (G, G-1)$ der Anteil $\lambda_2 = \frac{G-1}{G}$. Damit ist der Rucksack voll und hat den Wert $1 + \lambda_2(G-1)$, also mehr als zuvor. Dass dieses gute Verhalten des Greedy-Verfahrens beim Rucksackproblem kein Einzelfall ist, zeigt folgendes Theorem.

Theorem 4 *Bei beliebig teilbaren Gegenständen findet der Greedy-Algorithmus in Zeit $O(n \log n)$ stets eine optimale Lösung für das Rucksackproblem.*

Beweis. Es wird zunächst Zeit $O(n \log n)$ gebraucht, um die Gegenstände $a_i = (g_i, w_i)$ nach absteigendem Nutzen $v_i = \frac{w_i}{g_i}$ zu sortieren; danach läuft Algorithmus 8 in linearer Zeit. Zum Nachweis der Optimalität stellen wir uns vor, wir hätten einen Gewichtsanteil γ des Rucksacks mit a_i oder a_j zu füllen, und es sei

$$\frac{w_i}{g_i} = v_i > v_j = \frac{w_j}{g_j}.$$

Nehmen wir a_j, so passt der Anteil $\frac{\gamma}{g_j}$ hinein und liefert den Wert $\frac{\gamma}{g_j} \cdot w_j$. Von a_i können wir den Anteil $\frac{\gamma}{g_i}$ einpacken und erhalten den Wert

$$\frac{\gamma}{g_i} \cdot w_i > \frac{\gamma}{g_j} \cdot w_j.$$

Also sollten stets Gegenstände mit maximalem Nutzen hinzugenommen werden. □

Es gibt also Fälle, in denen der Greedy-Ansatz versagt, und andere, in denen er optimale Lösungen findet. Manchmal kommt er dem Optimum sehr nahe, ohne es ganz zu erreichen. Um die *Approximationsgüte* für eine Problemklasse Π und die Instanzen $P \in \Pi$ zu messen, betrachten wir folgende Abschätzungen.

Sei Π ein Optimierungsproblem. Sei $Greed(P)$ der Gewinn der Greedy-Lösung und $OPT(P)$ der Gewinn der optimalen Lösung für eine Probleminstanz $P \in \Pi$. Gilt für ein

Maximierungsproblem Π:

$$Greed(P) \;\geq\; \frac{1}{C} \cdot OPT(P) \quad - \quad A$$

Minimierungsproblem Π:

$$Greed(P) \;\leq\; C \cdot OPT(P) \quad + \quad A$$

für alle Instanzen $P \in \Pi$, so sagen wir, dass der Greedy-Algorithmus eine *C-Approximation* der optimalen Lösung liefert. Wichtig ist der *Approximationsfaktor* C, der nur von Π, nicht aber von P abhängen darf. Die additive Konstante A ist dazu da, konstanten Mehraufwand abzuschätzen; dazu gleich ein Beispiel beim Bepacken von Behältern.

Diese Definitionen kann man nicht nur für den Greedy-Algorithmus, sondern für beliebige Lösungsverfahren treffen. Solche *Approximationsalgorithmen* spielen eine wichtige Rolle bei der Lösung schwerer Probleme.

5.2 Das optimale Bepacken von Behältern

Das Problem des optimalen Bepackens von mehreren Behältern gleicher Gewichtskapazität G mit verschiedenen Paketen (Binpacking) gestaltet sich wie folgt. Die Pakete p haben keinen Preis sondern lediglich verschiedene Gewichte g. Insgesamt haben wir zunächst n Behälter b_j für $j = 1, \ldots, n$ mit Kapazität G und n Gegenstände p_i mit Gewichten g_i für $i = 1, \ldots, n$. Wir können $g_i \leq G$ annehmen, sonst brauchen wir einen Gegenstand gar nicht zu betrachten.

Ziel der Aufgabenstellung ist es, die Anzahl der benötigten Behälter zu minimieren. Im Prinzip bestimmen wir eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ die jedem p_i einen Behälter b_j über $Z(i) = j$ zuweist. Wir suchen das minimale k, für das es eine gültige Zuordnung Z gibt. Gültig heißt, dass die Summe aller g_i mit $Z(i) = j$ nicht größer als G ist.

Aufgabenstellung: Minimiere k, so dass eine Zuordnung $Z : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$ existiert mit

$$\sum_{Z(i)=j} g_i \leq G \text{ für alle } i \in \{1, \ldots, n\}\,. \tag{5.1}$$

Für eine Greedystrategie benutzen wir hier keine besondere Kandidatenliste, wir nehmen die Pakete so wie sie gegeben sind in der Reihenfolge

$a_1, a_2, \ldots, a_n$. Die Annahme ist wirklich sinnvoll, da wir uns vorstellen können, dass die Pakete so vom Band kommen oder so geliefert werden und zugeordnet werden müssen.

Wir berücksichtigen nicht, welche Pakete in der Zukunft auftreten. Die einzige Entscheidung die wir treffen, ist also, in welchen Behälter b_j legen wir das Paket a_i nachdem bereits die Pakete $a_1, a_2, \ldots, a_{i-1}$ einsortiert wurden. Der maximale Index j soll klein gehalten werden. Naheliegend sind die folgenden beiden Vorgehensweisen. Angenommen, wir haben bereits $a_1, a_2, \ldots, a_{i-1}$ Pakete zugeordnet. Platziere nun a_i.

First-Fit: Unter allen Behältern wähle den ersten aus, in den a_i noch hineinpasst. Genauer: Finde kleinstes j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt.

Best-Fit: Packe a_i in die aktuell vollste Kiste, in die es noch hinein passt. Genauer: Finde ein j, so dass $g_i + \sum_{1 \leq l \leq (i-1), Z(l)=j} g_l \leq G$ gilt und dabei $\sum_{1 \leq l \leq (i-1), Z(l)=j} g_l$ maximal ist.

Beispiel: Betrachten wir Behälter der Kapazität $G = 10$ und 7 Gegenstände der Größen $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ dann belegt First-Fit die Behälter wie in Abbildung 5.1 und benötigt 4 Behälter. Optimal wäre es gewesen, drei Behälter jeweils mit (g_1, g_7), (g_2, g_3, g_5) und (g_4, g_6) zu belegen. Beide Algorithmen lassen sich mit einer Laufzeit von

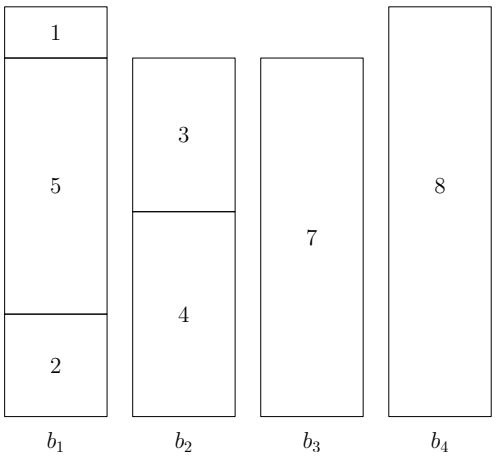

Abbildung 5.1: First-Fit für $g_1 = 2$, $g_2 = 5$, $g_3 = 4$, $g_4 = 7$, $g_5 = 1$, $g_6 = 3$ und $g_7 = 8$ benötigt einen Behälter zu viel.

$O(n^2)$ realisieren. Beispielsweise wird bei First-Fit für alle Behälter j der aktuelle Belegungswert gespeichert und wir überprüfen sukzessive von $j =$

$1, \ldots, n$ in welchen Behälter der Gegenstand a_i noch hineinpasst. Dieses Verfahren wird wiederum sukzessive für $i = 1, \ldots, n$ für die Gegenstände a_i durchgeführt. Insgesamte werden zwei FOR-Schleifen geschachtelt, die von $j = 1, \ldots, n$ respektive von $i = 1, \ldots, n$ laufen. Ähnlich kann bei Best-Fit vorgegangen werden.

Wir wollen nun zeigen, dass beide Algorithmen eine 2-Approximation der optimalen Anzahl der benötigten Behälter liefern. Wir gehen auch hier der Einfachheit halber davon aus, dass die Gegenstände ganzzahliges Gewicht haben.

Lemma 5 *Für das Binpacking Problem liefern die Algorithmen First-Fit und Best-Fit eine 2-Approximation für die optimale (minimale) Anzahl an benötigten Behältern.*

Beweis. Wir betrachten zunächst den Algorithmus First-Fit. Angenommen, er benötigt k Behälter, während die optimale Lösung mit $k_{\min}$ Behältern auskommt. Wesentlich ist nun die folgende

Beobachtung: Alle bis auf höchstens einen der von First-Fit benutzten Behälter sind mindestens halb voll.

Denn angenommen, es gäbe am Schluss zwei höchstens halb volle Behälter b_i und b_j mit $i < j$. Sei a der erste Gegenstand, der in b_j abgelegt wurde. Er hat höchstens das Gewicht $G/2$ (weil er in dem höchstens halb vollen Behälter b_j liegt) und hätte deshalb in b_i noch Platz gehabt (weil b_i selbst am Ende höchstens halb voll ist). Das ist ein Widerspruch zu $i < j$, denn FirstFit hätte ja erst testen müssen, ob a in b_i hineinpasst.

Aus der Beobachtung folgt:

$$\text{Gesamtgewicht} \;\geq\; (k-1) \cdot \frac{G}{2},$$

also

$$k \;\leq\; 2\,\frac{\text{Gesamtgewicht}}{G} \;+\; 1 \;\leq\; 2\,k_{\min} \;+\; 1,$$

denn auch die optimale Lösung benötigt mindestens so viele Behälter wie das Gesamtgewicht aller Gegenstände, geteilt durch die Behälterkapazität. Man sieht, dass hier tatsächlich eine additive Konstante in der Abschätzung auftritt.

Für BestFit wird die obige Beobachtung fast genauso bewiesen: Angenommen, Behälter b_i ist schon benutzt worden, wenn der erste Gegenstand a in b_j abgelegt wird. Dann wäre für a auch in b_i oder in einem anderen, noch volleren Behälter Platz gewesen. □

Durch schärfere Analyse lässt sich zeigen, dass FirstFit und BestFit sogar $\frac{17}{10}$ - Approximationen liefern, aber nichts Besseres. Wir zeigen jetzt, wie man die beiden Verfahren so "ärgern" kann, dass sie $\frac{5}{3}$ mal so viele Behälter brauchen wie die optimale Lösung.

Lemma 6 *Für das Binpacking Problem und die Algorithmen First-Fit und Best-Fit gibt es Beispiel-Sequenzen, die beliebig lang sein können und für die die Lösung aus den beiden Algorithmen niemals besser sein kann als $\frac{5}{3}$ mal die optimale Lösung.*

Beweis. Sei dazu $n = 18m$ und $G = 131$. Gegenstände mit Gewicht 20 plus Gewicht 45 plus Gewicht 66 füllen genau einen Behälter.

Insgesamt betrachten wir $18m$ Gegenstände, wobei in der Sequenz der Gegenstände

1. ... die ersten $6m$ ein Gewicht $g_i = 20$ für $i = 1, \ldots, 6m$,
2. ... die nächsten $6m$ ein Gewicht von $g_i = 45$ für $i = 6m + 1, \ldots, 12m$,
3. ... und die letzten $6m$ ein Gewicht von $g_i = 66$ für $i = 12m+1, \ldots, 18m$

besitzen. Wir haben also drei Sequenzblöcke 1., 2. und 3. und exakt $6m$ Behälter reichen aus.

Wir zeigen, dass für diese Sequenz First-Fit und Best-Fit jeweils $10m$ Behälter befüllen. Beide Algorithmen füllen für den ersten Sequenzblock sukzessive die ersten m Behälter mit Gewichten $6 \times 20 = 120$. Jeder Block hat eine Restkapazität von 11. Danach werden für den zweiten Block $3m$ Behälter mit jeweils 2×45 Gewicht und Restkapazität von 41. Danach werden $6m$ Behälter mit Gewicht 66 gefüllt und Restkapazität jeweils 65.

Insgesamt ergibt sich in beiden Fällen ein Quotient von $\frac{Greedy(P)}{OPT(P)} = \frac{10m}{6m} = \frac{5}{3}$. □

Wenn die Sequenzen aus dem obigen Beweis in absteigend sortierter Reihenfolge vorgelegen hätten, dann hätten beide Algorithmen die optimale Belegung mit $6m$ Behältern erzielt.

5.3 Aktivitätenauswahl

Wir betrachten das Problem, dass eine Ressource für verschiedene zeitlich beschränkte Aktivitäten benutzt werden soll. Die Aktivitäten können die Resource nicht zeitgleich benutzen. Es sollen aber soviel wie möglich Aktivitäten eingeplant werden. Beispielsweise soll die Belegung eines Konzertsaals für verschiedene Veranstaltungen geplant werden. Allgemein gibt es eine Menge

von n Aktivitäten $a_i = (s_i, t_i)$ die jeweils durch einen Starttermin, s_i, und einen Endtermin, t_i, mit $s_i < t_i$ bestimmt sind.

Zwei Aktivitäten a_i und a_j sind kompatibel, wenn die Intervalle $[s_i, t_i)$ und $[s_j, t_j)$ nicht überlappen.

Aufgabenstellung Aktivitäten-Auswahl: Für eine Menge von n Aktivitäten $a_i = (s_i, t_i)$ für $i = 1, \ldots, n$ finde die maximale Menge gegenseitig kompatibler Aktivitäten. Mit maximal ist hier die Anzahl der einplanbaren Aktivitäten gemeint.

Zunächst ist klar, dass wir die Aktivitäten zwischen dem kleinsten Startzeitpunkt $S = \min\{s_i | i = 1, \ldots, n\}$ und dem größten Endzeitpunkt $T = \max\{t_i | i = 1, \ldots, n\}$ einplanen müssen. Eine naheliegende Greedy-Idee ist, die bislang nicht verwendete Zeit zu maximieren. Deshalb sortieren wir die Aktivitäten nach den Endzeitpunkten t_i. Nehmen wir an, dass wir die Folge a_i bereits so vorliegen haben, das heißt $t_1 \leq t_2 \leq \cdots \leq t_n$.

Greedy-Strategie: Wähle stets die Aktivität, die den frühesten Endzeitpunkt hat und noch legal eingeplant werden kann.

Beispiel:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
t_i	4	5	6	7	8	9	10	11	12	13	14

Hier sind zum Beispiel $\{a_3, a_9, a_{11}\}$ kompatibel, $\{a_1, a_4, a_8, a_{11}\}$ ist eine maximale kompatible Teilmenge, die durch die Greedy Strategie ausgewählt wird. Auch $\{a_2, a_4, a_9, a_{11}\}$ ist optimal bezüglich der Anzahl der einplanbaren Aktivitäten.

Algorithmisch beschreiben wir die Greedy-Strategie ganz einfach wie in Algorithmus 9.

Algorithmus 9 AktivitätenGreedy($a_i = (s_i, t_i)$ nach t_i sortiert)

```
1:  A_1 := {a_1};
2:  last := 1;    // last ist der Index der zuletzt eingefügten Aktivität
3:  for i = 2 to n do
4:     if s_i ≥ t_last then
5:        A_i := A_{i-1} ∪ {a_i}; last := i;
6:     else
7:        A_i := A_{i-1};
8:     end if
9:  end for
10: RETURN A_n
```

Laufzeit: Es müssen lediglich vorab die Endzeitpunkte sortiert werden. Das kann in $O(n \log n)$ durchgeführt werden. Danach wird die Liste in $O(n)$ Zeit

in der FOR-Schleife abgearbeitet.

Korrektheit: Der Algorithmus ist einfach und liefert eine Lösung, das ist ein Grundprinzip der Greedy-Algorithmen. Die Analyse, dass die Lösung optimal ist, kann manchmal sehr schwer sein. Im Folgenden versuchen wir die Optimalität formal zu beweisen.

Theorem 7 *Für das Problem der Aktivitäten-Auswahl berechnet der Greedy-Algorithmus 9 in Zeit $O(n \log n)$ eine optimale Lösung.*

Beweis. Am Anfang müssen die Aktivitäten nach Endzeitpunkten sortiert werden; danach kann Algorithmus 9 in linearer Zeit laufen. Die Optimalität lässt sich folgendermaßen beweisen. Angenommen, das Greedy-Verfahren wählt k Aktivitäten $a_{i_j} = (s_{i_j}, t_{i_j})$ aus. Für jedes j sei T_j die Menge aller Aktivitäten aus der Eingabe, die den Endpunkt t_{i_j} enthalten. Alle Intervalle in T_j überlappen sich; also kann auch eine optimale Lösung OPT aus jeder Menge T_j höchstens *ein* Intervall enthalten. Das sind insgesamt höchstens k viele.

Angenommen, in OPT kommt ein weiteres Intervall $a = (s, t)$ vor, das in keiner Menge T_j auftaucht, weil es keinen der Endpunkte t_{i_j} enthält. Dieses Intervall a kann dann nur komplett vor einem Intervall a_{i_j} liegen, oder dessen Startpunkt s_{i_j} enthalten, den Endpunkt t_{i_j} aber nicht. Dann hätte das Greedy-Verfahren aber Intervall a ausgewählt, weil es sich mit $a_{i_{j-1}}$ nicht überlappt und sein Endzeitpunkt t vor t_{i_j} kommt. Widerspruch!

Daraus folgt, dass OPT nicht mehr Intervalle enthalten kann, als das Greedy-Verfahren auswählt. □

Insgesamt haben wir drei Greedy Beispiele kennengelernt, in denen entweder eine optimale Lösung, eine gute Approximation oder eine beliebig schlechte Lösung erzielt wird.

Kapitel 6

Datenstrukturen

Nachdem wir einige algorithmische Standardtechniken betrachtet haben, wollen wir uns nun mit der geschickten Repräsentation von Daten im Rechner beschäftigen und stellen ein paar einfache Datenstrukturen vor.

Die einfache Datenstruktur des Arrays haben wir bereits kennengelernt, für ein Array $A[1..n]$ können wir die Werte $A[i]$ in $O(1)$ Zeit auslesen oder ändern. Arrays stehen in allen gängigen Programmiersprachen zur Verfügung, über die interne Realisierung müssen wir uns zunächst keine Gedanken machen. Wie bereits gesehen, lassen sich die Arrays auch in mehreren Dimensionen verwenden.

Im Kapitel 2 haben wir gesehen, wie ein Array A mit n Werten sortiert werden kann. Den direkten Zugriff über $A[i]$ kann man zum Beispiel nutzen, um effizient festzustellen, ob ein Element x im Array $A[1..n]$ vorhanden ist. Falls wir die Einträge eines sortierten Arrays A sukzessive mit x vergleichen, liegt die Laufzeit dieser Vorgehensweise in $O(n)$.

Das Prinzip der *binären Suche* erlaubt eine effizientere Lösung. Dabei wird der Wert x jeweils mit dem Wert des Arrays an der Mitte des Suchbereiches verglichen und je nach Ergebnis wird in den Teilbereichen weitergesucht. Algorithmus 10 beschreibt dieses Verfahren.

Laufzeitabschätzung:

Dieser rekursive Algorithmus wird mit BinarySearch$(A, 1, n, x)$ aufgerufen und testet für $n = 1$ in konstanter Zeit, ob $A[1] = x$ ist. Sonst wird rekursiv ein Array der Größe höchstens $\lfloor \frac{n}{2} \rfloor$ betrachtet und konstanter Aufwand c für die konstante Anzahl an zusätzlichen Operationen verwendet.

Die zugehörige Rekursionsgleichnung lautet $T(n) = T\left(\lfloor \frac{n}{2} \rfloor\right) + c$ und $T(n)$ liegt beweisbar in $O(\log n)$.

Korrektheit: Falls das Array nur einen Eintrag hat, testen wir, ob dieser dem Element x entspricht. Ansonsten betrachten wir das mittlere Element

Algorithmus 10 BinarySearch(A,p,r,x)
Ist x ein Element des aufsteigend sortierten Arrays A zwischen Index p und r?

```
1: if p < r then
2:    q := ⌊(p + r)/2⌋;
3:    if x < A[q] then
4:       BinarySearch(A, p, q − 1, x);
5:    else if x > A[q] then
6:       BinarySearch(A, q + 1, r, x);
7:    else
8:       RETURN A[q] = x;
9:    end if
10: else
11:    if A[q] = x then
12:       RETURN A[q] = x;
13:    else
14:       RETURN x ∉ A;
15:    end if
16: end if
```

$q := \lfloor (p+r)/2 \rfloor$ des aufsteigend sortierten Arrays $A[p..r]$, falls $x < A[q]$ suchen wir im linken Teil, sonst im rechten Teil des Arrays. Falls beides nicht zutrifft, muss $x = A[q]$ gelten und wir geben das Element aus.

6.1 Stacks

Die oben beschriebenen Arrays lassen sich auch für die Beschreibung komplizierterer Strukturen verwenden. Wir betrachten hier einen sogenannten *Stack* auch Stapel oder Keller genannt.

Der Stack ist intern realisiert durch ein unendliches Array $S[1..\infty]$ mit einem ausgewiesenen Kopfelement $S[\text{Top}]$ für die Integer-Variable Top. Dabei ist $S[1], S[2], \ldots, S[\text{Top}]$ der aktuelle Stack und $S[\text{Top}]$ das Kopfelement, der Stack ist leer, falls $\text{Top} = 0$ gilt.

Extern stehen dem Benutzer folgende Operationen zur Verfügung.

- Push(x), ein neues Kopfelement x wird auf den Stack gelegt. Intern wird $\text{Top} := \text{Top} + 1$ und $S[\text{Top}] := x$ gesetzt
- Pop(x), gibt das aktuelle Top-Element in x zurück und löscht es aus dem Stack. Intern wird $x := S[\text{Top}]$ und $\text{Top} := \text{Top} - 1$ gesetzt. Falls Top bereits 0 ist, wird berichtet, dass der Stack leer ist, extern beispielsweise über den Wert NIL.

- Top(x) gibt das aktuelle Kopfelement in x zurück. Intern über S[Top].

Wir wollen auch hier ein Beispiel für die effiziente Verwendung von Stacks angeben. Nehmen wir an, die Entwicklung verschiedener Aktien verläuft über einen bestimmten Zeitraum $I = [0, d]$ jeweils linear, das heißt beschrieben durch einen Funktion $f_i(x) = a_i x + b_i$ für $x \in I$. Nun möchte ein Analyst für jedes einzelne $x \in I$ schnell ermitteln, welche Aktie den größten Wert erzielt.

Naiv müsste man für den jeweiligen Wert x alle n Funktionswerte $f_i(x)$ miteinander vergleichen und könnte dann den maximalen Wert ermitteln.

Besser wäre es, gleich die sogenannte *obere Kontur* aller Funktionen explizit anzugeben. Zu jedem Zeitpunkt beschreibt eine der Funktionen f_i die momentan maximalen Werte. Wenn wir die Übergänge der jeweiligen *maximalen* Funktionen speichern könnten, können wir für die jeweiligen Teil-Intervalle schnell den optimalen Wert angeben.

Aufgabenstellung: Wir wollen die Funktion

$$f_{\max} : x \mapsto \max\{f_1(x), f_2(x), \ldots, f_3(x)\}$$

explizit durch Intervalle und zugehörige Teil-Funktionen beschreiben. $f_{\max}$ bezeichnen wir als die *obere Kontur* von $f_1, f_2, \ldots, f_n$ über dem Intervall $I = [0, d]$, siehe auch Abbildung 6.1.

Der Einfachheit halber nehmen wir an, dass die Werte b_i paarweise verschieden sind. Zunächst sortieren wir die Funktionen an der Stelle $x = 0$ nach den Y-Koordinaten b_i. Nehmen wir nun an, das o.B.d.A. $b_1 > b_2 > b_3 > \cdots > b_n$ gilt.

Wir verwenden einen Stack, und der Stack soll am Ende sukzessive die maximalen Funktionen und die Übergänge enthalten.

Zunächst ist f_1 maximal und wir legen $(f_1, (0, b_1))$ auf den Stack als Kopfelement. Das ist die momentane obere Kontur der betrachteten Funktionen. Nun nehmen wir f_2 und betrachten den aktuellen Stack. Falls die beiden Funktionen einen Schnittpunkt $s(f_1, f_2) = (x_{f_1,f_2}, y_{f_1,f_2})$ besitzen, wechselt ab diesem Punkt das Maximum von f_1 zu f_2. Wir legen somit f_2 mit Push$((f_2, s(f_1, f_2)))$ auf den Stack, der Stack repräsentiert die aktuelle obere Kontur. Das unterste Element f_1 startet bei $(0, b_1)$ und wird durch das nächste Element f_2 bei $s(f_1, f_2)$ abgewechselt.

Nun betrachten wir f_3. Falls f_3 keinen Schnittpunkt mit f_2 bildet, kann f_3 ausgelassen werden, die Funktion liegt vollständig unterhalb von f_2. Falls jedoch f_3 mit f_2 einen Schnittpunkt $s(f_2, f_3) = (x_{f_2,f_3}, y_{f_2,f_3})$ hat, entscheidet die Lage des Schnittpunktes über die aktuelle Kontur der beteiligten Funktionen.

Fall 1.: Liegt der Punkt $s(f_2, f_3)$ mit seiner X-Koordinate x_{f_2,f_3} links vom X-Wert x_{f_1,f_2} des Startpunktes $s(f_1, f_2)$ des aktuellen Top-Elementes f_2, so

kann f_2 offensichtlich insgesamt gar nicht an der oberen Kontur teilnehmen. In diesem Fall wird f_2 durch Pop(x) vom Stack genommen. Der gleiche Test wird mit dem nächsten Top-Element (hier f_1) wiederholt.

Fall 2.: Der Punkt $s(f_2, f_3)$ liegt mit seiner X-Koordinate x_{f_2,f_3} rechts vom X-Wert x_{f_1,f_2} des Startpunktes $s(f_1, f_2)$ des aktuellen Top-Elementes f_2. Dann wird mit Push($(f_3, s(f_2, f_3))$) der nächste Abschnitt der Kontur auf den Stack ge*pusht* und der aktuelle Stack beschreibt tatsächlich die obere Kontur der beteiligten Funktionen mit den jeweiligen Übergängen.

In Abbildung 6.1 sehen wir, wie dieses Prinzip im Allgemeinen funktioniert. Nachdem bereits sukzessive $(f_1, (0, b_1))$, $(f_2, s(f_1, f_2))$, $(f_3, s(f_2, f_3))$, $(f_4, s(f_3, f_4))$ auf den Stack ge*pusht* wurden, wird nun f_5 betrachtet. Wir betrachten jeweils die Lage des Schnittpunktes mit dem Top-Element. Da die Schnittpunkte $s(f_4, f_5)$ und $s(f_3, f_5)$ jeweils links von den Punkten $s(f_3, f_4)$ respektive $s(f_2, f_3)$ liegen, werden f_4 und f_3 nacheinander vom Stack ge*poppt*. Dann wird mit dem dann aktuellen Top-Element $(f_2, s(f_1, f_2))$ der Schnittpunkttest mit f_5 durchgeführt. Der Test der Schnittpunkte $s(f_2, f_5)$ und $s(f_1, f_2)$ ergibt, dass $s(f_2, f_5)$ rechts von $s(f_1, f_2)$ liegt und somit $(f_5, s(f_2, f_5))$ auf den Stack ge*pusht* werden muss, $(f_2, s(f_1, f_2))$ bleibt erhalten. Der Stack gibt die aktuelle Kontur wieder.

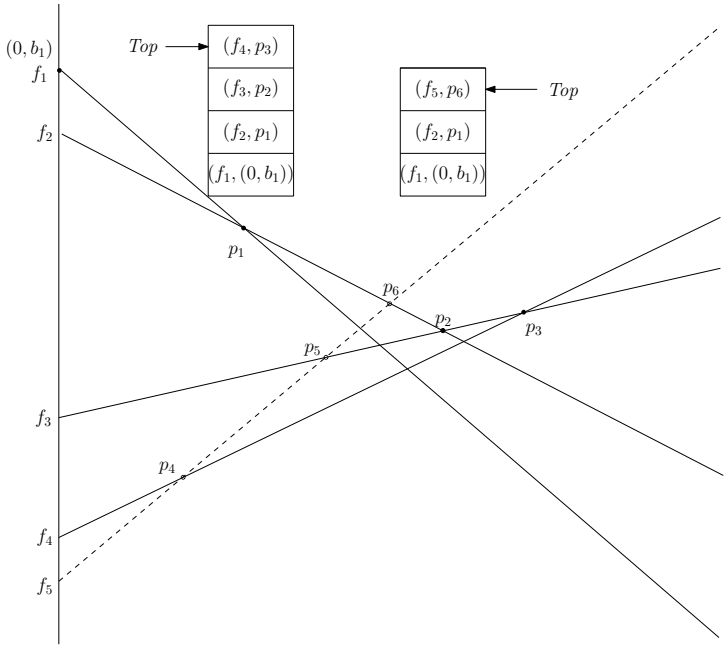

Abbildung 6.1: Vor dem Einfügen von f_5 beschreibt der Stack die obere Kontur durch die Funktionen f_1, f_2, f_3 und f_4 mit den jeweiligen Intervallgrenzen. Wenn f_5 betrachtet wird, müssen f_3 und f_4 den Stack verlassen.

Korrektheit: Wir wissen, dass sich zwei Geraden nur einmal schneiden können. Induktiv nehmen wir an, dass der aktuelle Stack die obere Kontur für die ersten $f_1, f_2, \ldots, f_{k-1}$ Geraden wiedergibt. Für das erste Element f_1 ist das der Fall.

Das aktuelle Top-Element $(f_j, s(f_i, f_j))$ des Stacks ist das letzte Stück der Kontur und wird mit der nächsten Geraden f_k verglichen.

Falls der Schnittpunkt $s(f_j, f_k)$ links von $s(f_i, f_j)$ liegt, kann f_j nicht zur oberen Kontur gehören. Rechts von $s(f_j, f_k)$ dominiert f_k die Gerade f_j, links von $s(f_i, f_j)$ dominiert die Gerade f_i die Gerade f_j. Die Push-Operation ist also in diesem Fall korrekt und das nächste Top-Element des Stacks wird mit f_k verglichen.

Falls der Schnittpunkt $s(f_j, f_k)$ rechts von $s(f_i, f_j)$ liegt, wird korrekterweise $(f_k, s(f_j, f_k))$ ans Ende der Kontur und in den Stack eingefügt.

Zu jedem Zeitpunkt ist die Kontur korrekt, am Ende gibt der Stack die Kontur mit den jeweiligen Intervallen korrekt wieder.

Laufzeitabschätzung: Gemäß der Beschreibung betrachten wir sukzessive die Geraden $f_1, f_2, \ldots, f_n$ und vergleichen diese mit dem jeweiligen Top-Element. Jede Gerade kann nur einmal auf den Stack ge*pusht* werden wird aber auch nur einmal vom Stack ge*poppt*. Insgesamt haben wir somit nur $O(n)$ viele Push− und Pop−Operationen, das gleiche gilt dann für die Top-Operationen. Für die Vergleiche muss das Top-Element betrachtet werden, das geht dann aber stets einer Push− oder Pop−Operation voraus. Insgesamt bestimmen wir die obere Kontur in $O(n)$ Zeit, für das anfängliche Sortieren war $O(n \log n)$ Zeit nötig.

6.2 Dynamische Listen

Eine weitere sehr nützliche Datenstruktur ist die verkettete Liste. Hierbei besitzt jedes Datenelement einen *Zeiger* auf das nachfolgende Element. Der Zeiger gibt an, in welcher Speicherzelle das nächste Listenobjekt zu finden ist.

Eine schematische Darstellung einer linearen Liste findet sich in Abbildung 6.2. Die Liste ist als Zeiger auf das Kopfelement gegeben, jedes weitere Element enhält einen Dateneintrag und einen Zeiger auf das nächste Element. Das letzte Element enthält den speziellen Zeiger NIL. Rechnerintern

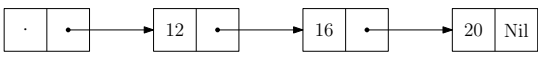

Abbildung 6.2: Die schematische Darstellung einer Liste.

kann man sich eine lineare Liste wie folgt vorstellen. Eine Liste von Zahlen $(16, 20, 33, 14, 5)$, die mit Adresse 7 angesprochen wird, kann beispielsweise so abgespeichert sein.

Adresse	1	2	3	4	5	6	7	8	9	10	11	...
Wert	20		33	16			·	5		14	12	...
Zeiger	3		10	1			4	NIL		8		...

Im Gegensatz zum Array mit fester Länge lassen sich verkettete Listen sehr effizient dynamisch verändern. Es gibt dazu (programmiersprachenspezifische) Befehle zum *allokieren* von neuen Speicherelementen mit Datenobjekt und zugehörigem Zeiger. Außerdem können die Zeiger entsprechend umgelenkt werden.

Soll in die obige Liste nach dem Wert 33 ein neues Element 17 eingefügt werden, so wird zunächst (programmiersprachenspezifisch) eine freie Adresse $A = 6$ für ein neues Datenobjekt mit noch nicht spezifizertem Wert W_A und noch nicht spezifiziertem Zeiger Z_A erzeugt. Danach wird $W_A := 17$ eingetragen. Der Zeiger vom Element 33 wird auf die Adresse 6 des neuen Eintrags gesetzt und der Zeiger Z_A verweist auf 10, also auf das nachfolgende Element 14. In Java läßt sich das beispielsweise durch Listen*objekte* mit der entsprechenden Ausgestaltung realisieren. Für ein neues Element wird ein neues Listenobjekt angelegt.

Der Vorteil der Listen liegt in der einfachen dynamischen Veränderung, für das Einfügen und Löschen von Listenelementen ist jeweils nur konstanter Aufwand notwendig. Will man bei Arrays den Zusammenhang erhalten, dann sind Kopiervorgänge mit linearer Laufzeit für das Einfügen und Löschen notwendig.

6.3 Bäume und Suchbäume

Eine natürliche Verallgemeinerung von Listen sind Bäume. Formal können Bäume wie folgt rekursiv definiert werden.

1. Ein *Baum* $T = (w, T_1, T_2, \ldots, T_m)$ besteht aus der *Wurzel* w und Teilbäumen T_i, wobei alle T_i Bäume mit paarweise disjunkten Knotenmengen sein müssen.

2. T_i ist dann der i-te *direkte Teilbaum* von T.

3. m ist der *Grad* des Knoten w.

4. Ein Knoten von Grad 0 ist ein *Blatt*, ein Knoten vom Grad > 0 ist ein *innerer Knoten*

5. Eine Liste lässt sich als ein Baum interpretieren, bei dem jeder Knoten bis auf den letzten den Grad 1 hat und der letzte Grad 0.

Bei Abbildung 6.3 handelt es sich um ein Beispiel einer *grafischen Realisation* eines Baumes. Im Folgenden werden wir die grafische Realisation und die obige formale Definition als identisch betrachten. Der i-te *direkte Teilbaum* soll von links nach rechts betrachtet an der i-ten Stelle per Kante angefügt werden.

Klassische Bezeichnungen: Für einen Baum T mit Wurzel v und direkten Teilbäumen $T_1, T_2, \ldots, T_m$ mit Wurzeln $w_1, w_2, \ldots, w_m$ heißt w_i der i-te *Sohn von* v und v der *Vater von* w_i. Kurzschreibweisen $v = \text{Wurzel}(T)$, $v = \text{Vater}(w_i)$, $w_i = \text{Sohn}(v, i)$. Ein Knoten v ist *Nachfolger* eines Knoten w innerhalb eines Baumes T falls es eine Folge von Knoten $(v_1, v_2, \ldots, v_n)$ mit $v_1 = v$ und $v_n = w$, so dass $v_{i+1} = \text{Vater}(v_i)$ für $i = 1, 2, \ldots, n-1$ gilt.

Die Tiefe eines Knoten $v \in T$ wird rekursiv wie folgt definiert.

$$\text{Tiefe}(v, T) = \begin{cases} 0 & \text{falls } v = \text{Wurzel}(T) \\ 1 + \text{Tiefe}(v, T_i) & \text{falls } T_i \text{ direkter Teilbaum} \\ & \text{von } T \text{ mit } v \in T_i \end{cases} \tag{6.1}$$

Die Höhe eines Baumes T wird durch

$$\text{Höhe}(T) = \max\{\text{Tiefe}(b, T) | b \text{ ist Blatt von } T\} \tag{6.2}$$

festgelegt und entspricht damit der Anzahl der Kanten auf dem längsten Pfad von der Wurzel zu einem Blatt.

Equivalent ist die rekursive Definition

$$\text{Höhe}(T) = \begin{cases} 0 & \text{falls } v = \text{Wurzel}(T) \text{ ein Blatt ist} \\ 1 + \max \text{Höhe}(T_i) & \text{falls } T_i \text{ direkter Teilbaum von } T \end{cases} \tag{6.3}$$

Die Abbildung 6.3 zeigt ein Beispiel eines Baumes der Höhe 4. Die Blätter v_{12} und v_{13} haben die maximale Tiefe 4.

Interne Realisation: Intern kann ein Baum analog zur verketteten Liste durch Zeiger realisiert werden. Ein Datenobjekt *Knoten* enthält dabei zum Beispiel einen Datensatz, einen Schlüssel (siehe unten) und eine Liste von Zeigern auf die Knoten der jeweiligen Söhne und ggf. einen Zeiger auf den Vaterknoten. Methoden für den Knoten erlauben es, auf den i-ten Sohn zuzugreifen, die Daten auszugeben oder die Zeiger zu verändern. Blätter werden als spezielle Knoten realisiert (zum Beispiel durch NIL-Zeiger, wenn keine Information in den Blättern gebraucht wird).

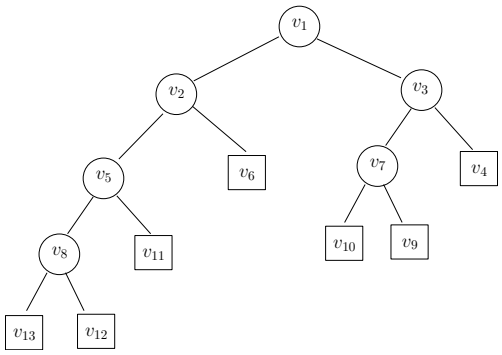

Abbildung 6.3: Die grafische Darstellung eines Baumes der Höhe 4.

6.3.1 Binärbäume und Suchbäume

Bäume, deren innere Knoten alle den Grad 2 haben werden Binärbäume genannt. Die beiden direkten Teilbäume eines inneren Knotens bezeichnet man dann als linken und rechten Teilbaum gemäß der eindeutigen grafischen Realisation. In den Knoten sollen außerdem Schlüssel abgespeichert werden. Über diese Schlüssel soll dann auf bestimmte Daten zugegriffen werden können. Je nachdem, ob sich die Schlüssel in den inneren Knoten oder nur in den Blättern befinden, gibt es die folgende allgemeine Unterteilung:

Art der Informationsspeicherung:

Suchbaum: Die Schlüssel befinden sich nur an den inneren Knoten, die Blätter sind leer (klassisch).

Blattsuchbaum: Die Schlüssel befinden sich nur in den Blättern, in den inneren Knoten gibt es nur *Wegweiser* zu den Knoten. Blattsuchbäume erlauben beispielsweise effiziente Bereichsanfragen.

Abbildung 6.4 zeigt einen Suchbaum (I) und einen Blattsuchbaum (II) für die Schlüssel $(1, 3, 5, 7, 12, 15)$.

Im Folgenden werden wir zunächst Suchbäume betrachten, in den Bättern gibt es also keine Informationen. Insgesamt ergeben sich nun Möglichkeiten, wie man bei einem Durchlauf (Besuch aller Knoten) durch den Baum die Ausgabe der Schlüssel oder Daten (oder die Besuchsreihenfolge der Knoten) einfach beschreiben kann.

Präorder: Besuche die Wurzel (gebe die Daten aus), besuche dann den linken Teilbaum rekursiv und danach den rechten Teilbaum rekursiv, kurz wLR.

Postorder: Besuche zuerst den linken Teilbaum rekursiv und danach den rechten Teilbaum rekursiv und danach die Wurzel (gebe die Daten aus), kurz LRw.

Lexikographische Ordnung (In-Order): Besuche zuerst den linken Teilbaum rekursiv und danach die Wurzel (gebe die Daten aus) und dann den rechten Teilbaum rekursiv, kurz LwR.

Wird beispielsweise der Baum in Abbildung 6.4(I) nach der lexikographischen Ordnung durchlaufen, erhalten wir die Schlüssel in der Reihenfolge $(1, 3, 5, 7, 12, 15)$. Die Präorder-Ausgabe ergibt: $(7, 5, 3, 1, 15, 12)$.

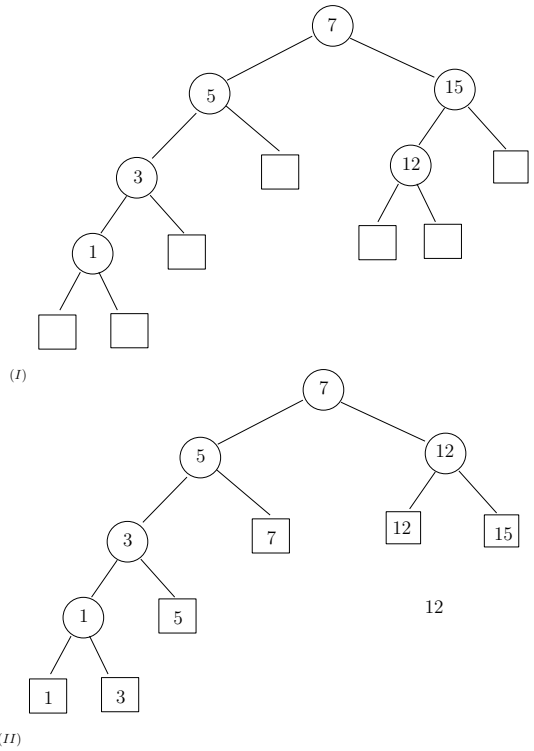

Abbildung 6.4: (I) Der Baum speichert Daten(Schlüssel) in den inneren Knoten, der Durchlauf nach *lexikographischer Ordnung* LwR gibt die Schlüssel sortiert in der Reihenfolge $(1, 3, 5, 7, 12, 15)$ aus. (II) Ein Blattsuchbaum für die gleichen Schlüssel. Innere Knoten enthalten als Wegweiser zu den Blättern den maximalen Schlüssel des linken Teilbaums.

Für einen Knoten v eines Baumes bezeichne Schlüssel(v) den zugehörigen Schlüssel. Ein binärer Suchbaum hat die sogenannte *Suchbaumeigenschaft* falls für die Knoten und Schlüssel des Baumes gilt:

- Falls v ein Knoten des binären Suchbaumes ist und w ein Knoten im linken Teilbaum von v ist, dann gilt Schlüssel(w) $\leq$ Schlüssel(v).
- Falls v ein Knoten des binären Suchbaumes ist und w ein Knoten im rechten Teilbaum von v ist, dann gilt Schlüssel(w) $>$ Schlüssel(v).

In den Blattsuchbäumen erfüllen die Wegweiser zusammen mit den Schlüsseln der Blätter die Suchbaumeigenschaft. Suchbäume sollen es ermöglichen, möglichst effizient auf Daten über die Schlüssel zuzugreifen. Datenstrukturen sollen es insgesamt ermöglichen einfache Operationen effizient durchzuführen.

Ein klassisches Beispiel ist das **Wörterbuch-Problem** (Dictionary-Problem). Für eine Menge von Wörtern(Daten) wird dabei wie bereits oben erwähnt für jedes Wort ein Zahlenschlüssel verwendet, über den auf das Wort zugegriffen werden soll. Typische Operationen für dieses Problem sind:

Suche: Suche das Datenobjekt zu einem vorgegebenen Schlüssel k.

Einfügen: Füge ein neues Datenobjekt mit dem Schlüssel k ein.

Entfernen: Entferne ein Datenobjekt mit Schlüssel k.

Berichte Min/Max: Gebe das Datenobjekt mit kleinsten/größten Schlüssel aus.

Nächster Nachbar: Für einen Schlüssel m, finde ein Datenobjekt mit größten (kleinsten) Schlüssel $k \leq m$ ($k \geq m$).

Bereichsanfrage: Für ein Intervall $[l, r]$ finde alle Datenobjekte mit Schlüssel $k \in [l, r]$.

Im Folgenden beschreiben wir die Operationen jeweils für einen binären Suchbaum mit zugehörigen Schlüsseln. Dafür nehmen wir an, dass wir für einen Knoten mit Schlüssel(v) auf den gespeicherten Schlüssel zugreifen können. Die leeren Blätter werden als Blatt repräsentiert; so können Knoten als Blatt identifiziert werden. Die Operation Suchen kann nun wie folgt implementiert werden.

Laufzeit: Mit jedem rekursiven Aufruf der Funktion steigen wir definitiv eine Stufe tiefer in den Baum ab, Die Laufzeit der Suchoperation hängt somit von der Höhe des Baumes ab. Für h = Höhe(T) liegt die Laufzeit in $O(h)$. Für einen Baum mit n Knoten kann die Laufzeit sogar n Schritte betragen, falls der Baum zu einer Liste entartet ist.

Korrektheit: Wenn der Baum die Suchbaumeigenschaft erfüllt, dann rufen wir die Funktion rekursiv jeweils mit den richtigen Teilbaum auf, solange wir den Schlüssel k noch nicht gefunden haben. Falls der Schlüssel existiert,

Algorithmus 11 Search(T,k), mit $T = (w, T_1, T_2)$ oder $T =$ Blatt.
Suche im Suchbaum T den Knoten v mit Schlüssel k oder gebe Blatt zurück, wenn kein solcher Knoten vorhanden ist.

```
1: if T = Blatt then
2:    RETURN Blatt
3: else if k = Schlüssel(w) then
4:    RETURN w
5: else if k < Schlüssel(w) then
6:    RETURN Search(T_1, k)
7: else if k > Schlüssel(w) then
8:    RETURN Search(T_2, k)
9: end if
```

treffen wir auf den entsprechenden Knoten in T und geben ihn zurück, andernfalls enden wir an einem Blatt und geben es zurück.

Bei den Operationen Einfügen und Entfernen soll natürlich die Suchbaumeigenschaft erhalten bleiben. Wir stellen hier das Einfügen vor. Zunächst wurde dabei bereits ein Knoten v mit Schlüssel(v) $= k$ als Datenobjekt angelegt.

Algorithmus 12 Insert(T,v), mit $T = (w, T_1, T_2)$ oder $T =$ Blatt.
Füge einen Knoten v mit neuem Schlüssel(v) $= k$ sortiert in Suchbaum T ein und gib T zurück.

```
1: if T = Blatt then
2:    T := (v, Blatt, Blatt)
3: else if k ≤ Schlüssel(w) then
4:    T := (w, Insert(T_1, v), T_2)
5: else if k > Schlüssel(w) then
6:    T := (w, T_1, Insert(T_2, v))
7: end if
8: RETURN T
```

Korrektheit und Laufzeit: Auch hier steigen wir mit jedem rekursiven Aufruf tiefer in den Baum T hinab, um das Element an der richtigen Stelle einzufügen. Der neue Knoten v ersetzt ein Blatt von T. Der neue Teilbaum, der den eingefügten Knoten enthält wird zurückgegeben und rekursiv im jeweiligen Vaterknoten aktualisiert. Die Laufzeit liegt wiederum in $O(h)$, wobei $h =$ Höhe(T) gilt.

Das Löschen eines Knoten v eines binären Suchbaumes ist nicht ganz so einfach und hängt davon ab, welche Söhne der jeweilige Knoten besitzt. Der Knoten wird wiederum über seinen Schlüssel in $O(h)$ Zeit gefunden. Wir unterscheiden folgende Fälle.

Entfernen eines Knoten v:

- Falls der Knoten v nur Blätter als Söhne hat, entfernen wir ihn einfach und ersetzen ihn durch ein Blatt. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 1 in der Abbildung 6.4(I).

- Falls der Knoten v ein Blatt als Sohn hat und einen Teilbaum T_1 mit Wurzel w, ersetzt der Knoten w den Knoten v, der Knoten v wird ausgeschnitten. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 5 in der Abbildung 6.4(I).

- Falls der Knoten v zwei echte Teilbäume T_1 und T_2 als Unterbäume hat, wählen wir den am weitesten rechts gelegenen Knoten w in T_1 aus. Dieser enthält den nächstkleineren Schlüssel als v und hat mindestens ein Blatt als Sohn. Dieser Vorgängerknoten w wird ausgeschnitten und durch seinen Nachfolger wie in den ersten beiden Fällen ersetzt. Der Knoten v wird nun allerdings durch w ersetzt. Zum Beispiel beim Entfernen des Knoten mit Schlüssel 7 in der Abbildung 6.4(I), kann der Knoten mit Schlüssel 5 den Knoten mit Schlüssel 7 ersetzen.

In jedem der beschriebenen Fälle haben wir im von der Struktur her einen Teilbaum durch seinen linken oder rechten Unterbaum ersetzt, der andere Teilbaum war dabei stets ein Blatt. Wir mussten dabei den Knoten des zugehörigen Schlüssels finden und ggf. den Knoten des Nachfolgeschlüssels. Die Operation kann in $O(h)$ ausgeführt werden.

Übungsaufgabe: Man zeige: falls ein Knoten v mit Schlüssel k in einem Suchbaum zwei echte Unterbäume hat, dann hat der Knoten mit dem k nachfolgenden Schlüssel mindestens ein Blatt als Unterbaum.

Für Binärbäume lassen sich viele interessante Eigenschaften ableiten bzw. beweisen. Exemplarisch beweisen wir hier, dass für einen Binärbaum mit n Blättern die Höhe in $\Omega(\log n)$ liegt. Außerdem hängt die Anzahl der inneren Knoten fest von der Anzahl der Blätter ab und umgekehrt.

Lemma 8

1. *Ein Binärbaum mit n inneren Knoten hat $n+1$ Blätter.*

2. *Ein Binärbaum mit n Blättern hat $n-1$ innere Knoten.*

3. *In jedem Binärbaum gibt es maximal 2^i Knoten der Tiefe i.*

4. *Die Höhe eines Binärbaumes mit insgesamt n Knoten (oder Blättern) liegt in $\Omega(\log n)$.*

Beweis. 1.: Zunächst beweisen wir induktiv die Aussage über die Anzahl der Blätter bei n inneren Knoten:

Induktionsanfang: $n = 0$. Ein Baum mit keinem inneren Knoten hat genau ein Blatt.

Induktionsschritt: Sei w die Wurzel eines Binärbaumes mit insgesamt $n \geq 1$ inneren Knoten. Dann teilt sich der Baum bei w in zwei Teilbäume T_1 und T_2 auf, die beide zusammen $n_1 + n_2 = n - 1$ innere Knoten besitzen. Nach Induktionsannahme gilt: T_1 hat genau n_1+1 und T_2 hat genau n_2+1 Blätter, insgesamt hat T genau $n_1 + n_2 + 2 = n + 1$ Blätter.

2.: Der Beweis kann analog zum obigen Beweis geführt werden, eine leichte Übungsaufgabe. Man kann aber auch einfach $n := n - 1$ setzen.

3.: Dass es in jedem Binärbaum nur maximal 2^i Knoten der Tiefe i geben kann, läßt sich leicht induktiv zeigen bzw. folgt direkt aus der Definition der Tiefe und ist eine leichte Übungsaufgabe.

4.: Nun betrachten wir die Höhe eines Binärbaumes T_n mit n Knoten. Wir zeigen direkt, dass Höhe$(T_n) \geq \lceil \log(n + 1) \rceil - 1$ für alle $n \geq 1$ gilt.

Die Höhe des Baumes ist die maximale Tiefe und deshalb wollen wir einen Binärbaum erstellen, dessen Knoten eine kleinstmögliche Tiefe haben. Nach obiger Aussage kann der Baum maximal $\sum_{i=1}^{l} 2^l = 2^{l+1} - 1$ Knoten der Tiefe $\leq l$ besitzen. Somit muss für n Knoten eine gewisse Mindesttiefe gelten, um alle Knoten unterzubringen.

Sei nun $k = \min_l \{l | 2^{l+1} - 1 \geq n\}$, offensichtlich ist k eine kleinstmögliche Höhe eines Baumes für n Knoten. Dann gilt offensichtlich $k \geq \lceil \log(n + 1) \rceil - 1$ denn aus $2^{k+1} - 1 \geq n$ folgt $2^{k+1} \geq n + 1$ und wenn wir darauf den Logarithmus anwenden, dann folgt wegen der Ganzzahligkeit von $k+1$, dass $k+1 \geq \lceil \log(n+1) \rceil$ gilt. Kein Baum mit geringerer Höhe kann alle n Knoten unterbringen und die Aussage gilt. □

Vorbereitend auf das nächste Kapitel wollen wir zeigen, dass wir in einem Suchbaum Knoten lokal umorganisieren können, ohne die Suchbaumeigenschaft zu verletzen. Die sogenannten Rotationen wollen wir dann im nächsten Kapitel für die Einhaltung von Höhenbalancen nutzen.

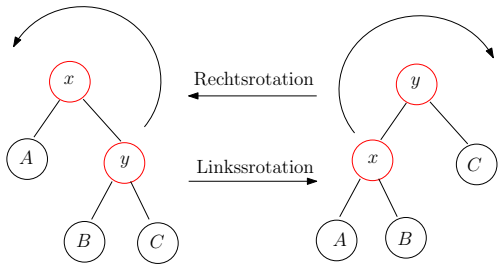

Abbildung 6.5: Bei einer Links- bzw. Rechtsrotation bleibt die Suchbaumeigenschaft erhalten.

Linksrotation eines Knoten y mit Knoten x: Sei T ein Teilbaum mit folgender Eigenschaft. Die Wurzel x von T ist ein innerer Knoten mit linken Teilbaum A und rechten Teilbaum mit Wurzel y und seien B und C die Teilbäume (links und rechts) von y wie in der Abbildung 6.5. Dann können die Knoten y und x so nach *links* rotiert werden, dass y die Wurzel von T bildet mit rechten Teilbaum C. Der Knoten x wird linker Teilbaum von y mit linken und rechten Teilbäumen A und B.

Rechtsrotation eines Knoten x mit Knoten y: Symmetrisch zur Linksrotation, siehe Abbildung 6.5.

Bei einer Linksrotation werden die Tiefen der Knoten in C um eins verringert, die Tiefen in B bleiben erhalten und die Tiefen in A werden um eins erhöht.

Bei einer Rechtsrotation werden die Tiefen der Knoten in A um eins verringert, die Tiefen in B bleiben erhalten und die Tiefen in C werden um eins erhöht.

Lemma 9 *Die Linksrotation erhält die Suchbaumeigenschaft des Teilbaumes T und des Gesamtbaumes. Der In-Order-Durchlauf gibt die gleiche Reihenfolge aus. Die Rotation kann mit konstanten Aufwand implementiert werden.*

Beweis. Dass die Rotation mit konstanten Aufwand implementiert werden kann, ergibt sich daraus, dass bei einer Implementierung mit Zeigern nur konstant viele Zeiger für die Rotation verändert werden müssen.

Die Schlüssel in A sind kleiner gleich Schlüssel(x) und die Schlüssel in B und C sind größer gleich Schlüssel(x). Es gilt Schlüssel(x) $\leq$ Schlüssel(y). Schlüssel(y) ist größer gleich den Schlüsseln in B aber kleiner gleich den Schlüsseln in C. Somit bleibt die Suchbaumeigenschaft insgesamt erhalten.

Der In-Order-Durchlauf beginnt jeweils mit einem In-Order-Durchlauf in A, gibt dann x aus, führt einen In-Order-Durchlauf von B durch, gibt dann y aus und endet mit einem In-Order-Durchlauf von C. □

Naheliegend ist folgende Eigenschaft. Übungsaufgabe: Ein binärer Baum erfüllt genau dann die Suchbaumeigenschaft, wenn der In-Order-Durchlauf die Schlüssel in aufsteigend sortierter Reihenfolge ausgibt.

Im Allgemeinen werden auch Bäume mit einem Grad von ≤ 2 für jeden inneren Knoten als Binärbäume bezeichnet. In diesem Kontext wird ein Binärbaum mit Grad exakt 2 für jeden inneren Knoten als *vollständig* bezeichnet. Insbesondere weil für die Suchbäume manchmal auch die eigentlichen Blätter weggelassen werden, findet man in der Literatur auch allgemeine Binärbäume. Wir werden stets die (leeren) Blätter mit notieren und somit vollständige Binärbäume betrachten.

6.4 AVL-Bäume

Wie wir bereits im letzten Abschnitt gesehen haben, ist die Höhe des Baumes entscheidend für elementare Operationen auf der Datenstruktur eines Binärbaumes. Das Lemma 8 zeigt uns allerdings bereits, dass eine bessere Laufzeit als $\Omega(\log n)$ für das Einfügen, Suchen und Entfernen nicht zu erwarten ist. Als Beispiel für balancierte Bäume, die möglichst nahe an dieser Grenze bleiben, betrachten wir nun AVL-Bäume.

Ein binärer Suchbaum heißt AVL-Baum, falls für *jeden* Knoten v gilt, dass die Höhe der beiden Teilbäume von v sich nur um Eins unterscheiden.

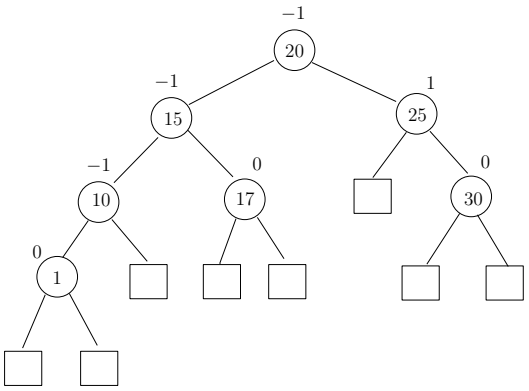

Abbildung 6.6: An jedem Knoten eines AVL-Baumes beträgt der Höhenunterschied der Teilbäume maximal 1.

Für jeden Knoten v gibt es einen sogenannten *Balancefaktor*.

$$\text{bal}(v) := \text{Höhe}(\text{rechts}(v)) - \text{Höhe}(\text{links}(v)) \tag{6.4}$$

In AVL-Bäumen (AVL nach Adelson, Velskij und Landis) gilt für alle Knoten $\text{bal}(v) \in \{-1, 0, 1\}$ ein Beispiel zeigt die Abbildung 6.6. Wir wollen nun die Höhe eines AVL-Baumes ermitteln. Dazu geben wir eine Höhe h vor und stellen fest, wieviele Knoten $n(h)$ ein AVL-Baum der Höhe h mindestens hat. Dann können wir umgekehrt feststellen, welche maximale Höhe ein AVL-Baum für n Knoten aufweisen kann.

Ein AVL-Baum der Höhe h mit minimaler Knotenanzahl $n(h)$ heißt auch *minimaler AVL-Baum der Höhe h*. Wir zählen die Blätter $b(h)$ eines solchen Baumes und können nach Lemma 8 dann die Anzahl der Knoten ermitteln. Sei T_h ein minimaler AVL-Baum mit $b(h)$ Blättern. Betrachte Abbildung 6.7. Offensichtlich gilt:

- Ein minimaler AVL-Baum der Höhe 0 hat ein Blatt, $b(0) = 1$.

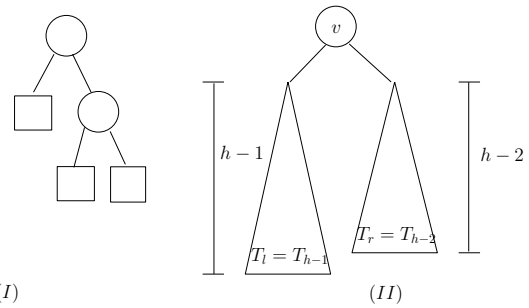

Abbildung 6.7: (I) Der minimale AVL-Baum der Höhe 2 hat $b(2) = 3$ Blätter. (II) Der minimale AVL-Baum der Höhe $h \geq 2$ hat $b(h) = b(h-1) + b(h-2)$ Blätter.

- Ein minimaler AVL-Baum der Höhe 1 hat zwei Blätter, $b(1) = 2$.
- Ein minimaler AVL-Baum der Höhe 2 hat drei Blätter, $b(2) = 3$.
- Sei T_h eine minimaler AVL-Baum der Höhe h mit Wurzel v und seien T_l und T_r die Teilbäume von v. Dann hat zumindest einer der beiden Teilbäume eine Höhe von $h-1$ und der andere muss mindestens eine Höhe von $h-2$ haben. Da wir die Anzahl der Bätter minimieren wollen, wählen wir o.B.d.A. $T_l = T_{h-1}$ und $T_r = T_{h-2}$ und für $h \geq 2$ gilt somit:

$$b(h) = b(h-1) + b(h-2) \tag{6.5}$$

Die Zahlen $b(h)$ sind die Fibonacci-Zahlen $b(h) = \text{fib}(h+1)$, die wir beim dynamischen Programmieren betrachtet hatten. Wir leiten jetzt eine geschlossene Formel für die Fibonacci-Zahlen her und gewinnen daraus eine obere Schranke für die Höhe von AVL-Bäumen.

Mit der Abkürzung $f_i = \text{fib}(i)$ haben wir also die Rekursion

$$f_0 = 1,\ f_1 = 1,\ f_{i+2} = f_{i+1} + f_i$$

für alle $i \geq 0$. Wie bei der Anzahl der Beklammerungen von n Faktoren verwenden wir die Methode *erzeugender Funktionen,* um die Rekursion zu lösen. Sei also $f(X) = \sum_{i=0}^{\infty} f_i X^i$. Dann ist

$$\begin{aligned} f(X) &= f_0 + f_1 X + f_2 X^2 + f_3 X^3 + \ldots \\ X\,f(X) &= f_0 X + f_1 X^2 + f_2 X^3 + \ldots \\ X^2\,f(X) &= f_0 X^2 + f_1 X^3 + \ldots . \end{aligned}$$

Zieht man die zweite und dritte Gleichung von der ersten ab, ergibt sich

$$f(X) - X\,f(X) - X^2\,f(X) = f_0 + (f_1 - f_0)\,X = 1,$$

also

$$f(X) \;=\; \frac{1}{1-X-X^2}. \tag{6.6}$$

Wir würden hieraus gern eine Potenzreihe machen, um dann die f_i durch Koeffizientenvergleich bestimmen zu können. Bei einem linearen Nenner wie $1-\lambda X$ könnten wir einfach die Formel für die geometrische Reihe anwenden und bekämen

$$\frac{1}{1-\lambda X} \;=\; \sum_{i=0}^{\infty} \lambda^i\, X^i.$$

Deshalb wenden wir Partialbruchzerlegung an und suchen Zahlen ϕ, ψ und a, b mit

$$\frac{1}{1-X-X^2} \;=\; \frac{a}{1-\phi X} \;+\; \frac{b}{1-\psi X}. \tag{6.7}$$

Weil $1/\phi$ und $1/\psi$ Nullstellen des Nenners $1-X-X^2$ sein sollen, müssen ϕ und ψ die Nullstellen der Gleichung

$$X^2 - X - 1$$

sein, die sich aus der vorigen ergibt, wenn man X durch $1/X$ ersetzt und mit X^2 multipliziert. Also ist etwa

$$\phi \;=\; \frac{1+\sqrt{5}}{2} \quad \text{und} \quad \psi \;=\; \frac{1-\sqrt{5}}{2}. \tag{6.8}$$

Außerdem folgt aus (6.7) durch Addition der beiden Brüche

$$a \,+\, b \;=\; 1 \quad \text{und} \quad a\,\psi \,+\, b\,\phi \;=\; 0.$$

Setzt man in der zweiten Gleichung $b = 1-a$ und $\psi - \phi = -\sqrt{5}$ ein, ergibt sich $a = \frac{\phi}{\sqrt{5}}$ und $b = 1 - \frac{\phi}{\sqrt{5}} = -\frac{\psi}{\sqrt{5}}$. Also folgt aus (6.7)

$$f(X) \;=\; \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} \phi^i\, X^i \;-\; \frac{1}{\sqrt{5}} \sum_{i=0}^{\infty} \psi^i\, X^i,$$

und durch Koeffizientenvergleich erhalten wir folgende Formel für die Fibonacci-Zahlen:

Lemma 10 *Für die i-te Fibonacci Zahl f_i mit $f_0 = f_1 = 1$ und $f_{i+2} = f_{i+1} + f_i$ gilt*

$$f_i \;=\; \frac{1}{\sqrt{5}}\,\phi^{i+1} \;-\; \frac{1}{\sqrt{5}}\,\psi^{i+1},$$

wobei

$$\phi \;=\; \frac{1+\sqrt{5}}{2} \;=\; 1.6180\ldots \quad \textit{und} \quad \psi \;=\; \frac{1-\sqrt{5}}{2} \;=\; -0.6180\ldots.$$

Die Zahl ϕ ist der "goldene Schnitt": Teilt man eine Strecke so in zwei Teile mit Längen v und w, dass $v = \phi w$ gilt, so ist auch $v + w = \phi v$.

Das Wachstum von f_i ist durch den ersten Term in Lemma 10 bestimmt, denn der zweite Term ist eine (alternierende) Nullfolge in i.

Nun können wir eine scharfe obere Schranke für die Höhe von AVL-Bäumen angeben.

Theorem 11 *Ein AVL-Baum mit n Blättern (und $n-1$ inneren Knoten) hat höchstens die Höhe* $\log_\phi(n+1) \in O(\log n)$.

Beweis. Für einen AVL-Baum der Höhe h gilt

$$n \;\geq\; b(h) \;=\; f_{h+1} \;\geq\; \frac{1}{\sqrt{5}}\,\phi^{h+2} - 1,$$

also haben wir

$$h \;\leq\; \log_\phi\left(\sqrt{5}\,(n+1)\right) \;-\; 2 \;\leq\; \log_\phi(n+1).$$

□

Als nächstes wollen wir die Operationen Einfügen und Entfernen für AVL-Bäume betrachten. Diese Operationen sollen

- ... *strukturerhaltend* durchgeführt werden, die AVL-Eigenschaft soll erhalten bleiben.
- ... *kostengünstig* durchgeführt werden, der Durchlauf in $O(\log n)$ und eventuelles Ausbalancieren in konstanter Zeit durch Umhängen von Zeigern.

6.4.1 Einfügen eines neuen Knoten

Dazu betrachten wir das Beispiel aus Abbildung 6.6. Im Baum notieren wir den Wert bal(v) für jeden Knoten. Wir merken uns an jedem Knoten die Höhe des linken und rechten Teilbaumes, so können wir schnell den Wert bal(v) bestimmen bzw. neu berechnen. Wir wollen einen Knoten z mit Schlüssel 5 einfügen.

Zunächst fügen wir dabei abwärtslaufend wie in der Prozedur 12 den Knoten ein; siehe Abbildung 6.8. Danach werden aufwärtslaufend die bal(v)-Werte geändert bis zum ersten Mal eine Disbalance auftritt. Die neuen bal(v)-Werte werden durch die veränderten Höhen sukzessive berechnet. Am Knoten y mit Schlüssel 10 tritt zum ersten Mal eine Disbalance auf. Die Teilbäume des Knoten sollen umgeordnet werden. Eine einfache Rechtsrotation am Knoten x mit y führt hier leider nicht zum Erfolg, der neu eingefügte Knoten z (Baum

B aus Abbildung 6.5) würde seine Höhe nicht verlieren und am neuen Knoten x wird dann lediglich der Wert $\text{bal}(x) = +2$ notiert.

Deshalb führen wir hier die folgende *Doppelrotation* aus:

1. Zunächst eine Linksrotation des Knoten z mit x.
2. Danach eine Rechtsrotation des Knoten z mit y.

Zunächst verliert dabei der Knoten x an Höhe, das wird aber durch die zweite Rotation wieder ausgeglichen. In Abbildung 6.9 sehen wir die Zwischener-

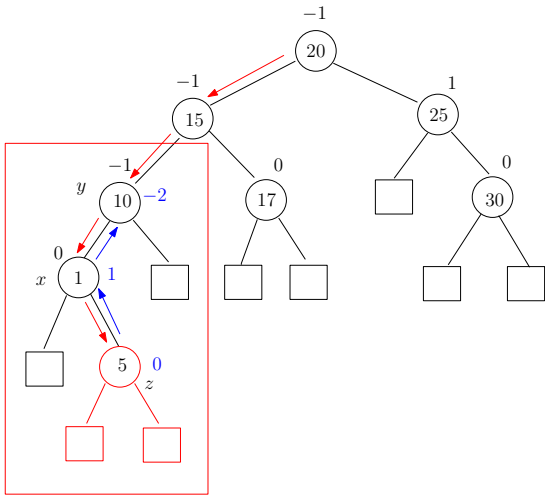

Abbildung 6.8: Das Einfügen eines Knoten z mit Schlüssel 5 in den AVL-Baum aus Abbildung 6.6 durch einen Abwärtslauf zum entsprechenden Blatt. Danach werden aufwärtslaufend die $\text{bal}(v)$-Werte geändert, bis zum ersten Mal eine Disbalance auftritt. Der Teilbaum in der eingezeichneten Box muss ausbalanciert werden.

gebnisse der Rotationen. Der zugehörige Teilbaum ist danach ausbalanciert. Abschließend stellt sich die Frage ob gegebenenfalls noch die $\text{bal}(v)$ Einträge bis Wurzel aktualisiert müssen. Wir haben den Teilbaum des Knoten y ausbalanciert aber er hat seine Höhe behalten, somit ändern sich die Werte nicht. Die Abbildung 6.10 zeigt das Ergebnis nach dem Einfügen.

Das Einfügen gestaltet sich insgesamt in den folgenden Schritten:

1. Abwärtsdurchlauf gemäß der Schlüssel und Eintrag des neues Knoten. Laufzeit: $O(\log n)$
2. Aufwärtsdurchlauf zum Anpassen der $\text{bal}(v)$-Werte bis die erste Disbalance an einem Knoten v auftritt. Laufzeit: $O(\log n)$.

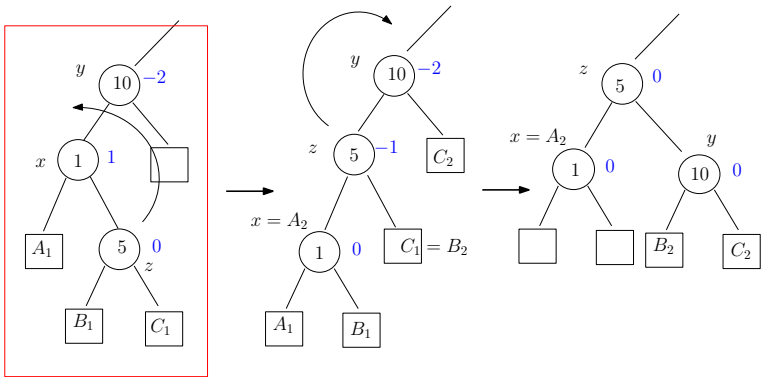

Abbildung 6.9: Zuerst wird eine Linksrotation am Knoten z mit x vorgenommen, danach eine Rechtsrotation am Knoten z mit y. Der Teilbaum ist danach balanciert.

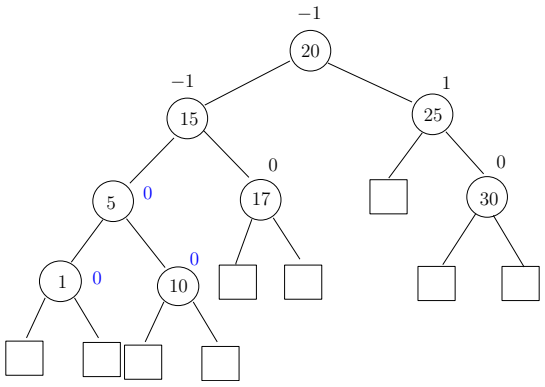

Abbildung 6.10: Der ausbalancierte Baum nach dem Einfügen des Knoten z des Schlüssels 5. Die bal(v)-Werte alle anderen Knoten bleiben gleich, da die Höhe des Originalteilbaumes erhalten bleibt.

3. Ausbalancierung des zugehörigen Teilbaums durch Rotation. Laufzeit: $O(1)$, inklusive umsetzen der Zeiger.

Kategorisierung der Rotationen:

Die Doppelrotation (erst links, dann rechts) im obigen Beispiel war notwendig, weil der Weg vom Knoten y der ersten Disbalance zum neu eingefügten Knoten z zuerst links und danach rechts abgebogen ist. Der neue Knoten liegt also vom Knoten y aus gesehen im rechten Unterbaum des linken Unterbaumes. Dieser rechte Unterbaum muss an Höhe verlieren.

Falls der besagte Weg zuerst rechts und danach links abbiegt, wird die Doppelrotation in der Reihenfolge rechts/links ausgeführt. Der neue Knoten liegt dann also vom Knoten y aus gesehen im linken Unterbaum des rechten Unterbaumes.

Manchmal reicht auch eine einfach Rechts- oder Linksrotation aus. Das ist der Fall, wenn der Weg vom Knoten y der ersten Disbalance zum neu eingefügten Knoten z nur rechts oder nur links abbiegt.

Falls wir beispielsweise in den neuen Baum aus Abbildung 6.10 einen weiteren Knoten z mit Schlüssel 32 einfügen, so erhalten wir wie in Abbildung 6.11 die erste Disbalance am Knoten y und führen eine einfache Linksrotation des Knoten x mit y aus. Das Ergebnis ist in Abbildung 6.12 zu sehen.

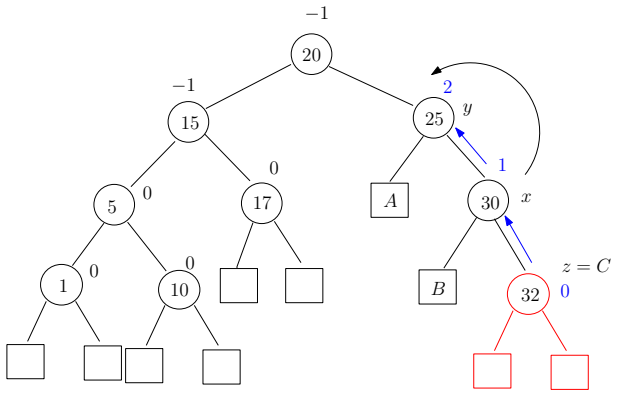

Abbildung 6.11: Nach dem Einfügen des Knoten z mit Schlüssel 32 reicht eine einfache Linksrotation.

Nach diesen Beispielen beschreiben wir die Doppelrotation (links/rechts) und die einfache Rotation (links) schematisch. Die Doppelrotation (rechts/links) und die einfache Rotation (rechts) verlaufen analog.

Doppelrotation (links/rechts): Nach dem Einfügen eines Knoten z und dem Ermitteln des untersten Disbalance-Knoten y läuft der Weg zum Knoten z in den rechten Unterbaum des linken Unterbaumes, siehe Abbildung 6.13.

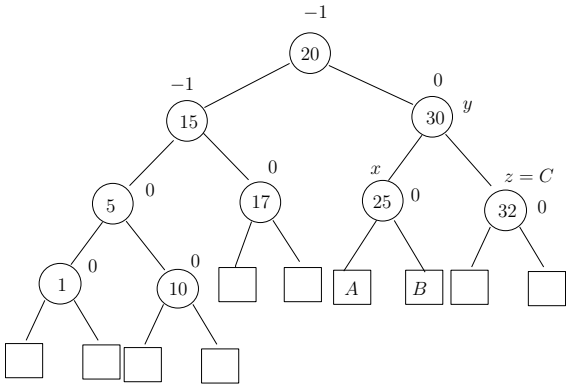

Abbildung 6.12: Die Höhe des ausbalancierten Baumes ändert sich nicht, alle anderen bal(v)-Werte bleiben gleich.

Zunächst wird eine Linksrotation des Knoten w mit x ausgeführt, siehe das

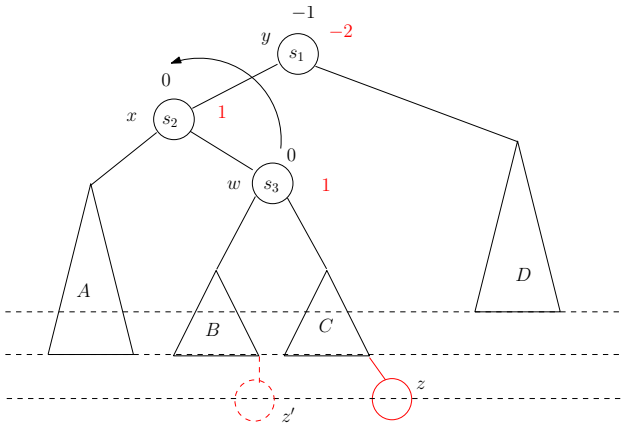

Abbildung 6.13: Die Situation vor einer (links/rechts) Doppelrotation. Der Knoten z könnte auch in B eingefügt worden sein.

Ergebnis in Abbildung 6.14. Danach wird eine Rechtsrotation des Knoten w mit y ausgeführt, siehe das Ergebnis in Abbildung 6.14. Der Baum ist ausgeglichen und das wäre auch der Fall, wenn der Knoten z in B eingefügt worden wäre, siehe z' in Abbildung 6.14.

Einfache Rotation (links): Nach dem Einfügen eines Knoten z und dem Ermitteln des untersten Disbalance-Knoten y läuft der Weg zum Knoten z in den rechten Unterbaum des rechten Unterbaumes, siehe Abbildung 6.16.

Wir beschreiben nun in der Prozedur 13, wie die Zeiger bei einer Linksrotation verändert werden müssen.

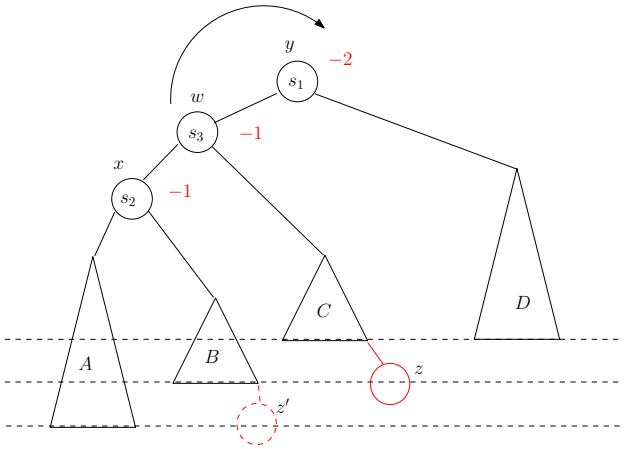

Abbildung 6.14: Die Situation vor der zweiten Rotation. Der Knoten z könnte auch in B eingefügt worden sein.

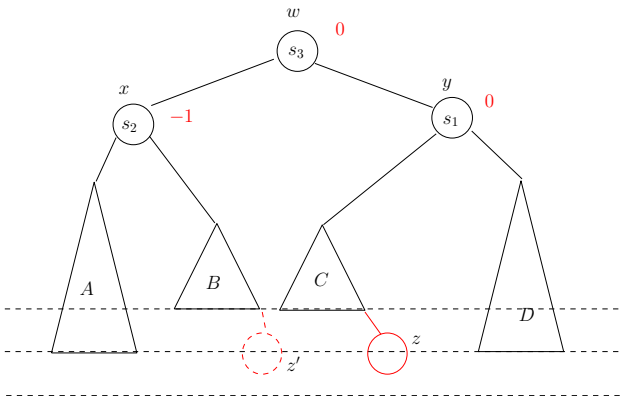

Abbildung 6.15: Nach der zweiten Rotation ist der Baum ausgeglichen. Falls der Knoten z in den Teilbaum B eingefügt worden wäre, wäre der Baum ebenfalls ausgeglichen. In jedem Fall ist die Höhe des Baumes wieder identisch zur Ausgangshöhe und damit alle $bal(v)$-Werte von w an aufwärts unverändert.

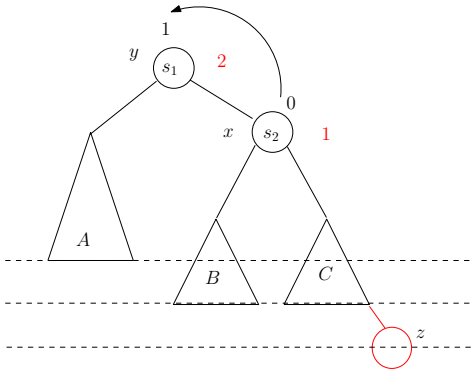

Abbildung 6.16: Die Situation vor einer einfachen Rotation (links). Der neue Knoten z liegt im rechten Unterbaum des rechten Unterbaumes von y.

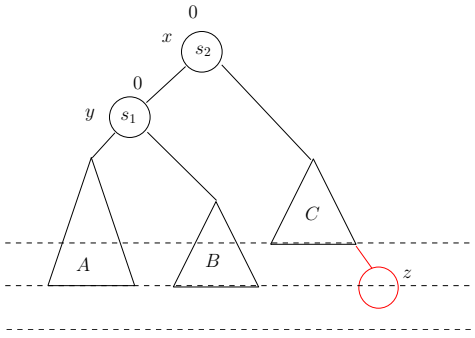

Abbildung 6.17: Die Situation nach einer einfachen Rotation (links). Der Teilbaum C hat an Höhe verloren, der Teilbaum A an Höhe gewonnen, der Baum ist ausgeglichen. Außerdem ist die Höhe des Baumes wieder identisch zur Ausgangshöhe.

Prozedur 13 Linksrotation(T,x)
Linksrotation des Knoten x mit $y = \text{Vater}(x)$

```
1: y := Vater(x); w := Vater(y);
2: if w ≠ NIL and y = LinkerSohn(w) then
3:    marker := links;
4: else
5:    marker := rechts;
6: end if
7: rechts(y) := links(x);
8: links(x) := y;
9: if marker := links then
10:   links(w) := x;
11: else if marker := rechts then
12:   rechts(w) := x;
13: end if
```

Eine Rechtsrotation kann analog ausgeführt werden. Die Doppelrotation (links/rechts) die für einen Disbalance-Knoten y ausgeführt wird, kann dann durch folgende Prozedur beschrieben werden.

Prozedur 14 DoppelrotationLR(T,y)
Doppelrotation (links/rechts) am Disbalance-Knoten y

```
1: x := links(y);
2: w := rechts(x);
3: Linksrotation(T, w);
4: Rechtsrotation(T, w);
```

Übungsaufgabe: Beschreiben Sie die Schemata für eine Doppelrotation (rechts/links) und eine einfache Rotation (rechts) und die zugehörigen Zeigerprozeduren.

Korrektheit: Von der Logik her können Disbalancen am Knoten y durch das Einfügen eines Knoten z nur durch eine der vier Einfügefälle von z (links/rechts, rechts/links, links/links, rechts/rechts) entstehen. Wird der Knoten z beispielsweise direkt nur links unter y eingefügt dann kann rechts(y) nur aus einem Blatt oder einem Baum der Höhe 1 bestanden haben. In jedem Fall entsteht bei y gar keine Disbalance. Der andere Fall geht analog.

Außerdem wird durch das Beseitigen der Disbalance beim Knoten y stets der gesamte Baum ausgeglichen. Falls beispielsweise die Doppelrotation (links/rechts) durchgeführt wurde, muss der Knoten y vor dem Einfügen den Wert -1 gehabt haben. Nach der Rotation hat der Teilbaum mit neuer Wurzel w zwar den bal-Wert 0 aber die Höhe des Teilbaumes bleibt erhalten. Das heißt, alle anderen bal(v)-Werte bleiben erhalten. Der Baum ist insgesamt

ausgeglichen. Die anderen Fälle gelten analog.

Insgesamt führen wir ein korrektes Einfügen eines Knoten z in Laufzeit $O(\log n)$ durch.

6.4.2 Entfernen eines Knoten

Beim Entfernen von Knoten kommen analoge Operationen zur Anwendung. Wie bereits in Abschnitt 6.3.1 bemerkt, wird stets ein Knoten v durch seinen linken oder rechten Teilbaum w ersetzt, der andere Teilbaum war dabei ein Blatt. Wenn der Baum nach dieser Operation am Knoten w unbalanciert ist, muss der Baum rebalanciert werden.

Wir geben zunächst ein Beispiel an und beschreiben dann die allgemeine Vorgehensweise. Wenn in Abbildung 6.12 der Knoten mit Schlüssel 15 und zwei echten Teilbäumen entfernt werden soll, so wird der Knoten zuerst durch den Knoten mit nächstgrößtem Schlüssel 17 ersetzt und der vormalige Knoten mit Schlüssel 17 entfernt. Dieser hatte mindestens ein Blatt, in unserem Fall sogar zwei Blätter. Der Teilbaum wurde entfernt und in diesem Fall durch ein Blatt ersetzt. Auf dem Weg zur Wurzel werden dann die bal(v)-Werte angepasst, bis zum ersten Mal eine Disbalance entdeckt wird. Nun müssen wiederum Rotationen ausgeführt werden, um die Disbalance zu beseitigen. Offensichtlich reicht hier eine einfache Rechtsrotation am Knoten mit Schlüssel 5 aus, siehe Abbildung 6.18. Es kann ebenfalls sein, dass Dop-

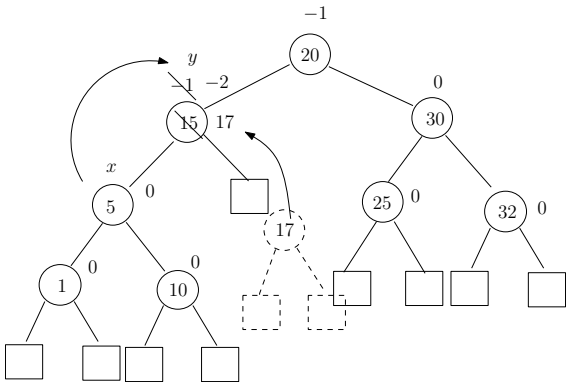

Abbildung 6.18: Die Situation vor einer einfachen Rotation nach dem Entfernen des Knoten mit Schlüssel 15. Der ehemalige Knoten des Schlüssels 17 wir entfernt.

pelrotationen notwendig sind. Wir beschreiben kurz das allgemeine Schema. Im Gegensatz zur Beschreibung des Einfügens eines Knoten beschreiben wir hier die einfache Rotation (rechts) und die Doppelrotation (links/rechts) die durch eine -2 Disbalance entstehen. Analoge Operationen einfache Rotation

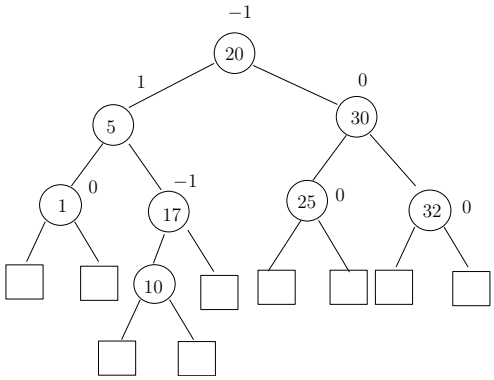

Abbildung 6.19: Nach der Rotation ist der Baum ausgeglichen.

(links) und Doppelrotation (rechts/links) entstehen bei +2 Disbalancen.

Angenommen die Disbalance tritt am Knoten y zuerst auf. Falls die Disbalance im Knoten y durch einen -2 Wert gegeben ist, wurde ein rechter Teilbaum C von y angehoben. Für den linken Nachfolger x ergeben sich zwei Fälle. Falls der bisherige bal(x)-Wert 0 oder -1 war, reicht eine einfache Rechtsrotation am Knoten x mit Knoten y aus, siehe Abbildung 6.20. Der Teilbäume A von x wird um eins angehoben, der Teilbaum C um eins gesenkt. Der Teilbaum B respektive B' behält seine Höhe. Es kann auch vorkommen (Fall B'), dass die Höhe des Baumes insgesamt um Eins sinkt.

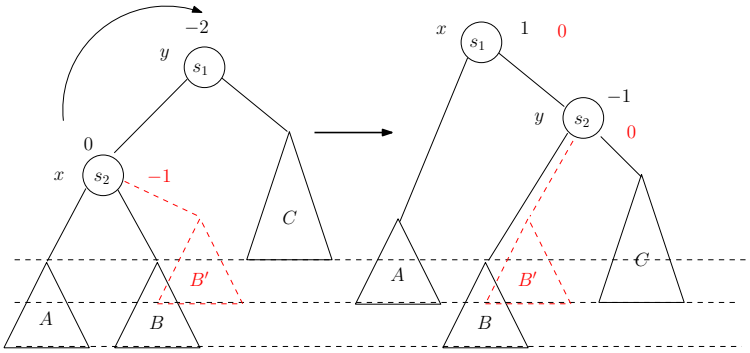

Abbildung 6.20: Die Situation vor einer einfachen Rotation. Der Disbalance-Knoten y (bal(y) $= -2$) hat einen linken Nachfolger x mit bal(x) $\in \{0, -1\}$. Dann reicht eine einfache Rechtsrotation des Knoten x mit y aus.

Falls allerdings der bal(x) Wert $+1$ war, hat der rechte Nachfolger w von x zwei Teilbäume B und C, von denen mindestens einer zusammen mit w eine höhere Tiefe als der linke Nachfolger A von x aufweist. Eine einfache Rechtsrotation von x mit y wird die Höhe des Baumes w erhalten, die Höhe

von A aber absenken, somit entsteht eine $+2$ Disbalance.

Hier ist somit wiederum eine Doppelrotation notwendig. Zunächst eine Linksrotation des Knoten w mit x und danach eine Rechtsrotation des Knoten w mit y. Das vollständige Ergebnis ist in Abbildung 6.21 zu sehen. Beachte, dass C und B auch die Rollen tauschen können. Das heißt, dass entweder B oder C die Disbalance am Knoten y zusammen mit D verursacht. In jedem Fall wird der Baum rebalanciert, hat aber insgesamt in jedem Fall die Höhe um Eins verringert. Übungsaufgabe: Beschreiben Sie die Schemata für die

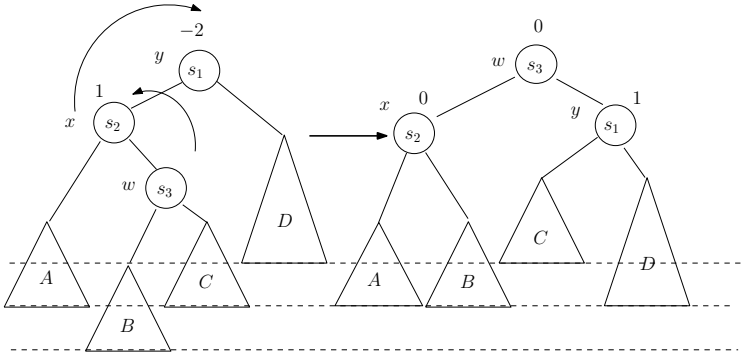

Abbildung 6.21: Die Situation vor einer Doppelrotation (links/rechts). Der Disbalance-Knoten y ($\text{bal}(y) = -2$) hat einen linken Nachfolger x mit $\text{bal}(x) = 1$. Dann ist eine Doppelrotation (zuerst Linksrotation Knoten w mit x und danach Rechtsrotation Knoten w mit y) notwendig.

Rotationen nach dem Entfernen, falls sich die Disbalance durch einen ersten -2 Wert an einem Knoten y ergibt und somit ein rechter Teilbaum von y an Höhe verloren hat. Beschreiben Sie die zwei Unterfälle, dass der linke Nachfolger x von y den Wert $\text{bal}(x) = -1$ bzw. $\text{bal}(x) \in \{0, 1\}$ aufweist.

Korrektheit: Für den ersten Disbalance Knoten y gibt es nur die beiden Möglichkeiten $\text{bal}(y) \in \{2, -2\}$, wobei die Disbalance nur durch die jeweils beschriebenen Fälle aufgetreten sein konnten. Entweder wurde die Höhe eines rechten oder eines linken Teilbaumes von y reduziert. Für den jeweils anderen Nachfolger x von y ergeben sich dann die zwei möglichen Unterfälle. Insgesamt wird zur Rebalancierung von y entweder eine einfache Rotation (links oder rechts) oder eine Doppelrotation (links/rechts oder rechts/links) ausgeführt. Danach ist der Knoten y rebalanciert. Wir sehen aber, dass der Teilbaum von y (im Vergleich zur Situation vor dem Enfernen) insgesamt immer um 1 an Höhe verliert, deshalb kann es sein, dass die verringerte Höhe von y weitere $\text{bal}(v)$-Werte auf dem Weg zur Wurzel verändert.

Das ist nicht weiter tragisch. Wir wenden die Argumentation sukzessive wieder an. Für den nächsten Disbalance-Knoten auf dem Weg zur Wurzel führen wir die gleichen Operationen durch. Spätestens wenn wir an der Wurzel an-

gekommen sind, ist der gesamte Baum ausgeglichen.

Laufzeit: Für das Entfernen wird zunächst ein Aufwand in $O(\log n)$ benötigt. Für die Rebalancierung laufen wir aufsteigend zu den nächsten zu rebalancierenden Knoten. Der Weg zur Wurzel hat eine Länge von $O(\log n)$. Jede einzelne Balancierungsoperation kostet $O(1)$ Zeit. Insgesamt wird nur $O(\log n)$ Zeit benötigt.

Damit haben wir folgendes Ergebnis:

Theorem 12 *In einem AVL-Baum mit n Blättern kann man einen Schlüssel in Zeit $O(\log n)$ suchen, einfügen oder entfernen.*

Theoretisch könnte man AVL-Bäume auch zum Sortieren benutzen: Erst fügt man die n Schlüssel nacheinander in einen anfangs leeren Baum ein, mit Kosten n mal $O(\log n)$, dann gibt man sie in Zeit $O(n)$ durch einen In-Order-Durchlauf sortiert aus. In der Praxis würde man aber Mergesort oder andere spezielle Sortierverfahren bevorzugen, weil sie einfacher zu implementieren sind und kleinere Konstanten in den O-Abschätzungen stecken.

Man kann AVL-Bäume auch als Blattsuchbäume verwenden; dann stehen die Schlüssel in den Blättern, und die internen Knoten enthalten Wegweiser, zum Beispiel jeweils den maximalen Schlüssel im linken Teilbaum. Beim Einfügen und Entfernen müssen die Wegweiser ebenfalls aktualisiert werden. An der logarithmischen Laufzeit ändert sich dadurch nichts.

Manchmal möchte man neben Suchen, Einfügen und Entfernen auch *Bereichsanfragen* beantworten: Für ein beliebiges Intervall $[l, r]$ berichte alle Datenobjekte, deren Schlüssel in diesem Intervall liegt. Bei AVL-Blattsuchbäumen kann man diese Aufgabe auf verschiedene Arten lösen. Sehr bequem ist es, die Blätter mit Zeigern zu verketten. Dann kann man zum Beispiel nach der linken Intervallgrenze l suchen, und von dem Blatt, in dem die Suche endet, den Zeigern nach rechts folgen, bis man den ersten Schlüssel mit Wert $> r$ trifft; das geht offenbar in Zeit $O(\log n + k)$, wobei k die Anzahl der gefundenen Schlüssel bedeutet. Weil die Verzeigerung der Blätter sich bei Rotationen nicht ändert, müssen die zusätzlichen Zeiger nur beim Einfügen und Entfernen aktualisiert werden, was genauso funktioniert wie bei dynamischen Listen.

6.4.3 Sweep

Als Beispiel betrachten wir noch einmal das Problem des dichtesten Punktepaars: Gegeben sind n Punkte in der Ebene, gesucht ist ein Paar mit minimalem Abstand.

Mit Divide-and-Conquer hatten wir eine Laufzeit von $O(n \log^2 n)$ erreicht. Jetzt wenden wir eine Technik an, die in der algorithmischen Geometrie

als *Sweep* bekannt ist. Der Einfachheit halber sei vorausgesetzt, dass alle n Punkte paarweise verschiedene X-und Y-Koordinaten besitzen und dass alle Punktepaare unterschiedliche Abstände haben; dann dürfen wir auch von *dem* dichtesten Paar sprechen.

Wir stellen uns vor, dass wir die Ebene von links nach rechts mit einer senkrechten Geraden L, der sogenannten *Sweepline*, überstreichen. Zu jedem Zeitpunkt wollen wir das dichteste Punktepaar *aller Punkte links von* L kennen. Anfangs liegen genau zwei Punkte links von L und bilden das dichteste Paar. Am Ende liegen alle n Punkte links von L, und unser Problem ist gelöst. Was passiert zwischendurch?

Angenommen, das dichteste Paar aller Punkte links von L hat zur Zeit den Abstand M. Daran kann sich frühestens dann etwas ändern, wenn L auf einen neuen Punkt p trifft. Wenn tatsächlich p zu einem alten Punkt q links von L einen Abstand $< M$ hat, kann q von L nicht weit entfern sein: q muss in einem senkrechten Streifen der Breite M links von L liegen, und zwar in einem Rechteck R der Höhe $2M$ mit p als Mittelpunkt der rechten Kante; siehe Abbildung 6.22.

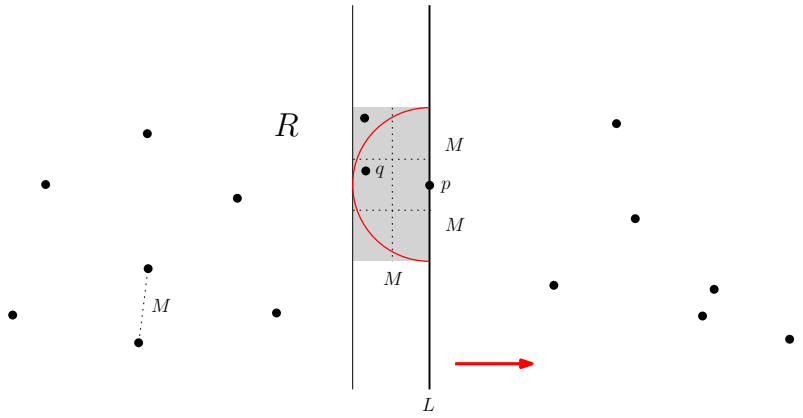

Abbildung 6.22: Wenn p einen Abstand $< M$ zu einem Punkt q links der Sweepline L hat, muss q im Rechteck R liegen.

Erfreulicherweise kann dss Rechteck R nicht beliebig viele Punkte enthalten:

Lemma 13 *Ein Rechteck R mit Kantenlängen $M \times 2M$ kann höchstens 6 Punkte enthalten, die paarweise einen Abstand $\geq M$ haben.*

Beweis. Wir teilen R in sechs gleich große Rechtecke der Größe $\frac{2M}{3} \times \frac{M}{2}$

auf. Die kleinen Rechtecke haben Diagonalen der Länge

$$\sqrt{\left(\frac{2M}{3}\right)^2 + \left(\frac{M}{2}\right)^2} = M\sqrt{\frac{4}{9} + \frac{1}{4}} = M\frac{5}{6} < M,$$

also kann jedes kleine Rechteck höchstens einen solchen Punkte enthalten. □

Der Sweep-Algorithmus funktioniert nun folgendermaßen: Anfangs sortieren wir die gegebenen n Punkte nach X-Koordinaten, in Zeit $O(n \log n)$. Während des Sweeps speichern wir nur die Punkte im senkrechten Streifen der Breite M; dazu verwenden wir einen AVL-Blattsuchbaum T mit verketteten Blättern.

Wenn Sweepline L einen neuen Punkt $p = (p_x, p_y)$ erreicht, löschen wir zunächst alle alten Punkte aus T, die nicht mehr im Streifen liegen, weil ihre X-Koordinaten $< p_x - M$ sind. Dann starten wir eine Bereichsabfrage auf T und berichten alle Punkte, deren Y-Koordinaten im Intervall $[p_y - M, p_y + M]$ liegen; nach Lemma 13 sind das höchstens sechs Stück. Wir berechnen ihre Abstände zu p, prüfen, ob ein Abstand $< M$ darunter vorkommt, verkleinern gegebenenfalls den Streifen durch Löschen weiterer Punkte aus T, aktualisieren M und fügen p als neuen Punkt in T ein.

Jeder der n Punkte wird genau einmal in T eingefügt und genau einmal gelöscht, zusammen in Zeit $O(\log n)$. Die n Bereichsabfragen kosten jeweils $O(\log n + k)$ Zeit, wobei k die Anzahl der berichteten Punkte ist; aber nach Lemma 13 ist $k \leq 6$. Insgesamt haben wir damit folgendes verbesserte Ergebnis:

Theorem 14 *Das dichteste Punktepaar von n Punkten in der Ebene lässt sich in Zeit $O(n \log n)$ und linearem Platz bestimmen.*

Man kann zeigen, dass diese Schranken optimal sind.

6.5 B-Bäume

Wenn die Datenmenge so groß wird, dass sie nicht mehr in den schnellen Hauptspeicher passt, muss sie in langsamere Speicher ausgelagert werden. Hier gilt das Modell der REAL RAM nur noch bedingt, denn die Laufzeit von Algorithmen wird jetzt von der *Anzahl der Zugriffe auf das langsame Speichermedium* dominiert.

Zum Beispiel muss bei externen Festplatten ein Schreib-/Lesekopf auf die richtige Spur bewegt werden; das dauert zwar nur einige Millisekunden, aber doch fast 10^3 mal so lang wie ein Hautpspeicherzugriff. Dafür liefert ein Festplattenzugriff nicht nur einen einzelnen Wert, sondern eine Datenseite (Datenblock) mit Volumen 512 Byte bis 1 KByte.

Diesem Umstand trägt das Modell der *B-Bäume* Rechnung.

Definition 15 Ein *B-Baum der Ordnung k* ist durch folgende Eigenschaften bestimmt:

1. Alle Blätter haben dieselbe Tiefe.
2. Alle Knoten außer der Wurzel haben mindestens k und höchstens $2k-1$ viele Söhne.
3. Die Wurzel hat mindestens 2 und höchstens $2k-1$ viele Söhne.

Wenn h die Höhe eines solchen B-Baums bezeichnet und n die Anzahl seiner Blätter, so gilt

$$2 \cdot k^{h-1} \;\leq\; n \;\leq\; (2k-1)^h;$$

die untere Schranke wird erreicht, wenn die Wurzel nur 2 und alle anderen internen Knoten nur k Söhne haben, die obere, wenn alle Knoten maximal viele, nämlich $2k-1$, Kinder besitzen. Daraus folgt

$$\log_{2k-1} n \;\leq\; h \;\leq\; 1 + \log_k \frac{n}{2} \tag{6.9}$$

Abbildung 6.23 zeigt, wie ein Knoten eines B-Suchbaums aufgebaut ist.

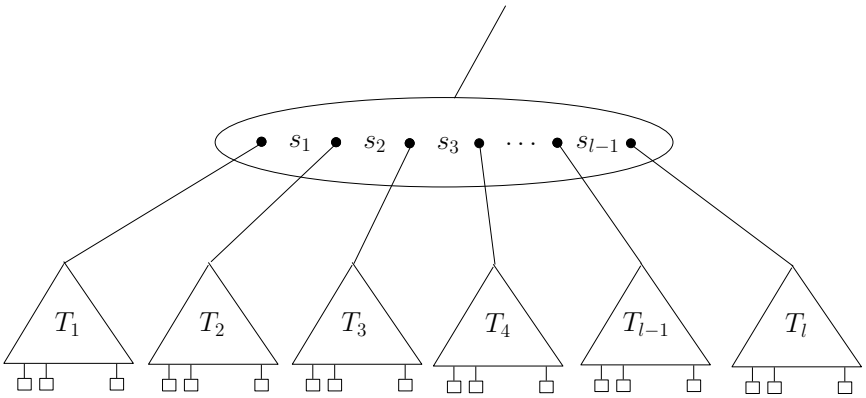

Abbildung 6.23: Im Knoten eines B-Baums wechseln Zeiger auf Teilbäume und Schlüssel einander ab. Alle Schlüssel in T_i sind kleiner als s_i, und s_i ist kleiner als die Schlüssel in T_{i+1}.

Schlüssel stehen nur in den inneren Knoten, zwischen den Zeigern auf die Teilbäume. Sie sind aufsteigend sortiert: Für alle i gilt

$$\text{Schlüssel von } T_i \;<\; s_i \;< \text{Schlüssel von } T_{i+1}.$$

Der Parameter k wird meist so gewählt, dass ein maximal voller Knoten mit $2k-1$ Zeigern und $2k-2$ Schlüsseln gerade auf eine Datenseite passt. Wegen

der unteren Schranke k für die Anzahl der Söhne ist dann jede Datenseite (außer der Wurzel) zumindest halb voll.

Beim *Suchen* eines Schlüssels x startet man im Wurzelknoten und durchläuft von links nach rechts die dort gespeicherten Schlüssel $s_1, \ldots, s_{l-1}$, bis man zum letzten Schlüssel s_i mit $s_i \leq x$ gelangt. Gilt $s_i = x$, so ist man fertig. Andernfalls setzt man die Suche im Teilbaum T_{i+1} fort.

Um einen Schlüssel x *einzufügen*, wird x zunächst gesucht. Kommt x bereits vor, so wird eine Meldung generiert. Andernfalls endet die Suche erfolglos in einem Blatt b, das in seinem Vaterknoten v hinter einem Schlüssel $s_i < x$ angehängt ist. Wir fügen x als neuen Schlüssel hinter s_i in die Liste ein und ein neues Blatt b' hinter x. Wenn Knoten v jetzt immer noch $\leq 2k-1$ viele Söhne hat, ist man fertig. Wenn nicht, hat v jetzt genau $2k$ viele Söhne und wird in zwei gleich große Knoten v_1 und v_2 aufgeteilt; dabei wandert der k-te Schlüssel (nach Einfügen von x) in den Vater v' von v, siehe Abbildung 6.24. Möglicherweise ist jetzt auch Knoten v' überfüllt und muss geteilt werden. Das kann sich bis zur Wurzel w fortpflanzen; in dem Fall wird w in zwei Knoten aufgeteilt und eine neue Wurzel darübergesetzt, die genau diese zwei Söhne hat.

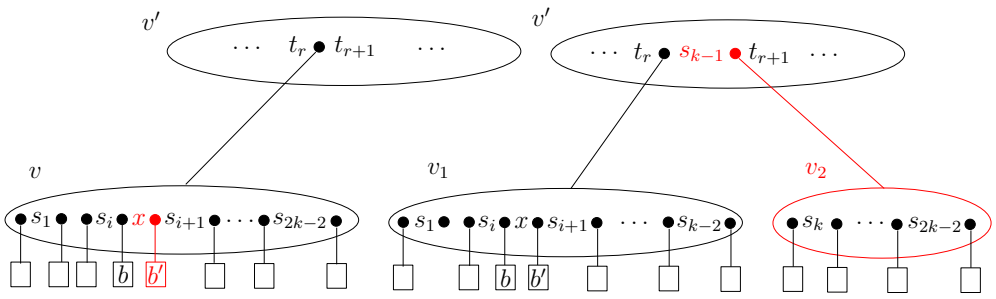

Abbildung 6.24: Knoten v wird geteilt, wenn er nach Einfügen von Schlüssel x und einem neuen Blatt b' nun $2k$ viele Söhne hat.

Auch beim *Entfernen* eines Schlüssels x wird dieser zunächst gesucht. Falls x nicht vorkommt, wird eine Meldung generiert. Andernfalls steht x in einem Knoten v hinter dem Zeiger auf einen Teilbaum T_j. Im ersten Fall sind T_j und alle anderen Söhne von v Blätter. Dann werden T_j und x einfach entfernt. Dabei kann es passieren, dass v nur noch $k-1$ Söhne bleiben; wir überlegen uns gleich, was dann zu tun ist. Im zweiten Fall hat v echte Teilbäume als Söhne. Dann sei s der größte Schlüssel in T_j, enthalten in einem

Knoten w. Rechts von s steht in w nur noch der Zeiger auf ein Blatt b; siehe Abbildung 6.25. Schlüssel s ist der Vorgänger von x im Baum. Darum ersetzen wir im Knoten v den Schlüssel x durch s und entfernen dann im Knoten w den Schlüssel s und das Blatt b (so, als hätten wir von Anfang an s entfernen wollen, wie im ersten Fall). Auch hier kann es passieren, dass w nur $k-1$ Söhne behält.

Dann wird w mit einem Nachbarknoten u mit $l \geq k$ Söhnen vereinigt; das ist gerade die Umkehrung der Knotenteilung beim Einfügen. Falls $l > k$ war, können wir den Vereinigungsknoten gleich wieder in zwei Knoten mit jeweils $\geq k$ vielen Söhnen teilen und sind nach dieser Umverteilung von Schlüsseln fertig. Andernfalls hat der Vater w' von w und u durch die Vereinigung einen Sohn verloren und könnte jetzt selbst unterbelegt sein. Dann muss w' mit einem Nachbarknoten vereinigt werden, und so fort. Sollte die Wurzel nur noch einen Sohn haben, wird sie entfernt, und der Sohn wird zur neuen Wurzel.

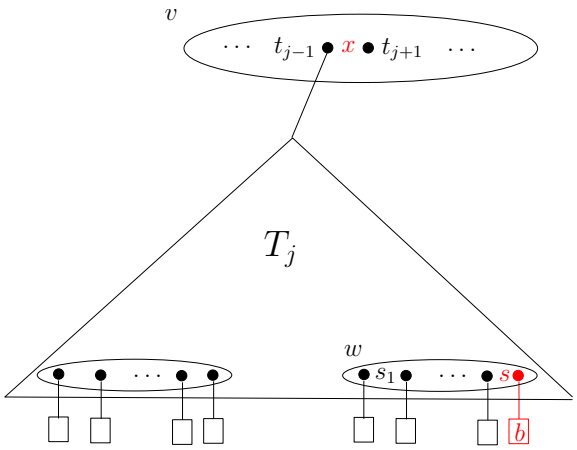

Abbildung 6.25: Der zu entfernende Schlüssel x wird durch seinen Vorgänger s ersetzt. Blatt b wird entfernt.

Bei den Operationen Suchen, Einfügen und Entfernen erfordert jeder Knotenbesuch einen Externzugriff auf eine Datenseite. Beim Einfügen läuft man erst längs des Suchpfads zu einem Blatt hinab; dann kann es nötig sein, zur Wurzel hinaufzulaufen und dabei jeden Knoten zu teilen. Mit der Abschätzung (6.9) für die Höhe eines B-Baums erhalten wir also folgendes Ergebnis:

Theorem 16 *In einem B-Baum der Ordnung k lassen sich Suchen, Einfügen und Entfernen mit maximal*

$$3 \log_k n + 1 \in O(\log_k n)$$

Externzugriffen ausführen, wobei n die Anzahl der Schlüssel bedeutet.

Für $k = 2$ sind B-Bäume auch als 2-3-Bäume bekannt und können als Alternative zu AVL-Bäumen als interne Datenstrukturen im Hauptspeicher verwendet werden.

6.6 Mittelwertbildung

Bevor wir weitere Datenstrukturen betrachten, wollen wir im Hinblick auf spätere Anwendungen verschiedene Arten der Mittelwertbildung diskutieren, die bei der Analyse vorkommen können, und jeweils typische Anwendungen untersuchen.

6.6.1 Amortisierte Kosten: Berechnung der konvexen Hülle

Manchmal muss eine Folge von n gleichartigen Aktionen ausgeführt werden, von denen einige hohe Kosten verursachen, andere aber nur geringe. Dann ist es vernünftig, die Gesamtkosten aller Aktionen zu betrachten und durch n zu teilen, um ein Maß für die mittleren Kosten pro Aktion zu bekommen. Dabei kann eine teure Aktion als "Investition in die Zukunft" angesehen werden, die sich später durch viele Aktionen mit niedrigen Kosten *"amortisiert"*.

Als Beispiel betrachten wir einen Algorithmus zur Konstruktion der *konvexen Hülle* von n Punkten in der Ebene, also der (bezüglich Inklusion) kleinsten Menge, die die gegebenen Punkte enthält und mit je zwei beliebigen Punkten a und b auch das Liniensegment $\overline{ab}$ von a nach b. Ihren Rand kann man sich als ein gespanntes Gummiband vorstellen, das die gegebenen Punkte umschließt; siehe Abbildung 6.26 (i).

Zu Beginn werden die n Punkte nach aufsteigenden X-Koordinaten sortiert; der Einfachheit halber wird angenommen, dass diese Koordinaten paarweise verschieden sind und dass außerdem keine drei Punkte auf einer Geraden liegen. Jetzt verfährt man inkrementell. Die konvexe Hülle von p_1, p_2, p_3 ist einfach das Dreieck mit diesen Eckpunkten. Angenommen, die konvexe Hülle C_i von $p_1, \ldots, p_i$ ist schon konstruiert, und p_{i+1} soll nun als weiterer Punkt hinzugenommen werden. Wegen der Sortierung liegt C_i links von der Senkrechten durch p_i, und p_{i+1} liegt rechts davon; siehe Abbildung 6.26 (ii). Wir starten bei p_i und durchlaufen den Rand von C_i nach oben und unten, bis die beiden Tangentenpunkte v und w gefunden sind. Unterwegs muss man bei jedem Eckpunkt p testen, ob die Halbgerade von p_{i+1} durch p die konvexe Hülle C_i berührt oder schneidet; das geht in konstanter Zeit, weil man nur die beiden Kanten von C_i betrachen muss, die sich in p treffen.

Wenn v und w gefunden sind, ergibt sich die neue konvexe Hülle C_{i+1}, indem man die Segmente $\overline{vp_{i+1}}$ und $\overline{wp_{i+1}}$ hinzufügt und das Randstück von C_i

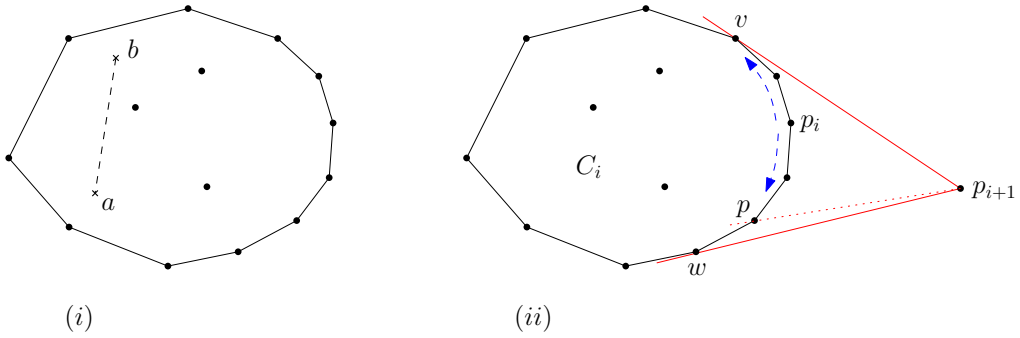

Abbildung 6.26: (i) Die konvexe Hülle von Punkten in der Ebene. (ii) Punkt p_{i+1} kommt hinzu.

zwischen v und w, welches p_i enthält, entfernt.

Diese Aktion kann Kosten in $O(n)$ verursachen, wenn zwischen v und w viele Eckpunkte p liegen, die besucht werden müssen. Aber alle diese Eckpunkte werden anschließend entfernt und kommen nie wieder vor, weil sie jetzt innerhalb von C_{i+1} liegen. Also betragen die Gesamtkosten aller Einfügeoperationen $O(n)$, weil es ja nur n Punkte gibt, und eine einzelne Operation braucht im Mittel nur konstante Zeit. Damit haben wir folgendes Ergebnis:

Theorem 17 *Die konvexe Hülle von n Punkten in der Ebene lässt sich nach Sortieren der Punkte in Zeit $O(n)$ konstruieren, insgesamt also in Zeit $O(n \log n)$.*

Der Speicherplatzbedarf ist linear, wenn man die konvexen Hüllen als verkettete Listen verwaltet. Beide Schranken sind optimal. Der oben vorgestellte Algorithmus ist sowohl ein Beispiel für *Inkrementelle Konstruktion* als auch für *Sweep*, je nach Betrachtungsweise.

6.6.2 Randomisierter Input: Bucketsort

Bei manchen Algorithmen hängt die Laufzeit stark von der Eingabe ab. Dann kann es sinnvoll sein, die *mittlere Laufzeit* zu betrachten. Dazu muss man wissen, mit welcher Wahrscheinlichkeit jede mögliche Eingabe an die Reihe kommt.

Betrachten wir als Beispiel den Algorithmus *Bucketsort*, einen einfachen Verwandten vom Radixsort aus Kapitel 2. Um n reelle Zahlen x aus dem Intervall $[0, 1)$ zu sortieren, werden zunächst alle Eingabewerte in denselben Behälter (Bucket) gelegt, die im gleichen Intervall $[\frac{i}{n}, \frac{i+1}{n})$ liegen, für $i = 0, 1, 2, \ldots, n-1$. Nun werden die Zahlen eines jeden Behälters einzeln

sortiert, zum Beispiel mit einem quadratischen Verfahren wie Insertionsort. Schließlich werden die sortierten Behälterinhalte von links nach rechts ausgegeben.

Offensichtlich braucht Bucketsort Zeit $\Omega(n^2)$, wenn alle n Eingabewerte im selben Intervall $[\frac{i}{n}, \frac{i+1}{n})$ liegen, denn dann wird die gesamte Arbeit von der quadratischen Prozedur Insertionsort geleistet. Allgemein hat Bucketsort die Laufzeit

$$T(n) \;=\; d\,n \;+\; c\sum_{i=0}^{n-1} B_i^2,$$

wobei B_i die Anzahl der Eingabewerte im Intervall $[\frac{i}{n}, \frac{i+1}{n})$ bedeutet; dabei misst $d\,n$ den Aufwand für die Aufteilung in Buckets.

Die Summe der Quadrate der B_i entspricht genau der Anzahl aller Paare (x, y) von Eingabewerten, bei denen x und y im selben Intervall liegen. Das sind alle Paare (x, x), und außerdem gewisse Paare (x, y) mit $x \neq y$. Wenn man nun annehmen darf, dass die Eingabewerte gleichverteilt und unabhängig aus dem Intervall $[0, 1)$ gezogen werden, dann liegt zu gegebenem x der Wert y genau mit Wahrscheinlichkeit $\frac{1}{n}$ im gleichen Intervall wie x, denn die Intervalle haben alle die gleiche Länge $\frac{1}{n}$, und die Ziehung von y hängt nicht von der Ziehung von x ab. Für den Erwartungswert der Laufzeit ergibt sich demnach

$$\begin{aligned} E(T(n)) &= d\,n \;+\; c\Big(\sum_x 1 \;+\; \sum_x \sum_{y\neq x} \frac{1}{n}\Big) \\ &= d\,n \;+\; c\Big(n \;+\; n(n-1)\,\frac{1}{n}\Big) \\ &< (d+2c)\,n \end{aligned}$$

aus der Linearität des Erwartungswerts. Also gilt:

Theorem 18 *Bei gleichverteilten und unabhängig gezogenen Eingabewerten aus* $[0, 1)$ *hat Bucketsort die mittlere Laufzeit* $O(n)$.

Auch bei Bucketsort finden, wie schon bei Radixsort, nicht nur Schlüsselvergleiche statt, denn die Eingabewerte x müssen ja ihren Intervallen zugeordnet werden. Um die Bedingung $\frac{i}{n} \leq x < \frac{i+1}{n}$, oder äquivalent: $i \leq nx < i + 1$, testen zu können, wird die Rundungsfunktion $\lfloor nx \rfloor$ benötigt. Daher widerspricht Theorem 18 nicht der unteren Schranke in Theorem 2.

In manchen Fällen ist man nicht daran interessiert, die Laufzeit über alle möglichen Eingaben zu mitteln, sondern nur in der Umgebung besonders "schwieriger" Inputs; so kann man Aufschluss darüber gewinnen, ob es sich um singuläre Ausreißer handelt.

6.6.3 Randomisierter Algorithmus: Bestimmung des Maximums

In den meisten Fällen ist nicht bekannt, welcher Wahrscheinlichkeitsverteilung die Eingaben unterliegen; dann muss ein Algorithmus mit den konkreten Eingabewerten arbeiten, die er bekommt. Trotzdem kann er Zufall ins Spiel bringen, indem er würfelt, welche von mehreren möglichen Aktionen als nächste ausgeführt werden soll.

Wir beginnen mit einem ganz einfachen Beispiel. Um das Maximum von n Schlüsseln a_i zu bestimmen, verfährt man wie folgt: Eine Variable max wird zunächst auf a_1 gesetzt. Dann wird für $i = 2$ bis n getestet, ob $\max < a_i$ gilt, und gegebenfalls das bisherige Maximum durch $\max := a_i$ aktualisiert. Das erfordert $n-1$ Tests und schlimmstenfalls $n-1$ Aktualisierungen, wenn die Schlüssel aufsteigend sortiert sind.

Angenommen, ein Test kostet Zeit T, eine Aktualisierung aber Zeit $A >> T$, weil dabei Daten bewegt werden müssen. Dann würde man gern die Anzahl der Aktualisierungen begrenzen. Ein naheliegender Ansatz besteht darin, die gegebenen n Schlüssel zunächst in eine zufällige Reihenfolge zu bringen und dann obigen Algorithmus zu starten.

Wie wahrscheinlich ist es nun, dass die Aktualisierung $\max := a_i$ ausgeführt werden muss? Das geschieht genau dann, wenn a_i der größte Schlüssel in $A_i := \{a_1, a_2, \ldots, a_{i-1}, a_i\}$ ist. Die Wahrscheinlichkeit hierfür beträgt $\frac{1}{i}$, denn nach der anfänglichen Zufallspermutation ist a_i eine beliebige Zahl in der i-elementigen Menge A_i. Also hat der randomisierte Algorithmus die zu erwartenden Kosten

$$
\begin{aligned}
E(T(n)) &= (n-1)\cdot T \;+\; \sum_{i=2}^{n} A \cdot \frac{1}{i} && (6.10)\\
&< (n-1)\cdot T \;+\; A \cdot \int_1^n \frac{1}{x}\,dx && (6.11)\\
&< n\cdot T \;+\; \ln(n)\cdot A, && (6.12)
\end{aligned}
$$

was deutlich besser ist als der Worst Case $n \cdot (T + A)$. Die Abschätzung der Summe durch das Integral ergibt sich durch Vergleich der Flächen unter den Funktionen.

6.6.4 Randomisierter Algorithmus: Quicksort

Eine einfache und weit verbreitete Variante des Sortierens mit Schlüsselvergleichen funktioniert folgendermaßen: Gegeben ist ein Array A mit n Schlüsseln $a_1, \ldots, a_n$, die sortiert werden sollen. Eine dieser Zahlen wird als sogenanntes *Pivotelement* p ausgewählt. Dafür gibt es verschiedene Möglichkeiten; zum Beispiel könnte man einfach $p := a_n$ setzen. Jetzt werden alle

$a_i < p$ in einer Folge L zusammengefasst, und alle $a_j > p$ in einer Folge R; das geht in Zeit $O(n)$. Nun wendet man Quicksort rekursiv auf L und R an, schreibt die sortierten Teilfolgen hintereinander, dazwischen p, und ist fertig. (In der Literatur wird beschrieben, wie man direkt im Array A die Folge L links von p anordnet und R rechts von p, ohne zusätzlichen Speicherplatz zu benutzen.)

Die Effizienz von Quicksort hängt von der Größe von L und R ab und damit von der Wahl des Pivotelements p. Im besten Fall sind L und R stets gleich groß. Dann ergibt sich für die Laufzeit die Rekursion $T(n) = 2T(\frac{n}{2}) + c\,n$, woraus $T(n) \in O(n \log n)$ folgt; vergleiche Kapitel 3. Tatsächlich könnte man in Zeit $O(n)$ den sogenannten *Median* von $a_1, \ldots, a_n$ bestimmen, also das Element, welches in der Mitte der sortierten Folge liegt, und damit $|L| \approx |R|$ garantieren. Der Median-Algorithmus ist aber selbst rekursiv, und die Konstante in der $O(n)$-Abschätzung recht hoch.

Der schlimmste Fall für Quicksort tritt ein, wenn die Eingabefolge bereits aufsteigend sortiert ist. Wir müssen dann $O(n)$ Zeit aufwenden, um festzustellen, dass die ersten $n-1$ Elemente kleiner als $p = a_n$ sind. Dann enthält L diese $n-1$ Elemente, und R ist leer. In diesem Fall braucht Quicksort quadratische Laufzeit.

Es liegt deshalb nahe, das Pivotelement p *zufällig* in $a_1, \ldots, a_n$ auszuwählen. Dann ist jede Position von p in der sortierten Folge gleich wahrscheinlich, und für die mittlere Laufzeit ergibt sich die Rekursion

$$\begin{aligned} E(T(n)) &= c\,(n+1) + \frac{1}{n}\sum_{i=1}^{n}\Big(E(T(i-1)) + E(T(n-i))\Big) && (6.13)\\ &= c\,(n+1) + \frac{2}{n}\sum_{i=0}^{n-1} E(T(i)), && (6.14) \end{aligned}$$

denn bei beiden Summanden läuft der Index von 0 bis $n-1$. Zur Abkürzung sei $e_i := E(T(i))$. In der Rekursion (6.14) können wir die Summation durch Differenzenbildung eliminieren und bekommen die Gleichungen

$$\begin{aligned} e_{n+1}(n+1) - e_n\,n &= c\Big((n+2)(n+1) - (n+1)n\Big) + 2\sum_{i=0}^{n} e_i - 2\sum_{i=0}^{n-1} e_i \\ &= 2\,c\,(n+1) + 2\,e_n \\ (n+1)\,e_{n+1} &= 2\,c\,(n+1) + (n+2)\,e_n \\ \frac{e_{n+1}}{n+2} &= \frac{2c}{n+2} + \frac{e_n}{n+1} \end{aligned}$$

Aus der letzten Zeile ergibt sich durch iteriertes Einsetzen

$$\begin{aligned} \frac{e_{n+1}}{n+2} &= 2\,c\left(\frac{1}{n+2} + \frac{1}{n+1} + \ldots + \frac{1}{2}\right) + \frac{e_0}{1} \\ &< 2\,c\,\ln(n+2) \end{aligned}$$

wie in (6.12), wobei $e_0 = 0$ ist. Damit haben wir folgendes Ergebnis:

Theorem 19 *Die mittlere Laufzeit von Quicksort, angesetzt auf eine beliebige Eingabe von n Schlüsseln, liegt in $O(n \log n)$.*

Quicksort und Mergesort sind beides Algorithmen vom Typ Divide-and-Conquer. Quicksort leistet die Hauptarbeit bei der Aufteilung in Teilmengen, Mergesort beim Zusammenfügen der Ergebnisse.

Im nächsten Kapitel werden wir Hash-Verfahren untersuchen, bei denen sowohl randomisierte Eingaben als auch amortisierte Kosten eine Rolle spielen.

6.7 Hashing

Wir haben gesehen, dass sich balancierte Bäume sehr gut dazu eignen, sortierte Schlüssel zu speichern. Für einen gegebenen Schlüssel findet sich so der zugehörige Datensatz in $O(\log n)$ Zeit. Das Einfügen und Entfernen eines neuen Objektes mit neuem Schlüssel hat die gleiche Zeitkomplexität.

Wie man sich leicht überlegt, gelten die gleichen Laufzeiten auch für die *mittlere* Laufzeit dieser Operationen. Wenn wir also eine Folge von n Operationen betrachten und $K(i)$ die Kosten der i-ten Operation beschreibt, so erhalten wir durch

$$\frac{1}{n} \sum_{i=1}^{n} K(i)$$

die mittleren Kosten der Operationen.

Für das Einfügen und Entfernen liegen die mittleren Kosten in jedem Fall in $\Theta(\log n)$, da bei jedem Einfügen und bei jedem Entfernen die Mindesttiefe $O(\log n)$ aus dem Beweis von Lemma 8 zu durchlaufen ist und andererseits die Kosten nach den Betrachtungen des vorherigen Kapitels stets in $O(\log n)$ liegen.

Für Datenbanken mit sehr großen Datenmengen verzichtet man beim erwähnten **Wörterbuch-Problem** auf die sortierte Speicherung der Schlüssel. Stattdessen sollen die Operationen (Exakte) Suche, Einfügen und Entfernen im Mittel nur *konstante* Zeit verursachen.

In solchen Fällen kommt das sogenannte *Hashing* zur Anwendung. Gegeben ist also eine Menge von Objekten (Daten) die über einen eindeutigen Schlüssel (ganze Zahl) identifizierbar sind und es wird eine Struktur zur Speicherung gesucht, die mindestens die Operationen

- (Exakte) Suche eines Objektes mit Schlüssel x
- Einfügen eines Objektes mit Schlüssel x

- Entfernen eines Objektes mit Schlüssel x

effizient (im Mittel konstant) ausgeführt werden können. Dafür wird das Schlüsseluniversum partitioniert. Die Position des Datenelementes im Speicher ergibt sich durch eine Berechnung aus dem Schüssel.

Formal steht uns für die Verwaltung einer ungeordneten Menge S von Schlüsseln zu zur Verfügung:

1. ... ein Feld $T[0..m-1]$ für m Behälter $T[0], T[1], \ldots, T[m-1]$ die sogenannte *Hashtabelle* und
2. ... eine *Hash-Funktion* $h : U \to [0..m-1]$ die das Universum der Schlüssel auf das Feld T abbildet

Ziel ist es, für eine Schlüsselmenge $S \subseteq U$ für jedes Element $x \in S$ das zugehörige Datenobjekt in $T[h(x)]$ zu speichern. Im Folgenden lassen wir der Übersichtlichkeit halber die Datenobjekte weg und Speichern lediglich die Schlüssel in der Hashtabelle.

Beispiel: Sei $m = 5$ und $S = \{8, 22, 16, 29\}$ und $h(x) = x \mod 5$. Die Hashtafel sieht dann wie folgt aus:

0	
1	16
2	22
3	8
4	29

Falls wir hier einen weiteren Schlüssel 21 verwenden gilt $h(21) = 1$ und wir sprechen dann von einer *Kollision*. Für eine Menge S von n Schlüsseln und für eine Menge von m Behältern $T[0], \ldots, T[m-1]$ mit $n \leq m$ heißt die Hashfunktion h *perfekt für* S falls es keine Kollisionen bei der Zuordnung gibt. Die Hashfunktion soll effizient ausgewertet werden können.

6.7.1 Kollisionsbehandlung mit verketteten Listen

Eine sehr einfache Lösung in der Kollisionsbehandlung ist die Verwendung verketteter Listen in den Einträgen $T[i]$ der Hashtabelle. Neue Einträge werden dann an den Anfang der Liste gesetzt und somit kann nach Auswertung der Hashfunktion in kostanter Zeit ein neuer Schlüssel eingefügt werden.

Beispiel: Wie betrachten die Schlüsselmenge $S = \{10, 22, 16, 29, 17, 3\}$ für $m = 3$ und die Hashfunktion $h(x) = x \mod 3$. Die Abbildung beschreibt die Hashtabelle nach dem sukzessiven Einfügen der Schlüssel mit Kollisionsvermeidung mittels verketteter Listen.

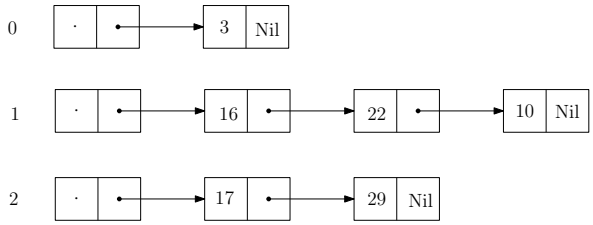

Abbildung 6.27: Die Kollisionsbehandlung durch die Verwendung von verketteten Listen $S = \{10, 22, 16, 29, 17, 3\}$ und $h(x) = x \mod 3$.

Wir wollen nun die Verwendung von verketteten Listen für die Kollisionsbehandlung analysieren und dabei die mittleren Kosten berücksichtigen. Die Kosten einer Operation (Exakte) Suche und Entfernen ergeben sich jeweils durch das Berechnen des Hashwertes und den anschließenden linearen Suchdurchlauf durch die Liste des entsprechenden Eintrags.

Die Funktion $\delta_h(x, S)$ bezeichnet die Anzahl der Elemente $y \in S$ für die Funktion $h(x) = h(y)$ gilt. Dann kann eine Operation (Exakte) Suche und Entfernen für einen Schlüssel x in Zeit $O(1 + \delta_h(x, S))$ ausgeführt werden.

Wir betrachten dabei auch eine Folge von n möglichen zufälligen Operationen (Exakte) Suche, Einfügen und Entfernen und wollen die gesamte Laufzeit dafür nach oben Abschätzen.

Wir machen dafür die folgende Annahmen:

1. Die Hashfunktion $h : U \to [0..m]$ streut das Universum gleichmäßig über das Intervall $[0..m]$. Das ist zum Beispiel für $h(x) = x \mod m$ gegeben.

2. Sämtliche Elemente des Universums sind mit gleicher Wahrscheinlichkeit Argument der nächsten Operation.

Falls nun x_k der Schlüssel ist, der in der k-ten Aktion verwendet wird, dann folgt nun, dass die Wahrscheinlichkeit, dass $h(x_k) = i$ ist für $i \in [0..m]$ durch $\frac{1}{m}$ gegeben ist, kurz $\text{prob}(h(x_k) = i) = \frac{1}{m}$.

Theorem 20 *Für die einzelnen Operationen (Exakte) Suche, Einfügen und Entfernen bezüglich einer Hashtabelle für die Schlüsselmenge S wird im worstcase $\Omega(|S|)$ Zeit benötigt.*

Unter den obigen Annahmen ist der Erwartungswert für die von einer Folge von n Operationen (Exakte) Suche, Einfügen und Entfernen verursachten Kosten kleiner gleich $\left(1 + \frac{\alpha}{2}\right) n$, wobei $\alpha = \frac{n}{m}$ der maximale Belegungsfaktor der Hashtafel ist.

Beweis. Der erste Teil des Theorems ist leicht zu sehen. Im schlechtesten Fall gilt für alle Schlüssel $x \in S$, dass $h(x)$ auf den gleichen Wert i_0 abbildet, dann muss im schlimmsten Fall bei jeder Operation eine Liste der größe $|S|$ vollständig durchlaufen werden.

Für den zweiten Teil der Behauptung verwenden wir für die ersten k Operationen und für $i = 1, \ldots, m$ einen Zähler $l_k(i)$, so dass $l_k(i)$ nach k Operationen in jedem Fall einen Wert größer gleich der dann aktuellen Listenlänge der Liste von $T_k[i]$ hat. Falls nun eine weitere Operation die i-te Liste betrifft, wird $l_{k+1}(i) := l_k(i) + 1$ gesetzt und $l_k(i)$ ist dann stets eine obere Schranke der aktuellen Listenlänge nach k Operationen.

Anfangs sind alle Zähler auf 0 gesetzt. Immer wenn eine Operation die i-te Liste betrifft, erhöhen wir den Wert von $l_k(i)$ um Eins, unabhängig von der Art der Operation. Sei EK_{k+1} der Erwartungswert des Aufwandes der $(k+1)$-ten Operation bei insgesamt n Operationen. Nehmen wir an, dass x_{k+1} der Schlüssel der $(k+1)$-ten Operation ist, und es gelte $h(x_{k+1}) = i$. Sei nun $\mathrm{prob}(l_k(i) = j)$ die Wahrscheinlichkeit, dass der Zähler $l_k(i)$ nach der k-ten Operation den Wert j hat. Dann ist

$$\mathrm{EK}_{k+1} = 1 + \sum_{j=0}^{k} \mathrm{prob}(l_k(i) = j) \cdot j \,,$$

da wir den Schlüssel auswerten müssen und relativ zur Wahrscheinlichkeit der Größe von $l_k(i) = j$ die entsprechende Länge j durchlaufen müssen.

Für $j \geq 1$ gilt nun aber

$$\mathrm{prob}(l_k(i) = j) = \binom{k}{j} \left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{k-j} .$$

Diese Aussage läßt sich leicht kombinatorisch erklären. Es gibt insgesamt $\binom{k}{j}$ j-elementige Sequenzen aus den ersten k-Operationen die zu beachten sind. Für jede dieser Sequenzen ist die Wahrscheinlichkeit, dass genau j-Operationen die i-te Liste betreffen $\left(\frac{1}{m}\right)^j \left(1 - \frac{1}{m}\right)^{k-j}$. Genauer, das Produkt der jeweiligen Wahrscheinlichkeit $\frac{1}{m}$, dass eine einzelne der j-Operation die i-te Liste betrifft mal dem Produkt der jeweiligen Wahrscheinlichkeit $\left(1 - \frac{1}{m}\right)$, dass eine einzelne der restlichen $k-j$-Operationen *nicht* die i-te Liste betrifft.

Zunächst berechnen wir nun EK_{k+1}:

$$
\begin{aligned}
\mathrm{EK}_{k+1} &= 1 + \sum_{j=0}^{k} \binom{k}{j} \left(\frac{1}{m}\right)^{j} \left(1 - \frac{1}{m}\right)^{k-j} \cdot j \\
&= 1 + \frac{k}{m} \sum_{j=1}^{k} \binom{k-1}{j-1} \left(\frac{1}{m}\right)^{j-1} \left(1 - \frac{1}{m}\right)^{k-j} \\
&\qquad \left(\tfrac{k}{j} \text{ und } \tfrac{1}{m} \text{ Ausklammern}\right) \\
&= 1 + \frac{k}{m} \left(1 - \frac{1}{m}\right)^{-1} \sum_{j=1}^{k} \binom{k-1}{j-1} \left(\frac{1}{m}\right)^{j-1} \left(1 - \frac{1}{m}\right)^{k-(j-1)} \\
&\qquad \left(\left(1 - \tfrac{1}{m}\right)^{-1} \text{ Ausklammern}\right) \\
&= 1 + \frac{k}{m} \frac{m}{m-1} \left(\frac{1}{m} + \left(1 - \frac{1}{m}\right)\right)^{k-1} \\
&\qquad \text{(Binominaltheorem)} \\
&= 1 + \frac{k}{m-1}
\end{aligned}
$$

Nun kann der Erwartungswert aller n Operationen, $\mathrm{EK}(n)$, folgendermaßen abgeschätzt werden.

$$
\begin{aligned}
\mathrm{EK}(n) &= \sum_{k=1}^{n} \mathrm{EK}_k = \sum_{k=0}^{n-1} \mathrm{EK}_{k+1} \\
&= \sum_{k=0}^{n-1} \left(1 + \frac{k}{m-1}\right) \\
&= n + \frac{1}{m-1} \sum_{k=0}^{n-1} k = n + \frac{1}{m-1} \frac{n(n-1)}{2} \\
&= n \cdot \left(1 + \frac{n-1}{2(m-1)}\right) < \left(1 + \frac{n}{2m}\right) \cdot n
\end{aligned}
$$

Dabei nehmen wir für den letzten Schritt $n < m$ an. Somit gilt für $\alpha = \frac{n}{m}$, dass $\left(1 + \frac{\alpha}{2}\right) n$ eine obere Schranke für den erwarteten Aufwand der n Operationen darstellt. □

Falls $\alpha = \frac{n}{m}$ konstant bleibt, sagt dass obige Theorem, dass im Mittel nur konstant viel Aufwand zu erwarten ist, also eine deutliche Verbesserung im Vergleich zur AVL-Baumlösung.

Abschließend erwähnen wir noch, wie sich die Bedingung, dass $\alpha = \frac{n}{m}$ konstant bleibt, gewährleisten lässt, ohne an Performanz zu verlieren.

Wir passen das m sukzessive an. Dazu verwenden wir eine Folge von Hashtafeln $T_0, T_1, T_2, \ldots, T_i, \ldots$ der Größen $m, 2m, \ldots, 2^i m, \ldots$ mit Hashfunktionen $h_i : U \to [0..2^i m - 1]$. Die Tafeln werden so verwendet, dass der aktuelle Belegungsfaktor α stets strikt zwischen $\frac{1}{4}$ und 1 liegt.

Die Tabelle T_i muss dann umkopiert werden, wenn der Belegungsfaktor aus dem Intervall herausfällt:

α wird zu 1: Wir speichern alle $2^i m$ Elemente von T_i nach T_{i+1} in $\Theta(2^i m)$ Zeit. Der Belegungsfaktor von T_{i+1} ist dann $\frac{1}{2}$. Bis zur nächsten Umspeicherung müssen mindestens $\frac{1}{4}2^{i+1}m$ Operationen durchgeführt werden bevor die nächste Umspeicherung notwendig wird.

α wird zu $\frac{1}{4}$: Wir können alle $\frac{1}{4}2^i m$ Elemente in $\Theta(\frac{1}{4}2^i m)$ Zeit von T_i nach T_{i-1} speichern. Der Belegungsfaktor von T_{i-1} ist dann $\frac{1}{2}$ und mindestens $\frac{1}{4}2^{i-1}m$ Operationen können durchgeführt werden bevor die nächste Umspeicherung notwendig wird.

In beiden Fällen sind die Kosten für die Umspeicherung maximal zweimal so groß, wie die Anzahl der Operationen. Die Kosten für das Umspeichern verteilen wir deshalb auf die einzelnen Operationen. Der Erwartungswert $O(1)$ für die mittleren Kosten bleibt dann erhalten. Der Aufwand amortisiert sich hier über die Zeit. Auch wenn gelegentlich viele Elemente umkopiert werden müssen, so ist der Aufwand im Mittel gering.

6.8 Union-Find Datenstruktur

Neben der Speicherung einer Menge von Objekten ist es manchmal nützlich, ein System von Mengen zu speichern. Die Union-Find Datenstruktur dient zur Verwaltung einer Menge von disjunkten Mengen $S_1, \ldots, S_k$. Ein Universum $U = \{x_1, x_2, \ldots, x_n\}$ von Elementen ist gegeben und wir haben anfangs n disjunkte Mengen $S_i = \{x_i\}$. Über ein Array $U[1..n]$ können wir stets auf die Elemente durch $U[i] = x_i$ zugreifen. Jede Menge besitzt einen eindeutigen Repräsentanten $s_i \in S_i$. Im Folgenden beschreiben wir die Mengen der Einfachheit halber über die Indizes des Arrays U, das heißt $S_i = \{i\}$ und auch $s_i = i$.

Die Datenstruktur soll die folgenden Operationen effizient unterstützen:

- UNION(x, y): Falls zwei Mengen S_x und S_y mit $x \in S_x$ und $y \in S_y$ existieren, so werden diese entfernt und durch die Menge $S_x \cup S_y$ ersetzt. Der neue Repräsentant von $S_x \cup S_y$ kann ein beliebiges Element dieser vereinigten Menge sein.

- FIND(x): Liefert den Repräsentanten der Menge S mit $x \in S$ zurück.

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

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

Intern kann eine solche Struktur effizient wie folgt realisiert werden. Eine einzelne Menge S_i wird durch eine verkettete Liste dargestellt, wobei aber jedes einzelne Listenelement auch einen Zeiger auf den Kopf S_i der Liste speichert. Der Kopf der Liste enthält den Namen S_i, den Repräsentanten i und die Größe der Liste. Zusätzlich verwenden wir ein Feld $U[1..n]$ dass für jedes Element j durch $U[j]$ einen Zeiger auf das Listenelement j in einer Liste S_i bereithält. Abbildung 6.28 illustriert die Situation für eine Menge $S_5 = \{1, 5, 4, 8\}$. Die Operation FIND(x) kann nun in $O(1)$ auf das Element

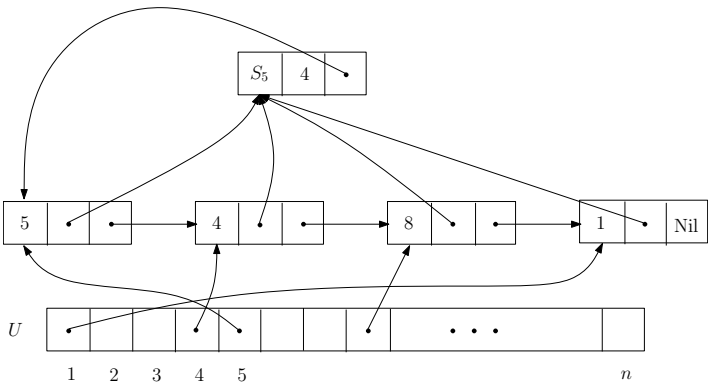

Abbildung 6.28: Die Zeigerstruktur einer Union-Find Realisierung der Menge $S_5 = \{1, 5, 4, 8\}$.

x über $U[x]$ zugreifen und findet das Kopfelement und den Repräsentanten i von S_i durch maximal zwei Verweise.

Die Operation UNION(x, y) kann wie folgt realisiert werden. Sei $|S_x| \geq |S_y|$.

- Jeder Namenszeiger von S_y wird auf den Namen von S_x gesetzt.

- Die Verkettete Liste von S_x wird an S_y angehängt. Der Mengenanfangszeiger von S_x wird auf den Anfang von S_y gesetzt. Die Größen werden addiert und im Kopfelement S_x eingetragen.

- Das alte Kopfelement S_y wird entfernt.

Es ergibt sich ein Aufwand von $O(|S_y|)$.

Theorem 21 *Eine Folge von m* FIND*-Operationen und $k < n$* UNION*-Operationen läßt sich in Zeit $O(m + n \log n)$ ausführen.*

Beweis. Die Laufzeit $O(m)$ für die FIND-Operationen ist klar. Die abschließende Frage lautet, wie oft ein Namenszeiger eines Elementes x insgesamt umgehängt wird. Das Element gehörte dann immer zur kleineren Menge. Nach dem ersten Umhängen gehört x zu einer Menge mit größer gleich 2 Elementen. Nach dem zweiten Umhängen gehört x zu einer Menge mit größer gleich 4 Elementen. Nach dem i-ten Umhängen gehört x also zu einer Menge mit größer gleich 2^i Elementen. Eine solche Menge S_j kann aber nur maximal n Elemente enthalten. Also kann x nur maximal $\log n$ mal ungehängt worden sein. Das gilt für jedes $x \in U$ und somit ergibt sich die Laufzeit. □

Gelegentlich wird in einigen Beschreibungen zur Union-Find Datenstruktur auch eine Operation MAKE-SET(x) angegeben, die eine Menge $S_x = \{x\}$ initialisiert. Wir haben hier diese Initialisierung vorweggenommen. An den Laufzeiten ändert sich nichts.

6.9 Datenstrukturen für Graphen

Im nächsten Kapitel wollen wir einige klassische Algorithmen betrachten, die auf ein Netzwerk von Knoten und Kanten angewendet werden und viele praktische Anwendungen haben. Dazu betrachten wir zunächst geeignete Datenstrukturen für solche Netzwerke.

Ein sogenannter *Graph* ist eine natürliche Erweiterung der Bäume. Formal beschreiben wir einen (ungerichteten) Graphen durch eine Menge $V = \{v_1, v_2, \ldots, v_n\}$ von Knoten und eine Menge $E \subseteq V \times V$ von Kanten, kurz $G = (V, E)$. Jeder formale Graph G kann grafisch (oder geometrisch) in der Ebene analog zu den Bäumen realisiert werden. Für die Knoten $v_i \in V$ legen wir disjunkte Punkte in der Ebene fest und für je zwei Knoten v_1, v_2 in V mit $(v_1, v_2) \in E$ *zeichnen* wir einen einfachen Weg (ohne Selbstschnitte) zwischen den zugehörigen Orten. Falls wir eine bestimmte geometrische Realisation meinen, sprechen wir auch von einem *geometrischen* Graphen.

Nicht jeder Graph läßt sich so geometrisch realisieren, dass die Kantenwege paarweise schnittfrei bleiben. Graphen, die eine schnittfreie geometrische Realisiation erlauben, heißen auch *planare* Graphen. Die Realisation selbst, also der geometrische Graph heißt dann auch *kreuzungsfrei*. Bei einem kreuzungsfreien geometrischen Graphen kann man in der Ebene eindeutige Flächen charakterisieren, die von Kanten eingeschlossen sind.

Wir erlauben auch Kanten der Form (v, v). Der *Grad* eines Knoten v ist die Anzahl der Kanten, die diesen Knoten als Endpunkt haben. Für *ungerichtete* Graphen ist die Kante (v_i, v_j) gleichbedeutend mit (v_j, v_i). Bei *gerichteten* Graphen wird durch (v_i, v_j) eine Kante beschrieben, die von v_i zu v_j verläuft. Im Allgemeinen kann E auch eine Multimenge sein, das heißt es können mehrere Kanten (v_i, v_j) in E enthalten sein. In diesem Fall wird der Graph auch als *Multigraph* bezeichnet.

Für $G = (\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_1), (v_2, v_3), (v_4, v_5), (v_3, v_4), (v_1, v_5)\})$ zeigt Abbildung 6.29 eine kreuzungsfreie geometrische Realisation von G. Wir interessieren uns hier zunächst dafür, wie Graphen im Allgemei-

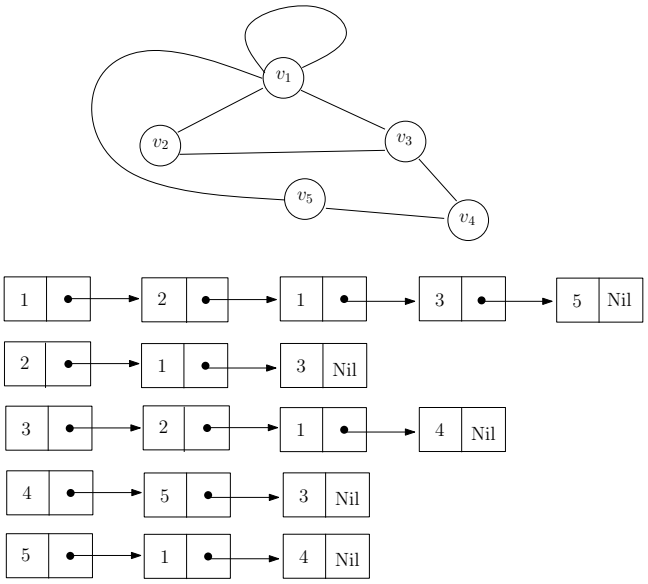

Abbildung 6.29: Eine geometrische Realisation des Graphen $G = (\{v_1, v_2, v_3, v_4, v_5\}, \{(v_1, v_2), (v_1, v_3), (v_1, v_1), (v_2, v_3), (v_4, v_5), (v_3, v_4), (v_1, v_5)\})$. Die Adjazenzliste speichert für jeden Knoten die adjazenten Knoten in einer Liste ab. Es gibt Crossreferenzen zu den jeweiligen Listen.

nen so abgespeichert werden, dass wir effizient auf die notwendigen Informationen zugreifen können. Ein *Weg* in einem Graphen ist eine Folge von Knoten, die über Kanten miteinander verbunden sind. Bei gerichteten Graphen muss die Kantenfolge entsprechend *gerichtet* sein.

Beispielsweise möchten wir nun wissen, ob tatsächlich alle Knoten von einem Startknoten aus sukzessive über Kanten erreichbar sind. Wir stellen also die Frage, ob der Graph insgesamt *zusammenhängend* ist. Zusammenhangskomponenten (maximale zusammenhängende Teilmengen) des Graphen werden über die Knoten definiert. Außerdem kann es sinnvoll sein, alle Knoten und Kanten (bzw. weitere Daten dazu) des Graphen sukzessive auszugeben.

6.9.1 Adjazenzlisten

Eine klassische Speichermethode ist die sogenannte *Adjazenzliste*. Falls für einen Graphen $G = (V, E)$ eine Kante (v_i, v_j) existiert, sind v_i und v_j *Nachbarknoten* und werden als *adjazent* bezeichnet.

Für jeden Knoten v gibt es in der Adjazenzliste eine verkettete Liste aller Knoten, die zu v adjazent sind, also eine Kante bilden, siehe das Beispiel in Abbildung 6.29 ist ein Beispiel angegeben. Genauer handelt es sich um eine verkettete Liste von den zu v adjazenten Knoten.

Der Einfachheit halber können wir Knoten v_i auch durch Indizes $i = 1, \ldots, n$ repräsentieren. Dann können wir von einem Knoten in einer Adjazenzliste leicht zu seiner Adjazenzliste gelangen und auch für jeden Knoten global eine Information abspeichern. Die Listenköpfe lassen sich dann beispielsweise durch ein Array $A[1..n]$ erreichen und die Information für die Knoten wird in $V[1..n]$ gespeichert. Das läßt sich aber genauso durch Zeiger und Namen v_i realisieren.

Offensichtlich benötigen wir für die Adjazenzliste einen Speicherplatz von $O(|V| + |E|)$. Das ist in dem Sinne optimal, dass unser Graph auch $|V| + |E|$ Objekte enthält. Wir stellen nun klassische Traversierungsmöglichkeiten für Graphen vor. Wir wollen alle Knoten des Graphen von einem Startknoten aus berichten.

Tiefensuche (DFS): Geht von einem Startknoten aus rekursiv in die *Tiefe* und besucht rekursiv den nächsten noch nicht besuchten Nachbarknoten. Falls kein solcher Knoten mehr existiert, wird zum letzten Ausgangsknoten der rekursiven Suche zurückgegangen (Backtracking). Dort wird rekursiv der nächste noch nicht besuchte Knoten in die Tiefe *exploriert*.

Breitensuche (BFS): Geht von einem Startknoten aus rekursiv in die *Breite* und besucht zunächst sukzessive durch Vorwärts- und Rückwärtsbewegung alle unbesuchten Nachbarknoten mit Abstand 1 zum Startknoten. Wenn alle Nachbarknoten besucht wurden, geht die Suche rekursiv beim ersten Nachbarknoten weiter.

Wir beschreiben das Verfahren DFS zunächst als Pseudocode und geben ein

Beispiel an. Hierbei markieren wir die Knoten in der DFS Reihenfolge. Wir verwenden einen Stack S.

Prozedur 15 DFS(A, s)
Graph $G = (V, E)$ repräsentiert durch (kopierte) Adjazenzliste A. $A[v]$ ist Zeiger auf Nachbarschaftsliste von v. Vom Startknoten s aus werden alle Knoten markiert, die schon besucht wurden.

```
 1: Push(s);
 2: Markiere s
 3: Top(v);
 4: while v ≠ NIL do
 5:    if A[v] ≠ NIL then
 6:       v' := first(A[v]);
 7:       Lösche v' aus A[v]
 8:       if v' nicht markiert then
 9:          Push(v');
10:          Markiere v'
11:       end if
12:    else
13:       Pop(v);
14:    end if
15:    Top(v);
16: end while
```

Wenn wir beispielsweise den Graphen aus Abbildung 6.29 betrachten, und den Knoten v_5 als Startknoten verwenden, dann können wir die Markierungen leicht in einem Array $M[1..5]$ festhalten. Die Abbbildung 6.30 zeigt einen Zwischenstand bei der Durchführung des Algorithmus für Startknoten v_5. Jeder Knoten wird nur einmal markiert und somit gibt es $|V|$ Push-Operationen. Da in jedem While-Durchlauf entweder ein Element aus einer nichtleeren Liste gestrichen wird, oder ein Element vom Stack gepoppt wird, liegt die Laufzeit des Algorithmus in $O(|V| + |E|)$.

Bei einem *gerichteten Graphen* G hat jede Kante $e = (v, w)$ eine Richtung, nämlich vom Knoten v zum Knoten w. In der Adjazenzliste eines Knotens v stehen dann nur diejenigen Knoten w, für die eine gerichtete Kante (v, w) vorhanden ist.

Betrachten wir als Beispiel den vollständigen Binärbaum T mit 15 inneren Knoten, die von oben nach unten und von links nach rechts mit den Buchstaben $a, b, c, \ldots, o$ bezeichnet sind (in Tiefe 3 befinden sich also die Knoten h bis o). Angenommen, die Kanten sind jeweils vom Vater zu den Söhnen gerichtet, und in den Adjazenzlisten steht jeweils erst der linke Sohn, dann der rechte.

Wenn wir den DFS-Algorithmus in Prozedur 15 auf die Wurzel als Startkno-

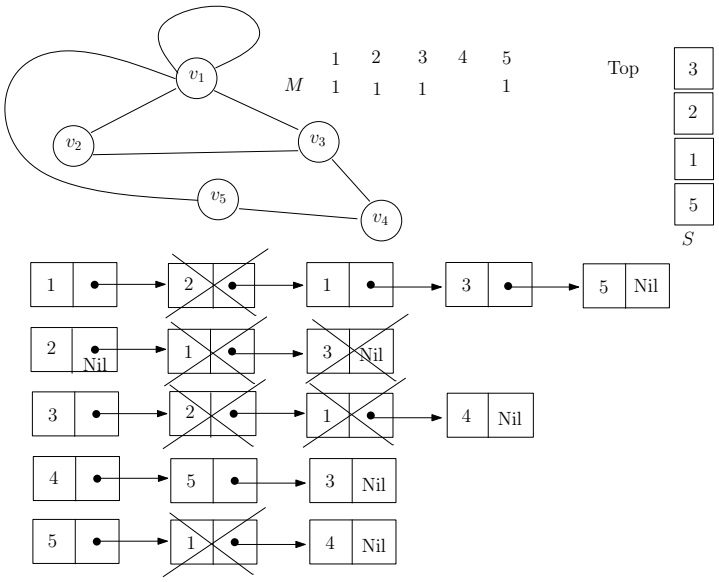

Abbildung 6.30: Ein Zwischenstand des DFS-Aufrufs für den Startknoten v_5. Im nächsten Durchlauf der While-Schleife wird das verbleibende Element der Liste von v_3 betrachtet.

ten anwenden, werden die Knoten in der Reihenfolge a, b, d, h, i, e, j, k, c, f, l, m, g, n, o erstmals besucht und markiert; das entspricht genau der Präorder-Besuchsreihenfolge aus Section 6.3 und illustriert die Bezeichnung "Tiefensuche".

In einem beliebigen Graphen bildet die Menge aller Kanten, über die ein Knoten zum ersten Mal besucht wird, einen Baum mit Wurzel im Startknoten s. Der Stack enthält zu jedem Zeitpunkt genau die Knoten auf dem Pfad vom Startknoten s zum zuletzt markierten Knoten.

Das Verfahren DFS hat zahlreiche Anwendungen. Zum Bespiel kann man die zeitlichen Abhängigkeiten zwischen einzelnen Schritten in einem Ablaufplan (“erst a, dann b”) durch einen gerichteten Graphen darstellen. Von Interesse ist die Frage, ob es in diesem Graphen gerichtete Kreise gibt; falls ja, läßt sich der Plan nicht implementieren. Solch ein gerichteter Kreis entsteht zum Beispiel, wenn man zu obigem Baum T die Kante (k, b) hinzufügt. Dagegen würden die zusätzlichen Kanten (a, g) oder (c, e) keine Kreise verursachen. Das heißt: Ein Kreis entsteht genau dann, wenn DFS eine sogenannte *Rückwärtskante* vom zuletzt markierten Knoten zu einem Vorfahren (oder zu ihm selbst), also zu einem Knoten im Stack, entdeckt. Ob ein markierter Knoten sich im Stack befindet, läßt sich leicht testen, indem man ihn während dieser Zeit färbt. Damit haben wir folgendes Ergebnis:

Theorem 22 *Ob ein gerichteter Graph $G = (V, E)$ einen Kreis enthält, läßt sich mit DFS in Zeit und Speicherplatz $O(|V| + |E|)$ testen.*

Besser geht es nicht, denn die Adjazenzlisten der Eingabe müssen zumindest einmal gelesen werden.

Jetzt ersetzten wir in Prozedur 15 den Stack durch eine *Queue* (Schlange); hier werden neue Elemente stets am Ende der Schlange hinzugefügt, während sich Top und Pop auf das Element am Kopf der Schlange beziehen. Aus der Tiefensuche wird dadurch die *Breitensuche*, und die Knoten des Binärbaums werden nun in der Reihenfolge des Alphabets erstmals besucht und markiert.

6.9.2 Nachbarschaftsmatrix

Eine weitere simple Methode, um einen Graphen effizient abzuspeichern, ist die Verwendung einer Nachbarschaftsmatrix. Wir identifizieren die n Knoten des Graphen wiederum durch die ganzzahligen Indizes und für $V = \{1, 2, \ldots, n\}$ verwenden wir eine $n \times n$ Matrix N mit Einträgen

$$N_{ij} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{sonst} \end{cases}$$

Die Matrix im Beispiel aus Abbildung 6.29 sieht dann wie folgt aus:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Der Vorteil der Speicherung in einer Nachbarschaftsmatrix liegt darin, dass wir in konstanter Zeit testen können, ob eine Kante zwischen zwei Knoten existiert. Allerdings benötigen wir $\Omega(|V|^2)$ viel Speicherplatz.

Literatur

Eine andere Darstellung der hier betrachteten Datenstrukturen findet sich beispielsweise in [2].

Literaturverzeichnis

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. MIT Press, 3. Auflage, 2009.

[2] Norbert Blum. Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung. Oldenbourg Verlag, 2004.