Informatik II
Algorithmen und Datenstrukturen

Thomas Wilke

16. Juli 2007

Inhaltsverzeichnis

Kapitel 1

Einführung und Grundlagen

1.1 Über die Veranstaltung

Motivation Google!

Informatik II = Algorithmen (Alg.) und Datenstrukturen (DS)

Geeignete Alg. und DS sind die Grundlage für schnelle und in der Speichernutzung sparsame (wirtschaftliche, effiziente) Programme bzw. Software.

Umfassende Behandlung von Alg. und DS in der Vorlesung:

- Vorstellung von Standardalg. und -DS,
- Beschreibung zugehöriger Entwurfsmethodik,
- Bewertung der Effizienz von Alg. und DS (Komplexitätsanalyse),
- Implementierung in Java,
- Einsatz in Beispielen.

Bemerkung Festlegung auf Java ist unwesentlich!

Beispiel JavaSort!

1.2 Insertion Sort – Sortieren durch Einfügen

Wir studieren ein konzeptuell einfaches, aber auch langsames Feld-Sortierverfahren, das sich schrittweise einem sortierten Feld nähert.

Datenstruktur = systematische Anordnung von Daten

Felder

In einem *Feld* werden eine feste Anzahl – man spricht von der Feldlänge – von Werten in einer Reihe so angeordnet, dass auf sie über ihren *Index* schnell zugegriffen werden kann. Der Wert mit Index i heißt *Feldelement* mit Index i. Wenn alle Feldelemente ganze Zahlen sind, so sprechen wir von einem ganzzahligen Feld. Auf diese beschränken wir uns vorerst.

Tafelzusatz Graphische Darstellung.

Vorstellung Zusammenhängender Speicherbereich, daher schneller Zugriff.

In Algorithmen erlauben wir den folgenden Umgang mit Feldern (Java-ähnlich):

lege ganzzahliges Feld a der Länge n an (und initialisiere es mit 0): Es wird ein Speicherbereich reserviert, der Platz für n Feldelemente hat. Außerdem werden ggf. alle Feldelemente auf 0 gesetzt. Ein Verweis auf den Speicherbereich wird in der Variablen a für den weiteren Zugriff abgelegt.

setze $z = a[i]$: Das Feldelement mit Index i des Feldes, auf das die Referenz in a verweist, wird der Variablen z als Wert zugewiesen.

setze $a[i] = z$: Der Wert der Variablen z wird dem Feldelement mit Index i des Feldes, auf das die Referenz in a verweist, zugewiesen.

setze $a = b$: (Hierbei müssen a und b zwei Feldvariablen sein.) Dann wird die Referenz in b der Variablen a zugewiesen. Das heißt, a und b enthalten Referenzen auf dasselbe Feld. Möglicherweise ist ein Zugriff auf das Feld, auf das a vorher verwies, nicht mehr möglich.

Zusätzlich: Wenn wir $a = b$ oder $a \neq b$ als logische Bedingungen benutzen, dann ist damit die Frage gemeint, ob die jeweiligen Referenzen gleich sind

oder nicht.

Beispiel

lege ganzzahliges Feld a der Länge 3 an
lege ganzzahliges Feld b der Länge 5 an
setze $a[0] = 1$
setze $b[0] = 2$
falls $a = b$ gib „gleich“ sonst „ungleich“aus
setze $a = b$
falls $a = b$ gib „gleich“ sonst „ungleich“aus
gib $a[0]$ und $b[0]$ aus

Der Reihe nach wird augegeben: „ungleich“, „gleich“, 2 und dann noch einmal 2 ausgegeben.

Anstelle von z ist bei der zweiten Zuweisung auch ein beliebiger Ausdruck zugelassen, dessen Wert dann anstelle des Wertes der Variable genutzt wird. Genauso ist anstelle von i ein Ausdruck zugelassen, der zu einem Feldindex ausgewertet werden kann.

Schreibweise $a[0..n-1]$ für ein Feld mit Namen a und Feldlänge n; $a[i]$ für das Feldelement mit Index i; $[2, 4, -10, 3]$ für konkretes Feld. Das Teilstück von Index i bis Index j wird *Feldsegment* genannt und mit $a[i..j]$ bezeichnet.

Mathematische Modellierung: totale Funktion mit Definitionsbereich $\{0, \ldots, n-1\}$.

Algorithmus

Idee Wir sorgen dafür, dass nach der i-ten Runde die ersten $i + 1$ Feldelemente sortiert sind und der Rest unverändert ist. In Runde $i + 1$ wird dann das $(i + 2)$-te Feldelement an der richtigen Stelle eingefügt, und zwar durch wiederholtes Vertauschen mit größeren Elementen.

InsertionSort(a)

Vorbedingung: a ist ein ganzzähliges Feld der Länge n

für $i = 1$ bis $n - 1$
    setze $j = i$
    solange $j > 0$ und $a[j] < a[j - 1]$
        Swap$(a, j, j - 1)$
        setze $j = j - 1$

Nachbedingung: a ist eine geordnete Permutation (Umordnung) des ursprünglichen Feldes

Swap(a, i, j)

Vorbedingung: a ist ein Feld und i und j sind Indizes des Feldes

1. setze $m = a[i]$
2. setze $a[i] = a[j]$
3. setze $a[j] = m$

Nachbedingung: a ist aus dem ursprünglichen Feld durch Vertauschen der Feldelemente mit Index i und j entstanden

Bemerkung Das Feld wird nur durch die Swap-Operation verändert. Dadurch handelt es sich immer um eine Umordnung des ursprünglichen Feldes.

Tafelzusatz Beispiellauf.

Spezielle Notation für Algorithmen:

Zuweisung, zum Beispiel „setze $j = i + 3$“: Weise der Variablen j den Wert zu, der sich durch Addition von 3 zum Wert der Variablen i ergibt.

für-Schleife, z. B. „für $i = 1$ bis n“: Führe den eingerückten Schleifenrumpf mehrfach hintereinander aus. Zunächst mit dem Wert 1 für i, dann mit dem Wert 2 für i usw. und zum Schluss mit dem Wert n für i. Im Fall $n < 1$ wird der Schleifenrumpf nicht ausgeführt.

solange-Schleife, z. B. „solange $j > 0$“: Falls der Wert der Variablen j größer als 0 ist, führe den eingerückten Schleifenrumpf aus. Überprüfe danach, ob der Wert von j immer noch größer als 0 ist. Falls ja, führe den Schleifenrumpf erneut aus. Und so weiter ... Wenn die Bedingung $j > 0$ nicht (mehr) zutrifft, fahre hinter dem Schleifenrumpf (gleiche Einrücktiefe) fort.

Komplexitätsanalyse

Lemma

1. Sei a ein ganzzahliges Feld der Länge n. Dann werden bei der Ausführung von InsertionSort(a) höchstens $\frac{1}{2}n(n-1)$ Vergleiche zwischen Feldelementen durchgeführt.
2. Sei n eine beliebige natürliche Zahl. Dann gibt es ein Feld der Länge n, so dass bei der Ausführung von InsertionSort(a) genau $\frac{1}{2}n(n-1)$ Vergleiche zwischen Feldelementen durchgeführt werden.

Beweis In Runde i der äußeren für-Schleife nimmt j im schlechtesten Fall nacheinander die Werte i, $i-1$, $\ldots$, 1 an und für jeden dieser Werte wird höchstens ein Vergleich zwischen Feldelementen durchgeführt. Also gibt es insgesamt höchstens

$$\sum_{i=1}^{n-1} i = \frac{1}{2}n(n-1) \tag{1.1}$$

Vergleiche zwischen Feldelementen.

Wir betrachten nun das Feld $a = [n-1, n-2, \ldots, 0]$. Dann werden in Runde i genau i Vergleiche zwischen Feldelementen durchgeführt, so dass insgesamt $\frac{1}{2}n(n-1)$ Vergleiche zwischen Feldelementen durchgeführt werden.

Bemerkung Um bei Komplexitätsanalysen auf der sicheren Seite zu sein (um keine bösen Überraschungen zu erleben), analysieren wir Algorithmen im Hinblick auf ihr schlechtestes (pessimales) Verhalten.

Sprechweise Die *pessimale Anzahl* der von InsertionSort durchgeführten Vergleiche auf einem Feld der Länge n ist $\frac{1}{2}n(n-1)$.

1.3 Binary Search – Binäre Suche

Wir wollen zu einem gegebenen geordneten ganzzahligen Feld a und gegebener Zahl m feststellen, ob m im Feld vorkommt. Außerdem würden wir gern noch wissen, wo das gesuchte Element in Feld steht bzw. wo es eingefügt werden könnte.

Algorithmus

Idee Wir führen eine Intervallschachtelung durch. Zunächst müssen wir annehmen, dass das gesuchte Element irgendwo im Feld vorkommt. Wir bestimmen dann das Feldelement in der Mitte und vergleichen es mit der gesuchten Zahl: Wenn diese größer ist, brauchen wir nur noch in der rechten Hälfte zu suchen; wenn es kleiner ist, nur noch in der linken Hälfte, ansonsten steht sie genau in der Mitte. Wenn wir in einer der Hälften weitersuchen, verfahren wir mit dieser wie wir mit dem ganzen Feld verfahren sind. Wir können dann aufhören, wenn das verbleibende Feldsegment die Länge Null hat.

BinarySearch(a, x)

Vorbedingung: a ist ein ganzzahliges geordnetes Feld der Länge $n > 0$, x ist eine ganze Zahl.

1. *Bestimme Grenzen des Anfangintervalls.*
   setze $l = 0$ und $r = n - 1$
2. *Eigentliche Intervallschachtelung.*
3. solange $l \leq r$
   (a) *Bestimme mittleren Index.*
   $m = (l + r)$ div 2
   (b) *Überprüfe, ob x evtl. in der rechten Hälfte liegt.*
   falls $x > a[m]$
   i. setze $l = m + 1$
   (c) sonst
   i. *Überprüfe, ob x evtl. in der linken Hälfte liegt.*
   falls $x < a[m]$
   A. setze $r = m - 1$
   ii. sonst
   A. *Zahl liegt genau in der Mitte.*
   setze $l = m + 1$ und $r = m - 1$

Nachbedingung: $a = \bar{a}$; $x = \bar{x}$; $x \in a$ gdw. $r + 2 = l$.

Definition Ein Algorithmusstück B ist *(total) korrekt* bezüglich der Vorbedingung ϕ und der Nachbedingung ψ, falls gilt:

1. *(Terminierung)* Falls ϕ vor der Ausführung von B gilt, dann terminiert die Ausführung von B.
2. *(partielle Korrektheit)* Falls ϕ vor der Ausführung von B gilt, dann gilt ψ nach der Termination von B.

Um über den Anfangswert einer Variablen sprechen zu können, folgen wir der folgenden Konvention: Wird der Name einer Variablen überstrichen, dann steht diese Bezeichnung für den Wert der Variablen vor der Ausführung des betrachteten Programmstücks.

Beispiel Vor- und Nachbedingung für InsertionSort formal. Unter einem Feld der Länge n verstehen wir ein Funktion mit Definitionsbereich $\{0, \ldots, n-1\}$, allerdings schreiben wir $a[i]$ anstelle von $a(i)$ für einen Funktionswert. Ein ganzzahliges Feld a der Länge n heißt *geordnet,* wenn $a[i] \leq a[j]$ für alle i und j mit $0 \leq i \leq j < n$ gilt. Ein Feld b der Länge n ist eine Permutation eines Feldes a der Länge n, falls es eine bijektive Abbildung $\pi\colon \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ gibt, so dass $b = a \circ \pi$ gilt.

Vor- und Nachbedingung von InsertionSort: a ist ganzzahliges Feld der Länge n bzw. a ist geordnet und Permutation von $\bar{a}$.

Bemerkung Die Nachbedingung von BinarySearch kann erweitert werden um:

- $-1 \leq r < l \leq n$
- wenn $x \in a$, so $a[r+1] = a[l-1] = x$
- wenn $x \notin a$, so $r + 1 = l$ und $a[r] < x < a[l]$ mit $a[-1] = -\infty$ und $a[n] = \infty$

Entwurfsmethodik = genereller Ansatz, um einen Algorithmus zur Lösung eines algorithmischen Problems zu entwickeln; generell im Sinne von problemunspezifisch

Entwurfsmethodik bei InsertionSort: *schrittweise Verbesserung.*

Entwurfsmethodik hier: Teile-und-herrsche (divide-and-conquer). Das Problem wird zerlegt in voneinander unabhängige Teilprobleme, die separat gelöst werden und deren Einzellösungen dann zur Gesamtlösung zusammengesetzt werden. Hier kann man sich sogar darauf beschränken, ein (!) Teilproblem zu lösen.

Komplexität

Ziel Wir wollen zeigen, dass der Algorithmus tatsächlich schnell ist und durch ständiges Halbieren des Intervalls logarithmische Laufzeit in jedem Fall garantiert werden kann.

Lemma Sei a ein ganzzahliges Feld der Länge $n > 0$ und $x \in \mathbf{Z}$. Dann wird bei der Ausführung von BinarySearch(a, x) die while-Schleife höchstens $(1 + \lfloor \log(n-1) \rfloor)$-mal durchlaufen.

Hierbei bezeichnet $\lfloor r \rfloor$ für jede reelle Zahl r die größte ganze Zahl, die nicht größer als r ist. (Abrunden!)

Sprechweise Wir sagen, $(1 + \lfloor \log(n-1) \rfloor)$ ist eine *obere Schranke* für die Anzahl der Schleifendurchläufe von BinarySearch oder auch die Anzahl der Schleifendurchläufe von BinarySearch ist *logarithmisch* beschränkt.

Beweis Wir zeigen zunächst, dass Folgendes gilt: Wenn $0 \leq l < r \leq n$ zu Beginn der Ausführung des Schleifenrumpfes gilt, dann gilt danach $r - l \leq (\bar{r} - \bar{l})/2$.

Wir treffen eine Fallunterscheidung.

1. Fall, $x > a[m]$. Dann gilt

$$\begin{aligned} r - l &= \bar{r} - ((\bar{l} + \bar{r}) \operatorname{div} 2 + 1) && \text{3.(b)i.} \\ &\leq \bar{r} - (\bar{l} + \bar{r})/2 + 1/2 - 1 && \text{Abschätzung div} \\ &= (\bar{r} - \bar{l})/2 - 1/2 && \text{Vereinfachung} \\ &\leq (\bar{r} - \bar{l})/2 && \text{Abschätzung .} \end{aligned}$$

2. Fall, $x < a[m]$. Dann gilt

$$\begin{aligned} r - l &= (\bar{l} + \bar{r}) \operatorname{div} 2 - 1 - \bar{l} \\ &\leq (\bar{l} + \bar{r})/2 + 1/2 - 1 - \bar{l} && \text{Abschätzung div} \\ &\leq (\bar{r} - \bar{l})/2 && \text{Vereinfachung .} \end{aligned}$$

Im dritten Fall gilt die Behauptung trivialerweise.

Nun gilt allgemein: Ist d eine beliebige Zahl, so ist $d/2^{\log d} = 1$, also $d/2^{\lfloor \log d \rfloor + 1} < 1$. Der Schleifenrumpf wird also höchstens $(1 + \lfloor \log(n-1) \rfloor)$-mal durchlaufen, denn zu Beginn gilt $l = 0$ und $r = n - 1$.

Übungsaufgabe Konstruiere Felder, bei denen die binäre Suche möglichst lange dauert.

Korrektheit

Ziel Wir möchten die Korrektheit von BinarySearch beweisen!

Beobachtung BinarySearch besteht aus einer Initialisierung und einer darauf folgenden solange-Schleife.

Wir überlegen uns deshalb zunächst, wie man die Korrektheit eines Algorithmus

I
solange β
    R

beweisen kann.

Dessen Korrektheit kann man bezüglich einer Vorbedingung ϕ und einer Nachbedingung ψ beweisen, indem man eine möglichst starke Bedingung sucht, die sowohl vor dem Eintritt in die Schleife, nach dem Austritt aus der Schleife und jeweils zu Beginn einer Ausführung des Rumpfes der Schleife gilt. Es handelt sich also um eine Bedingung, die während der gesamten Ausführung der Schleife gilt, man spricht von einer Invarianten!

Genauer:

1. Man sucht eine Bedingung χ, die die folgenden Bedingungen erfüllt:
   (a) I ist korrekt bzgl. der Vorbedingung ϕ und χ.
   (b) Der Rumpf der Schleife ist korrekt bezüglich der Vorbedingung, die χ und β garantiert, und der Nachbedingung χ.
   (c) Aus χ und der Negation von β folgt ψ.
2. Die Schleife terminiert unter der Vorbedingung ϕ.

Eine Bedingung χ, die die obigen Eigenschaften erfüllt, wird *(Schleifen-)Invariante* genannt.

Für BinarySearch haben wir den zweiten Punkt schon erledigt. Nun müssen wir noch eine geeignete Invariante finden. Wir versuchen es mit:

1. $a = \bar{a}$, a geordnet, $x = \bar{x}$, $n = \bar{n}$,
2. $0 \leq l \leq n$, $-1 \leq r < n$, $r + 2 \geq l$,
3. wenn $l \leq r$, so $0 \leq l \leq r < n$,
4. wenn $l \leq r$, so ($x \in a$ gdw. $x \in a[l..r]$),
5. wenn $r + 1 = l$, so $x \notin a$,
6. wenn $r + 2 = l$, so $x \in a$.

Wir prüfen nun die obigen Bedingungen 1.(a)–1.(c) nach. Zunächst ist 1.(a) trivialerweise erfüllt. Zu 1.(c): Der erste Teil der Invariante besagt schon $a = \bar{a}$. Also brauchen wir noch zu zeigen, dass $x \in a$ gdw. $r + 2 = l$ aus der Invariante und $l > r$ folgt. Aus letzterem und dem 2. Teil der Invariante ergibt sich, dass nur $r + 1 = l$ und $r + 2 = l$ möglich sind, woraus dann mit Teil 5 und 6 der Invariante die gewünschte Bedingung folgt.

Um 1.(b) nachweisen zu können, nehmen wir eine Fallunterscheidung vor. Vorab halten wir jedoch fest, dass der Schleifenrumpf weder a, noch x oder n verändert. Deshalb bleibt der 1. Teil der Invariante erhalten. Außerdem folgt aus $\bar{l} \leq \bar{r}$ und dem 3. und 4. Teil der Invariante sofort $0 \leq \bar{l} \leq \bar{r} < n$ und ($x \in a$ gdw. $x \in a[\bar{l}..\bar{r}]$). Über den Wert von m wissen wir wegen 3.(a): $\bar{l} \leq m \leq \bar{r}$. Da im Rumpf entweder l auf $m+1$ oder r auf $m-1$ gesetzt wird oder beides geschieht, können wir auch schließen, dass nach der Ausführung des Rumpfes der 2. und 3. Teil der Invariante gilt.

1. Fall, $x > a[m]$. Da a geordnet ist, gilt also $x \in a$ gdw. $x \in a[m+1..n-1]$. Aus ($x \in a$ gdw. $x \in a[\bar{l}..\bar{r}]$) folgt dann $x \in a$ gdw. $x \in a[m+1..\bar{r}]$, also $x \in a$ gdw. $a[l..r]$, denn wegen 3.(b)i. gilt $l = m + 1$ und $r = \bar{r}$. Wegen $\bar{l} \leq \bar{r}$ und 3.(b)i. kann $r+2 = l$ ausgeschlossen werden. Im Fall $r+1 = l$ wissen wir aber $x \notin a$, da $a[l..r+1]$ dann leer ist. Also wissen wir, dass sowohl der 4. wie auch

der 5. und der 6. Teil der Invariante nach Ausführung des Schleifenrumpfes gelten.

2. Fall, $x < a[m]$. Hier verläuft die Argumentation analog zum 1. Fall.

3. Fall, $x = a[m]$. Dann sieht man sofort ein, dass die Teile 4, 5 und 6 gelten.

Damit ist die Korrektheit von BinarySearch nachgewiesen.

Übungsaufgabe Finde eine (stärkere) Invariante, die es erlaubt, auch die Korrektheit bzgl. der stärkeren Nachbedingung zu beweisen.

1.4 Größenordnungen (orders of magnitude)

Vorbemerkung Wir werden Resourcenverbrauch von Algorithmen und Datenstrukturen in der Regel in Abhängigkeit von der Eingabegröße betrachten. Häufig werden wir nur am Wachstum, an dem so genannten *asymptotischen Verhalten* der Laufzeit interessiert sein. Das führt zu den folgenden Definitionen.

Definition Eine partielle Funktion $f\colon \mathbf{N} \to \mathbf{R}$ heißt *schließlich positiv (sp),* wenn es ein k gibt, so dass $f(n)$ für alle $n \geq k$ definiert und > 0 ist.

Definition Eine sp Funktion $f\colon \mathbf{N} \to \mathbf{R}$ ist *asymptotisch kleiner gleich* einer sp Funktion $g\colon \mathbf{N} \to \mathbf{R}$, i. Z. $f \leq_{as} g$, wenn es $d \in \mathbf{R}_{>0}$ und $k \in \mathbf{N}$ gibt, so dass $f(n) \leq d \cdot g(n)$ für alle $n \in \mathbf{N}$ mit $n \geq k$ gilt.

Beispiel Seien f und g gegeben durch $f(n) = 100 \cdot n + 1000$ und $g(n) = n^2$. Dann gilt $f \leq_{as} g$, aber nicht $g \leq_{as} f$.

Begründung Es gilt $\lim_{n\to\infty} f(n)/g(n) = 0$, d. h., es gibt $k \in \mathbf{N}$, so dass für alle $n \in \mathbf{N}$ mit $n \geq k$ gilt: $f(n)/g(n) \leq 1$. Dann gilt aber $f(n) \leq g(n)$, da $g(n) \geq 0$ für alle $n \in \mathbf{N}$.

Umgekehrt gilt $\lim_{n\to\infty} g(n)/f(n) = \infty$, so dass es für jedes $d \in \mathbf{R}_{>0}$ und jede Zahl $k \in \mathbf{N}$ ein $n \geq k$ gibt mit $g(n)/f(n) > d$, also $g(n) > d \cdot f(n)$.

Allgemeiner:

Lemma Seien $f, g\colon \mathbf{N} \to \mathbf{R}$ sp Funktionen, so dass es ein k gibt mit $g(n) > 0$ für alle $n \geq k$. Dann sind folgende Aussagen äquivalent:

- $f \leq_{as} g$,
- $\limsup_{n\to\infty} f(n)/g(n) < \infty$.

Bemerkung Bei der Bestimmung von Laufzeiten werden wir häufig versuchen, komplizierte Summen oder Rekursionsgleichungen mit einfachen Funktionen asymptotisch zu vergleichen. Von dieser Art sind alle folgenden Beispiele.

Beispiel Wir betrachten f und g definiert durch $f(n) = \sum_{i=0}^{n} i$ und $g(n) =$

n^2 für alle $n \in \mathbf{N}$. Dann gilt $f(n) = \frac{n(n+1)}{2}$, also $\lim f(n)/g(n) = 1$ und damit sowohl $f \leq_{as} g$ wie auch $g \leq_{as} f$ nach obigem Lemma.

Definition Sei $f\colon \mathbf{N} \to \mathbf{R}$ eine sp Funktion. Dann sind die folgenden Mengen von Funktionen definiert:

$$O(f) = \{g\colon \mathbf{N} \to \mathbf{R} \mid g \text{ sp und } g \leq_{as} f\} \ , \tag{1.2}$$

$$\Omega(f) = \{g\colon \mathbf{N} \to \mathbf{R} \mid g \text{ sp und } f \leq_{as} g\} \ , \tag{1.3}$$

$$\theta(f) = O(f) \cap \Omega(f) \ , \tag{1.4}$$

$$\Omega_\infty(f) = \{g\colon \mathbf{N} \to \mathbf{R} \mid g \text{ sp und es gibt } c > 0 \text{ und } \infty \text{ viele } n \in \mathbf{N} \text{ mit } cf(n) \leq g(n)\} \ . \tag{1.5}$$

Sprechweise „geh ist groß oh von eff“, „geh ist groß omega von eff“ und „geh ist teta von eff“.

Schreibweise Der Einfachheit halber ersetzen wir häufig eine Funktion, an der wir interessiert sind, durch den Term, der Funktionswert an der Stelle n beschreibt, etwa $\sum_{i=0}^{n} i \in O(n^2)$.

Beispiel $100n + 1000 \in O(n^2)$, aber $n^2 \notin O(100n + 1000)$.

Beispiel Wir betrachten f und g mit $f(n) = \sum_{i=0}^{n} 2^i$ und $g(n) = 2^n$. Dann ist $f(n) = 2^{n+1} - 1$, also $f \in \theta(g)$, da $\lim_{n\to\infty} f(n)/g(n) = 2$.

Schreibweise $\sum_{i=0}^{n} 2^i \in \theta(2^n)$.

Bemerkung Die binäre Relation $\leq_{as}$ ist eine Präordnung (reflexiv und transitiv). Die zugehörige Äquivalenzrelation $\leq_{as} \cap \geq_{as}$ wird mit $\equiv_{as}$ bezeichnet.

Definition Zwei sp Funktionen $f, g\colon \mathbf{N} \to \mathbf{R}$ heißen *asymptotisch gleich,* wenn $f \equiv_{as} g$ gilt.

Bemerkung $\theta(f)$ ist die Äquivalenzklasse von f bezüglich $\equiv_{as}$.

Beispiel $n^2 \equiv_{as} \sum_{i=0} i$ und $2^n \equiv_{as} \sum_{i=0} 2^i$.

Beispiel Die *Stirlingsche Formel* zur asymptotischen Beschreibung der Fa-

kultätsfunktion: Es gibt eine Funktion f mit $f \in \theta\left(\frac{1}{n^2}\right)$, so dass

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + f(n)\right) \tag{1.6}$$

gilt.

Schreibweise

$$n! \in \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \theta\left(\frac{1}{n^2}\right)\right) \quad . \tag{1.7}$$

Beispiel Wir wollen $\sum_{i=0}^{n} 2^i(n-i)$ abschätzen. Einfache Analyse:

$$\begin{aligned}
\sum_{i=0}^{n} 2^i(n-i) &= 2^n \sum_{i=0}^{n} 2^{i-n}(n-i) \\
&= 2^n \sum_{i=0}^{n} \frac{n-i}{2^{n-i}} \\
&= 2^n \sum_{i=0}^{n} \frac{i}{2^i} \\
&\leq 2^n \sum_{i=0}^{\infty} \frac{i}{2^i} \\
&\leq 2^n c
\end{aligned}$$

für ein geeignetes c nach dem Quotientienkriterium, denn $\frac{i+1}{2^{i+1}}\frac{2^i}{i} = \frac{i+1}{2i} \leq 3/4 < 1$ für $i \geq 2$, also $\sum_{i=0}^{n} 2^i(n-i) \in \theta(2^n)$ bzw. $2^n \equiv_{as} \sum_{i=0}^{n} 2^i(n-i)$.

Beispiel Es seien f und g sp Funktionen mit $g(n) \leq g(\lceil n/2 \rceil) + g(\lfloor n/2 + \rfloor) + f(n)$ für alle n und $f \in O(n)$.

Ansatz Für eine Zweierpotenz n und $f(n) = n$ gilt: $g(n) \leq 2g(n/2) + n \leq 4g(n/4) + 2n \leq 8g(n/8) + 3n = \ldots$ Vermutung: $g(n) \in O(n \log n)$.

Wir versuchen deshalb, einen Induktionsbeweis zu finden für:

Lemma Seien g und f sp Funktionen derart, dass $g(n) \leq g(\lceil n/2 \rceil) + g(\lfloor n/2 \rfloor) + f(n)$ für alle n und $f \in O(n)$ gilt. Dann gilt $g \in O(n \log n)$.

Beweis Wir betrachten zunächst Hilfsfunktionen f' und g' definiert durch $f'(n) = \max\{f(0), \ldots, f(n)\}$ und $g'(n) = \max\{g(0), \ldots, g(n)\}$. Dann gilt

$$\begin{aligned} g'(n) &= \max\{g(0) + g(0) + f(0), \ldots, g(\lceil n/2 \rceil) + g(\lfloor n/2 \rfloor) + f(n)\} \\ &\leq \max\{g(0) + g(0), \ldots, g(\lceil n/2 \rceil)\} + \max\{g(0), \ldots, g(\lceil n/2 \rceil)\} + \max\{f(0), \ldots, f(n)\} \\ &= g'(\lceil n/2 \rceil) + g'(\lfloor n/2 \rfloor) + f'(n) \ . \end{aligned}$$

Da $f' \in O(n)$ gilt (einfache Rechung!), erfüllen g' und f' dieselben Voraussetzungen wie g und f, zusätzlich ist aber g' monoton und es gilt $g \leq g'$. Wenn wir also die Behauptung für g' zeigen können, folgt auch die Behauptung für g.

Da $f' \in O(n)$, gibt es k und c derart, dass $f'(n) \leq cn$ für $n \geq k$ gilt. Ohne Einschränkung können wir annehmen, dass k eine Zweierpotenz, etwa $k = 2^m$, und ≥ 2 ist. Sei $b \geq c$ so groß, dass außerdem $g'(n) \leq bk \log k$ gilt.

Wir behaupten nun zunächst, dass $g'(n) \leq bn \log n$ für alle n der Form 2^l mit $l \geq m$ gilt. Dies zeigen wir per Induktion über l.

Der Induktionsanfang, $l = m$, ist trivial, denn wir haben b entsprechend gewählt.

Für den Induktionsschritt sei $l > m$. Dann wir können schließen:

$$\begin{aligned} g'(n) &= g'(2^l) && n = 2^l \\ &\leq 2g'(2^{l-1}) + c2^l && \text{Voraussetzung} \\ &\leq 2b2^{l-1}(l-1) + c2^l && \text{Induktionsannahme} \\ &\leq b2^l l + (c-b)2^l \\ &\leq b2^l l && b \geq c \ . \end{aligned}$$

Zum Schluss zeigen wir, dass $g'(n) \leq dn \log n$ für alle $n \geq k$ und für $d = 4b$ gilt.

Sei dazu $n \geq k$. Dann gibt es ein $l > m$, so dass $2^{l-1} \leq n < 2^l$ gilt. Wir

wissen dann:

$$
\begin{aligned}
g'(n) &\leq g'(2^l) && g' \text{ monoton} \\
&\leq b2^l l && \text{oben bewiesen} \\
&= 2b2^{l-1}(l-1) + 2b2^{l-1} && \\
&\leq 4b2^{l-1}(l-1) && l > m \geq 1, \text{ also } l \geq 2 \\
&\leq d2^{l-1}(l-1) && \text{Definition von } d \\
&\leq dn \log n && 2^{l-1} \leq n \ .
\end{aligned}
$$

1.5 Resourcenverbrauch: Laufzeiten (running time) und Speichernutzung (space)

Vorbemerkung Häufig werden zur Beschreibung des Resourcenverbrauchs Laufzeit und Speicherbedarf herangezogen. Dazu unten mehr. Es gibt aber häufig auch andere sinnvolle Kennzahlen: Anzahl der Vergleiche, Anzahl der Zugriffe auf eine Datenstrukture, ...

Laufzeit

— Fast immer hängt die Laufzeit am stärksten (!) von der Größe der Eingabe ab. Deshalb messen wir Laufzeiten in Abhängigkeit von der so genannten Eingabegröße *(input size)*, z. B. Länge eines zu sortierenden Feldes.

— Wir interessieren uns in der Regel für den schlechtesten Fall, um auf der sicheren Seite zu sein: Für eine Eingabegröße n betrachten wir das Maximum aller Laufzeiten für Eingaben der Größe n. Wir sprechen dann von der *pessimalen Laufzeit* (worst-case running time).

— Tatsächliche Laufzeiten hängen stark von der Hardware ab. Z. B. bewirkt ein schnellerer Prozessor eine Laufzeitverkürzung um einen konstanten Faktor. Außerdem interessiert die Laufzeit für kleine Eingabegrößen (winzige Probleme) nicht sonderlich. Daher beschränken wir uns auf asymptotische Betrachtungen.

— Um Laufzeiten für Programme (und damit auch Algorithmen) bestimmen zu können, müssen wir den einzelnen Programmkonstrukten (fiktive) Laufzeiten zuordnen. Dabei lassen wir uns von der Laufzeit auf einem Modell einer üblichen Rechnerarchitektur leiten. Außerdem reicht es, dass wir dabei asymptotische Angaben machen, wenn wir insgesamt nur an einer asymptotischen Analyse interessiert sind. Die Gesamtlaufzeit ergibt sich dann als Summe der Einzellaufzeiten.

Zu den Einzellaufzeiten:

- Wenn keine komplizierten Daten verabeitet werden, gehen wir von konstanter Laufzeit aus, d. h., wir setzen eine Zeiteinheit an. Insbesondere

setzen wir für den Zugriff auf ein Feldelement und die Zuweisung von Werten eine Zeiteinheit an.

- Bsp. für nicht-konstante Laufzeit:
  1. Das Anlegen eines Feldes der Länge n hat Laufzeit $\theta(n)$. Wir sprechen von *linearer Laufzeit* und setzen n als Laufzeit an.
  2. Die Konkatenation zweier Zeichenfolgen der Länge m bzw. n hat eine Laufzeit, die linear in $m+n$ ist, wir setzen also einfach $m+n$ an.

Beispiel Ein Programmstück, in dem Feldelemente eine Feldes a der Länge n aufsummiert werden (schrittweise Verbesserung):

Sum(a)

Vorbedingung: a ist ein ganzzahliges Feld der Länge n.

1. *Initialisiere Partialsumme und Index.*
   setze $s = 0$ und $i = 0$
   Invariante: $a = \bar{a}$; $n = \bar{n}$; $s = \sum_{j=0}^{i-1} a[j]$
2. solange $i < n$
   (a) *Passe Partialsumme an.*
   setze $s = s + a[i]$
   (b) *Gehe zu nächstem Index.*
   setze $i = i + 1$

Nachbedingung: $a = \bar{a}$; $n = \bar{n}$; $s = \sum_{j=0}^{n-1}$

Laufzeitabschätzung:

- Die Initialisierung nimmt konstante Laufzeit in Anspruch, wir setzen 2 Zeiteinheiten an.
- Die Schleife wird n-mal durchlaufen. Dafür werden $n + 1$ Tests $i < n$ durchgeführt und n Zuweisungen $s = s+a[i]$ sowie $i = i+1$. Wir setzen $n + 1 + 2n$ an.
- Insgesamt ergibt sich $3n + 3$. Das heißt, die (pessimale) Laufzeit ist $\theta(n)$.

Bemerkung Wir hätten für die Zuweisung auch eine andere Rechnung aufmachen können: eine Zeiteinheit für die Feldabfrage, eine Zeiteinheit für die Addition, eine Zeiteinheit für die Zuweisung, ... Insgesamt hätte sich dann aber auch konstante Zeit ergeben, so dass sich an der Abschätzung $\theta(n)$ nichts ändern würde.

Beispiel *Sortieren durch Zählen (counting sort).* Wir wollen ein ganzzahliges Feld $a[0..n-1]$ sortieren, von dem wir wissen, dass es nur Zahlen zwischen 0 und 255 enthält.

Tafelzusatz Beispiel.

Einfacher Ansatz: Wir durchlaufen das Feld einmal und zählen, wie häufig jede Zahl vorkommt. Danach legen wir entsprechend viele Elemente der Reihe nah ab:

CountingSort256(a)

Vorbedingung: a ist ein ganzzahliges Feld der Länge n und für alle $i < n$ gilt $0 \leq a[i] < 256$.

1. *Lege Zählfeld an und initialisiere es.*
   lege Feld $c[0..255]$ an und setze jedes Feldelement auf 0
2. *Zähle Vorkommen in gegebenem Feld.*
   für $i = 0$ bis $n - 1$
   (a) *Erhöhe entsprechenden Zähler.*
   setze $c[a[i]] = c[a[i]] + 1$
   Zusicherung: für alle $j < 256$ gilt $c[j] = |\{i < n \mid a[i] = j\}|$
3. *Überschreibe Feld entsprechend der Angaben im Zählfeld.*
   setze $i = 0$
   für $j = 0$ bis 255
   für $k = 1$ bis $c[j]$
   setze $a[i] = j$ und $i = i + 1$

Nachbedingung: a ist eine Permutation von $\bar{a}$ und geordnet.

Zur Laufzeit für Feld der Länge n:

- Anlegen von c: 256 Zeiteinheiten, also $\theta(1)$,
- erste for-Schleife: $7n$ (pro Durchlauf einmal Inkrementieren, einmal mit 256 vergleichen, vier Feldzugriffe, einmal Inkrementieren), also $\theta(n)$,
- Initialisieren von i: eine Zeiteinheit, also $\theta(1)$
- zweite for-Schleife: $3 \times 256 + 2 \times n$, also $\theta(n)$,
- insgesamt: $\theta(n)$.

Satz Die (pessimale) Laufzeit von Sortiern-durch-Zählen ist $\theta(n)$.

Wir haben in dem Algorithmus den Begriff „Zusicherung“ benutzt.

Definition Eine *Zusicherung* ist eine Bedingung an eine Variablenbedingung. Sie können in Algorithmen stehen. Dann heißt eine Zusicherung *korrekt,* wenn gilt: Ist vor der Ausführung des Algorithmus die Vorbedingung erfüllt und gelangt der Algorithmus zur Zusicherung, dann ist auch diese erfüllt.

Bemerkung Eine Schleifeninvariante ist eine besondere Zusicherung: Sie ist eine Zusicherung, die korrekt ist, wenn man sie unmittelbar vor die Schleife, zu Beginn des Schleifenrumpfes und unmittelbar nach die Schleife setzt.

Speicherbedarf

— Asymptotische Analyse.

— Objekte beanspruchen soviel Platz, wie bei einer Implementierung auf einem konkreten Rechner benötigt würde.

— Die eigentlichen Referenzwerte benötigen konstanten Platz, genau wie die primitiven Werte.

Wichtige Unterschiede bzw. zu beachten:

— Die Eingabe(größe) bleibt unberücksichtigt, es sei denn, die Eingabe wird verändert.

— In der Regel wird eine pessimale Analyse durchgeführt: maximale Speicherbelegung während der Ausführung aller Programmläufe zu einer festen Eingabegröße. Wir gehen davon aus, dass die Freigabe nicht mehr benötigten Speichers immer rechtzeitig erfolgt.

— Jeder Aufruf einer Prozedur belegt Speicherplatz: Verwaltungsinformation und ggf. lokale Variablen und Parameter. Insbesondere bei rekursiver Programmierung zu beachten.

Beispiel *Sortieren durch Zählen.*

- c benötigt konstanten Platz: $\theta(1)$,
- a benötigt Platz n, also $\theta(n)$,
- i, j, k belegen konstanten Platz: $\theta(1)$,

- insgesamt: $\theta(n)$.

Kapitel 2

Sortierverfahren

2.1 Einführung

Aufgabe

- Einfache Sicht: Zahlen in eine sortierte Reihenfolge bringen.
- Allgemeinere Fragestellung: Objekte mit Sortierschlüssel (Elemente einer linearen Ordnung) in eine sortierte Reihenfolge bringen.

Wichtige Kriterien:

- Wie sind die Objekte gegeben?- Liste, Feld, Datei, Strom, . . . ?
- Wohin sollen die Objekte sortiert werden?- Liste, Feld, Datei, Strom, . . . ?
- Welche Art und wieviel Zwischenspeicher werden zur Verfügung gestellt?
- Haben die Sortierschlüssel eine spezielle Struktur und wenn ja, welche?
- Was geschieht mit Objekten, die den selben Sortierschlüssel haben?

Daraus ergeben sich eine Reihe von Definitionen.

Ein Sortieralgorithmus heißt *vergleichsbasiert,* wenn auf die Sortierschlüssel ausschließlich über deren Vergleichsrelation zugegriffen wird. Im Gegensatz dazu haben wir bei CountingSort ausgenutzt, dass wir wussten, dass die Sor-

tierschlüssel Bytes (Zahlen zwisch 0 und 255) waren. Bei vergleichsbasierten Verfahren werden wir anstelle der Laufzeit häufig die Anzahl der durchgeführten Vergleiche durchführen. Die sind in diesem Fall fast immer asymptotisch gleich der Laufzeit.

Wir werden häufig davon ausgehen, dass die zu sortierenden Objekte in einem Feld liegen und fordern, dass die sortierten Objekte in ein Feld abgelegt werden. Wir sprechen dann von *Feld-Sortieralgorithmen.*- Man spricht von *externen* Sortierverfahren, wenn die Eingabedaten in Dateien oder Strömen vorliegen und so umfangreich sind, dass der Hauptspeicher (bzw. Felder) nicht benutzt werden können, um alle Elemente zwischenzuspeichern.

In-place-/In-situ-Verfahren sortieren Felder allein durch (Vergleiche und) Vertauschen von Feldelementen; Feldelemente werden nicht zwischengespeichert.

Ein Sortierverfahren heißt *stabil,* wenn die Reihenfolge von Objekten mit gleichem Sortierschlüssel durch das Verfahren nicht verändert wird.

Im weiteren Verlauf der Vorlesung:

- konkrete vergleichsbasierte Sortierverfahren für ganzzahlige Felder mit unterschiedlichen Eigenschaften
- Feld-Sortierverfahren für Objekte mit Sortierschlüssel
- nicht vergleichsbasierte Verfahren
- Strom-Sortierverfahren
- untere Schranke für vergleichsbasierte Sortierverfahren

2.2 Merge Sort - Sortieren durch Verschmelzen

Wir studieren ein effizientes Sortierverfahren, dass eine andere Divide-and-Conquer-Strategie verfolgt.

2.2.1 Algorithmus

Idee Wir teilen das Feld in der Mitte, sortieren die beiden Hälften separat und führen die beiden sortierten Hälften geeignet zusammen. Dann sollten wir auch im pessimalen Fall $O(n \log n)$ erhalten.

Frage Wie führen wir geeignet zusammen? Wie „verschmelzen“ wir geeignet?

Antwort Wir vergleichen zuerst die beiden ersten Elemente der Teilfelder und verschieben das kleinere in das Ergebnisfeld. Dann verfahren wir rekursiv mit den Restfeldern. Wenn ein Teilfeld leer ist, wird das andere einfach an das Ergebnisfeld angehängt.

Tafelzusatz Beispiel für Verschmelzen.

Problem Das Ergebnis können wir nicht direkt im Ausgangsfeld ablegen.

Lösung Wir benutzen ein Hilfsfeld bzw. legen ein Hilfsfeld an.

Merge(u, l, m, r, v)

Vorbedingung: u und v sind ganzzahlige Felder der Länge n; $u \neq v$; $l \leq m \leq r < n$; $u[l..m-1]$ und $u[m..r]$ sind geordnet

1. *Initialisiere: Setze Zeiger L und R für Beginn der Restsegmente.*
   setze $L = l$ und $R = m$
2. *Übertrage jeweils das kleinste noch verbleibende Element in das Ergebnisfeld.*
   solange $L < m$ oder $R < r + 1$

Überprüfe, ob rechtes Restsegment leer oder linkes nicht leer und Anfangselement kleiner als das des rechten Restsegmentes.
falls $R > r$ oder ($L < m$ und $u[L] \leq u[R]$)
Übertrage linkes Anf.-Element in Ergebnisfeld und erhöhe Zeiger.
setze $v[l] = u[L]$ und $L = L + 1$
sonst
Übertrage rechtes Anf.-Element und erhöhe Zeiger.
setze $v[l] = u[R]$ und $R = R + 1$
setze $l = l + 1$

Nachbedingung: $v[\bar{l}..\bar{r}]$ ist ein geordnete Permutation von $u[\bar{l}..\bar{r}]$; $u[i] = \bar{u}[i]$ und $v[i] = \bar{v}[i]$ für alle $i \in \{0, \ldots, \bar{l}-1, \bar{r}+1, \ldots, \bar{n}-1\}$

Die Schleifeninvariante ist:

- $m = \bar{m}$, $r = \bar{r}$, $\bar{l} \leq l \leq m \leq R \leq r + 1$
- $v[\bar{l}..l-1]$ ist geordnete Permutation von $u[\bar{l}..L-1]u[m..R-1]$
- für alle $i \in \{\bar{l}, \ldots, l-1\}$ und $j \in \{L, \ldots, m-1, R, \ldots, r\}$ gilt $u[i] \leq v[j]$
- $u[i] = \bar{u}[i]$ für alle $i \in \{0, \ldots, \bar{l}-1, L, \ldots, m-1, R, \ldots, n-1\}$

MergeSort(a)

Vorbedingung: a ist ein Feld der Länge n

1. *Erzeuge Hilfsfeld b und kopiere a nach b.*
   lege Feld b der Länge n an
   für $i = 0$ bis $n - 1$
   setze $b[i] = a[i]$
2. *Sortiere a mit Hilfsfeld b.*
   ParaMergeSort$(a, 0, n-1, b)$

Nachbedingung: a ist geordnete Permutation von $\bar{a}$

ParaMergeSort(u, l, r, v)

Vorbedingung: u und v sind Felder der Länge n; $u \neq v$; $0 \leq l \leq r < n$; $u[i] = v[i]$ für alle $i \in \{l, \ldots, r\}$

Überprüfe, ob Restfeld noch mindestens zwei Elemente hat.
falls $l < r$

(a) *Bestimme die Segmentmitte.*
setze $m = l + (r - l + 1)$ div 2
(b) *Sortiere linke Hälfte von v mit Hilfe von u.*
ParaMergeSort$(v, l, m - 1, u)$
(c) *Sortiere rechte Hälfte von v mit Hilfe von u.*
ParaMergeSort(v, m, r, u)
(d) *Verschmelze von v nach u.*
Merge(v, l, m, r, u)

Nachbedingung: $u[\bar{l}..\bar{r}]$ ist geordnete Permutation von $\bar{u}[\bar{l}..\bar{r}]$; $v[i] = \bar{v}[i] = u[i] = \bar{u}[i]$ für $i \in \{0, \ldots, \bar{l} - 1, \bar{r} + 1, \ldots, \bar{n} - 1\}$

Bemerkung Zur Parameterübergabe in unseren Algorithmen: Es werden einfach die Werte übergeben (call-by-value). Wenn es sich also zum Beispiel um Zahlen handelt, werden Zahlen übergeben. Wenn es sich aber zum Beispiel um Variablen für Felder handelt, werden die entsprechenden Referenzen übergeben.

2.2.2 Pessimale Laufzeit

Tafelzusatz Aufrufstruktur für die Laufzeitabschätzung.

Satz Merge Sort ist ein stabiles Sortierverfahren, das höchstens $1{,}7\, n \log n$ Vergleiche durchführt und damit eine pessimale Laufzeit von $\theta(n \log n)$ hat. Es ist nicht in-situ und benötigt einen Hilfsspeicher der Größe n.

Beweis Sei v_n die pessimale Anzahl von Vergleichen, die Merge Sort für Felder der Länge n durchführt. Dann gilt $v_2 = 1$ und

$$v_n \leq v_{\lfloor n/2 \rfloor} + v_{\lceil n/2 \rceil} + n - 1 \tag{2.1}$$

für $n > 2$. Dann gilt nach dem, was wir aus dem ersten Kapitel wissen: $v_n \in O(n \log n)$, sogar $v_n \leq 1{,}7\, n \log n$ (letzteres ohne Beweis).

2.3 Quick Sort – Sortieren durch Partitionieren

Wir studieren ein im Mittel schnelles Sortierverfahren, dass dem Divide-and-Conquer-Ansatz folgt.

2.3.1 Algorithmus

Der Divide-and-Conquer-Ansatz kann beim Sortieren eines Feldes $a[0..n-1]$ wie folgt verwirklicht werden. Wir nehmen ein beliebiges Feldelement x als so genanntes *Pivotelement,* ordnen das Feld um, so dass es eine Position j gibt, an der x steht und für die gilt, dass die Elemente in $a[0..j-1]$ kleiner gleich x und die Elemente in $a[j+1..n-1]$ größer gleich x sind. Dann sortieren wir $a[0..j-1]$ und $a[j+1..n-1]$ separat rekursiv. Das entstehende Feld ist wird sein sortiert.

GenericQuickSort(a)

Vorbedingung: a ist ein ganzzahliges Feld der Länge $n > 0$

ParaGenericQuickSort$(a, 0, n-1)$

Nachbedingung: a ist eine geordnete Permutation von $\bar{a}$

ParaGenericQuickSort(a, l, r)

Vorbedingung: a ist ein ganzzahliges Feld der Länge $n > 0$; falls $l \leq r$, so $0 \leq l \leq r < n$

1. *Überprüfe Abbruchbedingung.*
   falls $l < r$
   (a) *Wähle Pivotindex.*
   belege j nach Pivotwahlstrategie $(i \leq j \leq r)$
   (b) *Partitioniere Feldsegment* $a[l..r]$.
   setze $j =$ Partition(l, j, r)
   (c) *Verfahre rekursiv.*
   ParaGenericQuickSort$(a, l, j-1)$
   ParaGenericQuickSort$(a, j+1, r)$

Nachbedingung: $a[\bar{l}..\bar{r}]$ ist geordnete Permutation von $\bar{a}[\bar{l}..\bar{r}]$; für alle $i \in \{0, \ldots, \bar{l}-1, \ldots, \bar{r}+1, \ldots, n-1\}$ gilt $a = \bar{a}[i]$

Frage Wie wählen wir das Pivotelement? Wie ordnen wir entsprechend um?

Antwort Das Wählen des Pivotelements kann auf viele Arten erfolgen. Z. B., kann man das erste Element, das mittlere Element, den Median des ersten, mittleren und des letzten Elementes oder immer ein zufälliges Element wählen. Zum Umordnen benutzen wir eine einfache Prozedur.

Partition(a, l, j, r)

Vorbedingung: a ist ein ganzzahliges Feld der Länge n; $0 \leq l \leq j \leq r < n$

1. *Bringe Pivotelement an den Anfang.*
   Swap(a, l, j)
2. *Initialisiere Pivotelement x, linken und rechten Rand des betrachteten Segments, L bzw. R.*
   setze $x = a[l]$, $L = l$ und $R = r + 1$
3. *Finde erstes zu vertauschendes Element von links.*
   wiederhole setze $L = L + 1$ bis $L = r + 1$ oder $a[L] \geq x$
4. *Finde erstes zu vertauschendes Element von rechts; immer erfolgreich.*
   wiederhole setze $R = R - 1$ bis $a[R] \leq x$
5. *Führe jeweils eine Vertauschung durch und identifiziere die nächste.*
   solange $L < R$
   (a) *Führe Vertauschung durch.*
   Swap(a, L, R)
   (b) *Suche die nächsten zu vertauschenden Elemente.*
   wiederhole setze $L = L + 1$ bis $a[L] \geq x$
   wiederhole setze $R = R - 1$ bis $a[R] \leq x$
6. *Bringe anfängliches Pivotelement an die richtige Stelle.*
   Swap(a, l, R)
7. *Rückgabe des Indexes des Pivotelementes.*
   gib R zurück

Schleifeninvariante Übung.

Mit *Simple Quick Sort* bezeichnen wir das generische Quick Sort mit der einfachen Pivotwahlstrategie, die $j = l$ wählt.

2.3.2 Durchschnittliche Laufzeit

Wenn Simple Quick Sort das Feld der Form $[0, 1, \ldots, n-1]$ (für ein $n > 0$) sortiert, dann sieht die rekursive Aufrufstruktur wie folgt aus.

Tafelzusatz Baum der Aufrufstruktur.

Folgerung Die pessimale Laufzeit von Simple Quick Sort ist $\Omega(n^2)$.

Beobachtung Dennoch sortiert Simple Quick Sort in der Praxis recht schnell. Begründung folgt.

Im Folgenden konzentrieren wir uns deshalb auf eine Analyse des durchschnittlichen Laufzeitverhaltens.

Für ein Feld $a[0..n-1]$ legen wir die folgenden Größen fest:

- $q(a) =$ Anzahl der beim Aufruf von SimpleQuickSort(a) durchgeführten Feldvergleiche,
- $p(a) =$ Anzahl der beim Aufruf von Partition$(a, 0, 0, n-1)$ durchgeführten Feldvergleiche,
- $P_l(a) =$ linkes Feldsegment nach der Partitionierung von a,
- $P_r(a) =$ rechtes Feldsegment nach der Partitionierung von a.

Definition Für jedes $n \geq 0$ sei S_n die Menge aller Felder der Länge n, in denen jede Zahl aus $\{0, \ldots, n-1\}$ genau einmal vorkommt.

Dann gilt $|S_n| = n!$.

Definition Für jedes $n \geq 0$ ist die *durchschnittliche Anzahl der von Simple Quick Sort durchgeführten Feldvergleiche* definiert durch

$$q(n) = \frac{1}{n!} \sum_{a \in S_n} q(a) \ . \tag{2.2}$$

Die *durchschnittliche Anzahl der von Partition durchgeführten Feldvergleiche*

ist definiert durch

$$p(n) = \frac{1}{n!} \sum_{a \in S_n} p(a) \ . \tag{2.3}$$

Wir können dazu eine geeignete Rekursionsgleichung aufstellen:

Lemma Es gilt

$$q(0) = 0 \ , \tag{2.4}$$
$$q(1) = 0 \ , \tag{2.5}$$
$$q(n) = p(n) + \frac{1}{n!} \sum_{a \in S_n} (q(P_l(a)) + q(P_r(a))) \qquad \text{für } n > 1. \ . \tag{2.6}$$

Beweis Die beiden ersten Behauptungen ergeben sich sofort aus dem Algorithmus. Zum Beweis der dritten Behauptung überlegt man sich zunächst, dass der Algorithmus die Gleichung

$$q(n) = \frac{1}{n!} \sum_{a \in S_n} (p(a) + q(P_l(a)) + q(P_r(a))) \tag{2.7}$$

für jedes $n > 1$ liefert. Durch Aufspalten der Summe erhält man die Behauptung.

Die obige Beziehung wollen wir im weiteren Verlauf vereinfachen. Dazu führen wir die beiden folgenden Bezeichnungen für ein Feld der Länge n und eine Zahl $x < n$ ein:

$$I_{<x}(a) = \{i < n \mid a[i] < x\} \ , \tag{2.8}$$
$$I_{>x}(a) = \{i < n \mid a[i] > x\} \ . \tag{2.9}$$

Jetzt können wir beweisen:

Lemma Sei $n > 0$, $a \in S_n$ und $x = a[0]$. Dann gibt es zwei Funktionen $f_l \colon \{0, \ldots, x-1\} \to \{1, \ldots, n\}$ und $f_r \colon \{0, \ldots, n-x-1\} \to \{1, \ldots, n\}$ mit folgenden Eigenschaften.

1. Für jedes $i < x$ gilt $P_l(a)[i] = a[f_l(i)]$ und für jedes $i < n - x - 1$ gilt $P_r(a)[i] = a[f_r(i)]$.

2. Sei a' ein Feld der Länge n mit $a'[0] = x$ und $I_{<x}(a') = I_{<x}(a)$ (oder $I_{>x}(a') = I_{>x}(a)$). Dann gilt 1. mit a' anstelle von a.

Beweis Zum Beweis der Behauptung brauchen wir nur zu bemerken, dass die Vertauschungen, die Partition durchführt, nicht von den eigentlichen Zahlen im Feld abhängt, sondern lediglich davon, an welchen Positionen Zahlen stehen, die größer als das Pivotelement sind, und an welchen Stellen Zahlen stehen, die kleiner als das Pivotelement sind. Dazu beachte man, dass in Partition nur Feldvergleiche der Form $a[R] \geq x$ und $a[L] \leq x$ erfolgen.

Aus dem obigen Lemma können wir sofort eine wichtige Folgerung ziehen, wenn wir zusätzlich die folgende Notation benutzen. Es sei S_n^x die Menge aller Felder der Länge $n - x - 1$, in denen jedes Element aus $\{x + 1, \ldots, n - 1\}$ genau einmal vorkommt.

Folgerung Sei $x < n$, $I \subseteq \{1, \ldots, n - 1\}$ mit $|I| = x$, $b_l \in S_x$ und b_r. Dann gibt es genau ein Feld $a \in S_n$ mit $a[0] = 0$ und $I_{<x}(a) = I$, für das $P_l(a) = b_l$ und $P_r(a) = b_r$ gilt.

Daraus können wir nun schließen:

Lemma Sei $n > 1$. Dann gilt

$$q(n) = p(n) + \frac{2}{n} \sum_{x<n} q(x) \quad . \tag{2.10}$$

Beweis Sei $n > 1$. Dann gilt:

$$q(n) = p(n) + \frac{1}{n!} \sum_{a \in S_n} (q(P_l(a)) + q(P_r(a))) \tag{2.11}$$

$$= p(n) + \frac{1}{n!} \sum_{x<n} \binom{n-1}{x} \sum_{b_l \in S_x, b_r \in S_n^x} (q(b_l) + q(b_r)) \ , \tag{2.12}$$

$$= p(n) + \frac{1}{n!} \sum_{x<n} \binom{n-1}{x} (\sum_{b_l \in S_x} x! q(b_l) + \sum_{b_r \in S_n^x} (n-x-1)! q(b_r)) \ , \tag{2.13}$$

$$= p(n) + \frac{1}{n!} \sum_{x<n} (\sum_{b_l \in S_x} x! \binom{n-1}{x} q(b_l) + \sum_{b_r \in S_n^x} (n-x-1)! \binom{n-1}{x} q(b_r)) \ , \tag{2.14}$$

$$= p(n) + \frac{1}{n!} \sum_{x<n} (\sum_{b_l \in S_x} \frac{(n-1)!}{x!} q(b_l) + \sum_{b_r \in S_n^x} \frac{(n-1)!}{(n-x-1)!} q(b_r)) \ , \tag{2.15}$$

$$= p(n) + \frac{1}{n!} (\sum_{x<n, b_l \in S_x} \frac{(n-1)!}{x!} q(b_l) + \sum_{x<n, b_r \in S_n^x} \frac{(n-1)!}{(n-x-1)!} q(b_r)) \ , \tag{2.16}$$

$$= p(n) + \frac{2}{n!} \sum_{x<n, b_l \in S_x} \frac{(n-1)!}{x!} q(b_l) \ , \tag{2.17}$$

$$= p(n) + \frac{2}{n} \sum_{x<n, b_l \in S_x} \frac{1}{x!} q(b_l) \ , \tag{2.18}$$

$$= p(n) + \frac{2}{n} \sum_{x<n} q(x) \ . \tag{2.19}$$

Im nächsten Schritt beschäftigen wir uns mit $p(n)$:

Lemma Sei $n \geq 0$. Dann gilt

$$n \leq p(n) \leq n + 1 \ . \tag{2.20}$$

Beweis Sei $a \in S_n$ beliebig. Wir führen eine Fallunterscheidung durch.

1. Fall, $a[0] = n - 1$. Dann durchläuft L im dritten Schritt von Partition das

gesamte Feld und es werden $n-1$ Feldvergleiche durchgeführt. Im vierten Schritt wird 1 Vergleich durchgeführt, insgesamt also n.

2. Fall, $a[0] < n$. Dann überkreuzen sich die Zeiger L und R innerhalb des Feldes, so dass $n+1$ Feldvergleiche durchgeführt werden.

Es ergibt sich also:

$$p(n) = \frac{1}{n!}((n-1)!n + (n! - (n-1)!)(n+1)) \quad = n + \frac{n! - (n-1)!}{n!} \ , \tag{2.21}$$

woraus die Behauptung folgt.

Wir definieren nun q' durch $q'(0) = 0$ und $q'(n) = n + 1 + \frac{2}{n}\sum_{i<n} q'(n)$ für $n > 0$. Dann gilt:

Bemerkung Für alle n gilt $q(n) \leq q'(n)$.

Beweis per Induktion über n.

Es reicht also, wenn wir eine Abschätzung für q' erzielen.

Lemma Es gilt $q'(n) \in O(n \log n)$.

Beweis Sei $n > 1$. Dann erhalten wir durch Multiplizieren:

$$nq'(n) = n^2 + n + 2\sum_{i<n} q'(i) \ , \tag{2.22}$$

$$(n-1)q'(n-1) = n^2 - n + 2\sum_{i<n-1} q'(i) \ . \tag{2.23}$$

Durch Subtraktion erhalten wir

$$nq'(n) - (n-1)q'(n-1) = 2n + 2q'(n-1) \ , \tag{2.24}$$

was äquivalent ist zu

$$nq'(n) = (n+1)q'(n-1) + 2n \ , \tag{2.25}$$

was wiederum zu

$$\frac{q'(n)}{n+1} = \frac{q'(n-1)}{n} + \frac{2}{n+1} \tag{2.26}$$

äquivalent ist. Durch wiederholtes Einsetzen ergibt sich daraus (formaler Beweis per Induktion):

$$\frac{q'(n)}{n+1} = \frac{q'(1)}{2} + \sum_{3}^{i=n+1} \frac{2}{i} \ . \tag{2.27}$$

Nun gilt aber

$$\sum_{3}^{i=n+1} \frac{2}{i} \leq \int_{2}^{n+1} \frac{1}{x} dx = \log_e(n+1) - \log_e(2) \in O(\log n) \ . \tag{2.28}$$

Daraus folgt $q'(n) \in O(n \log n)$.

Insgesamt:

Satz Quick Sort ist ein nicht-stabiles vergleichsbasiertes in-situ-Feldsortierverfahren, dessen pessimale Laufzeit $\Omega(n^2)$ und dessen durchschnittliche Laufzeit $\emptyset(n \log n)$ ist.

2.4 Heap Sort – Sortieren mit Halden

Ziel Ein vergleichsbasiertes In-place-Verfahren mit pessimaler Laufzeit $O(n \log n)$.

Idee Verwendung einer geeigneten Datenstruktur!

Wir nehmen zunächst an, wir hätten eine Datenstruktur, die ganze Zahlen aufnehmen kann, und die die folgenden Operationen zulässt:

- Erzeugen einer leeren Datenstruktur,
- Einfügen einer ganzen Zahl,
- Extrahieren der größten Zahl.

Eine solche Datenstruktur nennen wir *Prioritätsschlange.*

Dann könnten wir auf einfache Weise (extern) sortieren:

1. Erzeuge eine leere Datenstruktur.
2. Füge nacheinander alle Feldelemente in die Datenstruktur ein.
3. Extrahieren nacheinander jeweils die größte verbleibende Zahl und überschreibe mit diesen Zahlen das Feld von rechts nach links.

2.4.1 Halden

Wir betrachten nun eine konkrete Datenstruktur für eine Prioritätsschlange.

Definition (nicht formal) Eine *(Max-)Halde* ist ein binärer, von links nach rechts ebenengefüllter Baum, dessen Knotenbeschriftungen aus Zahlen bestehen. Zusätzlich gilt für jeden Knoten: Die Beschriftung jedes Kindes ist kleiner gleich der Beschriftung des Knotens.

Tafelzusatz Beispiele und Nicht-Beispiele.

Tafelzusatz Beispiele für Einfügen.

Das Einfügen einer Zahl wird nach folgenden Regeln durchgeführt:

1. Der Baum erhält ein zusätzliches Blatt, das mit der einzufügenden Zahl beschriftet ist.
2. (Hinaufprügeln) Entlang des Pfades zur Wurzel wird die Zahl so lange mit der Beschriftung des Elternknotens vertauscht, bis sie an der Wurzel angelangt ist oder der Elternknoten eine Beschriftung aufweist, die größer oder gleich der einzufügenden Zahl ist.

Pessimale Laufzeit: Höhe des Baumes = $\theta(\log n)$, denn ein ebenengefüllter Binärbaum mit n Knoten besitzt genau $\lfloor \log n \rfloor + 1$ Ebenen.

Tafelzusatz Beispiele Extrahieren.

Das Extrahieren der größten Zahl wird nach folgenden Regeln durchgeführt:

1. Die Wurzelbeschriftung wird ausgegeben.
2. Die Wurzel wird mit der Beschriftung des letzten Knotens (in Ebenenordnung) beschriftet und dieser Knoten wird entfernt.
3. (Hinunterprügeln) Auf dem Weg an die Baumfront wird die neue Beschriftung der Wurzel so lange mit der Beschriftung eines Kindknotens vertauscht, wie es einen Kindknoten mit größerer Beschriftung gibt. Vertauscht wird mit der Beschriftung eines Kindknotens mit maximaler Beschriftung.

Pessimale Laufzeit: $\theta(n)$ (siehe oben).

Pessimale Laufzeit des gesamten Sortierverfahrens:

- Der zweite Schritt hätte Laufzeit $\theta(n \log n)$, weil ungefähr die Hälfte aller Elemente von der Front an die Wurzel gebracht werden müssten. Im Fall $n = 2^k - 1$ sind es genau 2^{k-1} Elemente, die von Ebene k an die Wurzel gebracht werden müssen.
- Der dritte Schritt hätte Laufzeit $\theta(n \log n)$, weil ungefähr die Hälfte aller Elemente von der Wurzel an die Front gebracht werden müssten. Im Fall $n = 2^k - 1$ sind dies genau 2^{k-1} Elemente.

Die Gesamtlaufzeit wäre $\theta(n \log n)$.

Zur Verbesserung des zweiten Schrittes überlegen wir uns, wie man aus n vorgegebenen Elementen schnell eine Halde konstruieren kann, die genau diese Elemente enthält.

Tafelzusatz Beispiele.

Das Erzeugen einer Halde bestehend aus n vorgegebenen Elementen erfolgt nach den folgenden Regeln:

1. Es wird ein von links nach rechts ebenengefüllter Baum mit n Knoten erzeugt, dessen Knoten der Reihe nach mit den vorgegebenen Zahlen beschriftet werden.
2. Nacheinander werden die ersten $\lfloor n/2 \rfloor$ Elemente hinuntergeprügelt.

Pessimale Laufzeit: $O(n + \sum_{i=1}^{\lfloor n/2 \rfloor}(\log i + 1)) = O(n)$ (nach altem Lemma).

2.4.2 Algorithmus

Ziel Implementierung als schnelles In-place-Verfahren.

Problem Die Datenstruktur der Halde ist extern.

Lösung Wir bilden die Datenstruktur in einem Feld ab. Das linke Kind von Feldelement mit Index i wird am Index $2i+1$ und dann rechte Kind am Index $2i+2$ abgelegt.

Definition Sei a ein ganzzahliges Feld der Länge n und gelte $l \leq r < n$. Wir definieren induktiv, wann das Feldsegment $a[l..r]$ eine *(Max-)Halde* ist.

- Falls $2l \geq r$, dann ist $a[l..r]$ eine Halde.
- Falls $2l + 1 = r$ und $a[l] \geq a[r]$, dann ist $a[l..r]$ eine Halde.
- Falls $2l + 2 \leq r$ und $a[l] \geq [2l + 1], a[2l + 2]$ und $a[2l + 1..r]$ sowie $a[2l + 2..r]$ Halden sind, so ist $a[l..r]$ eine Halde.

Tafelzusatz Graphische Veranschaulichung.

HeapSort(a)

Vorbedingung: a ist ein ganzzahliges Feld der Länge n

1. *Haldenaufbau nach der schnellen Methode.*
   Zusicherung: a ist Permutation von $\bar{a}$ und $a[j..n-1]$ ist eine Halde für j mit $(n-2)$ div $2+1 \leq j < n$.

für $i = (n-2)$ div 2 bis 0
DownHeap$(a, i, n-1)$
Zusicherung: $a[j..n-1]$ ist Halde für j mit $i \leq j < n$.
Zusicherung: a ist Permutation von $\bar{a}$ und $a[0..n-1]$ ist eine Halde.

2. *Haldenabbau mit r als aktuellem Haldenende.*
   für $r = n-2$ bis 0
   (a) *Bringe maximales verbliebenes Element (mit Index 0) in Endposition.*
   Swap$(a, 0, r+1)$
   (b) *Bringe Restfeld in Haldenform.*
   (c) DownHeap$(a, 0, r)$
   Zusicherung: a ist eine Permutation von $\bar{a}$; $a[0..r]$ ist eine Halde; $a[r+1..n-1]$ ist geordnet; für alle i und j mit $0 \leq i \leq r$ und $r+1 \leq j < n$ gilt $a[i] \leq a[j]$.

Nachbedingung: a ist geordnete Permutation von $\bar{a}$.

Definition Sei a ein ganzzahliges Feld der Länge n und gelte $l \leq r < n$. Wir definieren, wann das Feldsegment $a[l..r]$ eine *Fasthalde* ist.

- Falls $2l + 1 \geq r$, dann ist $a[l..r]$ eine Fasthalde.
- Falls $2l + 2 \leq r$ und $a[2l+1..r]$ sowie $a[2l+2..r]$ Halden sind, so ist $a[l..r]$ eine Fasthalde.

Definition Seien $l \leq r$. Dann ist $S(l, r)$ induktiv definiert durch:

1. Ist $2l \geq r$, dann ist $S(l, r) = \{l\}$.
2. Ist $2l + 1 = r$, dann ist $S(l, r) = \{l, r\}$.
3. Ist $2l + 2 \leq r$, dann ist $S(l, r) = \{l\} \cup S(2l+1, r) \cup S(2l+2, r)$.

Sei zusätzlich $n > r$. Dann heißt eine Permutation $\pi\colon \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ eine (l, r)-*Permutation*, falls $\pi(i) = i$ für alle $i < n$ mit $i \notin S(l, r)$ gilt.

Ein Feld a der Länge n ist eine (l, r)-Permutation eines Feldes b, wenn es eine (l, r)-Permutation π gibt, so das $b[i] = a[\pi(i)]$ für alle $i < n$ gilt.

DownHeap(a, l, r)

Vorbedingung: a ist ganzzahliges Feld der Länge n; $l \leq r < n$; $a[l..r]$ ist eine Fasthalde.

Es ist nur dann etwas zu tun, wenn mindestens zwei Feldelemente betroffen sind.
falls $2l + 1 \leq r$
(a) setze $j = 2l + 1$
(b) setze $k = 2l + 2$
(c) falls $k \leq r$ und $a[k] > a[j]$
setze $j = k$
Zusicherung: j ist maximales Kind (im Haldensinne).
(d) falls $a[l] < a[j]$
Swap(a, l, j)
DownHeap(a, j, r)

Nachbedingung: $\bar{a}$ ist eine $(\bar{l}, \bar{r})$-Permutation von a; $a[\bar{l}, \bar{r}]$ ist eine Halde.

Satz Heap Sort ist ein vergleichsbasiertes In-Place-Sortierverfahren mit pessimaler Laufzeit $O(n \log n)$.

Beweis Die Korrektheit kann mit Hilfe der Vor- und Nachbedingungen sowie der Zusicherungen nachgewiesen werden. Dass es sich um ein In-Place-Verfahren handelt ist offensichtlich.

Zur Laufzeit des Hinunterprügelns: In jedem rekursiven Aufruf werden maximal zwei Vergleiche durchgeführt. Bei jedem Aufruf von DownHeap(a, l, r) gilt $j \geq 2l+1$ für den inneren rekursiven Aufruf DownHeap(a, j, r). Wenn wir die Folge $l_0, l_1, \ldots, l_k$ der Werte des zweiten Paramaters betrachten, gilt also $l_{i+1} \geq 2l_i + 1$. Daraus ergibt sich per Induktion $l_i \geq (l_0 + 1)2^{i-1}$. Weiterhin wissen wir, dass $k = 0$ oder $2l_{k-1} + 1 \leq r$ gelten muss. Aus Letzterem erhalten wir dann $2(l_0 + 1)2^{k-2} \leq r$, also $(l_0 + 1)2^{k-1} \leq r$, mit anderen Worten $\log(l_0+1)+k-1 \leq \log r$, d. h., $k \leq 1+\log(r/(l_0+1))$. Wir haben also höchstens $2(2+\log(r/(l+1)))$ Vergleiche bei einem Aufruf von DownHeap(a, l, r).

Zur Laufzeit des Haldenaufbaus: Wir nehmen an, dass $2^k \leq n < 2^{k+1}$ gilt.

Die Anzahl der Feldvergleiche ist dann höchstens

$$\sum_{l=0}^{\lfloor n-2/2 \rfloor} 2(2+\log(n/(l+1)) \leq 4n + 2\sum_{l=0}^{n-1} \lceil \log(n/(l+1)) \rceil \tag{2.29}$$

$$\leq 4n + 2\sum_{l=0}^{n-1} \lceil \log(2^{k+1}/(l+1)) \rceil \tag{2.30}$$

$$\leq 4n + 2\sum_{l=0}^{n-1} \lceil k+1-\log(l+1) \rceil \tag{2.31}$$

$$\leq 4n + 2\sum_{j=0}^{k+1} 2^j (k+1-j) \tag{2.32}$$

$$\in O(n) \ , \tag{2.33}$$

wobei der letzte Schritt aus den Betrachtungen im Abschnitt über Größenordnungen folgt.

Zur Laufzeit des Haldenabbaus: Die Anzahl der Feldvergleiche ist höchstens

$$\sum_{r=0}^{n-2} 2\log n \in O(n \log n) \ . \tag{2.34}$$

Insgesamt ergibt sich dann ebenfalls $O(n \log n)$.

2.5 Bewertung der Sortierverfahren

Ziel Wir wollen eine umfassende Bewertung der behandelten Feld-Sortierverfahren durchführen.

Zuerst einmal zu den theoretisch ermittelten Charakteristika:

	Insertion Sort	Quick Sort	Merge Sort	Heap Sort
pessimale Laufzeit	$\theta(n^2)$	$\theta(n^2)$	$\theta(n \log n)$	$\theta(n \log n)$
durchschnittl. Laufzeit	$\theta(n^2)$	$\theta(n \log n)$	$\theta(n \log n)$	$\theta(n \log n)$
in-place	ja	ja	nein	ja
stabil	ja	nein	ja	nein
zus. Speicherbedarf	$O(1)$	$\theta(n)$	$\theta(n)$	$O(1)$

Zusätzlich machen wir Laufzeittests (Angaben in Sekunden) und ermitteln die exakte Anzahl an Vergleichen:

	Quick Sort		Merge Sort		Heap Sort	
100.000	0.29	2.131.703	0.26	1.536.319	0.49	3.019.653
200.000	0.72	6.661.105	0.62	4.808.986	1.25	9.458.967
300.000	1.17	13.800.956	1.00	9.893.672	2.02	19.460.391
400.000	1.65	23.504.418	1.39	16.839.127	2.85	33.138.768
500.000	2.17	35.689.635	1.85	25.676.320	3.93	50.535.909
600.000	2.68	50.803.585	2.23	36.445.582	4.85	71.738.311
700.000	3.13	68.263.383	2.67	49.168.991	5.83	96.804.188
800.000	3.74	88.511.032	3.14	63.860.111	7.05	125.760.878
900.000	4.20	111.501.040	3.57	80.538.435	7.68	158.629.342
1.000.000	4.92	137.761.894	4.02	99.212.510	8.67	195.423.350

Wir erkennen, dass Heap Sort insgesamt schlecht abschneidet. Das bedeutet, Heap Sort wird nur benutzt, wenn man sehr knapp an Speicherplatz ist und das pessimale (quadratische) Verhalten von Quick Sort vermeiden muss. Man hat dann aber kein stabiles Verfahren mehr.

Zum Vergleich von Quick Sort und Merge Sort: Quick Sort ist nur unwesentlich langsamer, benötigt aber mehr Vergleiche. Wenn also die Vergleiche

wirklich schnell ausgeführt werden können, wenn also z. B. Daten primitiven Typs verglichen werden sollen, wird Quick Sort schneller sein (und Stabilität spielt keine Rolle mehr).

Wir vergleichen noch Merge Sort und Insertion Sort auf kleinen Feldern und stellen fest, dass Insertion Sort bis zu Feldern der Größe 17 schneller ist. Es lohnt sich also, eine Fallunterscheidung durchzuführen und bei kleinen Felder Insertion Sort zu nutzen, insbesondere dann, wenn die Vergleiche schnell ausgeführt werden können. Konkret: bei der rekursiven Implementierung von Quick Sort (!) wird irgendwann auf Insertion Sort umgeschaltet.

2.6 Sortieren von Objekten in Java

Ziel Wir wollen (Felder von) Objekten sortieren, nicht einfach nur ganze Zahlen.

Wenn man Objekte sortiert, hängt die Vorgehensweise stark von der jeweils benutzten Programmiersprache ab. Wir wollen uns ansehen, wie man in Java vorgeht.

2.6.1 Die Schnittstelle Comparable

Grundsätzlich (Ausnahmen siehe unten) benutzen wir die *Schnittstelle (interface)* Comparable.

Erinnerung Eine Schnittstelle wird benutzt, um

- Vorgaben für Klassen machen zu können und
- um überprüfen (und signalisieren) zu können, dass eine Klasse die Vorgaben erfüllt.

Die Vorgaben bestehen syntaktisch aus Bestandteilen, die zu implementieren sind. Sie bestehen weiterhin aus entsprechenden Kommentaren, die *Vertrag (contract)* genannt werden und aus von den Methoden zu erfüllenden Vorgaben bestehen.

Beispiel

```
 * @(#)Comparable.java 1.22 03/12/19
 *
 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package java.lang;
```

```
/**
 * This interface imposes a total ordering on the objects of each class that
 * implements it.  This ordering is referred to as the class's <i>natural

 [...]

public interface Comparable<T> {
    /**
     * Compares this object with the specified object for order.  Returns a
    public int compareTo(T o);

  [...]
}
```

Folie javadoc zu Comparable

Beispiel

```
/*
 * @(#)Integer.java 1.90 04/05/11
 *
 * Copyright 2004 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

package java.lang;

[...]

public final class Integer extends Number implements Comparable<Integer> {
    /**

[...]

    public boolean equals(Object obj) {
if (obj instanceof Integer) {
    return value == ((Integer)obj).intValue();
}
return false;
    }
```

```
[...]

    public int compareTo(Integer anotherInteger) {
int thisVal = this.value;
int anotherVal = anotherInteger.value;
return (thisVal<anotherVal ? -1 : (thisVal==anotherVal ? 0 : 1));
    }

[...]

}
```

Wenn man auf allen Objekten vom Typ Comparable drei Relationen α, β und γ definiert durch:

$$o\ \alpha\ o' \qquad \text{gdw.} \qquad \mathsf{o.compareTo(o')} <= 0\ , \tag{2.35}$$

$$o\ \beta\ o' \qquad \text{gdw.} \qquad \mathsf{o.compareTo(o')} = 0\ , \tag{2.36}$$

$$o\ \gamma\ o' \qquad \text{gdw.} \qquad \mathsf{o.compareTo(o')} >= 0\ , \tag{2.37}$$

$$\tag{2.38}$$

Dann wird von der Schnittstelle gefordert: α eine Präordnung, γ die Umkehrrelation von α und β die zugehörige Äquivalenzrelation. α wird auch als die *natürliche Ordnung (natural ordering)* auf all diesen Elementen bezeichnet.

Die natürliche Ordnung heißt *konsistent mit equals,* wenn die Relation δ definiert durch

$$o\ \delta\ o' \qquad \text{gdw.} \qquad \mathsf{o.equals(o')} = \text{true}\ , \tag{2.39}$$

mit β übereinstimmt.

Erinnerung Die Methode equals ist für alle Objekte deklariert, da sie für java.lang.Object deklariert ist und alle Klassen eine Erweiterung dieser Klasse sind.

2.6.2 Java-Sortierverfahren für Objekte und primitive Typen

Frage Wie und wo sammeln wir unsere eigenen Sortierverfahren?

Antwort In einer eigenen Klasse ...

Beispiel

```
// package algsnds;

public class SortingAlgs{

  /**
   * Sorts array using heap sort: not stable, in place, worst-case running
   * time O(n log n); uses O(n) method to build a heap.
   *
   * @param a a <code>Comparable[]</code> value: the array to be sorted
   */
  public static void heapSort(Comparable[] a){
    /* Build heap. */
    for (int i = (a.length-2) / 2; i >= 0; i--)
      downHeap(a, i, a.length-1);
    assert isHeap(a);
    /* Convert heap to sorted array. */
    for (int l = a.length-1; l > 0; l--){
      swap(a, 0, l);
      downHeap(a, 0, l-1);}
    assert sorted(a);}

  /**
   * Pushes an element down an almost heap (embedded in an array) in order to
   * convert it into a heap. Does not change elements other than the ones on
   * the path from the root to the front.
   *
   * @param a a <code>Comparable[]</code> value
   * @param i an <code>int</code> value: root of the almost heap in
   * question; index of element to be pushed down
   * @param l an <code>int</code> value: end of almost heap in question
   */
  private static void downHeap(Comparable[] a, int i, int l){
    if (2 * i + 1 <= l){
      int j = 2 * i + 1;
      int k = 2 * i + 2;
      if (k <= l && a[k].compareTo(a[j]) > 0)
```

```
j = k;
      if (a[i].compareTo(a[j]) < 0){
swap(a, i, j);
downHeap(a, j, l);}}}

  /**
   * Checks whether specified subtree of an array (viewed as a tree) is
   * a heap.
   *
   * @param a a <code>Comparable[]</code> value: root of the subtree in
   * question; index of element to be pushed down
   * @return a <code>boolean</code> value: end of the subtree in question
   */
  private static boolean isHeap(Comparable[] a){
    for (int i = a.length-1; i > 0; i--)
      if (a[(i-1)/2].compareTo(a[i]) < 0)
return false;

    return true;}

  /**
   * Checks if array is sorted. Can be used, in particular, with the
   * assertion facility for checking that sort is implemented correctly.
   *
   * @return a <code>boolean</code> value: true if array is sorted and
   * else false.
   */
  public static boolean sorted(Comparable[] a){
    if (a.length == 0)
      return true;
    else {
      Comparable c = a[0];
      for (int i = 1; i < a.length; i++)
if (c.compareTo(a[i]) <= 0)
  c = a[i];
else
  return false;
      return true;}}

  /**
   * Swaps two elements of an array.
```

```
   *
   * @param a a <code>Comparable[]</code> value: the array in question
   * @param i an <code>int</code> value: one of the two indices
   * @param j an <code>int</code> value: the other one of the two indices
   */
  public static void swap(Comparable[] a, int i, int j){
    Comparable x = a[i];
    a[i] = a[j];
    a[j] = x;}}
```

Andere Variante: abstrakte Klasse Sorter ...

Frage Was stellt die Java API an Sortierverfahren zur Verfügung?

Antwort **Folie** javadoc Arrays

2.6.3 Sortieren mit speziellen Sortierschlüsseln

Frage Was machen wir, wenn wir die Struktur von Sortierschlüsseln (wie bei CountingSort256) ausnutzen möchten?

Antwort Wir können nicht mit Comparable arbeiten, sondern müssen die Schlüssel explizit machen.

Beispiel Bei CountingSort256 benutzen wir eine Schnittstelle WithByteKey als Erweiterung der Schnittstelle Comparable und fordern im Vertrag, dass die induzierte Ordnung mit der durch CompareTo induzierten Ordnung (der natürlichen Ordnung) übereinstimmen sollte.

```
package algsnds;

/**
 * Provides objects with a byte key, intended for sorting.
 *
 * This interface must be implemented in such a way that <code>O.compareTo(O')
 * <= 0</code> iff <code>O.getByteKey() <= O'.getByteKey()</code>.
 *
```

```
 * @author <a href="mailto:wilke-ads@ti.informatik.uni-kiel.de>Thomas Wilke</a>
 * @version 1.0
 */
public interface WithByteKey extends Comparable{

  /**
   * Returns the actual byte key of the object.
   *
   * @return a <code>byte</code> value: actual byte key.
   */
  public byte getByteKey();}
```

Außerdem implementieren wir Counting Sort in unserer Klasse SortingAlgs.

Problem Bei unserem ursprünglichen Counting Sort waren die zu sortierenden Zahlen auch gleichzeitig die Sortierschlüssel. Jetzt sind diese beiden Dinge unabhängig voneinander.

Lösung Außer dem Zähler für die Häufigkeiten gibt es noch ein Feld mit einem Eintrag für jeden Byte-Wert, in dem wir uns merken, wo das nächste Objekt mit dem jeweiligen Schlüssel stehen muss. Das rechnen wir zunächst einmal aus. Dann gehen wir alle Objekte im Ausgangsfeld durch und schreiben sie Schritt für Schritt in ein Ergebnisfeld, abhängig von den Werten in dem Hilfsfeld, die ständig aktualisiert werden. Zum Schluss kopieren wir alles ins Ursprungsfeld zurück.

Beispiel

```
  /**
   * Performs counting sort for objects with a byte key in linear time!!!
   *
   * @param a a <code>WithByteKey[]</code> value
   */
  public void countingSortForByteKeys(WithByteKey[] a){
    // For each key count how often it occurs as a key in the given array.
    int[] c = new int[256];
    for (int i = 0; i < a.length; i++)
      c[a[i].getByteKey() + 128]++;

    // For each key determine the index of the first element with this
```

```
// key in the sorted array.
int[] d = new int[256];
for (int j = 1; j < 256; j++)
  d[j] = d[j-1] + c[j-1];

// Construct a sorted array by moving each object into the right place.
WithByteKey[] b = new WithByteKey[a.length];
for (int i = 0; i < a.length; i++)
  b[d[a[i].getByteKey()+128]++] = a[i];

// Copy the sorted array back into the original one.
for (int i = 0; i < a.length; i++)
  a[i] = b[i];}
```

2.7 Zwei weitere Sortierverfahren

Zum Abschluss wollen wir noch zwei weitere Sortierverfahren betrachten, eins, das die Struktur der Sortierschlüssel ausnutzt, und ein externes.

2.7.1 Radix Exchange Sort

Wir nehmen an, dass die Sortierschlüssel aus Bitvektoren gleicher Länge, etwa k, bestehen. Es sei $a_0 \dots a_{k-1} < b_0 \dots b_{k-1}$ genau dann, wenn es ein $i < k$ gibt, so dass gilt:

- $a[j] = b[j]$ für $j < i$,
- $a[i] = 0$ und $b[i] = 1$.

Diese Ordnung wird auch als die *lexikographische Ordnung* bezeichnet.

Dann kann man zum Sortieren wie folgt vorgehen. Zuerst sortiert man nach dem ersten Bit, indem man ähnlich wie bei Partition mit zwei Zeigern von links und rechts nach innen läuft und Vertauschungen durchführt. Dann wendet man dasselbe Austauschverfahren auf die beiden entstehenden Teilfelder an, mit dem Unterschied, dass nun nach dem zweiten Bit sortiert wird ...

Tafelzusatz Beispiel für Radix Exchange Sort.

RadixExchangeSort(a, k)

Vorbedingung: a ist ein Feld der Länge bestehend aus Bitvektoren der Länge k

Rufe die parametrisierte Fassung auf.
RadixExchangeSort$(a, 0, n-1, 0)$

Nachbedingung: a ist eine geordnete Permutation von $\bar{a}$

ParaRadixExchangeSort(a, k, l, r, i)

Vorbedingung: Übung!

Überprüfe, ob
falls $i < k$ und $l < r$
(a) setze $L = l - 1$ und $R = r + 1$
(b) setze $L = L + 1$ bis $L > r$ oder $a[L][i] = 1$
(c) setze $R = R - 1$ bis $R < l$ oder $a[R][i] = 0$
(d) solange $L < R$
  i. Swap(a, L, R)
  ii. setze $L = L + 1$ bis $a[L][i] = 1$
  iii. setze $R = R - 1$ bis $a[R][i] = 0$
(e) ParaRadixExchangeSort$(a, k, l, R, i + 1)$
(f) ParaRadixExchangeSort$(a, k, L, r, i + 1)$

Nachbedingung: Übung!

Satz Radix Exchange Sort ist ein nicht-stabiles In-place-Sortierverfahren, das Bitvektoren der Länge k in Zeit $O(nk)$ sortiert.

Beweis Den Beweis der Korrektheit sparen wir uns; er kann mit Hilfe der Vor- und Nachbedingungen geführt werden. Dass das Verfahren nicht-stabil und in-situ ist, ist offensichtlich.

Zur Laufzeit: Im Rumpf von ParaRadixExchangeSort werden höchstens $r - l + 3$ Bitabfragen ausgeführt, wenn wir von den rekursiven Aufrufen absehen. Außerdem überkreuzen sich L und R um genau ein Feld, d. h., $R + 1 = L$. Insgesamt können wir dann die Anzahl der Bitvergleiche durch $4(r - l)(k - i) + 3$ nach oben abschätzen. Dies beweisen wir per Induktion über $k - i$, wobei der Induktionsanfang ($i = k$) trivial ist. Mit $c = 4$ und $d = 3$ erhalten wir im Induktionsschritt für die Anzahl der Vergleiche:

$$
\begin{aligned}
&\leq r - l + 3 + c(R - l)(k - i) + d + c(r - L)(k - i) + d \\
&= r - l + 3c(r - l)(k - i) + d + d \\
&= (c(k - i) + 1)(r - l) + d + d \\
&= (c(k - i + 1) - c + 1)(r - l) + d + d \\
&= c(k - i + 1) + d
\end{aligned}
$$

2.7.2 Sortieren von Dateien und Strömen durch Verschmelzen

Sortieren von Dateien und Strömen Wenn große Datenmengen nur extern vorliegen, etwa in Dateien oder Strömen, kann man zum Sortieren ein modifiziertes Merge Sort benutzen.

Wir bezeichnen einen maximalen geordneten Abschnitt einer Datei als *Lauf.*

Die gegebene Datei verteilen wir nun auf zwei Dateien, und zwar Lauf für Lauf. Dann verschmelzen wir jeweils zwei Läufe aus den beiden Hilfsdateien zu einem neuen Lauf und schreiben diesen in die ursprüngliche Datei. Dann hat die Datei höchstens halb so viele Läufe wie ursprünglich. Dieses Verfahren müssen wir $\log n$-mal wiederholen, bis die Datei nur noch aus einem Lauf besteht.

Tafelzusatz Beispiel.

Wichtig: Beim Verschmelzen müssen tatsächlich immer ganze Läufe miteinander verschlossen werden.

Verallgemeinerung: Anstatt auf zwei Dateien zu verteilen kann man auch auf eine andere Zahl von Dateien verteilen.

Satz Durch Verschmelzen können Dateien pessimal in Zeit $O(n \log n)$ sortiert werden.

Kapitel 3

Lineare Datenstrukturen

3.1 Einführung

lineare Datenstruktur = Datenstruktur, bei der die Daten „räumlich“ nebeneinander - wie in einer Folge - angeordnet werden

Prinzipiell zwei Ansätze:

1. Durchnummerieren und Zugriff schnell (in konstanter Zeit) über Index: Feld,
2. Verknüpfen durch Verweise (Referenzen) auf nächstes und möglicherweise vorheriges Objekt: einfach und doppelt verkettete Strukturen (singly, doubly linked).

In der Vorlesung werden wir wichtige abstrakte Datentypen unter Zuhilfenahme von lineare Datenstrukturen implementieren: Keller, Schlangen, Listen.

Erinnerung Abstrakter Datentyp: nur Funktionalität vorgegeben, aber keine Implementierung. Beispiel: Wir haben uns überlegt, wie man den abstrakten Datentyp *Prioritätsschlange* durch Halden implementieren kann.

3.2 Keller (stacks)

Beispiel Arithmetische Ausdrücke auswerten; wohlgeklammerte Ausdrücke erkennen.

Tafelzusatz Beispiele für wohlgeklammerte Ausdrücke und Gegenbeispiele.

Tafelzusatz Animation des Algorithmus unter Benutzung eines Kellers.

Frage Wie sieht ein geeigneter ADT aus?

Antwort Der gesuchte ADT sollte Objekte aufnehmen und wieder abgeben können und zwar nach dem *LIFO-Prinzip:* last in, first out. Das zuletzt aufgenommene Objekt wird als erstes wieder abgegeben.

Der ADT des *Kellers* arbeitet nach dem LIFO-Prinzip und soll die folgenden Operationen zulassen:

- isEmpty(),
- pushItem(o): im Keller ablegen,
- popItem(): aus dem Keller holen.

Implementierung Durch einfach verkettete Container. Neu aufgenommene Objekte werden an den Anfang der Kette gesetzt.

Tafelzusatz Veranschaulichung.

ContainerWithLink

- Object item
- ContainerWithLink successor
- Konstruktur newContainerWithLink(o, s)
  setze item = o
  setze successor = s

Bemerkung Algorithmen: Wir wählen einen objektorientierten Ansatz. Die Algorithmen werden als Methoden an Objekte gebunden. Syntax Java-ähnlich, aber ohne „syntactic sugar“, auch keine Vererbung.

LinkedStack

- ContainerWithLink top
- Konstruktor LinkedStack()
  setze top = null
- Methode isEmpty()
  gib top = null zurück
- Methode popItem()
  falls isEmpty()
  FEHLER!
  sonst
  * setze o = top.item
  * top = top.successor
  * gib o zurück
- Methode pushItem(o)
  top = newContainerWithLink(o, top)

Bemerkung Beim Löschen wird der zu löschende Container nicht explizit zerstört. Die Programmierinfrastruktur (Compiler, Interpreter) sorgt in der Regel dafür, dass nicht mehr benötigter Speicherplatz (Speicherplatz, auf den nicht mehr verwiesen wird) gelöscht bzw. freigegeben wird.

Satz Bei der Implementierung LinkedStack eines Kellers haben alle Operationen eine pessimale Laufzeit von $\theta(1)$.

WellFormedExpressions(a)

Vorbedingung: a ist ein Feld der Länge n, dessen Elemente Klammersymbole sind.

1. *Lege Keller an.*
   setze stack = newLinkedStack()
2. *Initialisiere (Zwischen-)Ergebnis und Zähler.*
   setze ok = true

setze $i = 0$

3. *Durchlaufe die Eingabe von links nach rechts.*
   Invariante: ...
   solange ok and $i < n$
   falls $a[i]$ öffnende Klammer ist
   stack.pushItem($a[i]$)
   sonst
   falls stack.isEmpty()
   setze ok = false
   sonst
   setze p = stack.popItem()
   falls p und $a[i]$ nicht zueinander passen
   setze ok = false
   setze $i = i + 1$
   falls not stack.isEmpty()
   setze ok = false

Nachbedingung: ok = true genau dann, wenn $\bar{a}$ ein wohlgeklammerter Ausdruck ist.

Satz Die pessimale Laufzeit des Algorithmus WellFormedExpressions ist $\theta(n)$.

Beweis Zum Beweis bemerken wir zunächst, dass jede Ausführung des Rumpfes der Schleife pessimal Zeit $O(1)$ benötigt, da es in ihm keine Schleifen gibt und die einzigien komplexen Operationen solche sind, die auf den Keller zugreifen. Von diesen haben wir aber bewiesen, dass sie pessimal in Zeit $O(1)$ durchgeführt werden. Außerdem wird durch das Erhöhen von i in jedem Schleifendurchlauf und die entsprechende Abrage $i < n$ sowie die Initialisierung $i = 0$ garantiert, dass die Schleife nur n-mal durchlaufen wird.

Bemerkung Es gibt viele weitere Anwendungen ...

3.3 Schlangen (Queues)

Motivation Häufig möchte man Daten nur (effizient) zwischenspeichern. Dann benutzt man eine so genannte Schlange.

Frage Wie sieht ein geeigneter ADT aus?

Antwort Der gesuchte ADT sollte Objekte aufnehmen und wieder abgeben können und zwar nach dem *FIFO*-Prinzip: first in, first out. Das zuerst aufgenommene Objekt wird als erstes wieder abgegeben. Wie man es aus dem Supermarkt gewohnt ist, aber im Gegensatz zur Supermarktschlange kann aber ein Objekt mehrfach in eine Schlange einsortiert werden.

Der ADT der *Schlange* arbeitet nach dem FIFO-Prinzip und lässt die folgenden Operationen zu:

- isEmpty(),
- enqueueItem(o): anstellen in der Schlange,
- dequeueItem(): aus der Schlange herausgehen.

Implementierung Wieder durch einfach verkettete Container. Jetzt mit einer Referenz auf den Container des zuerst eingegangenen Objektes, um das nächste Objekt schnell entnehmen zu können, und einer Referenz auf den Container des zuletzt eingegangenen Objektes, um ein Objekt schnell einstellen zu können.

Tafelzusatz Graphische Darstellung.

Problem Wenn die Schlange nur noch ein Element enthält und dieses entnommen wird ...

Lösung Wir setzen einen so genannten Markierungscontainer (sentinel container) ohne Objekt an das Ende der Schlange und vereinbaren.

Tafelzusatz Graphische Darstellung.

SinglyLinkedQueue

- ContainerWithLink front
- ContainerWithLink back
- Konstruktor newSinglyLinkedQueue()
  setze front = newContainerWithLink(null,null)
  setze back = front
- Methode isEmpty()
  gib front = back zurück
- Methode dequeueItem()
  falls isEmpty()
    FEHLER!
  sonst
    setze o = front.item
    setze front = front.successor
    gib o zurück
- Methode enqueueItem(o)
  setze back.item = o
  setze back.successor = newContainerWithLink(null,null)
  setze back = back.successor

Satz Bei Implementierung SinglyLinkedQueue einer Schlange haben alle Operationen eine pessimale Laufzeit von $\theta(1)$.

Beispiel Sortieren von Objekten mit Sortierschlüssel in $\{0, \ldots, 255\}$: *Sortieren mit Eimern (bucket sort).*

In der ersten Phase wird das gegebene Feld einmal von links nach rechts durchlaufen. Dabei werden die Feldelemente in ein Feld der Länge 256 geschrieben, wobei jedes Feldelement aus einer Schlange besteht: Für jeden Sortierschlüssel gibt es eine Schlange, in der alle zugehörigen Objekte gesammelt werden.

In der zweiten Phase werden die Objekte der Reihe nach aus den Schlangen entnommen und in das Ergebnisfeld geschrieben.

Tafelzusatz Graphische Veranschaulichung ...

BucketSort256(a)

Vorbedinung: a ist ein Feld der Länge n von Objekten mit Sortierschlüssel sortingKey $\in \{0, \ldots, 255\}$.

1. *Lege Hilfsfeld an.*
   setze b = newArray[256]
   für $j = 0$ bis 255
       setze $b[j]$ = newSinglyLinkedQueue()
2. *Schreibe Objekte in das Hilfsfeld.*
   für $i = 0$ bis $n - 1$
       $b[a[i]$.sortingKey].enqueueItem($a[i]$)
3. *Schreibe Objekte ins Ausgangsfeld zurück.*
   setze $i = 0$
   für $j = 0$ bis 255
       solange not $b[j]$.isEmpty()
           setze $a[i] = b[j]$.dequeueItem()
           setze $i = i + 1$

Nachbedingung: a ist eine geordnete Permutation von $\bar{a}$.

Satz BucketSort256 ist ein stabiles Sortierverfahren für Objekte mit Sortierschlüssel in $\{0, \ldots, 255\}$, dessen pessimale Laufzeit $\theta(n)$ ist.

Beweis Dass es sich um einen stabiles Sortierverfahren handelt, liegt an der FIFO-Eigenschaft der Schlange. Die pessimale Laufzeit kann wie folgt abgeschätz werden.

Die erste Schleife hat pessimale Laufzeit $O(1)$, da sie 256 Mal durchlaufen wird und der Rumpf pessimal in konstanter Zeit durchlaufen wird, denn das Anlegen einer Schlange benötigt nur konstante Zeit.

Die zweite Schleife hat Laufzeit $\theta(n)$, da sie n Mal durchlaufen wird und die Ausführung des Rumpfes in Zeit $\theta(1)$ erfolgt.

Die dritte Schleife ist etwas schwieriger zu analysieren. Wichtig ist, zunächst festzuhalten, dass die Gesamtlänge aller Schlangen im Feld b vor eintritt in die Schleife n ist. Dann wird der Rumpf der inneren Schleife genau n Mal durchlaufen, der Rumpf der äußeren genau 256 Mal. Also ergibt sich auch hier die Laufzeit $\theta(n)$.

Insgesamt ergibt sich ebenfalls $\theta(n)$.

3.4 Listen (lists)

Unter einer Liste wollen wir einen ADT verstehen, der in etwa so funktioniert wie ein einzeiliger Editor: Die Daten werden in einer Folge abgelegt, es gibt einen Positionszeiger (cursor), der nach links und rechts bewegt werden kann, und es können Löschungen und Einfügungen vorgenommen werden. Der Positionszeiger soll, wie bei einem Editor üblich, nicht auf einen Container zeigen, sondern zwischen zwei Container oder vor den ersten oder vor den letzten.

Tafelzusatz Graphische Veranschaulichung.

Genauer: Der abstrakte Datentyp (ADT) der *unidirektionalen Liste* soll die folgenden Operationen zulassen:

1. isEmpty(),
2. cursorIsAtFirstPosition(),
3. cursorIsAtLastPosition(),
4. moveCursorToFirstPosition(),
5. moveCursorToNextPosition(),
6. getItem(),
7. setItem(o),
8. insertItem(o),
9. deleteItem().

Implementierung Wieder durch einfach verkettete Container.

Tafelzusatz Veranschaulichung.

Problem Es bietet sich an, einen Container als Cursor zu benutzen. Zwei Möglichkeiten: Container steht für Position davor oder dahinter. Dann kann man aber die letzte oder die erste Position nicht repräsentieren.

Lösung Deshalb setzen wir wieder einen so genannte Markierungscontainer ein.

Außerdem fügen wir zunächst geeignete Methoden zu unseren Containern hinzu:

ContainerWithLink

...

- Konstruktor newContainerWithLink(o, s)
  Erzeuge neuen Container, der Objekt o hält und auf c zeigt.
  setze item = o
  setze successor = s
- Methode insertAfter(o)
  Füge einen neuen Container, der das Objekt o hält, zwischen dem aktuellen Container und dessen Nachfolger ein. Falls kein Nachfolger existiert, hänge neuen Container an.
  setze successor = newContainerWithLink(o, successor)
- Methode deleteSuccessor()
  Verbinde aktuellen Container mit dem Nachfolger des Nachfolgers. Falls dieser nicht existiert, lösche die Verbindung zum Nachfolger.
  Überprüfe, ob Nachfolger existiert.
  falls successor $\neq$ null
  Verbinde aktuellen Container mit übernächstem.
  setze successor = successor.successor

Tafelzusatz Beispiele.

Nun zur Listenimplementierung:

SinglyLinkedUnidirectionalList

- ContainerWithLink leftSentinel
- ContainerWithLink lastContainer
- ContainerWithLink cursor

- Konstruktor newSinglyLinkedUnidirectionalList()
  setze leftSentinel = newContainerWithLink(null,null)
  setze lastContainer = leftSentinel
  setze cursor = leftSentinel
- Methode isEmpty()
  gib leftSentinel = lastContainer zurück
- Methode cursorIsAtFirstPosition()
  gib cursor = leftSentinel zurück

  ...
- Methode getItem()
  falls cursor = lastContainer
    gib Fehlermeldung aus
  sonst
    gib successor.item zurück
- Methode setItem(o)
  falls cursor = lastContainer
    gib Fehlermeldung aus
  sonst
    setze cursor.successor.item = o
- Methode insertItem(o)
  cursor.insertAfter(o)
  falls cursor = lastContainer
    setze lastPosition = cursor.successor

  ...

Satz Bei der Implementierung SinglyLinkedUnidirectionalList einer unidirektionalen Liste benötigen alle Operationen ein pessimale Laufzeit von $\theta(1)$.

Beispiel Sieb des Eratosthenes:

Sieve(n)

Vorbedingung: $n \geq 2$ ist eine natürlich Zahl

1. *Erzeuge aufsteigende Liste mit allen Zahlen* $\leq n$ *und* > 1.
   setze l = newSinglyLinkedUnidirectionalList()
   setze $m = 2$

solange $m \leq n$
l.insertItem(m)
l.moveCursorToNextPosition()
setze $m = m + 1$

2. *Initialisiere Ergebnisliste.*
   setze r = newSinglyLinkedUnidirectionalList()
3. *Bestimme schrittweise gesuchten Primzahlen.*
   Invariante: ...
   solange nicht l.isEmpty()
   (a) *Bestimme nächste Primzahl.*
   l.moveCursorToFirstPosition()
   setze $p = l$.getItem()
   (b) *Füge Primzahl zur Ergebnisliste hinzu.*
   r.addItem(m)
   r.moveCursorToNextPosition()
   (c) *Eliminiere Vielfache (Siebschritt).*
   solange nicht l.cursorIsAtLastPosition()
   falls p teilt l.getItem()
   l.deleteItem()
   sonst
   l.moveCursorToNextPosition()

Nachbedingung: r ist eine Liste, die alle Primzahlen $\leq n$ in aufsteigender Reihenfolge enthält.

Der ADT der *bidirektionalen Liste* lässt zusätzlich zu den Operationen der unidirektionalen Liste auch noch zu:

- moveCursorToLastPosition(),
- moveCursorToPreviousPosition().

Problem Es wäre äußerst aufwändig, den Cursor eine Position nach links zu bewegen, wenn wir mit einer einfach verketteten Datenstruktur arbeiten würden.

Lösung Wir benutzen ContainerWithTwoLinks. Dann werden das Einfügen wie auch das Löschen etwas komplizierter, aber wir können dadurch auch leicht einen Schritt zurück gehen.

Tafelzusatz Graphische Veranschaulichung.

Satz Bei der Implementierung einer bidirektionalen Liste durch Zweifachverkettung haben alle Operationen eine pessimale Laufzeit von $\theta(1)$.

3.5 Lineare Datenstrukturen in Java

3.5.1 Eigene Implementierungen

Problem In Java müssten wir eigentlich Keller von Integer-Objekten, Keller von String-Objekten, ... haben, um nicht ständig Typanpassungen vornehmen zu müssen. Alle Keller funktionieren gleich, weshalb wir den Algorithmus eigenlich nur einmal implementieren möchten.

Lösung *Generische Klassen:* Generische Klassen haben Typvariablen als Parameter. Belegt man die Variable mit einem Typ, so erhält man eine parametrisierte Klasse. Bsp.: `LinkedStack<T>` ist eine generische Klasse, `LinkedStack<Integer>` eine parametrisierte.

Die Typvariable kann benutzt werden, um den Typ von Variablen zu deklarieren, um den Rückgabetyp einer Methode zu deklarieren, usw.

```
package algsnds;

class ContainerWithLink<T>{

  private T item;
  private ContainerWithLink<T> successor;

  protected ContainerWithLink(T t, ContainerWithLink<T> s){
    item = t;
    successor = s;}

  protected T getItem(){
    return item;}

  protected ContainerWithLink<T> getSuccessor(){
    return successor;}

  protected void setSuccessor(ContainerWithLink<T> c){
    successor = c;}

  protected void deleteSuccessor(){
    if (successor == null)
```

```
      throw new NoSuccessorException();
    else
      successor = successor.getSuccessor();}

  protected void insertAfter(T t){
    if (successor == null)
      successor = new ContainerWithLink<T>(t, null);
    else
      successor = new ContainerWithLink<T>(t, getSuccessor());}}
```

Problem Wir möchten sicher gehen, dass die Implementierungen von ADT, die wir erstellen, wenigstens syntaktisch alle Erfordernisse erfüllen, die an einen ADT gestellt werden.

Lösung Wir benutzen geeignete Schnittstellen, in denen die Operationen, die ein ADT zur Verfügung stellt, festgelegt werden.

```
package algsnds;

public interface Stack<T>{

  public boolean isEmpty();

  public void pushItem(T t);

  public T popItem();}
```

Problem Wir müssen Fehler vernünftig behandeln! Zum Beispiel, wenn der Nachfolger eines Containers gelöscht werden soll, ein solcher aber gar nicht existiert.

Lösung Wir benutzen geeignete *Ausnahmen (exceptions).* Der Einfachheit halber lassen wir das Programm terminieren. Deshalb benutzen wir eine Erweiterung der Ausnahme RuntimeException.

```
package algsnds;

public class NoSuccessorException extends RuntimeException{
```

```
  public NoSuccessorException(){
    super();}}
```

Die eigentliche Implementierung der Klasse LinkedStack ist sehr nah an der Darstellung in der Vorlesung:

```
package algsnds;

public class LinkedStack<T> implements Stack<T>{

  private ContainerWithLink<T> top;

  public LinkedStack(){
    top = null;}

  public boolean isEmpty(){
    return (top == null);}

  public void pushItem(T t){
    top = new ContainerWithLink<T>(t, top);}

  public T popItem(){
    if (isEmpty())
      throw new StackEmptyException();
    else {
      T t = top.getItem();
      top = top.getSuccessor();
      return t;}}

  public String toString(){
    if (isEmpty())
      return "";
    else {
      String s = top.toString();
      for (ContainerWithLink c = top.getSuccessor(); c != null;
   c = c.getSuccessor())
s = s + " " + c.getItem().toString();
      return s;}}}
```

3.5.2 Java-API

Beamerzusatz Java-Doku!

Besonders zu bemerken:

- ADT in Form von Schnittstellen vorgegeben
- Standardimplementierung der ADT in Form von Klassen vorgegeben
- Besonderheit: Iteratoren + passende for-Schleife

3.6 Feldimplementierungen und amortisierte Laufzeiten

Frage Ist unsere Implementierung einer Schlange wirklich gut?

Antwort Es gibt bessere!

Idee Wir legen die Daten nicht in Containern, sondern in einem Feld ab; wenn das Feld zu klein wird, schreiben wir die Daten in ein größeres Feld.

Tafelzusatz Beispiel für Feldimplementierung.

Implementierung Wir merken uns den Anfang der Schlange in frontOfQueue und die Größe der Schlange in size. Wenn das Feld zu klein wird, legen wir ein größeres an und kopieren die Daten in das neue Feld (und löschen das alte). Wenn das Feld zu groß wird, legen wir ein kleineres an, kopieren die Daten in das neue Feld und löschen das alte Feld, damit wir nicht zu viel Platz verschwenden. Wenn die Schlange über die obere Feldgrenze hinauswächst, nutzen wir den Anfang des Feldes, sofern dieser nicht belegt ist (modulo-Rechnung). Wir nennen letzteres auch auch *zirkuläre Implementierung.*

Frage Wie groß sollen wir das Feld wählen, wenn es zu klein wird? Wann sollen wir das Feld um wieviel verkleinern?

Antwort Eine gute Variante besteht aus Verdopplung und Halbierung, bei Unterschreitung eines Viertels der Größe. Erklärung erfolgt später.

ArrayQueue

- Array array
- int front
- int size
- int minLength
- Konstruktor newArrayList()
  setze minLength = 16
  setze array = newArray[minLength]

setze size = 0
setze front = 0

Methode isEmpty()
gib size = 0 zurück

Methode dequeueItem()
falls isEmpty()
FEHLER!
sonst
setze o = array[front]
setze size = size - 1
setze front = front + 1 mod array.length
falls minLength < size < array.length div 4
Halbierung des Feldes bei Unterschreitung eines Viertels.
setze a = newArray[array.length div 2]
toNew(a)
gib o zurück

- Methode enqueueItem(o)
falls size = array.length
Verdopplung des Feldes wegen Überlauf.
setze a = newArray[array.length $*$ 2]
toNew(a)
enqueueItem(o)
Eigentliches Anstellen.
setze array[front + size mod array.length] = o
setze size = size + 1

- Methode toNew(a)
Kopiere Inhalt der Schlange in neues Feld a.
für $i = 0$ bis size $-$ 1
setze $a[i]$ = array[front + i mod array.length]
setze array = a
setze front = 0

Leicht einzusehen:

Satz Der Platzverbrauch der Feldimplementierung der Schlange ist zu jedem Zeitpunkt $\theta(n)$, wobei n die Anzahl der in der Schlange gespeicherten Objekte ist (und der Platz für die Objekte wie üblich nicht mitgerechnet wird).

Beweis Wir halten zunächst fest, dass aufgrund der Implementierung die beiden folgenden Ungleichungen immer gelten:

$$\text{array.length} \leq \max\{16,\ 4 \times \text{size}\}\ , \tag{3.1}$$

$$\text{array.length} \geq \text{size}\ . \tag{3.2}$$

Daraus folgt sofort die Behauptung.□

Leicht einzusehen:

Bemerkung Das Hinzufügen eines Objektes kann durch die Feldvergrößerung beliebig viel Zeit in Anspruch nehmen, da möglicherweise eine Feldverdopllung vorgenommen werden muss. Analoges gilt für das Entfernen eines Objektes.

Beobachtung Bevor eine Verdopplung durchgeführt wird, sind viele Operationen mit kurzer Laufzeit nötig. Vermutung: Wir können eigentlich so tun, als würde jede Operation „im Schnitt“ konstante Laufzeit benötigen.

Satz Bei der Feldimplementierung einer Schlange beträgt die pessimale Gesamtlaufzeit für die Hintereinaderausführung von m Operationen einschließlich der anfänglichen Erzeugung der Schlange $\theta(m)$.

Der Beweis ist etwas aufwändiger und wir führen ihn nicht durch.

Wir können uns also vorstellen, jede Operation habe eine Laufzeit von $\theta(1)$. Dann erhalten wir die pessimale Gesamtlaufzeit, indem wir einfach die Summe der vorgestellten (fiktiven) Einzellaufzeiten bilden.

Sprechweise Wir sagen, die *amortisierte Laufzeit* jeder Operation ist $\theta(1)$.

Kapitel 4

Verzweigte Datenstrukturen

4.1 Einführung

Lineare Datenstrukturen sind in vielen Fällen nicht ausreichend, um schwierige Datenverwaltungsprobleme zu lösen, aber die einzige Möglichkeit, wenn die zu verwaltenden Daten nicht weiter strukturiert sind. Wenn diese jedoch geordnet sind, gibt es die Möglichkeit, die Daten in verzweigten Datenstrukturen, insbesondere Binärbäumen, geordnet abzulegen, um sie dann effizienter verwalten zu können.

Wichtigste Eigenschaft von Binärbäumen: Die Länge der Äste von „höhenbalancierten“ Binärbaum mit n Knoten ist $O(\log n)$. Wenn die Daten also in „höhenbalancierten“ Bäumen abgelegt würden und für eine Operation im wesentlichen ein Ast durchlaufen werden müsste, käme man auf logarithmische Laufzeiten.

Implementierung wie üblich durch Container, jetzt mit zwei oder drei Verweisen:

- einfach verkettet: jeder Container hat linken und rechten Nachfolger,
- doppelt verkettet: jeder Container hat linken und rechten Nachfolger und Vorgänger.

ContainerWithTwoSuccessors

- ContainerWithTwoSuccessors leftSuccessor
- ContainerWithTwoSuccessors rightSuccessor
- WithSortingKey item

In der Vorlesung werden wichtige abstrakte Datentypen unter Zuhilfenahme von verzweigten Datenstrukturen implementiert, insbesondere Prioritätsschlangen und Wörterbücher.

Ein *Wörterbuch* ist ein ADT, der Objekte mit Sortierschlüsseln verwaltet, und die folgenden Operationen zulässt:

- findItem(k): gibt Element mit Schlüssel k zurück,
- addItem(o): fügt Element o (mit Sortierschlüssel o.sortingKey) hinzu
- deleteItem(k): entfernt Element mit Sortierschlüssel k

Dabei wird vereinbart, dass ein Wörterbuch keine zwei Objekte mit demselben Sortierschlüssel aufnehmen kann. Eine entsprechender addItem-Aufruf führt zu einem Fehler.

Einfache Implementierungen. Es gibt zwei einfache Möglichkeiten, Wörterbücher zu implementieren:

- 1. Variante: durch eine unsortierte Liste,
- 2. Variante: durch ein sortiertes Feld.

Tafelzusatz Beispiel für beide Implementierungen.

Bei der Implementierung durch eine unsortierte Liste kann das Hinzufügen in konstanter Laufzeit erfolgen (wenn man einmal von dem Problem absieht, dass Mehrfachvorkommen eines Schlüssels nur durch eine vorangehende Suche vermieden bzw. identifiziert werden), während das Suchen im pessimalen Fall lineare Laufzeit benötigt. Durchschnittlich benötigt das Finden sogar lineare Zeit.

Bei der Implementierung durch ein sortiertes Feld kann zwar in Zeit $\log n$ gefunden werden. Ebenso kann in Zeit $\log n$ festgestellt werden, wo ein einzufügendes Element eingefügt werden muss, für das Einfügen selbst müssen aber viele Feldelemente nach links oder rechts verschoben werden. Nimmt man zum Beispiel an, dass ein neues Objekt mit gleicher Wahrscheinlichkeit zwischen zwei vorhandene Objekte oder an den Anfang oder ans Ende kommt, dann ist der Erwartungswert für die Laufzeit des Einfügens bei einer zirkulären Implementierung auch $\theta(n)$.

Unter Benutzung von verzeigten Datenstrukturen kann man Implementierungen von Wörterbüchern finden, bei den alle Operationen pessimal in Zeit $O(\log n)$ laufen, während die Speichernutzung bei $O(n)$ liegt.

Wir werden dies in der Vorlesung nicht ganz erreichen, aber fast ...

4.2 Binärbäume (binary trees) und Durchlaufstrategien

Wir wollen Bäume „in der Praxis“ zwar aus Containern zusammensetzen und in den Algorithmen auch Container in der üblichen Form verwenden, bei der Modellierung wollen wir jedoch einen etwas mathematischeren Zugang wählen.

Mit $\square$ bezeichnen wir den *Nullknoten.* Wir nehmen an, U ist eine unendliche Menge von *eigentlichen Knoten,* die $\square$ nicht enthält. Mit U' bezeichnen wir $U \cup \{\square\}$, die Menge der *Knoten.* Außerdem gehen wir davon aus, dass eine Menge M von *Markierungen* sowie partielle Funktionen $\lambda\colon U \to U'$, $\rho\colon U \to U'$ und $l\colon U \to M$ gegeben sind.

Tafelzusatz Graphische Veranschaulichung.

Bemerkung Die Menge U entspricht der Menge aller potenziellen Container, M der Menge aller potenziellen Objekte, λ und ρ entsprechen left- bzw. rightSuccessor, l entspricht item, ...

Wir bezeichnen mit W das Tripel (λ, ρ, l) und nennen es die *(aktuelle) Umgebung.*

Die Menge der Knoten, die *einen Baum repräsentieren* (bezüglich W), ist induktiv definiert durch:

- Der Nullknoten $\square$ repräsentiert den leeren Baum und $N^W(\square) = \emptyset$.
- Ein eigentlicher Knoten v repräsentiert einen Baum, falls
  - $\lambda(v)$ und $\rho(v)$ definiert sind,
  - $\lambda(v)$ und $\rho(v)$ Bäume repräsentieren und
  - $N^W(\lambda(v)) \cap N^W(\rho(v)) = \emptyset$ sowie $v \notin N^W(\lambda(v)) \cup N^W(\rho(v))$ gelten.

  Dann ist $N^W(v) = N^W(\lambda(v)) \cup N^W(\rho(v)) \cup \{v\}$.

Die Menge $N^W(v)$ bezeichnet die Menge aller Nachfahren von v. Ein Knoten v *repräsentiert einen unbeschrifteten Baum* (bezüglich W), wenn v einen Baum bezüglich W repräsentiert und $l(v')$ für alle $v' \in N^W(v)$ undefiniert ist. Analog: *beschrifteter Baum.*

Tafelzusatz Graphische Veranschaulichung, Begriffe Wurzel, Blätter, Kno-

ten, innere Knoten, ...

Konvention Der Einfachheit halber nennen wir Knoten auch Bäume. Die Menge $N^W(t)$ heißt dann auch die Menge der Knoten eines Baumes t bzw. die Menge der Teilbäume eines Baumes t.

Größe und Höhe von Bäumen sind induktiv definiert, genau so wie die Menge der Beschriftungen (für beschriftete Binärbäume):

$$h^W(\Box) = s^W(\Box) = 0 \ , \tag{4.1}$$
$$\mathrm{lab}^W(\Box) = \{\ \}, \tag{4.2}$$
$$h^W(t) = \max(h^W(\lambda(t)), h^W(\rho(t))) + 1 \ , \tag{4.3}$$
$$s^W(t) = s^W(\lambda(t)) + s^W(\rho(t)) + 1 \ , \tag{4.4}$$
$$\mathrm{lab}^W(t) = \{l(t)\} \cup \mathrm{lab}^W(\lambda(t)) \cup \mathrm{lab}^W(\rho(t)) \ . \tag{4.5}$$

Konvention Sofern es nicht zu Verwechslungen kommen kann, lassen wir die Umgebung W weg.

Algorithmus zur Berechnung der Höhe eines Baumes:

Height(t)
Vorbedingung: t ist ein Baum.
    Überprüfe, ob Rekursion abgebrochen wird.
    falls t = null
        gib 0 zurück
    sonst
        setze h_l = Height(t.leftSuccessor)
        setze h_r = Height(t.rightSuccessor)
        setze $h = \max\{h_l, h_r\} + 1$
        gib h zurück
Nachbedingung: Es gilt $W = \bar{W}$ und der Rückgabewert ist $h^W(\bar{t})$.

Satz Die Größe und die Höhe eines Baumes können in Zeit $\theta(n)$ berechnet werden (wobei n die Größe des jeweiligen Baumes bezeichne).

Beweis Die Korrektheitsbeweis für den Algorithmus Height ist leicht, denn er orientiert sich direkt an der induktiven Definition von h.

Zur Laufzeit: Da der Rumpf des rekursiven Algorithmus keine Schleifen enthält, erfolgt seine Ausführung in konstanter Zeit. Wir müssen also nur noch die Anzahl der rekursiven Aufrufe des Algorithmus bestimmen.

Für jeden Knoten wird der Algorithmus zweimal aufgerufen, nämlich für seinen linken und seinen rechten Nachfolger. Mit anderen Worten, es gibt $2n$ rekursive Aufrufe. Daraus folgt die Behauptung.

Für die Größe eines Baumes sind der Algorithmus und dessen Analyse ähnlich.□

Häufig sind wir daran interessiert, die Knoten eines Baumes in geeigneter Weise aufzuzählen. Es bieten sich dazu insgesamt vier natürliche Auflistungen an. Für alle diese Auflistungen gilt, dass die Auflistung der Beschriftungen des leeren Baums die leere Folge ist.

Tafelzusatz Beispiele.

Flachordnung Die Auflistung Knoten in Flachordnung (in order) ist definiert durch

$$l_{in}(t) = l_{in}(\lambda(t))tl_{in}(\rho(t)) \ . \tag{4.6}$$

Verzeichnisordnung Die Auflistung der Knoten in Verzeichnisordnung (pre order) ist definiert durch

$$l_{pre}(t) = tl_{pre}(\lambda(t))l_{pre}(\rho(t)) \ . \tag{4.7}$$

Tiefenordnung Die Auflistung der Knoten in Tiefenordnung (post order) ist definiert durch

$$l_{post}(t) = l_{post}(\lambda(t))l_{post}(\rho(t))t \ . \tag{4.8}$$

Ebenenordnung Die Auflistung der Knoten in Ebenenordnung (level order) ist schwieriger zu definieren. Dazu wird für jeden Baum induktiv eine Liste der Beschriftungen der Tiefe i definiert:

$$l_{lev_0}(t) = t \ , \tag{4.9}$$

$$l_{lev_{i+1}}(t) = \begin{cases} l_{lev_i}(\lambda(t))l_{lev_i}(\rho(t)) & \text{falls } h(\lambda(t)), h(\rho(t)) \geq i+1 \\ l_{lev_i}(\lambda(t)) & \text{falls } h(\lambda(t)) \geq i+1 \text{ und } h(\rho(t)) \leq i \\ l_{lev_i}(\rho(t)) & \text{falls } h(\lambda(t)) \leq i \text{ und } h(\rho(t)) \geq i+1 \end{cases} \tag{4.10}$$

Die Auflistung in Ebenenordnung ist dann einfach

$$l_{lev_0}(t) l_{lev_1}(t) l_{lev_2}(t) \dots l_{lev_{h(t)-1}}(t) \ . \tag{4.11}$$

Algorithmus zur Berechnung der Flachordnung:

FlatOrder(t)

Vorbedingung: t ist ein Baum

Überprüfe, ob Rekursion abgebrochen wird.
falls t = null
gib zurück
sonst
setze f_l = FlatOrder(t.leftSuccessor)
setze f_r = FlatOrder(t.rightSuccessor)
setze f = Aneinanderkettung von f_l, t und f_r
gib f zurück

Nachbedingung: Es gilt $\bar{W} = W$ und der Rückgabewert ist $l_{in}(\bar{t})$.

Satz Alle o. g. Ordnungen können in Zeit $\theta(n)$ berechnet werden.

Beweis Analog zum obigen Satz über Breite und Höhe. Lediglich bei der Ebenenordnung ist es etwas komplizierter ...

4.3 Suchbäume (search trees)

Jetzt nehmen wir zusätzlich an, es gäbe eine Funktion $\kappa\colon M \to K$, die jedem Element aus M einen Sortierschlüssel aus einer linear geordneten Menge $K, \leq)$ zuordnet. Die Menge M ist dann linear prägeordnet durch $m \leq m'$ gdw. $\kappa(m) \leq \kappa(m')$. Damit modellieren wir, dass unsere Objekte über Sortierschlüssel verfügen.

Sei t ein beschrifteter Binärbaum über einer linear prägeordneten Menge $(M, \leq)$. Sei $l_{in}(t) = v_0 \dots v_{n-1}$. Dann ist t ein *Suchbaum* genau dann, wenn $\lambda(v_i) < \lambda(v_{i+1})$ für alle i mit $i < n-1$ gilt.

Wir beweisen zuerst eine alternative Charakterisierung:

Lemma [Suchbaum-Charakterisierung] Für jeden Binärbaum t über einer prägeordneten Menge $(M, \leq)$ sind äquivalent:

- t ist ein Suchbaum.
- Für jedes $t' \in N(t) \setminus \{\Box\}$ gilt:
  - $l(v) < l(t')$ für jedes $v \in N(\lambda(t'))$.
  - $l(t') < l(v)$ für jedes $v \in N(\rho(t'))$.

Beweis per Induktion. Für $t = \Box$ ist die Behauptung trivial. Sei also $t =\neq \Box$ ein $v_0 \dots v_{i-1} t v_{i+1} \dots v_{n-1} = l_{in}(t)$ die Auflistung der Knoten von t in Flachordnung. Dann ist $v_0 \dots v_{i-1} = l_{in}(\lambda(t))$ und $v_{i+1} \dots v_{n-1} = l_{in}(\rho(t))$ die Auflistung der Knoten von $\rho(t)$ in Flachordnung.

Angenommen, t ist ein Suchbaum. Dann gilt $l(v_j) < l(v)$ für alle $j < i$ und $l(v) < l(v_j)$ für alle j mit $i < j < n$. Damit gilt die Bedingung für den Teilbaum t von t. Jeder andere Teilbaum ist aber ein Teilbaum von $\lambda(t)$ oder $\rho(t)$. Diese beiden Bäume sind nach Induktionsannahme aber Suchbäume, da $v_0 \dots v_{i-1}$ und $v_{i+1} \dots v_{n-1}$ geordnet sind. Dann gilt aber auch die Bedingung über die Teilbäume von $\lambda(t)$ und $\rho(t)$.

Angenommen, t erfüllt die obige Bedingung über die Teilbäume. Nach Induktionsannahme sind dann aber $\lambda(t)$ und $\rho(t)$ Suchbäume. Daher sind $l(v_0) \dots l(v_{i-1})$ und $l(v_{i+1}) \dots l(v_{n-1})$ geordnet. Andererseits gilt $l(v_j) < l(v)$ für alle $j < m$

und $l(v) < l(v_j)$ für alle j mit $i < j < n$. Das impliziert aber, dass $l(v_0) < \cdots < l(v_{n-1})$ gilt.$\Box$

Aus dem Lemma folgt sofort:

Folgerung [Finden] Sei t ein Suchbaum und $m \in M$. Dann sind äquivalent:

- $m \in \mathrm{lab}(t)$.
- $t \neq \Box$ und entweder
  - $m = l(t)$ oder
  - $m < l(t)$ und $m \in \mathrm{lab}(\lambda(t))$ oder
  - $l(t) < m$ und $m \in \mathrm{lab}(\rho(t))$.

Daraus ergibt sich ein einfacher rekursiver Algorithmus für das Suchen eines Objektes in einem Suchbaum (ausgehend von einem Knoten) anhand eines gegebenen Schlüssels.

FindItem(t, k)

Vorbedingung: t ist ein Suchbaum

```
falls t = null
    gib null zurück
sonst
    falls k = t.item.sortingKey
        gib t zurück
    sonst
        falls k < t.item.sortingKey
            gib FindItem(t.leftSuccessor, k) zurück
        sonst
            gib FindItem(t.rightSuccessor, k) zurück
```

Nachbedingung: Es gilt $W = \bar{W}$ und wenn es $v \in N(\bar{t})$ mit $\bar{\kappa}(\bar{l}(v)) = \bar{k}$ gibt, so ist v der Rückgabewert, sonst ist er null.

Tafelzusatz Beispiel für Suche.

Das Hinzufügen eines Elementes kann auch in einfacher Weise rekursiv durchgeführt werden, indem von der Wurzel aus einen Knoten suchen, an dem das Element angefügt werden kann.

Tafelzusatz Beispiel für Einfügen.

Problem Wenn der Baum ursprünglich leer war, müssen wir nachher wissen, wie der neue Baum heißt.

Lösung Wir geben den Baum zurück.

AddItem(t, o)

Vorbedingung: t ist ein Suchbaum, für den $\kappa(o) \notin \{\kappa(o') \mid o' \in \text{lab}(t)\}$ gilt.

```
falls t = null
    setze c = newContainerWithTwoSuccessors(o, null, null)
    gib c zurück
sonst
    falls o.sortingKey < t.item.sortingKey
        setze t.leftSuccessor = AddItem(t.leftSuccessor, o)
    sonst
        setze t.rightSuccessor = AddItem(t.rightSuccessor,
        o)
    gib t zurück
```

Nachbedingung: Für den Rückgabewert t' gilt $\text{lab}^W(t') = \text{lab}^{\bar{W}}(\bar{t}) \cup \{\bar{o}\}$.

Für den Nachweis der Korrektheit des Algorithmus ist die Nachbedingung nicht stark genug. Um eine genügend starke Nachbedingung formulieren zu können, führen wir zusätzliche Terminologie ein.

Seien $W = (\lambda, \rho, l)$ und $W' = (\lambda', \rho', l')$ gegeben. Wir sagen, *ein Baum t wurde von W zu W' erweitert,* falls es zwei Knoten $u \in \text{lab}^W(t)$ und $u' \notin \text{lab}^W(t)$ gibt, so dass gilt:

1. $\rho(v) = \rho'(v)$ und $\lambda(v) = \lambda'(v)$ für alle $v \notin \{u, u'\}$,
2. $l(v) = l'(v)$ für alle $v \neq u'$,
3. $\rho(u')$, $\lambda(u')$ und $l(u')$ sind undefiniert,
4. $\lambda'(u) = u'$ und $\rho(u) = \text{null}$, oder $\rho'(u) = u'$ und $\lambda(u) = \text{null}$,
5. $\rho'(u') = \lambda'(u') = \text{null}$, $l'(u')$ ist definiert.

Nun können wir die genauere Nachbedingung formulieren: Es ist $\bar{t}$ der leere Baum und der Rückgabewert t' ist ein Suchbaum, für den $\text{lab}^W(t') = \{\bar{o}\}$

gilt, oder der Rückgabewert ist $\bar{t}$ und $\bar{t}$ ist ein Suchbaum, für den $o \in \text{lab}^W(\bar{t})$ gilt, und der von $\bar{W}$ zu W erweitert wurde.

Einen genauen Korrektheitsbeweis sparen wir uns!

Zum Schluss überlegen wir uns noch, wie wir ein Element entfernen können. Wenn das zu entfernende Element keinen oder einen Nachfolger hat, ist dies einfach. Ansonsten:

Tafelzusatz Graphische Veranschaulichung der beiden Varianten des Löschens.

Idee Das minimale Element des rechten Teilbaums nimmt den Platz des zu entfernenden Objektes ein und wird an seiner ursprünglichen Stelle eliminiert (oder symmetrische Variante).

FindMinParent(t)

Vorbedingung: t ist ein nicht-leerer Baum

```
falls t.leftSuccessor = null
    gib null zurück
sonst
    setze v' = t
    setze v = λ(v')
    solange λ(v) ≠ null
        setze v' = v
        setze v = λ(v')
    gib v' zurück
```

Nachbedingung: Es gilt $W = \bar{W}$ und: t besitzt genau einen Knoten und v = null, oder der linke Nachfolger von v existiert und ist der erste in der Flachordnung von t.

DeleteItem(t, k)

Vorbedingung: t ist ein Suchbaum und k ein Schlüssel

```
falls t = null
    gib null zurück
sonst
    falls t.item.sortingKey = k
```

```
            falls t.leftSuccessor = null und t.rightSuccessor =
            null
              gib null zurück
            sonst falls t.leftSuccessor = null
              gib t.rightSuccessor zurück
            sonst falls t.rightSuccessor = null
              gib t.leftSuccessor zurück
            sonst setze v = FindMinParent(t.rightSuccessor)
              falls v = null
                setze t.item = t.rightSuccessor.item
                setze t.rightSuccessor = t.rightSuccessor.rightSuccessor
              sonst
                setze t.item = v.leftSuccessor.item
                setze v.leftSuccessor = v.leftSuccessor.rightSuccessor
        sonst falls k < t.item.sortingKey
            setze t.leftSuccessor = DeleteItem(t.leftSuccessor,k)
        sonst
            setze t.rightSuccessor = DeleteItem(t.rightSuccessor,k)
    gib t zurück
```

Nachbedingung: ...

4.4 Implementierung von Wörterbüchern durch Suchbäume

Frage Wie implementieren wir nun ein Wörterbuch?

Antwort Wir können ein Wörterbuch als SearchTreeDictionary implementieren. Dazu brauchen wir nur die obigen Algorithmen zu verwenden.

SearchTreeDictionary

- ContainerWithTwoSuccessors root
- Methode addItem(o)
  setze root = addItem(root, o)
- ...

4.4.1 Speicherbedarf

Wir unterscheiden zwischen dem Speicherbedarf für die Datenstruktur selbst und für die Ausführung der Operationen:

Satz

1. Der pessimale Speicherbedarf der obigen Wörterbuchimplementierung durch Suchbäume ist $\theta(n)$ (wenn n die Anzahl der zu verwaltenden Objekte bezeichnet.
2. Bei der Ausführung der Wörterbuchoperationen ist der pessimale Speicherbedarf $\theta(h(t))$ (wenn t den Suchbaum vor Ausführung der Operation bezeichnet) und damit auch $\theta(n)$.
3. Betrachtet man iterative Varianten der Suchbaumalgorithmen, kommt man zu einem pessimalen Speicherbedarf von $\theta(1)$.

Beweis Für die Speicherung von n Objekten werden n Container (Knoten) benötigt und eine Referenz auf einen Container (root). Daraus folgt Nummer 1.

Der zusätzliche Speicherbedarf im Rumpf der obigen Algorithmen ist konstant, allerdings sind die Algorithmen rekursiv programmiert, so dass sich im schlimmsten Fall ein Aufwand ergibt, der proportional zur Rekursionstiefe, also der Suchbaumhöhe ist. Daraus ergibt sich Nummer 2.

Bei den iterativen Varianten werden auch nur eine konstante Anzahl von Variablen benötigt, die mit Referenzen auf Container belegt werden. Daraus ergibt sich Nummer 3.

4.4.2 Laufzeit

Offensichtlich ist, dass alle Operationen eine pessimale Laufzeit von $\theta(n)$ haben, wenn n die Anzahl der verwalteten Objekte bezeichnet.

Genauer:

Bemerkung Fügt man nacheinander 1, 2, ..., n in einen Suchbaum ein, so erhält man eine Suchbaum t mit $h(t) = s(t) = n$.

Daraus können wir schließen:

Satz Bei der Implementierung eines Wörterbuchs durch Suchbäume haben alle Operationen eine pessimale Laufzeit von $\theta(h(t))$ und damit auch $\theta(n)$.

Beweis Zunächst zum Beweis der oberen Schranke. Für findItem und addItem gilt offensichtlich, dass ihre pessimale Laufzeit durch die Länge eines längsten Pfades durch den Baum beschränkt ist, denn es erfolgt jeweils ein rekursiver Aufruf an einem der beiden Nachfolger.

Für deleteItem ist festzustellen, dass hier die rekursiven Aufrufe auch entlang eines Pfades wandern. Zusätzlich gibt es aber im letzten Aufruf noch einen Aufruf von FindMinParent. Dieser erfolgt an einem der Nachfolger des gefundenen Knotens und läuft auch wieder entlang eines Pfades. Insgesamt wird also auch ein Aufwand betrieben, der durch die maximale Pfadlänge in dem gegebenen Baum beschränkt ist.□

Nun zur unteren Schranke. Sei t der Suchbaum aus dem obigen Beispiel. Anschließendes Finden und Entfernen von n und Einfügen von $n + 1$ haben die angegebenen Laufzeiten.□

Tafelzusatz Beispiele.

Im Hinblick auf amortisierte Laufzeiten halten wir Folgendes fest.

Bemerkung Fügt man nacheinander $1, 2, \ldots, n$ in einen Suchbaum ein und sucht dann m Mal nach n, so erhält man eine Gesamtlaufzeit von $\theta(n^2+mn)$.

Mit anderen Worten, die amortisierte Laufzeit der Finde-Operation kann nicht besser als n sein!

Andererseits wird man erwarten, dass die entstehenden Bäume in „vielen“ Fällen einigermaßen ausgeglichen sind und von daher die durchschnittliche Tiefe eines Knotens $O(\log n)$ ist.

Die totale Pfadlänge eines Binärbaumes ist definiert durch:

$$p(\square) = 0 \ , \tag{4.12}$$

$$p(t) = s(\lambda(t)) + s(\rho(t)) + p(\lambda(t)) + p(\rho(t)) + 1 \tag{4.13}$$

$$= p(\lambda(t)) + p(\rho(t)) + s(t)) \ . \tag{4.14}$$

Damit ist $p(t)/s(t)$ ein Maß für die durchschnittliche Zeit, die benötigt wird, um ein Element von t zu finden.

Für $n \geq 0$ sei S_n sei die Menge aller Folgen, in denen jedes $i < n$ genau einmal auftritt. Zu einer Folge $\pi = i_0 \ldots i_{r-1}$ über $\mathbf{N}$ sei t_π der Binärbaum, der ausgehend vom leeren Binärbaum dadurch entsteht, dass nacheinander i_0, i_1, ... eingefügt werden. Dann sei die *durchschnittliche totale Pfadlänge* für n gegeben durch

$$a(n) = \frac{1}{|S_n|} \sum_{\pi \in S_n} p(t_\pi) \tag{4.15}$$

$$= \frac{1}{n!} \sum_{\pi \in S_n} p(t_\pi) \ . \tag{4.16}$$

Es gilt:

Satz Die durchschnittliche totale Pfadlänge für n Elemente ist $O(n \log n)$.

Damit ist nach obiger Überlegung die durchschnittliche Zeit, die benötigt wird, eine Element zu finden, $O(\log n)$.

4.5 Spreizbäume (splay trees)

Frage Wie schaffen wir es, eine pessimale Laufzeit von $\theta(\log n)$ für die Wörterbuchoperationen zu erzielen?

Antwort Wir sorgen „einfach“ dafür, dass die entstehenden Suchbäume höhenbalanciert ($h(t) \in O(\log s(t))$) sind. Das ist aber schwierig!

Frage Gibt es ein weniger ambitioniertes, aber dennoch gutes und einfacher zu erreichendes Ziel?

Antwort Amortisierte Laufzeit $\theta(\log n)$! Das heißt, wir erlauben, dass einzelne Operationen im schlechtesten Fall mehr als logarithmische Laufzeit haben. Diese zusätzliche Zeit muss dann aber in vorherigen Operationen schon eingespart worden sein.

Sei T ein ADT mit Operationen $o_0, \ldots, o_{r-1}$ und D eine Implementierung von T. Sei weiterhin zu jedem $i < r$ eine Funktion $f_i \colon \mathbf{N} \to \mathbf{R}_{\geq \mathbf{0}}$ gegeben. Für jede Folge $o_{j_0}, \ldots, o_{j_{m-1}}$ von Operationen gelte die Ungleichung

$$\sum_{i<m} T_i \leq \sum_{i<m} f_{j_i}(n_i) \ , \tag{4.17}$$

in der T_i für die tatsächliche Laufzeit von o_{j_i} und n_i für die Größe der Datenstruktur nach Ausführung von $o_{j_0}, \ldots, o_{j_{i-1}}$, ausgehend von der leeren Datenstruktur, stehen. Dann sagen wir, dass die Laufzeiten der Operationen der Datenstruktur amortisiert durch $f_0, \ldots, f_{r-1}$ beschränkt sind.

Bemerkung Die f_i sind fiktive Laufzeiten, die benutzt werden können, um eine obere Schranke für die tatsächliche Gesamtlaufzeit einer Folger von Operationen benutzt werden können.

Tafelzusatz Graphik.

Beispiel Es gibt eine Konstante c, so dass die Laufzeiten der Operationen einer zirkulären Feldimplementierung eines Kellers oder einer Schlange amortisiert durch c beschränkt sind.

Sprechweise Die amortisierten Laufzeiten der Operationen einer zirkulären Feldimplementierung eines Kellers oder einer Schlange sind $O(1)$.

Wir wollen eine Implementierung eines Wörterbuchs finden, bei dem die Laufzeiten der Operationen amortisiert durch $O(\log n)$ beschränkt sind.

Vorüberlegung Wenn eine Beschriftung tief im Baum hängt und auf sie z. B. durch eine Finde-Operation zugegriffen wird, muss der Baum umstrukturiert werden, denn sonst würden Wiederholungen derselben Operation zu zu langen Laufzeiten führen.

Idee Wird auf eine Beschriftung (z. B. durch eine Finde-Operation) zugegriffen oder wird sie hinzugefügt, so wird der Baum so umstrukturiert, dass die Beschriftung nach der Umordnung an der Wurzel steht. Dies nennen wir *Umkrempeln* bezüglich der jeweiligen Beschriftung.

Das Löschen wird dann entsprechend modifiziert: Zuerst wird bezüglich der zu löschenden Beschriftung umgekrempelt. Danach wird die Wurzel entfernt und der Baum zerfällt in zwei Teilbäume. Daraufhin wird der rechte Teilbaum bezüglich des Minimums umgekrempelt und der linke Teilbaum an die Wurzel des umgekrempelten gehängt.

Tafelzusatz Beispiel für Löschen.

Aufgabe

1. Die richtigen Umstrukturierungen finden!
2. Die amortisierte Laufzeit nachweisen!

Einfachstes Mittel zur Umstrukturierung: Rotationen.

Tafelzusatz Beispiel für Rotationen an einem Knoten u.

Definition Sei t ein Baum in W und $v \in N(t) \setminus \{t\}$ und $u \in N(t)$ derart, dass $\lambda(u) = v$. Dann ist $\mathrm{rot}(t, v)$ definiert durch die folgenden Veränderungen von W zu W':

- $\lambda'(u) = \lambda(v)$, $\rho'(u) = v$,
- $\lambda'(v) = \rho(v)$, $\rho'(v) = \rho(u)$,
- $l'(u) = l(v)$ und $l'(v) = l(u)$.

Symmetrische Definition für den Fall $\rho(u) = v$.

Wir schreiben einfach $\mathrm{rot}(t, v)(W) = W'$.

Lemma Ist t ein Suchbaum und v wie oben, so ist t in $\mathrm{rot}(t, v)$ ein Suchbaum mit $\mathrm{lab}^W(t) = \mathrm{lab}^{W'}(t)$.

Ansatz Jedes Mal, wenn wir auf einen Knoten zugreifen, bringen wir ihn an die Wurzel durch wiederholtes Rotieren. Dies nennen wir *Umkrempeln durch Rotieren.*

Beispiel Ein Beispiel.

Frage Bringt das wirklich eine gute amortisierte Laufzeit?

Antwort Nein!

Beispiel Einfügen von $1, \ldots, n$ und dann Finden in umgekehrter Reihenfolge.

Bemerkung Umkrempeln durch Rotieren hat weder pessimale noch amortisierte Laufzeit von $O(\log n)$.

Lösung Wir ziehen auch noch den Großvater mit in Betracht: Wir rotieren unter gewissen Umständen dreimal anstatt zweimal. Die zugehörige Operation nennen wir Spreizung!

Definition Sei t ein Baum in W. Für Knoten $w \in N(t)$ definieren wir $\mathrm{splay}(t, w)$ unter den folgenden Umständen wie folgt:

(Zick-Zick) Falls es u und v gibt mit $\lambda(u) = v$ und $\lambda(v) = w$, dann ist $\mathrm{splay}(t, w)$ definiert durch die folgenden Veränderungen von W zu W':
- $\lambda'(u) = \lambda(w)$, $\rho'(u) = v$,
- $\lambda'(v) = \rho(w)$, $\rho'(v) = w$,
- $\lambda'(w) = \rho(v)$, $\rho'(w) = \rho(u)$,
- $l'(u) = l(w)$, $l'(w) = l(u)$.

(Zick-Zack) Falls es u und v gibt mit $\lambda(u) = v$ und $\rho(v) = w$, dann ist $\mathrm{splay}(t, w)$ definiert durch die folgenden Veränderungen von W zu W':
- $\lambda'(u) = v$, $\rho'(u) = w$,
- $\lambda'(v) = \lambda(v)$, $\rho'(v) = \lambda(w)$,
- $\lambda'(w) = \rho(w)$, $\rho'(w) = \rho(u)$,
- $l'(u) = l(w)$, $l'(w) = l(u)$.

∘ Die beiden anderen Fälle werden symmetrisch behandelt.

Wir schreiben analog zu oben $\mathrm{splay}(t, w)(W) = W'$.

Tafelzusatz Spreizungen.

Sprechweise Wiederholte Anwendungen von Spreizungen um einen Knoten an die Wurzel zu bringen bezeichnen wir als *Umkrempeln durch Spreizungen.* Eine einfache Anwendung nennen wir *Spreizung.* Beim Umkrempeln muss ggf. zum Schluss ein einfache Rotation durchgeführt werden, abhängig davon, ob der Knoten ungerade oder gerade „Tiefe“ im Baum hat.

Zur Definition von Tiefe:

$$d_t(u) = \begin{cases} 0 & \text{falls } u \notin N(t), \\ 1 & \text{falls } t = u, \\ d_{\lambda(t)}(u) + 1 & \text{falls } u \in N(\lambda(t)), \\ d_{\rho(t)}(u) + 1 & \text{falls } u \in N(\rho(t)). \end{cases} \tag{4.18}$$

Das Umkrempeln durch Spreizen definieren wir als Veränderung von W zu W^* durch $\text{splay}^*(t, u)$, zunächst induktiv für die Fälle, in denen u ungerade Tiefe hat:

- Falls $t = u$, so $W^* = W$.
- Falls $t \neq u$, so sei $w \in N(t)$ der Knoten mit $d_t(w) = 3$ und $u \in N(w)$. Weiterhin sei $W' = \text{splay}^*(w, u)(W)$ und $W^* = \text{splay}(t, w)(W')$.

Die Fälle gerader Tiefe werden wie folgt behandelt: Es sei v derart, dass $d_t(v) = 2$ und $u \in N(v)$. Weiterhin sei $W' = \text{splay}^*(v, u)(W)$ und $W^* = \text{rot}(t, v)(W')$.

Beispiel Altes Beispiel für die Anwendung der Regeln.

Sprechweise Die zugehörige Datenstruktur nennen wir *Spreizbaum.* Ebenso werden die Bäume, die aus einem leeren Baum durch Anwendung der Wörterbuchoperationen mit Umkrempeln entstehen, *Spreizbäume* genannt.

Beobachtung Die Tiefe des Baumes wird ungefähr halbiert durch das Umkrempeln durch Spreizungen am ersten Knoten!

Satz Die Platznutzung von Spreizbäumen beträgt $\theta(n)$, die pessimale Laufzeit aller Spreizbaum-Operationen ist $\theta(n)$ und die zusätzliche Speichernutzung der Spreizbaum-Operationen ist $\theta(1)$ (bei einer iterativen Implementierung).

Wir werden im nächsten Abschnitt aber beweisen, dass die amortisierte Laufzeit $O(\log n)$ ist.

4.6 Amortisierte Analyse von Spreizbäumen

Satz Die amortisierte Laufzeit der Spreizbaum-Operationen ist $O(\log n)$.

Zuerst müssen wir ein genaues Maß für die tatsächlichen Laufzeiten der Spreizbaum-Operationen festlegen. Dabei betrachten wir den Leerheitstest wobei W' für $\text{splay}^*(t,u)(W)$ stehtnicht.

Für das Finden können wir einfach die Tiefe des zu findenden Knotens bzw. des zuletzt besuchten Knotens, an dem das Umkrempeln begonnen wird, ansetzen. Beim Einfügen setzen wir die Tiefe des neuen Knotens an, an dem ja direkt umgekrempelt wird. Beim Löschen setzen wir die Tiefe des Knotens an, der gelöscht wird, zuzüglich der Tiefe des Minimums des rechten Teilbaum, der von der Wurzel getrennt wird.

Ansatz Wir setzen $f(n) = c \operatorname{Log} n$ mit $\operatorname{Log} n = \max\{\log n, 1\}$ für ein geeignetes c und konstruieren eine Funktion Φ, die jedem Suchbaum t in einer Umgebung W eine nicht-negative reelle Zahl $\Phi^W(t)$ zuordnet, so dass $\Phi^W(\square) = 0$ gilt und folgende Bedingung erfüllt ist.

Ist t ein Suchbaum, o eine Wörterbuchoperation, t' das Ergebnis der Anwendung von o auf t, T die Laufzeit dieser Operation und $T^a = f(s(t))$, so gilt

$$T \leq T^a + (\Phi^W(t) - \Phi^{W'}(t')) \ . \tag{4.19}$$

Dann kann $\Phi(t)$ als Zeitguthaben interpretiert werden und die Ungleichung etwa wie folgt: Die tatsächliche Laufzeit ist beschränkt durch die amortisierte Laufzeit plus der vom Konto abgehobenen Zeit.

Sprechweise Die Funktion Φ wird *Potenzialfunktion* genannt. Die obige Ungleichung wird als *Potenzialbeziehung* bezeichnet.

Wir werden die Potenzialbedingung auch häufig in der folgenden Form lesen:

$$\Phi^{W'}(t') - \Phi(t)^W \leq T^a - T \ . \tag{4.20}$$

Es würde reichen, wenn die Potenzialbeziehung gelten würde. Dann hätten wir für eine Folge $o_0, \ldots, o_{m-1}$, die ausgehend von der leeren Datenstruktur

ausgeführt wird und nacheinander die Bäume $t_0, \ldots, t_m$ in den Umgebungen $W_0, \ldots, W_m$ erzeugt:

$$\sum_{i<m} T_i \leq \sum_{i<m} (T_i^a + (\Phi^{W_i}(t_i) - \Phi^{W_{i+1}}(t_{i+1}))) \tag{4.21}$$

$$\leq \sum_{i<m} f(n_j) + (\Phi^{W_0}(\Box) - \Phi^{W_m}(t_m))) \tag{4.22}$$

$$\leq \sum_{i<m} f(n_j) \ . \tag{4.23}$$

Schwierig Finden der Potenzialfunktion.

Motivation Wir sollten ein um so größeres Zeitguthaben haben, je weniger ausgeglichen der Baum ist. Wir werden es mit folgender Funktion versuchen:

$$\Phi(\Box) = 0 \ , \tag{4.24}$$

$$\Phi(t) = \log s(t) + \Phi(\lambda(t)) + \Phi(\rho(t)) \ . \tag{4.25}$$

Schreibweise Für $\log s(t)$ werden wir in Zukunft auch einfach $r(t)$ schreiben und $r(t)$ den Rang von t nennen.

Dann ist der Rang eines Baumes größenordnungsmäßig gleich der amortisierten Laufzeitschranke, deren Gültigkeit wir beweisen möchten. Wir können die Potenzialfunktion anders beschreiben:

Lemma Sei t ein Suchbaum. Dann gilt:

$$\Phi(t) = \sum_{t' \in N(t)} r(t') \ . \tag{4.26}$$

Für den Beweis des obigen Satzes, zeigen wir zuerst, dass die Potenzialbedingung im Wesentlichen gilt, wenn wir nur eine Spreizung durchführen.

Lemma Sei t ein Baum und $u \in N(t)$.

(Rotation) Ist $d_t(u) = 2$ und $W' = \mathrm{rot}(t, u)(W)$, so gilt

$$\Phi^{W'}(t) - \Phi^{W}(t) \leq 3(r^{W'}(t) - r^{W}(u)) \ . \tag{4.27}$$

(Spreizung) Ist $d_t(u) = 3$ und $W' = \text{splay}(t, u)(W)$, so gilt

$$\Phi^{W'}(t) - \Phi^W(t) \leq 3(r^{W'}(t) - r^W(u)) - 2 \ . \tag{4.28}$$

Man beachte, dass 2 die tatsächliche Laufzeit von Spreizungen ist.

Beweis Wir unterscheiden die einzelnen Fälle.

Zick. Zunächst ergibt sich:

$$\Phi^{W'}(t) - \Phi^W(t) = r^{W'}(u) - r^W(u) \ . \tag{4.29}$$

Aus $r^{W'}(t) \geq r^{W'}(u)$ folgt dann

$$\Phi^{W'}(t) - \Phi^W(t) \leq r^{W'}(t) - r^W(u) \ . \tag{4.30}$$

Daraus folgt aber wegen $r^{W'}(t) \geq r^W(u)$ schon die Behauptung.

ZickZick. Wir setzen $v = \lambda^W(t)$. Offensichtlich gilt

$$\Phi^{W'}(t) - \Phi^W(t) = r^{W'}(v) + r^{W'}(u) - r^W(v) - r^W(u) \ . \tag{4.31}$$

Außerdem gilt:

$$s^W(v) + s^{W'}(v) \leq s^{W'}(t) \ . \tag{4.32}$$

Daraus folgt mit $\log a + \log b \leq 2\log(a + b) - 2$:

$$r^W(u) + r^{W'}(u) \leq 2r^{W'}(t) - 2 \ , \tag{4.33}$$

also auch

$$\Phi^{W'}(t) - \Phi^W(t) \leq 2r^{W'}(t) - 2r^W(u) + r^{W'}(v) - r^W(v) - 2 \ . \tag{4.34}$$

Wegen $r^{W'}(v) \leq r^{W'}(t)$ und gleichsam $r^W(v) \geq r^W(u)$ ergibt sich schließlich

$$\Phi^{W'}(u) - \Phi^W(t) \leq 3r^{W'}(t) - 3r^W(u) - 2 = 3(r^{W'}(t) - r^W(u)) - 2 \ . \tag{4.35}$$

Die dritte Ungleichung wird in analoger Weise bewiesen.□

Als Folgerung erhalten wir sofort per Induktion:

Folgerung Ist t ein Baum, $u \in N(t)$ und $W' = \text{splay}^*(t, u)(W)$, so gilt

$$\Phi^{W'}(t) - \Phi^W(t) \leq 3(r^{W'}(t) - r^W(u)) - d_t^W(u) + 1 \ . \tag{4.36}$$

Daraus können wir leicht den folgenden Schluss ziehen.

Folgerung Es gibt eine Konstante c_0, so dass für jedes $c \geq c_0$, jeden Baum t und jedes $u \in N(t)$ die Ungleichung

$$d_t^W(u) \leq c \operatorname{Log}(s^W(t)) + (\Phi^W(t) - \Phi^{W'}(t)) \tag{4.37}$$

mit $W' = \text{splay}^*(t, u)(W)$ gilt.

Beweis Wir haben nach obiger Folgerung:

$$\begin{aligned} d_t^W(u) &\leq 3(r^{W'}(t) - r^W(u)) + 1 + (\Phi^W(t) - \Phi^{W'}(t)) \\ &= 3(\log(s^W(t)) - \log(s^W(u))) + 1 + (\Phi^W(t) - \Phi^{W'}(t)) \\ &\leq 3\log(s^W(t)) + 1 + (\Phi^W(t) - \Phi^{W'}(t)) \\ &\leq 4\operatorname{Log}(s^W(t)) + (\Phi^W(t) - \Phi^{W'}(t)) \ . \end{aligned}$$

Wir brauchen also nur $c_0 = 4$ zu wählen, damit die Behauptung stimmt.□

Nachweis der Potenzialbeziehung. Wir setzen $c = 2c_0 + 2$ und betrachten die einzelnen Operationen separat. Wir lässen die Fälle außer Acht, in denen $t = \square$ oder $t' = \square$ gilt; diese sind besonders leicht zu einzusehen.

Finden. Zu beachten ist zunächst, dass $t = t'$ gilt. Aus obiger Folgerung erhalten wir:

$$\begin{aligned} T &= d_t^W(u) \\ &\leq c_0 \operatorname{Log}(s^W(t)) + (\Phi^W(t) - \Phi^{W'}(t)) \\ &\leq c \operatorname{Log}(s^W(t)) + (\Phi^W(t) - \Phi^{W'}(t)) \\ &= T^a + (\Phi^W(t) - \Phi^{W'}(t)) \\ &= T^a + (\Phi^W(t) - \Phi^{W'}(t')) \ . \end{aligned}$$

Einfügen. Dazu sei W^* die Umgebung, die man durch Einfügen ohne Um-

krempeln erhält und u der eingefügte Knoten. Dann ergibt sich:

$$\begin{aligned}
T &= d_t^{W^*}(u) \\
&\leq c_0 \operatorname{Log}(s^{W^*}(t)) + (\Phi^{W^*}(t) - \Phi^{W'}(t)) \\
&= c_0 \operatorname{Log}(s^{W}(t) + 1) + (\Phi^{W^*}(t) - \Phi^{W'}(t)) \\
&\leq 2c_0 \operatorname{Log}(s^{W}(t)) + (\Phi^{W^*}(t) - \Phi^{W'}(t)) \\
&\leq 2c_0 \operatorname{Log}(s^{W}(t)) + \log(s^{W}(t) + 1) + (\Phi^{W}(t) - \Phi^{W'}(t)) \\
&\leq 2c_0 \operatorname{Log}(s^{W}(t)) + 2\operatorname{Log}(s^{W}(t)) + (\Phi^{W}(t) - \Phi^{W'}(t)) \\
&\leq c \operatorname{Log}(s^{W}(t)) + (\Phi^{W}(t) - \Phi^{W'}(t)) \\
&\leq T^a + (\Phi^{W}(t) - \Phi^{W'}(t)) \\
&= T^a + (\Phi^{W}(t) - \Phi^{W'}(t')) \ .
\end{aligned}$$

Tafelzusatz Graphik für schwierige Abschätzung in der Mitte.

Löschen. Wir betrachten nur den schwierigen Fall, in dem das zu löschende Element nicht das Minimum ist. Es sei W^* die Umgebung nach Umkrempeln an dem zu löschenden Knoten u, außerdem sei $v = \lambda^{W^*}(t)$ und $w = \rho^{W^*}(t)$. Desweiteren sei W^{**} die Umgebung nach Umkrempeln von w am Minimum x und $y = \rho^{W^{**}}(w)$. Schließlich sei W' das Endergebnis mit w als neuer Wurzel, d. h., $w = t'$.

Wir haben dann:

$$\begin{aligned}
T &= d_t^{W}(u) + d_x^{W^*}(w) \\
&\leq c_0 \operatorname{Log} s^{W}(t) + \Phi^{W}(t) - \Phi^{W^*}(t) + c_0 \operatorname{Log} s^{W^*}(w) + \Phi^{W^*}(w) - \Phi^{W^{**}}(w) \\
&\leq 2c_0 \operatorname{Log} s^{W}(t) + \Phi^{W} t - (\Phi^{W^*}(t) + \Phi^{W^{**}}(w) - \Phi^{W^*}(w)) \ .
\end{aligned}$$

Desweiteren gilt aber:

$$\begin{aligned}
\Phi^{W'}(w) &= \Phi^{W'}(v) + \Phi^{W'}(y) + r^{W'}(w) \\
&= \Phi^{W^{**}}(v) + \Phi^{W^{**}}(y) + r^{W'}(w) \\
&= \Phi^{W^{**}}(w) - r^{W^{**}}(w) + \Phi^{W^{**}}(v) + r^{W'}(w) \\
&= \Phi^{W^{**}}(w) + \Phi^{W^*}(t) - \Phi^{W^*}(w) - r^{W^*}(t) - r^{W^{**}}(w) + r^{W'}(w) \\
&\leq \Phi^{W^{**}}(w) + \Phi^{W^*}(t) - \Phi^{W^*}(w) - r^{W}(t) \\
&\leq \Phi^{W^{**}}(w) + \Phi^{W^*}(t) - \Phi^{W^*}(w) - \operatorname{Log} s^{W}(t) \ .
\end{aligned}$$

Daraus und dem obigen ergibt sich

$$\begin{aligned} T &\leq (2c_0 + 1)\,\mathrm{Log}(s^W(t)) + \Phi^W(t) - \Phi^{W'}(w) \\ &\leq c\,\mathrm{Log}(s^W(t)) + (\Phi^W(t) - \Phi^{W'}(t'))\ . \end{aligned}$$

Damit ist auch dieser Fall gelöst.□

Kapitel 5

Grundlegende Graphalgorithmen

5.1 Einführung

Graphen sind ein wichtiges Modellierungsmittel in der Informatik.

Beispiel Klassendiagramme, Netzwerktopologien, Zustandsdiagramme (für Prozesse), Abhängigkeitsgraphen in der Ablaufplanung, ...

Frage Was ist ein Graph?

Antwort Alles, was irgendwie so aussieht, also im Wesentlichen aus Punkten besteht, die durch Linien miteinander verbunden sind.

Tafelzusatz Viele Bildchen von unterschiedlichen Graphen.

Unterscheidungsmerkmale und Charakteristika:

- gerichtete oder ungerichtete Kanten,
- beschriftete oder unbeschriftete Kanten,
- gewichtete oder ungewichtete Kanten,
- Mehrfachkanten erlaubt,
- Schleifen erlaubt,

- unendlich viele Knoten erlaubt,
- ...

Wir werden uns auf einfache Graphklassen beschränken und grundlegende algorithmische Probleme behandeln, wie zum Beispiel die Frage, zu welchen Knoten man von einem gegebenen Knoten aus gelangen kann.

Im Folgenden: Zunächst alle Definitionen für ungerichtete Graphen.

5.2 Ungerichtete Graphen

Ein *ungerichteter Graph* ist ein Paar (V, E) bestehend aus einer Menge V von *Knoten* und einer Menge $E \subseteq \binom{V}{2}$ von *Kanten,* wobei $\binom{M}{2}$ für jede Menge M aus der Menge aller zweielementigen Teilmengen von M besteht.

Tafelzusatz Beispielgraphen.

Also: ohne Schleifen, keine Mehrfachkanten.

5.2.1 Grundlegende Definitionen

Wenn ein ungerichteter Graph mit G bezeichnet wird, so werden die Knoten- und die Kantenmenge von G auch mit V_G und E_G bezeichnet. Die Menge E_G heißt auch *Adjazenzrelation.* Wenn $\{u, v\} \in E$, so heißt v *adjazent* zu u.

Ist G ein ungerichteter Graph und $u \in V_G$, so heißt $deg_G(u) = |\{v \mid \{u, v\} \in E_G\}|$ der *Grad* von u (in G).

Eine nicht leere Folge $P = (u_0, \ldots, u_n)$ von Knoten heißt *Pfad* in einem ungerichteten Graphen G, wenn $\{u_i, u_{i+1}\} \in E_G$ für alle $i < n$ gilt. P ist ein Pfad von u_0 nach u_n und hat die *Länge* n. Er *enthält* die Knoten $u_0, \ldots, u_n$ und die Kanten $\{u_0, u_1\}, \ldots, \{u_{n-1}, u_n\}$. Der Knoten u_0 ist durch P mit u_n *verbunden* und u_n ist von u_0 über P *erreichbar.* Wir schreiben $u_0 \rightsquigarrow_G^P u_n$ oder auch $u_0 \rightsquigarrow_G u_n$ oder auch $u_0 \rightsquigarrow u_n$.

Tafelzusatz Beispiele.

Ein ungerichteter Graph G ist *zusammenhängend* genau dann, wenn $u \rightsquigarrow_G v$ für alle $u, v \in V_G$ gilt.

Lemma Sei G ein ungerichteter Graph. Die Relation $\rightsquigarrow_G$ is reflexiv, symmetrisch und transitiv, also eine Äquivalenzrelation auf den Knoten von G.

Beweis einfach.

Wir schreiben deshalb auch $u \leftrightsquigarrow_G v$ anstelle von $u \rightsquigarrow_G v$.

Bemerkung Sei G ein ungerichteter Graph. Dann sind folgende Aussagen äquivalent:

- G ist zusammenhängend.
- $\leftrightsquigarrow_G$ besteht aus höchstens einer Äquivalenzklasse.

Tafelzusatz Beispiele.

Die Äquivalenzklassen von $\leftrightsquigarrow_G$ heißen *Zusammenhangskomponenten.* Die Äquivalenzklasse eines Knotens u heißt auch Zusammenhangskomponente des Knotens und wird mit $CC_G(u)$ bezeichnet.

Ein Pfad $P = (u_0, \ldots, u_n)$ heißt *einfach,* wenn $u_i \neq u_j$ für alle $i, j \leq n$ mit $i \neq j$ gilt. P heißt *geschlossen,* wenn $u_0 = u_n$ und $n > 0$ gilt. Ein *Teilpfad* von P ist von der Form $(u_i, \ldots, u_j)$ mit $0 \leq i \leq j \leq n$.

Lemma Sei G ein Graph und $u, v \in E_G$, so dass v von u in G über einen Pfad P erreichbar ist. Dann ist v von u auch über einen einfachen Pfad in G erreichbar, der nicht länger als P ist.

Beweis per Induktion über die Länge von $P = (u_0, \ldots, u_n)$. Ist $n = 0$, so ist P einfach. Sei nun $n > 0$. Wenn P einfach ist, ist nichts zu zeigen. Wenn P nicht einfach ist, so gibt es i und j mit $0 \leq i < j \leq n$ und $u_i = u_j$. Falls $j = n$ ist, so verbindet $P' = (u_0, \ldots, u_i)$ den Knoten u mit v und die Behauptung folgt aus der Induktionsannahme. Falls $j < n$ ist, so verbindet $P' = (u_0, \ldots, u_i, u_{j+1}, \ldots, u_n)$ den Knoten u mit v und die Behauptung folgt ebenfalls aus der Induktionsannahme.□

Tafelzusatz Illustration.

Ein Pfad $(u_0, \ldots, u_n)$ in einem ungerichteten Graph ist ein *Zykel,* falls $u_0 = u_n$, $n \geq 3$ und $(u_0, \ldots, u_{n-1})$ einfach ist. Ein ungerichteter Graph ohne Zykel heißt *azyklisch.*

Ein ungerichteter Graph G heißt *Baum,* wenn er azyklisch und zusammenhängend ist. Er heißt *Wald,* wenn er azyklisch ist.

Bemerkung Wenn ein ungerichteter Graph ein Wald ist, dann ist jede Zusammenhangskomponente ein Baum.

Lemma Für einen ungerichteten Graph G sind die folgenden Bedingungen

äquivalent.

- G ist ein Baum.
- Jeweils zwei Knoten von G sind durch genau einen einfachen Pfad verbunden.

Beweis Wir beweisen nur, dass 2 aus 1 folgt. Wenn G ein Baum ist, so ist G zusammenhängend. Also gibt es nach obigem Lemma zu je zwei Knoten einen einfachen Pfad, der beide verbindet. Angenommen, es gäbe zwei Knoten, die durch zwei einfache Pfade verbunden wären. Dann gäbe es zwei einfache Pfade, etwa $P = (u_0, \ldots, u_m)$ und $P' = (v_0, \ldots, v_n)$, mit $u_0 = v_0$ und $u_m = v_n$. Unter allen diesen wählen wir solche, für die $m + n$ minimal ist. Dann gilt $m, n > 0$. Außerdem gilt $u_i \neq v_j$ für alle i und j mit $0 < i < m$ und $0 \leq j \leq n$, da sonst $m + n$ nicht minimal wäre oder P oder P' nicht einfach wäre. Dann ist aber $(u_0, \ldots, u_n, v_{m-1}, \ldots, v_0)$ ein Zykel.□

Ein *verwurzelter Baum* ist ein Baum zusammen mit einem Knoten des Baums, d. h., ein Paar (T, ρ) mit T Baum und $\rho \in V_T$. Der Knoten ρ ist die *Wurzel* des Baumes.

Die schon für Bäume bekannten Begriffe übertragen sich in natürlicher Weise: Zu jedem Knoten $v \in V_T$ sei P_v der eindeutige einfache Pfad von ρ nach v. Es seien $u, v \in V_G$. Der Knoten v ist

- *Nachfahre* von u, wenn $u \neq v$ und P_u ein Teilpfad von P_u ist,
- *Vorfahre* von u, wenn u Nachfahre von v ist,
- *Nachfolger* von u, wenn v ein Nachfahre von u und adjazent zu u ist,
- *Vorgänger* von u, wenn u ein Sohn von v ist,
- ein *Blatt,* wenn er keinen Nachfahren besitzt,
- ein *innerer Knoten,* wenn er kein Blatt ist.

Ein ungerichteter Graph G ist ein *Teilgraph* eines Graphen G', wenn $V_G \subseteq V_{G'}$ und $E_G \subseteq E_{G'}$ gelten. Er heißt *induzierter Teilgraph,* wenn $E_G = E_G \cap \binom{V_{G'}}{2}$. Dann schreiben wir $G[V_{G'}]$ für diesen Graphen.

Für $G[CC_G(u)]$ schreiben wir auch $GCC_G(u)$.

Ab jetzt betrachten wir nur noch endliche Graphen.

5.2.2 Algorithmische Behandlung

Vorbemerkung Wir beschäftigen uns hauptsächlich mit Algorithmen auf Graphen, die aus den Graphen interessante Eigenschaften oder Objekte berechnen, und weniger mit Operationen auf Graphen, die diese verändern und damit im Sinne von Datenstrukturen manipulieren. Deshalb werden wir abgesehen vom Hinzufügen von Kanten zunächst keine Methoden zur Manipulation von Graphen bereit stellen. Außerdem werden wir davon ausgehen, dass die Knotenmenge immer von der Form $\{0, \ldots, n-1\}$ ist.

Abstrakter Datentyp für Graphen:

- numberOfVertices, numberOfEdges, degree(i),
- addEdge(i, j), hasEdge(i, j),
- arrayOfAdjacentVertices(i).

Im Wesentlichen gibt es zwei Möglichkeiten, Graphen zu implementieren:

- Darstellung durch Adjazenzmatrix,
- Darstellung durch Adjazenzlisten.

Adjazenzmatrix Die Menge E wird kodiert durch eine $(n \times n)$-Matrix, die so genannte *Adjazenzmatrix* von V: eine boolesche Matrix, die an der Stelle (i, j) genau dann true enthält, wenn $\{v_i, v_j\} \in E$. Die Matrix wird implementiert durch ein zweidimensionales Feld.

Tafelzusatz Beispiel.

Adjazenzlisten Die Menge E wird kodiert in einem Feld der Länge n. Das Objekt an der Stelle $a[i]$ ist eine einfach verkettete Liste, die die zu i adjazenten Knoten enthält, möglichst in geordneter Reihenfolge.

Tafelzusatz Beispiel.

Für die Laufzeiten der oben beschriebenen Methoden ergibt sich dann:

	Adjazenzmatrix	Adjazenzlisten
hasEdge(u,v)	$\theta(1)$	$\theta(deg_G(u))$
arrayOfAdjacentVertices(u)	$\theta(n)$	$\theta(deg_G(u))$

Da wir in unseren Algorithmen häufig Aufzählungen adjazenter Knoten benötigen werden, werden wir in der Regel von einer Darstellung in Form von Adjazenzlisten ausgehen. Dann benötigt man nämlich für das Durchlaufen aller Adjzenzlisten nur $O(|E_G|)$.

5.3 Entfernungen, kürzeste Pfade und Breitensuche

Sei G ein ungerichteter Graph und $u, v \in V_G$. Die *Entfernung (Abstand)* von u zu v, i. Z., $dist_G(u, v)$, ist unendlich, wenn v von u nicht erreichbar ist. Sonst ist $dist_G(u, v)$ die kleinste Länge eines Pfades von u nach v. Gilt $dist_G(u, v) < \infty$, so heißt jeder Pfad von u nach v der Länge $dist_G(u, v)$ ein *kürzester Pfad* von u nach v.

Wir suchen einen Algorithmus, der zu einem gegebenen Knoten u in einem ungerichteten Graphen G die Entfernungen von u zu allen Knoten von G bestimmt und dazu ggf. passende kürzeste Pfade.

Frage Können wir die kürzesten Pfade gut darstellen?

Antwort Ja, in einem Teilgraphen, der ein Baum ist.

Tafelzusatz Beispiel.

Definition Sei G ein ungerichteter Graph und $u \in V_G$. Ein verwurzelter Baum (T, u) heißt *Entfernungsbaum* für u in G, wenn T ein Teilgraph von G ist, dessen Knotenmenge aus den von u erreichbaren Knoten besteht und für den außerdem gilt: Ist $v \in V_T$ und P der einfache Pfad von u nach v in T, dann ist P ein kürzester Pfad von u nach v in G.

Dass es immer Entfernungsbäume gibt, beweisen wir später.

Frage Wie repräsentiert (implementiert) man (T, u)?

Antwort Man hält in einem Feld zu jedem Knoten fest, wer der Vater ist. Die Wurzel hat den Vater -1; die nicht zum Baum gehörenden Knoten haben den Vater ∞. Heißt das Feld p, so bezeichnen wir den verwurzelten Baum dazu mit T_p.

Idee Die Knoten, die adjazent zu u sind, haben Abstand 1. Die Knoten, die adjazent zu den Knoten sind, die Abstand 1 haben, und selbst nicht Abstand 0 oder 1 haben, haben Abstand 2 ... Die Knoten müssen also abgearbeitet werden in der Reihenfolge, wie wir sie „entdecken“, wenn wir ausgehend von u den Graphen „erkunden“. Konsequenz: Wir benutzen eine Schlange.

BFS(G, u)

Vorbedingung: G ist ungerichteter Graph mit n Knoten und u ist ein Knoten von G

1. *Erzeugung und Initialisierung der Hilfsfelder.*
   setze c = newArray$[n]$, p = newArray$[n]$, d = newArray$[n]$
   für $v = 0$ bis $n - 1$
       setze $c[v]$ = white, $p[v] = \infty$, $d[v] = \infty$
   setze $c[u]$ = gray, $p[u] = -1$, $d[u] = 0$
2. *Erzeuge und initialisiere Schlange.*
   setze q = newQueue()
   q.enqueue(u)
3. *Hauptschleife.*
   solange nicht q.isEmpty()
   (a) *Bestimme nächsten Knoten.*
       setze $v = q$.dequeue()
   (b) *Bestimme und bearbeite alle Nachbarn.*
       setze $a = G$.arrayOfAdjacentVertices(v)
       für $i = 0$ bis a.length $- 1$
           setze w = a[i]
           Bearbeite Knoten falls weiß.
           falls $c[w]$ = white
              setze $c[w]$ = gray — *neue Farbe*
              setze $p[w] = v$ — *Vorfahre im Baum*
              setze $d[w] = d[v] + 1$ — *Abstand*
              q.enqueue(w) — *Einreihen*
   (c) *Färbe v schwarz.*
       setze $c[v]$ = black

Nachbedingung: d ist ein Feld, das die Entfernungen von u zu allen Knoten in G enthält, und p kodiert einen Entfernungsbaum von u in G

weiß	unentdeckt
grau	entdeckt
schwarz	bearbeitet

Tafelzusatz Beispiel nach CLR.

Bemerkung Das Farbhilfsfeld ist eigentlich nicht notwendig.

Satz Sei G ein endlicher ungerichteter Graph und u ein Knoten von G.

1. Nach Beendigung von BFS(u) gilt:
   (a) $d[v] = dist_G(u, v)$ für jeden Knoten v von G.
   (b) Der durch p definierte Baum ist ein Entfernungsbaum für u in G.
2. Die Laufzeit von BFS(u) ist $\theta(m_u + n)$, wenn $n = |V_G|$ und m_u die Anzahl der Kanten von $GCC_G(u)$ ist.

Beweis Zuerst zum 2. Teil. Zeitaufwand:

- Initialisierung: $\theta(n)$,
- Anzahl der Wiederholungen der solange-Schleife: $\theta(n_u)$, wenn $n_u = |CC_G(u)|$,
- Anzahl der Wiederholungen der für-Schleife: $\theta(m_u)$.

Da damit alle Schleifen erfasst sind, erhalten wir insgesamt $\theta(n+m_u+n_u) = \theta(n + m_u)$, da $n_u \leq m_u + 1$.

Den zweiten Teil beweisen wir durch Angabe einer geeigneten Schleifeninvariante. Zusätzliche Notation:

$$\text{reach}_G(v) = \{w \mid v \rightsquigarrow_G w\} \ , \tag{5.1}$$

$$\text{reach}_G(U) = \bigcup_{v \in U} \text{reach}_G(v) \ , \tag{5.2}$$

für $v, w \in V_G$ und $U \subseteq V_G$. Mit V_w, V_g und V_b bezeichnen wir die Menge der Knoten, die weiß, grau bzw. schwarz gefärbt sind, mit $[q]$ die Menge der Elemente in der Schlange q.

Die Invariante besteht aus den folgenden Bedingungen:

1. V_G ist disjunkte Vereinigung von V_w, V_g und V_b.
2. $V_g = [q]$.
3. $\text{reach}_G(u) = V_b \cup \text{reach}_G(V_g)$.
4. $d[v] = dist_G(u, v)$ für alle $v \in V_b \cup V_g$.
5. $d[v] = \infty$ für alle $v \in V_w$.
6. T_p ist ein Entfernungsbaum für u in $G[V_b \cup V_g]$.
7. $[q]$ enthalte $v_0, \ldots, v_{r-1}$ (mit v_{r-1} am ältesten). Dann gilt:
   (a) $d[v_i] \geq d[v_j]$ für alle i, j mit $0 \leq i \leq j < r$.
   (b) $d[v_0] \leq d[v_{r-1}] + 1$.
   (c) $\{v \mid dist_G(u, v) \leq d[v_{r-1}]\} \subseteq V_b \cup V_g$.
   (d) Jedes v mit $dist_G(u, v) = d[v_{r-1}] + 1$, für das es kein $w \in V_g$ mit $dist_G(u, w) = d[v_{r-1}]$ und $\{w, v\} \in E$ gibt, gehört zu V_g.

Folgerung Zu jedem ungerichteten Graphen G und Knoten $u \in V_G$ gibt es einen Entfernungsbaum.

5.4 Gerichtete Graphen

Ein *gerichteter Graph* ist ein Paar (V, E) bestehend aus einer Menge V von *Knoten* und einer Menge $E \subseteq V \times V$ von *Kanten.*

Fast alle Definitionen, die wir für ungerichtete Graphen getroffen haben, übertragen sich auf gerichtete Graphen in natürlicher Weise. Besonderheiten treten an folgenden Stellen auf.

Ist G gerichtet, so heißt $deg_G^{in}(u) = |\{v \mid (v,u) \in E_G\}|$ der *Eingangs-* und $deg_G^{out}(u) = |\{v \mid (u,v) \in E_G\}|$ der *Ausgangsgrad* von u (in G).

Ein Pfad $P = (u_0, \ldots, u_n)$ in einem gerichteten Graph heißt *Zykel,* wenn $u_0 = u_n$ gilt. Ein *einfacher* Zykel ist ein geschlossener Pfad, bei dem $(u_0, \ldots, u_{n-1})$ einfach ist.

Ein gerichteter Graph ist *stark zusammenhängend,* wenn jeder Knoten von jedem Knoten erreichbar ist. Entsprechend reden wir von *starken Zusammenhangskomponenten,* i. Z. $SCC_G(u)$ und $GSCC_G(u)$.

Ein gerichteter azyklischer Graph heißt auch *DAG* (directed acyclic graph).

Ein gerichteter Graph G heißt *Baum,* wenn es einen verwurzelten Baum (T, ρ) gibt, so dass $V_G = V_T$ und $E_G = \{(u,v) \mid v \text{ ist Sohn von } u \text{ in } (T,\rho)\}$.

Die Lemmas, die wir bewiesen haben, sind auch übertragbar, insbesondere das Lemma über einfache Pfade. Auch die BFS kann in gerichteten Graphen benutzt werden und berechnet dort auch Entfernungen und Entfernungsbäume.

Die algorithmische Modellierung von gerichteten Graphen erfolgt analog zu der Modellierung von ungerichteten Graphen: der ADT unterscheidet sich lediglich beim Grad eines Knotens und kann, wie gehabt, durch Adjazenzlisten oder -matrizen implementiert werden.

Wir werden uns mit dem Problem beschäftigen, alle starken Zusammenhangskomponenten eines Graphens effizient zu berechnen und in geeigneter Weise aufzuzählen.

5.5 Tiefensuche (depth-first search), topologische Sortierungen und starke Zusammenhangskomponenten (strongly connected components)

Ziel Berechnung der starken Zusammenhangskomponenten eines Graphen!

Ansatz Wir führen eine Ordnung auf den starken Zusammenhangskomponenten ein und schreiben $U \leq V$ für starke Zusammenhangskomponenten U und V, wenn es $u \in U$ und $v \in V$ mit $u \rightsquigarrow v$ gibt. Wenn wir einen Knoten u in einer maximalen starken Zusammenhangskomponente bestimmen könnten, dann könnten wir die Elemente von $SCC_G(u)$ einfach dadurch bestimmen, dass wir alle von u aus erreichbaren Knoten bestimmen, z. B. durch eine Breitensuche. Nachdem wir dann $SCC_G(u)$ bestimmt hätten, könnten wir weiter fortfahren.

Problem Es ist nicht einfach, einen Knoten zu finden, der in einer maximalen Zusammenhangskomponente liegt.

Alternative Wenn wir einen Knoten u in einer minimalen starken Zusammenhangskomponente bestimmen könnten, dann könnten wir die Elemente von $SCC_G(u)$ einfach dadurch bestimmen, dass wir alle Knoten, von denen u aus erreichbar ist, bestimmen.

Zusätzliches Problem Wie können wir zu einem gegebenen Knoten u die Knoten bestimmen, von denen aus u erreichbar ist.

Lösung Dadurch, dass wir im *transponierten Graphen* $G^T = (V_G, \{(u, v) \mid (v, u) \in E_G\})$ die von u erreichbaren Knoten bestimmen.

Frage Wie bestimmen wir denn nun einen Knoten, der in einer minimalen Zusammenhangskomponente liegt?

Antwort Durch Tiefensuche! Der Algorithmus erkundet einen Graphen Knoten für Knoten, aber nicht zuerst alle Nachbarn eines Ausgangsknotens und dann deren Nachbarn ..., sondern zuerst nur einen Nachbarn des Ausgangsknotens und dann nur einen von dessen Nachbarn usw. Wenn er zu einem Knoten kommt, dessen Nachbarn er schon entdeckt hat, dann geht

er zu dem davor entdeckten Knoten zurück und betrachtet einen weiteren seiner Nachbarn ... Wir erhalten ein rekursives Verfahren. Hat man alle von einem Knoten aus erreichbaren Knoten erkundet, geht man zu einem noch nicht besuchten Knoten über und ruft die Tiefensuche wieder auf. Dann ist der Knoten, den man als letzten verlässt, d. h., der Knoten für den man die Tiefensuche zuletzt neu aufruft, ein Knoten einer minimalen Zusammenhangskomponente.

Es gilt sogar noch mehr als wir oben behauptet haben: Wenn man die Knoten in der umgekehrten Reihenfolge, in der sie verlassen werden, aufschreibt, erhält man eine schwache topologische Sortierung.

Definition Sei G ein gerichteter Graph. Eine Folge $(v_0, \ldots, v_n)$ aller Knoten ohne Wiederholungen heißt *schwache topologische Sortierung* von G, wenn für alle i, j mit $0 \leq i < j < n$ gilt: Falls $v_j \rightsquigarrow_G v_i$, so gibt es $k \leq i$ mit $v_k \leftrightsquigarrow v_j$.

Tafelzusatz Beispiel für Tiefensuche.

DFS(G)

Vorbedingung: G ist ein gerichteter Graph mit n Knoten

1. *Erzeugung und Initialisierung der Hilfsfelder.*
   setze c = newArray$[n]$
   für $u = 0$ bis $n - 1$
       $c[u]$ = white
   setze s = newArray$[n]$
   setze $i = n - 1$
2. *Durchlauf aller Knoten.*
   für $u = 0$ bis $n - 1$
       falls $c[u]$ = white
           setze i = DFS-visit(G, u, c, i, s)
   gib s zurück

Nachbedingung: $s[0..n-1]$ ist eine schwache topologische Sortierung von G

DFS-visit(G, u, c, i, s)

Vorbedingung: siehe unten.

1. *Markiere Knoten als entdeckt.*

setze $c[u]$ = gray

2. *Durchlauf aller Nachbarknoten.*
   setze $a = G$.arrayOfAdjacentVertices(u)
   für $j = 0$ bis a.length -1
   (a) *Bestimme Nachbarn.*
   setze $v = a[j]$
   (b) *Behandle Nachbarn, sofern noch unentdeckt.*
   falls $c[v]$ = white
   setze i = DFS-visit(G, v, c, i, s)
   (c) *Vollende Bearbeitung des aktuellen Knotens.*
   setze $s[i] = u$ — *Einfügen in die Ordnung.*
   setze $c[u]$ = black — *abgearbeitet!*
   gib $i - 1$ zurück

Nachbedingung: siehe unten.

Satz Sei G ein gerichteter endlicher Graph.

1. Nach Beendigung von DFS ist $sort[0..n-1]$ eine schwache topologische Sortierung von G.
2. Die Laufzeit von DFS ist $\theta(|V_G| + |E_G|)$.

Beweis Zum Beweis des ersten Teils. Zusätzliche Notation: $\text{reach}_G^W(v)$ = Menge der von v über Knoten aus W erreichbaren Knoten; analog $\text{reach}_G^W(U)$.

Wir geben geeignete Vor- und Nachbedingungen für DFS-visit an.

Vorbedingung, wobei $W = \text{reach}_G^{V_w}(u)$:

- $|W| \leq i - 1$,
- $c[u]$ = white.

Nachbedingung:

- $c[v] = \bar{c}[v]$ für alle $v \notin W$,
- $c[v]$ = black für alle $v \in W$,
- $s[j] = \bar{s}[j]$ für alle j mit $i < j < n$,
- $s[i+1..\bar{i}]$ ist eine schwache topologische Sortierung von $G[W]$.

In der letzten Bedingung bezeichne i den Wert, den DFS-visit zurück gibt.

Wir beweisen dies durch Induktion und zwar per Induktion über die maximale Rekursionstiefe.

Induktionsanfang. Wenn innerhalb von DFS-visit kein weiterer Aufruf von DFS-visit erfolgt, dann sind alle Nachbarn von u nicht weiß, womit die Behauptung trivialerweise gilt.

Induktionsschritt. Wir geben eine Invariante für die für-Schleife an, wobei wir $U = V_w$ (nach Ausführung von $c[u] = gray$), $V_j = \text{reach}_G^U(\{a[0], \ldots, a[j-1]\})$ und $r = a.length$ setzen:

- $c[v] = \bar{c}[v]$ für alle Knoten $v \notin V_j$,
- $c[v] =$ black für alle Knoten $v \in V_j$,
- $s[i+1..\bar{i}]$ ist eine schwache topologische Sortierung von $G[V_j]$,

Wir zeigen, dass daraus der vierte Teil der obigen Nachbedingung folgt. Es ist leicht einzusehen, dass die restlichen drei Bedingungen gelten.

Es sei $G' = G[V_r]$, $G'' = G[W]$, $i < j < k \leq \bar{i}$ und $s[k] \rightsquigarrow_{G''} s[v]$. Beachte, dass W die disjunkte Vereinigung von V_r und $\{u\}$ ist und dass $s[i+1] = u$ gilt.

1. Fall, $s[k] \rightsquigarrow_{G'} s[j]$. Dann folgt aus der Schleifeninvariante $s[j] \rightsquigarrow_{G'} s[k]$, also auch $s[j] \rightsquigarrow_{G''} s[k]$, oder es existiert l mit $i+1 < l < j < k$ und $s[l] \leftrightsquigarrow_{G'} s[k]$, also auch $s[l] \leftrightsquigarrow_{G''} s[k]$.

2. Fall, sonst. Dann gilt $s[k] \rightsquigarrow_{G''} s[i+1] = u \rightsquigarrow_{G''} s[j]$. Andererseits gilt natürlich $u \rightsquigarrow_{G''} s[k]$. Damit ist die Behauptung bewiesen.

Die Gültigkeit der obigen Invariante beweist man in ähnlicher Weise, wobei man $c[v] =$ white und $c[v] \neq$ white unterscheidet.

Dass aus Vor- und Nachbedingung für DFS-visit die Korrektheit für DFS folgt, ergibt sich wie folgt. Man betrachte einfach einen neuen Graphen H der einen zusätzlichen Knoten u hat, von dem aus es zu jedem Knoten des alten Graphen eine Kante gibt. Dann verhält sich DFS auf G wie DFS-visit(u) auf dem neuen Graphen.

Der zweite Teil ist leicht zu zeigen, analog zu BFS.□

Wir können nun ein Lemma formulieren, dass uns sagt, wie man starke Zu-

sammenhangskomponenten effizient berechnet. Gleichzeitig können wir sogar noch stärkere Arten von topologischer Sortierungen bestimmen.

Definition Sei G ein gerichteter Graph. Eine Folge $(v_0, \ldots, v_n)$ aller Knoten ohne Wiederholungen heißt *topologische Sortierung* von G, wenn für alle i, j mit $0 \leq i < j < n$ gilt: Falls $v_j \rightsquigarrow_G v_i$, so $v_i \rightsquigarrow_G v_j$ (also $v_i \leftrightsquigarrow v_j$). Sie heißt *starke topologische Sortierung,* wenn $v_j \rightsquigarrow v_i$ für $i < j$ sogar $v_i \leftrightsquigarrow v_{i+1} \leftrightsquigarrow \ldots \leftrightsquigarrow v_j$ impliziert.

Das heißt, in einer starken topologischen Sortierung werden die Knoten von starken Zusammenhangskomponenten gemäß der Graphstruktur aufgezählt.

Lemma Sei G ein gerichteter endlicher Graph und u_0 der erste Knoten einer schwachen topologischen Sortierung $(u_0, \ldots, u_{r-1})$ von G, $U = \text{reach}_{G^T}(u_0)$ und $W = V_G \setminus U$.

1. $SCC_G(u_0) = U$.
2. Entfernt man aus $(u_0, \ldots, u_{r-1})$ die Elemente von U, so erhält man eine schwache topologische Sortierung von $G[W]$.
3. Die starken Zusammenhangskomponenten von G sind die starken Zusammenhangskomponenten von $G[V_G \setminus U]$ erweitert um U.

Beweis *1. Teil.* Sei $S = SCC_G(u)$.

$S \subseteq U$. Es gilt $v \rightsquigarrow_G u_0$ für jeden Knoten $v \in S$, also $u_0 \rightsquigarrow_{G^T} v$, d. h., $S \subseteq U$.

$U \subseteq S$. Sei $v \in U$ beliebig, $v \neq u$. Dann gilt $v \rightsquigarrow_G u_0$ und es gibt $i > 0$ mit $v = u_i$. Wegen $u_i \rightsquigarrow_G u_0$ gilt dann auch $u_0 \rightsquigarrow_G u_i$, also $u_0 \leftrightsquigarrow_G u_i$, d. h., $v \in S$.

2. Teil. Sei $H = G[W]$ und sei $i_0 < i_1 < \cdots < i_{s-1}$ derart, dass $W = \{u_{i_0}, \ldots, u_{i_{s-1}}\}$, insbesondere $i_0 > 0$. Angenommen, $u_{i_k} \rightsquigarrow_H u_{i_j}$ für $j < k$. Dann gilt $u_{i_k} \rightsquigarrow_G u_{i_j}$, also existiert $l \leq i_j$ mit $u_l \leftrightsquigarrow_G u_{i_k}$. Falls $u_l \in U$ gälte, so hätten wir $u_{i_k} \in \text{reach}_G(u_0)$, also $u_{i_k} \in U$ – ein Widerspruch. Also $l = i_m$ für ein $m \leq j$.

3. Teil. Ähnliche Argumentation.□

Insgesamt ergibt sich damit folgender einfacher Algorithmus für die Berechnung einer starken topologischen Sortierung und damit auch der starken Zusammenhangskomponenten eines gerichteten Graphen.

StrongTopologicalSort(G)

Vorbedingung: G ist ein gerichteter Graph mit n Knoten.

1. *Bestimme eine schwache topologische Sortierung.*
   setze $s = \mathrm{DFS}(G)$
2. *Bestimme transponierten Graphen mit umgeordneten Knoten.*
   setze $H = G.\mathrm{transpose}(s)$
3. *Führe erneute Tiefensuche, aber auf neuem Graphen durch.*
   setze $s2 = \mathrm{DFS}(H)$

Nachbedingung: $s2[0..n-1]$ ist eine starke topologische Sortierung von G.

Dabei soll $G.\mathrm{transpose}(s)$ einen Graphen erzeugen, in dem die Kanten von G umgedreht sind und die Knoten in der Reihenfolge des Feldes s aufgezählt werden.

Frage Wie transponieren wir einen Graphen schnell?

Antwort Das geht einfach, sowohl bei Repräsentation durch Adjazenzmatrix ($\theta(|V_G|^2)$) wie auch durch Adjazenzlisten ($\theta(|V_G| + |E_G|)$).

Tafelzusatz Bildchen zur Veranschaulichung.

Satz Der Algorithmus Strong Topological Sort berechnet eine starke topologische Sortierung eines gerichteten endlichen Graphen G in Zeit $\theta(|V_G| + |E_G|)$.

Bemerkung In DAG fallen alle drei Arten von topologischen Sortierungen zusammen.

Weitere Motivation für topologische Sortierungen: Angenommen, die Knoten eines gerichteten Graphen stehen für durchzuführende Arbeiten und eine Kante von u nach v besagt, dass u ausgeführt werden muss, bevor v ausgeführt werden kann. Dann bringt eine topologische Sortierung die Arbeiten in eine Reihenfolge, die tatsächlich durchgeführt werden kann, sofern der Graph azyklisch ist.

5.6 Gewichtete Graphen und Dijkstra's Algorithmus

In diesem Paragraphen betrachten wir eine Verallgemeinerung der Fragestellung aus dem letzten Paragraphen. Wir wollen in einem gewichteten Graphen einen gewichteten Entfernungsbaum berechnen.

Ein *gewichteter ungerichteter Graph* ist ein Tripel (V, E, ν), bei dem (V, E) ein ungerichteter Graph und $\nu\colon E \to \mathbf{R}_{\geq \mathbf{0}}$ eine *Gewichtsfunktion* ist, die jeder Kante ein nicht negatives *Gewicht* zuordnet.

Ein Pfad $P = (u_0, \ldots, u_n)$ durch einen gewichteten Graphen hat die *gewichtete Länge* $||P|| = \sum_{i<n} \nu(\{u_i, u_{i+1}\})$. Die *gewichtete Abstandsfunktion* ist definiert durch $dist_G(u, v) = \inf\{||P|| \mid u \rightsquigarrow_G^P v\}$.

Tafelzusatz Ein normales Beispiel und ein Beispiel, wo es kein Minimum sondern nur ein Infimum gibt.

Ein Pfad P mit $u \rightsquigarrow_G^P v$ mit $||P|| = dist_G(u, v)$ heißt kürzester Pfad von u zu v.

Bemerkung

1. Wenn es einen kürzesten Pfad von u zu v, so auch einen einfachen kürzesten Pfad.
2. In einem endlichen gewichteten ungerichteten Graphen gibt es zwischen zwei Knoten immer einen kürzesten Pfad.

Definition Ein *Entfernungsbaum* zu einem Knoten u in einem gewichteten Graphen G ist ein verwurzelter Baum (T, u), bei dem T ein Teilgraph von (V_G, E_G) ist und der zu jedem von u erreichbaren Knoten einen kürzesten einfachen Pfad von u zu diesem Knoten enthält.

Frage Wie berechnet man gewichtete Abstände und Entfernungsbäume (falls es sie gibt)?

Antwort Durch trickreiche Abwandlung der Breitensuche.

Ansatz Offensichtlich hat u den gewichteten Abstand 0 von u. Seien $v_0, \ldots, v_{r-1}$

die zu u adjazenten Knoten. Wir setzen $a = \min_{i<r}(\nu(u, v_i))$ und $A = \{v_i \mid \nu(u, v_i) = a\}$. Wir wissen außerdem, dass für jeden beliebigen Pfad $P = (w_0, w_1, w_2, \dots)$ der Länge ≥ 1 mit $w_0 = u$ gilt: $||P|| = \nu(w_0, w_1) + \cdots \geq a$. Damit erhalten wir für jedes $v \in A$: $dist_G(u, v) = a$. Dadurch finden wir also für einige Knoten die Entfernung zu u.

Verallgemeinerung Sei G ein ungerichteter Graph und $u \in V_G$. Zu jeder Menge $U \subseteq V_G$ definieren wir den *Rand* von U in G durch

$$border_G(U) = \{w \in V_G \setminus U \mid \exists(v \in U \wedge \{v, w\} \in E_G)\} \ . \tag{5.3}$$

Sei $U \subseteq V_G$ und $W = border_G(U)$. Die *Randapproximation* d zu u und U ist die Funktion $W \to \mathbf{R}_{\geq \mathbf{0}}$, die wie folgt definiert ist. Für jedes $w \in W$ ist $d(w) = \min\{dist_G(u, v) + \nu(\{v, w\}) \mid v \in U, \{v, w\} \in E\}$.

Lemma Sei G ein ungerichteter gewichteter Graph, $u \in V_G$, $U \subseteq V$ eine Knotenmenge mit $u \in U$ und d die Randapproximation zu u und U. Sei $w \in border_G(U)$ mit $d(w) \leq d(w')$ für alle $w' \in border_G(U)$.

Dann gilt $d(w) = dist_G(u, w)$.

Beweis Wir setzen $W = border_G(U)$. Nach Definition der Randapproximation gilt $d(w') \geq dist_G(u, w')$ für alle $w' \in W$. Wir brauchen also nur noch $d(w) \leq dist_G(u, w)$ zu zeigen.

Wir betrachten einen kürzesten gewichteten Pfad $P = (u_0, \dots, u_n)$ von u zu w. Dann gilt $||(u_0, \dots, u_i)|| = dist_G(u, u_i)$ für alle $i \leq n$. Außerdem gilt $dist_G(u, u_i) \leq dist_G(u, u_{i+1})$ für alle $i < n$.

Sei $i \leq n$ maximal mit $u_i \in U$. Dann gilt $i < n$ und $u_{i+1} \in W$. Also:

$$\begin{aligned} dist_G(u, w) &= ||P|| \\ &\geq dist_G(u, u_{i+1}) \\ &= dist_G(u, u_i) + \nu(\{u_i, u_{i+1}\}) \\ &\geq d(u_{i+1}) \\ &\geq d(w) \ . \end{aligned}$$

□

Tafelzusatz Graphik zur Veranschaulichung.

Konsequenz Wir vergrößern eine Menge $U \subseteq V_G$ schrittweise, so dass sie nur Knoten enthält, für die wir die exakte gewichtete Entfernung von u kennen. Wir nutzen obiges Lemma und berechnen damit immer die geeignete Randapproximation.

Dazu: Ist d eine Randapproximation zu U und sind w und W wie oben, dann ist die Randapproximation d' zu $U' = U \cup \{w\}$ wie folgt gegeben.

1. Fall, $w' \in \mathit{border}_G(U') \setminus \mathit{border}_G(U)$ *mit* $\{w, w'\} \in E_G$. Dann ist $d(w) + \nu(\{w, w'\})$.

2. Fall, $w' \in \mathit{border}_G(U') \cap \mathit{border}_G(U)$ *mit* $\{w, w'\} \in E_G$. Dann ist $d'(w') = \min(d(w'), d(w) + \nu(\{w, w'\}))$.

Sonst gilt $d'(w') = d(w)$.

Damit ist der Algorithmus im Prinzip klar.

Tafelzusatz Beispielrechnung.

Frage Wie verwalten wir die Ränder und Randapproximationen?

Antwort In einer geeigneten Datenstruktur!

Frage Wie sollte die Datenstruktur beschaffen sein?

Antwort Sie sollte eine partielle Abbildung $\{0, \ldots, n-1\} \to \mathbf{R}_{\geq \mathbf{0}}$ verwalten. Operationen: Feststellen auf leeren Definitionsbereich, Auslesen eines Wertes, Hinzufügen eines neues (Argument, Wert)-Paares bzw. Abändern eines Wertes zu einem kleineren Wert, Bestimmen einer Stelle mit minimalem Wert, Löschen eines (Argument, Wert)-Paares mit minimalem Wert.

Eine *aktualisierbare Prioritätsschlange* verwaltet partielle Abbildungen $\{0, \ldots, n-1\} \to \mathbf{R}_{\geq \mathbf{0}}$ und unterstützt die folgenden Operationen:

- isEmpty(): Gibt an, ob die Prioritätsschlange leer ist.
- getValue(i): Gibt den zu i gehörenden Wert der aktuell verwalteten partiellen Abbildung zurück, wenn diese an der Stelle i definiert ist und ∞ sonst.
- update(i, c): Setzt den zu i gehörenden Wert der aktuell verwalteten partiellen Abbildung auf den Wert c, wenn bisher für i kein Wert de-

finiert war. Ansonsten wird der zu i gehörige Wert auf das Minimum von c und dem aktuell zu i gehörenden Wert gesetzt.

- getMin(): Gibt den minimalen Funktionswert der aktuell verwalteten partiellen Abbildung zurück, wenn die Funktion an mindestens einer Stelle definiert ist. Es wird ein Fehler ausgegeben, wenn die Funktion vollständig undefiniert ist.
- extractMin(): Gibt einen Wert aus $\{0, \ldots, n-1\}$ zurück, an dem die aktuell verwaltete partielle Abbildung den minimalen Funktionswert annimmt, sofern die Funktion an mindestens einer Stelle definiert ist. Es wird ein Fehler ausgegeben, wenn die Funktion vollständig undefiniert ist.

Tafelzusatz Beispiel für den Datentyp.

Dann lässt sich der Gesamtalgorithmus wie folgt schreiben:

Dijkstra(G, u)

Vorbedingung: G ist ein gewichteter endlicher Graph mit n Knoten und u ein Knoten von G.

1. *Erzeuge und initialisiere Hilfs- und Ergebnisfelder.*
   setze c = newArray$[n]$, p = newArray$[n]$, ds = newArray$[n]$
   für $v = 0$ bis $n - 1$
   setze $c[v]$ = white, $p[v] = \infty$, $ds[v] = \infty$
   setze $c[u]$ = gray, $p[u] = -1$
2. *Erzeuge und initialisiere Schlange.*
   setze q = newUpdatablePriorityQueue(n)
   q.update$(u, 0)$
3. *Erweitere Entfernungsbaum schrittweise bis Rand lerr ist.*
   solange nicht q.isEmpty()
   (a) *Bestimme nächsten Knoten für den Entfernungsbaum und lege dessen Entfernung fest.*
   setze $d = q$.getMin()
   setze $v = q$.extractMin()
   setze $ds[v] = d$
   setze $c[v]$ = black
   (b) *Passe Rand und Randapproximation an.*
   setze a = arrayOfAdjacentVertices(v)
   setze da = arrayOfIncidentEdgeWeights(v)
   für $i = 0$ bis a.length $- 1$

setze $w = a[i]$ — *Bestimme Nachbarn.*
setze $dw = d + da[i]$ — *neue Approx.*
Aktualisiere Eintrag ggf.
falls $c[w]$ = white oder
($c[w]$ = gray und q.getValue(w) > dw)
setze $c[w]$ = gray
setze $p[w] = v$
q.update(w, dw)

Nachbedingung: ds ist ein Feld, dass die gewichteten Entfernungen von u zu allen Knoten enthält und p kodiert einen gewichteten Entfernungsbaum von u in G.

Laufzeitanalyse

- $O(n + t_i)$ für die Initialisierung, wenn t_i für die Laufzeit von von newUpdatablePriorityQueue(n) steht,
- $O(nt_m)$ für die die äußeren Schleife ohne die innere Schleife, wenn t_m für die Laufzeit von getMin, removeMin und getValue steht,
- $O(mt_u)$ für die Anzahl der Durchläufe der inneren Schleife, wenn t_u für die Laufzeit von update und getValue steht.

Insgesamt: $O(n + t_i + nt_m + mt_u)$. Dazu reicht es natürlich, wenn t_i, t_m und t_u amortisierte Laufzeiten sind.

Unterschiedliche Implementierungen liefern unterschiedliche Laufzeiten.

Alternativen

- Feld: $a[i]$ enthält den Wert von Knoten i; wenn Knoten nicht zum Rand gehört, wird -1 eingetragen.
- Liste: In der Liste stehen alle Knoten des Randes mit zugehörigen Werten.
- Spreizbaum (+ Feld): Der Baum enthält für jeden Knoten des Randes einen Eintrag mit dem zugehörigen Wert als Schlüssel. Außerdem gibt es ein Feld wie bei der ersten Alternative.
- Fibonacci-Halde: sehr clever!

	t_i	t_m	t_u	Dijkstra
Feld	$\theta(n)$	$\theta(n)$	$\theta(1)$	$O(n^2 + m)$
Liste	$\theta(1)$	$\theta(n)$	$\theta(n)$	$O(n^2 + mn)$
Spreizbaum	$\theta(1)$	$\theta(\log n)^*$	$\theta(\log n)^*$	$O((m + n) \log n)$
Fibonacci-Halde	$\theta(1)$	$\theta(\log n)^*$	$\theta(1)^*$	$O(n \log n + m)$