Informatik 2

effiziente Algorithmen und Datenstrukturen

Mitschrift von www.kuertz.name

Hinweis: Dies ist **kein offizielles Script**, sondern nur eine private Mitschrift. Die Mitschriften sind teweilse **unvollständig, falsch oder inaktuell**, da sie aus dem Zeitraum 2001–2005 stammen. Falls jemand einen Fehler entdeckt, so freue ich mich dennoch über einen kurzen Hinweis per E-Mail – vielen Dank!

Klaas Ole Kürtz (klaasole@kuertz.net)

Inhaltsverzeichnis

1 Kurze Einführung

1.1 über die Veranstaltung

- Informatik II = effiziente Algorithmen (Alg.) und Datenstrukturen (DS)
- Warum wichtig? - Effiziente Algorithmen und Datenstrukturen sind die Grundlage für schnelle und speicherplatzsparende Programme
- Umfassende Behandlung von Algorithmen und Datenstrukturen in der Vorlesung:
  - Entwurf von effizienten Algorithmen und Datenstrukturen für Standardprobleme
  - Bewertung der Effizienz von Algorithmen und Datenstrukturen
  - Implementierung in JAVA
  - Einsatz in Beispielen

1.2 Größenordnungen (orders of magnitude)

Wir werden Laufzeiten und Speicher(platz)bedarf von Algorithmen bemessen. Dabei ist zu beachten:

- kleine Eingabegrößen sind eher uninteressant,
- eine genaue Bestimmung des Resourcenverbrauchs ist häufig sehr mühsam.

Daher sehen wir den Resourcenverbrauch zweier Algorithmen schon als gleich an, wenn sich beide für große Eingaben nur um einen multiplikativen Faktor unterscheiden, z.B. $n \mapsto n^2$ und $n \mapsto \frac{1}{2}n^2 - \frac{1}{2}n$ sollen als gleich angesehen werden.

Mathematisierung: Für jedes Tupel $n = (n_0, ..., n_{l-1}) \in \mathbb{N}^l$ setzen wir fest: $\|n\| = n_0 + ... + n_{l-1}$. Funktionen $f, g : \mathbb{N}^l \to \mathbb{R}_{\geq 0}$ heißen *asymptotisch äquivalent*, in Zeichen $f \equiv g$, wenn es $c, d \in \mathbb{R}_{\geq 0}$ und $k \in \mathbb{N}$ gibt, so daß $c \cdot f(n) \leq g(n) \leq d \cdot f(n)$ für alle n mit $\|n\| \geq k$ gilt.

Beispiel: $f : n \mapsto \frac{3}{2}n^2 + 32n - 1$ und $g : n \mapsto n^2$ sind asymptotisch äquivalent, d.h. $f \equiv g$.

Schreibweise: Die Äquivalenzklasse einer Funktion f wie oben wird mit $\Theta(f)$ bezeichnet, d.h. $\Theta(f) = \{g \mid f \equiv g\}$.

Weitere Definitionen: Die Funktion f ist *asymptotisch kleiner gleich* g, i.Z.

$f \sqsubseteq g$, wenn es $d \in \mathbb{R}_{\geq 0}$ und $k \in \mathbb{N}$ gibt, so daß $f(n) \leq d \cdot g(n)$ f.a. n mit $\|n\| \geq k$. Dazu passende Menge von Funktionen:

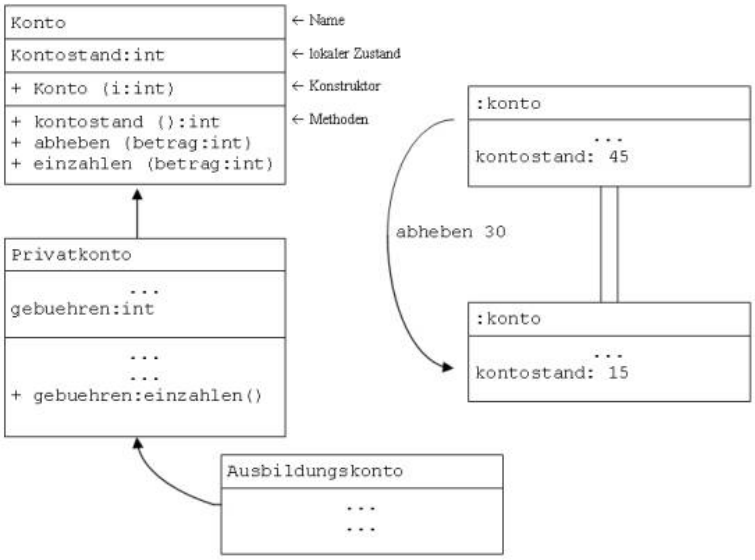
$$O(f) = \{g \mid g \sqsubseteq f\}$$

$$\Omega(f) = \{g \mid f \sqsubseteq g\}$$

$$\Theta(f) = \Omega(f) \cap O(f)$$

1.3 Graphische Notation für Objekte (objects) und Klassen (classes)

Wir zeichnen sowohl Klassen als auch Objekte in Form von Boxen. Modifikatoren der Eigenschaften und Methoden werden symbolisch dargestellt: - für private, # für protected und + für public. Schnittstellen erhalten den Zusatz `<<interface>>`. Statische Methoden/Eigenschaften werden unterstrichen. Beispiel Bankkonto (siehe letztes Semester):



1.4 Eine fundamentale Datenstruktur: das Feld (array)

In einem Feld werden einfache Daten oder Referenzen auf Objekte hintereinander angeordnet. Dabei wird die Anzahl der Einträge bei Erzeugung des Feldes festgelegt. Auf die Daten kann über ihren Index ohne Einschränkung schnell (in konstanter Zeit) zugegriffen werden. Vorstellung: zusammenhängender Speicherbereich, daher schneller Zugriff.

Notation: $a[0\,..\,n-1]$ für ein Feld der Länge n mit Namen a; Zugriff auf das Element mit dem Index $i < n$: $a[i]$.

Zeitverbrauch und Platzbedarf:

- Anlegen eines Feldes der Länge n einschließlich Initialisierung mit Standardwert: $\Theta(n)$
- Zugriff auf ein Feldelement (Auslesen oder ändern): $\Theta(1)$
- Platzbedarf $\Theta(n)$

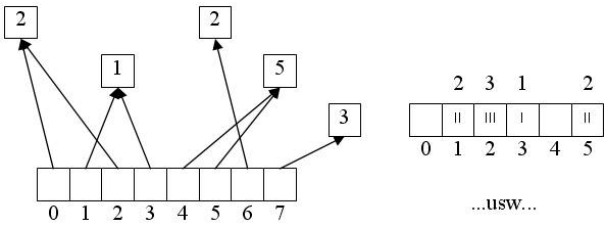
JAVA: Für jeden Datentyp gibt es einen Feldtyp, der Daten dieses Typs als Elemente hat.

- Typbeschreibung: `Elementtyp[]`, z.B. `int[]`
- Anlegen: `new Elementtyp[n]`, z.B. `new int[5]`
- Anlegen mit Initialisierung: z.B. `new Elementtyp[n]{..., ...}`
- Zugriff auf Feldelement: `Name[Ausdruck]`, z.B. `a[3]`

Anwendung[1]: Wir wollen ein Feld $a[0\,..\,n-1]$ von Objekten sortieren. Dabei besitzt jedes Objekt einen Sortierschlüssel *key*, der aus einer reellen Zahl besteht. Die Sortierung soll nach aufsteigenden Schlüsseln erfolgen. Wir betrachten den Spezialfall, bei dem die Schlüssel aus einem kleinen Bereich kommen und das Feld groß sein soll, etwa Schlüssel aus $0, 1, ...N-1$ mit $N = 256$ und $n \geq 1.000.000$.

Einfacher **Ansatz** (*CountingSort*): Wir durchlaufen das Feld einmal und zählen, wie häufig jeder Schlüssel vorkommt. Ergebnis wird in Feld notiert. Danach legen wir ein neues Feld an, in das wir die Objekte in sortierter Reihenfolge schreiben. Dazu rechnen wir für jeden Schlüssel aus, wo das erste Objekt mit diesem Schlüssel stehen müsste. Ergebnis wird in neuem Feld von Positionsvariablen notiert. Dann durchlaufen wir das ursprüngliche Feld von links nach rechts und schreiben die Objekte an die richtige Stelle in das Ergebnisfeld, wobei die richtige Stelle durch die entsprechende Positionsvariable vorgegeben wird. Diese wird aktualisiert, nachdem ein Objekt übertragen wurde, und zwar durch erhöhen um 1.

[1] siehe Alorithmus *CountingSort* im Netz



Bemerkung zur Notation der Algorithmen: Obige Notation eines Algorithmus wird Pseudocode genannt, da er nicht lauffähig ist, es sich also nicht um eine wirkliche Programmiersprache handelt. Dieser Code ist kurz und knapp und vermittelt die wesentliche Funktionsweise des Algorithmus.

Laufzeitanalyse:

- Zeile 1: N Durchläufe mit N Zuweisungen, d.h. $\Theta(N)$
- Zeile 2-3: n Durchläufe mit n Zuweisungen, d.h. $\Theta(n)$
- Zeile 4: eine Zuweisung, $\Theta(1)$
- Zeile 5: $N-1$ Durchläufe/Zuweisungen, also $\Theta(N)$
- Zeile 6-9: $\Theta(n)$

$\Rightarrow$ Insgesamt: $\Theta(n+N)$, bei konstantem N (z.B. 256) ist die Laufzeit $\Theta(n)$, man sagt *linear*.

Besonderheit des obigen Verfahrens: Laufzeit ist für alle Eingaben einer festen Länge gleich. Wenn das nicht der Fall ist, interessieren wir uns in der Regel für den schlechtesten Fall.

Definition: Ist A ein Algorithmus, bei dem die Größe jeder Eingabe durch Parameter $n_0, ..., n_{l-1}$ beschrieben wird, so ist die *pessimale Laufzeitfunktion* des Algorithmus die Funktion $t_A : \mathbb{N}^l \to \mathbb{N}$, die jedem Tupel n das Maximum aller Rechenschritte zuordnet, die der Algorithmus A bei der Lösung der Eingaben mit Größenparameter n ausführt.

Speicherplatzanalyse:

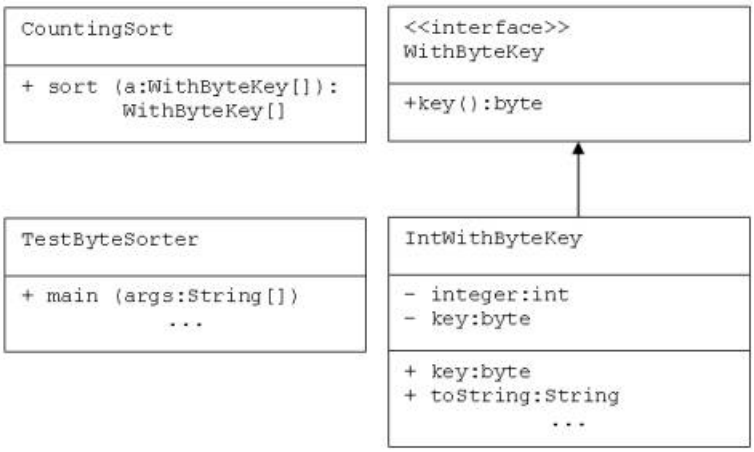
- Eingabefeld a wird ignoriert
- Ausgabefeld b verbraucht $\Theta(n)$
- Zählerfelder c und d verbrauchen jeweils $\Theta(N)$

$\Rightarrow$ Insgesamt: $\Theta(n+N)$ neben dem Platz für das Eingabefeld

Bemerkung: Üblicherweise wird bei der Betrachtung des Speicherplatzhbedarfs nur der Speicher gezählt, der zusätzlich zur Darstellung der Eingabe - die sowieso da ist - benötigt wird.

Satz: Sowohl Laufzeit als auch Speicherplatzbedarf von *CountingSort* sind (in) $\Theta(n + N)$.

Java-Implementierung: Wir sortieren Objekte, die einen Sortierschlüssel haben, der aus einem Byte besteht. Dazu deklarieren wir eine Schnittstelle $WithByteKey$. Um das Sortierverfahren testen zu können, deklarieren wir weiterhin eine Klasse $IntWithByteKey$, die $WithByteKey$ implementiert und deren Objekte aus einer int-Zahl und einem byte-Schlüssel bestehen. Das eigentliche Sortierverfahren ist als Methode *sort* in der Klasse $CountingSort$ implementiert[2].



Laufzeit:

Feldlänge	10^4	10^5	$6 \cdot 10^5$	$8 \cdot 10^5$	$10 \cdot 10^5$	$12 \cdot 10^5$
CountingSort	0	5	35	44	58	70
JavaSort	2	37	301	411	528	648

[2]Code ist im Netz zu finden

2 Lineare Datenstrukturen

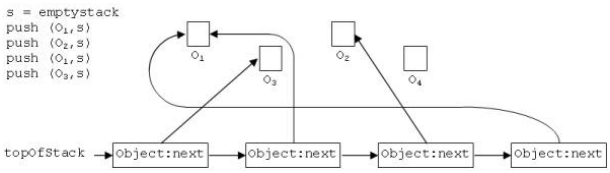
Wir studieren Datenstrukturen, bei denen die Daten lineare angeordnet sind, wie an einer Schnur aufgehängt. Beispiel aus dem letzten Kapitel: Felder.

2.1 Keller (stack)

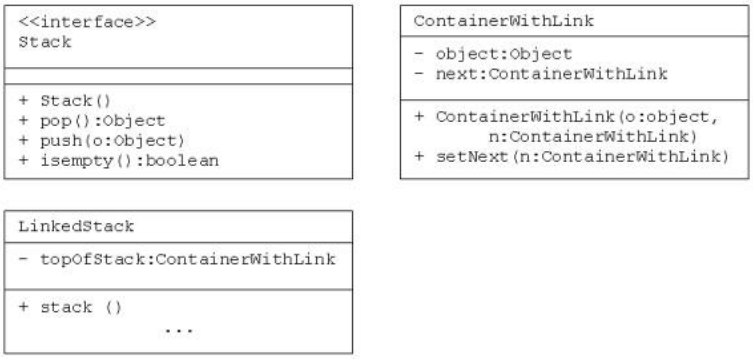
Ein Keller ist ein abstrakter Datentyp, der nach dem *LiFo*-Prinzip arbeitet, last in first out. Der Einfachheit halber werden wir *top* nicht betrachten.

```
S = {emptystack/0; isempty?/1; top/1; pop/1; push/2}
X = {s, x}
pop(push(x, s)) = x
top(push(x, s)) = x
isempty?(emptystack()) = true
isempty?(push(x, s)) = false
```

Implementierung: Jedes Element eines Stacks wird in einen Container gepackt, d.h. der Container referenziert das Objekt durch eine Eigenschaft *object*. Jeder Container hat außerdem eine Referenz *next* auf das Element, das direkt davor in den Keller gepackt wurde. Außerdem gibt es eine zusätzliche Referenz $TopOfStack$ auf das oberste Element, damit wir wissen, wo wir einfügen und löschen können[3].



Java: Wir definieren eine Schnittstelle *Stack* und eine Klasse *LinkedStack*, die einen Stack nach obigem Muster implementiert. Außerdem wird eine Klasse *ContainerWithLink* deklariert.



[3]Code im Netz

Laufzeitanalyse: Laufzeitanalyse: Alle Operationen haben eine Laufzeit von $\Theta(1)$.

Anwendung: Wir wollen erkennen, ob eine Zeichenkette „wohlgeklammert“ ist, z.B. ist `()[()]` wohlgeklammert, `()[` aber nicht. *Wohlgeklammerte Ausdrücke* werden durch die folgenden Regeln beschrieben:

1. Die leere Zeichenkette ist ein wohlgeklammerter (wgk.) Ausdruck
2. Ist A ein wgk. Ausdruck, so auch (A), $[A]$ und $\{A\}$.
3. Sind A und B wgk. Ausdrücke, so auch AB.
4. Wgk. Ausdrücke entstehen nur durch Anwendung der Regeln 1 bis 3.

- Zusätzliche Vereinbarung: Die Zeichenkette `()`, `[]`, `{}` werden als Klammersegmente bezeichnet.

Vorüberlegung: Wenn wir in einem wohlgeklammerten Ausdruck Klammersegmente in der Reihenfolge streichen, wie sie entstanden sind, dann bleibt nur die leere Zeichenkette übrig.

Vermutung: Wenn wir in beliebiger Reihenfolge streichen, dann bleibt genau dann die leere Zeichenkette übrig, wenn die ursprüngliche Zeichenkette wohlgeklammert war.

Lemma: Ist A ein beliebiger wohlgeklammerter Ausdruck und ist $A = A'()A''$ eine beliebige Zerlegung von A, so ist auch $A'A''$ ein wgk. Ausdruck. Gleiches gilt für eckige und geschweifte Klammern.

Beweis: Induktion über den Aufbau von A. Induktionsanfang: Ist A die leere Zeichenkette, so ist die Behauptung trivialerweise erfüllt. Induktionsschritt:

1. Sei $A = BC$ mit B und C wgk. und nicht leer. Da B nicht auf eine offene Klammer enden kann, gibt es eine Zeichenkette D, so daß $B = A'()D$ oder $C = D()A''$ gilt. Dann ist nach Induktionsvoraussetzung $B = A'D$ bzw. $C = DA''$ wohlgeklammert, also auch $A'DC = A'A''$ bzw. $BDA'' = A'A''$.
2. Sei $A = (B)$ (analog $A = [B]$ oder $A = \{B\}$). Falls B leer ist, ist die Behauptung trivial. Ist B nicht leer, so kann B nicht mit) beginnen und nicht auf (enden. Dann muß es D' und D'' geben, so daß $A' = (D'$, $A'' = D'')$, $B = D'()D''$ und $A'A'' = (D'D'')$. Nach Induktionsvoraussetzung ist $D'D''$ ein wohlgeklammerter Ausdruck, also auch $(D'D'')$.

□

Festlegung: Eine Zeichenkette heißt *klammersegmentfrei*, wenn sie kein Klammersegment enthält.

Folgerung: Ist A ein wohlgeklammerter Ausdruck und B eine klammersegmentfreie Zeichenkette, die aus A durch Streichen von Klammersegmenten entstanden ist, so ist B leer.

Beweis: Nach obigem Lemma muß B ein wohlgeklammerter Ausdruck sein. Da aber jeder nicht-leere wohlgeklammerte Ausdruck ein Klammersegment enthält, muß B leer sein. □

Frage: Können wir auch eine leere Zeichenkette erhalten, wenn wir von einem nicht-wohlgeklammerten Ausdruck ausgehen? Antwort: nein; dazu:

Lemma: Ist A ein beliebiger wohlgeklammerter Ausruck und $A = A'A''$ eine beliebige Zerlegung, so ist $A'()A''$ auch wohlgeklammert (analog für [] und {}).

Beweis: siehe Übung.

Folgerung: Ist A eine beliebige nicht-wohlgeklammerte Zeichenkette und B eine klammersegmentfreie Zeichenkette, die aus A durch sukzessives Steichen von Klammersegmenten entstanden ist, so ist B nicht leer.

Beweis: siehe Übung.

Idee für Algorithmus: Solange es in der gegebenen Zeichenkette Klammersegmente gibt, steichen wir diese. Genau dann, wenn die leere Zeichenkette übrigbleibt, war der ursprüngliche Ausdruck wohlgeklammert. Verfeinerung: Wir gehen beim Streichen von links nach rechts vor. Beispiel:

```
"([]){{}}"
"(){{}}"
"{{}}"
"{}"
""
```

Problem: Jedes löschen bedeutet, daß der Rest der Zeichenkette nach links verschoben werden muß - zu aufwendig!

Lösung: Wir löschen nicht, sondern merken uns separat, was links von der aktuellen Position steht.

Beispiel: Input: `([]){{}}`, Abfolge im Speicher: `([` - streichen - `(` - streichen - ε[4] - `{{` - streichen - `{` - streichen - fertig.

[4] leere Zeichenkette

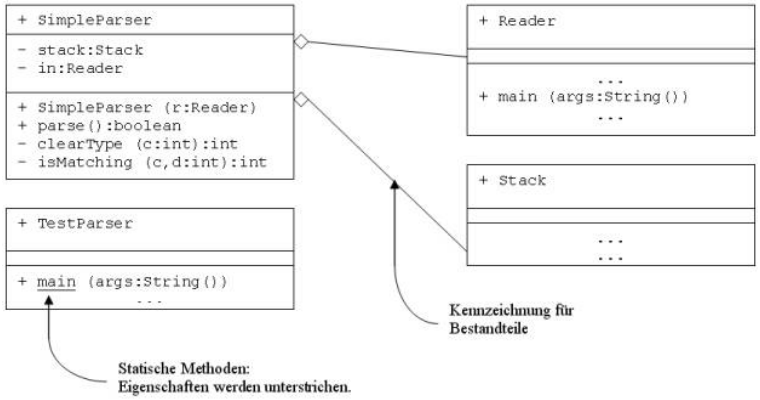
Beobachtung: Die separate Zeichenkette wird eigentlich wie ein Keller benutzt.

Algorithmus: Wir benutzen einen Keller und lesen die fragliche Zeichenkette von links nach rechts. Wenn wir eine öffnende Klammer lesen, schreiben wir diese in den Keller; wenn wir eine schließende Klammer lesen, holen wir eine Klammer vom Keller und vergleichen beide: Passen sie zusammen, geht es weiter; andernfalls war der Audruck nicht wohlgeklammert. Können wir die ganze Zeichenkette lesen, signalisieren wir dann einen Fehler, wenn der Keller nicht leer ist. Laufzeitanalyse: $\Theta(n)$.

JAVA: Wir schreiben eine Klasse *SimpelParser*, deren Objekte einen Ausdruck auf Wohlgeklammertheit überprüfen. Unterschied zu oben: die Eingabe ist kein Feld, sondern ein Zeichenstrom, der in JAVA durch eine Klasse *Reader* realisiert ist. Der Konstruktor für einen SimpleParser hat ebenfalls ein *Reader*-Object als Parameter.

Neben einer public-Methode *parse*() gibt es eine private Methode *charType*, die feststellt, ob die Eingabe aus einer offenen, geschlossenen oder gar keiner Klammer besteht, und eine private Methode *isMatching*, die überprüft, ob zwei Klammern zueinander passen.

Jeder *Reader* hat eine Methode *read*, die den nächsten Buchstaben liefert oder -1, falls es keinen nächsten Buchstaben gibt.



2.2 Schlange (queues)

Schlangen arbeiten nach dem *FiFo*-Prinzip, wie man es aus dem Supermarkt kennt: first in, first out. Einziger Unterschied zum Supermarkt: mehrfach anstellen möglich.

Operationen:

- *isEmpty* (leer?)
- *enqueue* (anstellen)
- *dequeue* (aus der Schlange entfernen)

Implementierung: Analog zu einem Keller, unter Benutzung von Containern. Allerdings benutzen wir jetzt zwei zusätzliche Referenzen, $frontOfQueue$ und $rearOfQueue$. Am Ende der Schlange halten wir einen zusätzlichen leeren Container.

Laufzeitanalyse: Alle Operationen $\Theta(1)$.

Implementierung: Analog zur Stack-Implementierung (siehe Netz).

Anwendung: Wir wollen wieder sortieren, und zwar in der gleichen Situation wie bei *CountingSort*. Neue Idee (*BucketSort*, einfache Variante): Wir legen für jeden Sortierschlüssel eine Schlange an. Dann durchlaufen wir das Feld einmal und reihen jedes gesuchte Objekt in die entsprechende Schlange ein. Zum Schluß schreiben wir nacheinander die Schlangen wieder zurück.

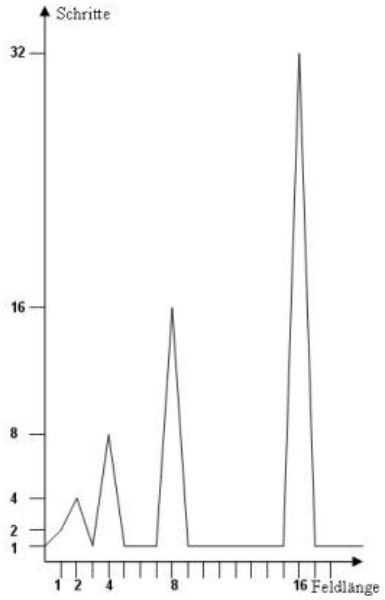
Laufzeitanalyse: $\Theta(N + n)$, also wie bei *CountingSort*.

Implementierung: siehe Netz, Vergleich der Laufzeiten[5]:

Datensätze	10	10^4	10^5	$6 \cdot 10^5$	$8 \cdot 10^5$	$10 \cdot 10^5$	$12 \cdot 10^5$
CountingSort	0	0	5	35	47	58	70
JavaSort	0	2	37	301	411	528	648
BucketSort	0	0	52	151	188	225	312

Alternativimplementierung für Schlangen: Wir realisieren eine Schlange durch ein Feld. Wenn das Feld zu klein wird, legen wir ein neues, doppelt so großes Feld an und kopieren die Daten in das neue Feld. Wir merken uns Anfang und Länge der Schlange, schreiben von links nach rechts, u.U. auch über die Feldgrenze hinaus und beginnen dann wieder vorn (modulo-Rechnung).

[5] nachträglich wenig aussagekräftig, vorher war keine Garbage Collection durchgeführt worden `Runtime.getRuntime().gc();`



Beobachtung: Wenn wir mit Kapazität 1 beginnen, so ist enqueue immer dann aufwendig, wenn die Schlange $n = 2^k$ viele Elemente enthält. Dann braucht der Algorithmus eine lineare Anzahl von Schritten, d.h., die Laufzeit ist $O(n)$. Sie ist aber nicht $\Omega(n)$ und damit auch nicht $\Theta(n)$:

Feldgröße	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Schritte	2	4	1	8	1	1	1	16	1	1	1	1	1	1	1	32

Definition: Die Funktion $f : \mathbb{N}^l \to \mathbb{N}$ ist asymptotisch immer wieder größer gleich g, in Zeichen: $f \geq_\infty g$, wenn es $d \in \mathbb{R}_{>0}$ und eine Folge $(a_n)_{n\in\mathbb{N}}$ gibt, so daß $f(a_i) \geq d \cdot g(a_i)$ und $\|a_i\| \geq i$ für alle i. Es sei:

$$\Omega_\infty(f) := \{g \mid g \geq_\infty f\} \quad \text{und} \quad \Theta_\infty(f) = O(f) \cap \Omega_\infty(f)$$

Vergleich der Laufzeiten: $BucketSort_2$ ist mit *ArrayQueue* implementiert statt mit *LinkedQueue*; Angaben in Millisekunden

Algorithmus	200T	400T	600T	800T	1.000T	1.200T
$CountingSort$	110	216	325	435	548	655
$BucketSort$	495	942	1367	1821	2336	3288
$BucketSort_2$	363	687	827	1358	1561	1745
$JavaSort$	1007	2141	3325	4602	5906	7244

Erklärung: Bei einer normalen Schlange (verkettete Implementierung) ist die Laufzeit für die Ausführung von m Operationen $\Theta(m)$. Das gleiche gilt

für die Feldimplementierung.

Satz: Wird eine Schlange durch Felder implementiert, so ist die Gesamtlaufzeit für m Operationen einschließlich der anfänglichen Erzeugung $\Theta(m)$.

Beweis: $\Omega(m)$ ist leicht einzusehen. Wir zeigen noch, daß die Laufzeit $O(m)$ ist und nehmen an, daß die Anfangskapazität $= 1$ ist. Seien c und d Konstanten, so daß die Laufzeit aller Operationen durch c beschränkt ist, abgesehen von *enqueue*: deren Laufzeit sei durch $c + d \cdot n$ beschränkt, wobei n die Größe der Schlange bezeichnet. Sei nun m ein beliebige Zahl mit $2^k \leq m < 2^{k+1}$. Dann ist die Gesamtlaufzeit von m Operationen beschränkt durch

$$\begin{aligned}
\sum_{i=0}^{m-1} c + \sum_{j=0}^{k} d2^j &= c \cdot m + d\left(\sum_{j=0}^{k} 2^j\right) \\
&= cm + d(2^{k+1} - 1) \\
&\leq cm + d \cdot 2m \\
&\leq (c + 2d)m
\end{aligned}$$

Damit ist die Laufzeit in $O(m)$. *Das erklärt's!* □

Definition: Sei eine Datenstruktur gegeben. Laufzeitangaben für die Operationen auf der Datenstruktur dürfen *amortisierte Laufzeit* genannt werden, wenn die tatsächliche Laufzeit von *aufeinanderfolgenden* Operationen der Summe der amortisierten Laufzeiten entspricht.

Satz: Die amortisierten Laufzeiten der Schlange bei zirkulärer Feldimplementierung sind jeweils $\Theta(1)$.

2.3 Listen

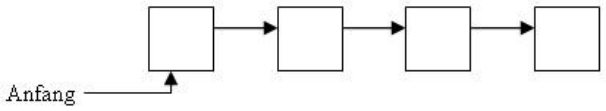
Listen sind Datenstrukturen, in denen alle Daten linear angeordnet sind, d.h., bei denen Elemente durchnummeriert sind, und die viele sinnvolle Operationen implementiert, um Felder, Keller, Schlangen etc. zu erweitern. Typischer Satz von Operationen:

- $newList$
- $size$
- $get(i)$ liefert Objekt an Stelle i
- $set(i, o)$ ersetzt Objekt an Stelle i durch o
- $add(i, o)$ Objekt wird an Stelle i gesedtzt; was vorher an Stelle $j \geq i$ war, ist nachher an Stelle $j + 1$

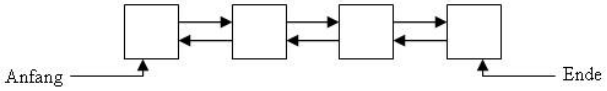
- $remove(i)$ entferne Stelle i, d.h., was vorher an Stelle $j > i$ war, ist nachher an Stelle $j - 1$
- $firstIndexOf(o)$ und $lastIndexOf(o)$ lieferen erste und letzte Stelle, an der o steht; wenn Objekt nicht vorhanden: -1
- $getFirst$, $getLast$, $removeFirst$, $removeLast$, $addFirst$, $addLast$

Implementierung: Viele Möglichkeiten mit unterschiedlichen Laufzeitverhalten und Speicherplatzbedarf:

1. Containerimplementierung
   (a) einfach verkettet (singly linked): Stelle i zeigt auf $i + 1$



   (b) doppelt verkettet (doubly linked): zwei Verweise pro Container, einer zeigt auf das nächste, der andere auf das vorhergehende Element



2. Feldimplementierung
   (a) einfache: Liste wird im Bereich $0..(size - 1)$ abgelegt
   (b) zirkulär: wie bei Schlange

Laufzeitanalysen: Parameter n ist die Listenlänge.

1. (a) pessimal: newList, size, getFirst, removeFirst, addFirst $\in \Theta(1)$
   pessimal: alle anderen $\in \Theta(n)$
   (b) verbesserte Laufzeiten $\Theta(1)$ bei getLast, removeLast und addLast
2. (a) pessimal: newList, size, get, set, newList, getLast, removeLast, getFirst $\in \Theta(1)$
   pessimal: addLast $\in \Theta_\infty(n)$
   pessimal: Rest $\in \Theta(n)$
   amortisiert: addLast $\in \Theta(1)$
   (b) amortisiert: addFirst $\in \Theta(1)$

	verkettet einfach	verkettet doppelt	Feld normal	Feld zirkulär
get	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
set	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
add	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$remove$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$firstIndexOf$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$lastIndexOf$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
$getFirst$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
$removeFirst$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
$addFirst$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)_\infty, \Theta(1)^*$
$getLast$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
$removeLast$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
$addLast$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)_\infty, \Theta(1)^*$	$\Theta(n)_\infty, \Theta(1)^*$

$^* \hat{=}$ amortisierte Laufzeit

JAVA:

```
<<interface>> list:                 +size:int
                                    +isEmpty:boolean
                                    ...

LinkedList implements list:         -size:int
                                    -frontOfList:ContainerWithLink
                                    ...

DoubleLinkedList implements list:   -size:int
                                    -frontOfList:ContainerWithTwoLinks
                                    -rearOfList:ContainerWithTwoLinks
                                    ...

ArrayList implements list:          -size:int
                                    -capacity:int
                                    -array:Object[]
                                    ...

CircularArrayList implement list:   -size:int
                                    -capacity:int
                                    -array:Object[]
                                    -frontOfList:int
                                    ...

ContainerWithLink:                  -object:Object
                                    -next:ContainerWithLink
                                    +getObject():Object
                                    +getNext():ContainerWithLink
                                    ...

ContainerWithTwoLinks:              -object:Object
                                    -next:ContainerWithTwoLinks
                                    -previous:ContainerWithTwoLinks
                                    +getObject():Object
                                    ...
```

Frage: Warum betrachten wir überhaupt Containerimplementierungen, wenn Feldimplementierungen theoretisch genauso gut bzw. besser sind?

- *Hat die Feldimplementierung Nachteile?*
  - Das Feld wird nie verkleinert, also Platzverschwendung! Genauer: Es gibt keinen Zusammenhang zwischen aktueller Feldgröße und Listenlänge.

    Lösungsansatz: Wir verkleinern das Feld bei Bedarf, aber: wenn wir es verkleinern, wenn die Listenlänge halb so groß wie die Feldlänge ist, dann ist die amortisierte Laufzeit nicht mehr konstant. Beispiel: Liste hat 16 Elemente, Feld hat Größe 16. Hinzufügen eines Elementes verdoppelt die Liste, wegnehmen halbiert wieder, hinzufügen ...

    Richtige Lösung: Halbieren des Feldes, wenn ein Viertel der Feldgröße erreicht wird.

- *Hat die Containerimplementierung Vorteile?*
  - Wenn man eine Referenz auf einen Container hat, kann man den Container dahinter schnell löschen oder einen weiteren Container dahinter schnell einfügen.
  - Außerdem ändert sich eine Refernz auf einen Container nicht, wenn die Liste verändert wird. Abstrakt: zusätzliche Adressierungsmöglichkeit. Wir sprechen von *Positionen*. Es bieten sich also zusätzliche Listenoperationen an, z.B.
    - $remove(p)$ löscht Objekt nach Position p
    - $add(p, o)$ fügt Objekt o direkt nach Position p ein
    - $get(p)$, $set(p)$
    - $getIndexOfPosition(p)$, $getPosition(i)$, ...
    - $getFirstPosition()$, $getLastPosition()$, ...
    - $getPredecessor(p)$, $getSuccessor(p)$, ...

Satz: Die pessimalen Laufzeiten der zusätzlichen Listenoperationen auf den Containerimplementierungen ist $\Theta(1)$, abgesehen von *getLastPosition* und *getPredecessor* für die einfach verkettete Liste und *getIndexOfPosition* und *getPosition* für beide Implementierungen, deren pessimale Laufzeit ist $\Theta(n)$.

Java: Wir erweitern die Schnittstelle *List* zu einer Schnittstelle *PositionalList* mit den neuen Methoden. Wir ergänzen die Containerimplementierungen um diese Methoden und deklarieren sie als Implementierungen der neuen

Schnittstelle. Außerdem fassen wir die beiden Containerklassen geeignet zusammen.

3 Verzweigte Datenstrukturen

Lineare Datenstrukturen sind einfach, haben aber Defizite. Deshalb betrachten wir komliziertere Datenstrukturen[6].

3.1 Motivation

Frage: Wie müßte eine einfache Datenstruktur aussehen, damit wir leicht damit sortieren können? Sie bräuchte lediglich zwei Operationen zu unterstützen: *add* und *extractMin*, Sortierverfahren: *GenericSort*.

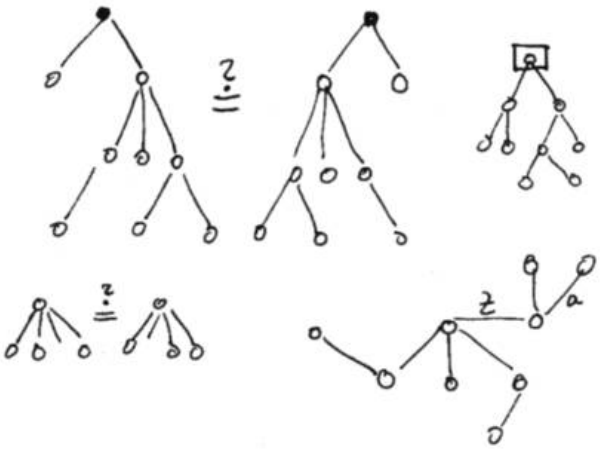
Sprechweise: Eine solche Datenstruktur heißt Prioritätsschlange. Wie implementiert man eine Prioritätsschlange? Z.B. durch eine Liste:

- erste Variante: *add* fügt Elemente beliebig (vorn) ein. Dann muß *extractMin* die Liste im schlechtesten Fall einmal ganz durchlaufen.
- zweite Variante: *add* fügt Elemente so ein, daß die Liste alle Elemente in sortierter Reihenfolge enthält. Dann muß die Liste im schlimmsten Fall bei *add* einmal durchlaufen werden, dafür ist *extractMin* dann schnell

Laufzeitanalyse: In beiden Fällen ergibt sich als pessimale Laufzeit $\Theta(n^2)$, geht's besser? Klar! Wenn man eine bessere Implementierung der Prioritätswarteschlange wählt, die verzweigte Datenstrukturen nutzt.

3.2 Baum ist nicht gleich Baum

Frage: Was ist ein Baum? Alles, was irgendwie so aussieht:



[6] „Wer hätte das gedacht!?“

Wir betrachten zunächst Bäume,

- die endlich viele, beschriftete Knoten besitzen
- die gerichtet und geordnet sind
- bei denen die Kinder einfach durchnummeriert sind

Definition: Eine endliche Folge von natürlichen Zahlen heißt *Knoten*. Eine nichtleere, endliche Menge T von Knoten heißt *unbeschrifteter Baum*, wenn

(T_1) Ist $(i_0, ..., i_l) \in T$, so ist auch $(i_0, ..., i_{l-1}) \in T$.

(T_2) Ist $(i_0, ..., i_l) \in T$, so ist auch $(i_0, ..., i_{l-1}, j)$ mit $j < i_l \in T$.

Beispiel: $T = \{(), (0), (1), (1,0), (1,1), (1,2)\}$

Definition: Sei B eine Menge. Ein *B-beschrifteter Baum* ist ein Paar (T, β), wobei T ein unbeschrifteter Baum ist und $\beta : T \to B$ eine sogenannte Beschriftungsfunktion ist.

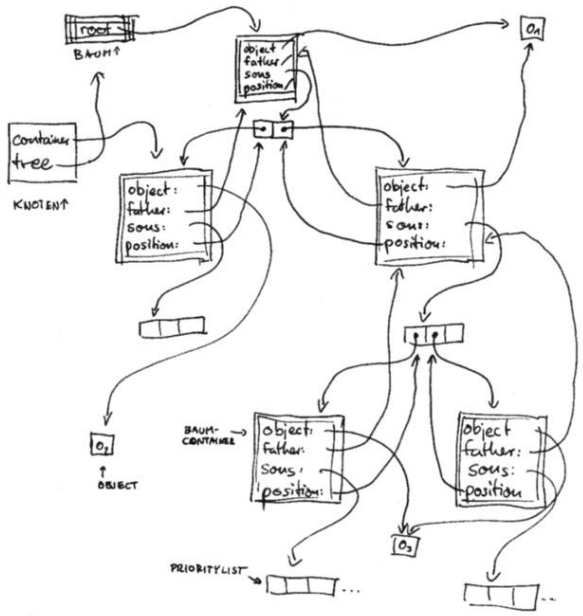
Definitionen: Jeder Baum besitzt die leere Folge () als Knoten. Diese wird *Wurzel* genannt. Ist $(i_0, ..., i_{l-1})$ ein Knoten eines Baumes T, so heißt $(i_0, ..., i_{l-2})$ sein *Vater* $(l > 0)$. Die Knoten von T der Form $(i_0, ..., i_{l-1}, j)$ sind seine *Söhne*. Alle Knoten von T der Form $(i_0, ..., i_{l-1}, ...)$ heißen *Nachfahren*. Die Knoten der Form $(i_0, ..., i_{l-2}, j)$ sind seine *Brüder*, für *ältere* Brüder gilt $j < i_{l-1}$ und für *jüngere* Brüder gilt $i_{l-1} < j$. Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Söhne. Ein Knoten mit Verzweigungsgrad 0 heißt *Blatt* und ein Knoten mit Verzweigungsgrad > 0 heißt *innerer Knoten*. Die *Größe* eines Baumes ist die Anzahl seiner Knoten. Die *Höhe* eines Baumes ist die maximale Folgenlänge seiner Knoten.

Abstrakter Datentyp Baum: An Operationen lassen wir (der Einfachheit halber) nur wenige zu, da wir Baume nur auf- und nicht abbauen wollen:

- *getObject*(v) liefert Objekt zu Knoten v
- *getRoot*() liefert die Wurzel

- $getFather(v)$ liefert den Vater
- $getOldestSon(v)$ liefert ältesten Sohn
- $getYoungerBrother(v)$ liefert jüngeren Bruder
- $addSon(v, o)$ wird jüngster Sohn
- Zusätzlicher Abstrakter Datentyp Knoten

Implementierung: Ein Baum besteht aus miteinander verknüpften $TreeContainern$. Jeder $TreeContainer$ besitzt eine Refernz $father$ auf seinen Vater, eine Refernz $sons$ auf die Liste seiner Söhne, eine Referenz auf seine Position in der Liste der Söhne seines Vaters und eine Referenz auf das gespeicherte Objekt. Der Baum selbst besitzt eine Referenz $root$ auf seine Wurzel. Ein Knoten besteht aus einer Referenz $container$ auf einen $TreeContainer$ und einer Referenz $tree$ auf seinen Baum.



Java: Es gibt eine Schnittstelle $Tree$, in die durch $linkedTree$ implementiert wird. Es gibt eine Klasse $TreeContainer$ und eine abstrakte Klasse $Node$. Diese wird erweitert um $LinkedTreeNode$.

```
<<interface>> Tree            getObject()
                              ...
LinkedTree implements Tree    root:TreeContainer
                              ...
```

`TreeContainer`	`object:Object`
	`father:TreeContainer`
	`sons:PositionalList`
	`...`
`Node`	`getOldestSon():Node`
	`...`
`LinkedTreeNode`	`container:TreeContainer`
	`tree:LinkedTree`
	`...`
`PositionalList`	`...`

3.3 Algorithmen auf Bäumen - Rekursion

Algorithmische Probleme auf Bäumen werden häufig durch **Rekursion** gelöst. Dabei zerlegt man einen Baum in die „Teilbäume“, die unterhalb der Wurzel liegen, löst das Problem für diese kleineren Bäume zuerst und fügt dann die berechneten Lösungen zusammen (*„divide and conquer!“*).

Lemma: Ist T ein unbeschrifteter Baum und $(i_0, ..., i_{l-1}) \in T$ ein Knoten, so ist

$$T' = \{ (j_0, ..., j_{n-1}) \mid (i_0, ..., i_{l-1}, j_0, ..., j_{n-1}) \in T \}$$

ein Baum. **Beweis:**

- T' ist nicht leer, da $(i_0, ..., i_{l-1}) \in T$ und damit $() \in T'$ ist.
- (T_1) Ist $(j_0, ..., j_m) \in T'$, so ist $(i_0, ..., i_{l-1}, j_0, ..., j_m) \in T$ und damit auch $(i_0, ..., i_{l-1}, j_0, ..., j_{m-1}) \in T$, also gilt $(j_0, ..., j_{m-1}) \in T'$.
- (T_2) analog. □

Schreibweise: Der obige Baum T' wird mit $T \downarrow (i_0, ..., i_{l-1})$ bezeichnet und heißt *Teilbaum* von T mit Wurzel $(i_0, ..., i_{l-1})$.

Beispiel Wir wollen die Größe eines Baumes ausrechnen. Einfacher rekursiver Ansatz: Hat die Wurzel k Söhne und sind $s_0, ..., s_{k-1}$ die Größen der zugehörigen Teilbäume, so ist die Größe des Gesamtbaumes gegeben durch $1 + \Sigma_{i=0}^{k-1} s_i$.

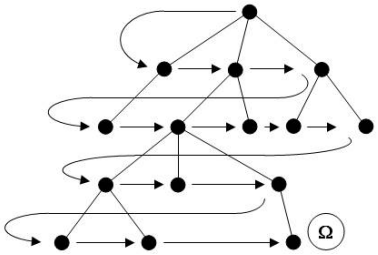
3.4 Baumordnungen (tree orderings) und Traversierungen (traversals)

Erinnerung: Listen haben eine natürliche lineare Ordnung.

Frage: Was machen wir mit Bäumen? **Antwort:** Es gibt viele Möglichkeiten.

Definition: Eine Ordnungs(relation) auf einer Menge M ist eine irreflexive, transitive Relation auf M. Eine Ordnungsrelation $<$ auf einer Menge M heißt *linear*, wenn für alle $x, y \in M$ entweder $x < y$, $y < x$ oder $x = y$ gilt. Ein $x \in M$ heißt *maximales Element*, wenn es kein y mit $x < y$ gibt; analog: minimales Element. Ein Element $x \in M$ heißt *größtes Element*, wenn für alle $y \in M$ gilt: $y \leq x$; analog: *kleinstes Element*. Ein Element y heißt *Nachfolger* von x, wenn y ein minmales Element von $\{z \in M \mid z > x\}$ ist; analog *Vorgänger*. Eine lineare Ordnung heißt *diskret*, wenn jedes Element einen Nachfolger hat oder größtes Element ist und wenn jedes Element einen Vorgänger hat oder kleinstes Element ist.

3.4.1 Ebenenordnung (level order)

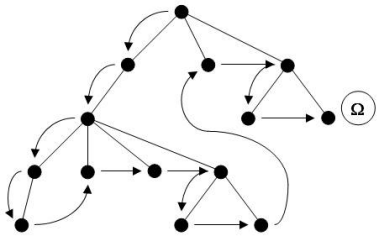


Definition: Ist T ein Baum, so ist die Ebenenordnung $<^T_{lev}$ auf T definiert wie folgt: Es gilt $(i_0, ..., i_{l-1}) <^T_{lev} (j_0, ..., j_{m-1})$ genau dann, wenn

$$l < m$$

$$\vee \; l = m \text{ und es gibt } p < l \text{ mit } i_r = j_r \text{ für } r < p \text{ und } i_p < j_p$$

3.4.2 Verzeichnisordnung (pre order)



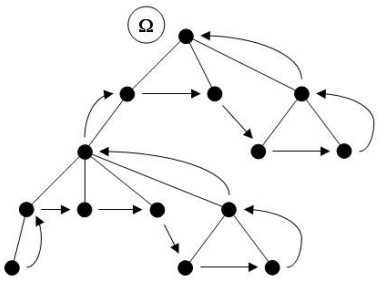
Definition: Ist T ein Baum, so ist die Verzeichnisordnung $<^T_{pre}$ auf T definiert wie folgt: Es gilt

- Falls T einen Knoten hat, so ist $<^T_{pre}$ leer.

- Falls T mehr als einen Knoten hat, so sei k der Verzweigungsgrad der Wurzel. Sei $T_i := T \downarrow (i)$ für jedes $i < k$. Es gilt $(i_0, ..., i_{l-1}) <^T_{pre} (j_0, ..., j_{m-1})$ genau dann, wenn

  $$\begin{aligned} & l = 0 \text{ und } m > 0 \\ \vee\ & l > 0 \text{ und } m > 0, \text{ sowie } i_0 < j_0 \\ \vee\ & l, m > 0,\ i_0 = j_0 \text{ und } (i_1, ..., i_{l-1}) <^{T_{i_0}}_{pre} (j_1, ..., j_{m-1}) \end{aligned}$$

3.4.3 Tiefenordnung (post order)



Definition: Ist T ein Baum, so ist die Tiefenordnung $<^T_{post}$ auf T definiert wie folgt: Es gilt

- Falls T einen Knoten hat, so ist $<^T_{post}$ leer.
- Falls T mehr als einen Knoten hat, so sei k der Verzweigungsgrad der Wurzel. Sei $T_i := T \downarrow (i)$ für jedes $i < k$. Es gilt $(i_0, ..., i_{l-1}) <^T_{post} (j_0, ..., j_{m-1})$ genau dann, wenn

  $$\begin{aligned} & l > 0 \text{ und } m = 0 \\ \vee\ & l > 0 \text{ und } m > 0, \text{ sowie } i_0 < j_0 \\ \vee\ & l, m > 0,\ i_0 = j_0 \text{ und } (i_1, ..., i_{l-1}) <^{T_{i_0}}_{post} (j_1, ..., j_{m-1}) \end{aligned}$$

3.4.4 Implementierungen etc.

Implementierung: Die Algorithmen zur Berechnung einer Liste der Knoten eines Baumes in Verzeichnis- oder Tiefenordnung haben einen einfachen rekursiven Ansatz, der durch die Definition vorgegeben ist.

Problem: Bei der Ebenenordnung bietet sich keine rekursive Lösung an: Die Wurzeln der Teilbäume müssen vor den anderen Knoten behandelt werden. Das deutet auf Abarbeitung gemäß einer (einfachen) Priorität hin, genauer: nach Benutzung einer Schlange (sic!).

Ansatz: Zunächst die Wurzel in die Schlange einreihen, danach: Solange die Schlange nicht leer ist: oberstes Element der Schlange entnehmen, einreihen in die Ordnung, dessen Kinder in die Schlange stecken - fertig!

Frage: Können wir auch leicht zu einem vorgegebenen Knoten den Nachfolger bezüglich einer der Ordnungen bestimmen?

Antwort: Mit ein wenig mehr Aufwand, z.B. Nachfolger von K in der Verzeichnisordnung:

1. Falls K ein innerer Knoten ist, ist der älteste Sohn sein Nachfolger.
2. Ist K ein Blatt, so ist es sein jüngerer Bruder.
3. Gibt es den nicht, dann ist es der jüngere Bruder des Vaters von K.
4. ... usw.

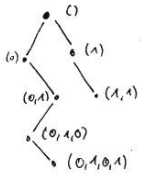
Satz: Alle bisherigen Baumalgorithmen haben $\Theta(n)$ als pessimale Laufzeit, wobei n für die Größe des Baumes steht. Alle Baumoperationen haben die Laufzeit $\Theta(1)$ bei Implementierung der Sohnliste durch eine doppelt verkettete Liste.

Beweisansatz: Laufzeitabschätzung für den Preorder-Algorithmus:

1. Trivialerweise ist die Laufzeit $\Omega(n)$
2. Die Laufzeit ist $\leq c \cdot$ (Anzahl der rekursiven Aufrufe + Durchläufe der *while*-Schleife), also $\leq c \cdot (n + n) = 2 \cdot c \cdot n$ für geeignetes c.

3.5 Binärbäume

Alle verzweigten Datenstrukturen im Rest des Kapitels basieren auf so genannten Binärbäumen:



Definition: Eine endliche Menge T von Knoten heißt *binärer Baum*, wenn $(T1)$ erfüllt ist und die Folgenglieder aus $\{0, 1\}$ stammen. Der *unendliche Binärbaum* ist die Menge aller endlichen Folgen über $\{0, 1\}$. Der unendliche Binärbaum wird mit T_B bezeichnet, die zugehörige Ebenenordnung mit $<_{lev}$. Die Definitionen der übrigen Begriffe wie *Sohn*, *Verzweigungsgrad* etc. lassen

sich in natürlicher Weise übertragen. Die Höhe des leeren Baumes ist $-\infty$. Verzweigungsgrad eines Knotens = Anzahl der Söhne.

Wichtigste Eigenschaft: binäre Bäume $2^{n+1}-1$ haben logarithmische Höhe, sofern sie „gut gepackt", d.h. binärer Baum T heißt *vollständig*, wenn für jeden Knoten $v \in T$ gilt: Ist u ein Knoten des unendlichen Binärbaums mit $u <_{lev} v$, so ist $v \in T$.

Lemma: Für jeden vollständigen Baum T gilt: $h(T) \leq \log(s(T))$.

Beweis: Trivial für $s(T) = 0$. Sei $v = (i_0, ..., i_{l-1})$ der gröste Knoten von T bezüglich der Ebenenordnung. Dann ist $T = \{u \in T_b \mid u <_{lev} v\}$. Daraus folgt $h(T) = l$ und

$$\begin{aligned} s(T) &\geq |\{v\}| + |\{(j_0, ..., j_{n-1}) \mid 0 \leq n < l, j_i \in \{0,1\}\}| \\ &= 1 + \sum_{i=0}^{l-1} 2^i \\ &= 1 + 2^l - 1 = 2^l \end{aligned}$$

Also: $h(T) = l = \log 2^l \leq \log(s(T))$. □

Einsatz von Binärbäumen (in diesem Kapitel): Verwaltung von Objekten, die einen Sortierschlüssel *key* besitzen und dadurch linear angeordnet sind.

Implementierung: Wir setzen Binärbäume aus $BinaryTreeContainern$ zusammen, die jeweils eine Referenz $leftSon$ und $rightSon$ auf den linken bzw. rechten Sohn besitzen und eine Referenz $father$ auf den Vater. Außerdem besitzen sie eine Referenz *object* auf ein Objekt, das einen Sortierschlüssel besitzt.

JAVA: Objekte einer Klasse, die einen Sortierschlüssel besitzen, implementieren die Schnittstelle $WithSortingKey$. Durch die Ordnung auf den Schlüsseln müssen sie linear angeordnet sein, d.h. Objekte mit gleichem Schlüssel müssen gleich sein.

Als Sortierschlüssel erlauben wir Objekte, die die JAVA-eigene Schnittstelle *Comparable* implementieren die für jedes Objekt die Methode *compareTo* vorsieht: $o_0.compareTo(o_1)$ liefert

- einen negativen Wert, wenn $o_0 < o_1$ gilt
- Null, wenn $o_0.equals(o_1)$ wahr ist
- einen positiven Wert, wenn $o_0 > o_1$ gilt

Definition: Für Binäbäume ist die *Flachordnung* (*inorder*) $<_{in}^T$ induktiv definiert wie folgt:

- Besteht T aus keinem oder einem Knoten, so ist $<_{in}^{T}$ leer.
- Besteht T aus mehr als einem Knoten, so sei $T_0 := T \downarrow (0)$ und $T_1 := T \downarrow (1)$. Es gilt dann $(i_0, ..., i_{l-1}) <_{in}^{T} (j_0, ..., j_{m-1})$ genau dann, wenn gilt:
  - $l > 0$ und $i_0 = 0$ und $m = 0$
  - $l = 0$ und $j_0 = 1$ und $m > 0$
  - $l, m > 0$ und $i_0 < j_0$
  - $l, m > 0$ und $i_0 = j_0$ und $(i_1, ..., i_{l-1}) <_{in}^{T_{i_0}} (j_1, ..., j_{m-1})$

3.6 Prioritätaschlangen (priority queues) und Halden (heaps)

Erinnerung: Wir suchen wieder nach einer effizienten Implementierung einer Prioritätsschlange.

Ansatz: Bei einer sortierten Liste ist ein Element kleiner gleich seinem Nachfolger. In einer Halde ist ein Knoten kleiner gleich seinen Söhnen.

Definition: Ein beschrifteter Binärbaum (T, β) heißt *Halde*, wenn gilt:

(H_v) Für jeden Sohn w von v gilt: $\beta(v) \leq \beta(w)$

(H) Für jeden Knoten v gilt H_v

(V) T ist vollständig

Satz:

1. Die Höhe einer Halde ist höchstens der Logarithmus der Größe.
2. Die Wurzel einer nichtleeren Halde hat minimale Beschriftung.

Beweis:

1. folgt aus dem obigen Lemma
2. Sei $v = (i_0, ..., i_{l-1})$ ein Knoten mit minimaler Beschriftung und kleinstem l unter diesen. Falls $l = 0$, so stimmt die Behauptung. Falls $l > 0$, so betrachten wir den Vater $w = (i_0, ..., i_{l-2})$ von v. Es würde $\beta(w) > \beta(v)$ gelten, dann wäre (H_w) verletzt, Widerspruch. □

Ziel: Implementieren von „Wurzel entfernen“ und „hinzufügen“, $extractRoot$ und add, in Zeit $O(h(T))$

Idee: fürs Einfügen: Wir erzeugen am Ende der Halte einen neuen Knoten und beschriften ihn mit dem neuen Wert. Dann ist die Haldeneigenschaft höchstens für den Vater verletzt. Wenn das so ist, tauschen wir Vater- und Sohnbeschriftung aus. Dann ist die Haldeneigenschaft höchstens für den Großvater verletzt. Wenn das so ist... An der Wurzel ist nichts weiter zu tun.

Idee fürs Entfernen der Wurzel: Wir entfernen die Beschriftung der Wurzel und ersetzen sie durch die Beschriftung des letzten Knotens (in der Ebenenordnung) und entfernen diesen. Dann ist die Haldeneigenschaft höchstens an der Wurzel verletzt. Wenn das so ist, tauschen wir die Wurzelbeschriftung mit der Beschriftung des Sohnes, der minmale Beschriftung hat. Dann ist die Haldeneigenschaft höchstens fur diesen Sohn verletzt. Wenn das so ist, tauschen wir...

Implementierung: Wir schreiben Prozeduren $downHeap$ und $upHeap$, die jeweils nur einen Korrekturschitt ausführen und den neuen problematischen Knoten ausgeben oder null, falls keine Verletzung vorlag.

Java: Das Verkleinern und Vergrößern der Halde erfordert Methoden, die dem Bestimmen des Nachfolgers oder Vorgängers bezüglich der Ebenenordnung ähnlich sind. Wir schreiben eine Schnittstelle $PriorityQueue$ für den abstrakten Datentyp und eine Klasse $Heap$, die diese Schnittstelle implementiert und $BinaryTree$ erweiter.

Korrektheit: Um uns über die Korrektheit der obigen Algorithmen Klarheit zu verschaffen, betrachten wir die wesentlichen Schleifen.

Sprechweise: Ist (T, β) ein Binärbaum, so sagen wir, daß die Haldeneigenschaften nur in $u \in T$ verletzt ist, wenn T vollständig ist und (H_v) für alle $v \in T \setminus \{u\}$ gilt.

Sie ist einfach verletzt in u bezüglich v, wenn v ein Sohn von u ist, so daß $\beta(u) > \beta(v)$ gilt, und u keinen weiteren Sohn hat oder für den anderen Sohn keine Verletzung vorliegt, etwa w, $\beta(u) \leq \beta(w)$ gilt.

Sie ist schwach verletzt in u, wenn sie in u verletzt ist und u keinen Vater hat oder aber $\beta(v) \leq \beta(w)$ für den Vater v von u und jeden Sohn w von u gilt.

Lemma: (T, β) ein Binärbaum, der die Haldeneigenschaft nur in u verletzt.

1. Ist die Haldeneigenschaft nur schwach verletzt in u, so gilt nach Ausführung von $w = downHeap(u)$ eine der beiden folgenden Bedingungen:

   - Der Baum ist eine Halde und w ist null

- Die Haldeneigenschaft ist höchstens in w verletzt, und dann nur schwach, und er ist Sohn von u.

2. Ist die Haldeneigenschaft einfach verletzt in u bezüglich v, so gilt nach der Ausführung von $w = upHeap(v)$ eine der beiden folgenden Bedingungen:

   - Der Baum ist eine Halte und w ist *null*
   - Die Haldeneigenschaft ist höchstens im Vater von u verletzt, und dann nur einfach bezüglich u, und $w = u$

   Außerdem ändern sich durch Ausführung von *upHeap* oder *downHeap* die Elemente der Halde nicht.

Beweis: Einfaches Nachrechnen mit vielen Fallunterscheidungen nach Beschriftungen der beteiligten Knoten und der vorhandenen Söhne. Daraus folgt die Korrektheit der Algorithmen (totale Korrektheit). □

Satz: Bei einer Haldenimplementierung einer Prioritätsschlange ist die pessimale Laufzeit der Operationen $\Theta(\log n)$ Außerdem ist der Platzverbrauch $\Theta(n)$.

Folgerung: Das Sortierverfahren *GenericSort* hat pessimale Laufzeit $\Theta(n \cdot \log n)$, wenn die Prioritätsschlange durch eine Halde implementiert wird! Optimal (sortieren in geringerer Zeit als $\Theta(n \log n)$ ist theoretisch nicht möglich, siehe (4.4))!

Laufzeitverhalten:

Feldlänge	10^2	10^3	10^4	10^5	10^6
SortedDoublyLinkedListGenericSort	12	230	32.970	...	...
LinkedHeapGenericSort	14	11	144	2.435	35.516
WithSortingKeyJavaSort	3	32	480	6.347	78.628

3.7 Lexika (dictionaries)

Motivation: Prioritätsschlangen unsterstützen nur zwei Operationen: *add* und *extractMin*. Häufig möchte man mehr: $add(o)$, $remove(k)$, $find(k)$, $getSuccessor(k)$, $getPredecessor(k)$, $enumerateAll()$, $findMin()$, $findMax()$, ... mit offensichtlicher Semantik. In diesem Abschnitt untersuchen wir sogenannte *geordnete Wörterbücher*.

Definition: Ein ADT zur Verwaltung von Objekten mit Sortierschlüsseln, die die Operationen *add*, *remove*, *find*, *findMin*, *findMax*, *getSuccessor*, *getPredecessor* unterstützt, wird *geordnetes Wörterbuch* genannt.

Ziel: Implementierung von geordneten Wörterbüchern, bei der alle Operationen in Zeit $O(\log n)$ durchgeführt werden können.

Einfache Ansätze: Implementierung durch Listen: pessimale Laufzeit $\Theta(n)$, Implementierung durch Halden: pessimale Laufzeit von Finden und Löschen beliebiger Elemente: $\Theta(n)$.

Idee: Wir benutzen wieder binäre Bäume, ordnen die Elemente aber günstiger an als bei einer Halde. Wenn die Bäume dann fast vollständige Bäume sind, sollten alle Operationen in Zeit $O(\log n)$ durchführbar sein.

Definitionen: Für Knoten u und v schreiben wir $u < v$, falls v Nachfahre von u ist. Wir schreiben $u <_l v$ (*linkskleiner*), falls v der linke Sohn oder ein Nachfahre des linken Sohns von u ist. Analog *rechtskleiner*.

3.8 Suchbäume

Definitionen: Ein beschrifteter Binärbaum (T, β) heißt *Suchbaum*, wenn er die folgende Bedingung erfüllt:

(S) Für alle Knoten $u, v \in T$ gilt: Falls $u <_l v$, so ist $\beta(u) > \beta(v)$; falls $u <_r v$, so $\beta(u) < \beta(v)$.

Lemma: Ist (T, β) ein Suchbaum und sind $u, v \in T$, so gilt $u <_{in}^T v$ genau dann, wenn $\beta(u) < \beta(v)$ gilt.

Beweis: Siehe Übung.

Frage: Wie implementiert man die drei Operationen?

- **Finden** ist einfach: Wir suchen von der Wurzel aus die Stelle, an der sich das Element befinden müßte. Dazu vergleichen wir jeweils den gesuchten mit dem aktuellen Schlüssel.
- **Einfügen:** Wir fügen ein entsprechendes Blatt hinzu
- **Löschen:** einfach, wenn das zu löschende Element an einem Knoten mit höchstens einem Sohn steht, sonst ersetzen wir die Beschriftung durch das nchstgrößere Element und löschen dieses.

Die **Korrektheit**[7] folgt in dem Fall aus folgendem Lemma, das die folgende **Schreibweise** nutzt: Ist v ein Knoten eines Baumes T, so steht $T \uparrow v$ für den Baum, der aus T durch entfernen von v und dessen Nachfolgern entsteht. Anolog für beschriftete Bäume.

[7] diese Vorlesung hat Inga geTeXt - dankeschön!

Lemma: Sei (T, β) ein beschrifteter Baum und $u \in T$ vom Grad 2 mit Söhnen v_1 und v_2. Weiterhin seien $V_l = \{v \in T | u <_l v\}$, $V_r = \{v \in T | u <_r v\}$ und $V = T \uparrow u$. Ist (T, β) ein Suchbaum, so gilt:

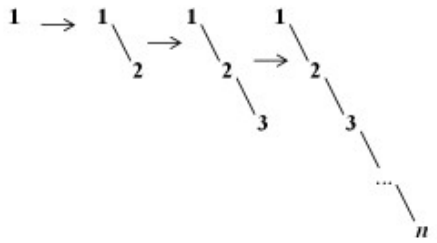
- Für alle $w \in V_l, w' \in V_r$ gilt $\beta(w) < \beta(v) < \beta(w')$
- Für jedes $w \in V$ gilt eine der beiden folgenden Bedingungen:
  - Für jedes $w' \in V_l \cup V_r \cup \{u\}$ gilt $\beta(w) < \beta(w')$
  - Für jedes $w' \in V_l \cup V_r \cup \{u\}$ gilt $\beta(w') < \beta(w)$
- Ist $(T, \beta) \downarrow v_r$ ein Suchbaum und auch $(T, \beta) \uparrow v_r$ und gelten die obigen Bedingungen, so ist (T, β) ein Suchbaum.

Beweis: in den Übungen!

Laufzeitanalyse: Offensichtlich ist die Laufzeit aller Operationen $\Theta(h(T))$.

Frage: Wir wissen: $h(T) \in \Theta(log(n))$ für vollständige Bäume. Gilt das auch für Suchbäume, die durch obige Operationen entstehen? **Antwort**: Leider nicht!

Beispiel: Einfügen von 1,2,3,...n



DIe Höhe des letzten Baumes ist n und die Laufzeit ist $\Theta\left(\sum_{i=0}^{n} i^2\right) = \Theta(n^2)$

Satz: Für jedes n gibt es n Wörterbuchoperationen, deren Hintereinanderausführung bei der einfachen Implementierung durch Suchbäume die Laufzeit $\Theta(n^2)$ hat.

Folgerung: Weder die pessimale noch die ammortisierte Laufzeit der Wörtebuchperationen ist $O(log(n))$ bei der einfachen Implementierung.

JAVA: Ähnlich zur Implementierung von Halden als Erweiterung der Klasse *BinaryTree*.

3.8.1 Umkrempeln (splaying)

Frage: Wie schaffen wir es, eine pessimale Laufzeit von $\Theta(log(n))$ zu erreichen?

Antwort: Wir sorgen einfach dafür, dass die entstehenden Suchbäume höhenbalanciert sind, d.h. $h(T) \in O(logS(t))$.

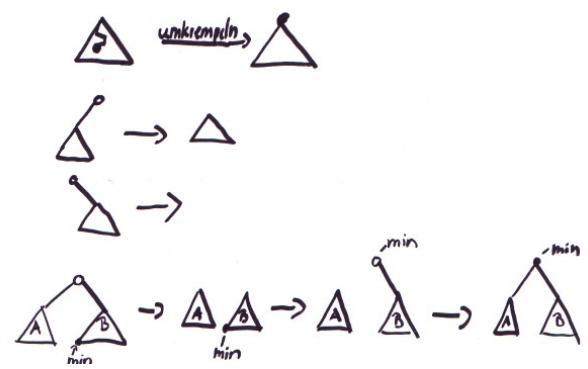
Weniger ambitioniertes Ziel: Amortisierte Laufzeit $log(n)$. Das heißt, wir erlauben, daß einzelne Operationen im schlechtesten Fall mehr als logarithmische Laufzeit haben, wenn vorher entsprechend Zeit gespart wurde.

Vorüberlegung: Wenn ein Knoten tief im Baum hängt und auf ihn durch eine $find$-Operation zugegriffen wird, muß der Baum umstrukturiert werden, damit Wiederholungen nicht zu zulangen Laufzeiten führen.

Definition: Die *Tiefe* eines Knotens $v = (i_0, ..., i_{l-1})$ ist $d(v) = l$

Einfache Idee: Wird auf einen Knoten durch $find$ zugegriffen, dann wird der Baum so umstrukturiert, daß die Beschriftung des Knotens nachher an der Wurzel steht. Beim Hinzufügen verfahren wir analog: wird tatsächlich eingefügt, wird der eingefügte Knoten an die Wurzel gebracht, sonst der schon Vorhandene.

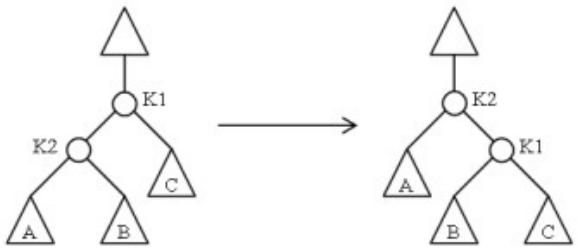
Beim Löschen: Zuerst wird der Knoten gefunden und an die Wurzel gebracht. Dann wird die Wurzel entfernt und der Baum zerfällt in zwei Teile. Danach wird das Minimum des rechten Teils an die Wurzel gebracht und der linke Teil dort angehängt.



Was ist zu tun?

1. die richtigen Umstrukturierungen finden
2. die Analyse vornehmen

Einfachstes Mittel zur Umstrukturierung: Rotation



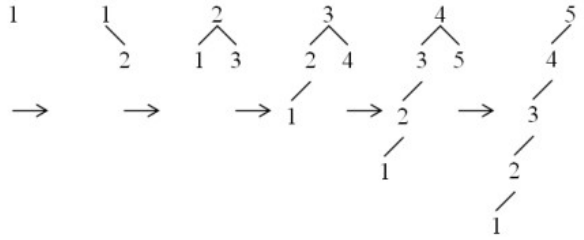
Beispiel:

Definition: Sei (T, β) ein beschrifteter Baum, $u = (i_0, ..., i_{l-1}) \in T$ und $v = (i_0, ..., i_l) \in T$ Dann entsteht (T', β') aus (T, β) durch *Rotation*, wenn folgendes gilt (dabei $v_{a,b} = (i_o, ..., i_{l-1}, a, b)$):

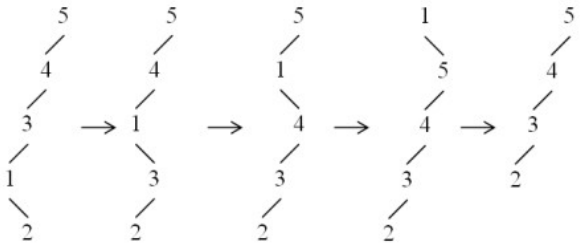
1. Fall: $i_l = 0$. Es gehören u und $v' = (i_0, ..., i_{l-1}, 1)(= v_1)$ zu T' und es gilt:
   - $(T, \beta) \uparrow u = (T', \beta') \uparrow u$
   - $(T, \beta) \downarrow v_{0,0} = (T', \beta') \downarrow v_0$
   - $(T, \beta) \downarrow v_{0,1} = (T', \beta') \downarrow v_{1,0}$
   - $(T, \beta) \downarrow v_1 = (T', \beta') \downarrow v_{1,1}$
   - $\beta(u) = \beta'(v')$ und $\beta(v) = \beta'(u)$
2. Fall: $i_l = 1$ symmetrisch

Lemma: Sind T, T', β, β' wie eben und ist (T, β) ein Suchbaum so auch (T', β').

Altes Beispiel: Aufbauen:



Löschen:

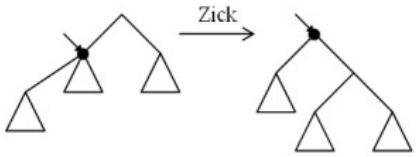


Bemerkung: Die obige einfache Art der Umstrukturierung hat weder pessimale noch ammortisierte Laufzeit $\Theta(n)$

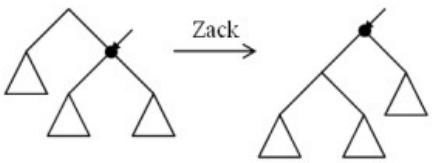
Lösung: Wir ziehen auch noch den Großvater mit in Betracht. Wir rotieren in gewissen Fällen dreimal statt zweimal.

3.8.2 ZZ-Umkrempeln

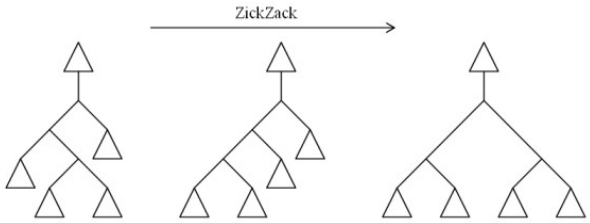
Zick:



Zack:

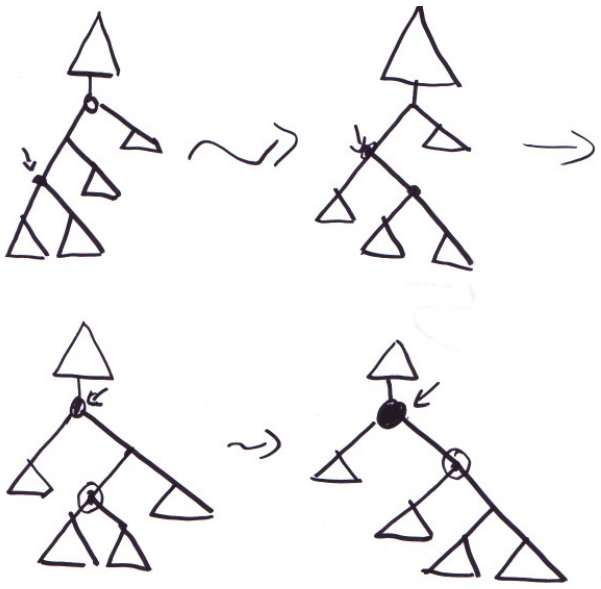


ZickZack:



ZackZick: symmetrisch zu ZickZack

ZickZick:



ZackZack: symmetrisch zu ZickZick

Sprechweise: Wenn wir den Knoten v in (T, β) durch diese Regeln an die Wurzel bringen und das Ergebnis (T', β') heißt, sagen wir, daß (T, β) bei v zu (T', β') zz-umgekrempelt wurde.

3.8.3 Amortisierte Analyse des ZZ-Umkrempelns

Ziel: Wir wollen zeigen, daß die amortisierte Laufzeit der geordneten-Wörderbuch-Operation $O(\log n)$ ist, wenn n die jeweilige Anzahl der gespeicherten Objekte ist.

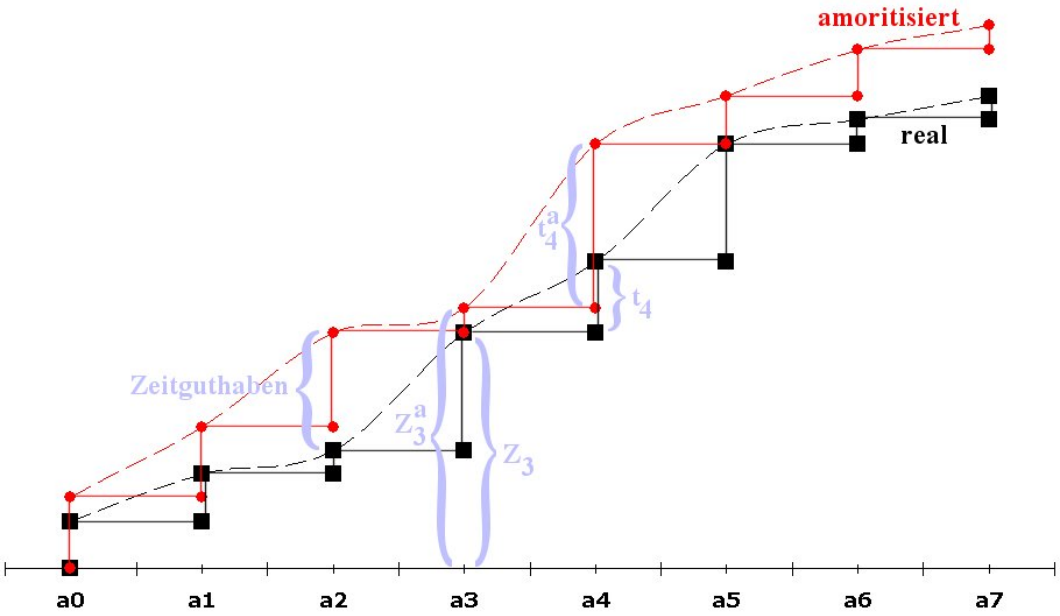
Genauer: Wir wollen zeigen, daß es eine Konstante c gibt, so daß für jede Folge $a_0, a_1, a_2, ...$ von Wörterbuchoperationen folgendes zutrifft:

> *Wenn diese Operationen nacheinander auf ein leeres Wörterbuch angewendet werden und die Zwischenergebnisse die Größe $n_0, n_1, ...$ haben*

und die tatsächlichen Laufzeiten $t_0, t_1, \ldots$ sind, so gilt für jedes m (und $\operatorname{Log} n := \max\{1, \log n\}$*):*

$$\sum_{i=0}^{m-1} t_i \leq \sum_{i=0}^{m-1} c \cdot \operatorname{Log} n_i$$

Notation: Die linke Seite bezeichnen wir mit $Z_m := \sum_{i=0}^{m-1} t_i$ und die rechte Seite mit $Z_m^a := \sum_{i=0}^{m-1} c \cdot \operatorname{Log} n_i$.



Ansatz für Beweis: Wir versuchen, zu jedem Zeitpunkt eine möglichst genaue Schätzung des Betrages $Z_m^a - Z_m$ zu haben. Dieser kann interpretiert werden als *Zeitguthaben*, denn die Operation a_m kann die Zeit $Z_m^a - Z_m + c \cdot \operatorname{Log} n_m$ verwenden, ohne daß $Z_{m+1} \leq Z_{m+1}^a$ verletzt wird.

Genauer: Sei B die Menge aller (unbeschrifteter) Binärbäume. Wir werden eine Funktion $\Phi : B \to \mathbb{R}_{\geq 0}$ bestimmen, die die folgende Ungleichung erfüllt, wenn T_m der unbeschriftete Baum des m-ten Zwischenergebnisses ist:

$$Z_m^a - Z_m \geq \Phi(T_m)$$

Dann ist wegen $\Phi(T_m) \geq 0$ sofort $Z_m \leq Z_m^a$ gegeben. Wir wären fertig! Die obige Ungleichung werden wir per Induktion beweisen:

- Der Induktionsanfang ist einfach, wenn wir $\Phi(\emptyset) = 0$ gilt. Dann gilt nämlich: $Z_0^a - Z_0 = 0 - 0 = 0 \geq 0 = \Phi(\emptyset)$

- Die Induktionsannahme ist $Z_m^a - Z_m \geq \Phi(T_m)$.
- Im Induktionsschritt müssen wir dann zeigen, daß

$$Z_{m+1}^a - Z_{m+1} \geq \Phi(T_{m+1})$$

  Es reicht zu zeigen: $t_m^a - t_m \geq \Phi(T_{m+1}) - \Phi(T_m)$

Beobachtung: Angenommen, es gibt eine Funktion $\Phi : B \to \mathbb{R}_{\geq 0}$, die folgende Bedingung erfüllt:

(P_0) $\Phi(\emptyset) = 0$

(P_n) Ist a eine Wörterbuchoperation, (T, β) ein Suchbaum, (T', β') das Ergebnis der Anwendung von a auf (T, β) und t die tatsächliche Laufzeit von a und t^a der Ansatz für die amortisierte Laufzeit, so gilt

$$t^a - t \geq \Phi(T') - \Phi(T)$$

Dann ist der Ansatz für die amortisierten Laufzeiten der Wörterbuchoperationen richtig! Die Funktion Φ wird Potenzialfunktion genannt.

Schwierig: Finden der Potenzialfunktion. Wir werden es mit folgender Funktion versuchen:

$$\Phi(T) = c' \cdot \Psi(T) \text{ mit } \Psi(T) = \sum_{v \in T} R_T(v) \text{ mit } R_T(v) = \log s(T \downarrow v)$$

für geeignetes c'. Dabei ist $R_T(v)$ der *Rang* von v in T.

Lemma: Ist n ein Knoten eines Suchbaume (T, β) und (T', β') der Baum, der durch ZZ-Umkrempeln von (T, β) bei u entsteht, dann gilt:

$$\Psi(T') - \Psi(T) \leq 3(\log s(T') - R_T(u)) - d(u)$$

wobei $d(u)$ die Tiefe von u ist.

Beweis[8]: per Induktion über $d(u)$. Wir gehen aber nicht alle möglichen Fälle durch, sondern exemplarisch nur einen. Außerdem benutzen wir die triviale Ungleichung[9]

$$\log a + \log b \leq 2 \cdot \log(a + b) - 2$$

[8] „Damit ich sie nicht langeweile mit tausenden Fällen, machen wir nur interessante Fälle. Genaugenommen nur einen."

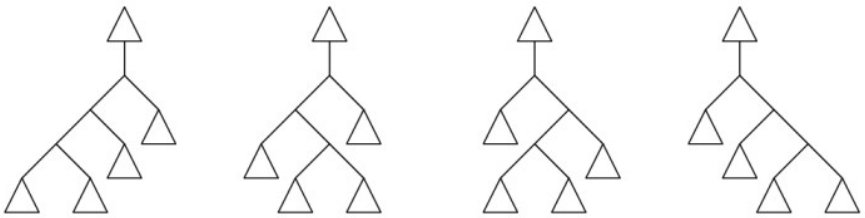
[9] Beweis: $(a - b)^2 \geq 0$, also $(a + b)^2 \geq 4ab$, d.h. $2\log(a + b) \geq \log a + \log b + 2$

- Induktionsanfang: Für $d(u) \leq 1$: Bei $d(u) = 0$ wird der Baum nicht verändert, also

$$\Psi(T') - \Psi(T) = 0 = 3(\log s(T') - \underbrace{R_T(u)}_{=\log s(T)}) - d(u)$$

  Den Fall $d(u) = 1$ sparen wir uns.

- Induktionsschritt: Für $d(u) > 1$. Dann kommt in ersten Schritt de Umkrempelns eine Großvaterregel zum Einsatz. Hier wird nur die ZickZick-Regel betrachtet:



  Nach Bild gilt offensichtlich:

$$R_T(w) = R_{T''}(u'')$$
$$s(T \downarrow u) + s(T'' \downarrow w'') \leq s(T'' \downarrow u'')$$

  Aus dem Lemma über den Logaritmus folgt:

$$R_T(u) + R_{T''}(w'') \leq 2R_{T''}(u'') - 2$$

  Zudem ist

$$R_{T''}(v'') \leq R_{T''}(u'') \text{ und } R_T(v) \geq R_T(u)$$

  Die Induktionsvoraussetzung liefert:

$$\Psi(T') - \Psi(T'') \leq 3(\log s(T') - R_{T''}(u'')) - d(u'')$$

  Also gilt:

$$\begin{aligned}
\Psi(T'') - \Psi(T) &= (R_{T''}(u'') + R_{T''}(v'') + R_{T''}(w'')) - (R_T(u) + R_T(v) + R_T(w)) \\
&= R_{T''}(v'') + R_{T''}(w'') - R_T(u) - R_T(v) \\
&\leq R_{T''}(v'') + 2R_{T''}(u'') - 2 - 2R_T(u) - R_T(v) \\
&\leq 3R_{T''}(u'') - 3R_T(u) - 2 \\
&= 3(R_T(u'') - R_T(u)) - 2 \\
\Psi(T') - \Psi(T) &= (\Psi(T'') - \Psi(T)) + (\Psi(T') - \Psi(T'')) \\
&\leq 3(R_{T''}(u'') - R_T(u)) - 2 + 3(\log s(T') - R_{T''}(u'')) - d(u'') \\
&= 3(\log s(T') - R_T(u)) - (d(u'') + 2) \\
&= 3(\log s(T') - R_T(u)) - d(u)
\end{aligned}$$

Satz: Wir betrachten die Wörterbuch-Implementierung durch Spreizbäume. Dann gilt:

1. Die Wörterbuchoperationen haben amortisierte Laufzeit $\Theta(\log n)$.
2. Das Wörterbuch hat Speicherbedarf $\Theta(n)$.
3. Bei der Ausführung der Wörterbuchoperationen wird ein zusätzlicher Speicherplatz von $\Theta(1)$ benötigt.

Beweis: Punkte 2 und 3 sind klar, zur Behauptung 1: Wir zeigen, daß (P_0) und (P_n) für geeignete Laufzeit $c \cdot \log s(T)$ und $\Phi(T) = c' \cdot \Psi(T)$ ansetzen. Der Einfachheit halber zählen wir eine Rotation als einen Zeitschritt. Ebenso zählen wir einen Wanderschritt in einem Baum als einen Zeitschritt. Außerdem betrachten wir nur $find$ und add.

1. $find$: Falls $T = \emptyset$, so ist die Behauptung leicht einzusehen. Sei u der Knoten, an dem umgekrempelt wird. Dann kostet das Finden Zeit $d(u) + 1$ und das Umkrempeln $1,5 \cdot d(u)$. Insgesamt: $t \leq 2,5d(u) + 1$. Damit ergibt sich:

$$\begin{aligned} t^a - t &= c \cdot \mathrm{Log}\, s(t) - 2,5d(u) - 1 \\ &\geq c' \cdot (3 \cdot \mathrm{Log}\, s(T') - d(u)) \\ &\geq \Phi(T') - \Phi(T) \\ &= c' \cdot ... - c' \cdot ... \end{aligned}$$

   D.h. wir müssen $c' \geq 3,5$ und $c \geq 3c'$ wählen

2. add: Falls das einzufügende Element schon im Baum ist, hat add den gleichen Aufwand wie $find$. In den anderen Fällen sei T'' der Baum, der nach der eigentlichen Einfügeoperation (und damit vor dem Umkrempeln) entsteht und es sei u der Knoten, der eingefügt wurde. Dann ist die tatsächliche Laufzeit $\leq 2,5d(u) + 2$ (siehe oben).

   Wir schätzen zuerst die Differenz zwischen $\Psi(T)$ und $\Psi(T'')$ ab. Sei V

die Menge der Vorfahren von U, dann gilt[10]:

$$\begin{aligned}
\Psi(T'') - \Psi(T) &= \sum_{v \in V} (\mathrm{Log}(s(T \downarrow v) + 1) - \mathrm{Log}\, s(T \downarrow v)) \\
&\leq \sum_{i=1}^{d(u)} (\mathrm{Log}(i+1) - \mathrm{Log}(i)) \\
&= \mathrm{Log}(d(u)+1) - \mathrm{Log}(d(u)) + \mathrm{Log}(d(u)) - ... - \mathrm{Log}(1) \\
&= \mathrm{Log}(d(u)+1) \\
t^a - t &\geq c \cdot \mathrm{Log}\, s(T) - 2,5d(u) - 2 \\
&\geq c' \cdot \mathrm{Log}(d(u)+1) + c' \cdot (3 \cdot \mathrm{Log}\, s(T') - d(u)) \\
&\geq c' \,\mathrm{Log}(d(u)+1) + \Phi(T') - \Phi(T'') \\
&\geq \Phi(T') - \Phi(T)
\end{aligned}$$

sofern $c \geq 5c'$ und $c' \geq 4,5$ gewählt wird.

3. Ähnliche Bedingungen erhält man für *remove*. Da die Bedingungen leicht erfüllbar sind, folgt die Behauptung. □

Bemerkung: Es gibt keine Implementierung, deren amortisierte Laufzeit besser ist. Bei anderen Bäumen (Rot-Schwarz-Bäume und AVL-Bäume) sind z.T. noch die pessimalen Laufzeiten im Bereich $\log(n)$.

[10] Die Ψ-Funktion verändert sich nur auf genau dem Pfad, der zum eingefügten Knoten u führt, dort kommt genau 1 hinzu.

4 Sortieralgorithmen (sorting)

4.1 Klassifizierung von Sortierproblemen/-Algorithmen

Art der zu sortierenden Objekte: *Vergleichsbasierte* Sortierverfahren haben als einzige Annahme, daß Objekte miteinander verglichen werden können (*compareTo*). Im Gegensatz dazu haben wir bei *CountingSort* ausgenutzt, daß die Sortierschlüssel Bytes waren.

Zwischenspeicher für zu sortieren Objekte: *In-Place-Verfahren* sortieren Felder allein durch Vergleich und Vertauschen von Feldelementen. Feldelemente dürfen nicht zwischengespeichert werden (haben wir noch nicht gesehen).

Struktur der Eingabe- und Ausgabedaten: Bei *externen Verfahren* sind die Eingabedaten in Dateien oder Strömen abgelegt und so umfangreich, daß Felder nicht benutzt werden können, um alle Elemente zwischenzuspeichern.

Ordnung von Elementen mit **gleichem Sortierschlüssel**: Ein Sortierverfahren heißt *stabil* (Beispiel Sortierung nach mehreren Schlüsseln), wenn die Reihenfolge von Objekten mit gleichem Sortierschlüssel durch das Verfahren nicht verändert wird. Z.B. sind *CountingSort* und *BucketSort* stabil (aber *BucketSort* mit Kellern anstelle von Schlangen nicht), genauso das generische Sortierverfahren mit geeigneter einfacher Listenimplementierung oder geeigneter sortierter Listenimplementierung, nicht aber mit der Haldenimplementierung der Prioritätsschlange (Beispielsweise wird $1\,2_0\,2_1$ sortiert zu $1\,2_1\,2_0$).

4.2 *InsertionSort* - Sortieren durch Einfügen

Erinnerung: Das generische Sortierverfahren mit Prioritätsschlangenimplementierung durch sortierte Liste: Es wird Schritt für Schritt eine sortierte Liste aufgebaut, die schließlich alle Elemente enthält. Danach wird sie wieder abgebaut und ins Feld zurückgeschrieben.

Verbesserung: Wir führen dies direkt im Feld (in-place) durch und sorgen dafür, daß nach der i-ten Runde die ersten $i+1$ Feldelemente sortiert sind und der Rest unverändert. In Runde $i+1$ wird dann das $(i+2)$-te Element an der richtigen Stelle durch sukzessives Vertauschen eingefügt.

Notation: Der Einfachheit halber schreiben wir $a[i] < a[j]$ anstelle von $a[i].key < a[j].key$.

Genaue Laufzeitanalyse: Wir zählen die Anzahl der durchgeführen Vergleiche. Die Laufzeit ist nämlich größenordnungsmäßig äquivalent dazu.

Konvention: Wir bezeichnen mit $\overline{a}$ immer das Ausgangsfeld.

Runden:

1. Runde: ein Vergleich

2. Runde: ein Vergleich, wenn $\overline{a}[2]$ größer gleich $\overline{a}[0]$ und $\overline{a}[1]$ ist, sonst zwei Vergleiche

3. Runde: ein Verlgeich, wenn $\overline{a}[3]$ größer gleich $\overline{a}[0]$, $\overline{a}[1]$ und $\overline{a}[2]$ ist; zwei Vergleiche, wenn $\overline{a}[3]$ das zweigrößte Element ist etc.

4. Runde: ...

i. Runde: Anzahl der Vergleiche ist $|\{\, j < i \mid \overline{a}[j] > \overline{a}[i]\}| + \delta(\overline{a}, i)$ mit $\delta(\overline{a}, i) := 1$, falls $j < i$ existiert mit $\overline{a}[j] \leq \overline{a}[i]$, sonst $\delta(\overline{a}, i) = 0$.

Definition: Sei $a[0\,..\,n-1]$ ein gegebenes Feld. Eine *Inversion* (Fehlstellung) ist ein Paar (i, j) mit $i < j$ und $a[j] < a[i]$. Die Anzahl der Inversionen eine Feldes wird mit $I(a)$ bezeichnet.

Lemma: Die Anzahl der Vergleiche, die $InsertionSort$ auf einem Feld $a[0\,..\,n-1]$ durchführt, ist $\Theta(I(a) + n)$.

Beweis: Die Anzahl der Vergleiche ist gegeben durch

$$\begin{aligned} & \sum_{i=0}^{n-1} (|\{\, j < i \mid \overline{a}[j] > \overline{a}[i]\}| + \delta(\overline{a}, i)) \\ = \;& I(a) + \underbrace{|\{\, i \mid \exists\, j < i\ (\overline{a}[j] \leq \overline{a}[i])\}|}_{=:m} \end{aligned}$$

Es folgt sofort, daß $m \leq n$ gilt, also ist die Anzahl der Vergleiche $\leq I(a) + n$. Andererseits gilt offensichtlich $I(a) + m \geq n - 1$, also $\square$

$$3(I(a) + m) \geq I(a) + n \;\text{ für }\; n > 1 \Rightarrow \frac{1}{3}(I(a) + n) \leq I(a) + m \;\text{ für }\; n > 1$$

Folgerung: $InsertionSort$ ist ein stabiler in-place-Sortieralgorithmus mit pessimaler Laufzeit $\Theta(n^2)$.

Beweis: Für das Feld $a = [n-1, ..., 0]$ ist $I(a)$ maximal und $I(a) \approx 1 + 2 + ... + (n-1) \in \Theta(n^2)$. $\square$

4.2.1 Partielle Korrektheit - Schleifeninvarianten

Ziel: Systematischer Zugang zu Korrektheitsbeweisen am Beispiel von $InsertionSort$.

Vorüberlegung: Um genau sagen zu können, was es bedeutet, daß $InsertionSort$

korrekt ist, müssen wir sagen, welche Zielsetzung $InsertionSort$ hat. Darum: Eine Belegung eines Feldes $a[0\,..\,n-1]$ fassen wir auf als Funktion mit Definitionsbereich $\{0,...,n-1\}$ auf. Die Anfangsbelegung bezeichnen wir mit $\overline{a}$. Dann ist die *Spezifikation* für $InsertionSort$:

- Vor der Ausführung von $InsertionSort$ gilt Φ_{pre}: Der Wertebereich von $\overline{a}$ besteht aus miteinander vergleichbaren Objekten.
- Nach der Ausführung gilt Φ_{post}:
  - Es gibt eine Permutation $\pi : \{0,...,n-1\} \to \{0,...,n-1\}$, so daß $\overline{a} = a \circ \pi$.
  - Für alle $i, j < n$ mit $i < j$ gilt: $a[i] \leq a[j]$. Wir sagen, a ist geordnet.

Sprechweise: Wir sprechen von *Vorbedingung* (*precondition*) und *Nachbedingung* (*postcondition*).

Definition: Ein Programm(stück) P ist *partiell korrekt* bezüglich gegebener Vor- und Nachbedingungen Φ bzw. Ψ, wenn für jede terminierende Ausführung des Programm(stück)s gilt: Ist zu Beginn der Ausführung Φ erfüllt, so ist nach ihrer Beendigung Ψ erfüllt, in Zeichen: $\vdash \{\Phi\}\, P\, \{\Psi\}$

Definition: Ein Programm(stück) P ist *total korrekt*, wenn zusätzlich gilt: Jede Ausführung, zu deren Beginn Φ erfüllt ist, terminiert.

Frage: Wie gehen wir vor, um die Korrektheit von $InsertionSort$ nachzuweisen?

Antwort: Wir nutzen aus, daß unser Programm aus kleineren Programmstücken modular zusammengesetzt ist: Für jedes Programmstück überlegen wir uns geeignete Vor- und Nachbedingungen, wir gehen dabei von außen nach innen vor: Wir überlegen uns, unter welchen Annahmen für den Rumpf die äußere Schleife korrekt ist und überprüfen später, ob die Annahmen zutreffen. Um diese zu überprüfen, machen wir weiter Annahmen und überprüfen diese noch später usw.

Erster Schritt: In der äußeren Schleife vergrößern wir den schon sortierten Bereich schrittweise. Deshalb betrachten wir folgende Bedingung $\Theta(i)$, die eine Momentaufnahme ist:

- Es gibt eine Permutation $\pi : \{0,...,i-1\} \to \{0,...,i-1\}$, so daß $\overline{a}|_{\{0,...,i-1\}} = a|_{\{0,...,i-1\}} \circ \pi$
- Für alle j mit $i < j < n$ gilt $\overline{a}[j] = a[j]$.

Wenn mit Q der Schleifenrumpf bezeichnet wird, gilt folgendes:

- Aus Φ_{pre} und $n > 1$ folgt: $\Theta(1)$, in Zeichen: $n > 1 \wedge \Phi_{pre} \models \Theta(1)$.
- $\models \{1 \leq i \leq n-1 \wedge \Theta(i)\}\, Q\, \{\Theta(i+1)\}$
- $n > 1 \wedge \Theta(n) \models \Phi_{post}$
- $n \leq 1 \wedge \Phi_{pre} \models \Phi_{post}$

Unter diesen Umständen sind for-Schleifen korrekt nach folgendem Lemma.
Lemma (for-Regel): Sei P ein Programmstück der Form

```
for i = l bis r
  Q
```

und Φ, Ψ und $\Theta(i)$ Bedingungen sind für die gilt:

- $L \leq r \wedge \Phi \models \Theta(1)$
- $\models \{l \leq i \leq r \wedge \Theta(i)\}\, Q\, \{\Theta(i+1)\}$
- $\models (l \leq r \wedge \Theta(r+1)) \vee (l > r \wedge \Theta) \models \Psi$

gilt, so gilt auch $\models \{\Phi\}\, P\, \{\Psi\}$. In Q dürfen dabei die Variablen i und die Variablen l und r nicht veränder werden.

Beweis: Ist nicht schwierig, aber setzt formale Semantik von Pseudocode voraus, daher im 4. Semester.

Zweiter Schritt: Nun müssen wir uns die Korrektheit des Schleifenrumpfes der for-Schleife. Passendes Lemma:

Lemma (Kompositionsregel): Falls $\models \{\Phi\}\, Q\, \{\Delta\}$ und $\models \{\Delta\}\, R\, \{\Psi\}$, so gilt auch $\models \{\Phi\} \binom{Q}{R} \{\Psi\}$. Das motiviert folgende Teilbehauptungen:

Wir versuchen zu zeigen, daß:

$(**)$ $\models \{1 \leq i < n-1 \wedge \Theta(i)\}\, j = i\, \{1 \leq i \leq n-1 \wedge \Theta(i) \wedge i = j\}$

$(***)$ $\models \{1 \leq i \leq n-1 \wedge \Theta(i) \wedge i = j\}\, P'\, \{\Theta(i+1)\}$ (wobei P' innere Schleife ist)

Denn dann wäre das Programm nach Lemma korrekt.

Dritter Schritt: Der erste Teil der obigen Behauptung ist total (sic!) einleuchtend, folgt aber auch aus den folgenden Lemma:

Lemma (Zuweisungsregel): Sei x eine Variable und E ein Ausdruck und $\Theta(x)$ ein Bedingung. Dann gilt: $\models \{\Phi(E)\}\, x = E\, \{\Phi(x)\}$

Lemma (Konsequenzregel): Sei P ein Programmstück, für das gilt: $\models \{\Phi\}\, P\, \{\Psi\}$, dann gilt:

- Falls $\Phi_0 \vdash \Phi$, so gilt $\vdash \{\Phi_0\}\, P\, \{\Psi\}$
- Falls $\Psi \vdash \Psi'$, so gilt $\vdash \{\Phi\}\, P\, \{\Psi'\}$

Nun können wir $(**)$ beweisen[11]: Nach Zuweisungsregel gilt:

$$\vdash \{1 \leq i \leq n-1 \wedge \Theta(i) \wedge i = i\}\, j = i\, \{1 \leq i \leq n-1 \wedge \Theta(i) \wedge i = j\}$$

Nach Konsequenzregel gilt dann aber $(**)$. **Vierter Schritt:** $(***)$ beweisen wir unter Zuhilfenahme der *while*-Regel.

Lemma (*while*-Regel): Sei P ein Programm der Form

```
solange B
  Q
```

und Θ eine Bedingung ist, für die $\vdash \{\Theta \wedge B\}\, Q\, \{\Theta\}$ gilt, dann gilt:

$$\vdash \{\Theta\}\, P\, \{\Theta \wedge \neg B\}$$

Hierbei ist Θ eine Bedingung, die durch Anwendung von Q nicht verändert wird und wird deshalb (Schleifen-)Invariante genannt. Bei der for-Schleife nennen wir $\Theta(i)$ auch (Schleifen-)Invariante, auch wenn dies nicht ganz korrekt ist.

Um $(***)$ zu zeigen, wählen wir Θ' wie folgt:

- $0 \leq j \leq i < n$
- Es gibt eine Permutation $\pi : \{0, ..., i\} \to \{0, ---, i\}$, so daß $\overline{a}|_{\{0,...,i\}} = a|_{\{0,...,i\}} \circ \pi$
- Es gilt für alle k mit $i < k < n$: $\overline{a}[k])a[k]$
- $a|_{\{j,...,i\}}$ und $a|_{0,...,j-1,j+1,...,i}$ ist geordnet.

Dann folgt aus obigem Lemma: $\vdash \{\Theta'\}\, Q'\, \{\Theta' \wedge \neg B\}$, sofern wir $(****)$ $\{\Theta' \wedge B\} Q' \{\Theta'\}$ nachweisen können. Außerdem folgt mit der Konsequenzregel auch

$$\vdash \{1 \leq i \leq n-1 \wedge \Theta(i) \wedge i = j\}\, P'\, \{\Theta(i+1)\}$$

Letzer Schritt: Wir müssen noch $(****)$ nachweisen. Dazu bräuchten wir eine *swap*-Regel. Die gibt's, ist aber aufwändig. Deshalb schenken wir uns das.

Der **komplette Beweis** ist noch mal in einer neuen Version (bottom-up) auf

[11] „Das sieht auf den ersten Blick etwas witzig aus..."

der Homepage von Prof. Wilke verfügbar.

Fazit: „Formale“ Korrektheitsbeweise sind möglich, aber aufwändig. Von zentraler Bedeutung sind die Schleifeninvarianten!

Definition: Ein Programm(stück) heißt *wohlkommentiert*, wenn Vor- und Nachbedingungen für das gesamte Programm(stück) und für jede Schleife eine geeignete Invariante angegeben sind und alle Übergänge schlüssig sind.

4.3 *HeapSort* - Sortieren mit Halden

Erinnerung: *GenericSort* mit einer durch eine Halde implementierte Prioritätsschlange ist ein sehr schnelles ($\Theta(n \cdot \log n)$) Sortierverfahren, aber nicht in-place.

Ziel: Umwandlung in ein in-place-Verfahren.

Problem: Kodierung eines vollständigen beschrifteten Binärbaumes in einem Feld.

Idee: Wir schreiben die Beschriftungen einfach in Ebenenordnung in das Feld.

Formalisierung: Sei $a[0\,..\,n-1]$ ein Feld und

$$\alpha : (i_0, ..., i_{l-1}) \mapsto \sum_{j=0}^{l-1} 2^{(l-1)-j}(i_j + 1)$$

die bijektive Abbildung, die jedem Knoten des unendlichen Binärbaumes eine natürliche Zahl zuordnet. Der zu a gehörige Baum (T, β) besteht aus dem vollständigen unbeschrifteten Binärbaum der Größe n und seine Beschriftung β ist gegeben durch $\beta(v) = a[\alpha(v)]$.

Bemerkung: Ist $a[0\,..\,n-1]$ ein Feld und (T, β) der zugehörige Baum, dann gilt für jeden Knoten $v \in T$:

- v hat einen linken Sohn genau dann, wenn $2\alpha(v) + 1 < n$, dann ist $\alpha^{-1}(2\alpha(v) + 1)$ dieser Sohn
- v hat einen rechten Sohn genau dann, wenn $2\alpha(v) + 2 < n$, dann ist $\alpha^{-1}(2\alpha(v) + 2)$ dieser Sohn
- v hat einen Vater genau dann, wenn $\alpha(v) > 0$, dann ist $\alpha^{-1}\left\lfloor \frac{\alpha(v)-1}{2} \right\rfloor$ dieser Vater

Einfache **Umsetzung** des generischen Sortierverfahrens in ein in-place-Verfahren:

1. Wir bauen die Halde direkt in Feld auf: Wir vergrößern den Anfangsbereich, in dem die Halden stehen soll, und verkleinern den Restbereich schrittweise.

2. Wir bauen die Halde durch wiederholtes Extrahieren des Minimums ab. Dabei entsteht Platz am Feldende. Dort schreiben wir die extrahierten Elemente hin.

Problem: Dann ist das Feld in umgekehrter Reihenfolge sortiert.

Lösung: Wir benutzen eine Variante der Halde, bei denen die Beschriftung des Vaters eines Knotens größergleich der Beschriftung des Knotens sein muß.

Implementierung:

- Wir müssen *add* und *extractMax* für verschieden große Halden im selben Feld implementieren. Dazu benutzen wir eine Variable *lastNode*, die auf das Ende der aktuellen Halde (bzw. des aktuellen Binärbaumes) zeigt.

- Wir ersetzen Container durch Indizes.

- Wir ersetzen *extractRoot* durch *adjustDown*, das eine Verletzung der Haldeneigenschaft an der Wurzel eleminiert.

- Wir ersetzen *add* durch *adjustUp* entsprechend

Beispiel:

```
---   Start   ---
0 | 4   2   5   3
0   4 | 2   5   3
4   0   2 | 5   3
4   0   2   5 | 3
4   5   2   0 | 3
5   4   2   0   3
---   Halde   ---
3   4   2   0 | 5
4   3   2   0 | 5
0   3   2 | 4   5
3   0   2 | 4   5
2   0 | 3   4   5
0 | 2   3   4   5
0   2   3   4   5
--- sortiert! ---
```

Laufzeitanalyse: Wir wissen schon, daß die pessimale Laufzeit $\Theta(n \log n)$ ist. Sie ist auch $\Omega(n \log n)$ aus folgendem Grund: Für $n = 2^k - 1$ ist die pessimale Laufzeit mindestens

$$\sum_{i=0}^{k-1} i \cdot 2^i \geq (k-1) \cdot 2^{k-1} \geq \frac{1}{4} n \log n$$

Laufzeitverbesserung: Den Haldenaufbau kann man effizienter gestalten, wenn man von vornherein das ganze Feld als einen Baum betrachtet. Wir modifizieren das Verfahren wie folgt: Wir „heilen“ die Verletzungen von unten nach oben.

Konkret: Wir modifizieren *adjustDown*: Der Algorithmus erhält einen Feldindex i als Parameter, setzt voraus, daß die Haldeneigenschaft von dem entsprechenden Teilbaum $(T, \beta) \downarrow \alpha^{-1}(i)$ nur an der Wurzel verletzt ist und beseitigt diese wie gewohnt.

```
---   Start   ---
0   5   4   2   3
0   5   4   2 | 3
0   5   4 | 2   3
0   5 | 4   2   3
0 | 5   4   2   3
0   5   4   2   3
5   3   4   2   0
---   Halde   ---
```

Lemma: Der Algorithmus *buildHeap* hat eine pessimale Laufzeit von $\Theta(n)$.

Beweis: $\Omega(n)$ ist klar. Die Laufzeit ist beschränkt durch[12]

$$c \cdot \left(\sum_{i=0}^{k} 2^i (k+1-i) \right) = \Psi(U_n)$$

Nach Übungsaufgabe ist das aber $O(n)$. □

4.4 Eine untere Schranke für vergleichsbasiertes Sortieren

Frage: Wieviele Vergleiche sind eigentlich erforderlich, um ein Feld zu sortieren, wenn auf Feldelemente nur vergleichend und über Vertauschungen

[12] wobei U_n der vollständige Binärbaum mit n Knoten ist

zugegriffen werden kann?

Problem: Wir müssen eine Aussage über alle treffen!

Ansatz: Wir betrachten ein beliebiges Sortierprogramm P und eine Eingabegröße n und studieren, wie P Felder der Länge n sortiert, deren Belegung eine Permutation der Zahlen $0...n-1$ ist. Deren Menge bezeichnen wir mit S_n.

Eine Berechnung von P können wir uns vorstellen als Folge der durchlaufenen Programmzeilen. Für jede Belegung $\pi \in S_n$ bezeichnen wir diese Folge mit v_π. Wir fügen künstlich eine Stop-Zeile ein, so daß jedes v_π mit der Nummer einer (der) Stop-Zeile aufhört.

Offensichtlich: Wenn π und $\pi' \in S_n$ unterschiedlich sind, so müssen auch v_π und $v_{\pi'}$ verschieden sein, denn sonst würden in beiden Feldern exakt dieselben Vertauschungen durchgeführt werden und das Ergebnisfeld könnte nicht in beiden Fällen $[0, ..., n-1]$ sein. Außerdem kann wegen der Stop-Zeile weder v_π ein Anfangsstück von $v_{\pi'}$ sein, noch umgekehrt.

Jetzt betrachten wir die Menge T aller Berechnungen v_π für $\pi \in S_n$ und deren Anfangsstücke. Dann ist T eine Menge von Knoten, die (T_1) erfüllt, d.h. T ist ein Baum mit Lücken bei den Söhnen. Die Blätter entsprechen den $\pi \in S_n$, d.h. es gibt $n!$ Blätter. Der Verzweigungsgrad jedes Knotens ist 0 (Stop-Zeile), 1 (normale Anweisung) oder 2 (if-Anweisung etc.), also ≤ 2. Verzweigungen im Programm können nur dann auftreten, wenn ein Vergleich zwischen Feldelementen durchgeführt wird, d.h. jeder Knoten mit Verzweigungsgrad 2 steht für einen Vergleich.

Also: Die Anzahl der Vergleiche in einer Berechnung v_π ist mindestens so groß wie die Anzahl der Vorfahren mit Grad 2. Definiere daher $p_2(v)$ als die Anzahl der Vorfahren von v vom Grad 2 und $l(T)$ die Anzahl der Blätter eines Baumes.

Lemma: Sei T eine endliche Menge von Knoten, die (T_1) erfüllt und deren Verzweigungsgrad ≤ 2 ist. Falls $p_2(v) \leq k$ für alle $v \in T$ gilt, so gilt auch $l(T) \leq 2^k$.

Beweis per Induktion über $s(T)$:

- *Induktionsanfang:* Wenn $s(T) \leq 1$, so ist die Behauptung trivial.
- *Induktionsannahme:* Sei $n \in \mathbb{N}$, die Behauptung gelte für $s(T) \leq n-1$.
- *Induktionsschritt:* Wenn $s(T) = n$, so unterscheiden wir zwei Fälle:
    1. Die Wurzel hat genau einen Sohn v. Dann ist $l(T) = l(T \downarrow v)$ und mit Induktionsannahme $l(T \downarrow v) \leq 2^n$

2. Die Wurzel hat Verzweigungsgrad 2 mit Söhnen v_L und v_R. Dann gilt: $l(T) = l(T \downarrow v_L) + l(T \downarrow v_R) \leq 2^{n-1} + 2^{n-1} = 2^n$ □

Folgerung: Jedes vergleichsbasierte Sortierverfahren führt im pessimalen Fall $\log n! \geq n \cdot (\log n - \log e) \geq n \cdot (\log n - 1.433)$[13] Vergleiche durch und hat damit pessimale Laufzeit $\Omega(n \cdot \log n)$.

Folgerung: *HeapSort* ist größenordnungsmäßig optimal.

Folgerung: Es kann weder Prioritätsschlangen noch Wörterbücher geben, deren pessimale oder amortisierte Laufzeit größenordnungsmäßig besser als $\log n$ ist.

4.5 *QuickSort* - ein schnelles, aber kein optimales Verfahren

Ansatz: Der Divide-and-Conquer-Ansatz kann beim Sortieren eines Feldes wie folgt verwirklicht werden: Wir nehmen ein beliebiges Feldelement k als sogenanntes *Pivotelement*, ordnen das Feld um, so daß es eine Position j gibt mit folgenden Eigenschaften:

- die Elemente in $a[0\,..\,j-1]$ sind $\leq k$
- die Elemente in $a[j+1\,..\,n-1]$ sind $\geq k$
- $a[j] = k$

Dann sortieren wir $a[0\,..\,j-1]$ und $a[j+1\,..\,n-1]$ seperat rekursiv. Das entstehende Feld ist dann sortiert.

Fragen:

1. Wir wählen wir das Pivot-Element?
2. Wie funktioniert *partition*?

Antworten:

1. Es gibt unterschiedliche Strategien für die Pivotwahl, z.B. immer das linke Element im Feld, immer das recht, immer das mittlere, immer den Median aus linkem mittleren und rechten Element oder einfach immer zufällig.
2. Zum Umordnen benutzen wir eine Prozedur, die große Elemente aus der linken Hälfte mit kleineren Elementen aus der rechten Hälfte vertauscht.

[13] Stirlingsche Formel: $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12} + O\left(\frac{1}{n^2}\right)\right)$

Beispiel:

- Ausgangsfeld: $[5, 7, 2, 9, 1, 4, 8]$
- Aufruf: $partition(0, 0, 6)$
- erste Vertauschung (Pivotelement nach links): $[5, 7, 2, 9, 1, 4, 8]$
- zweite Vertauschung: $[5, 4, 2, 9, 1, 7, 8]$
- dritte Vertauschung: $[5, 4, 2, 1, 9, 7, 8]$
- Pivotelement an die richtige Stelle bringen: $[1, 4, 2, 5, 9, 7, 8]$

Bezeichnung: Mit *SimpleQuickSort* bezeichnen wir das generische *QuickSort* mit der einfachen Pivotwahlstrategie, die $j = 1$ wählt.

Laufzeitanalyse: Prozeduraufruf bei Anwendung von *SimpleQuickSort* auf das Feld $[0, 1, ..., n-1]$:

- $QS(0, n-1)$
- $QS(0, -1)$ und $QS(1, n-1)$
- $QS(1, 0)$ und $QS(2, n-1)$
- $QS(2, 1)$ und $QS(3, n-1)$
- usw...

Jeder Aufruf $QS(i, n-1)$ beinhaltet einen Partitioniervorgang, also Aufwand i, Gesamtaufwand ist also $\geq \sum_{i=1}^{n-1} i \in \Omega(n^2)$.

Folgerung: Die pessimale Laufzeit von *SimpleQuickSort* ist $\Omega(n^2)$ **Beobachtung:** In der Praxis ist *QuickSort* ziemlich schnell! Widerspruch? Nein!

4.5.1 Durchschnittliche Laufzeit von *SimpleQuickSort*

Vorbemerkung: Wenn *partition* immer das mittlerer Element zurückgibt, ist die Laufzeit $\Theta(n \cdot \log n)$. Um eine Aussage über die durchschnittliche Laufzeit erzielen zu können, müssen wir *partition* genau analysieren.

Erster Schritt: Wir definieren die Laufzeit von *QuickSort* auf einem Feld $a[0\,..\,n-1]$ durch

$$v(a) = n + 1 + v(a_l) + v(a_r)$$

wobei a_l und a_r die Felder sein sollen, die nach der Partitionierung links und rechts von dem Pivotelement stehen. Außerdem sei $v(a) = n + 1$ für $n \leq 1$.

Die Anzahl der Vergleiche, die *partition* durchführt, ist n oder $n+1$.

Definition: Als Maß für die *durschnittliche Laufzeit* auf Feldern der Länge n setzen wir an:

$$A(n) = \frac{1}{n!} \sum_{a \in S_n} v(a)$$

Wir werden zeigen:

$$A(n) = n + 1 + \frac{2}{n} \sum_{k=0}^{n-1} A(k) \quad (*)$$

Zweiter Schritt: Satz: Die durchschnittliche Laufzeit von *QuickSort* auf S_n ist $\Theta(n \cdot \log n)$.

Beweis: Aus $(*)$ folgt[14]:

$$\begin{aligned}
n \cdot A(n) - (n-1) \cdot A(n-1) &= 2n + 2A(n-1) \\
\Rightarrow n \cdot A(n) &= 2n + 2A(n-1) + (n-1)A(n-1) \\
&= (n+1)A(n-1) + 2n \\
\Rightarrow \frac{A(n)}{n+1} &= \frac{A(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{A(1)}{2} + \frac{2}{3} + \frac{2}{4} + ... + \frac{2}{n+1} \\
&= 2 \sum_{i=2}^{n+1} \frac{1}{i} \\
&\leq 2 \int_1^n \frac{1}{x}\, dx \\
&= \frac{2}{\log e} \cdot \log n \\
&\leq 1.387 \cdot \log n
\end{aligned}$$

Also gilt

$$A(n) \leq (n+1) 1.387 \log n \in \Theta(n \log n)$$

[14] „Das kann man dadurch machen, daß man Integrale abschätzt - das werden Sie in Analysis 1 (!) bis zum Erbrechen noch üben!“

Ebenso zeigt man, daß $A(n) \in \Omega(n \log n)$. $\square$

Dritter Schritt: Beweis der Rekursionsgleichung $(*)$. Dazu müssen wir *partition* untersuchen. Sei x das partitionierende Element. Beim Aufruf von *partition* ergibt sich eine Folge $(l_0, r_0), ..., (l_s, r_s)$ von Paaren, die miteinander vertauscht werden, abgesehen vom letzten. Es gilt:

$$0 < l_0 < ... < l_{s-1} < r_{s-1} < ... < r_0 \leq n-1$$

$$l_{s-1} < l_s \leq r_{s-1} \text{ und } l_{s-1} \leq r_s < r_{s-1} \text{ und } l_s = r_s + 1$$

Die l_j und r_j sind genau bestimmt durch:

- $a[l_j] > x$ für $j \leq s$ und $j > 0$
- $a[r_j] < x$ für $j \leq s$
- $a[j] < x$ für $j < l_s$ und $j \notin \{l_0, ..., l_{s-1}\}$
- $a[j] > x$ für $j < r_s$ und $j \notin \{r_0, ..., r_{s-1}\}$

Für jedes a, das diese Bedingungen erfüllt, ergeben sich andere Partitionen, und es wird keine denkbare Partition ausgespart. Wir zählen nun für ein gegebenes $k \leq \frac{n-1}{2}$ die Anzahl der Folgen, die den obigen Bedingungen genügen und nennen eine solche Folge *k-Muster*. Offensichtlich kann s die Werte $0, ..., x$ annehmen. Und für jedes Muster gibt es $\binom{x}{s}$ Möglichkeiten, die l_j zu wählen und $\binom{n-x-1}{s}$ Möglichkeiten, die r_j zu wählen. Das heißt, zusammen

$$\binom{x}{s} \cdot \binom{n-x-1}{s}$$

Möglichkeiten für festes s. Insgesamt ergibt sich für festes x:

$$\sum_{i=0}^{x} \binom{x}{s} \cdot \binom{n-x-1}{s} = \sum_{i=0}^{x} \binom{x}{s} \cdot \binom{n-x-1}{n-x-1-s} = \binom{n-1}{x}$$

Also:

$$
\begin{aligned}
A(n) &= \frac{1}{n!}\sum_{a\in S_n} v(a) \\
&= n+1\frac{1}{n!}\sum_{a\in S_n}(v(a_l)+v(a_r)) \\
&= n+1\frac{1}{n!}\sum_{x=0}^{n-1}\left(\sum_{M\,x-Muster}\ \sum_{a\,hat\,M}(v(a_l)+v(a_r))\right) \\
&= n+1\frac{2}{n!}\sum_{x=0}^{n-1}\left(\sum_{M\,x-Muster}\ \sum_{a\,hat\,M} v(a_l)\right) \\
&= n+1\frac{2}{n!}\sum_{x=0}^{n-1}\left(\sum_{M\,x-Muster}(n-x-1)!x!A(x)\right) \\
&= n+1\frac{2}{n!}\sum_{x=0}^{n-1}\left(\binom{n-1}{x}(n-x-1)!x!A(x)\right) \\
&= n+1\frac{2}{n!}\sum_{x=0}^{n-1}\left(\frac{(n-1)!}{x!(n-x-1)!}(n-x-1)!x!A(x)\right) \\
&= n+1\frac{2(n-1)!}{n!}\sum_{x=0}^{n-1}A(x) \\
&= n+1\frac{2}{n}\sum_{x=0}^{n-1}A(x)
\end{aligned}
$$

4.6 *MergeSort* - Verschmelzen und externes Sortieren

Alternative zum Divide-and-Conquer-Ansatz beim *QuickSort*: Wir teilen das Feld in der Mitte, sortieren die beiden Hälften und führen die sortierten Hälften geeigent zusammen.

Frage: Wir führt man zusammen?

Lösung: Durch „Verschmelzen“: Wir vergleichen zuerst die beiden ersten Elemente der Teilfelder und verschieben das kleinere ins Ergebnisfeld, dann verfahren wir rekursiv mit den Restfeldern. Wenn ein Teilfeld leer ist, wird das andere Teilfeld einfach angehängt.

Eingabe	0	4_l	7_l	19	3000	2	4_r	5	7_r	5000
Zeiger	$\uparrow_1$	$\uparrow_3$	$\uparrow_5$	$\uparrow_8$	$\uparrow_9$	$\uparrow_2$	$\uparrow_4$	$\uparrow_6$	$\uparrow_7$	$\uparrow_1$ 1
Ausgabe	0	2	4_l	4_r	5	7_l	7_r	19	3000	5000

Satz: MergeSort ist ein stabiles Sortierverfahren, das höchstens $1.7n \log n$ Vergleiche durchführt und damit eine pessimale Laufzeit von $\Theta(n \log n)$ hat.

Beweis: Sei t_n die pessimale Anzahl von Vergleichen, die $MergeSort$ für ein Feld der Länge n durchführt. Dann gilt:

$$t_2 = 1 \text{ und } t_n \leq t_{\lfloor \frac{n}{2} \rfloor} + t_{\lceil \frac{n}{2} \rceil} + n - 1 \; \forall \; n > 2$$

Wir behaupten, daß dann $t_n \leq d \cdot n \cdot \log n$ für geeignetes d und $n \geq 2$ gilt.

Per Induktion über n

- Induktionsanfang: Behauptung stimmt für $n = 2$, wenn wir $d \geq 1$ wählen.
- Induktionsschritt: Für $n > 2$ setzen wir $m := \lfloor \frac{n}{2} \rfloor$ und $m' := \lceil \frac{n}{2} \rceil$. Dann gilt:

$$\begin{aligned}
t_n &\leq d \cdot m \cdot \log m + d \cdot m' \cdot \log m' + n - 1 \\
&\leq d \cdot (m' + m) \log m' + n - 1 \\
&= d \cdot n \cdot \log m' + n - 1 \\
&\leq d \cdot n \cdot \log m' + d \cdot n \cdot \log \frac{3}{2} \\
&\leq d \cdot n \cdot \log m' + d \cdot n \cdot \log \frac{2n}{n+1} \\
&\leq d \cdot n \cdot \log m' + d \cdot n \cdot \log \frac{n}{m} \\
&\leq d \cdot n \cdot \log m' + d \cdot n \cdot (\log n - \log m') \\
&\leq d \cdot n \cdot \log n
\end{aligned}$$

  wenn wir $d \geq \frac{1}{\log 3 - 1} \leq 1.71$ wählen. Da sich die beiden Bedingungen für d nicht widersprechen, folgt die Behauptung. □

Ziel: Verfahren, das eine Datei oder einen Strom sortiert.

Idee: Wir stellen uns die Datei zerlegt vor in maximal sortierte Teilstücke, sogenannte *Läufe*. Dann teilen wir die Läufe gleichmäßig auf zwei Dateien auf. Die beiden Dateien werden dann Lauf für Lauf (Run) verschmolzen und die entstehende Datei hat höchstens halb so viele Läufe. Wiederholtes Vorgehen liefert in $logn$ Runden eine sortierte Datei, d.h. pessimale Laufzeit ist $\Theta(n \log n)$.

5 Tabellenartige Datenstrukturen (Hash Tables)

Erinnerung: Ein geordnetes Wörterbuch unterstützt neben *add*, $find$ und *remove* auch $getMin$, $getMax$, $getSuccessor$, ... Eine Auflistung aller Objekte in der richtigen Reihenfolge wäre auch effizient möglich. Häufig reichen aber *add*, *remove* (zum Teil) und $find$.

Hoffnung: Kürzere Laufzeit als bei geordneten Wörterbüchern, die untere Schranke trifft nicht mehr zu.

5.1 Motivierendes Beispiel: direkte Adressierung

Ziel: Wir wollen Objekte verwalten, die einen eindeutigen ganzzahligen Schlüssel zwischen 0 und $N-1$ besitzen.

Idee: Wir verwalten die Objekte in einem Feld der Größe $N-1$. Zu Beginn wird das Feld mit null vorbelegt. Beim Einfügen des Objektes mit Schlüssel i wird eine Referenz auf das Objekt an Stelle i abgelegt; löschen ersetzt Referenz durch null und finden bedeutet Auslesen des Feldes.

Laufzeitanalyse: Finden, Einfügen und Löschen können in Zeit $\Theta(1)$ durchgeführt werden.

Problem: Initialisierung benötigt $\Theta(N)$; Speicherplatzbedarf auch $\Theta(N)$.

Beispiel: Ein Proxy verwaltet die Adressen der Seiten, die er im Cache lagert. Selbst wenn die Adressen die Länge 15 haben und aus Buchstaben bestehen, dann bräuchten wir einen Speicherplatz von mehr als $26^{15} = 16.8 \cdot 10^{20}$Bit, also viel mehr als ein Terabyte.

Ziel des Kapitels: Unter Beibehaltung der schnellen Laufzeiten von *add*, $find$ und *remove* wollen wir Platzverbrauch und Zeit für Initialisierung vermindern.

5.2 Ungeordnete Wörterbücher und Hashing mit Kompression durch Division und Kettenbildung

Allgemeine Situation Wir wollen nicht unbedingt annehmen, daß die Schlüssel ganze Zahlen sind. Wir setzen aber voraus, daß auf den benutzten Schlüsseln eine spezielle Methode, $getHashCode$ genannt, definiert ist, die zu einem Schlüssel eine ganze Zahl, seinen *Hashcode*, liefert. Die Abbildung $k \mapsto c$ heißt *Hashcoding*. Von den Schlüsseln sagen wir, daß sie *hashbar* sind.

Definition: Ein abstrakter Datentyp zur Verwaltung von Objekten mit eindeutigen hashbaren Schlüsseln, der die Operationen *add*, *remove* und $find$ unterstützt, wird (ungeordnetes) Wörterbuch genannt.

Frage: Wie können wir die Methode der direkten Adressierung verbessern?

Antwort: Wir benutzen kleinere Felder, müssen dann aber unter Umständen mehrere Elemente an der gleichen Stelle im Feld ablegen. Das müssen wir ohnehin, da die Hashcodes nicht eindeutig sind.

Frage: Wie ordnen wir den Schlüsseln bzw. Hashcodes Feldindizes zu?

Antwort: Wenn m die Feldlänge ist, so bilden wir jeden Hashcode c auf $c \bmod m$ ab, die Abbildung $c \mapsto c \bmod m$ heißt *Kompressionsabbildung*. Die Hintereinanderausführung von Kompressionsabbildung nach der Hashcodierung wird *Hashfunktion* genannt, ihre Werte heißen *Hashwerte*.

Frage: Wir gehen wir mit *Kollisionen* um, d.h. mit Paaren von Objekten, die denselben Hashwert haben?

Antwort: Im Feld werden die Objekte nicht direkt abgelegt, sondern in einer eigenen einfachen Datenstruktur abgelegt. Allgemeine Bezeichnung ist *Eimer*, wir stellen uns konkret einfach verkettete Listen vor. Die gesamte Datenstruktur heißt *Hashtabelle*.

Platzbedarf: Wir bezeichnen die Anzahl der verwalteten Objekte mit n. Also Platzbedarf $\Theta(n + m)$.

Laufzeit: im schlechtesten Fall $\Theta(n + 1)$.

Frage: Was haben wir überhaupt gewonnen?

Analyse: Allgemeine Analyse ist schwierig. Wir betrachten Spezialfall.

Definition: Der *Füllgrad* einer Hashtabelle ist $\frac{n}{m}$ und wird mit λ bezeichnet.

Satz: Wir betrachten das Finden in einer Hashtabelle mit Kettenbildung.

1. Wird nach einem Element gesucht, das nicht in der Tabelle steht, und ist dessen Hashwert zufällig und gleichverteilt, so ist die erwartete Laufzeit $\Theta(1 + \lambda)$.

2. Wird eine Tabelle so gefüllt, daß die Hashwerte der einzufügenden Objekte zufällig mit gleicher Wahrscheinlichkeit gewählt werden, und wird dann nach einem schon in der Tabelle vorhandenen Objekt zufällig mit gleicher Wahrscheinlichkeit gesucht, so ist die erwartet Laufzeit $\Theta(1 + \frac{\lambda}{2} \cdot (1 + \frac{1}{n}))$

Bemerkung: Die obigen Annahmen werden als *Uniformitätsannahmen* bezeichnet.

Beweis:

1. Das zu findende Objekt habe den Hashwert i. Dann muß die gesamte Liste $a[i]$ durchlaufen werden, um dies festzustellen. Das heißt, die Laufzeit ist $\Theta(1 + \text{length}(a[i]))$. Die erwartete Laufzeit ist also:

$$\begin{aligned} \frac{1}{m}\sum_{i=0}^{m}(1 + \text{length}(a[i])) &= 1 + \frac{1}{m} \cdot \sum_{i=0}^{m} \text{length}(a[i]) \\ &= 1 + \frac{1}{m} \cdot n = 1 + \lambda \end{aligned}$$

2. Wir betrachten den Füllprozess und nehmen an, daß $u_0, ..., u_{i-1}$ schon eingefügt wurden. Seien $l_0, ..., l_{m-1}$ die Längen der Listen. Da u_i mit der Wahrscheinlichkeit $\frac{1}{m}$ auf j gehasht wird, ist die erwartete Laufzeit zum Finden von u_i:

$$1 + \frac{1}{m} \cdot \sum_{j=0}^{m} l_j = 1 + \frac{i}{m}$$

   Wenn wir zufällig nach dem Füllen nach einem Element suchen, erhalten wir[15]

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n}\left(1 + \frac{i}{m}\right) &= 1 + \frac{1}{n}\sum_{i=0}^{n}\frac{i}{m} \\ &= 1 + \frac{1}{n \cdot m}\sum_{i=0}^{n} i \\ &= 1 + \frac{1}{n \cdot m} \cdot \frac{n(n-1)}{2} \\ &= 1 + \frac{\lambda}{2}\left(1 - \frac{1}{n}\right) \\ &\stackrel{?}{\neq} 1 + \frac{\lambda}{2}\left(1 + \frac{1}{n}\right) \end{aligned}$$

□

[15] Hier ist offenbar noch ein Fehler im Beweis, am Ende müßte laut Beweis $-\frac{1}{n}$ herauskommen...

Konsequenz für die Implementierung: Wir legen einen maximalen Füllgrad fest, der nicht überschritten werden soll. Bei drohender Überschreitung wird eine Vergrößerung des Feldes (verdoppelt) vorgenommen (analog zur Feldimplementierung von Listen). Die Objekte werden von der alten Tabelle in die neue übertragen (*Neuhashen*). Der Aufwand ist von amortisierter konstanter Zeit. Typische Schranke: < 0.9.

JAVA: in der Übung.

5.3 Kompressionsabbildungen und Hashkodierungen

Erinnerung: Im letzten Verfahren haben wir genau eine Hashcodierung (die triviale) und eine Kompressionsabbildung (Divisionsmethode) kennengelernt. In diesem Paragraphen geht es nun weiter.

Kompressionsmethoden:

- *Kompression durch Division:* Zu gegebener Tabellengröße m ist die Abbildung gegeben durch

$$C_m : \mathbb{Z} \to \{0, ..., m-1\} \text{ mit } c \mapsto c \bmod m$$

- *Kompression durch Multiplikation:* Wähle eine (Näherung einer) irrationalen Zahl $A \in (0, 1)$. Dann ist die Abbildung gegeben durch:

$$C_{a,m} : \mathbb{Z} \to \{0, ..., m-1\} \text{ mit } c \mapsto \lfloor m \cdot ((c \cdot A) \bmod 1) \rfloor$$

  Für $A = \frac{\sqrt{5}-1}{2}$ ergibt sich eine besonders „gute“ Verteilung der Hashwerte. Die Berechnung von $C_{A,m}$ ist unter gewissen Umständen einfach:

  > Annahme: Der Nachkommaanteil von A wird als l-Bit-Zahl geschrieben, es gilt $c < 2^l$ und $m = 2^k$ mit $k \leq l$.

- *Kompression durch modulare Multiplikation:* Für m Primzahl und Konstanten a, b mit $a \neq 0$ und $a, b < m$ ist die Abbildung gegeben durch:

$$C_{a,b,m} : \mathbb{Z} \to \{0, ..., m-1\} \text{ mit } c \mapsto (ac + b) \bmod m$$

  Wenn m keine Primzahl[16] ist und a nicht teilerfremd zu m ist, dann werden gewisse Hashwerte gar nicht vorkommen.

Hashcodierungsmethoden:

[16] Anmerkungen: Primzahl finden nur durch raten und testen; mehr Primzahlen als Zweierpotenzen; zwischen einer Zahl und dem Doppelten liegt mindestens eine Primzahl

- *Hashcodierung durch Typanpassung:* Wenn die Schlüssel Zahlen sind, können sie häufig als ganze Zahlen interpretiert werden, z.B. ist jedes Element von Typ *Byte* oder *Short* oder *char* leicht durch Typanpassung in ein Element vom Typ *int* umwandelbar.

- *Hashcodierung durch komponentenweise Summation:* Bei einer Zahl vom Typ *long* (64 Bits) fällt die Interpretation als *int* (32 Bits) nicht leicht. Einfaches Verfahren: Man betrachtet die niederwertigen 32 Bits als *int* und die höherwerigen ebenso und addiert beide: `(int)(i >> 32) + (int)i`

- *Polynomiale Hashcodierung für Zeichenketten:* Für eine Zeichenkette $z = z_0, ..., z_{n-1}$ bietet es sich an, jeden Buchstaben als Zahl aufzufassen und einen Hashcode durch Polynomauswertung zu bestimmen:

  $$h_x(z) = z_0 x^{n-1} + z_1 x^{n-2} + ... + z_{n-1} \text{ (Rechnen mit } ints)$$

  Die Auswertung kann effizient nach dem Horner-Schema erfolgen:

  $$h_x(z) = z_{n-1} + x \cdot (z_{n-2} + x \cdot (z_{n-3} + x \cdot (...)))$$

  Diese Methode ist generell nutzbar, wenn die Schlüssel Zahlen sind.

- *Hashcodierung für Zeichenketten durch zyklische Vertauschung:* Gleicher Ansatz wie vorher, jedoch wird das Multiplizieren ersetzt durch zyklische Vertauschung um eine feste Anzahl von Bits:

  $$h_i(z) = z_{n-1} + rot_l(i, z_{n-2} + rot_l(i, z_{n-3} + ...))$$

Bewertung: Ziel ist eine möglichst gleichmäßige Verteilung der Hashwerte der betrachteten Objekte.

Problem: Stark anwendungsabhängig:

Beispiele: (aus der Literatur)

- Nimmt man 50.000 Wörter aus einem englischen Standard-Wörterbuch, so erhält man bei der polynomialen Hashcodierung für $x = 33$ oder $x = 37$ weniger als 7 Kollisionen. JAVA benutzt $x = 31$.

- Nimmt man knapp 25.000 Wörter, so erhält man bei der Methode der zyklischen Vertauschung für $i = 5$ nur 4 Kollisionen.

5.4 Hashtabellen mit offener Adressierung

Beobachtung: Kettenbildung benötigt zusätzlichen Verwaltungsaufwand.

Alternative: Objekte werden wie bei direkter Adressierung direkt im Feld abgelegt. Bei einer Kollision wird auf eine Ersatzposition ausgewichen, man sucht ausgehend von der gerade berechneten Position weiter. Unter Umständen muß dann auch eine Ersatzposition einer Ersatzposition aufgesucht werden usw. Das Suchen wird als *Sondieren* bezeichnet.

Formulierung: Eine Kompressionsabbildung mit k-facher Sondierung ist eine Abbildung

$$C : \mathbb{Z} \times \{0, ..., k-1\} \to \{0, ..., m-1\}$$

Beim Sondieren werden nacheinander $C(c, 0), C(c, 1), C(c, 2), ...$ aufgesucht.

Lineare Sondierung: Sei $C_0 : \mathbb{Z} \to \{0, ..., m-1\}$ eine beliebige Kompressionsabbildung. Dann erhält man eine Kompressionsabbildung mit m-facher Sondierung durch $C_l : (c, i) \mapsto (C_0(c) + i) \bmod m$. Damit ist $C_l(\cdot, 0) = C_0$.

Problem: Lineares Sondieren führt schnell zu zusammenhängenden, besetzten Bereichen, sogenannten *Clustern*, und damit auch zu häufigen Kollisionen.

Begründung: Wenn schon $a[r \,..\, r+s-1]$ besetzt sind und das nächst Objekt zufällig gehasht wird, dann wir das Cluster mit Wahrscheinlichkeit $\frac{s}{m}$ getroffen. Es kommt durchschnittlich (Erwartungswert) zu mindestens

$$\frac{1}{m} \sum_{i=0}^{s} (s - i) = \frac{s(s-1)}{2m}$$

Sondierungsversuchen und mit Wahrscheinlichkeit $\geq \frac{s}{m}$ vergrößert sich das Cluster.

Quadratische Sondierung:

$$C_{d_0,d_1,d_2}(c, i) = (C_0(c) + (d_2 i^2 + d_1 i + d_0)) \bmod m$$

Problem: Es entsteht ein Clusterring zweiter Art[17]; für bestimmte Werte von m und der d_i werden nicht alle Tabelleneinträge sondiert. Dann spricht man von einer *unvollständigen Sondierung.*

Genauer: Eine Kompressionsabbildung mit k-facher Sondierung heißt *vollständig*, wenn für jedes $c \in \mathbb{Z}$ gilt: $\{0, ..., m-1\} = \{C(c, 0), ..., C(c, k-1)\}$

[17] wirklich?!

Doppeltes Hashen: Wir setzen eine weitere Kompressionsabbildung C_1 voraus und setzen

$$C_{C_1}(c, i) = C_0(c) + i \cdot C_1(c)$$

Typische Wahl: $C_0(c) = c \bmod m$ und $C_1(c) = 1 + (c \bmod m')$ und m und m' teilerfremd.

Satz: Wir betrachten das Finden in einer Hashtabelle mit offener Adressierung und vollständiger m-facher Sondierung. Uniformitätsannahme: Alle möglichen Sondierungsfolgen treten auf und sind gleich wahrscheinlich. Sei $\lambda < 1$. Dann gilt:

1. Die erwartete Anzahl von Sondierungen bei einer erfolglosen Suche ist

$$\leq \frac{1}{1-\lambda} = 1 + \lambda + \lambda^2 + \lambda^3 + ...$$

2. Die erwartete Anzahl von Sondierungen bei einer erfolgreichen Suche ist

$$\leq \frac{1}{\lambda \cdot \log e} \cdot \log \frac{1}{1-\lambda} + \frac{1}{\lambda}$$

Beweis: sparen wir uns[18]

Beispiel: Bei $\lambda = 0.5$ (typsich) ergibt sich in beiden Fällen ein Wert kleiner 3.387 und bei $\lambda = 0.9$ ein Wert < 3.670.

Problem: Löschen ist schwierig.

Lösung: Soll ein Element gelöscht werden, so wird es nicht durch eine Null-Referenz ersetzt, sondern durch eine Referenz auf ein entsprechendes Markerobjekt („hier wurde gelöscht!“). Außerdem wird häufiger neugehast (nicht nur zur Vergrößerung).

5.5 Universelles Hashing

Erinnerung: Wir haben bei unseren Analysen immer vorausgesetzt, daß die Hashwerte der einzufüfenden Objekte zufällig gewählt werden.

Problem: Letztendlich liegt die Auswahl der Operationen in der Hand des Benutzers bzw. ist durch die konkrete Anwendung bestimmt.

Idee: Wir bringen den Zufall ins Spiel: Wir wählen zufällig eine Hashfunktion und hoffen, daß sich für jede Anwendung mit hoher Wahrscheinlichkeit kurze Laufzeiten ergeben.

[18] „... aber wir können zum Beispiel mal was einsetzen, das ist doch auch schon mal was!“

Frage: Wie muß die Familie von Hashfunktionen gewählt werden?

Vorüberlegung: Wir berechnen, wie viele Kollisionen unvermeidbar sind: Sei $H : N \to M$ mit $M = \{0, ..., m-1\}$ und $N = \{0, ..., n-1\}$ eine Hashfunktion, die n Schlüssel auf m Hashwerte abbildet. Eine Kollision sei ein Paar (x, y) von unterschiedlichen Schlüsseln mit $H(x) = H(y)$.

Lemma: Die durchschnittliche (erwartete) Anzahl von Kollisionen ist größergleich $\frac{1}{m} \cdot \left(1 - \frac{m-1}{n-1}\right)$.

Beweis: Zu $j < m$ setze $k_j = |H^{-1}(j)|$. Dann ist die erwartete Anzahl von Kollisionen gegeben durch

$$\begin{aligned} \frac{1}{n(n-1)} \overbrace{\sum_{j=0}^{m-1} k_j(k_j-1)}^{(*)} &\geq \frac{1}{n(n-1)} \sum_{j=0}^{m-1} \left(\frac{n}{m} \cdot \left(\frac{n}{m} - 1\right)\right) \\ &= \frac{1}{n-1} \cdot \frac{n-m}{m} \\ &= \frac{1}{m} \frac{n-m}{n-1} \\ &= \frac{1}{m} \cdot \frac{n-1-(m-1)}{n-1} \end{aligned}$$

Die Summe $(*)$ wird minimal, wenn alle k_j gleich sind. □

Folgerung: Wenn $F = \{H_i\}_{i<k}$ eine Familie von Hashfunktionen ist, dann gibt es ein Paar (x, y) mit $x \neq y$, so daß die Wahrscheinlichkeit, daß (x, y) eine Kollision für eines der H_i ist, mindestens $\frac{1}{m} \cdot \left(1 - \frac{m-1}{n-1}\right)$ ist.

Beobachtung: Für $n \to \infty$ konvergiert der obige Ausdruck gegen $\frac{1}{m}$.

Definition: Eine Familie F wie oben heißt universelle Familie von Hashfunktionen (UFH), wenn für jedes Paar (x, y) mit $x \neq y$ die Wahrscheinlichkeit, daß (x, y) eine Kollision für eines der H_i ist, höchstens $\frac{1}{m}$ ist.

Frage: Ist die Definition gut? **Antwort:** Ja!

Satz: Wir betrachten Hashing mit Kettenbildung und zufälliger Auswahl der Hashfunktion aus einer universellen Familie von Hashfunktionen. Dann gilt für jede Folge von r Operationen (ohne Löschen), daß die erwartete Laufzeit $\Theta(r(1 + \lambda))$ ist. (ohne Beweis)

Frage: Wie findet man UFHs?

Antwort: leicht: Man nehme z.B. die Familie aller Funktionen $N \to M$. Diese ist eine UFH, aber völlig unpraktikabel! (siehe Übung).

Frage: Gibt es bessere Familien? **Antwort:** Ja!

Konstruktion: Wir wählen für m eine Prt. und nehmen an, daß k und r so gewählt sind, daß $m > 2^k$ und $2^{kr} > n$ gelten. Dann wird zu jeder Folge $a = (a_0, ..., a_{r-1})$ mit $a_i < m$ eine Hashfunktion definiert durch

$$H_a : \sum_{i=0}^{r-1} x_i 2^{ki} \mapsto \sum_{i=0}^{r-1} a_i x_i \bmod m \text{ mit } x_i < 2^k$$

Satz: Die obige Familie ist eine UFH.

Beweis: Seien $x, y \in N$ mit $x \neq y$ und $x = \sum_{i=0}^{r-1} x_i 2^{ki}$ bzw. $y = \sum_{i=0}^{r-1} y_i 2^{ki}$. Dann ist $H_a(x) = H_a(y)$ genau dann, wenn

$$\begin{aligned} \sum_{i=0}^{r-1} a_i x_i \bmod m &= \sum_{i=0}^{r-1} a_i y_i \bmod m \\ \Leftrightarrow 0 &= \sum_{i=0}^{r-1} a_i (x_i - y_i) \bmod m \end{aligned}$$

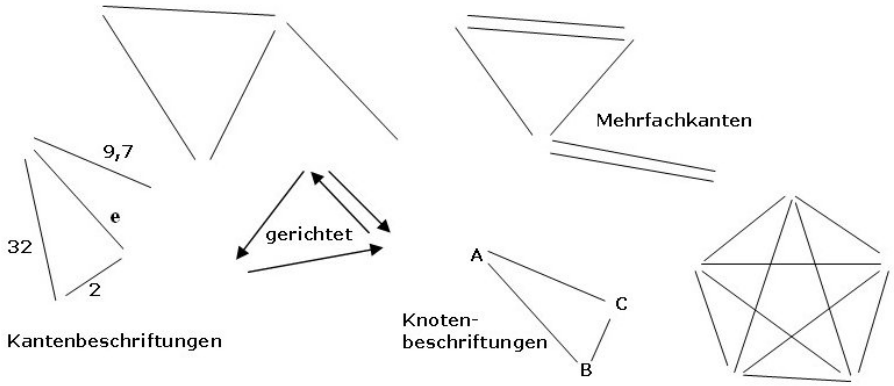
Damit ist die Anzahl der Kollisionen kleinergleich der Anzahl der Lösungen einer nichttrivialen linearen Gleichung mit r Variablen über einem Körper mit m Elementen (wegen $\bmod m$, funktioniert nur, wenn m Primzahl ist). Diese Anzahl ist m^{r-1}. Da es m^r Hashfunktionen gibt, folgt die Behauptung. $\square$

6 Grundlegende Graphenalgorithmen

Graphen gehören zu den wichtigsten Objekten, die in der Informatik betrachtet werden.

6.1 Graph ist nicht gleich Graph

Unterscheidungsmerkmale und Charakteristika:



- gerichtete oder ungerichtete Kanten
- beschriftete oder unbeschriftete Kanten
- gewichtete oder ungewichtete Kanten
- beschriftete oder unbeschriftete Knoten
- Schleifen erlaubt oder nicht
- Mehrfachkanten erlaubt (sog. Multigraphen) oder nicht
- (Hypergraphen, d.h. eine Kante verbindet u.U. mehr als zwei Punkte)
- usw.

6.2 Ungerichtete Graphen

Im Folgenden: zunächst nur ungerichtete Kanten.

Definitionen:

- 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 mit $\binom{V}{2} :=$ Menge aller zweielementigen Teilmengen von V. Also: keine Schleifen, keine Mehrfachkanten.

- Wenn G einen Graph bezeichnet, dann werden Knoten- und Kantenmenge mit V_G und E_G bezeichnet.

- Die Menge E_G wird auch als *Adjazenzrelation* bezeichnet, falls $\{u, v\} \in E_G$, so heißt v *adjazent* zu u.

- Wenn G ein ungerichteter Graph ist und $u \in V_G$, so ist der *Grad* des Knoten $d_G(u) = |\{ v \mid \{u, v\} \in E_G\}|$.

- Eine nichtleere Folge $P = (u_0, ..., u_n)$ von Knoten heißt *Pfad* in/durch G, wenn $\{u_i, u_{i+1}\} \in E_G$. P ist ein Pfad von u_0 nach u_n und hat die Länge n. Er enthält die Knoten $u_0, ..., u_n$ und die Kanten $\{u_0, u_1\}$ bis $\{u_{n-1}, u_n\}$. Der Knoten u_0 ist durch P mit u_n verbunden bzw. der Knoten u_n ist von u_0 über P erreichbar, in Zeichen:

$$u_0 \overset{P}{\underset{G}{\rightsquigarrow}} u_n$$

- P heißt *geschlossen*, wenn $u_0 = u_n$ und $n > 0$ gilt.

- P heißt einfach, wenn $u_i \neq u_j$ für alle $i, j \leq n$ mit $i \neq j$.

- Ein *Teilpfad* von P ist ein von der Form $(u_i, ..., u_{i+j})$ mit $0 \leq i \leq i+j \leq n$

- Beschreibung durch Äquivalenzrelationen: Wir schreiben $u \leftrightsquigarrow v$ genau dann, wenn $u \rightsquigarrow v$ und $v \rightsquigarrow u$ gilt (bei ungerichteten Graphen gilt natürlich $u \rightsquigarrow v$ genau dann, wenn $v \rightsquigarrow u$ genau dann, wenn $u \leftrightsquigarrow v$). Dann ist $\leftrightsquigarrow$ eine Äquivalenzrelation. Die Äquivalenzklassen von $\leftrightsquigarrow$ heißen Zusammenhangaskomponenten, in Zeichen: $CC(u)$

- Ein Pfad $(u_0, ..., u_n)$ in einem ungerichteten Graphen ist ein *Zykel*, wenn $u_0 = u_n$ gilt, $(u_0, ..., u_{n-1})$ einfach ist und $n > 2$ ist.

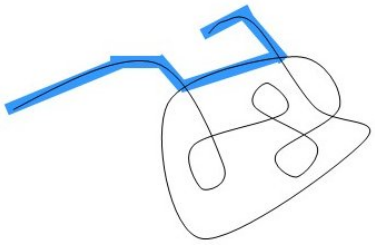
- Ein ungerichteter Graph ohne Zykel heißt *azyklisch*.

- Ein ungerichteter Graph ist *zusammenhängend*, wenn ein Knoten von jedem erreichbar ist, d.h. für alle $u, v \in V_G$ gilt: $u \rightsquigarrow v$ (und $v \rightsquigarrow u$). Er ist zusammenhängend genau dann, wenn $\leftrightsquigarrow$ höchstens eine Äquivalenzklasse besitzt.

- Ein ungerichteter Graph heißt *Baum*, wenn er azyklisch und zusammenhängend ist.
- Ein ungerichteter Graph heißt *Wald*, wenn er azyklisch ist.
- Ein *verwurzelter Baum* ist ein Baum zusammen mit einem Knoten, d.h. (T, ϱ) mit T Baum und $\varrho \in V_T$. Der Knoten ϱ heißt *Wurzel*. Dann übertragen sich die bekannten Begriffe in natürlicher Weise: Zu jedem Knoten $u \in V_T$ sei P_u der eindeutige einfache Pfad von ϱ zu u. Es seien $u, v \in V_T$. Der Knoten v ist
  - Nachfahre von u, wenn $u \neq v$ gilt und P_u ein Teilpfad von P_v ist,
  - Vorfahre von u, wenn u ein Nachfahre von v ist,
  - Sohn von u, wenn v ein Nachfahre von u und adjazent zu u ist,
  - Vater von u, wenn u ein Sohn von v ist,
  - ein Blatt, wenn v keine Nachfahren besitzt,
  - ein innerer Knoten, wenn er kein Blatt ist.
- Ein Graph G heißt *Teilgraph* eines Graphen H, wenn $V_G \subseteq V_H$ und $E_G \subseteq E_H$ gilt. Er heißt *induzierter* Teilgraph, wenn $E_G = E_H \cap \binom{V_G}{2}$. Dann schreiben wir $G = H[V_G]$. Für $G[CC_G(u)]$ schreiben wir auch $GCC_G(u)$.
- Die *Entfernung* von u nach v ist ∞, falls v nicht von u aus erreichbar ist. Sonst ist sie die kleinste Länge eine Pfades von u nach v.

Lemma: Sei G ein ungerichteter Graph und u, v Knoten von G. Wenn v von u erreichbar ist über einen Pfad, so auch über einen einfachen Pfad, der nicht länger ist.

Veranschaulichung:



Beweis: durch Induktion nach Länge des Pfades.

- Induktionsverankerung: Falls $n = 0$, so stimmt die Behauptung trivialerweise.

- Induktionsannahme: Sei $n \in \mathbb{N}$ und die Behauptung gelte für alle Pfade der Länge $< n$.
- Induktionsschluß: Wenn P einfach ist, brauchen wir nichts zu zeigen. Angenommen, P ist nicht einfach. Dann gibt es i, j mit $0 \leq i < j \leq n$ und $u_i = u_j$. Wenn $j = n$, so verbindet der Pfad $(u_0, ..., u_i)$ die Knoten u_0 und u_n und wir erhalten nach Induktionsannahme einen entsprechenden einfachen Pfad. Falls $j < n$, so verbindet auch $(u_0, ..., u_i, u_{j+1}, ..., u_n)$ den Knoten u_0 mit u_n und die Induktionsannahme ist anwendbar. □

Lemma: Für einen endlichen ungerichteten Graphen G sind die folgenden Bedingungen äquivalent:

1. G ist ein Baum.
2. Jeweils zwei Knoten von G sind durch genau einen einfachen Pfad verbunden.
3. G ist zusammenhängend und für jede Kante $e \in E_G$ ist $(V_G, E_G \setminus \{e\})$ nicht zusammenhängend.
4. G ist zusammenhängend und $|V_G| = |E_G| + 1$.
5. G ist azyklisch und $|V_G| = |E_G| + 1$.
6. G ist azyklisch und für jedes $e \in \binom{V_G}{2} \setminus E_G$ besitzt $(V_G, E_G \cup \{e\})$ einen Zykel.

Beweis: Wir beweisen nur $1 \Rightarrow 2$, weitere in der Übung.

1. $\Rightarrow$ 2. Wenn G ein Baum ist, so ist G zusammenhängend, d.h. von jedem Knoten zu jedem gibt es einen Pfad und damit auch nach obigem Lemma auch einen einfachen Pfad. Angenommen, es gäbe zwei Knoten, die durch zwei einfache Pfade verbunden sind. Dann gäbe es auch zwei einfache Pfade $P = (u_0, ..., u_m)$ und $P' = (v_0, ..., v_n)$ mit $u_0 = v_0$ und $u_n = v_n$. Unter all 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 $0 \leq j \leq n$ (sowie für alle i und j mit $0 \leq i \leq m$ und $0 < j < n$), da sonst $m + n$ nicht minimal wäre oder P oder P' nicht einfach wäre. Dann ist jedoch $(u_0, ..., u_m, v_{n-1}, ..., v_0)$ ein Zykel. □

6.3 Datenstrukturen für Graphen und ADT

Vereinbarung: Wir betrachten nur noch endliche Graphen.

Vorbemerkung: Wir beschäftigen uns mit Algorithmen auf Graphen, die die Graphen analysieren und nicht verändern, d.h. nicht im Sinne von Datenstruktur benutzen. Deshalb werden wir auch nur wenige Methoden zur Manipulation von Graphen bereitstellen. Außerdem werden wir davon ausgehen, daß die Knotenmenge von der Form $\{0, ..., n-1\}$ ist. Dafür erlauben wir aber, daß am Knoten i ein Objekt abgelegt ist.

ADT für Graphen:

- $numberOfVertices$, $numberOfEdges$, $degree(i)$
- $setObject(i, o)$, $getObject(i)$
- $addEdge(i, j)$
- $arrayOfAdjecentVertices(i)$, $hasEdge(i, j)$
- $newGraph(n)$

Darstellung von Graphen: Im Wesentlichen zwei Möglichkeiten:

- durch *Adjazenzmatrizen:* in einer boolschen $n \times n$-Matrix wird an der Stelle ij festgehalten, ob es eine Kante von j nach j gibt. Die Darstellung verbraucht mehr Platz, dafür ist jedoch ein schnellerer Zugriff auf die Informationen möglich:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

- durch *Adjazenzlisten:* in einem Feld der Länge n wird an der Stelle i eine Liste mit den zu i adjazenten Knoten abgelegt. Die Darstellung spart Platz, macht jedoch die Implementierun von z.B. $hasEdge(i, j)$ schwieriger:

0	2 4 1
1	0 3
2	0 4
3	4 1
4	3 0 2

6.4 Entfernungen, kürzeste Pfade und Breitensuche

Bezeichnung: Mit $\text{dist}_G(u, v)$ wird die Entfernung von u zu v bezeichnet. Gilt $\text{dist}_G(u, v) < \infty$, so heißt der Pfad von u nach v mit Länge $\text{dist}_G(u, v)$ ein kürzester Pfad von u nach v.

Ziel: Wir suchen einen Algorithmus, der $\text{dist}_G(u, v)$ für festes u und alle v ausdrechnet und kürzeste Pfade bestimmt.

Frage: Wie stellen wir die kürzesten Pfade dar?

Antwort: In einem Baum.

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 zu v in T, so ist P ein kürzester Pfad von u nach v in G.

Frage: Wie repräsentiert man (T, u)?

Antwort: In einem Feld $parent[0\,..\,n-1]$. Dabei wird an der Stelle i der Vater von i abgelegt. -1 bedeutet *Wurzel*, und ∞ bedeutet *gehört nicht zum Baum*. Der Baum wird dann mit T_{parent} bezeichnet.

Idee für den Algorithmus: Die Knoten, die adjazent zu u sind, haben Abstand 1. Die Knoten, die adjazent zu einem Knoten mit Abstand 1 sind und nicht selbst Abstand 1 oder 0 haben, haben Abstand 2. ...

Fazit: Die Knoten müssen also abgearbeitet werden in der Reihenfolge, in der wir sie entdecken.

Satz: Sei G ein endlicher ungerichteter Graph und $u \in V_G$.

1. Nach Beendigung von $BFS(u)$ gilt:
   (a) $dist[v] = dist_G(u, v)\ \forall\ v \in V_G$
   (b) Der durch *parent* definierte Baum T_{parent} ist ein Entfernungsbaum für $u \in G$
2. Die Laufzeit von $BFS(u)$ ist $\Omega(m_u + n)$, wenn $n = |V_G|$ und m_u die Anzahl der Knoten von $GCC_G(u)$ ist.

Beweis:

1. Wir geben eine entsprechende Schleifeninvariante an. Dazu einige Notationen:

$$\text{reach}_G(v) = \left\{ w \mid v \underset{G}{\rightsquigarrow} w \right\} \text{ und } \text{reach}_G(U) = \bigcup_{v \in U} \text{reach}_G(v)\ \forall\ v \in V, U \subseteq V$$

Mit V_g, V_w, V_b bezeichnen wir die Menge der grauen, weißen bzw. schwarzen Knoten. Mit $[q]$ bezeichnen wir die Menge der Knoten in q. Die Invariante besteht dann aus folgenden Bedingungen:

(a) V_G ist disjunkte Vereinigung von V_w, V_g und V_b.

(b) $V_g = [q]$

(c) $\text{reach}_G(u) = V_b \cup \text{reach}_G(V_g)$

(d) $\text{dist}[v] = \text{dist}_G(u, v) \; \forall \; v \in V_b \cup V_g$

(e) $\text{dist}[v] = \infty \; \forall \; v \in V_w$

(f) T_{parent} ist ein Entfernungsbaum für u in $G[V_b \cup V_g]$

(g) $[q]$ enthalte $v_0, ..., v_{r-1}$ (mit v_{r-1} am ältesten). Dann gilt:

i. $\text{dist}[v_i] \geq \text{dist}[v_j] \; \forall \; i, j$ mit $0 \leq i \leq j < r$

ii. $\text{dist}[v_0] \leq \text{dist}[v_{r-1}] + 1$

iii. $\{ v \mid \text{dist}_G(u, v) \leq dist[v_{r-1}] \} \subseteq V_b \cup V_g$

iv. Jedes v mit $\text{dist}_G(u, v) = dist[v_{r-1}] + 1$, für das es kein $w \in V_g$ mit $\text{dist}_G(u, w) = \text{dist}[v_{r-1}]$ und $\{w, v\} \in E$ gibt, gehört zu V_g.

Nachweiß, daß dies tatsächlich eine Invariante ist, ist aufwendig (machen wir hier nicht), aber in der Übung: Aus Invariante und der Bedingung, daß q leer ist, folgt die Nachbedingung.

2. Zeitaufwand:

- Initialisierung: $\Theta(n)$
- Anzahl der Durchläufe der solange-Schleife: $\Theta(n_u)$, wenn $n_u = |CC_G(u)|$
- Anzahl der Durchläufe der für-Schleife: $\Theta(m_u)$
- Insgesamt: $\Theta(n + m_u)$

6.5 Gewichtete Graphen und Dijkstras Algorithmen

Vorbemerkung: In diesem Paragraphen betrachten wir eine Verallgemeinerung der Fragestellung aus dem letzten Paragraphen: Wir wollen Entfernungen und Entfernungsbäume in gewichteten Graphen berechnen.

Definition: Ein gewichteter (ungerichteter) Graph ist ein Tripel (V, E, ν), bei dem (V, E) ein ungerichteter Graph und $\nu : E \to \mathbb{R}_{\geq 0}$ ist. ν heißt *Gewichtsfunktion* (Längen-/Bewertungsfunktion).

Notation: Für $\nu(\{u, v\})$ schreiben $\nu(u, v)$.

Definition: Ein Pfad $P(u_0, ..., u_n)$ durch einen gewichteten Graphen $G = (V, E, \nu)$ hat die *gewichtete Pfadlänge* $\|P\|_G = \sum_{i=0}^{n-1} \nu(u_i, u_{i+1})$. Die gewichtete Abstandsfunktion ist wie folgt definiert:

$$\mathrm{dist}_G(u, v) = \begin{cases} \infty & \text{falls} \quad \neg\exists\, v \rightsquigarrow u \\ \inf\{\, \|P\|_G \mid u \rightsquigarrow_G^P v \,\} & \text{sonst} \end{cases}$$

Bemerkung: Falls G endlich ist, kann in der Definition inf durch min ersetzt werden.

Definition: Ein gewichteter Entfernungsbaum von u in G ist ein verwurzelter Baum (T, u), bei dem T ein Teilgraph von (V_G, E_G) ist, der zu jedem Knoten v, der von u erreichbar ist, einen Pfad kürzester gewichteter Pfadlänge von u nach v enthält.

Frage: Wie berechnet man gewichtete Abstände und Entfernungsbäume?

Antwort: Durch geschickte Modifikation der Breitensuche.

Ansatz: Offensichtlich hat u den gewichteten Abstand 0 von u. Seien $v_0, ..., v_{r-1}$ die zu u adjazenten Knoten. Für $i := \min\{\nu(u, v_i) \mid i = 0, ..., r-1\}$ gilt $\mathrm{dist}_G(u, v_i) = \nu(u, v_i)$. Dadurch finden wir für mindestens einen weiteren Knoten die richtige Entfernung.

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

$$\mathrm{border}_G(U) = \{\, w \in V_G \setminus U \mid \exists\, v \in U(\{v, w\} \in E_G)\}$$

Sei $U \subseteq V_G$ und $W = \mathrm{border}_G(U)$. Die *Randapproximation* zu u und U ist die Funktion $W \to \mathbb{R}_{\geq 0}$ definiert durch

$$d(w) := \min\{\, \mathrm{dist}_G(u, v) + \mu(v, w) \mid v \in U, \{v, w\} \in E_G\}$$

Lemma: Sei G ein gewichteter Graph. $u \in V_G$, $U \subseteq V_G$ mit $u \in U$ und d die Randapproximation zu u und U. Sei $w \in \mathrm{border}_G(U)$ mit $d(w) \leq d(w')$ für alle $w' \in \mathrm{border}_G(U)$. Dann gilt $d(w) = \mathrm{dist}_G(u, w)$.

Beweis: Nach Definition der Randapproximation gilt $d(w') \geq \mathrm{dist}_G(u, w')$ für alle $w' \in \mathrm{border}_G(U) =: W$. Also brauchen wir nur $d(w) \leq \mathrm{dist}_G(u, w)$ zu zeigen. Sei $P(u_0, ..., u_n)$ ein kürzester gewichteter Pfad von u zu w. Dann gilt $\|(u_0, ..., u_i)\|_G = \mathrm{dist}_G(u, u_i)$ für alle $i \leq n$. Außerdem gilt $\mathrm{dist}_G(u, u_i) \leq \mathrm{dist}_G(u, u_{i+1})$ 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:

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

Konsequenz: Wir vergrößern eine Menge $U \subseteq V_G$ schrittweise. U soll nur Knoten enthalten, für die wir die exakte Entfernung schon kennen. Wir nutzen obiges Lemma, um U zu erweitern, d.h. wir berechnen immer geeignete Randapproximationen.

Dazu: Ist d eine Randapproximation zu U und sind W und w wie oben, dann ist die Randapproximation zu $U' := U \cup \{w\}$ wie folgt gegeben: Für jedes $w' \in W' := \text{border}_G(U')$ mit $\{w,w'\} \in E_G$ gilt:

$$d'(w') = \min\{d(w'), d(w) + \nu(w,w')\}$$

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

Frage: Wie verwalten wir die Ränder und Randapproximationen?

Antwort: In einer entsprechenden Datenstruktur!

Frage: Wie sollte die Datenstruktur beschaffen sein?

Antwort: Sie sollte partielle Abbildungen $\{0,...,n-1\} \to \mathbb{R}_{\geq 0}$ verwalten. Operationen: Festellen auf leeren Definitions-Bereich, Auslesen eines Wertes, Hinzufügen eines neuen (Argument,Wert)-Paares bzw. Abändern eines Wertes zu einem kleineren Wert, Bestimmen einer Stelle mit minimalem Wert, Löschen des (Argument,Wert)-Paares mit minimalem Wert.

Definition: Ein ADT, der partielle Abbildungen $\{0,...,n-1\} \to \mathbb{R}_{\geq 0}$ verwaltet und die Operation $getValue(i)$, $update(i,k)$, $getMin()$, $removeMin()$, $isEmpty()$ unterstützt, heißt *aktualisierbare Prioritätsschlange*. Dabei bedeutet $update(i,k)$, daß der Zahl i der Wert k zugeordnet wird, falls der Wert von i undefiniert ist oder $\geq k$ ist.

Laufzeitanalyse:

- $O(n + t_i + t_u)$ für die Initialisierung, wenn t_i für die Laufzeit der Initialisierung einer $UpdatablePriorityQueue$ steht und t_u für einen *update*-Schritt
- $O(n \cdot t_m)$ für die äußere Schleife ohne die innere, wenn t_m die Laufzeit für $getMin$, $removeMin$ und $getValue$ bezeichnet

- $O(m \cdot t_u)$ für die Anzahl der Durchläufe der innereren Schleifen, wenn t_u für die Laufzeit von *update* steht

Insgesamt: $O(n + t_i + n \cdot t_m + (m+1) \cdot t_u)$.

Unterschiedliche Laufzeiten für unterschiedliche Implementierungen:

Implementierung	t_i	t_m	t_u	$t_{Dijkstra}$
Feld	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n+n+n^2+m) = O(n^2+m)$
Liste	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n^2 + m \cdot n)$
Spreizbäume	$\Theta(1)$	$\Theta^*(\log n)$	$\Theta^*(\log n)$	$O(n+1+n \cdot \log n + m \log n) = O((m+n) \cdot \log n)$
Fibonacci – Halden	$\Theta(1)$	$\Theta^*(\log n)$	$\Theta^*(1)$	$O(n \cdot \log n + m)$

Satz: Bei Implementierung der aktualisierbaren Prioritätsschlange durch folgende Möglichkeiten berechnet der Dijkstra-Algorithmus Entfernungen in gewichteten Graphen in folgender Zeit:

- Feld: $O(n^2 + m)$
- Liste: $O(n^2 + m \cdot n)$
- Spreizbäume: $O((m+n) \cdot \log n)$
- Fibonacci-Halden: $O(n \cdot \log n + m)$

6.6 Gerichtete Graphen

Definition: 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.

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

Definitionen: Ist G gerichtet, so heißen

$$d_G^{in}(u) = |\{\, v \mid (v,u) \in E_G\}| \text{ und } d_G^{out}(u) = |\{\, v \mid (u,v) \in E_G\}|$$

Eingangs- und *Ausgangsgrad* von u in G. Ein Pfad $P = (u_0, ..., u_n)$ in einem gerichteten Graphen heißt Zykel, wenn $n \geq 1$ und $u_0 = u_n$ gilt. Ein gerichteter Graph ist *stark zusammenhängend*, wenn jeder Knoten von jedem erreichbar ist. Entsprechend reden wir von starken Zusammenhangskomponenten, in Zeichen $SCC_G(u)$ bzw. $GSCC_G(u)$. Ein gerichteter azyklischer Graph heißt auch *DAG* (directed acyclig graph). Ein gerichter Graph G heißt *Baum*, wenn es einen verwurzelten Baum (T, s) gibt, so daß

$$V_G = V_T \text{ und } E_G = \{\, (u,v) \mid v \,\text{Sohn von}\, u \in (T,s)\}$$

Bemerkung: Die Lemmata, die wir bewiesen haben für ungerichtete Graphen, sind auch übertragbar. Auch die Algorithmen können einfach übernommen werden: Sowohl BFS als auch Diklstras Algorithmus können in gerichteten Graphen benutzt werden und berechnen auch (gewichtete) Entfernungen und (gerichtete) Entfernungsbäume.

6.7 Tiefensuche und topologische Sortierung

Definition: Sei G ein gerichteter Graph. Eine Folge $(v_0, ..., v_{n-1})$ heißt *topologische Sortierung* von G, wenn für alle i, j mit $0 \leq i < j < n$ gilt:

$$v_j \underset{G}{\rightsquigarrow} v_i \Rightarrow v_i \underset{G}{\rightsquigarrow} v_j$$

Definition: Sei G ein gerichteter Graph. Eine Folge $(v_0, ..., v_{n-1})$ heißt *schwache topologische Sortierung* von G, wenn für alle i, j mit $0 \leq i < j < n$ gilt:

$$v_j \underset{G}{\rightsquigarrow} v_i \Rightarrow (v_i \underset{G}{\rightsquigarrow} v_j) \ \vee \ (\ \exists\, k \ : \ 0 \leq k < i \text{ mit } v_k \underset{G}{\leftrightsquigarrow} v_i)$$

Folgerung: Sei G ein gerichteter azyklischer Graph. Dann ist jede schwache topologische Sortierung von G eine topologische Sortierung von G.

Beispiele:

- $[0\ 1]$ hat die topologischen Sortierungen $(0, 1)$ und $(1, 0)$
- $[0 \rightarrow 1]$ hat die topologischen Sortierungen $(0, 1)$, aber nicht $(1, 0)$
- $[0 \rightleftarrows 1]$ hat die topologischen Sortierungen $(0, 1)$ und $(1, 0)$

Folgerung: Ist G ein gerichteter azyklischer Graph und $(v_0, ..., v_{n-1})$ eine topologische Sortierung von G, so gilt:

$$\forall\, i, j \text{ mit } 0 \leq i < j < n \text{ gilt} : (v_j, v_i) \notin E_G$$

Motivation: Angenommen[19], 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 ausführbare Reihenfolge, sofern der Graph azyklisch ist (sonst kann man ohnehin nicht vernünftig arbeiten).

Ziel: Berechnung einer topologischen Sortierung

Ansatz: Wir benutzen die sogenannte *Tiefensuche*. Der Graph wird Knoten

[19] „... die Lehrbuch-Motivation...“

für Knoten erforscht, aber nicht zuerst alle Nachbarn eines Ausgangsknotens und dann deren Nachbarn usw., sondern zuerst nur einen Nachbarn, dann einen Nachbarn von diesem usw.

Wenn wir zu einem Knoten kommen, der keine neuen Nachbarn mehr hat, so gehört er ans Ende der topologischen Sortierungen. Wir besuchen dann einen Nachbarn des Vorgängers usw. Wir erhalten ein rekursives Verfahren.

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 $\Theta(|E_G| + |V_G|)$.

Beweis:

1. Wir geben geeignete Vor- und Nachbedingungen für $DFS - visit$ an. Zusätzliche Notation: $\text{reach}_G^W(u)$ Sei die Menge der von u in G über Knoten aus W erreichbare Knoten. Setze $W := \text{reach}_G^{V_w}(u)$.

   - **Vorbedingung:**
     - $|W| \leq i - 1$
     - $u \in V_w$
   - **Nachbedingung:**
     - $c[v] = \overline{c}[v]\ \forall\ v \notin W$
     - $s[j] = \overline{s}[j]\ \forall\ j$ mit $i < j < n$
     - $c[v] = black\ \forall\ v \in W$
     - $s[i+1\,..\,\overline{i}]$ ist schwache topologische Sortierung von $G[W]$ (wobei i den Wert bezeichne, den $DFS - visit$ zurückgibt).

   Wir beweisen dies durch Induktion über die Rekursionstiefe =: n (Anzahl geschachtelter Aufrufe).

   - Induktionsanfang: Für $n = 0$: Wenn innerhalb von $DFS - visit$ kein weiterer Aufruf von $DFS - visit$ erfolgt, dann sind alle Nachbarn von u nicht weiß, d.h. $W = \{u\}$, womit die Behauptung trivialerweise stimmt.
   - Induktionsschritt: Wir geben eine Invariante für die for-Schleife an, wobei $U = V_W$ (nach Ausführung von $c[u] = gray$), $r = a.length$ ist und
     $$V_j = reach_G^{V_W}(\{a[0], ..., a[j-1]\})$$
     Invariante:

- $c[v] = \overline{c}[v] \; \forall \; v \notin V_j$
- $c[v] = black \; \forall \; v \in V_j$
- $s[i+1\,..\,\overline{i}]$ ist eine schwache topologische Sortierung von $G[V_j]$
- $s[j] = \overline{s}[j] \; \forall \; j > \overline{i}$

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

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

(a) $s[k] \rightsquigarrow_{G'} s[j]$: Dann folgt aus der Schleifeninvariante $s[j] \rightsquigarrow_{G'} s[k]$ und damit 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 $s[l] \leftrightsquigarrow_{G''} s[k]$.

(b) sonst: Dann gilt $s[k] \rightsquigarrow_{G''} s[i+1] = u \rightsquigarrow_{G''} s[j]$. Andererseits gilt natürlich $u \rightsquigarrow_{G''} s[k]$.

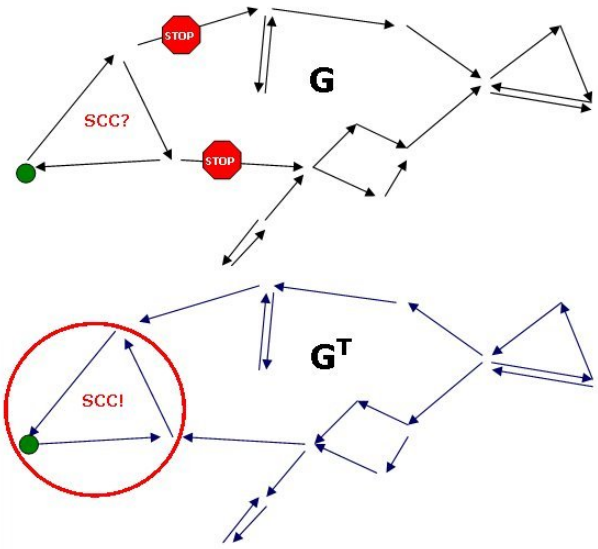
Damit ist die Behauptung bewiesen. Die Gültigkeit der obigen Invariante beweist man in ähnlicher Weise, wobei man $c[v] = white$ und $c[v] \neq white$ unterscheidet.

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 von G eine Kante gibt. Dann verhält sich $DFS - visit(u)$ wie $DFS(G)$.

2. Siehe Breitensuche (6.4).

6.8 Berechnung topologischer Sortierungen in beliebigen Graphen und starker Zusammenhangskomponenten

Idee:



Definition: Der zu einem gerichteten Graphen G *transponierte Graph* wird mit G^T bezeichnet und ist gegeben durch

$$G^T = (V_G, \{(v,u) \mid (u,v) \in E_G\})$$

Lemma: Sei G ein gerichteter endlicher Graph und u der erste Knoten einer schwachen topologischen Sortierung von G und $U := \text{reach}_{G_T}$ und $W = V_G \backslash U$. Dann gilt:

1. $SCC_G(u) = U$
2. Ist $(u_0, ..., u_{r-1})$ eine Aufzählung der Knoten von U und $(w_0, ..., w_{s-1})$ eine topologische Sortierung von $G[W]$, so ist $(u_0, ..., u_{r-1}, w_0, ..., w_{s-1})$ eine topologische Sortierung von G.
3. Entfernt man aus einer schwachen topologischen Sortierung von G die Elemente von U, so erhält man eine schwache topologische Sortierung von $G[W]$.
4. Die starken Zusammenhangs-Komponenten von G sind die starken Zusammenhangskomponenten von $G[W]$ und U.

Beweis: Sei $(v_0, ..., v_{n-1})$ eine schwache topologische Sortierung von G.

1. Sei $S = SCC_G(u)$. Wir zeigen: $S = U$.

   „$\subseteq$“ Es gilt $v \rightsquigarrow_G u$ für jeden Knoten $v \in S$, also $u \rightsquigarrow_{G^T} v$

„$\supseteq$“ Sei $v \in U$, $v \neq u$. Dann gilt $v \rightsquigarrow_G u$ und es gibt $i > 0$ mit $v = v_i$. Wegen $v_i \rightsquigarrow_G v_0 = u$ gilt dann auch $v_0 \rightsquigarrow_G v_i$, also $v_0 \leftrightsquigarrow_G v_i$, d.h. $v \in S$.

2. Wir unterscheiden drei Fälle:

   (a) $u_j \rightsquigarrow_G u_i$ mit $i < j$: Da $u_i, u_j \in SCC_G(u)$, gilt auch $u_i \rightsquigarrow_G u_j$.

   (b) $w_j \rightsquigarrow_G w_i$ mit $i < j$: Falls auch $w_j \rightsquigarrow_{G[W]} w_i$ gilt, so haben wir $w_i \rightsquigarrow_{G[W]} w_j$, also auch $w_i \rightsquigarrow_G w_j$. Falls nicht, so haben wir $w_j \rightsquigarrow_G u' \in U \rightsquigarrow_G w_i$, also auch $w_j \rightsquigarrow_G u \rightsquigarrow_G w_i$. Da u das erste Element einer schwachten topologischen Sortierung ist, folgt $u \rightsquigarrow_G w_j$, also $w_j \in U = SCC_G(u)$, Widerspruch!

   (c) $w_j \rightsquigarrow_G u_i$ mit $i < j$: Dann wissen wir wegen $u_i \leftrightsquigarrow_G u$ auch $u \rightsquigarrow_G w_j$ und damit auch $u \rightsquigarrow_G w_j$, Widerspruch!

Frage: Wie berechnen wir die von einem Knoten aus einem transponierten Graphen erreichbaren Knoten?

Antwort: Wir können z.B. $DFS - visit$ oder BFS oder entsprechende Vereinfachungen nehmen.

Frage: Wir transponieren wir einen Graphen?

Antwort: Das geht einfach, sowohl bei Repräsentation durch Adjazenzmatrix (Transponieren der Matrix: $\Theta(|V_G|^2)$) wie auch durch Adjazenzlisten ($\Theta(|V_G| + |E_G|)$)

Satz: Der Algorithmus $TopologicalSort$ berechnet eine topologische Sortierung eines Graphen in Zeit $\Theta(|V_G| + |E_G|)$.

Bemerkung: Der Algorithmus läßt sich abwandeln zu einem Algorithmus, der in der gleichen Zeit eine Liste der starken Zusammenhangskomponenten bestimmt.

Beweis: Die topologische Sortierung ist sogar einen *starke* topologische Sortierung im folgenden Sinn: wenn $v_j \rightsquigarrow v_i$ für $i < j$, gilt, so gilt sogar

$$v_i \leftrightsquigarrow v_{i+1} \leftrightsquigarrow ... \leftrightsquigarrow v_j$$

7 Aus- und Rückblick

Abstrakte Datentypen:

- Keller (stack) (2.1), Schlange (queue) (2.2)
- Liste (2.3), Positionsliste
- Prioritätsschlange (priority queue) (3.1)
- aktualisierbare Prioritätsschlange (updatable priority queue) (6.5)
- Wörterbücher (5.2), geordnete Wörterbücher (3.7)
- Bäume (3.1)
- Graphen (ungerichtet, gerichtet, gewichtet) (6.1)
- später: mehr!

Datenstrukturen:

- primitive Datenstrukturen
  - Feld (1.4)
  - Objekte
- lineare Datenstrukturen
  - Felder (1.4)
  - einfach verkettete Objekte (schon 1953 beim Hashing) (2.3)
  - doppelt verkettete Objekte (2.3)
- verzweigte/baumartige Datenstrukturen
  - Bäume (3.1)
  - Halde (3.6)
  - Suchbäume (3.8)
  - Spreizbäume (1985, Daniel Dominic Kaplan Sleator, CMU; Robert Endre Tarjan 30.04.1984, Princeton University, Turing-Award[20] 1986) (3.8.3)

[20] „Nobelpreis“ für Informatiker

  - später: AVL-Bäume, Binomialhalden, Fibonacci-Halden, B-Bäume, B^+-Bäume, B^*-Bäume, (n,k)-Bäume

- tabellenartige Datenstrukturen
  - Kettenbildung (1953, Hans Peter Luhn, 07.11.1896 - 1964)
  - offene Adressierung (1953, Gene M. Amdahl, Elaine M. Boehme, N. Rochester, Arthur L. Samuel)
  - universelles Hashing (1979, J. I. Carter, M. N. Wegman)

- Graphstrukturen
  - Adjazenzmatrizen
  - Adjazenzlisten (1973, John E. Hopcroft, 07.10.1939, Turing-Award 1986 zusammen mit Tarjan, Cornell University)

Algorithmen:[21]

- Algorithmen auf Datenstrukturen:
  - Verdoppeln bei Feldern
  - Haldenaufbau (1964, Robert W. Floyd, Turing-Award 1978): Anpassen von unten/oben (4.3)
  - ZZ-Umkrempeln (siehe Spreizbäume) (3.8.2)

- Sortieralgorithmen:
  - Sortieren durch Einfügen *InsertionSort* (4.2)
  - Sortieren durch Minimum-Finden *SelectionSort*? (siehe Übungen)
  - Sortieren durch Mischen *MergeSort* (1945, implementiert auf EDVAC durch (János) John von Neumann, 28.12.1903 - 08.02.1957) (4.6)
  - *QuickSort* (1962, Charles Antony Richard Hoare, 1948?, Turing-Award 1980, Oxford, auch verantwortlich für Korrektheitsregeln) (4.5)
  - später: Sortieralgorithmen für Parallelrechner

- Graphalgorithmen:

[21] Name stammt (wie *Algebra*) von *Abū' Abd Allāh Muhammed ibn Mūsā al – Khwārizmū* (um 825)

  - Tiefensuche (DFS) (6.7) (1973, John E. Hopcroft und Tarjan)
  - Breitensuche (BFS) (6.4) (1959, Edward F. Moore, 1961, C. Y. Lee)
  - topologische Sortierung (6.7)
  - starke Zusammenhangskomponenten (S. R. Kosaraju, M. Sharir) (6.8)
  - später: tausende von Graphproblemen und Graphprobleme, insbesondere schwierige Probleme, die nicht in polynomieller Zeit lösbar sind (wird zumindest vermutet)
- später:
  - parallele, verteilte, randomisierte, approximative Algorithmen etc.
  - Algorithmen für numerische, graphische, algebraische Probleme

Korrektheit:

- partielle, totale Korrektheit (4.2.1)
- Korrektheitsregeln (Hoare-Kalkül, Hoare) (4.2.1)
- Vor- und Nachbedingung, Invarianten
- später: Verifikation von Soft- und Hardware, automatischer Verifikation, Verifikation durch Theorembeweiser

Aufwandsabschätzungen:

- Komplexitätsbetrachtungen (1962, Donald E. Knuth, 01.10.1938, Stanford, Turing-Award 1974, TEX)
- Größenordnungen (1974, Alfred V. Aho, Columbia, Hopcroft, Jeffrey D. Ullman)
- pessimale, durchschnittliche, amortisierte, erwartete Laufzeiten
- Abschätzungen:
  - Summen:

$$\sum_{i=0}^{n-1} i \qquad \sum_{i=0}^{n-1} (n-i)2^i \qquad \sum_{i=0}^{n-1} \frac{1}{i}$$

– Rekursionsgleichungen:

$$t(n) = t\left(\frac{n}{2}\right) + c \qquad t(n) = at\left(\frac{n}{2}\right) + bn + c \qquad a_{k+2} = a_k + a_{k+1} + 1$$

– Fakultät: James Sterlings Formel (1730):

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

weitere wichtige Begriffe:

- Baumordnungen
- Potenzialmethode
- zufalls gesteuerter (randomisieter) Algorithmus

Beispiele:

1.
   - Vorbedingung: $\{x \in \mathbb{N}_0, y \in \mathbb{N}_0\}$
   - Programm:

```
setze z = 0
solange y > 0
    setze z = z + x
    setze y = y - 1
```

   - Nachbedingung: $\{z = \overline{x} \cdot \overline{y}\}$
   - Invariante: $(z = \overline{x} \cdot (\overline{y} - y)) \wedge (y \geq 0) \wedge (x = \overline{x})$
   - Es folgt: $y = 0 \Rightarrow z = \overline{x} \cdot \overline{y}$
   - Laufzeit: $\Theta(y)$

2.
   - Vorbedingung: $\{x \in \mathbb{N}_0, y \in \mathbb{N}_0\}$
   - Programm:

```
setze z = 0
solange y > 0
    falls y mod 2 = 0
        setze x = x * 2
        setze y = y div 2
    sonst
        setze z = z + x
        setze y = y - 1
```

   - Nachbedingung: $\{z = \overline{x} \cdot \overline{y}\}$
   - Invariante: $(\overline{x} \cdot \overline{y} = z + x \cdot y) \wedge (y \geq 0)$
   - Es folgt: $y = 0 \Rightarrow z = \overline{x} \cdot \overline{y}$

- Laufzeit[22]: $\Theta(\log n)$

3.
- Vorbedingung: $\{a[0\,..\,n-1]\text{ist Feld}, n > 0\}$
- Programm: $\Theta(y)$

```
fuer i = 0 bis (n-1) div 2
    swap(a, i n-i-1)
```

- Nachbedingung: $\{a[i] = \overline{a}[n-i-1]\ \forall\ i < n\}$
- Invariante:

$$\Theta(i) = \left[a[j] = \begin{cases} \overline{a}[n-j-1] & \text{falls} \quad (j < i) \vee (j > n-i-1) \\ \overline{a}[j] & \text{falls} \quad (i \leq j \leq n-i-1) \end{cases} \right]$$

- Aus $\Theta((n-1)\operatorname{div} 2 + 1)$ folgt die Nachbedingung.
- Laufzeit: $\Theta(n)$

4.
- Vorbedingung:
- Programm:

```
--- P0 ---
fuer j = 0 bis N-1
    setze c[j] = 0
--- P1 ---
fuer i = 0 bis n-1
    setze c[a[i]] = c[a[i]] + 1
--- P2 ---
setze d[0] = 0
fuer j = 1 bis N-1
    setze d[j] = c[j-1] + d[j-1]
--- P3 ---
fuer i = 0 bis n-1
    setze b[d[a[i]]] = a[i]
    setze d[a[i]] = d[a[i]] + 1
--- P4 ---
```

- Nachbedingung:
- Hilfsdefinition: $\text{count}(i, n) := |\{j < n \mid \overline{a}[j] = i\}|$
- Bedingungen:

(P0) $(a[0\,..\,n-1]\text{ist Feld}) \wedge (n > 0) \wedge (a[i] \in \{0..., N-1\})$
(P1) $(c[j] = 0\ \forall\ j < N) \wedge (a = \overline{a})$
(P2) $(c[j] = \text{count}(j, n)\ \forall\ j < N) \wedge (a = \overline{a})$
(P3) $(d[j] = \sum_{i=0}^{j-1} count(j, n)) \wedge (a = \overline{a})$
(P4) $\text{Perm}(b, n) \wedge \text{Ord}(a, n)$

- Hilfsdefinition: $\text{Count}(j) := \sum_{i=0}^{j-1} \text{count}(i)$

[22] genaue Anzahl der Durchläufe der Schleife: Anzahl der Einsen in Binärdarstellung plus Länge in der Binärdarstellung von y

- Invarianten:

  (P3) Definiere $c := b\big[0\,..\,d[0]-1\,...\,Count(N-1)\,..\,d[N-1]-1\big]$. Dann gilt:

  $$d[j] = \text{Count}(j) + \text{count}(i,j) \wedge \text{Perm}(c, i-1) \wedge \text{Ord}(c, i-1) \wedge (a = \overline{a})$$

- Laufzeit: $\Theta(N+n)$

8 Organisatorisches

ausgenommen von der Klausur sind:[23]

1. die Korrektheitsregeln (es können nur z.B. Invarianten vorkommen)
2. Beweis über die amortisierte Laufzeit bei Spreizbäumen
3. Beweis über durchschnittliche Laufzeit von *QuickSort*

Anmeldung zum Softwarepraktikum:

`http://www.ti.informatik.uni-kiel.de/Praktikum`

Evaluation der Vorlesung: nur mit Zugangscode, der in der Vorlesung verteilt wurde:

`http://www.ti.informatik.uni-kiel.de/evaluation/`

[23] natürlich keine Gewähr etc., aber dies sagte Thomas Wilke (wörtlich) in der Vorlesung

Index