Vorlesungsmitschrift zu Algorithmen und Programmierung 3

Naja v. Schmude
www.najas-corner.de/wordpress

13. Februar 2008

Inhaltsverzeichnis

1. Vorlesung, 16.10.2007

1.1. Organisatorisches

zum Thema Datenstrukturen gibt es gute Bücher von Goodwich, Tamissia, Weiss, Sedigewick usw.

Inhalt der Vorlesung

- Analyse von Algorithmen
- Sortierverfahren
- Datenabstraktionen, die durch Datenstrukturen realisiert werden
- Realisierung der Datenstrukturen in Java
- Kennenlernen von verschiedenen abstrakten Datentypen und zugehörigen Datenstrukturen wie z.B. Listen, Warteschlangen, Wörterbuchproblem
- Algorithmen auf Graphen
- schwere Probleme, dies sind Probleme, für die man (noch) keine schnellen Algorithmen kennt

1.2. Kapitel 1: Analyse von Algorithmen und Sortierverfahren

Problem: Gegeben ist eine Folge von n Objekten, die paarweise miteinander vergleichbar sind. Wir haben also eine lineare Ordnung.
Ziel: Wir wollen die Folgen aufsteigend sortiert ausgeben.

Jetzt kommen vier Sortieralgorithmen

1. (Slowsort) Wir konstruieren systematisch alle möglichen Reihenfolgen (Permutationen) der Elemente und für jede prüfen wir nach, ob die Elemente aufsteigend sortiert sind. Wenn ja, dann wird diese Reihenfolge ausgegeben.

2. (Selectionsort) Man durchlauft die Folge und finde das Maximum. Dann speichern wir dieses an die letzte Stelle. Dann wird die Restfolge, falls sie noch vorhanden ist, auf die gleiche Art.

3. (Mergesort) Falls die Länge $n = 1$ gib das Element aus. Anderenfalls teile die Folge in zwei Folgen der halben Länge. Dann werden die Teilfolgen rekursiv sortiert. Dann werden die sortieren Teilfolgen nach dem Reißverchlussverfahren gemischt zu einer sortierten Gesamtfolge.

4. (Quicksort) Falls $n = 1$, dann gib das Element zurück. Ansonsten wählen wir ein Element a aus der Folge aus. Jetzt werden alle restlichen Elemente durchlaufen und teile sie in die Elemente $< a$ (Folge S_1) und die $\geq a$ (Folge S_2) ein. Sortiere nun S_1 rekursiv und gib es aus. Dann wird a ausgegeben und dann muss noch S_2 rekursiv sortiert und ausgegeben werden.

Jetzt stellt sich die Frage, welcher Algorithmus der beste ist und wie gut diese sind?
Erstmal ein paar Kriterien, die die Güte eines Algorithmuses angeben könnten: z.B. kurze Laufzeit, geringer Platzverbrauch. Diese Kriterien werden als Funktion der Größe n der Eingabe verstanden. Unabhängig von der Implementierung und des speziellen Rechners ist z.B. die Anzahl der benötigten Vergleiche $C(n)$ bei einer Eingabe der Länge n. Dies ist übrigens auch proportional zur reellen Laufzeit ($O(C(n))$.

Jetzt zur Analyse der Sortieralgorithmen

1. Im schlechtesten Fall gibt es hier $n!$ Permutationen, die alle durchlaufen werden müssen. Man muss für jede Permutation mindestens einen Vergleich durchführen. Wir haben also $> n!$ Vergleiche im schlechtesten Fall. Das Mittel würde immer noch bei $\frac{n!}{2}$ liegen, was auch nicht so viel besser ist.

2. Man benötigt $n-1$ Vergleiche zum finden des Maximum. Im nächsten Duchlauf werden $n-2$ Vergleiche benötigt usw. Insgesamt also $\sum_{i=1}^{n-1} i = \frac{(n-1)(n-2)}{2}$. Also $O(n^2)$.

3. Hier ist das schwieriger. Um hier $C(n)$ zu finden, stellen wir ein paar Eigenschaften auf. Z.B. ist $C(1) = 0$. Außerdem ist durch die rekursiven Aufrufe $C(n) = 2C(\frac{n}{2}) + M(n)$, wobei $M(n)$ die Anzahl der Vergleiche repräsentiert, um zwei Folgen der Länge $\frac{n}{2}$ zu einer Folge der Länge n zu mischen. Es gilt $\frac{n}{2} \leq M(n) \leq n-1$. Also $C(n) \leq 2C(\frac{n}{2})+n$. Einfachheitshalber nehmen wir an, dass n eine Zweierpotenz ist, d.h. $n = 2^k, k \in \mathbb{N}$.

$$
\begin{aligned}
C(n) &\leq 2C(\frac{n}{2}) + n \\
&\leq 4C(\frac{n}{4}) + n + n \\
&\leq 2^l C(\frac{n}{2^l}) + l \cdot n, \forall l \leq k \\
&\leq 2^k C(1) + k \cdot n, l = k \\
&\leq n \log_2 n
\end{aligned}
$$

4. Noch schwieriger als Mergesort. Als erstes muss das Pivotelement gewählt werden. Eine Strategie wäre, das erste Element zu nehmen, eine bessere wäre zufällig eins zu wählen. Andererseits kann man drei zufällige nehmen und dann von denen das mittlere. Was ist nun die Anzahl der Vergleiche im schlechtesten Fall? Dieser tritt ein, wenn a bereits das kleinste ist, denn dann ist $S_1 = \emptyset$ und $|S_2| = n$. Bei S_2 ist dann auch immer das Pivotelement das kleinste usw. Für das Aufteilen benötigt man $n - 1$ Vergleiche, dann $n - 2$ usw. Hier also auch wieder $\frac{(n-1)(n-2)}{2}$ Vergleiche.
   Im Mittel ist Quicksort wesentlich besser als Mergesort, der schlechteste Fall tritt nämlich so gut wie nie ein. Der schlechteste Fall tritt insbesondere auf, wenn man als Pivotelement a immer das erste Element nimmt und die Folge bereits sortiert war. Allerdings im Mittel über alle möglichen Eingaben hat man $O(n \log n)$ Vergleiche. Die Analyse dafür ist allerdings anspruchsvoll.

2. Vorlesung, 18.10.2007

2.0.1. Analyse von Quicksort mit zufälliger Wahl des Pivotelements

Im schlechtesten Fall, wie letzte Vorlesung gesagt, benötigt man $\frac{1}{2}(n(n-1))$ Vergleiche.
Im Mittel ("erwartet") ist die Zahl der Vergleiche bei Eingabe der Länge n eine Zufallsvariable.
Wir wollen den Erwartungswert $C(n)$ bestimmen.

$$C(0) = 0$$
$$C(1) = 0$$
$$C(n) = \overbrace{\frac{1}{n}}^{\text{Wk. } k \text{ kl. Elem}} \sum_{k=1}^{n} (\underbrace{C(k-1) + C(n-k)}_{\text{rek. Aufrufe, falls } a \text{ k-kl. Elem}} + \overbrace{n-1}^{\text{Folge halbieren}})$$
$$n * C(n) = \sum_{k=1}^{n} C(k-1) + \sum_{k=1}^{n} C(n-k) + n(n-1) \quad \text{Summen zählen das Gleiche}$$
$$nC(n) = 2\sum_{k=0}^{n-1} C(k) + n(n-1) \quad \text{für alle } n > 2$$
$$(n-1)C(n-1) = 2\sum_{k=0}^{n-2} C(k) + (n-1)(n-2)$$

Wir subtrahieren die letzten beiden Formeln und erhalten:

$$nC(n) - (n-1)C(n-1) = 2C(n-1) + 2(n-1)$$

$$
\begin{aligned}
nC(n) &= (n+1)C(n-1) + 2(n-1) \\
C(n) &= \frac{n+1}{n}C(n-1) + \frac{2(n-1)}{n} \\
C(n) &\leq \frac{n+1}{n}C(n-1) + 2 \\
&\leq \frac{n}{n-1}C(n-2) + 2 \\
&\leq \frac{n+1}{n}\frac{n}{n-1}C(n-2) + 2(\frac{n+1}{n+1} + \frac{n+1}{n}) \\
&\leq \frac{n+1}{n-1}\frac{n-1}{n-2}C(n-3) + 2(\frac{n+1}{n+1} + \frac{n+1}{n} + \frac{n+1}{n-1}) \\
&\dots \\
&\leq \frac{n+1}{n-r}C(n-r-1) + 2(n+1)\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{n-r+1}\right)
\end{aligned}
$$

Für $r = n-1$:

$$C(n) \leq \frac{n+1}{1}C(0) + 2(n+1)\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right)$$

$$H_k = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}$$

heißt die $k$te Harmonische Zahl. Damit ist

$$C(n) \leq 2(n+1)(H_{n+1} - 1)$$

Die harmonische Zahl wächst übrigens proportional zum Logarithmus. Dies kann man einsehen, wenn man Rechtecke malt, die die Breite 1 haben und deren höhe von 1 immer abnimmt zu $\frac{1}{2}, \frac{1}{3}$ usw. Einleutenderweise ist diese Fläche der Quadrate kleiner als das Integral unter der Funktion $f(x) = \frac{1}{x}$.

$$H_k = 1 + \int_1^k \frac{1}{x}dx = 1 + \ln k$$

$$
\begin{aligned}
C(n) &= 2(n+1)\ln(n+1) \\
C(n) &= 2(n+1)\frac{2}{\log_2 e}(n+1)\log_2(n+1) \\
&\approx 1,38(n+1)\log_2(n+1)
\end{aligned}
$$

Diese Analyse gilt übrigens auch für die Methode, dass man das erste Element der Folge als Pivotelement wählt.
Falls jede Permutation als Eingabe gleichwahrscheinlich, so erhält ? diese Anzahl von Vergleichen als Mittelwert über alle Eingaben.
Die letzte Variante, aus drei zufälligen Elementen das mittlere zu nehmen, ist wie die erste Methode. Nur die Konstante, wird dadurch verbessert.

2.0.2. Vergleich der Funktionen

n	$n \log n$	$\frac{1}{2}n(n-1)$	$n!$
10	33	45	$3,6 * 10^6$
20	86	190	$2,4 * 10^{18}$
30	282	1225	$3,0 * 10^{64}$

Eine Maschine mit 10^9 Vergleichen pro Sekunde. Jetzt wird aufgeführt, wie viele Probleme in welcher Zeit gelöst werden können

1 sec	$4 * 10^7$	$4 * 10^4$	11
1 h	$1 * 10^{11}$	$2,6 * 10^6$	14

Bei einer 10x schnelleren Maschine sähe das wie folgt aus:

1 sec	$3,5 * 10^8$	10^5	12
1 h	$9,1 * 10^{11}$	$8,4 * 10^6$	15

2.0.3. Implementierung in Java

Für Arrays von ganzen Zahlen.

```
1  public class Sortieren {
2
3          public static void einfSort(int[] a) {
4
5                  int j, t;
6
7                  for (int i = 1; i <= a.length - 1; i++)
8
9                  {
10                         j = i;
11                         while (j >= 1 && a[j - 1] > a[j]) {
12                                 t = a[j];
13                                 a[j] = a[j - 1];
14                                 a[j - 1] = t;
15                                 j--;
16                         }
17                 }
18
19         }
20
21         public static void mische(int[] a, int al, int ar, int
               [] b, int bl, int br,
22                         int[] c, int cl) {
23
24                 // mischt ein sortiertes array-Segment a[al
                       ]...[ar] mit
```

```
25              // b[bl]..b[br] zu einem sortierten Segment c[
                   cl] ...
26
27              int i = al, j = bl;
28              for (int k = cl; k <= cl + ar - al + br - bl +
                   1; k++) {
29                      if (i > ar) // a abgearbeitet
30                      {
31                              c[k] = b[j++];
32                              continue;
33                      }
34
35                      if (j > br) // b abgearbeitet
36                      {
37                              c[k] = a[i++];
38                              continue;
39                      }
40
41                      c[k] = (a[i] < b[j]) ? a[i++] : b[j++];
42              }
43      }
44
45      public static void mergeSort(int[] a, int al, int ar) {
46
47              if (ar > al) {
48                      int m = (ar + al) / 2;
49
50                      // Rekursives Sortieren der H?lften:
51
52                      mergeSort(a, al, m);
53                      mergeSort(a, m + 1, ar);
54
55                      // Mischen ins Array b :
56
57                      int[] b = new int[ar - al + 1];
58                      mische(a, al, m, a, m + 1, ar, b, 0);
59
60                      // Zur¸ckspeichern:
61
62                      for (int i = 0; i < ar - al + 1; i++)
63                              a[al + i] = b[i];
64
65              }
66      }
67
68      public static void tausche(int[] a, int i, int j) {
```

```
69                int t = a[i];
70                a[i] = a[j];
71                a[j] = t;
72        }
73
74        public static void quickSort(int[] a, int al, int ar) {
75                // sortiert das Segment a[al],...,a[ar]
76
77                if (al < ar) {
78                        int pivot = a[al], // 1. Element als
                             Pivotelement
79                        i = al, j = ar + 1;
80
81                        // Aufspalten
82
83                        while (true) {
84                                while (a[++i] < pivot && i < ar
                                   ) {
85                                }
86                                while (a[--j] > pivot && j > al
                                   ) {
87                                }
88
89                                if (i < j)
90                                        tausche(a, i, j);
91                                else
92                                        break;
93                        }
94
95                        tausche(a, j, al);
96
97                        quickSort(a, al, j - 1);
98                        quickSort(a, j + 1, ar);
99
100               }
101       }
102
103 }
```

3. Vorlesung, 23.10.2007

Welche Lehren können aus den vorherigen Beispielen gezogen werden?

- Ein Algorithmus kann auch verbal formuliert werden, anstatt ein Programm zu schreiben. Auch in dieser Form ist eine Analyse möglich, zumindest der Laufzeit in Form von O.
- Die Analyse ist wichtig!
- "divide and conquer"-Methode ist eine Entwurfsstrategie. Dabei teilt man das Problem in mehrere Probleme kleinerer Größe. Diese kleineren werden rekursiv gelöst und dann zur Gesamtlösung zusammengebaut.
  MergeSort und QuickSort verwenden diese Methodik.
  Die Analyse von divide and conquer führt zu Rekursionsgleichungen für die Laufzeit, die gelöst werden müssen.
- Die Laufzeit für gewisse Problemgröße n kann sich im schlechtesten Fall und im Mittel unterscheiden. Das Beispiel wäre QuickSort, wo im schlechtesten Fall ca. $\frac{1}{2}n^2$ und im Mittel was mit dem Logarithmus.
- Es gibt Algorithmen, deren Ablauf vom Zufall abhängt. Dies sind die sogenannten randomisierten Algorithmen. So gibt es z.B. ein randomisiertes QuickSort. Hier muss in der Laufzeitanalyse der Erwartungswert berücksichtigt werden.

3.0.4. Allgemeines über Algorithmen und ihre Analyse

Algorithmus Ein Verfahren, das sich mechanisieren lässt, zur Lösung eines Problems.

Beschreibung eines Algorithmus

- Die Beschreibung kann natürlich wieder umgangssprachlich sein
- oder durch ein Programm in einer imperativen Programmiersprache
- Eine dritte Möglichkeit wäre eine Zwischenstufe von beiden ("Pseudocode").

3.0.5. mathematische Formalisierung

Die erste Formalisierung war die Turingmaschine, siehe Grundlagen der theoretischen Informatik.
Danach kam die Registermaschine. Diese besteht aus unendlich vielen Zellen oder Registern als Speicher, die durchnummeriert sind. Jedes dieser Register kann eine natürliche Zahl enthalten. Dazu enthält die Registermaschine ein Programm aus Assembler ähnlichen Befehlen. Die Eingabe wird in die ersten Register dann geschrieben und die Ausgabe dann weiter hinten.
Eine andere Formalisierung wäre ein Programm in einer höheren Sprache, deren Semantik formal definiert ist.

3.0.6. Analyse

Wie viele Ressourcen (wie Laufzeit, Speicherplatz etc.) benötigt ein Algorithmus als Funktion der Eingabegröße n?. Für jede der genannten Modelle muss natürlich genauer definiert werden, was mit den Ressourcen gemeint ist. Bei Turing entspricht z.B. ein Schritt einer Zeiteinheit und eine Zelle des Bandes einer Platzeinheit.
Für die Registermaschine ist dies schwieriger: Hier sagt man eine Zeiteinheit entspricht der Ausführung eines Befehls und eine Platzeinheit ist wieder ein Register. Diese Einteilung nennt man Einheitskostenmaß.
Allternativ gibt es das sogenannte logarithmische Kostenmaß. Hier kostet ein Befehl k Zeiteinheiten, wobei k die Gesamtgröße der Binärdarstellungen der manipulierten Werte ist. Ein Register kostet k Platzeinheiten, wobei hier das k die Größe der Binardarstellung vom gespeicherten Wert angibt.
Die Laufzeit eines Programms ist dann die Summe aller Zeiteinheiten für alle Befehle. Der Platzbedarf ergibt sich als insgesamt genutzte Register. Diese hängen von der Eingabe direkt ab, nicht nur von ihrer Größe n.
Die Ressourcen sind schon oft aus der verbalen Beschreibung (in Form von O) analysierbar. Schleifen liefern z.B. eine Summation über alle Iterationen. Bei Rekursionen, wie schon erwähnt, ergibt sich eine Rekursionsgleichung.

Achtung Es gibt versteckten Platzbedarf bei der Rekursion, nämlich den Laufzeit-Stack bei der Abarbeitung! Pro Aufruf wird ein Datensatz auf den Stack gelegt. Unter anderem Platz für lokale Variablen, Rücksprungadresse etc. Die Größe des Stacks ist die Verschachtelungstiefe der Rekursion zur Laufzeit.

3.0.7. Wachstum von Funktionen

Wir haben die Funktion $f_1(n) = n!$, $f_2(n) = \frac{1}{2}n(n-1)$ und $f_3(n) = n\lceil \log n \rceil$. f_1 wächst wesentlich stärker als f_2, dieses stärker als f_3.

O-Notation Die O-Notation gibt die obere Schranke für das Wachstum unter Vernachlässigung multiplikativer Konstanten an. $f, g : \mathbb{N} \to \mathbb{N}$

$f = O(g)$ heißt $\exists c : f(n) \leq c * g(n)$

$f = o(g)$ heißt $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f = \Theta(g)$ heißt $f = O(g) \wedge g = O(f)$

Allgemein: $(\log n)^k = o(n^k)$ für $k > 0, n > 0$. Die Laufzeit von SelectionSort ist $\frac{1}{2}n(n-1) = \Theta(n^2)$.

3.0.8. Liste von Funktionen

$O(1)$	durch eine Konstante beschränkte Funktion
$\log\log n$	Interpolationssuche
$\log n$	logarithmische Laufzeit, z.B. die Binärsuche
$\log^k n, k \in \mathbb{N}$	polylogarithmische Laufzeit
$n^\alpha, 0 < \alpha < 1$	
n	lineare Laufzeit
$n \log n$	
n^2	quadratisches Wachstum
n^3	kubisches Wachstum
n^k	polynomielles Wachstum
c^n	exponentielles Wachstum
$n! = O(2^{n \log n})$	
2^{n^2}	
2^{2^n}	doppelt exponentiell

4. Vorlesung, 25.10.2007

Auswählen beschreibt den Vorgang, bei dem aus einer Liste S von Zahlen das k-kleinste bestimmt werden soll. $k = 1$ ist das Minimum, $k = |S|$ das Maximum und $k = \lfloor \frac{|S|}{2} \rfloor$ nennt man den Median.

4.0.9. Quickselect

Quickselect übernimmt dieses Auswählen nach dem "divide & conquer"-Prinzip und ist analog zu Quicksort mit zufälliger Wahl des Pivotelements. Außerdem klappt das ganze dann auch in linearer Zeit.

Pseudocode

```
  Eingabe: Folge S mit n = |S| Zahlen, 1<=k<=n
  Ausgabe: k-kleinste Element

quickSelect(S,k)
  if n=1 then return das Element von S;
  nimm ein zufälliges Element a aus S (Pivotelement) und zerlege S in drei
  Teile, S<, S=, S> mit S< = Teilliste der Elemente, die kleiner als a sind usw;

  if k <= |S<| then quickSelect(S<,k);
  else if k<= |S<|+|S=| then return a;
  else quickSelect(S>,k-|S<|-|S=|);
```

Der Aufwand in einem Aufruf von quickSelet(S,k) exklusive der rekursiven Aufrufe ist $O(n)$.

Annahme Wir nehmen an, a ist das i-kleinste Element. Sei $T(n)$ eine obere Schranke für die Laufzeit für quickSelect mit $|S| = n$. Daraus ergibt sich die Rekursion

$$T(n) = \underbrace{O(n)}_{\leq c*n} + T()$$

Behauptung $T(n) \leq k * n$ für eine Konstante k, die sich noch aus der Rechnung ergeben wird.

$$\begin{aligned} T(n) &\leq c * n + \max(T(|S_<|), T(|S_>|) \\ &\leq cn + \max(T(i-1), T(n-i)) \end{aligned}$$

Wir berechnen den Erwartungswert

$$E(T(n)) \leq cn + \frac{1}{n} \sum_{i=1}^{n} [\max(E(T(i-1)), E(T(n-i))]$$

Behauptung: $E(T(n)) \leq k * n$. Der Beweis hierfür funktioniert per vollständiger Induktion über n.

Induktionsanfang n=1 $T(1) \leq c * 1 + 0 \leq k * 1$ für $k > c$

Induktionsschritt Annahme: Für alle $n' < n$ ist die Behauptung $E(T(n)) \leq k * n'$ gültig.

$$\begin{aligned} E(T(N)) &\leq c * n + \frac{1}{n} \sum_{i=1}^{n} \max(k(i-1), k(n-i)) \\ &\leq cn + \frac{1}{n} k \underbrace{\sum_{i=1}^{n} \max(i-1, n-i)}_{\leq n^2 \frac{3}{4}} \\ &\leq cn * \frac{1}{n} k n^2 \frac{3}{4} + cn + \frac{3}{4} kn \\ &= n(c + \frac{3}{4}k) \leq n * k \end{aligned}$$

Mit $k > c$ lässt sich die Behauptung durch Induktion beweisen.

Jetzt nochmal genauer zur Summe:

$$\sum_{i=1}^{n} \max(i-1, n-i) = \sum_{i=1}^{\frac{n+2}{2}} (n-i) + \sum_{\frac{n+i}{2}}^{n} (i-1)| \text{ Substitution } i = n+1-j$$

Jetzt verschieben sich die Grenzen der Summen zu

$$\sum_{i=1}^{\frac{n+1}{2}} (n-i) + \sum_{j=1}^{n-\frac{n-1}{2}} (n-j) = n^2 - \underbrace{(\sum_{i=1}^{\frac{?}{?}} i + \sum_{j=1}^{n-\frac{n-1}{2}} j)}_{S} = n^2 - \frac{n^2}{4} = \frac{3}{4} n^2$$

Fall 1 n ist $2t, t \in \mathbb{N}$. Arg, da bin ich nicht mehr hinterher gekommen. ...

Fall 2 n ist $2t-1$. $\frac{n+1}{2} = \frac{2t}{2} = t$.

$$S = \sum_{i=1}^{t} i + \sum_{j=1}^{t-1} j = \frac{t(t-1)}{2} + \frac{t(t-1)}{2} = \frac{1}{2}t^2 \geq (\frac{n}{2})^2 = \frac{n^2}{4}$$

Anmerkung: Kein Verlass auf meine Mitschrift. Gerade bei den Grenzen der Summen und so wurde der liebe Herr so klein, dass ichs nicht mehr so recht entziffern konnte. Außerdem hab ich eh nicht wirklich verstanden, was er da macht, sorry.
Für Aufklärung und Berichtigung wäre ich dankbar!

Satz 1. *QuickSelect mit zufälliger Wahl des Pivotelements bestimmt das k größte Element in erwarteter Laufzeit $O(n)$.*

4.0.10. Sortieren durch Fachverteilung (bucket-sort)

Buckesort ist geeignet, wenn die zu sortierenden Werte aus einem kleinen Bereich kommen. Z.B. ganze Zahlen von 0 bis U oder die Buchstaben des Alphabets.

Wie macht man das nun? Wir haben erstmal als Datenstruktur ein Feld $B[0 \ldots U]$ von verketteten Listen. Am Anfang ist die Bucketliste leer.
Schritt 1: Verteilen. Durchlaufe die Eingabeliste, füge jedes Element mit Schlüssel i am Ende von $B[i]$ ein.
Schritt 2: Aufsammeln. for $i = 1 \ldots u$ gib die Elemente von $B[i]$ aus.

Die Laufzeit für Schritt 1 ist $O(n)$. Für Schritt 2 benötigt man eine Laufzeit von $O(U)+O(n)$. Der Speicherbedarf ist $O(U)$ für das Feld $+ \sum_{i=1}^{U}$ Länge $(B[i]) * O(U+n)$

Satz 2. *n Elemente mit ganzzahligen Schlüsseln im Bereich $0 \leq i \leq U$ können durch Fachverteilung in $O(n+U)$ Zeit und $O(n+U)$ Speicher stabil sortiert werden.*

Definition 1 (stabil). *Ein Sortierverfahren heißt stabil, wenn Elemente mit gleichem Schlüssel in der Ausgabe die gleiche Reihenfolge habe, wie in der Eingabe.*

5. Vorlesung, 30.10.2007

Es sind $\Theta(n \log n)$ Vergleiche notwendig bei vergleichsbasierten Sortierverfahren.
Bucketsort läuft in Zeit $O(n)$, es ist aber nicht vergleichsbasiert!

5.0.11. Radixsort

Damit wird das Sortieren von Wörtern über einem endlichen Alphabet Σ, auf dem eine lineare Ordnung $\leq$ gegeben ist, bezeichnet. Diese Ordnung nennt man auch die lexikographische Ordnung.

Definition 2 (lexikographische Ordnung). *Funktioniert wie in einem Telefonbuch. Erweiterung von $\leq$ von Σ auf Σ^* durch eine rekursive Definition:*

$\epsilon \leq u$ *für alle* $u \in \Sigma^*$
$au \leq bv \Leftrightarrow a < b \vee (a = b \wedge u < v)$

Als Beispiele kann man z.B. das Binäralphabet ($\Sigma = \{0, 1\}$), das Dezimale Alphabet oder das ganz normale lateinische Alphabet nehmen.
Allgemeiner: $\Sigma = \{0, \ldots, R-1\}$ mit R heißt Basis (Radix)

Wie werden Briefe nach Postleitzahlen sortiert? Da gibt es mehrere Möglichkeiten:

- Fachverteilung bezüglich 1. Ziffer, danach bezüglich der 2. usw. Das würde dann 10^5 Fächer benötigen. Allgemein also R^k Fächer bei Strings der Länge k über dem Alphabet $\Sigma = \{0, \ldots, R-1\}$.
  Dieses Verfahren nennt sich MSD-Radix-Sort. Das MSD steht für "most significant digit". Dieses Verfahren ist aber öfters zu aufwändig.

- Eine Alternative bildet das LSD-Radixsort. LSD kann man sich ja schon denken steht für "least significant digit".
  Die Eingabe ist hier eine Liste von Wörtern $w_1, \ldots, w_n$ die alle die gleiche Länge k haben. Danach soll dann die Folge lexikographisch sortiert sein.
  Und so funktioniert das Ganze:

```
for j=1 to k
  - sortiere Strings der Folge L mit Bucketsort bezüglich j letztem
    Buchstaben
  - hänge die Folgen in der nach Fächern sortierten Reihenfolge
    zusammen
```

Beispiel Man schaue sich das mal genauer mit der Sortierung der Wochentage an ...

```
Liste   letzter Buchstabe     vorletzter           erster
-------------------------------------------------------------
MON     E: DIE, FRE          A: SAM               D: DIE, DON
DIE
MIT     M: SAM               I: DIE, MIT          F: FRE
DON
FRE     N: MON, DON, SON     O: MON, DON, SON     M: MIT, MON
SAM
SON     T: MIT               R: FRE               S: SAM, SON
```

Die Korrektheit hier beruht auf der Korrektheit des Bucketsort.

Behauptung Für $j = 0, 1 \ldots, k$ gilt: Nach j Iterationen der for-Schleife sind die Wörter $w_1 \ldots w_n$ der Liste L lexikographisch sortiert bezüglich ihrer Suffizes der Länge j.

Beweis Wir führen eine Induktion über j.

Induktsionsanfang $j = 0$ Alle Suffizes der Länge 0 sind ϵ. Damit ist die Folge also bezüglich der Suffizes sortiert.

Induktionsschritt Nach $j + 1$ter Iteration seien w', w'' zwei aufeinander folgende Wörter der Liste. Und seien au' und bu'' Suffizes der Länge $j + 1$ von w', w''
Man unterscheidet zwei Fälle

1. **Fall** $a \neq b$. Dann muss aber wegen des ersten Bucketsorts gelten $a < b$. Damit ist $au' \leq bu''$.
2. **Fall** $a = b$. Dann kamen w' und w'' beim $j + 1$. Bucketsort in das gleiche Fach und zwar w' vor w''. Deswegen müssen u' und u'' Suffizes der Länge j von w' und w'' sein. Nach der Induktionsvoraussetzung gilt $u' \leq u'' \Rightarrow au' \leq bu''$ nach der Definition der lexikographischen Ordnung.

Anmerkung Die lexikographische Ordnung auf natürlichen Zahlen stimmt nicht überein mit der üblichen Ordnung. So ist z.B. nach normaler Ordnung $3 \leq 12$ aber nicht nach lexikographischer Ordnung. Eine Lösung wäre hier, mit Nullen von links aufzufüllen, bis alle die selbe Länge haben.
Bei lateinischem Alphabet gäbe es die Möglichkeit mit einem Sondereiche (#) rechts aufzufüllen, bis alle Wörter die gleiche Länge haben. Im Allgemeinen wäre das aber zu aufwendig. Statt dessen nimmt man den normalen Radixalgorithmus und sortieren mit Bucketsort die Wörter ihrer Länge nach. Dabei entstehen Listen $L_q, L_{q-1}, \ldots$ mit allen Wörtern der Länge q usw., wobei q die maximale Länge ist.
Bucketsort wird nun angewandt auf die Liste L_q. Nach der i ten Iteration wird L_{q-i} vorne an die Liste rangehangen.

Laufzeitanalyse

Alle Wörter haben die gleiche Länge k. Man hat also k Iterationen und pro Iteration werden alle Wörter durchgegangen und bezüglich des $j$ten Zeichens von rechts einsortiert. Das kostet $O(n)$, da es pro Wort konstante Zeit braucht ($O(1)$). Insgesamt also $\Theta(k*n) = \Theta(m)$, wobei m die Gesamtzahl aller Zeichen.
Dies gilt auch für die Variante mit unterschiedlicher Länge.

6. Vorlesung, 1.11.2007

6.1. Kapitel 2: Datenabstraktion und ihre Realisierung in Java

6.1.1. Die Liste

Eine Liste ist eine endliche geordnete Folge von Objekten gleichen Typs. So bilden die bekannten Wochentage eine Liste.
Folgende Operationen sollen auf einer Liste ausgeführt werden können.:

- Einen "Zeiger", der die momentane Position in der Liste angibt
- Listenlänge soll man bekommen können
- Position soll aufs erste Element, letzte Element bzw. aufs nächste oder vorhergehende Element gesetzt werden können.
- Der Inhalt der aktuellen Position soll angeschaut werden können
- Einfügen nach der aktuellen Position von neuen Elementen
- Löschen des Elementes auf der momentanen Position.

Diese Anforderungen sind mathematisch Formal beschreibbar. Wir arbeiten mit Mengen von Objekten und Abbildungen.

Sei U die Menge, der die Elemente entstammen dürfen. Sei L die Menge der endlichen Folgen über U. Die Position wird durch eine natürliche Zahl ausgedrückt, zudem werden die Listenelemente durchnummeriert. Die Länge ist eine Funktion

$$laenge : L \to \mathbb{N}$$
$$erst : L \to L \times \mathbb{N} \;\; mit \;\; l \to (l, 1)$$
$$letzt : L \to L \times \mathbb{N} \;\; mit \;\; l \to (l, laenge(l))$$
$$naechstes : L \times \mathbb{N} \to L \times \mathbb{N} \;\; mit \;\; (l, i) \to (l, i + 1)$$
$$vorgaenger : L \times \mathbb{N} \to L \times \mathbb{N} \;\; mit \;\; (l, i) \to (l, i - 1)$$
$$einf : U \times L \times \mathbb{N} \to L \times \mathbb{N} \;\; mit \;\; (a, l, i) \to (l', i) \;\; falls \;\; l = a_1 \dots a_i \; l' = a_1 \dots a_i \; a \; a_{i+1} \dots a_n$$
$$loesch : L \times \mathbb{N} \to L \times \mathbb{N} \;\; mit \;\; (l, i) \to (l', i) \; falls \; l = a_1 \dots a_n \; dann \; l' = a_1 \dots a_{i-1} a_{i+1} \dots a_n$$

Wir treffen keine Aussage über die Grundmenge U, das ist das eigentlich abstrakte an der Chose.
Eine solche Sammlung von Mengen und Operationen heißt "abstrakter Datentyp" (ADT).

Eine Beschreibung der Spezifikation kann auch von einem Anwender kommen. Eine formale Spezifikation ist neben mathematischer Notation auch möglich in Haskell:

```
1 data Liste a = Ll|App (Liste a) a
2
3 laenge :: Liste a -> Int
4 laenge Ll = 0
5 laenge (App l x) = laenge l + 1
6
7 einf :: a -> Liste a -> Int -> Liste a
8 einf x Ll 1 = App Ll x
9 einf x (App l y) n = if n== laenge l+1 then App (App l y) x
      else App (einf x l n) y
```

Das heißt also, die formale Spezifikation in Haskell liefert auch schon gleich ein ausführbares Programm. Die Effizienz ist hierbei allerdings nicht kontrollierbar.

Spezifikation Eine Beschreibung welche Art von Objekten und welche Operationen darauf spezifiziert werden sollen. Eine Spezifikation kommt vom Benutzer, also direkt vom Anwender oder einem anderen Programmteil in einem größeren Softwareprojekt ("Client"). Bei sowas wird der ADT zur Schnittstelle (interface). ???

Kapselung (Geheimhaltungsprinzip), damit übersichtliche Struktur des Gesamtprogramms und somit viel weniger Fehlerquellen. Die ist modulares Programmieren. Objektorientierte Programmiersprachen unterstützen diese Prinzipien. In Java passiert das über das "interface" in Java.

Datenabstraktion ADT sind mehreutig verwendbar, so können Listen für Listen aller mögliche Objekte verwendet werden. Vergleichbar mit Typvariablen in Haskell. Das nennt man auch Polymorphie. Auch im objektorientierten Sprachen durch Klassenhierarchie. Insgesammt gibt es also drei Entwicklungsstufen:

- abstrakter Datentyp: Informal oder formal spezifiziert
- Datenstruktur für Objekte und Algorithmen für Operationen
- Javaklasse: unterstützt ADT-Idee

Entwurf und Analyse

Wir gucken uns das am Beispiel der Liste an. Eine mögliche Realisierung wäre z.B. ein Feld, bzw. ein Segment davon.

Wir haben also so ein tolles Array, wo der Anfang und das Ende jeweils durch die Zeiger anf und end gekennzeichnet sind. Auf der aktuellen Position ist natürlich auch nen Zeiger. Realisieren wir nun die oben angegebenen Operationen:

$leange \rightarrow end - anf + 1$
$erst \rightarrow position = anf$
$letzt \rightarrow position = end$
$naechstes \rightarrow position + +$
$vorgeanger \rightarrow position - -$
$inhalt \rightarrow O(1)$
$einf \rightarrow$ an der Pos muss freier Platz geschaffen werden.
Dazu Rest der Liste eine Stelle verschieben $\Theta(n)$
$loeschen \rightarrow$ Rest d. Liste um 1 nach links schieben $\Theta(n)$ oder Stelle markieren $O(1)$

7. Vorlesung, 6.11.2007

7.0.2. Die verkettete Liste

Die Idee ist, dass ein Listenelement aus zwei Komponenten besteht. Zum einen erhält es das abgespeicherte Element und zum anderen ein Verweis/ Zeiger auf das nächste Element bzw. den nächsten Listenknoten. Dadurch entsteht dann eine hübsche Liste, das letzte Element zeigt übrigens ins Nirgendwo. Dazu gibt es dann noch Zeiger, die auf den Anfang, das Ende und auf die aktuelle Position zeigen.

Analyse

Nun schauen wir uns an, wie man hier die Operationen analysieren kann.

- Für die Länge n ist das hier schwieriger als vorher, man braucht $\Theta(n)$ Zeit. Man kann natürlich auch eine Integervariable mitführen, die die Länge angibt und immer bei den anderen Operationen aktualisiert wird.
- Aufs letzte und erste Element können wir wieder in $O(1)$ zugreifen, da wir direkte Verweise haben.
- Das nächste Element bekommt man auch in $O(1)$, da man einfach nur dem Zeiger folgen muss. Das vorherige geht allerdings schwieriger: Man muss von anf aus die Listenknoten durchlaufen, bis man auf pos ankommt. Dabei merkt man sich immer den vorhergehenden Knoten. Dies geht leider nur in $\Theta(n)$.
- $einf$ hinter der Position pos funktioniert so, dass man ein neuen Knoten erzeugt und da das Element reinschreibt. Dann machen wir von diesem neuen Knoten einen Verweis auf den Nachfolger von pos und lenken den Zeiger von pos auf das neue um. Hier ist dann $O(1)$ die Laufzeit.
- Wenn man das Element an Stelle pos löschen will, dann muss man einfach den Zeiger des Vorgängers auf den Nachfolger umlenken (das Vorgängerfinden ist allerdings wie eben gesehen nicht so toll). Insgesamt also $\Theta(n)$.
  Das Löschen hinter der aktuellen Position geht dann in $O(1)$, weil wir dann ja nicht mehr den Vorgänger suchen müssen ...

Wenn wir jetzt auch effizient den Vorgänger bestimmen wollen bzw. ein Element löschen, brauchen wir die doppelt verkettete Liste.

7.0.3. Die doppelt verkettete Liste

Hierbei haben die Elemente haben hierbei zwei Zeiger, einen auf den Vorgänger und einen auf den Nachfolger.
Wie können wir nun die Operationen hierauf realisieren?

- Die Länge ist wie bei einfach verketteten Listen.
- Auf das erste und letzte Element können wir wieder direkt per Zeiger in $O(1)$ zugreifen.
- Das nächste und das vorhergehende Element finden wir in $O(1)$.
- Einfügen an der Stelle pos geht auch wieder in $O(1)$. Wir müssen diesmal bloß ein wenig mehr Zeiger umbiegen.
- Löschen geht auch ganz schnell in $O(1)$, weil ja auch wieder nur gut erreichbare Zeiger umgebogen werden müssen.

Man kann sich natürlich auch noch andere Listenkonstrukte vorstellen wie z.B. die zirkuläre Liste, wo alle Elemente im Kreis angeordnet sind.

7.0.4. Implementierung in Java

Eine Java-Klasse enthält neben dem abstrakten Datentyp Implementierungen der Datenstrukturen und Algorithmen für dessen Daten und Operationen.
Zur Realisierung eines abstrakten Datentyps brauchen wir Polymorphie.

Polymorphie Eine Variable ist polymorph (vielgestaltig), wenn sie je nach Kontext verschiedene Typen oder Operationen realisiert.
Ganz simple Polymorphie ist z.B. das Überladen, wie z.B. das $+$-Zeichen, mit dem man in Java sowohl double, int als auch Strings addieren kann.
In Haskell kann man z.B. Quicksort per Typvariable Ord so definieren, dass es auf alle Typen angewandt werden kann, die zur Typklasse Ord gehören.

Zur Erinnerung nochmals Vererbung und Klassenhierarchie in Java.
In Java kann eine Klassen A definiert werden mit dem Zusatz $extends\ B$, so dass sie auch auf alle Datenfelder und Methoden von B zugreifen kann, sofern sie nicht in A überschrieben werden. D.h., A erbt von B.
Dadurch lassen sich ganze Hierarchien von Klassen (Baumstruktur) aufbauen. Dabei kann jede Klasse nur von einer Klasse beerbt werden. Es gibt eine vordefinierte Klasse, die über allen anderen steht und diese heißt $Object$. Deswegen ist eine Möglichkeit Listen möglichst universell zu machen die, sie einfach über den Typ $Object$ zu definieren.

Beispiel

```
1  class A
2  {
3    void print(String s)
4    {...}
5    void p(char a)
6    {...print("a")...}
7    void p(int a)
8    {... print("b")...};
9  }
10
11 class B extends A
12 {
13   void p(int b)
14   {...print("c"); ...}
15 }
```

Beim Aufruf von `A a = new A();` mit `a.p(1);`, dann wird b ausgegeben. Hieran sieht man, dass man p überladen wurde, in dem sie mehrfach definiert ist. Welche Definition genommen wird, hängt vom Typ des Arguments ab. Bei `B a = new B();` mit `a.p(1);` würde jetzt c ausgedruckt werden, da die Methode überschrieben wurde.
Sowas nennt sich Einschlusspolymorphie, also, dass eine Methode in erbender und verebender Klasse definiert wird. Die Methode in einer vererbenden Klasse wird dabei überschrieben.
Es gibt (jetzt) in Java auch sowas wie Typvariablen, das nennt man parametrische Polymorphie.

8. Vorlesung, 8.11.2007

8.0.5. Schnittstellen und abstrakte Klasse

In Java gibt es auch die Bezeichnung "interface" (Schnittstelle) statt "class". Die Schnittstelle gibt nur die Namen und Signaturen der Methoden an, alle Methoden sind automatisch abstrakt, d.h. es wird kein Code zur Implementierung der Methode angegeben.
Die Liste als interface sähe dann wie folgt aus

```
1  public interface Liste {
2    public int laenge();
3    public Position erst();
4    public Position letzt();
5    public Position vorher(Position p);
6    public Position naechste(Position p);
7    public Object inhalt(Position p);
8    public void einf(Object a, Position p);
9    public void loesche(Position p);
10 }
11 public interface Position {
12   public Object element();
13 }
```

Im restlichen Code kann eine Variable vom Typ T eines Interfaces deklariert werden, also z.B. `T x;`, aber man darf nicht per `new T` eine Instanz erzeugen.

Klassen können Interfaces implementieren, und zwar per

```
1 public interface I {
2   ...
3 }
4 public class C implements I {
5   ...
6 }
```

C implementiert alle Methoden, die im Interface angegeben sind. Im Gegensatz zu `extends` darf hinter `implements` eine Liste von mehreren Interfaces stehen. Dies ist die einzige Möglichkeit der Mehrfachvererbung in Java. Eine Klasse kann so also von mehreren Interfaces erben.
Klassen können übrigens Interfaces implementieren und andere Klassen erben. Und Interfaces

können auch von einem Interface erben.

Und nun die einfach verkettete Liste als Implementierung der Interfaces:

```
1  public class Knoten implements Position {
2    private Object element;
3    private Knoten next;
4
5    public Knoten(Object a, Knoten n) {
6      this.element = a;
7      this.next = n;
8    }
9
10   public Knoten() {}
11
12   public Knoten getNext() {
13     return next;
14   }
15
16   public void setNext(Knoten nNext) {
17     next = nNext;
18   }
19
20   public void setElement(Object nElement) {
21     element = nElement;
22   }
23
24   public Object element() {
25     return this.element;
26   }
27 }
28
29 public class EVListe implements Liste {
30   protected int anz;
31   protected Knoten anf, end;
32
33   public EVListe() {
34     anz = 0;
35     anf = ende = null;
36   }
37
38   public int laenge() {
39     return anz;
40   }
41
42   public Position erst() {
43     return anf;
```

```
44    }
45
46    public Position vorh(Position p) {
47      if(p == anf || anf == null) {
48        throw new ...
49      }
50      Knoten v= anf;
51      Knoten c= anf.next;
52      while(c != p) {
53        if(c == null) {
54          throw new ...
55        }
56        v=c;
57        c=c.next;
58      }
59      return v;
60    }
61
62    public void einf(Object a, Posiiton p) {
63      if (anf == null && p != null) {
64        throw new ...
65      }
66      Knoten v = new Knoten();
67      w = (Knoten) p; // kann geparst werden, da Position
           Knoten beerbt
68      v.n
69      if(p==null) {//neues Element soll an Anfang der Liste
           eingefügt werden
70        v.setNext(anf);
71        anf = v;
72      }
73      else {
74        v.setNext(w.getNext());
75        w.setNext(v);
76      }
77      anz++;
78      if(p == ende) {
79        ende = v;
80      }
81    }
82
83    public Object inhalt(Position p) {
84      return p.element();
85    }
86  }
```

9. Vorlesung, 13.11.2007

9.0.6. Parametrische Polymorphie in Java

Die entspricht wie schon mal erwähnt den Typvariablen in Haskell. Und zwar werden die generischen Typen erst zur Laufzeit belegt.
Die Typbezeichnung wird in spitzen Klammern ($< >$) hinter dem Klassennamen beschrieben.

```
1 public class Pair <K,V> {
2   K key;
3   V value;
4   ...
5 }
```

Hier können K und V wie übliche Typen benutzt werden. Wenn ein Objekt dieser Klasse instanziert oder eine Variable von diesem Typ definiert wird, müssen für die Typvariablen konkrete Typen eingesetzt werden.

```
1 Pair <String, Integer> pair1 = new Pair <String, Integer>;
```

$extends$ ist hinter den genereischen Typen möglich.

```
1 class X <T extends Y, S extends Z ... >
```

blabla ... der Y erweitert. Das mit dem $extends$ entspricht den Typklassen in Haskell, z.B. $quickSort :: Ord\ a\ => \ldots$

9.0.7. Iteratoren

Der Iterator ist ein abstrakter Datentyp, der dem Durchlaufen aller Elemente einer Liste entspricht. Allgemeiner gibt es ein interface java.lang.Iterable, das die Methoden $hasNext()$ und $next()$ enthält.

10. Vorlesung, 15.11.2007

```
1 public interface Collection<E> extends Iterable<E> {
2   ...
3 }
4 public interface Iterable<T> {
5   Iterator<T> iterator(); // liefert erste Element
6 }
7 public interface Iterator<E> {
8   boolean hasNext();
9   E next();
10 }
```

Der Iterator durchläuft alle Elemente und $hasNext()$ besagt, wenn wahr, dass es noch ein nicht durchlaufenes Element gibt. $next()$ liefert dann das nächste Element.

for-Schleife

```
1 for (int i=1;i<=5; i++)
```

Die Initialisierung wird am Anfang einmal ausgeführt. Die Schleife wird so lange ausgeführt, wie bei der Bedingung zu true ausgewertet wird. Hier muss also immer ein boolean-Wert stehen. Am Ende jeder Iteration wird Inkrement ausgeführt, hier kann ein bliebiger Ausdruck stehen.
Man merke, dass alle Angaben optional sind.

```
1 for(Iterator<Typ> it = next.iterator(); it.hasNext(); ) {
2  Typ name = it.next();
3  Befehlsfolge bla bla
4}
```

Initialisierung muss blabla auf diesem wird ein Iterator (it) angelegt. Solange der noch ein nächstes Element liefert (was ja die Bedingung ist) wird die Befehlsfolge ausgeführt, also für alle Elemente in O.
Diese Schreibweise kann man auch abkürzen:

```
1 for(Typ name : aus) {
2   Befehlsfolge
3 }
```

Zum Beispiel liefert nachfolgendes die Summe aller ganzzahligen Werte in values.

```
1 List<Integer> values;
2 for( Integer i : values) {
3  sum += i;
4 }
```

10.1. Kaptiel 3: Weitere Abstrakte Datentypen

10.1.1. Keller (Stapel bzw. Stack)

DIe Idee ist, dass man alle Objekte auf einen Stapel legt, bei dem man auf das oberste zugreifen kann. Weiterhin kann immer noch ein neues aufgelegt werden (push) bzw. entfernt werden (pop). Man nennt das auch das LIFO-Prinzip (last in, first out).

Datentyp Als abstrakten Datentyp hat man also eine Menge A von möglichen Elementen und die Operationen $push(a), a \in A$ und $pop()$. Dazu kann man noch $top()$ definieren, dass das oberste Element liefert. $isEmpty()$ prüft auf Vorhandensein von Elementen.

Anwendungen Bearbeitung von verschachtelten Objekten. Z.B. wird beim Browser das vor und zurück auf einem Stack verwaltet oder auch die undo-Funktion bei Editoren.
Wir wollen uns jetzt genauer die Postfix-Notation bei Ausdrücken angucken, üblich ist ja eigentlich die Infixnotation wie

$$(10 - (3 + 5)) * 7) - 4 + 2$$

Postfix sähe so aus

$$10\ 3\ 5 + -7 * 4 - 2+$$

Vorteil hier ist, dass man keine Klammern mehr benötigt und auch keine Prioritätsregeln wie Punkt vor Strichrechnung. Außerdem braucht man auch nix für Assoziativität.
Die Postfixnotation wird auch UPN - "umgekehrte polnische Notaton" genannt.
Operatoren werden ausgeführt sobald sie auftreten und zwar auf den obersten Operanden im Stack. Das funktioniert auch für mehrstellige und einstellige Operationen.

Infix nach Postfix Um von Infix zu Postfix zu kommen, benutzt man ein Operatorenstack. Beim Einlesen von Operanden werden diese ausgegeben. Wenn ein Operator o_1 eingelesen wird: Solange der Operator an der Spitze des Stacks höhere oder gleiche Priorität wie o_1 hat, dann gib o_2 aus und entferne es vom Stack. Fall $o_2 = o_1$ und o_2 ist linksassoziativ, dann gib es aus. Und dann mach $push(o_2)$.
Bei einer öffnenden Klammer, packen wir sie einfach auf den Stack. Bei einer schließenden Klammer, gib alle Operatoren vom Stack aus, bis eine öffnende Klammer auftaucht, die dann gelöscht wird ($pop()$).

11. Vorlesung, 20.11.2007

11.0.2. Einschub: Typkonversion in Java

Die Konversion ist das Umwandeln von einem Typ in einen anderen.

Java ist streng typisiert. D.h. in Java ist bei jeder Variable der Typ anzugeben (also Klassen, Interface oder Grundtypen).
Man nennt A einen Obertyp von B, wenn in der Hierarchie der Klassen (und Interfaces) B von A erbt.
Es gibt verschiedene Arten von Konversion

ausweitende Konversion Funktioniert von Untertyp zu Obertyp

```
1 class A {}
2 class B extends A {...}
3 A x;
4 B y;
5 x = y; // ist immer möglich, der Typ von y wird implizit
       auf den von x ausgeweitet
```

Statt y kann hier auch ein anderer Ausdruck stehen, dessen Typ Untertyp von A ist.

einengende Konversion Funktioniert von Obertyp zu Untertyp und muss explizit mit Cast durchgeführt werden. Der Typ der Variablen x ist hier zur Compilezeit klar. Der Typ des Objekts, der x zugewiesen ist, kann A oder ein Untertyp von A sein.
Im Beispiel zeigt nach $x = y$ das x auf ein Objekt von Typ B, obwohl der Typ als A angegeben ist.

Dynamische Bindung von Methoden

Im Beispiel wird in A eine Methode f deklariert, die von B überschrieben wird. Angenommen $x.f()$ wird aufgerufen. So wird diejenigen ausgeführt, die den Typ des Objekts entspricht, auf das x zeigt. Z.B. bei

```
1 A x = new A(...);
2 x.f(...); // hier wird das in A definierte f aufgerufen
3 A x  = new B(...);
4 x.f(...); // hier wird das in B definierte f aufgerufen
```

Zur Erinnerung, das war die so genannte Einschlusspolymorphie.

Verdecken

Bei Attributen ist das Verhalten umgekehrt. Also, wenn das Attribut c definiert in A und in B so ist bei

```
1 A x = new B(..);
2 d= x.c
```

das in B definierte c verdeckt.

```
1 class A {
2  int c =1;
3 }
4 class B extends A {
5  int c=2;
6 }
7 A x = new B(...);
8 print(x.c) // druckt 1 aus!
```

Jetzt zurück zur Datenstruktur Stack

11.0.3. Implementierung des Stack

Wie kann man einen Stack implementieren? Zum Beispiel als verkettete Liste, wo der Anfang der Liste den Kopf bildet. Bei push hängt man einfach vorne ein neues Element an und bei pop löscht man das erste und setzt den Zeiger top weiter.
Alle Operationen gehen hier in $O(1)$ Zeit möglich.
Man kann natürlich auch den Stack als Array implementieren. Da geht das auch in $O(1)$.

11.0.4. Die Warteschlange

In Englisch queue genannt. Die Schlange ist eine FIFO Struktur, wo immer hinten neue Elemente angehangen werden ($enqueue()$) und vorne bearbeitet und entfernt werden ($dequeue()$).
Zusätzlich Operationen wären $size()$, $isEmpty()$, $front()$.

Anwendungen Halt überall dort, wo man Warteschlangen antrifft.
Ein Interface könnte so aussehen:

```
1 public interface Queue<E> {
2   public int size();
3   public boolean isEmpty();
4   public E front() throws EmptyQueueException;
5   public void enqueue(E element);
6   public E dequeue() throws EmptyQueueException;
7 }
```

14. Vorlesung, 29.11.2007

Die Inorder-Traversierung binärer Suchbäume liefert aufsteigend sortiert die Folge der Elemente.

14.0.5. Binärbaum

Der Datentyp hat folgende Operationen

- $left(v)$ liefert das linke Kind
- $right(v)$ liefter das rechte Kind
- $hasLeft(v)$ liefert true, wenn links ein Knoten
- $hasRight(v)$ liefert true, wenn rechts an v noch ein Knoten ist

Jetzt kommt die Implementierung in Java. Als erstes brauchen wir die Zeigerstruktur für die Knoten. Und zwar zeigt jeder Knoten mittels Zeiger auf den Vater, dann mit einem Zeiger aufs Element und jeweils mit einem Zeiger auf die beiden Kinder.

14.1. Kapitel 4: Prioritätswarteschlangen

Elemente in einer Prioritätwarteschlange haben verschiedene Grade von Dringlichkeit ("Priorität"). Ein Beispiel aus dem täglichen Leben wäre z.B. die Notaufnahme in einer Klinik. Wie wir aus TI 3 auch schon kennen, werden die Prozesse in einem Betriebssystem auch nach Prioritäten bearbeitet.
Allgemein hat man in einer Prioritätswarteschlange Paare (k, x) von Schlüssel und Wert des Elements zu verwalten, wobei k die Priorität repräsentiert.
Die Priorität muss aber nicht immer eine Zahl sein, es kann allgemein ein Element eines linear geordneten Universums sein. Kleiner in der linearen Ordnung bedeutet, dass das Element mit diesem Schlüssel eine höhere Priorität hat.

Operationen

- $size()$ gibt die Länge der Warteschlange zurück.
- $isEmpty()$ sagt und, ob die Warteschlange leer ist oder nicht.

- $deleteMin()$ bedeutet, dass das Element mit der höchsten Priorität aus der Warteschlange entfernt wird.
- $insert(k, x)$ heißt, dass das Paar (k, x) in die Warteschlange eingefügt wird.
- $min()$ liefert das nach Schlüssel kleinste Element, also das mit höchster Priorität.

Jetzt ist die Frage, welche Datenstruktur sich für eine Prioritätswarteschlange eignet.
Man kann z.B. eine einfach verkettete Liste nehmen, dann gehen alle Operationen bis auf $deletMin()$ und $min()$ in konstanter Zeit, diese beiden allerdings lineare Laufzeit. Für kleinere Prioritätswarteschlangen ist dies z.B. eine geeignete Struktur.
Für wirklich große Mengen ist der sogennante Heap sinnvoll.

14.1.1. Heap - Haufen

Ein Heap ist ein binärer Baum, bei dem jede Ebene d 2^d Knoten hat, bis auf möglicherweise die unterste Ebene. Diese ist von links nach rechts mit Knoten aufgefüllt.
Die Knoten des Heap sind beschriftet mit Schlüsseln aus einem linear geordnete Universum. Und zwar so, dass der Schlüssel in einem inneren Knoten kleinergleich dem Schlüssel in den Kindknoten ist.

Eigenschaften

Sei S eine Menge von n Schlüsseln in einem Heap B der Höhe h.
Das Minimum von S steht immer in der Wurzel von B. Das ist ganz einfach zu begründen, da die Kinder vom Schlüssel her immer größer sind als der Elternknoten - logisch. Formal müsste man einen Induktionsbeweis führen.
Als nächstes gilt, dass jeder Unterbaum von B auch ein Heap ist. Diese Eigenschaft folgt unmittelbar aus der Definition.
Als drittes ist die Höhe von B durch $\log n$ gegeben. Für $n = 2^k$ ist $h = k$, denn $\sum_{i=0}^{h} 2^i = 2^k - 1$. Für $n = 2^k$ erhöht sich die Höhe also um 1. Für die Anzahl der Elemente zwischen $2^k \leq n \leq 2^{k+1}$ ist die Höhe also k.

Algorithmen und Laufzeiten für die Operationen

$size()$, $isEmpty()$ und $min()$ geht in $O(1)$.
Beim Einfügen packen wir das neue Element erstmal ganz unten rechts rein, also da, wo noch was frei ist, ansonsten in eine neue Zeile. Wenn der Schlüssel vom Vater allerdings größer ist, dann werden die Knoten vertauscht. Dieses Vertauschen wird so oft praktiziert, bis es der Heapstruktur entspricht.

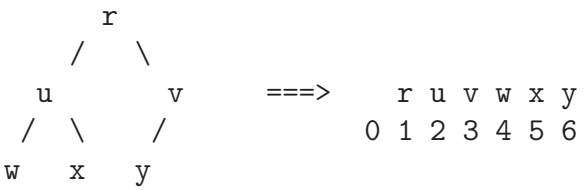
15. Vorlesung, 4.12.2007

Jetzt zum Löschen. Beim Löschen wird die Wurzel gelöscht und ausgegeben. Und zwar wird das letzte Element des Baumes in die Wurzel geschrieben. Jetzt muss dieses Element mit den beiden Kindern verglichen werden, und, falls es größer ist als die Kinder, mit dem kleineren der beiden Kinder getauscht werden. Und das muss wieder so oft geschehen, bis es wieder ein Heap ist.
Wenn man jetzt hier die Laufzeit betrachtet dann ergibt sich für das Löschen und Einfügen. $O(Hoehe(T)) = O(\log n)$.

Implementierung

Die naive Idee. Man nimmt einfach die übliche Baumimplementierung mit Knoten, die den Schlüssel, das Element und Zeiger auf den Elternknoten und zum linken und rechten Kind enthalten.
So macht man das aber nicht, dass hier der Nachteil ist, dass das Finden des nächsten leeren Platzes relativ aufwendig ist.

Ne coolere Variante bildet die Array-Implementierung. Die Idee ist hier, dass die Einträge der Knoten des Heaps schichtweise in ein Feld eingelesen werden, angefangen mit der Wurzel auf Index 1.



In unserem Array geht dann natürlich die Baumstruktur verloren. Allerdings kann man ganz geschickt z.B. den Index des Kindes ausrechnen. Es gilt:

Knoten	Index
v	i
Kind links	$2i$
Kind rechts	$2i+1$
Elternknoten	$\lfloor \frac{i}{2} \rfloor$

Jetzt kann man auch ganz einfach auf das letzte Element zugreifen, dass ist nämlich einfach der letzte Eintrag im Array.

Eine Beispielimplementierung gibt's natürlich wieder auf der schicken Seite im Internet.

Jetzt wollen wir mit unserem Heap sortieren.

15.0.2. Heapsort

Die Idee ist folgende:

1. Beginn mit leerer Halde und füge nacheinander die zu sortierenden Elemente ein.
2. Lösche nacheinander alle Einträge bis die Halde leer ist und trage die gelöschten Einträge in die Ausgabe ein.

Wir gucken uns jetzt die Laufzeit an. Für die erste Phase benötigen wir für jedes Element $\log n$, also $O(n * \log n)$. Für die Phase 2 müssen wir wieder für jedes Element die Löschoperation ausführen, also n mal. Demnach gilt auch hier wieder $O(n * \log n)$. Die Laufzeit insgesamt beträgt dann auch $O(n \log n)$.

Wir versuchen jetzt, die Phase eins noch ein wenig zu verbessern.

Bottom-Up Heap-Konstruktion

Was für ein Name!
Die Idee ist hier, dass für $n = 2^m - 1 = 2^{m-1} + 2^{m-2} + \ldots + 2 + 1$ annehmen, also n Knoten haben. Wir bauen das jetzt in m Stufen auf.

Stufe 1 Bilde 2^{m-1} Halden der Größe 1 mit den ersten 2^{m-1} Einträgen in der Wurzel

Stufe k+1 Fasse jeweils zwei Halden der 2^{m-k} Stufe k zu Paaren zusammen und bilde 2^{m-k-1} neue Halden durch Einführung einer Wurzel für jedes Paar und füge die nächsten 2^{m-k-1} Elemente der Eingabefolge in die entsprechende Wurzel ein. Dann müssen diese Einträge noch versickert werden.

Die Laufzeit ist hier proportional zur Anzahl der Vertauschungen der Einträge. Und zwar hat man 3 Vergleiche pro Vertauschung. Wir kennen zudem die Anzahl der Vertauschungen der Stufe k. Hier haben wir nämlich 2^{m-k} neue Halden geschaffen und wir müssen maximal die Höhe von n vertauschen. Es gilt $2^{m-k} * (k-1)$.
Insgesamt ergibt sich also $\sum_{k=1}^{m}(k-1) * 2^{m-k} = 1 * 2^{m-2} + 2 * 2^{m-3} + 3 * 2^{k-4} + \ldots + (n-2)2 + (n-1) * 1$. Man kann jetzt immer die Zweierpotenzen rausnehmen: $= \sum_{i=0}^{m-2} 2^i + \sum_{i=0}^{m-3} 2^i + \ldots + \sum_{i=0}^{1} 2^i + 1$. Und das sind ja alles geometrische Reihen, deshalb kann man auch schreiben $= (2^{m-1} - 1) + (2^{m-2} - 1) + \ldots + (2^2 - 1) + (2 - 1) = 2^m - (m+1) \leq n$. Man sieht also, dass man weniger als n Vertauschungen gemacht hat und somit das Bilden eines Haufens nur noch $O(n)$ und nicht mehr $O(n \log n)$ dauert.

Man kann auch eine andere Analyse machen. Und zwar bekommen wir für jedes Einfügen 2 Euro und jedes Löschen kostet uns 1 Euro. Wir behaupten dann für vollständige Halden, dass am Ende ein paar Euro übrig bleiben. Wir führen einen Induktionsbeweis:

Induktionsanfang $h = 0$ Kein Guthaben, aber auch keine Ausgaben

Induktionsschritt Die Aussage sei wahr für Halde der Höhe h. Sie sollda nun auch für Halde der Höhe $h + 1$ gelten. Nach Voraussetzung haben wir in der Linken und der rechten Halde jeweils h Euro übirg. Für die neue Wurzel bekommen wir 2 Euro dazu. Wir haben also dann $2h + 2$ Euro Guthaben. Wir können aber nur maximal die Höhe von $h + 1$ Tauschs durchführen. Es bleiben also trotzdem noch $h + 1$ Euro übrig.

Man muss beachten, dass die 2. Phase kann nicht in der Art und weise verbessert werden. Und zwar wäre das sonst ein Widerspruch zur unteren Schranke von $\Omega(n \log n)$ für Sortieralgorithmen. Sonst könnte man nämlich in linearer Zeit sortieren.
Wir wollen jetzt diese untere Schranke uns erklären. Dazu betrachten wir alle Permutationen Π für ?? A muss $\{1, \ldots, n\}$ ausgeben, d.h. es muss Π^{-1} ausrechnen. A mus also $n!$ Permutationen durch Vergleichfajföajfaö . Man erhält also einen Baum mit $n!$ Blättern, der ein Binärbaum ist. Wir wissen dann,dass die Höhe dann $Hoehe(T) \geq \log(n!) = \Theta(n \log n)$. Und die Höhe ist ja gerade die Laufzeit im worst-case.

16. Vorlesung, 6.12.2007

16.1. Kapitel 5: Wörterbücher

16.1.1. Der abstrakte Datentyp Wörterbuch

Ein Wörterbuch soll eine Menge D von n Einträgen enthalten, wobei diese die Form (Schlüssel, Wert) haben. Die Schlüssel entstammen einem Universum U, wobei U entweder eine lineare Ordnung (geordnetes Wörterbuch) oder keine lineare Ordnung (ungeordnetes Wörterbuch) hat.

Operationen

- $finde(k)$ Finde (einen) Eintrag mit Schlüssel k
- $einfuege(k, v)$ Füge den Eintrag (k, v) zu D hinzu
- $streiche(e)$ Streiche den Eintrag e
- $findeAlle(k)$, wie $finde(k)$ bloß, dass alle Einträge zurückgeliefert werden mit Schlüssel k
- $groesse()$ Die Anzahl der Elemente von D
- $istLeer()$ Prüft, ob die Menge leer ist oder nicht
- $eintraege()$ liefert eine Liste der Einträge in Java z.B. eine iterable collection.

Eine einfache Datenstruktur für den ADT wäre z.B. eine einfach verkettete Liste. Wie würde man jetzt die Operationen umsetzen?

- $finde(k)$ hier muss die Liste einfach nur durchlaufen werden, das geht in $O(n)$.
- $einfuege(k, v)$ Man kann einfach vorne oder hinten anhängen, dass kostet dann $O(1)$ Zeit.
- $streiche(e)$ Falls e ein Paar (k,v) ist, dann muss wieder die Liste durchgegangen werden, also $O(n)$. Wenn e eine Position in der Liste beschreibt, dann geht das in konstanter Zeit $O(1)$.

Das ist natürlich nicht wirklich cool, dass fast alles $O(n)$ braucht, deshalb gibt es was besseres, dass ist Hashing.

16.1.2. Umsetzung mit Hashing - Streuspeicherung

Idee Für die Menge D steht ein Array $A[0,\ldots,N-1]$, die Hashtabelle, zur Verfügung und es gibt eine Funktion $h : U \to \{0,\ldots,N-1\}$, die sogenannte Hashfunktion. Für einen Eintrag (k,v) ist $h(k)$ die Stelle in der Hashtabelle, wo der Eintrag abgelegt werden soll.
Im allgemeinen ist $|U| >> N$, also kann die Funktion h nicht injektiv sein. D.h. es gibt Schlüssel k_1, k_2 mit $k_1 \neq k_2$ und $h(k_1) = h(k_2)$. Wenn jetzt diese beiden Einträge so ein krasses Battle um eine Position in der Hashtabelle führen, dann nennt man das Kollision.
Möglichkeiten Kollisionen aufzulösen ist das Chaning.

Chaning Die Arrayeinträge A sind Listen und die $i$te Liste enthält alle Einträge von D mit dem Hashwert i.

Jetzt haben unsere Operationen natürlich andere Laufzeiten.

- $finde(k)$ muss man erstmal $j = h(k)$ berechnen und dann die $j$te Liste nach Schlüssel k durchsuchen.
- $einfuege(k,v)$ hier würde man auch wieder $j = h(k)$ berechnen und einfach an die $j$te Liste anhängen
- $streiche(k,v)$ wäre selbiges, bloß mit löschen.

Wie man sieht, sind die Laufzeiten also alle von der Länge der Liste j abhängig, da dass berechnen des Hashwertes konstant geht. Also O(Länge der Liste).
Die Hashfunktion sollte nun also so beschaffen sein, dass sie möglichst gut nach $\{0,\ldots,N-1\}$ streut, das heißt für ein zufällig gewähltes (z.B. Gleichverteilung) $k \in U$ die Wahrscheinlichkeit, dass $h(k) = j$ ist, für alle $j \in \{0,\ldots,N-1\}$ gleich ist, nämlich $\frac{1}{N}$.
Die mittlere Länge einer Liste ist in diesem Fall $\lambda = \frac{n}{N}$, der Ladefaktor der Hashtabelle. Bei einer Hashfunktion dieser Art mit konstantem Ladefaktor λ gilt, dass die Operationen des Datentyps Wörterbuch im Mittel $O(1)$ Zeit, vorausgesetzt natürlich die Berechnung der Hashfunktion geht auch in dieser Zeit. Im schlechtesten Fall ist es immernoch $\Theta(n)$ die Laufzeit.

Rehashing Das Rehashing sorgt für einen konstanten Ladefaktor, auch bei beliebig viele Einfügeoperationen. (In der Praxis will man $\lambda < 1$ oft $0,9$)
Die Idee ist hier folgende: Wenn sich D stark vergrößert, dann legen wir eine neue Hashtabelle an, etwa der doppelten Größe. Dann brauch man natürlich auch ne neue Hashfunktion. Dann fügt man alle bisherigen Daten in die neue Tabelle ein.
Trotz diese zusätzlichen Kosten beträgt die Laufzeit für n Einfügungen im Mittel immer noch $O(n)$. Es gibt hier natürlich einzelne Operationen, die teurer sind, aber die amortiesierte Laufzeit bleibt $O(1)$ beim Einfügen. Denn angenommen die Anfangsgröße der Hashtabelle ist k und der zu haltende Ladefaktor ist λ. Nach λk Einfügungen wird das erste mal verdoppelt auf $2k$ per Rehashing. Das kostet $O(\lambda k)$ Zeit. Nach $2\lambda k$ Einfügungen wird wieder gerehasht und das kostet jetzt $O(2\lambda k)$ Zeit usw. Insgesamt muss also n mal ein Rehashing gemacht

werden, so dass jetzt die Größe $2^m \lambda k \leq n$ ist, hier ist m dann maximal.
Die Zeit dafür ist dann proportional zu $\lambda k(1 + 2 + 4 + \ldots + 2^m) = \lambda k(2^{m+1} - 1)$. Und das ist in der gleichen Größenordnung wie das n, also $O(n)$.

Geeignete Hashfunktionen

$h : U \to \{0, \ldots, N-1\}$. Um $h(k)$ zu berechnen könnte man k in eine ganze Zahl verwandeln und im zweiten Schritt dann in den Zahlenbereich bringen, und zwar mit $\mod N$.
Wie kann man nun z.B. für Strings den ersten Schritt umsetzen?
Man kann so Zeichen für Zeichen den Buchstaben in eine Zahl umwandeln und dann alle Werte aufaddieren. Problem ist hierbei, dass Wörter mit gleichen Buchstaben den selben Hashwert ergeben. Außerdem werden bei kurzen Strings die kleinen Zahlen bevorzugt. Zudem werden Strings wie $x1$, $x2$ usw. auf aufeinanderfolgende Hashwerte abgebildet, das hat keine gute Streuung.
Dieses Verfahren ist also doch garnicht so cool. Besser ist es das ganze polynomiell zu machen, dass man also eine Zahlen(Zeichen)folge $a_0, \ldots, a_n$ abbildet auf $a_0 + a_1 * 37 + a_2 * 37^2 + \ldots + a_m * 37^m$. Man kann natürlich auch einen anderen Wert als 37 zu haben, es wäre aber eine Primzahl ratsam. Man muss aber aufpassen, dass die Größe der Hashtabelle kein Vielfaches von 37 ist und auch mit dem ggT sollte man irgendwie aufpassen, aber das hab ich mir nicht so ganz gemerkt, was da jetzt die Begründung für ist.
Beim Rehashing sollte man zudem statt verdoppeln die nächstgrößere Primzahl von 2N wählen.
Das eigentliche Hashing ist der Schritt eins, das zweite ist die Kompression.

17. Vorlesung, 11.12.2007

In Java gibt es übrigens schon die Funktion $hashCode()$, die in Object definiert ist und zu jedem Objekt einen int-Wert zurückliefert.

17.0.3. Binärsuche

Die Binärsuche für das Wörterbuchproblem. Hier sind die Schlüssel aufsteigend sortiert in einem Array A. Wie funktionieren jetzt hier unsere Operationen?

$finde(k)$ Ist wieder effizient machbar. Wir können nämlich direkt in die Mitte des Array gehen und dann k mit dem Schlüssel des Mittleren vergleichen. Da wir eine sortierte Folge haben, wissen wir dann bei Ungleichheit, in welcher Hälfte wir rekursiv suchen müssen. So muss also links gesucht werden, wenn die Mitte $> k$ und rechts, wenn die Mitte $< k$.

```
binSuch(A,k,l,r) { // durchsucht A nach k im Bereich A[l], A[r]
 if (l > r) gibt Element nicht gefunden aus
 else {
  m = (l+r) /2 abgerundet
  if(A[m] == k) then gefunden
  else if (A[m] > k) then binSuch(A,k,l,m-1)
  else binSuch(A,k,m+1,r)
 }
}
```

Jetzt zur Analyse. Der Vergleich kostet konstante Zeit $O(1)$. Danach der rekursive Aufruf für ein Array-Segment, dessen Größe $\leq \frac{1}{2}*$ vorherige Größe. Falls die ursprüngliche Größe n ist, dann bedeutet das, dass nach k Schritten die Größe des Segmentes $\leq \frac{n}{2^k}$ ist. Damit gibt es höchstens m Schritte, wobei $\frac{n}{2^m} < 1$. Also $\log n < m$. Die gesamte Laufzeit ist also $O(\log n)$. Die Anzahl der Vergleiche ist ebenfalls $\log n$

$einfuege(k, v)$ und $streiche(k, v)$ sind nicht effizient. Und zwar muss man hier ab der Stelle, wo es reinpasst oder gelöscht wird, verschieben, was $\Theta(n)$ Zeit kostet.

Da man im Telefonbuch nicht so doof wäre, immer zu halbieren sondern immer schon ungefähr weiß, wo man gucken muss, gucken wir uns also ne coolere Suche an, das das so macht, die Interpolationssuche.

Interpolationssuche

Annahme, das Universum aus dem die Schlüssel stammen, ist das Intervall $[0, 1]$. Das kann ja auch ganz einfach auf andere Universen umgerechnet werden. Die Schlüssel sind zudem zufällig aus dem Universum gezogen, d.h. bezüglich der Gleichverteilung.
Außerdem setzen wir $A[0] = 0$ und $A[n+1] = 1$. die eigentlichen Schlüssel sind die zwischen $A[1]$ und $A[n]$. Bei gegebenem Schlüssel k rechnen wir mit $m = l-1+\lfloor \frac{k-A[l]}{A[r]-A[l]} * (r-l+1) \rfloor$.
Ansonsten funktioniert es wie die Binärsuche, bloß muss halt das m anders berechnet werden.
Im Mittel kostet diese Suche $O(\log \log n)$.
Es gibt auch noch die quadratische Binärsuche, die die selbe Laufzeit hat.

17.0.4. Skip-Listen

Für ein Wörterbuch D sind das sortierte doppelt verkettete Listen $S_0, S_1, \ldots, S_h$, wobei S_0 die Schlüssel von D enthält und die Elemente $-\infty$ und $+\infty$ aufsteigend sortiert. S_i entsteht dabei aus S_{i-1} durch zufällige Auswahl der Elemente (Wahrscheinlichkeit $\frac{1}{2}$ pro Element) aus S_{i-1}. S_h besteht nur noch aus ∞ und $-\infty$.
Pro Position p hat man vier Verweise. Und zwar auf das vorherigen ($vorh(p)$) und nächste ($naechst(p)$) und auf die drunter ($runter(p)$) und drüber ($oben(p)$).

$finde(k)$

```
while (runter(p) != null) do
 p = runter(p)
 while(k >= Schlüssel von naechst(p)) do
   p = naechst(p)
return p
```

$einfuege(k, v)$ Im ersten Schritt führe $finde(k)$ aus, das liefert Position p
Als zweites wirf eine Münze bis Kopf kommt. Wenn man m mal Zahl gehabt hat, dann errichtet man einen Turm der Höhe m über k. Dann muss der Suchweg zurückverfolgt werden und die richtigen Zeiger gesetzt. Man muss neue Ebenen hinzufügen falls nötig.

18. Vorlesung, 13.12.2007

$loesche(k, v)$ Beim Löschen wird erste eine Suche bezüglich des Schlüssel ausgeführt, bis der Eintrag gefunden wird. Als zweites wird dann der Eintrag aus der untersten Liste gelöscht. Folge dann den $oben(p)$-Zeigern und lösche alle Einträge im Turm darüber. Da muss mann dann natürlich immer die Zeiger so geschickt neu setzen. Wenn Listen dann nur noch aus $-\infty$ und ∞ bestehen, werden bis auf die oberste gelöscht.

Jetzt kommt die so sehnlich erwartete Laufzeitanalyse.

Laufzeit

Im schlechtesten Fall wird beim Einfügen kein terminierter Zustand erreicht, weil er ja theoretisch unendlich oft Zahl kommen könnte. Dafür ist die Wahrscheinlichkeit allerdings 0. Darum gucken wir uns lieber die erwartete Laufzeit an.
Was ist hierbei die erwartete Höhe? p_i sei die Wahrscheinlichkeit, dass die Höhe $\geq i$ wird. Für einen Turm bedeutete das also, dass man bei einer Münze i mal hintereinander Zahl werfen müsste, demnach ist die Wahrscheinlichkeit hierfür $\frac{1}{2^i}$. Für alle n Türme ist dann die Wahrscheinlichkeit, dass das passiert $\frac{n}{2^i}$. Wenn jetzt die Höhe $\geq 3 \log n$ sein soll, dann ist die Wahrscheinlichkeit $\frac{n}{2^{3 \log n}} = \frac{n}{n^3} = \frac{1}{n^2}$. Und dementsprechend ist die Wahrscheinlichkeit für Höhe $c \log n$ und höher $\frac{1}{n^{c-1}}$. Für große n ist dann natürlich schon recht klein.
Also, mit großer Wahrscheinlichkeit p ist die Höhe in $O(\log n)$, d.h. $\exists c > 0 : p > 1 - \frac{1}{n^c}$.
Natürlich muss jetzt auch noch die Schrittanzahl von links nach rechts betrachtet werden. Sei n_i diese Anzahl auf der Ebene i.
Wir suchen jetzt den Erwartungswert von n_i. Der Erwartungswert sieht also so aus

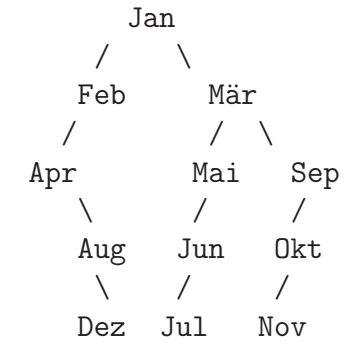
$$
\begin{aligned}
&0 * \frac{1}{2} + 1 * \frac{1}{4} + 2 * \frac{1}{8} + 3 * \frac{1}{16} + \ldots \leq \sum_{j=1}^{\infty} j * \frac{1}{2^{j+1}} \\
&= \frac{1}{4} \sum_{j=1}^{\infty} j * \frac{1}{2^{j-1}} = \frac{1}{4} \sum_{j=1}^{\infty} j * x^{j-1} \\
&= \frac{1}{4} \left[\frac{d}{dx} \sum_{j=1}^{\infty} x^j \right]_{x=\frac{1}{2}} = \frac{1}{4} \frac{d}{dx} \left(\frac{1}{1-x} - 1 \right)_{x=\frac{1}{4}} = \frac{1}{4} \left(\frac{1}{(1-x)^2} \right)_{x=\frac{1}{4}} = 1
\end{aligned}
$$

Seitlich erwartet man also nur eine Schrittlänge von 1. Die Gesamtlänge des Weges ist mit großer Wahrscheinlichkeit $O(\log n)$. Beim Finden haben wir einen Weg, der von oben nach

unten verläuft, also eine Laufzeit von $O(\log n)$. Beim Einfügen haben wir einen Weg von oben nach unten und einen von unten nach oben, also auch $O(\log n)$. Beim Streichen müssen wir auch von oben nach unten gehen, und dann noch den gefundenen Turm wieder nach oben, und auch hier ergibt sich wieder $O(\log n)$.

18.0.5. Binäre Suchbäume

Wir benutzen sie jetzt als Datenstruktur für das Wörterbuchproblem. Es ist ein Binärer Baum, wobei die Knoten mit den Einträgen aus dem Wörterbuch D beschriftet sind, so dass für jeden Knoten v mit Unterbäumen T_l, T_r gilt, dass die Schlüssel in $T_l \leq$ Schlüssel in v und dieser $\leq T_r$.



An den leeren Stellen fügen wir Blätter ein, die für erfolglose Suchen stehen. Nun die Operationen

Operationen

$finde(k)$ schon beschrieben, folge einem Weg von der Wurzel bis zu dem Knoten. Dabei liegt die Laufzeit bei $O(h)$, mit h der Höhe, wobei die Höhe zwischen logarithmisch und linear liegt.

$einfuege(k, v)$ Als erstes nehmen wir an, dass alle Schlüssel verschieden sind. Dann können wir uns ein $bbeinfuege(k, v, B)$ definieren, welches in den Binärbaum B einfügt.

```
if B = ein Blatt then schaffe einen Baum mit
 leeren Blättern und k in der Wurzel
else if (k < Wurzel(B)) then bbeinfuege(k,v,Tl)
else if(k > Wurzel(B)) then bbeinfuege(k,v,Tr)
```

$bbeinfuege$ durchläuft einen Weg von der Wurzel bis zu einem Blatt und braucht konstante Zeit pro Knoten. Demnach also wird $O(h)$ Zeit benötigt.

$streiche(k, v)$ Als erstes suchen wir nach k. Falls mindestens ein Kind von k ein Blatt ist, dann ersetze den Knoten durch das Kind. Wenn kein Kind ein Blatt ist, dann finden wir den Knoten y mit maximalem Schlüssel im linken Teilbaum. Ersetze Inhalt von k

durch Inhalt von y. Dann hat y kein rechtes Blatt mehr und deshalb kann einfach der ursprüngliche y-Knoten durch dessen linkes Kind ersetzt werden.
Die Laufzeit ist hier fürs Streichen $O(h)$.

19. Vorlesung, 18.12.2007

Die Laufzeit ist $\Omega(\log n) = h = O(n)$. Die Laufzeit ist genau dann logarithmisch, wenn auf jeder Ebene i 2^i Knoten liegen. Da ist dann $h = O(\log n)$. Im schlechtesten Fall haben wir $h = n$.
Jetzt ist die Frage, ob man den binären Suchbaum so organisieren kann, dass man immer mit dem guten Fall rechnen kann, also die logarithmische Höhe garantiert ist.

19.0.6. AVL-Bäume

Definition 3. *Ein binärer Suchbaum heißt AVL-Baum genau dann, wenn für jeden inneren Knoten v gilt, dass sich die Höhen von dessen linken und rechten Teilbaum um höchstens eins unterscheiden.*

Sei bei einem solchen Baum die Höhe $= h$. Jetzt wollen wir das so konstruieren, dass er bei der Höhe möglichst viele Knoten hat. Das ist natürlich der Fall, wenn jeder Ebene 2^i Knoten enthält, das macht dann $n = 2^h - 1$ Knoten insgesamt.
Wenn der Baum eine minimale Anzahl von Knoten besitzen soll, dann sieht das so aus, dass ein Teilbaum der Wurzel die Höhe $h - 2$ hat und der andere $h - 1$. Wir haben also eine Rekursionsgleichung für n_h.

$$n_0 = 0$$
$$n_1 = 1$$
$$n_h = n_{h-2} + n_{h-1} + 1$$

n_h ist also sowas ähnliches wie die Fibonaccizahlen f_i. Es gilt daher $n_h \geq f_h$. Wir wissen $f_h \geq \Phi^{h-1}$, mit $\Phi = \frac{\sqrt{5}+1}{2}$ der goldene Schnitt.
Demnach ist audh $n_h \geq \Phi^{h-1}$ und $\log n_h \geq (h-1)\log\Phi$. Demnach $\frac{1}{\log\Phi}\log n_h + \log\phi \geq h$.
Es gilt für h im AVL-Baum, dass

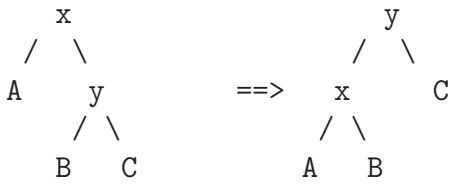
$$1,44 * \log n \geq h \geq \log(n+1)$$

Also $h = \Theta(\log n)$.

Es gilt daher für ein Wörterbuch mit n Einträgen, dass in einem AVL-Baum gespeichert ist, dass die Operation $finde(k)$ immer $O(\log n)$ Zeit kostet. Allerdings können durch Einfügen und Streichen von Knoten der Baum außer Balance geraten, so dass dann der Höhenunterschied der Teilbaume > 1 ist. Wir müssen dann also den Baum rebalancieren.
Die Operationen dazu sehen wie folgt aus:

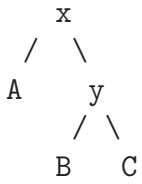
- beim Einfügen: Für einen Knoten x wird der rechte Unterbaum o.B.d.A. (links symmetrisch) zu hoch, also um zwei höher als der andere.

  Fall a.1 der Unterbaum rechts, rechts (C) ist zu hoch.



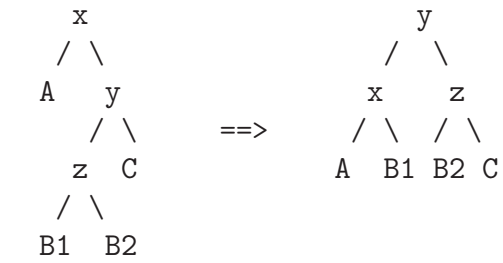
  Wobei die Höhe von $C = B + 1$ und von $A = B - 1$.
  Da machen wir das jetzt so, dass wir den y in den Vater packen, und dort als linkes Kind x schreiben, der dann als linken Teilbaum A und als rechten Teilbaum B erhält. Das C wird dann rechtes Kind von y.
  Dieses Verfahren wird auch Rotation genannt.

  Fall a.2

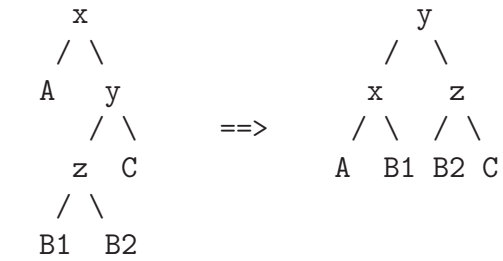


  Wobei diesmal aber B der Höhere ist. Rotation klappt hier nicht, deshalb müssen wir uns alles noch ein wenig genauer angucken. Wieder zwei Unterfälle

  1. rechts-links-links Teilbaum ($B1$) ist zu hoch



     Das nennt man auch Doppelrotation.
  2. Jetzt ist der rechts-links-rechts Teilbaum ($B2$) zu hoch



     Auch hier wieder eine Doppelrotation.

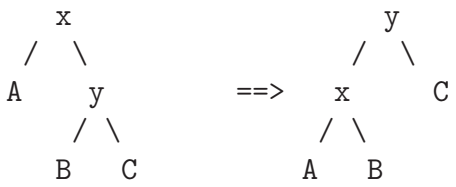
  Beim Einfügen mit einer dieser Operationen ändert sich die Höhe des Unterbaumes mit Wurzel x nicht. Damit ist höchstens eine Rebalancierungsoperation erforderlich. Wo das geschieht, ist durch Finden des Weges von der Wurzel zum

eingefügten Knoten zu ermitteln, das dauert $O(\log n)$. Das Rebalancieren geht dann in konstanter Zeit. Also wieder insgesamt $O(\log n)$ fürs Einfügen an Zeit benötigt.

20. Vorlesung, 20.12.2007

- Jetzt wollen wir streichen. Durch das normale Streichen wie in Binärbäumen kann sich die Höhe von Teilbäumen verkleinern um höchstens eins und die Balance des Vaters wird somit 2 oder -2. x sei wieder der niedrigste Knoten außer Balance. Jetzt gibt's natürlich wieder mehrere Fälle. Wir gehen jetzt o.B.d.A. davon aus, dass der linke Teilbaum zu kurz geworden ist (rechts ist ja analog).

  Fall b.1 Hier ist C zu lang

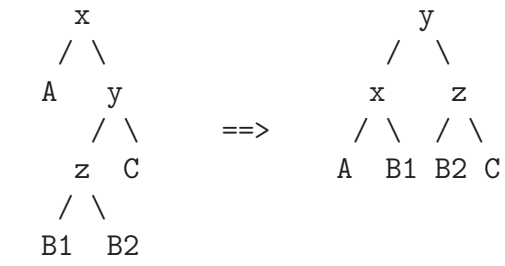
  

  Wir rotieren hier wieder, wie man ja sieht. Dadurch, dass wir jetzt hier die Balance wiederhergestellt haben, ist die Höhe um eins kleiner geworden. Dadurch kann es sein, dass weiter oben liegende Knoten auf dem Pfad außer Balance geraten sind.

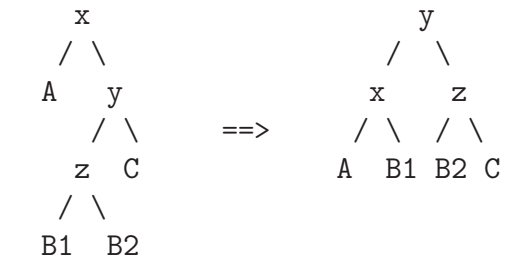
  Fall b.2 Hier ist jetzt das B das längste

  Wieder zwei Unterfälle

  1. rechts-links-links Teilbaum ($B1$) ist zu hoch

  2. Jetzt ist der rechts-links-rechts Teilbaum ($B2$) zu hoch

  Auch hier kann es wieder sein, dass durch das Rotieren weiter oben liegende Knoten außer Balance geraten und so entlang des Pfades zur Wurzel geprüft und

rotiert werden muss.

Insgesamt wissen wir über das Streichen in AVL-Bäumen, entstehende Inbalancen lassen sich durch Rotation bzw. Doppelrotation bereinigen. Eine solche Operation kostet konstante Zeit. Eventuell muss entlang des gesamten Suchpfades welche durchgeführt werden. Damit hat man also eine Laufzeit von $O(\log n)$.

Wir kriegen also alle drei Operationen, suchen, einfügen und löschen in $O(\log n)$ für alle Einträge durchführbar mit Erhaltung der AVL-Eigenschaft.

20.0.7. (ab)-Bäume

Das sind im allgemeinen keine Binärbäume. Mit $\rho(v)$ werden im folgenden die Anzahl der Kinder von v bezeichnet. Zudem seien $a, b \in \mathbb{N}$ mit $a \geq 2$ und $b \geq 2a - 1$.

Definition 4. *Ein Baum T heißt (ab)-Baum, wenn gilt, dass*

1. *alle Blätter haben gleiche Tiefe*
2. *für alle Knoten v ist $\rho(v) \leq b$*
3. *für alle Knoten v außer der Wurzel gilt, dass $\rho(v) \geq a$.*
4. *für die Wurzel r gilt, dass $\rho(r) \geq 2$.*

Ein (a,b)-Baum ist für jede Zahl von Blättern möglich.
Die Anzahl der Blätter eines (a,b)-Baums der Höhe h ist maximal, wenn jeder innere Knoten b Kinder hat. Somit hat man dann bei Höhe h natürlich b^h Blätter. Minimal hat die Wurzel 2 Kinder und die restlichen Knoten dann jeweils a Kinder. Demnach ergibt sich $2a^{h-1}$.

Bemerkung Für einen (a,b)-Baum der Höhe h mit n Blättern gilt

1. $2a^{h-1} \leq n \leq b^h$
2. $\log_b n = \frac{\log n}{\log b} \leq h \leq 1 + \log_a \frac{n}{2}$
3. $\log_b n \leq 1 + \log_a(n/2)$

Also $h = O(\log n)$ da $\log_a n = \frac{1}{\log n} \log a$

21. Vorlesung, 8.1.2008

Die Abspeicherung von n Daten (Schlüsseln) $S = \{x_1, \ldots, x_n\}$ mit $x_1 < x_2 < \ldots < x_n$ in einem (a,b)-Baum mit n Blättern soll realisiert werden.

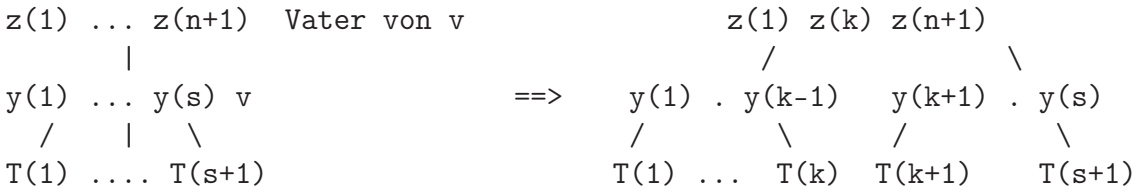
1. Diese Schlüssel stehen dabei aufsteigend sortiert von links nach rechts in den Blättern.
2. in jedem inneren Knoten v mit k Kindern stehen $y_1 \ldots y_{k-1}$ aufsteigend sortierte Schlüssel, wobei y_j das größte Element im $j$ten Unterbaum von v ist mit $j = 1, \ldots, k-1$.

Als Beispiel wieder die Wochentage in einem (2,3)-Baum

Jetzt zu der Implementierung des Wörterbuchproblems. Hierbei steht ein Knoten für eine Liste aus Schlüsseln.

$finde(k)$ Sei $y_1 \ldots y_k$ die Beschriftung der Wurzel. Finde i mit $y_i < k < y_{i+1}$ bzw. $i = 0$ falls $k \leq y_1$ und $i = k$ falls $k > y_k$.
Durchsuche nun rekursiv den $i$ten Unterbaum der Wurzel. Falls man bei einem Blatt angelangt ist, dann hat man das k gefunden. Ist es kein Blatt, dann ist k im Baum nicht vorhanden.
Diese Suche durchläuft einen Weg im Baum von der Wurzel bis zu einem Blatt. Dabei hat der Weg wie besprochen die Länge $\Theta(\log n)$. Man benötigt konstante Zeit für jeden Knoten des Weges (a und b sind ja Konstanten), also gilt für die Laufzeit des Suchens mal wieder $\Theta(\log n)$.

$einfuege(k, v)$ Als erstes führen wir $finde(k)$ aus, was ein Blatt w liefert. Falls k in w ist, sind wir fertig. Ansonsten schaffen wir ein neues Blatt, welches mit k beschriftet ist und hängen es links von w an den Vater v von w an. Füge dann k an entsprechender Stelle von v ein. Das funktioniert natürlich nicht immer, weil durch das Einfügen ein Knoten entstehen kann, der mehr als b Kinder hat.
Falls die Zahl der Kinder ($\rho(v)$) also zu groß ist und $\rho(v) = b + 1$, dann spalte v in zwei Knoten v' und v'' mit $\lfloor \frac{b+1}{2} \rfloor$ und $\lceil \frac{b+1}{2} \rceil$ Kindern. Beachte, dass beide Brüche $\geq a$ sind nach Definition von b.



Man muss nun gegebenenfalls diesen Schritt auf den Vater von v angewendet werden, im schlechtesten Fall bis hoch zu Wurzel. Falls auch diese gespalten werden muss, so schaffe darüber eine neue Wurzel mit 2 Kindern.
Die Laufzeit vom Einfügen ist setzt sich aus dem Weg von der Wurzel zum Blatt zusammen und dann muss man eventuell noch bis zur Wurzel zurücklaufen und die entsprechenden Knoten aufspalten. Kostet natürlich $\Theta(\log n)$ Zeit, da man pro Knoten konstante Zeit benötigt.

$streiche(k)$ Mit Hilfe von $finde(k)$ finden wir wieder das Blatt w, welches k enthält, falls es überhaupt vorhanden ist. Sei v der Vater von w. Dann entfernt man w sowie den entsprechenden Eintrag in v, falls w nicht das rechte Kind von v ist. Andernfalls müssen wir y, das ist der rechteste Schlüssel in v, nehmen und k im Baum finden und durch y ersetzen. Dies funktioniert so, dass man so lange nach oben läuft, solange man rechtestes Kind vom Vaterknoten ist. Falls v immer noch a Kinder hat, sind wir fertig. Falls v die Wurzel ist und nur noch ein Kind hat, dann entferne v und mache das Kind zur Wurzel. Ansonsten hat v jetzt $a-1$ Kinder. y sei wieder ein benachbarter Bruder von v. Wir gucken nach, wie viele Kinder y hat. Falls es a Kinder hat, dann verschmilzt man v und y zu einem Knoten mit $2a-1 \leq b$ Kindern. Falls y mehr als a Kinder hat, dann kann v das größte bzw. kleinste Kind von y adoptieren. Es müssen natürlich immer schön dann die Einträge in den oberen Knoten aktualisiert werden. Eventuell muss man diesen Vorgang auch öfters ausführen, maximal bis zur Wurzel.
Als Laufzeit ergibt sich mal wieder $\Theta(\log n)$.

22. Vorlesung, 10.1.2008

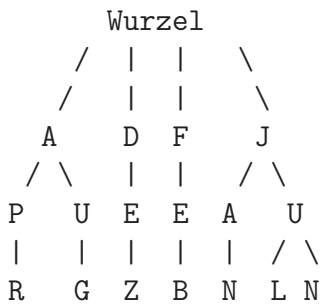
(2,3)-Bäume sind übrigens eine Alternative zu balancierten binären Suchbäumen.

Anwendung bei Hintergrundspeicher

Zugriff auf Hintergrundspeicher kostet mehrere tausend Mal mehr Zeit (oder noch mehr) als der Zugriff auf den Hauptspeicher. Deshalb sollte man logischerweise die Anzahl der Zugriffe minimieren. Ein Hintergrundspeicherzugriff liefert einen größere Datenbereich (Seite) und man kann jetzt die (a,b)-Bäume so einsetzen, der so groß gewählt ist, dass ein Knoten eine solche Seite enthält. a und b liegen also in einer Größenordnung von mehreren Hundert. Die Kanten im (a,b)-Baum entsprechen nun den Hingergrundspeicherzugriffen. Dazu sind für die Wörterbuchoperationen ungefähr $\log_a n$ notwendig. Dabei liegt $\log_a n$ in der Größenordnung von 3 oder 4, auch bei Datenmengen von Millionen von Daten.
Diese Bäume werden auch B-Bäume genannt. Üblicherweise wird $b = 2a - 1$ gewählt. Dann gibt es auch noch B*-Bäume und so nen Spaß, das ist aber alles recht ähnlich. Erfunden wurden die Teile übrigens 1972 von Bayer und McCreight oder so.

22.0.8. Tries

Das ist eine Datenstruktur für das Wörterbuchproblem für eine Menge $S \subset \Sigma^*$ von Wörtern über einem Alphabet Σ. Sei $S = \{w_1, \ldots, w_n\}$.
Alle inneren Knoten, außer der Wurzel, sind mit einem Zeichen $\in \Sigma$ markiert. Die Kinder eines Knotens sind von links nach rechts bezüglich ihrer Markierungen aufsteigend geordnet. Dabei nehmen wir natürlich eine Ordnung auf Σ an.
Jedes Wort w von S befindet sich auf dem Weg von der Wurzel bis zu einem Blatt.
Als Beispiel hier die Monatsnamen ...



Eigenschaften

Ein Trie T enthalte die Menge $S = \{w_1, \ldots, w_n\}$. Sei $|\Sigma| = d$. $l = \max_{i=1,\ldots,n} |w_i|$ $m = \sum_{i=1}^{n} |w_i|$

1. Jeder innere Knoten hat $\leq d$ Kinder
2. n Blätter
3. Höhe von T ist l.
4. Anzahl der inneren Knoten von T ist $\leq n + 1$.
5. kein Wort von S darf Präfix eines anderen Wortes sein. S präfixfrei ist erreichbar durch Anhängen eines Sonderzeichens $\$ \notin \Sigma$ an jedes Wort.

Jetzt wollen wir natürlich wieder damit ein Wörterbuch implementieren.

$finde(w)$ $w = a_1, \ldots a_k$. Von der Wurzel aus Knoten v der mit a_i beschriftet ist entlanggehen, bis man beim Blatt ist. Wenn man einen solchen Knoten mit korrekter Beschriftung einmal nicht findet, so wissen wir, dass der Knoten nicht vorhanden ist.
Die Laufzeit ist hierbei $O(k)$, wobei $k = |w|$, wenn der Zugriff von einem Knoten auf ein Kind mit gegebener Markierung in $O(1)$ Zeit möglich ist. Die Datenstruktur für einen Knoten könnte z.B. ein Array von Zeigern der Größe d sein. Für ein Zugriff auf ein Kind braucht man also $O(1)$ und Speicherplatz pro Knoten von $\Theta(d)$, das ist eine Konstante für ein festes Alphabet (natürlich nicht, wenn man jetzt z.B. Unicode hat ...). Bei sowas würde man dann lieber eine Liste nehmen, da dauert der Zugriff dann allerdings $\Theta(d)$.

$einfuege(w)$ Mit $finde(w)$ bekommen wir die Stelle, wo wir noch einfügen müssen. Ab dort muss man neue Knoten schaffen und die verbleibenden Buchstaben einfügen.
Die Laufzeit ist natürlich wieder $O(|w|)$.

$streiche(w)$ Mit $finde(w)$ wird wieder das Blatt gefunden, dass w zugeordnet ist. Dieses Blatt wird entfernt. Dann wird auf dem Weg zurück zur Wurzel solange der Knoten gelöscht, bis ein Knoten mehr als ein Kind hat.
Auch hier ist die Länge wieder in $O(|w|)$.

Die Wörterbuchoperationen für ein Wort w in einem Trie in Zeit $O(|w|)$ durchführbar.

Anwendung

Eine Anwendung ist die Wortsuche in einem Text T. Dieser Text besteht aus eine Abfolge von Wörtern $w_1, \ldots, w_n$, die durch Trennsymbole getrennt sind. Zudem ist das Wort X gegeben. Nun sollen alle Vorkommen von X in T sollen gefunden werden, d.h. jedes $w_i = X$.

1. Verarbeite den Text und schreibe alle Wörter in einen Trie. Falls ein Wort mehrfach vorkommt, so macht man eine Liste der entsprechenden Stellen am entsprechenden Blatt im Trie.

2. Suche nach dem Wort X im Trie

Für Schritt eins braucht man also $O(|T|)$ und für Schritt zwei wie oben besprochen $O(|X|)$. Also insgesamt eine Laufzeit von $O(|T| + |X|)$. Und das ist um einiges besser als eine Bruteforce-Suche, die nämlich $O(|T| * |X|)$ Zeit benötigt. Wenn die Vorverarbeitung nur einmal ausgeführt wird und Schritt zwei, die Anfrage, oft, so ist das ziemlich gut.
Bei der ganzen Chose sollte der Speicherplatz natürlich nicht außer Acht gelassen werden.
Ein allgemeineres Problem wäre die Mustererkennung für Strings (oder auch String-Matching genannt). Hierbei ist wieder ein Wort X gegeben und ein Text T und wieder sind alle Vorkommen von X in T gesucht. Dabei ist zu beachten, dass T nicht in Wörter zerfällt. Nach Bruteforce würde das wieder $O(|T| * |X|)$ gehen, es gibt allerdings auch hier Algorithmen, die das in $O(|T| + |X|)$ schaffen.
Eine wichtige Anwendung hiervon ist die Genomanalyse in der Bioinformatik.

Tries können auch komprimierter dargetellt werden. So können Ketten von Knoten ohne Verzweigungen einfach durch nur ein Blatt dargestellt werden. Das nennt man dann "Patricia trie". Und noch besser kann man das so machen, dass der Text bzw. die Wörter in einem Array abgespeichert sind und man nur den Index, die Stelle und die Länge in den Knoten angibt.

23. Vorlesung, 15.1.2008

23.1. Kapitel 6: Graphenalgorithmen

23.1.1. Definitionen

Ungerichteter Graph

Ein ungerichteter Graph ist ein Paar $G = (V, E)$, wobei V eine nichtleere endliche Menge ist. Die Elemente von V sind die Knoten. E ist eine Menge von zweielementigen Teilmengen von V, diese Elemente nennt man Kanten.
Fall $e = \{u, v\}$ eine Kante von G ist, so heißt u adjazent zu v, und u und v heißen inzident zu e, oder auch Endpunkte von e.
Beispiele für Graphen in der Realität sind z.B. Straßen- oder Eisenbahnnetze, Rechnernetze, soziale Netzwerke etc.

Sei $G = (V, E)$ eine Folge $v_1, v_2, \ldots, v_n$ von Knoten mit $\{v_j, v_{j+1}\} \in E$, mit $j = 0, \ldots, n$, dann heißt das ein Weg in G, oder auch Pfad. Ein Weg heißt einfach, wenn jeder Knoten höchstens einmal vorkommt. n ist hierbei die Länge des Weges.

Beim sozialen Netzwerk sagt man (Hypotese), dass zwischen zwei beliebigen Menschen auf der Welt es einen Weg der Länge ≤ 7 gibt.

Ein Weg $v_1, \ldots, v_n$ heißt Kreis, wenn $v_1 = v_n$ ist. Man unterscheidet auch wieder den einfachen Kreis, wenn jeder Knoten wieder nur einmal besucht wird.
Ein Graph $G' = (V', E')$ mit $V' \subset V$ und $E' \subset E$ heißt Teilgraph von $G = (V, E)$.
Falls $V' \subseteq V$ und $E = \{\{v, u\} \in E | u, v \in V'\}$ dann nennt man das den induzierten Teilgraph.
Ein Graph heißt kreisfrei oder azyklisch, wenn er keinen einfachen Kreis enthält. Im ungerichteten Graphen ist hier auch noch Wald eine Bezeichnung für einen kreisfreien Graphen.
Ein Graph heißt zusammenhängend, falls es zwischen beliebigen Knoten $u, v \in V$ einen Weg gibt.
Ein Teilgraph $G' = (V', E')$ von G heißt Zusammenhangskomponente, genau dann wenn G' der von V' induzierte Graph ist, G' zusammenhängend ist und es gibt kein $v \in V, E' \subset E$, so dass $G' = (V' \cup \{v\}, E')$ zusammenhängend ist.
Ein zusammenhängender azyklischer Graph heißt Baum. Nach dieser Definition hat der Baum übrigens keine Wurzel. Man kann einen Knoten als Wurzel festlegen, dann spricht man von

sogenannten Wurzelbäumen. Dann erhalten die Kanten des Baumes eine kanonische Richtung, und zwar von der Wurzel weg.
Im ungerichteten Fall ist der Grad eines Knoten gleich der Zahl seiner Nachbarn.

Gerichtete Graphen

$G = (V, E)$ wie bei ungerichteten Graphen, außer dass $E \subset V \times V$ mit $e = (u, v) \subset E$, wobei u der Anfangspunkt und v der Endpunkt. Zeichnerisch wurde man das natürlich durch einen Pfeil von u nach v darstellen. Die anderen Definitionen funktionieren analog.
Einen gerichteten Graphen nennt man stark zusammenhängend, genau dann wenn es für beliebige $u, v \in V$ einen gerichteten Weg von u nach v gibt. Jetzt kann man natürlich auch über starke Zusammenhangskomponenten reden. Das sind Teilgraphen, die stark zusammenhängend sind und nicht erweitert werden können, so dass sie stark zusammenhängend bleiben.
Im gerichteten Graphen spricht man von Ingrad und Ausgrad. Ein Ingrad liegt vor, wenn $\{u|(u, v) \in E\}$, und ein Ausgrad, wenn $\{u|(v, u) \in E\}$.
Ein gerichteter Baum liegt dann vor, wenn jeder Knoten Ingrad 1 hat, bis auf einen mit Ingrad 0, den man Wurzel nennt. Zudem muss der Graph kreisfrei sein.

23.1.2. Datenstrukturen für Graphen

Adjazenzlisten

Hier werden alle Knoten aufgelistet und für jeden Knoten eine Liste der zu ihm adjarzenten Knoten aufgelistet. Der benötigte Platz hierbei mit $n = |V|$ und $m = |E|$ ist $\Theta(n + m)$.

Adjanzenzmatrix

Das ist eine boolesche $n \times n$ Matrix A. Die Knoten werden hierbei durchnummeriet. Es steht eine 1, falls $(u, v) \in E$ ansonsten eine 0. Bei einem ungerichteten Graphen ist die Matrix symmetrisch. Der Platzbedarf hierbei beträgt $\Theta(n^2)$.
Bei vielen Anwendungen ist $m \ll n^2$ und m ist also in der Größenordnung von n, das nennt man dann auch dünnbesetzte Graphen. In diesem Fall sind Adjazenzlisten wesentlich günstiger.

24. Vorlesung, 17.1.2008

24.0.3. Der abstrakter Datentyp Graph

Folgende Operationen müssen bedacht werden

$knoten()$ liefert Liste aller Knoten

$adjListe(v)$ liefert eine Liste aller adjazenten Knoten von einem Knoten v.

$knElement(v)$ liefert das in v abgespeicherte Element

$kaElement(v, w)$ liefert das in der Kante (v, w) abgespeicherte Element bzw. das Element, mit der die Kante markiert ist.

$adj(v, w)$ liefert einen Wahrheitswert. Und zwar ist es wahr, wenn (v, w) adjazent sind, ansonsten falsch.

$setzeKnEl(v, x)$ setze das Element in v auf x

$setzeKaEl(v, w, x)$ setzt das Kantenelement von (v, w) auf x

$streicheKnoten(v)$ entfernt den Knoten v und die dazu inzidenten Kanten.

$streicheKante(v, w)$ entfernt die Kante (v, w).

Jetzt zu den Laufzeiten. Wir nehmen an, dass der Graph n Knoten hat und m Kanten.

Operation	Adjazenzliste	Adjazenzmatrix
1	$O(n)$	$O(n)$
2	$O(1)$	$O(n)$
3	$O(1)$	$O(1)$
4	$grad(v)$	$O(1)$
5	$grad(v)$	$O(1)$
6	$O(1)$	$O(1)$
7	$grad(v)$	$O(1)$
8	$O(m)$	$O(n)$
9	$grad(v)$ im gerichteten Fall und $grad(v) + grad(w)$ im ungerichteten Graphen	$O(1)$

24.0.4. Traversierung von Graphen

Man will systematisch alle Knoten aufsuchen durch Durchlaufen der Kanten.

Tiefensuche

In Englisch auch depth-first-search, abgekürzt dann zu DFS.
Die Idee hierbei ist, dass man ausgehend von einem Knoten v einen Nachbarn w besucht und dann einen Nachbarn von w usw. Dabei geht man dann zurück, wenn es keine unbesuchten Nachbarn mehr gibt. Ma will hier so schnell wie möglich möglichst tief in den Graphen vordringen.
Dabei nennt man die von der Tiefensuche durchlaufenden Kanten die "Baumkanten" und die übrig bleibenden die Rückwärtskanten.

Algorithmus Jetzt gucken wir uns den Algorithmus einmal in Pseudocode an. Wir beginnen hierbei in Knoten v.

```
DFS(G,v) {
 markiere v als besucht, verarbeite es in irgendeiner Form
 forall u adjanzent zu v do {
   if (u nicht besucht) {
     markiere u als besucht (optional (u,v) als Baumkante markieren)
     DFS(G,u) (optional (u,v) als Rückwärtskante markieren)
   }
 }
}
```

Was ist jetzt hier die Laufzeit?
Man braucht erstmal $O(n)$ Zeit, um Knoten als besucht zu markieren. Dann braucht man für die nächsten Schritte $O(m)$, da für jedes Element in jeder Adjazenzliste konstante Zeit verbraucht.
Die Laufzeit ist also insgesamt $O(n+m)$.

Der Algorithmus funktioniert übrigens für gerichtete sowie ungerichtete Graphen.

Behauptung $DFS(G,v)$ besucht genau die Knoten, die von v aus durch einen Weg erreichbar sind.
Das wird jetzt noch schick bewiesen:

$\Leftarrow$ Sei u von v aus erreichbar. Dann gibt es einen Weg $v_0, v_1, \ldots, v_n$ mit $v_0 = v$ und $v_n = u$.
Wir führen einen Beweis per vollständiger Induktion

Induktionsanfang $n = 0$ v wird besucht, damit ist man fertig.

Induktionsschritt Von v aus Weg er Länge $n-1$ zu v_{n-1}. Daraus folgt nach Induktionsvoraussetzung, dass v_{n-1} besucht wird in $DFS(G,v)$, d.h. $DFS(G,v_{n-1})$ wird innerhalb von $DFS(G,v)$ aufgerufen. In diesem Aufruf wird u getestet, da

es adjazent zu v_{n-1} ist. Ist das u bereits markiert, dann sind wir fertig. Wenn nicht, dann wird es jetzt in der letzten Zeile besucht.

$\Rightarrow$ u werde von $DFS(G, v)$ besucht. Dann ist entweder $u = v$ oder es muss $DFS(G, u)$ innerhalb von $DFS(G, v)$ aufgerufen werden. Wir machen jetzt eine Induktion über die Schachtelungtiefe t des Aufrufs.

Induktionsanfang Bei $t = 0$ gilt $u = v$ ist nach Definition ein Weg.

Induktionsschritt Sei v_t der Nachbar von v, in dessen Aufruf $DFS(G, v_t)$ der Aufruf $DFS(G, u)$ in Tiefe $t - 1$ liegt. Nach Induktionsvoraussetzung existiert ein Weg von v_t nach u, und da v_t mit v benachbart ist also auch ein Weg von v nach u.

Eigenschaften Die Tiefensuche erlaubt folgende Operationen in Zeit $O(n + m)$:

- Entscheiden, ob G zusammenhängend ist. Das macht man so, dass man einmal die Tiefensuche aufruft und falls alle Knoten dann markiert werden, dann weiß man, dass der Graph zusammenhängend ist, natürlich nur im ungerichteten Graphen.
- Falls G zusammenhängend ist, kann ein spannender Baum berechnet werden, das ist ein Baum, der den ganzen Graphen aufspannt, also alle Knoten enthält. Den nennt man dann auch passenderweise DFS-Baum.
- Berechnung der Zusammenhangskomponenten. Hierbei muss man dann DFS wiederholt aufrufen, bis alle Knoten als besucht markiert sind.
- Berechnung eines Weges zwischen zwei Knoten u und v, falls er existiert.
- Testen auf Kreisfreiheit, das tritt auf, wenn man keine Rückwärtskanten hat.

25. Vorlesung, 22.1.2008

Breitensuche

Auf Englisch dementsprechend "breadth first search" abgekürzt BFS.
Im Gegensatz zur Tiefensuche guckt man sich hier immer erst die ganzen Nachbarn eines Knotens an und dann dessen Umgebung und so weiter, bis man alle Knoten besucht hat. Auch hier erzeugt die Breitensuche einen Baum, den BFS-Baum. Die Kanten, die nicht zum Baum gehören, nennt man Querkanten, die anderen heißen wie im DFS-Baum Baumkanten.

Algorithmus Der Algorithmus benutz eine Warteschlange Q.

```
BFS(G,v) {
 Q.enqueue(v)
 markiere v als gesehen
 while Q nichtleer {
   w = Q.dequeue()
   (bearbeite w)
   forall u adjazent zu w {
     if (u nicht gesehen) {
       Q.enqueue(u)
       markiere u als gesehen
       [(w,u) ist Baumkante]
     }
     else
       [(w,u) ist Querkante]
   }
 }
}
```

Natürlich analysieren wir wieder die Laufzeit: In der While-Schleife betrachten wir alle Knoten, also braucht man zur Abarbeitung von n Knoten $O(n)$. In der $forall$ Schleife wird jede Adjazenzliste einmal inspiziert und man braucht konstante Zeit pro Eintrag. Insgesamt hier also die Gesamtlänge aller Adjazenzlisten, also $2m$, was also $O(m)$ entspricht.
Insgesamt ergibt sich also $O(n+m)$ als Laufzeit der Breitensuche.

Eigenschaften Und zwar von BFS auf ungerichteten Graphen.

- $BFS(G, v)$ besucht alle Knoten der Zusammenhangskomponente von v.
- Im BFS-Baum ist der Weg von v zu jedem Knoten w der kürzeste von v nach w den es gibt im Graphen G.
- Kanten, die nicht im BFS-Baum sind, verlaufen höchsten zwischen Knoten, die eine Ebene auseinander liegen. Es gibt also nur Querkanten und Baumkanten, keine Rückwärtskanten wie im DFS-Baum.

Folgerungen G sei ein ungerichteter Graph mit n Knoten und m Kanten. Die Breitensuche kann nun benutzt werden um in $O(m + n)$ Zeit folgendes zu berechnen:

- Testen, ob G zusammenhängend ist bzw. die Zusammenhangskomponenten finden
- Berechnung eines spannenden Baumes, falls G zusammenhängend ist. Ansonsten kriegt man für jede Zusammenhangskomponente einen spannenden Baum, insgesamt einen spannenden Wald
- Von einem Startknoten s aus kann man alle kürzesten Wege zu den anderen Knoten der Zusammenhangskomponente berechnen.
- Testen, ob G azyklisch ist, bzw. die Berechnung eines Kreises falls das nicht der Fall ist. Und zwar gibt es genau dann einen Kreis, wenn es eine Querkante gibt.

25.0.5. Topologisches Sortieren

Man will die Knoten eines azyklischen gerichteten Graphen sortieren.
Ein Beispiel wäre, dass ein Projekt aus kleineren Teilprojekten besteht, welche wir als Knoten des Graphen annehmen. Und es gibt eine gerichtete Kante zwischen zwei Teilprojekten A und B, wenn B erst nach Vollendung von A ausgeführt werden kann. Und gesucht ist natürlich eine mögliche Abarbeitungsreihenfolge aller Teilprojekte.

Definition 5. *Es sei $G = (V, E)$ ein gerichteter Graph. Eine topologische Sortierung der Knoten ist eine Reihenfolge $v_1, \ldots, v_n$ sodass $(v_i, v_j) \in E \Rightarrow i < j$.*

Satz 3. *G hat eine topologische Sortierung genau dann wenn G azyklisch ist.*

Beweis 3. *Wir zeigen Hin- und Rückrichtung.*

$\Rightarrow$ *klar, da alle Kanten von links nach rechts gehen, daraus folgt, dass es keinen Kreis gibt, sonst müsste es ja auch eine rechts-links Kante geben.*

$\Leftarrow$ *Angenommen G ist azyklisch. Dann gibt es mindestens einen Knoten mit Ingrad 0, in den also keine Kanten hineinführen. Und es gibt mindestens einen Knoten mit Ausgrad 0. (Das ist so, da wenn man annimmt, dass jeder Knotn Ausgrad ≥ 1 hat, dann beliebig lange Wege existieren. Ein Weg der Länge $n + 1$ bei n Knoten ist dann natürlich nur möglich, wenn man einen Knoten mehrfach besucht, also einen Kreis hat)*
Wir machen jetzt eine vollständige Induktion über die Knotenanzahl n:

Induktionsanfang $n = 1$ *ist kreisfrei, d.h. es gibt keine Kante. Die Folge, die nur aus dem Knoten besteht, ist natürlich eine topologische Sortierung.*

Induktionsschritt $n-1 \to n$. *Nach der Anmerkung oben existiert ein Knoten* v_1 *mit Ingrad 0. Diesen listen wir als ersten auf.* G' *sei der Graph, der aus* G *entsteht, wenn* v_1 *und alle inzidenten Kanten entfernt werden.* G' *ist wieder ein azyklischer Graph mit* $n-1$ *Knoten. Nach Induktionsvoraussetzung folgt, dass es eine topologische Sortierung* $v_2, \ldots v_n$ *in* G' *existiert. Daraus folgt also das* $v_1, v_2, \ldots, v_n$ *die topologische Sortierung für* G *ist.*

26. Vorlesung, 24.1.2008

Algorithmus

Ein Algorithmus folgt schon direkt aus dem Beweis.

```
topSortieren(G) {
 v := ein Knoten mit Ingrad 0
 gib v aus
 G := G ohne v und zu v inzidente Kanten
 topSortieren(G)
}
```

Alternativ könnte man das auch nichtrekursiv machen:

```
while (G nichtleer) {
  v := Knoten mit Ingrad 0
  gib v aus
  entferne v und alle inzidenten Kanten von v aus G
}
```

Die Knoten mit Ingrad 0 finden, kann man so machen, dass man am Anfang alle Adjazenzlisten durchläuft und für alle Knoten die Häufigkeit zählt, d.h. den Ingrad. Dann macht man eine Liste der Knoten mit Ingrad 0. So kann dann das tatsächliche Herausnehmen von einem solchen Knoten in $O(1)$ erledigen. Im letzten Schritt muss dann der Zähler aller adjazenten Knoten zu v um eins verringert werden. Die Knoten, bei denen der Zähler null wird, werden an die Liste der Knoten mit Ingrad 0 rangehangen.
Wir interessieren und für die Laufzeit. Das Prüfen, ob der Graph leer ist, geht in konstanter Zeit, das Herausnehmen eines Knotens v und das Ausgeben wie oben erwähnt auch. Im letzten Schritt benötigen wir so lange, wie es zu v adjazente Knoten gibt.
Insgesamt ergibt sich dann wieder $O(n+m)$.

26.0.6. Kürzeste Wege

Wir wollen von einem Knoten aus die kürzesten Wege ermitteln.
Es sei ein gerichteter Graph $G = (V, E)$ gegeben, der Kantengewichte besitzt. Das Gewicht ist eine reelle Funktion $c : E \to \mathbb{R}^+$. Außerdem ist ein Startknoten $s \in V$ bekannt. Für jeden Knoten aus V ist die Länge bzw. das minimale Gewicht des Weges von s nach v zu bestimmen. Das nennt man dann den kürzesten Weg von s nach v.

In Englisch sagt man dazu auch "singel source shortest paths"
Oft ist der kürzeste Weg zwischen zwei Knoten $s, t \in V$ zu suchem, aber das geht leider auch nicht effizienter oder schneller, als alle von s ausgehenden kürzesten Knoten zu suchen.

Anwendungen Ein klassisches Beispiel ist natürlich hier der Routenplaner. Die Knoten stellen hierbei die Kreuzungen dar und die Kanten die Straße. Das Gewicht ist dann beliebig, sowas wie Entfernung, benötigte Zeit oder Kombinationen davon.

Algorithmus von Dijkstra

Die Idee ist erstmal folgende: Wir benutzen eine Menge S von Knoten, für die die kürzesten Wege bereits gefunden sind. Zudem gibt es noch die Menge Q, in der die anderen Knoten enthalte sind. Der Wert $D(v)$ repräsentiert die Länge des bisher kürzesten Weges von s nach v.

```
Dijkstra(G,s) {
 D(s) = 0
 S = leer
 Q = V
 forall (v in V) {
   D(v) = unendlich
 }
 while (Q != leer) {
   v := Element aus Q mit kleinstem D(v)
   nimm v in S auf und entferne es aus Q
   forall (u adjazent zu v) {
     D(u) := min (D(u), D(v)+c(v,u))
     (falls D(u) verändert wurde, merke vorgänger(u) := v)
   }
 }
}
```

Aus den Vorgängern kann man dann einen Baum der kürzesten Wege extrahieren.

Korrektheit Wir wollen zeigen, dass der Algorithmus tatsächlich auch die kürzesten Wege berechnet.
Wir behaupten, dass nach jeder Iteration der While-Schleife gilt: Zu jedem Knoten $v \in S$ ist $D(v) = d(v)$, wobei $d(v)$ die Länge des kürzesten Weges von s nach v ist.
Wir führen jetzt einen Induktionsbeweis über die Anzahl i der Iterationen.

Induktionsanfang $i = 1$. Das bedeutet, $S = \{s\}$ und $D(s) = 0$ damit sind wir fertig.

Induktionsschritt $i \to i + 1$. Sei v der Knoten, der bei der $(i + 1)$ten Iteration nach S kommt. Angenommen $D(v) \neq d(v)$, also $D(v) > d(v)$. $D(v)$ ist immer die Länge eines Weges. Wir betrachten den kürzesten Weg π von s nach v. Sei S_i die Menge S nach i

Iterationen. Man muss also, da v noch außerhalb von S_i liegt, auf dem kürzesten Weg irgendwann zum ersten mal S_i verlassen. Sei (u, v) die Kante, bei der dies geschieht. Sei π_1 das Anfangsstück von π, nämlich von s nach u. π_2 sei nun das Stück von s nach w. Dabei ist die Länge von π_1 die Länge des kürzesten Weges, also $d(u)$, analog ist die Länge von $\pi_2 = d(w)$.
Als u in S eingefügt wurde, wurde $D(w)$ auf das Minimum von diesem und einen anderen Wert gesetzt: $D(w) \leq D(u) + c(u, w) = d(u) + c(u, w) = d(w)$. Also gilt $D(w) = d(w)$, wobei auch $d(w) \leq d(v) < D(v)$.
Dies ist ein Widerspruch, da v und nicht w im dritten Schritt ausgewählt wurde.

Laufzeit Man nimmt erstmal für Q eine Prioritätswarteschlange (z.B. einen Heap). Dann geht das Holen des kleinsten Elementes in $O(\log n)$. Durch das Ändern des $D(v)$ Wertes in der for-Schleife, benötigen wir auch $O(\log n)$ Zeit um den Heap zu aktualisieren. Die Initialisierung benötigt dann noch $O(n)$.
Insgesamt ergibt sich dann mit den Schleifen $O(n + n \log n + m \log n) = O((n + m) \log n)$.

27. Vorlesung, 29.1.2008

Floyd-Warshall-Algorithmus

Gegeben ist ein gerichteter, kantengewichteter Graph $G = (V, E, w)$ mit $w : E \to \mathbb{R}$. Wir nehmen dazu noch an, dass es keine Zyklen mit negativem Gesamtgewicht gibt.
Wir wollen jetzt für alle Paare $u, v \in V$ die Länge eines kürzesten Weges von u nach v (sonst $+\infty$) und einen realisierenden Weg suchen.
Zur Lösung können wir $|V|$ mal den Dijkstra ausführen. Es geht aber noch besser, sogar dann ohne Prioritätswarteschlange.
Dafür verwenden wir als Datenstruktur eine Adjazenzmatrix W, mit

$$w_{i,j} = \begin{cases} 0 & i = j \\ \text{Gewicht } i \to j & (i,j) \in E \\ \infty & \text{sonst} \end{cases}$$

Ein Beispiel:

	1	2	3	4	5	6
1	0	7	∞	∞	∞	4
2	∞	0	∞	∞	3	∞
3	-6	-1	0	∞	∞	∞
4	1	∞	∞	0	∞	2
5	∞	∞	∞	∞	0	∞
6	10	∞	∞	∞	∞	0

Man kann beobachten, dass kürzeste Wege kürzeste Wege enthalten. Und wenn ein Knoten x auf einem kürzesten Weg von u nach v auftritt, dann nur einmal.
Wir merken uns auch wieder den Vorgänger:

$$\pi_{i,j} = \begin{cases} i & (i,j) \in E, \\ nil & \text{sonst} \end{cases}$$

Sei $d_{i,j}$ die Länge eines kürzesten Wege von i nach j. Und sei $d^k_{i,j}$ die Länge eines kürzesten Weges von i nach j, wobei alle Zwischenknoten aus Kantenmenge $\{1, 2, \ldots, k\}$ stammen. Für $k = 0$ ist $d^0_{i,j} = w_{i,j}$ der Eintrag der Matrix. $d^n_{i,j} = d_{i,j}$.
Wir berechnen die $d^k_{i,j}$ bottom-up für wachsendes k. Sei D^k die daraus entstehende Matrix. Jetzt stellen wir die Rekursionsgleichung auf. Dabei betrachten wir zwei Fälle. einmal tritt

k nicht als Zwischenknoten auf, d.h. alle Zwischenknoten stammen aus $\{1, \ldots, k-1\}$ Das entspricht ja $d_{i,j}^{k-1}$. Im anderen Fall tritt k auf. Und laut unserer Beobachtung nur ein einziges mal. Hier ergibt sich dann $d_{i,k}^{k-1} + d_{k,j}^{k-1}$. Insgesamt ergibt sich

$$d_{i,j}^{k} = \min\{d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}\}$$

Das ist auch schon unser Algorithmus

```
for k=1 to n {
  for i=1 to n {
    for j=1 to n {
      d_(i,j)^k := ...
    }
  }
}
```

Die Laufzeit ist hiervon dann $O(n^3)$.
Wir berechnen parallel die realisierenden Wege. Dazu ist $\pi_{i,j}^{k}$ der Vorgänger von i auf kürzest Weg von i nach j nach Zwischenknoten $\{1, \ldots, k\}$.

$$\pi_{i,j}^{k} = \begin{cases} \pi_{i,j}^{k-1} & d_{i,j}^{k-1} < d_{i,k}^{k-1} + d_{k,j}^{k-1} \\ \pi_{k,j}^{k-1} & \text{sonst} \end{cases}$$

27.0.7. Transitive Hülle

Gegeben ist ein gerichteter Graph $G = (V, E)$ zu dem der Graph $G^* = (V, E^*)$ gesucht ist, der die transitive Hülle bildet. Dabei ist $(i, j) \in E^*$ falls es in G einen gerichteten Weg von i nach j gibt.
Das könnte man z.B. so machen, dass man den Floyd-Warshall-Algorithmus für Kantengewichte gleich 1 benutzt. Und wenn dann eine Kante ungleich unendlich ist, dann ist die Kante in G^*. Es geht auch eleganter. Hierbei verwalten wir boolesche Variablen $t_{i,j}^{k}$, wobei

$$t_{i,j}^{k} = \begin{cases} 1 & \text{es gibt Weg von } i \text{ nach } j \\ 0 & \text{sonst} \end{cases}$$

Dann ist

$$T^0 = \begin{cases} 1 & (i,j) \in E \vee i = j \\ 0 & \text{sonst} \end{cases}$$
$$T^k = (t_{i,j}^{k})$$
$$t_{i,j}^{k} = t_{i,j}^{k-1} \vee (t_{i,k}^{k-1} \wedge t_{k,j}^{k-1})$$

Die Laufzeit ist hier allerdings auch noch $O(n^3)$, aber boolesche Werte sind natürlich viel besser.

27.0.8. Längster Weg im gerichteten azyklischen Graphen

Sei w die Gewichtsfunktion auf Kanten. Im ersten Schritt sortieren wir das ganze topologisch. Dann bennenen wir die Knoten um in $\{1, \ldots, n\}$ entsprechend der Position in der topologischen Sortierung. Für Knoten $(i, j) \in E$ gilt $i < j$.
Die Idee ist jetzt für $i = 1$ bis n schaue alle Vorgängerknoten an, die Kanten haben, die in i enden. Sei l_i die Länge des längsten Weges, der in i endet. Sei $\pi(i) = \{j | (i, j) \in E\}$

$$l_i = \max_{j \in \pi(i)} \{l_j + w(j - i), 0\}$$

$l_{opt} = \max_{1 \leq i \leq n} l_i$

28. Vorlesung, 31.1.2008

28.0.9. Kürzeste Spannbäume

Im Englischen wird das Minimum-Spanning-Tree (MST) genannt. Wir werden dieses Problem dann später mit dem Algorithmus von Prim lösen.
Gegeben ist ein ungerichteter zusammenhängender Graph $G = (V, E)$ mit Kantengewichten $c : E \to \mathbb{R}$. Gesucht ist jetzt ein Spannbaum $T \subseteq E$ mit möglichst geringen Kosten $c(T) = \sum_{e \in T} c(e)$.
Damit wir uns alle Beweise etc. vereinfachen können, nehmen wir an, dass alle Kanten verschiedene Längen haben.

Lemma $V = S \cup Q$ und $S \cap Q = \emptyset$. Die Kante $(u, v) \in E$ sei die kürzeste Kante mit $u \in S$ und $v \in Q$. Dann gehört (u, v) zu jedem kürzesten Spannbaum.
Das Beweisen wir jetzt kurz. Sei T ein Spannbaum, der (u, v) nicht enthält. W sei der Weg von u nach v in T. W geht von S nach Q und muss dabei mindestens eine Kante (u', v') mit $u' \in S$ und $v' \in Q$ enthalten. $T \backslash \{(u', v')\}$ zerfällt in zwei Komponenten, so dass u und v in verschiedenen Komponenten liegen. $T \backslash \{(u', v')\} \cup \{u, v\} = T'$ wobei der Graph jetzt wieder zusammenhängend ist und einen Baum bildet. $c(T') = c(T) - c((u', v')) + c((u, v))$. Nach der Voraussetzung ist $c((u, v)) < c((u', v'))$. Und somit ist $c(T') < c(T)$ und somit ist T nicht optimal. □

Algorithmus von Prim

Wähle einen beliebigen Startknoten s. S ist die Menge der Knoten, die schon mit s verbunden sind. In Q sind dann die übrigen Knoten enthalten. Für jedes $v \in Q$ merkt man sich die kürzeste Kante $D(v) = \min\{c((u, v)) | u \in S\}$.

```
S := s
Q := V \ s
forall (v in Q) {
  D(v) := c(s,v) oder D(v) = unendlich, wenn es keine solche Kante gibt
  Vorg(v) := s
}
while (Q != leer) {
  v := Element von Q mit kleinstem Wert D(v)
  verschiebe v von Q nach S
```

```
  forall (u adjazent zu v) {
    if (c((u,v)) < D(u)) {
      D(u) := c((u,v))
      Vorg(u) := v
    }
  }
}
```

Der Baum ergibt sich dann natürlich wieder aus den Vorgängern.

Korrektheit

Die Korrektheit ergibt sich aus dem Lemma. Blabla. Invarianten 1 und 2 werden nach jedem Schleifendurchlauf erhalten. Am Ende ist T ein Spannbaum und $Q = \emptyset, S = V$ und jede Kante von T gehört zum kürzesten Spannbaum. Demnach muss es der kürzeste Spannbaum sein.

Der Algorithmus ist dem von Dijkstra ziemlich ähnlich. Zur Implementierung benötigt man D und $Vorg$ als Felder. Q wäre wieder eine Prioritätswarteschlange.

Laufzeit

Man benötig natürlich wieder $O((n+m)\log n)$ da jeder Knoten behandelt wird und man sich jede Kante anschaut. Das $\log n$ kommt durch die Prioriätswarteschlange zustande, genauer durch das Finden des Minimums. Da man viel mehr Kanten hat, kann man auch $O(m \log n)$ sagen.

Bis jetzt wollten wir ja immer unterschiedliche Kosten pro Kante. Jetzt wollen wir die Kosten ganz beliebig wählen lassen, also auch Gleichheit zulassen. Sei ϵ die kleinste Differenz zwischen verschiedenen Kantenkosten. Bei ganzen Zahlen könnte man z.B. $\epsilon = 1$ wählen. $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$ sei eine Anordnung der Kanten und zwar kommen zuerst die Kanten aus T, in der Reihenfolge, wie sie im Algorithmus ausgewählt werden. Und dann kommen die übrigen Kanten in beliebiger Reihenfolge. Jetzt kann man neue Kosten $c'(u_i, v_i)$ definieren, also $c(u_i, v_i) + \epsilon\frac{i}{2m}$. c' hat jetzt folgende Eigenschaften: Erstens sind alle Kosten nun verschieden, und wenn sie vorher verschieden waren, dann haben sie nicht die Reihenfolge vertauscht, also $c(u, v) > c(u', v') \Rightarrow c'(u, v) > c'(u', v')$.
Auf c' trifft dann wieder unsere Annahme der verschiedenen Gewichte zu. Mit c' macht der Algorithmus genau die gleichen Entscheidungen wie mit c. Der Algorithmus bestimmt den kürzesten Spannbaum mit Kosten c'.

29. Vorlesung, 5.2.2008

29.1. Schwere Probleme

Es gibt in der Praxis auftretende Probleme, für die kein effizienter Algorithmus bekannt ist.

Beispiele

- Gegeben ist ein Graph $G = (V, E)$. Die Frage ist, ob es einen Kreis in dem Graphen gibt, der jeden Knoten enthält.
  Das ist effizient lösbar, da bei einem ungerichteten Graphen die Aussage, das der Graph einen Kreis bildet, der alle Knoten enthält äquivalent damit ist, dass der Graph zusammenhängend ist. Man kann also z.B. per Breitensuche effizient auf die Lösung kommen.

- Gegeben ist ein Graph $G = (V, E)$. Jetzt ist die Frage, ob es einen Kreis gibt, der jeden Knoten genau einmal enthält. Das nennt man auch einen Hammeltonschen Kreis. Dieses Problem ist übrigens bis heute noch nicht effizient lösbar. Eine Lösungsmethode wäre, alle Permutationen von Knoten zu produzieren und für jede dieser Permutationen wird dann geprüft, ob es sich um einen Kreis handelt, wenn man die Knoten in der Reihenfolge der Permutation durchläuft. Das hat dummerweise eine Laufzeit von $O(n!)$, da es so viele Permutationen gibt.

- Gegeben ist eine Folge von natürlichen Zahlen $a_1, \ldots, a_n$ und eine natürliche Zahl b. Es ist die Frage, ob es eine Teilmenge von $a_1, \ldots, a_n$ gibt, deren Elemente sich zu b addieren. Bruteforce kann man das so machen, dass man alle 2^n Teilmengen durchprobiert, hätte dementsprechend eine exponentielle Laufzeit. Das nennt man das Knapsack-Problem (oder auch Rucksackproblem). Das ist bis heute auch noch nicht effizienter zu lösen.

- Gegeben ist ein Graph $G = (V, E)$ und Knoten $u, v \in V$. Wir wollen jetzt hier den kürzesten Weg von u nach v finden. Effizient geht das bekanntlich mit der Breitensuche.

- Gegeben ist ein Graph $G = (V, E)$ und Knoten $u, v \in V$. Wir wollen jetzt den längsten einfachen Weg von u nach v finden. Das ist dann auch ein schwieriges Problem.

- Gegeben ist eine boolesche Formel ϕ und wir wollen wissen, ob sie erfüllbar ist, d.h. existiert eine Wahrheitswert-Belegung, die die Formel wahr macht. Das ist auch ein

schweres Problem. Bruteforce-mäßig würde man alle Belegungen durchprobieren, das hätte ein Laufzeit von $\Omega(2^n)$. Das nennt man auch Erfüllbarkeitsproblem.

- Gegeben ist ein vollständiger gerichteter Graph $G = (V, E)$ und eine Kostenfunktion $c : E \to \mathbb{R}$. Wir wollen jetzt eine Rundreise minimaler Kosten finden, d.h. einen Kreis, der jeden Knoten einmal besucht. Das ist das so genannte "traveling salesperson problem" (früher auch einfach nur "traveling salesman problem" aber wir sind ja heute so emanzipiert, dass man das umbenennen muss). Das würde natürlich wieder eine Laufzeit von $\Omega(n!)$ brauchen, da man erstmal alle Reihenfolgen generieren müsste.

- Eine Zahl N sei in Binärdarstellung gegeben und n Bits lang. Wir wollen wissen, ob es eine Primzahl ist. Ein möglicher Algorithmus wäre, dass man bis $\lfloor\sqrt{N}\rfloor$ alle Zahlen durchgeht, und probiert N durch die entsprechenden Zahlen zu teilen. Das ist also $\Omega(\sqrt{N}) = \Omega(2^{\frac{n}{2}})$ was exponentiell ist.
  Mittlerweile gibt es aber auch schon einen Algorithmus, der in polynomieller Laufzeit das Problem löst. Dieser wurde 2002 gefunden.

- Gegeben seien zwei Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ und wir wollen wissen, ob die beiden Graphen isomorph sind. D.h. ob eine Funktion existiert, so dass $f : V_1 \to V_2$ mit $\{u, v\} \in E_1$ genau dann wenn $\{f(u); f(v)\} \in E_2$. Bei der Bruteforcelösung müsste man wieder alle Bijektionen durchprobieren, dass wäre dann wieder $\Omega(n!)$.

Einige dieser Probleme erwarten eine Ja oder Nein Beantwortung, daher nennt man sie auch Entscheidungsprobleme. Diese Probleme entsprechen quasi formalen Sprachen.
Berechnungsprobleme sind Probleme, bei denen eine kompliziertere Funktion als Antwort berechnet werden soll. Oft sind dies auch Optimierungsprobleme, bei denen Größen minimerit bzw. maximiert werden sollen. Man kann die übrigens in Entscheidungsprobleme umwandeln.

29.1.1. Definitionen

Das Berechnungsmodell, welches wir verwenden werden, sind die Turingmaschinen. Die Laufzeit einer Turingmaschine M bei Eingabe w soll die Anzahl der Schritte von M bei der Eingabe sein, bis sie hält.
Für eine Turingmaschine M ist $T(n)$, wobei $T : \mathbb{N} \to \mathbb{N}$ die maximale Laufzeit bei einer Eingabe der Länge n.
P ist die Klasse der formalen Sprachen, die von einer deterministischen Turingmaschine polynomieller Laufzeit erkannt werden. $L \in P$ genau dann wenn $\exists k \in \mathbb{N}$ mit deterministischer Turingmaschine M der Laufzeit $O(n^6)$ mit $L(M) = L$.
NP ist die Klasse der formalen Sprachen, die von einer nichtdeterministischen Turingmaschine in polynomieller Laufzeit akzeptiert werden.

30. Vorlesung, 7.2.2008

Man kann eine Turingmaschine durch eine Registermaschine mit logarithmischem Kostenmaß ersetzen, deshalb ist alles, was in polynomieller Laufzeit auf der Turingmaschine ausführbar ist, in selber Laufzeit auf einer Registermaschine funktioniert.
Das Erfüllbarkeitsproblem ist Element von NP, klar ist, dass $P \subseteq NP$ gilt, denn deterministische Turingmaschinen sind ein Spezialfall nichtdeterministischer Turingmaschinen.
Die große Frage ist, ob P eine echte Teilmenge von NP ist oder ob sie gleich sind. Bis heute kann das niemand sagen.

30.0.2. SAT und 3SAT

Definition 6. *Seien $L_1, L_2 \subset \Sigma^*$ Sprachen (gleichbedeutend mit Entscheidungsproblemen). Man sagt L_1 ist polynomieller Zeit reduzierbar auf L_2, man schreibt das $L_1 \leq_P L_2$, genau dann wenn es eine in polynomieller Zeit berechenbare Funktion $f : \Sigma^* \to \Sigma^*$ gibt, mit $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

Definition 7. *Eine Sprache $L \subset \Sigma^*$ heißt*

- *NP-schwer, genau dann wenn $L' \leq_P L$ für jedes $L' \in NP$*
- *NP-vollständig, falls sie NP-schwer ist und selbst in NP liegt.*

Folgerungen

- $\leq_P$ ist transitiv. D.h., falls $L_1 \leq_P L_2$ und $L_2 \leq_P L_3$, dann ist $L_1 \leq_P L_3$.
- $L_1 \leq_P L_2$ und $L_2 \in P$ so folgt dass auch $L_1 \in P$ für alle $L_1, L_2 \in \Sigma^*$
- $L \subset \Sigma^*$ gilt folgendes: Wenn L NP-schwer ist und $L \in P$, dann folgt NP=P.
- $L_1, L_2 \in \Sigma^*$. Wenn L_1 NP-schwer ist und $L_1 \leq_P L_2$, dann folgt, dass L_2 NP-schwer ist.

Satz 4. *Das Erfüllbarkeitsproblem (SAT) ist NP vollständig.*

Wir machen hier nicht den Beweis. Hierfür wäre aber zu zeigen, dass 1. $SAP \in NP$ ist und 2. dass $L \leq_P SAT \ \ \forall L \in NP$.
SAT ist sogar NP-vollständig, wenn man sich auf Formeln in konjunktiver Normalform beschränkt.

Als 3SAT bezeichnet man die Erfüllbarkeit von Booleschen Formeln in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel.

Man kann NP auch anders definieren: Und zwar sind das in polynomieller Zeit verifizierbare Probleme. Wenn man also bei dem Rucksackproblem die Lösung schon kennt, dann kann man ganz schnell und effizient diese Nachprüfen.

Satz 5. *3SAT ist in NP-vollständig*

Beweis 5.

1. *3SAT $\subset$ NP. Gegeben Formel ϕ die erfüllbar ist. Dann existiert ein Zeuge, nämlich die Belegung der Variablen, die ϕ erfüllt. Das kann in polynomieller Zeit verifiziert werden.*

2. *3SAT ist NP-vollständig. Wird gezeigt durch SAT $\leq_p$ 3SAT. Dann ist nämlich 3SAT NP-schwer nach Folgerung 4.*
   Sei $\kappa = X_1 \vee X_2 \vee \ldots \vee X_n$ eine Klausel für SAT. Wir nehmen jetzt $\kappa' = (X_1 \vee y_1) \wedge (\overline{y_1} \vee X_2 \vee y_2) \wedge \ldots \wedge (\overline{y_{n-2}} \vee X_{n-1} \vee y_{n-1}) \wedge (\overline{y_n} \vee X_n)$ für 3SAT.
   Es gilt, eine Belegung erfüllt κ', daraus folgt, sie setzt mindestens ein Literal $X_1, \ldots, X_n$ auf wahr. Das heißt, sie erfüllt auch κ.
   Eine Belegung, die κ erfüllt, lässt sich fortsetzen auf eine, die κ' erfüllt. Dann kann man induktiv zeigen, dass die $y_1, y_2, \ldots y_{n-1}$ auf 1 gesetzt werden müssen und damit ist die letzte Klausel $\overline{y_{n-1}} \vee X_n$ nicht erfüllt ist, und das ist ein Widerspruch.
   Sei X_i auf 1 gesetzt. Dann setze $y_1, \ldots, y_{i-1}$ auf 1 und $y_i, \ldots y_{n-1}$ auf 0. Das erfüllt dann κ'. Für alle Klauseln wird das durchgeführt, dass führt dann zu einer Formel ϕ' mit gewünschten Eigenschaften.

31. Vorlesung, 12.2.2008

31.0.3. Cliquen-Problem

Gegeben sei ein Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$. Eine Clique ist ein vollständiger Teilgraph, bei dem jeder Knoten mit jedem anderen des Teilgraphen verbunden ist.
Jetzt ist die Frage, ob in G eine Clique der Größe k existiert.

Satz 6. *Die Clique ist NP-vollständig.*

Beweis 6. *Wir zeigen dazu, dass das Problem in NP liegt. Dazu kann man nicht deterministisch eine Teilmenge der Größe k auswählen und in polynomieller Zeit nachprüfen, ob es eine Clique ist.*
Als zweites muss noch gezeigt werden, dass das Problem NP-schwer ist. Wir zeigen dazu, dass $3SAT \leq_P CLIQUE$. Sei ϕ eine Boolesche Formel in 3KNF. und $\phi = (y_1 \vee y_2 \vee y_3) \wedge \ldots \wedge (y_{k1} \vee y_{k2} \vee y_{k3})$. Das wird übersetzt in einen Graphen G_ϕ, dieser hat pro vorkommendem Literal einen Knoten. Und wir verbinden Knoten mit Kanten, genau dann wenn sie nicht aus der gleichen Klausel stammen und sie keine Negation voneinander sind.
Die Behauptung ist jetzt, dass ϕ ist erfüllbar, genau dann wenn G_ϕ eine Clique der Größe k hat, wobei k die Anzahl der Klauseln ist. Diese Behauptung muss natürlich auch beweisen werden. Dazu wird Hin- und Rückrichtung gezeigt.

$\Rightarrow$ *ϕ ist erfüllbar, d.h. es existiert eine erfüllende Belegung, die in jeder Klausel mindestens ein Literal auf 1 setzt.*

$\Leftarrow$ *G_ϕ hat eine Clique der Größe k. Daraus folgt, aus jeder Klausel muss einzugehöriger Literalknoten beteiligt sein (da es keine Kanten innerhalb einer Klausel gibt).*
Wir können alle diese Literale auf 1 setzen, da sie einander nicht widersprechen. Damit haben wir pro Klausel eines auf 1 gesetzt, damit ist also ϕ erfüllbar. Wähle aus jeder Klausel ein solches Literal. Die entsprechenden Knoten bilden eine Clique der Größe k, denn je zwei sind aus verschiedenen Klauseln und widersprechen sich nicht.

□

Aus dem Cliquenproblem leitet sich gleich das unabhängige Knotenmengen-Problem ab.

31.0.4. Unabhängige Knotenmenge

Gegeben sie ein Graph $G = (V, E)$ und ein $k \in \mathbb{N}$. Die Frage ist, ob es eine unabhängige Knotenmenge der Größe k gibt, d.h. alle paarweise nicht durch eine Kante verbundenen.

Satz 7. *Auch dieses unabhängige Knotenproblem ist NP-vollständig.*

Beweis 7. *Das es in NP liegt ist klar. Und jetzt müssen wir noch zeigen, dass es NP-schwer ist. Also* $CLIQUE \leq_P UK$.
G *hat eine Clique der Größe* k *genau dann wenn* $\overline{G}$ *hat unabhängige Knotenmenge der Größe* k. *Und somit ist es also auch NP-vollständig.*

31.0.5. Überdeckende Knotenmenge

Gegeben ist wieder ein Graph $G = (V, E)$ und $k \in \mathbb{N}$. Die Frage ist, ob eine überdeckende Knotenmenge der Größe k existiert, d.h. das eine Teilmenge von Knoten, so dass jede Kante mindestens einmal berührt wird.

Satz 8. *Das Problem der überdeckenden Knotenmenge ist NP-vollständig.*

Beweis 8. *Das es in NP ist, ist wieder klar. Wir müssen uns nun noch darauf konzentrieren, zu zeigen, dass es NP-schwer ist. Dazu versuchen wir* $CLIQUE \leq_P UEK$ *zu zeigen.* G *hat Clique der Größe* k *genau dann wenn* $\overline{G}$ *hat ÜK der Größe* $n - k$, *mit* $n = |V|$. *Wenn* G *hat Clique* $V' \subseteq V$ *der Größe* k. *Dann ist* V' *? in* $\overline{G}$. *Das ist genau dann der Fall, wenn alle Kanten in* $\overline{G}$ *haben mindestens 1 Endpunkt in* $V \backslash V'$ *und das ist genau dann wenn* $V \backslash V'$ *ÜK.*

Bemerkungen Die Berechnungsprobleme wie finde eine Clique maximaler Größe, Finden einer unabhängigen Knotenmenge oder das Finden einer minimalen überdeckenden Knotenmenge sind Optimierungsprobleme und sind mindestens so schwer wie die entsprechenden Entscheidungsprobleme. Man nennt sie auch NP-schwer.

31.0.6. Knapsack-Problem

Gegeben sei eine Menge $A = \{a_1, \ldots, a_n\} \subseteq \mathbb{N}$ und ein $b \in \mathbb{N}$. Jetzt ist die Frage $\exists A' \subset A$ mit $\sum A' = b$?

Satz 9. *Knapsack ist NP-vollständig.*

Beweis 9. *Für Knapsack* $\in NP$ *ist das wieder einfach. Das NP-schwer ist wieder schwieriger. Wir wollen wieder* $UEK \leq_P Knapsack$ *zeigen. Gegeben sei ein Graph* $G = (V, E)$ *und ein* $k \in \mathbb{N}$. *für das Knapsackproblem. Wir können für den Graphen dann eine Inzidenzmatrix aufstellen. Eine überdeckende Knotenmenge erhält man dann, wenn man die Knoten auswählt, so dass in jeder der Spalten der ausgewählten Knoten mindestens eine 1 steht. Bla bla. Und so kann man dann den überdeckenden Graph in entsprechene Zahlen übersetzen. Dann existiert eine Auswahl der Zeilen, das sich zu* b *addieren lässt, genau dann wenn G hat eine überdeckende Menge der Größe* k. □