Inhaltsverzeichnis

1 Einführung

Ziel der Vorlesung

- Kenntnis weit verbreiteter Datenstrukturen und Algorithmen
- Fähigkeit, eigene Programme zu analysieren

1.1 Literatur

- Robert Sedgewick, Algorithmen (in C++ / in Java)
- Ottmann / Widmayer, Algorithmen und Datenstrukturen
- Reß / Viebeck, Datenstrukturen und Algorithmen in C++

1.2 Algorithmus

Definition (ALGORITHMUS)

Ein Algorithmus ist ein (schematisches) Verfahren zur Lösung einer Problemstellung unter Verwendung tatsächlich ausführbarer Schritte. ■

Begriff geht zurück auf Al-Khowarizmi.

Algorithmen

- verarbeiten Eingabe und produzieren Ausgabe (Resultat).
- können informell oder in einer Programmiersprache beschrieben werden.

Beispiel: Euklids Algorithmus zur Berechnung des größten gemeinsamen Teilers $ggT(a,b)$ zweier ganzer Zahlen $a>0$ und $b>0$.

a) falls $a=b$, dann Ausgabe a.

b) falls $a<b$, dann
   - falls $b \bmod a=0$, dann Ausgabe a.
   - sonst berechne $ggT(a, b \bmod a)$

c) falls $a>b$, dann
   - falls $a \bmod b=0$, dann Ausgabe b
   - sonst berechne $ggT(a \bmod b, b)$

$ggT(24,16)$
$24 \bmod 16=8$
$\rightarrow ggT(16,8)$
$\rightarrow$ Ausgabe: 8

Fragestellungen:

- Terminierung: Bricht der Algorithmus stets ab?
- Korrektheit?
- Ressourcenbedarf? (Zeit, Speicher)

1.3 Komplexitätsbegriff

1.3.1 Definitionen

Definition:

Die Zeitkomplexität $T(n)$ eines Algorithmus gibt an, wie viel Zeit der Algorithmus bei einer gegebenen Problemgröße n benötigt. ■

Zeitkomplexität

- wird oft in der Zahl der benötigten charakteristischen Operationen ausgedrückt (z.B. Anzahl Vergleiche und Verschiebungen bei Sortierverfahren).
- ist eine positive, monoton wachsende Funktion.

Beispiel: Matrixmultiplikation

A, B, C seien $n \times n$-Matrizen, Berechnung von $C = A \cdot B$

```
for (i=0; i<n; i++)
     for (j=0; j<n; j++) {
         c[i][j]=0;
         for (k=0; k<n; k++)

             c[i][j] += a[i][k] * b[k][j];
     }
```

Drei geschachtelte Schleifen mit je n Durchläufen
$\Rightarrow T(n) = n^3$ Multiplikationen und Additionen
(Beschränkung auf Gleitpunktoperationen)

Definition: (Best Case, Average Case, Worst Case)

Die Zeitkomplexitäten für den besten, mittleren und schlechtesten Fall sind definiert als:

$$T_{best}(n) := \min_{i \in I} T_i(n)$$

$$T_{avg}(n) := \sum_{i \in I} T_i(n) p_i$$

$$T_{worst}(n) := \max_{i \in I} T_i(n)$$

Hier ist I als die Menge aller möglichen Eingaben, i ein konkreter Eingabefall, p_i ist die Wahrscheinlichkeit des Auftretens des Eingabefalls i. ■

Best Case meist irrelevant.

Beispiel: Suche eines Eintrags in einer Liste der Länge n durch einfaches Ablaufen und Vergleichen der Einträge

- Best Case (erstes Element ist das gesuchte): $T_{best}(n) = 1$
- Worst Case (letztes oder kein Element ist das gesuchte): $T_{worst}(n) = n$

- Average Case, gesuchtes Element mit Wahrscheinlichkeit 0,5 enthalten:
  $T_{avg}(n)=\frac{3}{4}n+\frac{1}{4}$

Ein Algorithmus heißt zeitoptimal, falls es keinen anderen Algorithmus gibt,

- der das Problem löst, und
- dessen Worst Case-Zeitkomplexität ab einer gewissen Problemgröße stets kleiner ist.

Die Speicherkomplexität $S(n)$ ist das auf den Speicherplatzbedarf bezogene Analogon der Zeitkomplexität.

Niedrige Zeitkomplexität geht oft einher mit hoher Speicherkomplexität (Speichern von Zwischenergebnissen) und umgekehrt.

Aufgabe 1.1

1. Buchstaben in Buchladen?
   Bücher: 10^4
   Seiten: $2\cdot10^2$
   Zeilen: $5\cdot10^1$
   Buchstaben: $8\cdot10^1$
   $8\cdot10^9$
2. Tonnen Essen in FH Ingolstadt?
   Studenten: $5\cdot10^2$
   Essen / Student: $5\cdot10^2 g$
   Tage: $1{,}5\cdot10^2$
   $3{,}8\cdot10^7 g \quad 38 t$
3. Geschwindigkeit Haarwachstum in km/h?
   $\underset{0{,}00000072}{W} \quad \underset{10^{-5}Km}{1cm/Monat} \quad \underset{30\cdot24=720}{}$
   $10^{-8}\frac{km}{h}$
4. Anzahl Silben, die Menschheit seit 1400 n. Chr. gesprochen hat?
   B: $5\cdot10^8$
   W / T: 10^4
   T: $3{,}65\cdot10^2\cdot6\cdot10^2=2\cdot10^5$
   10^{18}

1.3.2 O-Notation

Präzise Anzahl benötigter Operationen ist

- schwierig zu ermitteln.
- nicht interessant.

Statt dessen: Abschätzung der asymptotischen Komplexität

Bei Verdopplung der Problemgröße, wie viel Rechenzeit brauche ich?

- Genau so viel?
- Doppelt so viel?
- Viermal so viel?

Formalisierung über O-Notation.

Definition: (O-Notation nach Landau)

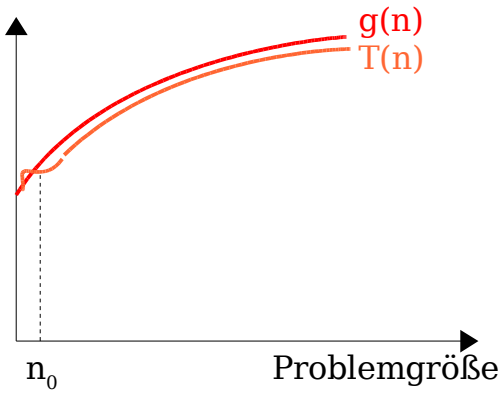
Die O-Notation vergleich die Komplexität $T(n)$ eines Algorithmus mit einer Funktion $g(n)$. Man schreibt

$$T(n)=O(g(n))$$

falls für alle hinreichend großen $n>n_0$ gilt:

$$T(n)\leq c_0 g(n), c_0\in\mathbb{R}^+ \text{ beliebig}.$$ ■

Visualisierung: $T(n)=O(g(n))$



Achtung: O-Notation ist kein Gleichheitszeichen im klassischen Sinn, weder symmetrisch noch transitiv.

O-Notation beschreibt obere Schranke.

Beispiele:

1. Multiplikation von $n\times n$-Matrizen: $T(n)=O(n^3)$
   Kurz: „Matrixmultiplikation ist $O(n^3)$“
2. Suche eines Eintrags in Liste der Länge n ist im Worst und Average Case $O(n)$.

Grundlegende Eigenschaften:

- Eliminationsregel (ER): $O(cf(n))=O(f(n))$
  „Vernachlässigung von Konstanten“
- Additionsregel (AR): $O(f(n)+g(n))=O(\max\{f(n),g(n)\})$
  „Der teuerste Teilschritt dominiert.“
- Multiplikationsregel (MR): $O(f(n)\cdot g(n))=O(f(n))\cdot O(g(n))$

Beispiel: Programmschleife ruft $f(n)=5n^2+3n+1$ mal Funktion auf, die $g(n)=n^3+n^2+22n+1$ viele Operationen benötigt.

Asymptotische Zeitkomplexität:

$$O(f(n)\cdot g(n)) \overset{MR}{=} O(f(n))\cdot O(g(n))$$
$$\overset{ER, AR}{=} O(n^2)\cdot O(n^3)$$
$$\overset{MR}{=} O(n^5)$$

Mit Komplexität ist im folgenden immer die asymptotische (Zeit-)Komplexität gemeint, wobei wir besten, mittleren und schlechtesten Fall unterscheiden.

1.3.3 Komplexitätsklassen

Einteilung von Algorithmen in Klassen:

- polynominale Komplexitäten $O(n^k), k\in\mathbb{N}$
  - $k=0$: konstante Komplexität $O(1)$
  - $k=1$: lineare Komplexität $O(n)$
  - $k=2$: quadratische Komplexität
  - $k=3$: kubische Komplexität
- logarithmische Komplexitäten $O(\log n)$
- $O(n\log n)$-Komplexität
- NP-Komplexität $O(a^n), a>1$

Interpretation der Komplexitätsklassen:

$$O(1)\ \ O(\log n)\ \ O(n)\ \ O(n\log n)\ \ O(n^2)\ \ O(n^3)\ \ O(a^n)$$
$$\xrightarrow[\text{langsamer}]{}$$

Aufgabe 1.2

Komplexitätsklassen (aufsteigend geordnet):

1. Konstant $O(1)$
2. Logarithmisch $O(\log_3 n)=O(\log n)$

   [Es ist $\log_b n=\frac{\log_a n}{\log_a b}$

   und damit $O(\log_3 n)=O(const*\log n)=O(\log n).$]
3. Linear $O(n)$
4. $O(n\log n)$
5. quadratisch $O(3n^2+9n)=O(n^2)$

6. $O(n^2 \log n)$
7. kubisch $O(2n^3)=O(n^3)$
8. NP $O(2^n)$

Aufgabe 1.3

1. int y=0;
   for (int i=1; i<n; i++) ← wird n mal durchlaufen
   for (int k=1; k<n; k *= 2) ← wird log(n) mal durchlaufen
   y++; (k=1, 2, 4, 8, 16)
   $O(n \log n)$

2. int y=0;
   while (n>0) {
   n = n/2-1; ← n wird in jedem Durchlauf halbiert (ca.)
   y++;
   }
   $O(\log n)$

3. int y=0;
   for (int i=1; (i*i) <= n; i++) ← wird ca. $\sqrt{n}$ -mal durchlaufen
   for (int k=1; (k*k) <= n; k++) ← wird ca. $\sqrt{n}$ -mal durchlaufen
   y++;
   $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$

4. int y=0;
   for (int i=1; i<n; i++) ← wird n-mal durchlaufen
   for (int k=i; k>0; k--) ← wird O(n)-mal durchlaufen
   y++; (im Mittel ca. $\frac{n}{2}$ -mal)
   $O(n^2)$

Rechnergeneration 1 kann in akzeptabler Zeit Problemgröße N bewältigen. Rechnergeneration 2 ist zehnmal schneller. Wie groß sind jetzt die Probleme, die in akzeptabler Zeit bewältigt werden?

- Linear: $10N$
- $O(n \log n)$: $<10N$
- logarithmisch: N^{10}
- quadratisch: $\sqrt{10} \cdot N \approx 3N$
- $O(3^n)$: $N + \log_3 10 \approx N + 2$

1.4 Komplexität rekursiver Algorithmen

Analyse nicht-rekursiver Algorithmen meist einfach.

Bei rekursiven Algorithmen helfen Rekurrenzgleichungen.

1.4.1 Lineare Regression

Problemgröße reduziert sich bei rekursivem Aufruf um 1.

Satz: Gegeben sei eine rekursive Funktion, die sich für Problemgröße n selbst a-mal mit um 1 verringerter Problemgröße aufruft und außerhalb der rekursiven Aufrufe $bn+c$ Operationen benötigt.

1. $a=1, b=0: T(n)=O(n)$
2. $a=1, b>0: T(n)=O(n^2)$
3. $a>1: T(n)=O(a^n)$ ■

Illustration:

```
void f(int n) {
f(n-1);          (ohne Abbruchbedingung)
   ⋮             a-mal
f(n-1);
for (int j=0; j<b; j++) {
    for (int i=0; i<n; i++) {          bn + c Operationen
        ...
    }
}
```

Beispiel: Fakultät

$$fac(n)=\begin{cases}1, n=0\\ n\cdot fac(n-1) \text{ sonst}\end{cases}$$

Hier ist $a=1, b=0$ und damit $T(n)=O(n)$

Beispiel: Fibonacci-Zahlen

$$fib(n)=\begin{cases}1, n=0,1\\ fib(n-1)+fib(n-2), \text{sonst}\end{cases}$$

$a=2 \Rightarrow T(n)=O(2^n)$!!!

1.4.2 Geometrische Regression

Problemgröße nimmt beim rekursiven Aufruf geometrisch ab, z.B. statt eines Problems der Größe N zwei Probleme der Größe $\frac{N}{2}$.

(„Teile und Herrsche“, „divide and conquer“)

Satz

Gegeben sei eine rekursive Funktion, die sich für Problemgröße n selbst a-mal mit Problemgröße $\frac{n}{d}$ $(d>1)$ aufruft, und außerhalb der rekursiven Aufrufe $bn+c$ Operationen benötigt.
Hier gilt:

1. $b=0, \frac{a}{d} \leq 1 : T(n)=O(\log n)$
2. $b=0, \frac{a}{d} > 1 : T(n)=O(n^{\log_d a})$
3. $b>0, \frac{a}{d} < 1 : T(n)=O(n)$
4. $b>0, \frac{a}{d} = 1 : T(n)=O(n \log n)$
5. $b>0, \frac{a}{d} > 1 : T(n)=O(n^{\log_d a})$ ■

Beispiele:

1. Binäre Suche im sortierten Feld, $a=1, d=2, b=0 \Rightarrow T(n)=O(\log n)$
2. Jeder Funktionsaufruf erzeugt rekursiv 8 Aufrufe der halben Größe, $a=8, d=2 \Rightarrow T(n)=O(n^{\log_2 8})=O(n^3)$

1.5 Effizienzbetrachtungen

Algorithmus mit z.B. linearer Komplexität kann für gleiche Problemgröße auf einem System langsamer sein als ein anderer mit quadratischer Komplexität.

Algorithmeneffizienz hängt wesentlich, aber nicht nur, von Komplexitätsklasse ab:

- Bei kleinen Problemgrößen den Faktor c_0 aus der Definition der O-Notation beachten.

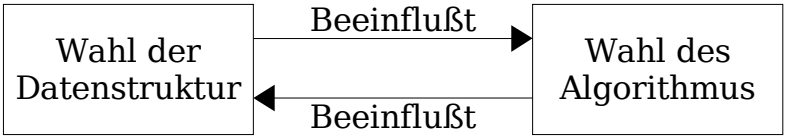
Weitere Gesichtspunkte:

- In Speicherhierarchie möglichst nur auf schnellem Speicher zugreifen.
  - Speicherzugriffe lokal halten, Page / Cache Misses vermeiden
  - Datenvolumen gering halten, wenig Platten-I/O
- Vektorrechner: für optimales Pipelining selten Grundoperationen wechseln, gut sind z.B. lange Additions- und Multiplikationsschleifen
- Parallelrechner: Kommunikation vermeiden, d.h. auf Unabhängigkeit der Daten voneinander achten.

1.6 Datenstrukturen

Datenstrukturen organisieren die Daten, so daß bestimmte Probleme leicht

lösbar sind.



Abstrakter Datentyp (ADT): Zusammenfassung von

- Datenstruktur (z.B. Liste) und dafür
- implementierten Methoden (z.B. Einfügen / Löschen / Suchen)

Aufgabe 1.4:

a)

```
iterative Variante:
public int fib (int n)
{
     int fibVonN = 1;
     int nMinus1 = 1;
     int nMinus2 = 1;
     if ((n != 0) && (n != 1)) {
         for (int i=2; i<=n; i++) {
             fibVonN = nMinus1 + nMinus2;
             nMinus2 = nMinus1;
             nMinus1 = fibVonN;
         }
     return fibVonN;
}
```

b) n Schleifendurchläufe mit konstanter Anzahl von Operationen:
$O(n)$

Aufgabe 1.8:

Kursdifferenzen in Feld a der Länge n. Mathematische Formulierung des Problems: Finde $\max_{i,j} \sum_{k=i}^{j} a_k \quad 0 \le i, j < n$

Vereinfachung: Wir berechnen nur Wert des Maximums, merken uns Intervallgrenzen nicht.

Algorithmus 1: Formel umsetzen

```
int maxsofar = 0;
for (int i=0; i<n; i++) {                          ← n Durchläufe
     for (int j=i; j<n; j++) {                     ← n/2 Durchläufe (im Mittel)
         int sum = 0;
         for (int k=i; k<=j; k++)                  ← n/3 Durchläufe (im Mittel)
             sum += a[k];                          (aus Analysis)
         maxsofar = max(maxsofar, sum);
     }
}                                                  O(n^3)
```

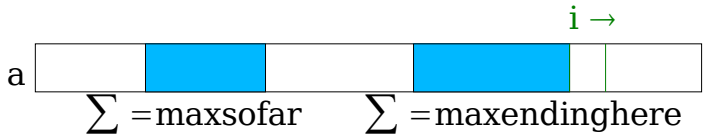
Algorithmus 2: Idee: Nutze $\sum_{k=i}^{j-1}$ zur Berechnung von $\sum_{k=i}^{j}$

```
int maxsofar = 0;
for (int i=0; i<n; i++) {                          ← n Durchläufe
     int sum = 0;
     for (int j=i; j<n; j++) {                     ← n/2 Durchläufe (im Mittel)
         sum += a[j];
         maxsofar = max(maxsofar, sum);
     }
}                                                  O(n^2)
```

Algorithmus 3: Scanning-Algorithmus

`maxsofar`: Bisheriges Maximum über alle Intervalle

`maxendinghere`: Kandidat für neues Maximum (wird auf 0 gesetzt, wenn es negativ wird)



i=Scanningpos

```
int maxsofar = 0;
int maxendinghere = 0;
for (int i=0; i<n; i++) {
     maxendinghere = max(maxendinghere + a[i], 0)
     maxsofar = max(maxsofar, maxendinghere);
}                                                  O(n)
```

2 Listen

Liste in der Informatik ≡ Begriff der endlichen Folge in Mathematik Menge von $n \in \mathbb{N}_0$ Objekten

$a_0, a_1, \ldots, a_{n-1}$

vom gleichen Grundtyp, von denen jedes außer dem ersten genau einen Vorgänger und jedes außer dem letzten genau einen Nachfolger hat.

2.1 Grundoperationen

- Einfügen
- Löschen
- Suchen
- Lesen / Schreiben

eines Eintrags.

In Bibliotheken weitere Methoden

- Anzahl Einträge berechnen
- Überprüfung, ob Liste leer
- ...

2.2 Anwendungen

- Vektoren
- Darstellung von Polynomen
- Liste von Teilnehmern einer Vorlesung
- ...

2.3 Implementierungen

2.3.1 Sequentielle Speicherung

```
int a[] = new int[n];
```

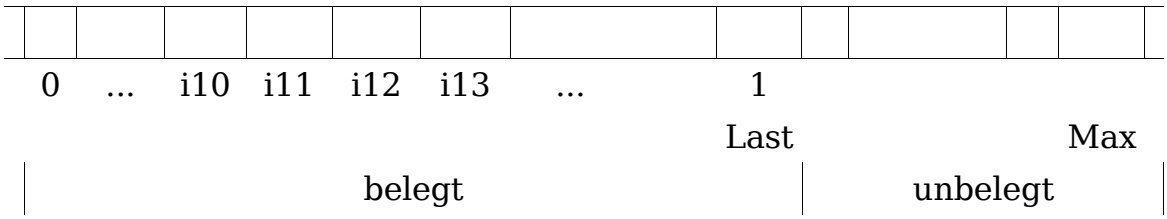
oder etwas komplizierter (als Klasse):

```
public class SequentialList {
// max. Anzahl von Einträgen in Liste
private int max;
//gegenwärtig letzter Eintrag
private int last;
private int list[];

public SequentialList(int max) {
     this.max = max;
     list = new int[max];
     last = -1;
}
... weitere Methoden
}
```

Sequentielle Listen

- können kompakt gespeichert werden.
- gewähren wahlfreien Zugriff.



Lesen / Schreiben des i-ten Elements:

Zugriff auf list[i]:

```
public int read (int i) {
     if (i<0 || i>last) {
         ...                 //Fehler
     }
     return list[i];
}
```

Schreiben etwas komplizierter.

Suchen eines Eintrags nach Wert:

```
pulbic int find (int key) {
     for (int i=0; i<=last; i++) {
         if (list[i]==key)
             return i;
     }
     return -1;              //nicht gefunden
}
```

Durchlaufen der Elemente von 0 bis last und Vergleich des Elementwerts mit dem Schlüssel. Index des ersten gefundenen Elements wird zurückgegeben.

Trick: Sentinel- oder Stopwert
Speichere gesuchten Wert an Position last+1;
Spart Indexvergleiche in der Schleife, da Suche garantiert erfolgreich ist.

```
public int find (key) {
     if (last<max-1)
         list[last+1]=key;
     else
         ...                 //Fehler
     i=0;
     while (list[i]!=key)
         i++;
     if (i==last+1)          //nicht gefunden
         return -1;
     return i;
}
```

Einfügen eines Eintrags: Umkopieren nötig: Im ungünstigsten Fall (Einfügen an

erster Position) alle Einträge 0 ... last.

Löschen eines Eintrags: Umkopieren auch hier. Beim Löschen des ersten Eintrags alle Einträge 1 ... last.

O-Komplexitäten für die sequentielle Liste (n=Listenlänge):

Einfügen	O(n)
Löschen	O(n)
Suche	O(n)
Lesen / Schreiben	O(1)

... im Avg- und Worst-Case

- Sequentielle Listen haben eher statischen als dynamischen Charakter
- Datensätze werden oft nicht direkt in der sequentiellen Liste gespeichert, sondern nur Zeiger / Referenzen darauf.

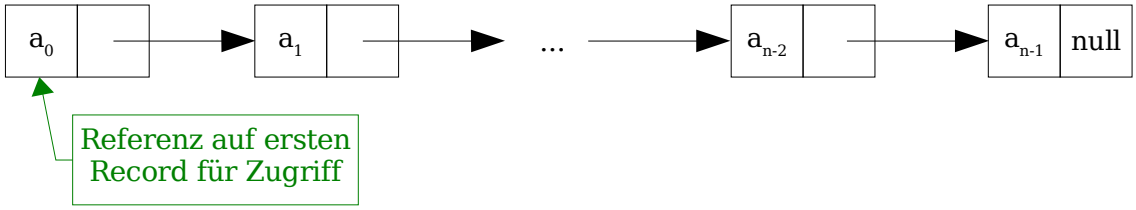
2.3.2 Einfach verkettete Liste

Bilden von Records mit je

- einem Listenelement (Nutzdaten)
- einer Referenz auf nächsten Record

Letzter Record enthält statt der Referenz null. Records können über den ganzen Speicher verstreut sein. Zugriff erfolgt über Listenkopf (Head).

Übliche Darstellungsweise:



in Java:

```
public class LinkedList {
     private class ListRecord {
         Object element;         //Nutzdaten
         ListRecord next;
     }

     private ListRecord head;    //Listenkopf
     private ListRecord tail;    //letzter Record
     ...                         //zusätzliche Methoden
}
```

Anmerkungen:

- Es gibt bereits Klasse java.util.LinkedList
  - In realem Projekt eigene Klasse anders benennen
  - In realem Projekt java.util.LinkedList verwenden
- Für schnelles Anfügen (append) wird oft Referenz auf letzten Record

gespeichert (tail).

Suchen eines Elements:

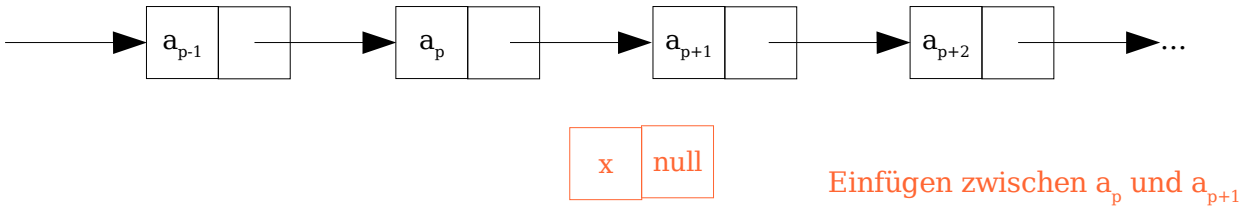
```
public int find (Object key) {
     int pos=0;
     ListRecord iter=head;
     while (iter!=null) {
         if (iter.element == key    //referentielle Gleichheit
         || iter.element.equals(key))  //wertmäßige Gleichheit
             return pos;
         iter=iter.next;
         pos++;
     }
     return -1       //nichts gefunden
}
```

- Auch hier läßt sich Sentinel-Trick anwenden, indem man zu dem letzten Record einen Record mit dem Schlüssel anfügt.
- Alternative Rückgabewerte: Referenz auf gefundenes Element
  Referenz auf listRecord, der das Element enthält

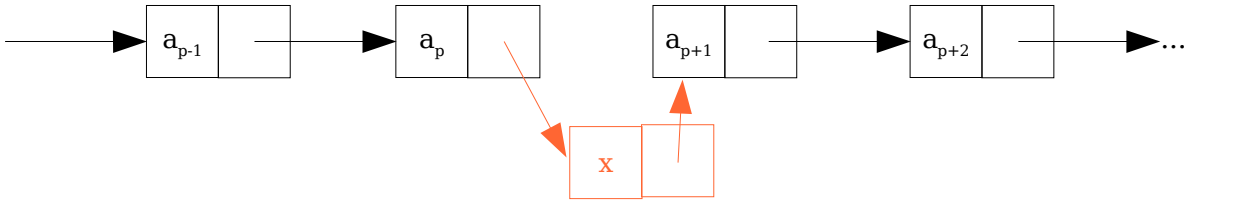
Einfügen eines Eintrags:

Objektreferenzen müssen umgesetzt werden:

Vorher:



Nachher:



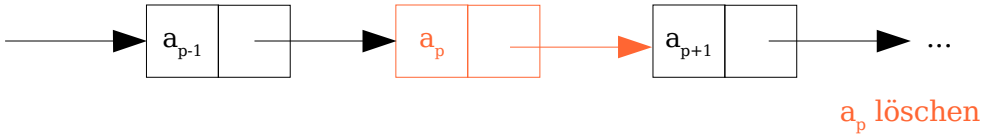
Möglichkeiten, die Einfügeposition zu spezifizieren:

- Index (ungünstig)
- Zeiger auf Vorgängerrecord (günstig)
- Zeiger auf Nachfolgerrecord (ungünstig)

Löschen eines Eintrags:

Auch hier Umsetzen von Objektreferenzen

Vorher:



Nachher:

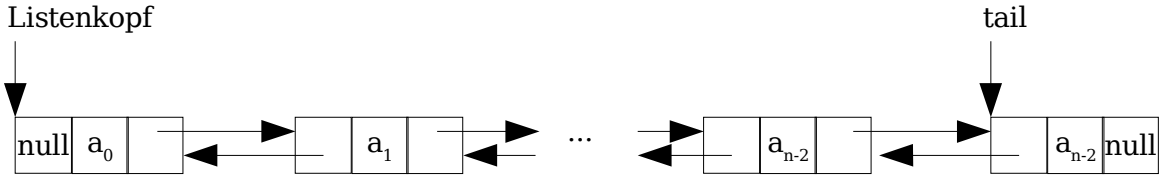
Möglichkeiten, Löschposition zu spezifizieren:

- Index (ungünstig)
- direkte Adressierung: Zeiger auf das zu löschende Element (ungünstig, Vorgängerrecord muß beginnend vom Listenkopf gesucht werden)
- indirekte Adressierung: Zeiger auf Vorgängerrecord (günstig, Löschen des ersten Records zu spezifizieren)

O-Komplexitäten für einfach verkettete Liste:

	Index	Zeiger auf Vorgängerrecord
Einfügen	$O(n)$	$O(1)$
Löschen	$O(n)$	$O(1)$
Lesen / Schreiben	$O(n)$	$O(1)$

2.3.3 Doppelt verkettete Liste



- Wegen Rückwärtsreferenz: Löschen läßt sich effizienter implementieren
- Rückwärtsdurchlauf von tail aus möglich
- Speicheraufwändiger

O-Komplexitäten wie bei einfach verketteter Liste

2.3.4 Geordnete Listen

Falls Ordnungsrelation $\leq$ für Listenelemente existiert und diese so angeordnet sind, daß

$$a_0 \leq a_1 \leq \ldots \leq a_{n-2} \leq a_{n-1}$$

gilt, so spricht man von einer geordneten Liste

Einfügeoperation muss Ordnungsrelation beachten.

Für sequentielle Listen läßt sich mit Hilfe der Ordnung eine binäre Suche

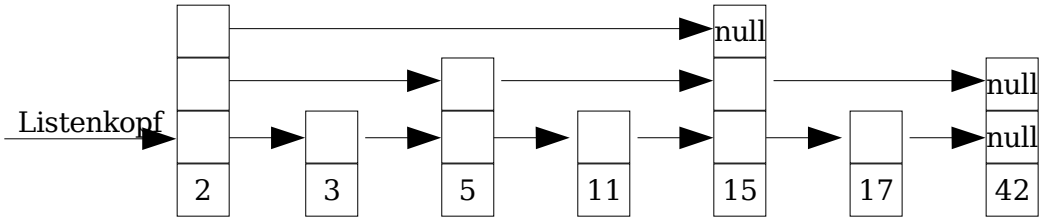
implementieren.

Rekurrenzgleichung für geometrische Regression: binäre Suche ist $O(\log n)$.

Für verkettete Listen bringt Ordnung leichte Vorteile bei der Suche (schnelleres Erkennen, daß Element nicht enthalten ist); binäre Suche nicht anwendbar.

Modifikation: Skip-Listen (Ausblick)

Speichere in jedem 2^k-ten Record Zeiger auf den 2^k Positionen entfernten Record.

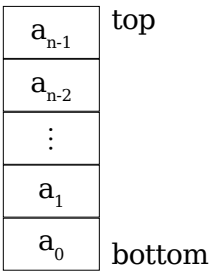


$$k = \lceil \log n \rceil$$

Erlauben logarithmischer Suchkomplexität $1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots=2 \mid \rightarrow$ Random Skip

3 Stapel

Keller, Stack



Neue Einträge werden auf die bereits existenten „gelegt“. Die oberen Einträge sind die jüngsten und werden gegebenenfalls als erste entfernt (LIFO – last in, first out).

3.1 Anwendungen

- Stack im Prozessadressraum: Parameter und Rückgabewerte von Funktionsaufrufen, lokale Variablen
- Umwandlung rekursiver in iterative Algorithmen
- Speichern von Operanden in Postfixnotation
- ...

3.2 Grundoperationen

- push: legt ein Element oben auf Stapel
- pop: nimmt ein Element von Stapel

3.3 Implementierung

Stapel werden meist als Listen mit eingeschränkter Funktionalität implementiert, z.B. mit einfach verketteter Liste

- push = Einfügen am Listenkopf
- pop = (Verbrauchendes) Lesen am Listenkopf

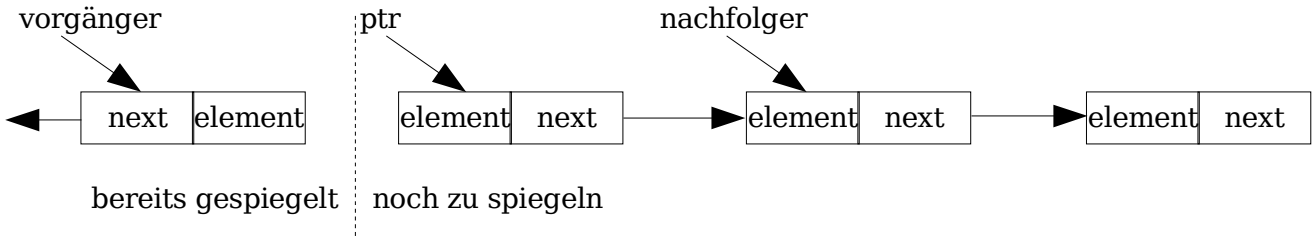
O-Komplexitäten:

push $O(1)$

pop $O(1)$

Aufgabe 2.1

Spiegeln der Liste: Umdrehen aller Zeiger unter Zuhilfenahme von drei Hilfszeigern




```
public void revert() {
     ListRecord vorgänger = null;
     ListRecord ptr = head;
     ListRecord nachfolger = null;

     if (ptr != null)
         nachfolger = ptr.next;
     while (ptr != null) {
         ptr.next = vorgänger;
         vorgänger = ptr;
         ptr = nachfolger;
         if (nachfolger != null)
             nachfolger = nachfolger.next;
     }
     head = vorgänger;   //Zuweisung gespiegelte Liste an LK
}
```

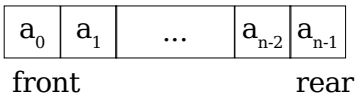
Aufgabe 2.3

```
iter.previous.next = iter.next;
iter.next.previous = iter.previous;
head.previous = iter;
iter.next = head;
head = iter;
head.previous = null;
```

zusätzliche Überprüfungen nötig:
- head != null
- iter 1. Element
- iter letztes Element

4 Schlangen

Warteschlangen, Queues



Im Gegensatz zum Stapel werden die jüngsten Einträge der Schlange zuletzt entfernt (FIFO – first in, first out).

Es wird an einem Ende angefügt, am anderen entfernt.

4.1 Anwendungen

- Simulation realer Warteschlangen
- Digital-Oszillatoren
- Drucker-Spooler
- FCFS- oder Round Robin-Scheduler im BS

4.2 Grundlegende Operationen

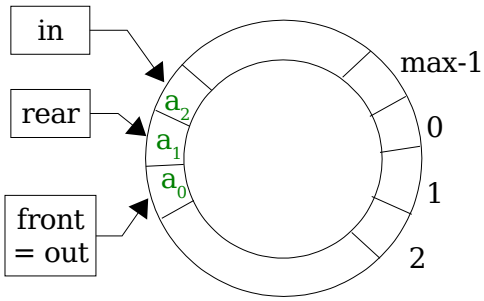
- enqueue fügt am Ende der Schlange einen Eintrag ein
- dequeue nimmt den ersten Eintrag aus der Schlange

4.3 Implementierungen

4.3.1 Über einfach verkettete Liste

Mit Referenzen auf head und tail einfach.

4.3.2 Über zirkuläres Array




```
public class Queue {
     private Object[] list;  //Array
     private int in;         //nächster Einfügepunkt
     private int out;        //nächster Entnahmepunkt
     private int max;        //maximale Länge
     private int n;          //Anzahl Einträge

     public void enqueue (Object entry) {
         if (n == max) {
             ... //Fehler
         }
         list[in] = entry;
         n++;
         in = (in + 1) % max;
     }

     public Object dequeue() {

     }
```

Bemerkung: Verallgemeinerung des Prinzips des Stapels (Einfügen, Verbrauchen an einer Seite) und des Prinzips der Schlange (Einfügen, Verbrauchen an entgegengesetzten Seiten) ist eine Deque (double-ended queue), wo man an beiden Enden sowohl einfügen als auch verbrauchen kann.

Mit doppelt verketteter Liste implementierbar.

5 Graphen

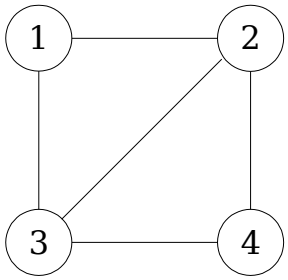
5.1 Grundlegende Definitionen

Definition (GRAPH)

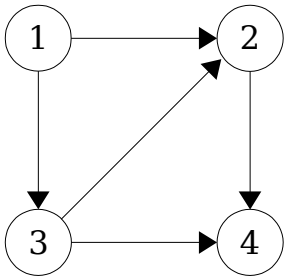
Ein Graph $G=(V,E)$ besteht aus
- einer endlichen Menge V von Knoten (Vortices), $V\neq 0$
- einer Menge E von Kanten (Edges), $E\subseteq V\times V$

Seien $v,w\in V$. Falls $(v,w)\in E$ geordnete Paare sind, dann heißt G gerichteter Graph, sonst ungerichteter Graph. ■

Graphische Darstellung:



ungerichteter Graph



gerichteter Graph (Digraph)

Definition:

Ein Weg der Länge $n-1$ in einem Graphen $G=(V,E)$ ist eine Folge von Kanten der Form

$$(v_1,v_2),(v_2,v_3),\ldots,(v_{n-2},v_{n-1}),(v_{n-1},v_n)$$

(1,3)(3,2)(2,4)

Weglänge: 3

Ein Zyklus ist ein Weg mit $v_1=v_n$.

Sei $G=(V,E)$ ein gerichteter Graph, $(v,w)\in E$:

- w ist Nachfolger von v
- v ist Vorgänger von w
- die Kante geht von v nach w. ■

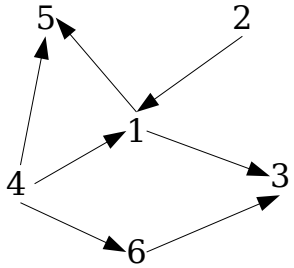
Definition: (GRAD)

v sei ein Knoten eines Graphen.

- $g^+(v):=|\{w\in V|(v,w)\in E\}|$ heißt Ausgangsgrad von v
- $g^-(v):=|\{u\in V|(u,v)\in E\}|$ heißt Eingangsgrad von v

Für ungerichtete Graphen ist $g^+(v)=g^-(v)=:g(v)$ der Grad des Knotens v. Ein Knoten mit dem Ausgangsgrad 0 heißt Senke, ein Knoten mit Eingangsgrad 0 heißt Quelle, sind Ein- und Ausgangsgrad 0, so heißt der Knoten isoliert. ■

Beispiel:



$g^+(1)=2, g^-(1)=2$, ferner sind 4 und 2 Quellen, 3 und 5 sind Senken und 7 ist ein isolierter Knoten.

Für ungerichtete Graphen gilt: $\sum_{v\in V} g(v)=2|E|$

Definition: (ZUSAMMENHANG)

Ein Graph heißt zusammenhängend, wenn für alle $v, w \in V$ ein Weg von v nach w existiert.

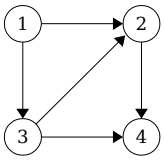
Gilt diese Aussage für einen gerichteten Graphen, so ist dieser stark zusammenhängend. ■

5.2 Implementierung

5.2.1 Adjazenzmatrix

Die Adjazenzmatrix A für einen Graphen $G=(V,E)$ ist eine $|V|\times|V|$ -Matrix mit

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

z.B. für 

hat man $A=\begin{pmatrix}0&1&1&0\\0&0&0&1\\0&1&0&1\\0&0&0&0\end{pmatrix}$

Für Digraphen ist A unsymmetrisch, für ungerichtete Graphen symmetrisch.

Vorteil:

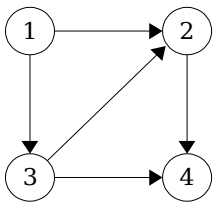
- einfacher Test, ob $(v,w)\in E$ gilt ($O(1)$)
- gut geeignet, falls das Problem statisch ist, und der Graph haben Vernetzungsgrad aufweist.

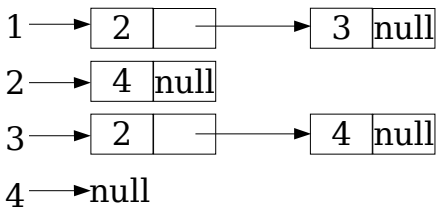
Nachteil:

- kein Einfügen neuer Knoten / Löschen
- $|V|^2$ Speicherbedarf, auch wenn Graph nur sehr wenig Kanten hat.

5.2.2 Adjazenzliste

In einer Adjazenzliste wird für jeden Knoten die Liste seiner Nachfolger verwaltet.

z.B. für 



Vorteil:

- $O(|V|+|E|)$ Speicherbedarf; sehr vorteilhaft, wenn $|E|\ll|V|^2$
- Einfügen neuer Knoten leicht möglich

Nachteil:

- Test, ob $(v,w)\in E$ teurer ($O(|V|+|E|)$ worst case)

Adjazenzlisten sind besser geeignet für Graphen mit niedrigem

Vernetzungsgrad, die sich dynamisch ändern können.

5.3 Anwendungen von Graphen

Graphen eignen sich zur Abbildung von Relationen zwischen Elementen einer Menge, z.B. bei Personen „sind befreundet miteinander“, oder bei Webseiten „enthält einen Link auf“.

Graphen werden in folgenden Bereichen oft eingesetzt:

- Verkehrs- und Transportprobleme
  - kürzeste Route
  - kürzeste Reisezeit
  - Rundreiseprobleme (Travelling Salesman)
- Produktions- und Ablaufplanung, Projektplanung
- Neuronale Netze, Markov-Ketten
- Elektrizitäts- und Datennetze
  - Ausfallsicherheit
  - maximaler Fluß
  - Zusammenhang

5.4 Standardproblemstellungen

Grundlegende Operationen:

- Überprüfung der Existenz einer Kante zwischen zwei Knoten
- Bestimmung der Nachbarknoten (bzw. Vorgänger und Nachfolger)
- Einfügen / Löschen von Knoten und Kanten
- Suche eines vorgegebenen Knotens

Diese Grundoperationen lassen sich leicht implementieren.

Grundoperationen werden verwendet, um immer wiederkehrende Standardprobleme zu lösen.

Standardprobleme mit effizienten Lösungen:

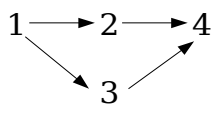
– Feststellen, ob ein Graph zusammenhängend ist

– Feststellen, ob ein Graph einen Zyklus hat, und diesen ggf. bestimmen

– Weg zwischen zwei Knoten finden

– Feststellen, ob ein Graph planar (kreuzungsfrei) ist.

– Eine topologische Sortierung konstruieren (nummeriere die Knoten eines gerichteten Graphen so, daß für jede Kante (v_i, v_j) stets gilt, daß $i<j$).



5.4.1 Durchwandern von Graphen

5.4.1.1 Tiefensuche / Depth-First-Suche (DFS)

Methode:

– Starte bei einem Knoten v

– Suche Kante (v,w) für noch nicht besuchtes w

– Falls existent: Setze Wanderung bei w fort, sonst: kehre zurück

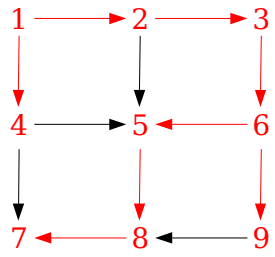
Falls der Graph nicht zusammenhängend ist, muß die Tiefensuche mehrmals gestartet werden (beginnend jeweils mit einem noch nicht besuchten Knoten).

Pseudocode:

```
procedure DFS(v)
begin
     besuche v
     markiere v als besucht
     für alle (v,w)∈E:
         falls w noch nicht besucht, dann DFS(w)
end
```

Komplexität: Für zusammenhängende Graphen $O(|E|)$

Beispiel:



Anwendung Tiefensuche:

– Suche nach einem ersten passenden Element

– Suche nach allen passenden Elementen

– Durchlaufen aller Knoten

schwarze Kanten: Nachfolgeknoten wurden schon besucht

DFS(1)
mögliche Reihenfolge: 1 – 2 – 3 – 6 – 5 – 8 – 7 – 9 – 4

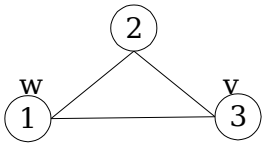
Die in der Tiefensuche verwendeten Kanten (rot) erzeugen einen Spannbaum.

Tiefensuche läßt sich zur Zyklenerkennung einsetzen. Dazu

- nummeriert man die Knoten v des Graphen während der Tiefensuche in Präordnung $o(v)$, d.h. bevor Nachfolger besucht werden. Initial setzt man die Präordnung aller Knoten auf ∞.
- merkt man sich für jeden Knoten, ob die Tiefensuche abgeschlossen ist (true / false).

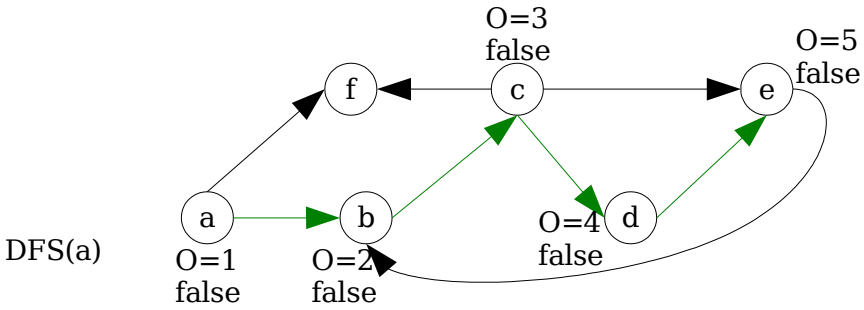
Ein Zyklus liegt vor, falls wir eine Rückwärtskante (v,w) entdecken.

Für Rückwärtskanten ist $o(w)<o(v)$ und die Tiefensuche für w noch nicht abgeschlossen.



Kante (v, w) mit O(w) ≤ O(v)

Beispiel:



Wir haben a,b,c,d,e mit Hilfe der grünen Kanten besucht. Von e aus untersuchen wir die weiteren Knoten. (e,b) Ist Rückwärtskante, da o(b)<o(e) und Tiefensuche für b nicht abgeschlossen (false).
→ Ein Zyklus liegt vor.

5.4.1.2 Breitensuche / Breadth-First-Search (BFS)

Methode:

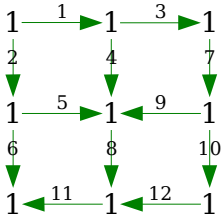
- Starte bei einem Knoten v.
- Besuche zuerst alle Nachbarn von v, bevor die Suche weiter in die Tiefe expandiert wird.

Pseudocode (q sei eine objektorientierte Implementierung einer Queue):

```
procedure BFS(v)
begin
     besuche v
     markiere v als besucht
     q.enqueue(v)
     solange q nicht leer
         s = q.dequeue()     //s ist schon am längsten in q
         für alle (s, w) aus E
             falls w nicht besucht
                 besuche w
                 markiere w als besucht
                 q.enqueue(w)
end
```

Komplexität: wie bei DFS O(|E|)

Beispiel:



Ziffern über Pfeilen = Reihenfolge, in der Kanten untersucht werden

BFS(1):
mögliche Reihenfolge: 1 – 2 – 4 – 3 – 5 – 7 – 6 – 8 – 9

Anmerkung: Falls $G=(V,E)$ ein Baum ist, dann entspricht eine an der Wurzel beginnende Tiefensuche einem Präorder-Durchlauf, Breitensuche entspricht einem schichtenweisen sequentiellen Durchlauf.

DFS(1): 1 – 2 – 4 – 5 – 3 – 6 – 7
BFS(1): 1 – 2 – 3 – 4 – 5 – 6 – 7

5.4.2 Transitive Hülle

Welche Knoten eines Digraphen können von einem beliebigen Startknoten aus erreicht werden (auf einem Weg beliebiger Länge)?

Algorithmus von Warshall

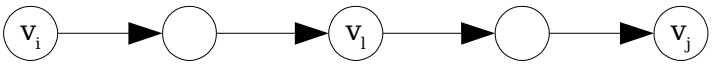
nutzt Darstellung von Graphen als Adjazenzmatrix

Grundidee: Überprüfe für alle Knoten v_i, v_j, für die man noch keinen verbindenden Weg hat, ob ein Weg von v_i nach v_j über einen weiteren Knoten v_k existiert.

Sei A^0 die Adjazenzmatrix, so errechnen wir eine Matrix A^1, die neben direkten Verbindungen zwischen den Knoten auch indirekte Verbindungen über einen Hilfsknoten enthält.

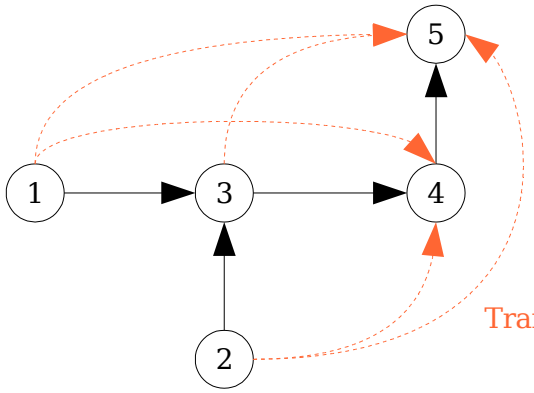
Der Prozeß wird solange wiederholt, bis sich die Matrix nicht mehr ändert (A^∞). Beim Übergang von A_{ij}^{k-1} nach A_{ij}^{k} gilt:

1. Ist $A_{ij}^{k-1}=1$: Weg zwischen v_i und v_j existiert: beibehalten
2. Noch kein Weg zwischen v_i und v_j gefunden: $A_{ij}^{k-1}=0$. Gibt es einen Weg von v_i nach v_l und von v_l nach v_j, so führt auch ein Weg von v_i nach v_j.



Setze $A_{ij}^{k}=1$.

Beispiel:



$$A^0=\begin{pmatrix}0&0&1&0&0\\0&0&1&0&0\\0&0&0&1&0\\0&0&0&0&1\\0&0&0&0&0\end{pmatrix}\qquad A^1=\begin{pmatrix}0&0&1&1&0\\0&0&1&1&0\\&&\dots&&\end{pmatrix}$$

$$A^\infty=\begin{pmatrix}0&0&1&1&1\\0&0&1&1&1\\0&0&0&1&1\\0&0&0&0&1\\0&0&0&0&0\end{pmatrix}$$

Aufgabe 5.1

Die Summe der Grade der Knoten wäre hier $7*5=35$, und dies kann nicht gleich zweimal der Anzahl der Kanten sein. Daher ist es nicht möglich.

Aufgabe 5.2

a) DFS: Durchlaufreihenfolge durch Knoten
   a – b – c – d – e – f – g – h – i
   DFS(a) DFS(g)

b) BFS:
   a – b – d – c – e – f – g – h – i
   BFS(a) BFS(g)

Sei A eine $n\times n$-Matrix:

```
for (int i=0; i<n; i++)
     for (int j=0; j<n; j++)
         if (A[i][j]==0)
             for (int l=0; l<n; l++)
                 if (A[i][l]==1 && A[l][j]==1) {
                     A[i][j]=1;
                     break;
                 }
```

In obigem Algorithmus wird nicht zwischen A^k und A^{k+1} unterschieden; aktualisierte Werte werden sofort verwendet.
→ Code berechnet A^∞ in einmaligem Durchlauf.

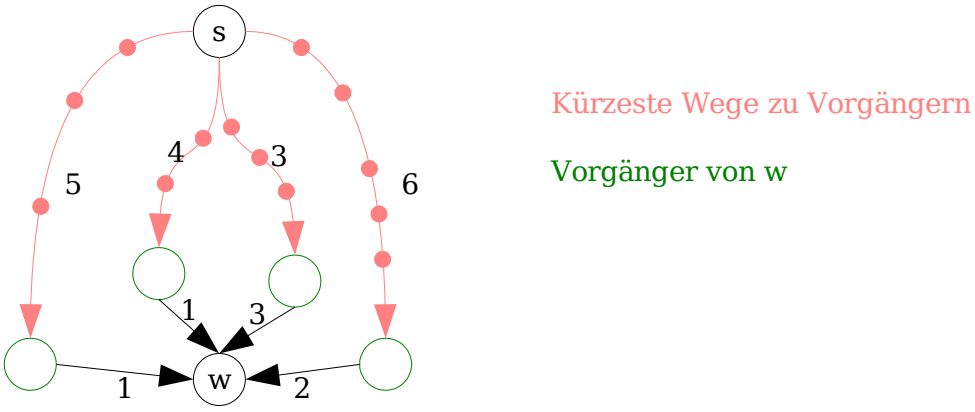
Komplexität: $O(|V|^3)$

5.4.3 Kürzeste Wege

Gegeben: Digraph mit einer Gewichtsfunktion $c: E \to \mathbb{R}^+$

Der Algorithmus von Dijkstra findet kürzeste Wege (mit minimaler Summe der Kantengewichte) von einem Startknoten S zu allen anderen Kanten.

Beobachtung: Sei w ein Knoten. Sind die kürzesten Wege von s zu den Vorgängern von w bekannt, so führt der kürzeste Weg von s nach w über denjenigen Vorgängerknoten v von w, so daß die Summe der Abstände von s nach v und von v nach w minimal ist.



Formal: Sei die Länge des kürzesten Weges zwischen zwei Knoten durch Funktion d gegeben und $V(w)$ die Menge der Vorgänger von w.

Dann ist

$$d(s,s)=0$$
$$d(s,w)=\min_{v \in V(w)} (d(s,v)+c(v,w))$$

Jeder Knoten ist entweder

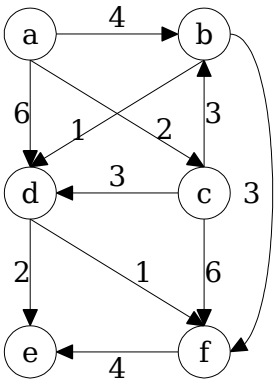
- innerer Knoten: minimaler Abstand von s ist bekannt.
- Randknoten: Weg von s zum Knoten ist bekannt, aber noch nicht unbedingt der kürzeste.
- äußerer Knoten: alle anderen.

Zu Beginn ist s der einzige innere Knoten. Die Nachfolger von s sind

Randknoten.

Die Menge der inneren Knoten wächst nun schrittweise von „innen nach außen“, indem unter allen Randknoten derjenige Knoten u mit dem kürzesten Abstand von s als nächster innerer Knoten genommen wird. Die Nachfolger von u werden, sofern sie bisher äußere Knoten sind, zu neuen Randknoten. Für Randknoten, die nun über u auf einem alternativen Weg erreicht werden können, kann sich ein neuer, kürzerer Abstand zu s ergeben, der dann den bisherigen kürzesten Weg ersetzt.

Beispiel:



Ermittlung der kürzesten Wege von s=a aus.

1. a ist innerer Knoten, b,c,d Randknoten
2. c hat minimalen Abstand von s, wird innerer Knoten. Nachfolger von c sind b,d,f. f wird neuer Randknoten. Dadurch, daß c innerer Knoten wird, gibt es einen kürzeren Weg von a nach d, nämlich über c.
3. usw.

Schritt	**Innere Knoten**		**Randknoten**		
	Knoten v	d(a, v)	Knoten	vorläufiger Abstand von a	Vorgänger
1	a	0	b	4	a
			c	2	a
			d	6	a
2	a	0	b	4	a
	c	2	d	5	c
			f	8	c
3	a	0	d	5	c
	c	2	f	7	b
	b	4			
4	a	0	f	6	d
	c	2	e	7	d
	b	4			
	d	5			

Schritt	Innere Knoten		Randknoten		
	Knoten v	d(a, v)	Knoten	vorläufiger Abstand von a	Vorgänger
5	a	0	e	7	d
	c	2			
	b	4			
	d	5			
	f	6			
6	a	0			
	c	2			
	b	4			
	d	5			
	f	6			
	e	7			

Komplexität Dijkstra-Algorithmus: $O((|V|+|E|)\cdot\log|V|)$

5.4.4 Flußprobleme, Transportnetze

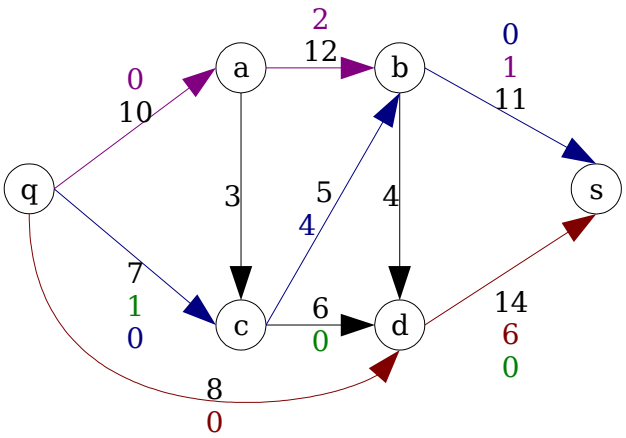
Simulation von Transportwegen (Straßen, Rohre, etc.) durch gerichtete azyklische Graphen.

Kantenfunktion: $c: E \to \mathbb{R}^+$ gibt Maximalkapazität an. Fluß durch Kante muß Kapazitätsbeschränkung der Kante einhalten.

Ein Knoten wird als Quelle q ausgezeichnet, einer als Senke s.

Ziel: Maximalen Fluß von u nach s ermitteln.

Beispiel:



Restkapazität nach q → a → b → s

Restkapazität nach q → d → s

Restkapazität nach q → c → d → s

Restkapazität nach q → c → b → s

Algorithmus nach Ford und Fulkerson:

- Setze alle Flüsse auf 0.
- Suche Pfad von der Quelle zur Senke, auf dem die genutzte Kapazität um selben Betrag erhöht werden kann.
- Falls kein solcher Pfad mehr gefunden wird, ist der maximale Fluß erreicht.

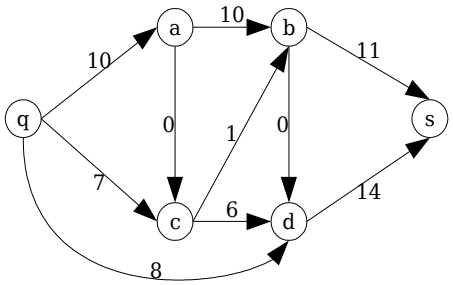
z.B. für Graphen oben:

1. Schritt: Setze alle Flüße auf 0.
2. Schritt: Wir wählen Weg $q \to a \to b \to s$. Maximaler Fluß auf diesem Weg: 10. Wir ziehen dies von den betroffenen Kanten ab.
3. Schritt: Weg $q \to d \to s$ mit maximalem Fluß auf diesem Weg: 8
4. Schritt: Weg $q \to c \to d \to s$ mit maximalem Fluß auf diesem Weg: 6
5. Schritt: Weg $q \to c \to b \to s$ mit maximalem Fluß auf diesem Weg: 1

Kanten von der Quelle zu Nachfolgern alle erschöpft → Fertig.

Maximaler Fluß durch Graphen $= \sum$ der maximalen Flüsse durch Wege $= 10+8+6+1=25$

Tatsächlich genutzte Kapazitäten:



Beobachtungen:

- In jedem Knoten (außer q und s) ist in der Lösung eingehender gleich ausgehender Fluß (Kirchhoffsches Gesetz)
- Von q ausgehender Fluß = in s eintreffender Fluß = Gesamtfluß.

Komplexität von Ford-Fulkerson: $O(|V| \cdot |E| \cdot \log |V|)$

5.4.5 Minimal aufspannende Bäume (MAB)

Bäume sind Spezialfälle von Graphen:

Definition (Baum):

Ein gerichteter Graph $G=(V,E)$ heißt Baum, wenn gilt:

a. G ist zyklenfrei

b. Es gibt genau einen Knoten $r \in V$ ohne Vorgänger, den Wurzelknoten.

c. Jeder Knoten außer r hat genau einen Vorgänger

d. Es gibt einen Weg von r zu jedem $v \in V$.

Für ungerichtete Graphen fordert man Zyklenfreiheit und Zusammenhang. ■

Teilgraphen $S=(V,T)$ eines zusammenhängenden, ungerichteten Graphen $G=(V,E)$ mit $T \subseteq E$ heißen aufspannende Bäume, wenn sie Bäume sind und alle Knoten verbinden.

Für die Kanten sei Kostenfunktion $c: E \to \mathbb{R}$ gegeben.

Gesucht: Minimal aufspannender Baum $S=(V,T)$ mit der Kantenmenge T, die

die Summe der Konsten minimiert:

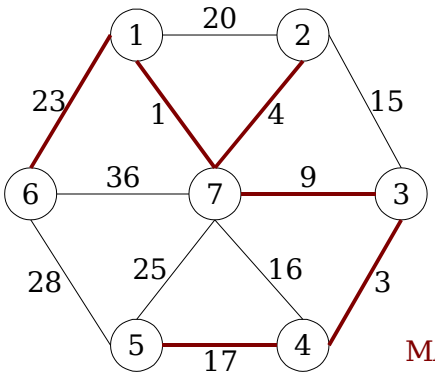
$$c(S) := \sum_{e \in T} c(e) \text{ minimal}$$

z.B. Stromversorgung:

$V =$ Ortschaften, $E:$ mögliche Streckenführungen für Stromleitungen

$c(e):$ Kosten Leitungserstellung + Unterhalt, $T:$ günstigstes Netz

Beispiel:



MAB

Algorithmus von Kruskal

- Starte mit lauter isolierten Knoten
- Wähle billigste Kante und füge sie hinzu
- Füge weiter billigste Kanten hinzu, solange Baumeigenschaft nicht verletzt wird.
- Ende erreicht, wenn alle Knoten verbunden.

Systematisches Vorgehen:

Geordnete Kantenmenge:

E	c
(1,7)	1
(3,4)	3
(2,7)	4
(3,7)	9
(2,3)	15
(4,7)	16
(4,5)	17
(1,2)	20
(1,6)	23
(5,7)	25
(5,6)	28
(6,7)	36

Schritt	E	Aktion	Verbundene Knotenmengen
1	(1,7)	add	{1, 7}, {2}, {3}, {4}, {5}, {6}
2	(3,4)	add	{1, 7}, {2}, {3, 4}, {5}, {6}
3	(2,7)	add	{1, 2, 7}, {3, 4}, {5}, {6}

Schritt	E	Aktion	Verbundene Knotenmengen
4	(3,7)	add	{1, 2, 3, 4, 7}, {5}, {6}
5	(2,3)	skip	
6	(4,7)	skip	
7	(4,5)	add	{1, 2, 3, 4, 5, 7}, {6}
8	(1,2)	skip	
9	(1,6)	add	{1, 2, 3, 4, 5, 6, 7}

Für durch Adjazenzliste dargestellte Graphen $G=(V,E)$ kann MAB in $O(|E|\log|E|)$ konstruiert werden.

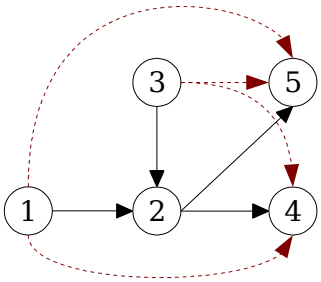
Aufgabe 5.7

Schritt	Innere Knoten		Randknoten		
	Knoten v	$d_{min}(a, v)$	Knoten v	d(a,v)	Vorgänger
1	a	0	b	1	a
			c	13	a
			e	29	a
			f	11	a
2	a	0	c	3	b
	b	1	d	18	b
			f	11	a
			e	29	a
3	a	0	d	6	c
	b	1	e	29	c
	c	3	f	11	a
4	a	0	e	11	d
	b	1	f	11	a
	c	3			
	d	6			
5	a	0			
	b	1			
	c	3			
	d	6			
	e	11			
	f	11			

Aufgabe 5.8

$$A^0=\begin{pmatrix}0&1&0&0&0\\0&0&0&1&1\\0&1&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{pmatrix}\qquad A^1=\begin{pmatrix}0&1&0&1&1\\0&0&0&1&1\\0&1&0&1&1\\0&0&0&0&0\\0&0&0&0&0\end{pmatrix}=A^\infty$$

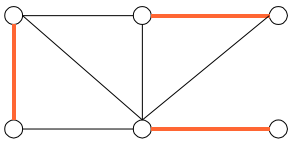
5.4.6 Matchings



Matching in einem Graphen $G=(V,E)$ ist Teilmenge der Kanten $M\subseteq E$, so daß keine zwei Kanten aus M einen Knoten gemeinsam haben.

Perfektes Matching: Alle Knoten des Graphen werden erfaßt.

Beispiel:

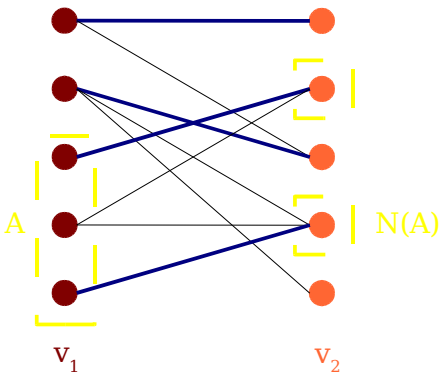


perfektes Matching

Anwendung: Bilden von möglichst vielen kooperierenden Zweiterteams aus einer Gruppe von Leuten (Kanten = Kooperationsbereitschaft).

Matchings werden oft für bipartite Graphen gesucht. Bei bipartiten Graphen können die Knoten in zwei Teilmengen V_1 und V_2 aufgeteilt werden, so daß keine Kanten zwischen Knoten aus der selben Menge existieren.

Beispiel:



nicht-perfektes Matching

Sei $N(A)$ die Menge der Nachbarn einer Knotenmenge A.

Satz (Heiratssatz von Hall)

Sei der biparte Graph $G=(V_1 \cup V_2, E)$ gegeben. Dann gibt es perfektes Matching genau dann, wenn $|N(A)| \geq |A|$ für alle Teilmengen A von V_1 oder V_2 gilt. ■

Warum Heiratssatz?

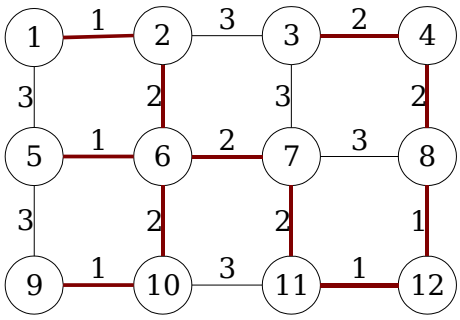
V_1=Damen, V_2=Herren, Kante vorhanden =Heirat nicht ausgeschlossen

Satz von Hall besagt: Genau dann Kommen alle Damen und Herren unter die Haube, wenn für jede Menge von Damen gilt, daß sie an mindestens genausovielen Herren Interesse haben und umgekehrt.

Anwendung: V_1=Projektleiter, V_2=Projekte. Jeder Projektleiter ist für bestimmte Projekte qualifiziert, eines soll ihm zugeteilt werden.

Lösung mit dem Algorithmus von Ford und Fulkerson: Quelle und Senke hinzufügen; Knoten von V_1 mit Quelle, von V_2 mit Senke verbinden; Kantengewichte auf konstant 1 setzen.

Aufgabe 5.4.a



MAR

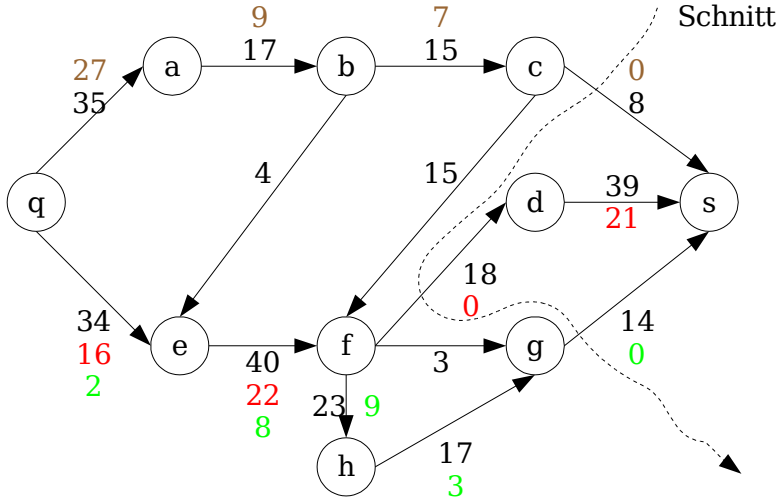
Geordnete Kantenmenge

(1,2)	1
(5,6)	1
(9,10)	1
(8,12)	1
(11,12)	1
(3,4)	2
(2,6)	2
(4,8)	2
(6,7)	2
(6,10)	2
(7,11)	2

E	Aktion	Verbundene Knotenmengen (ohne isolierte Knoten)
(1,2)	add	{1, 2}
(5,6)	add	{1, 2}, {5, 6}
(8,12)	add	{1, 2}, {5, 6}, {8, 12}
(9,10)	add	{1, 2}, {5, 6}, {8, 12}, {9, 10}
(11,12)	add	{1, 2}, {5, 6}, {8, 11, 12}, {9, 10}

E	Aktion	Verbundene Knotenmengen (ohne isolierte Knoten)
(3,4)	add	{1, 2}, {5, 6}, {8, 11, 12}, {9, 10}, {3, 4}
(2,6)	add	{1, 2, 5, 6}, {8, 11, 12}, {9, 10}, {3, 4}
(4,8)	add	{1, 2, 5, 6}, {3, 4, 8, 11, 12}, {9, 10}
(6,7)	add	{1, 2, 5, 6, 7}, {3, 4, 8, 11, 12}, {9, 10}
(6, 10)	add	{1, 2, 5, 6, 7, 9, 10}, {3, 4, 8, 11, 12}
(7, 11)	add	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Aufgabe 5.9



q → e → f → d → s: 18

q → a → b → c → s: 8

q → e → f → h → g → s: 14

40 maximaler Fluß

6 Bäume

6.1 Definitionen, Eigenschaften

Formale Definition eines Baums als Graph s. 5.4.5

Verallgemeinerung der Liste: Knoten kann mehr als einen Nachfolger haben.

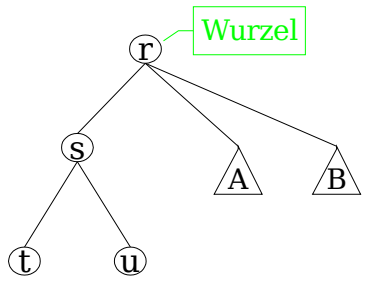
Definition (Rekursive Definition)

Ein Baum von Objekten vom Typ T ist ein Objekt vom Typ T (Wurzel), zusammen mit $n \geq 0$ Bäumen von Objekten vom Typ T (Unterbäume). ■

Gebräuchliche Darstellungen:

a. Graphendarstellung:
Notation für einen einzelnen Knoten: Ⓧ

Notation für einen einzelnen Unterbaum (UB):
Konvention in der Darstellung: Bäume wachsen von oben nach unten



(Kantenrichtung muss nicht angegeben werden)

b. Mengenschreibweise

$\{r, \underbrace{\{s,\{t\},\{u\}\}}_{\text{linker UB}}, \underbrace{\{\ldots\}}_{\text{mittlerer UB A}}, \underbrace{\{\ldots\}}_{\text{rechter UB B}}\}$

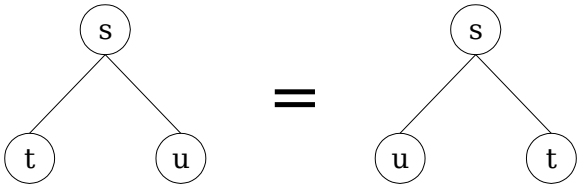
(Wurzel gefolgt von Unterbäumen)

Definition (Begriffe)

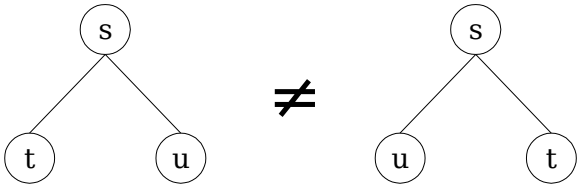
Sei Baum $T=(N,E)$ mit den Knoten N (nodes) und Kanten E gegeben, $E \subseteq N \times N$; m,n seinen Knoten.

1. $(m,n) \in E$: m ist Elternknoten von n und n ist Kind von m
2. $(m,n_1) \in E, (m,n_2) \in E$: n_1, n_2 sind Geschwister
3. - Grad eines Knotens = Anzahl seiner Kinder
   - Grad eines Baums = maximaler Grad seiner Knoten
4. - Ein Knoten mit Grad 0 heißt Blatt
   - Ein Knoten mit Grad >0 heißt innerer Knoten
5. Tiefe eines Knotens $n \in N$: Länge des Wegs von der Wurzel bis n.
6. - Höhe eines Knotens $n \in N$: Länge des längsten Weges von n bis zu einem Blatt
   - Höhe eines Baumes T: Höhe der Wurzel von T. ■

Bei orientierten Bäumen ist Reihenfolge der UB unwichtig:



Bei geordneten Bäumen ist Reihenfolge der UB wichtig:

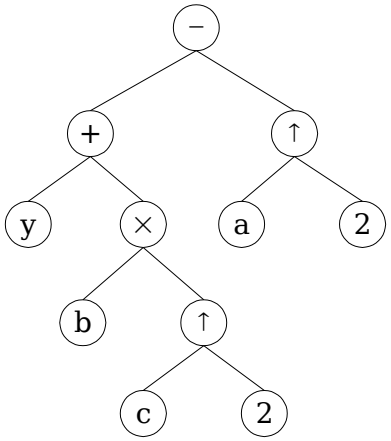


Beispiel: Syntaxbäume für arithmetische Ausdrücke

$y+b \times c \uparrow 2 - a \uparrow 2$

6.2 Anwendungen

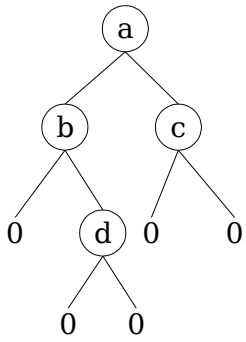
- Kompression
- Datenbanksysteme, Sortier- und Suchverfahren
- Backtracking-Algorithmen
- Dateisysteme
- Prozesshierarchie unter UNIX



- Syntaxbäume in Compilerbau
- Implementation von Mengen
- ...

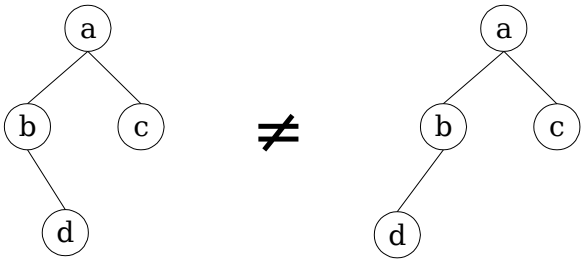
6.3 Binärbäume und m-Bäume

Binärbäume sind geordnete Bäume vom Grad 2. Ein Knoten hat immer genau zwei Unterbäume, wobei der leere Unterbaum (0) zugelassen ist.



Leere UB können weggelassen werden

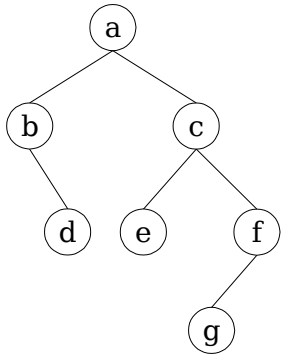
Wegen Binärbaum ist geordneter Baum:



Anmerkungen:

- Binärbaum kann zu Liste entartet sein.
- Alle Bäume sind zu Binärbäumen äquivalent. (Übung 6.2)

Durchlauf von Binärbaum:



a. Präordnung:
Knotenwert, linker UB, rechter UB (entspricht Tiefensuche)
a – b – d – c – e – f – g

b. Inordnung:
linker UB, Knotenwert, rechter UB
b – d – a – e – c – g – f

c. Postordnung:
linker UB, rechter UB, Knotenwert
d – b – e – g – f – c – a

d. Schichtenweise (entspricht Breitensuche)
a – b – c – d – e – f – g

Bei Syntaxbäumen: Präordnung → Präfixnotation
Inordnung → Infixnotation
Postordnung → Postfixnotation

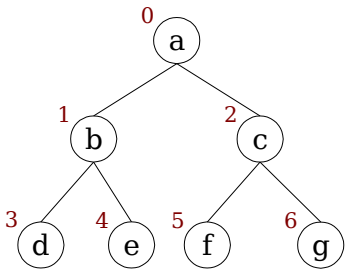
m-Bäume sind Verallgemeinerungen der Binärbäume für Grad >2.

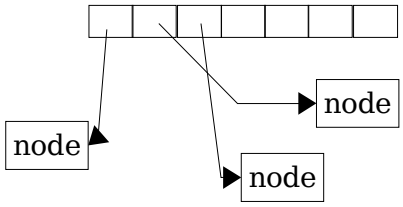
Ein m-Baum heißt vollständig, wenn jeder Pfad von der Wurzel zu einem Blatt gleich lang ist und alle inneren Knoten genau m nicht-leere Kinder haben.

6.3.1 Implementierung

Methode I

Schichtenweise, sequentielle Abspeicherung in einem Array



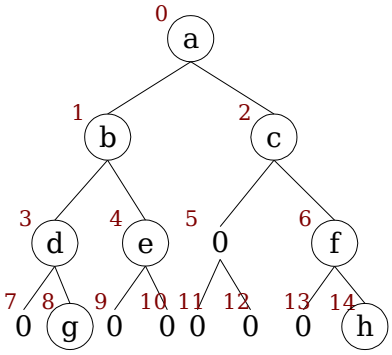


Ablage der Knoten (oder von Referenzen / Zeigern auf die Knoten) in Array x [0] ... x[6]

Zugriff durch Adressrechnung:

- Wurzel: x[0]
- linker UB von x[k]: x[2k+1]
- rechter UB von x[k]: x[2(k+1)]
- Elternknoten von x[k]: x[(k-1)/2] (Integerdivision)

Behandlung nicht-vollständiger Binärbäume:



Codierung:

a – b – c – d – e – 0 – f – 0 – g – 0 – 0 – 0 – 0 – 0 – h

Methode II

Implementierung als rekursive DS mit Zeigern / Referenzen

```
public class BinTree {
     private Object item;    //Nutzdaten
     private BinTree left;   //linker UB
     private BinTree right;  //rechter UB
     private BinTree parent; //Elternknoten
     ...
}
```

6.4 Heaps

Gegeben sei eine Ordnung auf den Nutzdaten (≤, <).

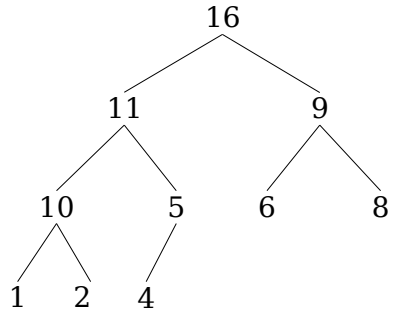
Definition (HEAP)

Ein Binärbaum T der Höhe h mit $n=2^h-1+l,\ 0<l<2^h$ Knoten heißt Heap, wenn er folgende Eigenschaften hat:
shape property: (links ausgerichteter vollständiger Baum)

- T ohne seine Blätter der Tiefe h ist ein vollständiger Binärbaum der Höhe $h-1$.
- Die am weitesten links stehenden $\left\lceil \frac{l}{2} \right\rceil$ Knoten der Höhe $h-1$ haben genau zwei Kinder (bis auf den letzten, der evtl. nur ein Kind hat).

order property: Für alle inneren Knoten gilt: Wert des Knotens ist größer (oder gleich) als Werte der Kindknoten. ■

Maximum der Werte eines Heaps steht im Wurzelknoten.



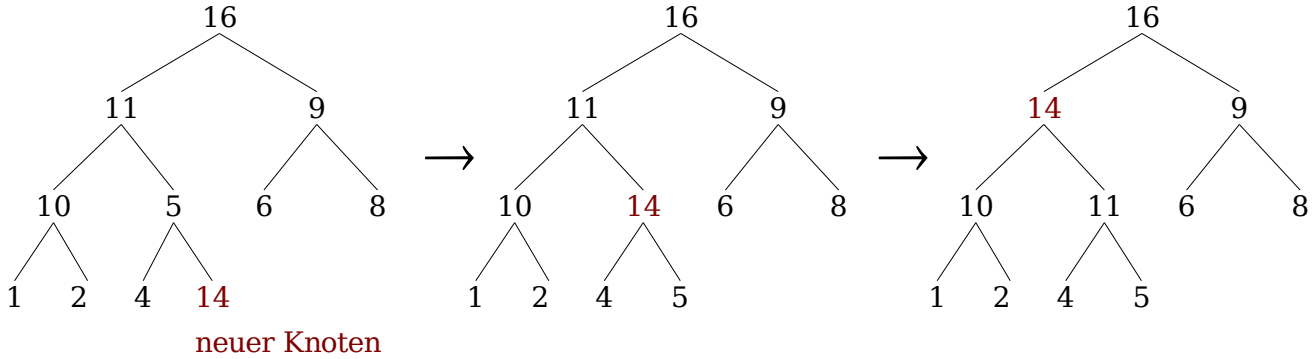
Heap mit $h=3, l=3, n=10$

Schichtenweise, sequentielle Abspeicherung bietet sich an.

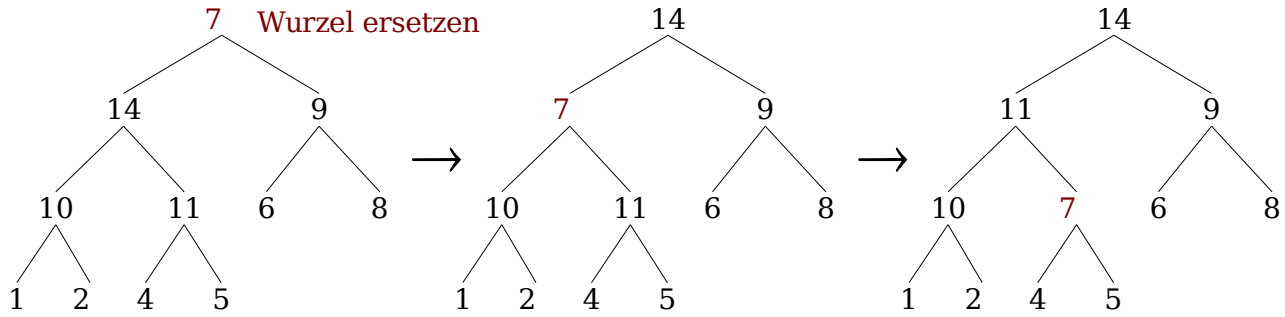
Order property: $x[k] \le x[(k-1)/2] \quad \forall 1 \le k \le n-1$

6.4.1 Basis-Operation auf einem Heap

1. siftup: Einfügen eines Elements am Ende des Heaps, dann nach oben „sickern“ lassen (d.h. mit Elternknoten vertauschen, falls der kleiner ist)
   → Wiederherstellen der order property.

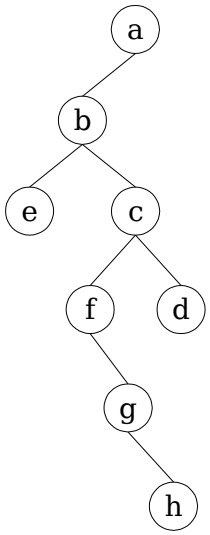


2. Siftdown: Ersetzen des Elements an der Spitze des Heaps $x[0]$. Wiederherstellen der „order property“ dadurch, daß man das neue Element nach unten „sickern“ läßt. Sind beide Kinder größer, vertauscht man mit dem größeren.



Komplexität siftup, siftdown: $O(\log n)$

Aufgabe 6.2



Aufgabe 6.3

```
     public class BinTree {
          private Object element; //Nutzdatum
          private BinTree left;
          private BinTree right;

          int hoehe = 0;
```

a)

```
          public int height() {
              int links=0, rechts=0;
              if (left != null)
                  links = left.height()+1;
              if (right != null)
                  rechts = right.height()+1;
              return max(links, rechts);
          }
```

b)

```
        public int nrInnerNodes() {
            if (right==null && left==null)
                return 0;
            if (left != null)
                return left.nrInnerNodes()+1;
            if (right != null)
                return right.nrInnerNodes()+1;
            return right.nrInnerNodes()+left.nrInnerNodes()+1;
        }
    }
```

6.4.2 Anwendungen von Heaps

Prioritätswarteschlangen

Priorisierte Tasks werden in beliebiger Reihenfolge eingefügt. Bei Bedarf wird der Task mit der höchsten Priorität entnommen.

Operationen:

- `void insert (int)` – Task mit Priorität einfügen
- `int extractMax()` – Task mit höchster Priorität entnehmen

Implementierung mit einem Heap x[0] .... x[n-1]

- insert läßt sich mit siftup realisieren
  - Neues Element ans Heap-Ende x[n] stellen
  - Aufruf von siftup
  - n inkrementieren
- extractMax
  - Entnahme x[0] (größter Wert)
  - Weise x[0] den Wert des letzten Heap-Eintrags x[n-1] zu
  - Dekrementiere n (Heap wird kleiner)
  - Aufruf von siftdown

Datenstruktur	insert	extractMax
Sortierte Liste	$O(n)$	$O(1)$
Unsortierte Liste	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$

(n = Anzahl der Tasks)

Heaps sind Listen bei der Implementierung von Prioritätswarteschlangen überlegen.

Heapsort

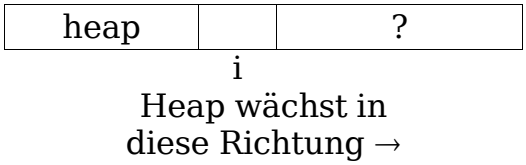
Einfache Implementierung mit Hilfsarray:

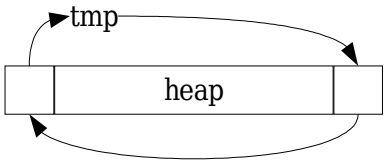
- Lies Elemente nacheinander vom Originalarray und füge sie mit insert in Hilfsarray ein, so daß dieses am Ende Heap enthält.

- Hole nacheinander alle Elemente aus dem Hilfsarray mit extractMax und schreibe sie von hinten nach vorn ins Originalarray zurück.

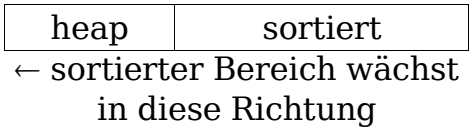
Komplexität: $O(n \log n)$ (optimale asymptotische Komplexität für Sortieralgorithmen)

Heapsort kann auch ohne Hilfsarray implementiert werden. Dazu wird das Array zunächst in-place in Heap umgewandelt, indem man für die Elemente von links nach rechts siftup aufruft:





Danach kann man mit extractMax (s.o.) die Maximalelemente von x[0] entnehmen und an das Ende des Heaps anfügen. Damit wächst sortierte Sequenz vom Array-Ende nach links:



6.5 Binäre Suchbäume

Im folgenden sei eine totale Ordnung „<“ auf den Elementen gegeben.

6.5.1 Definition

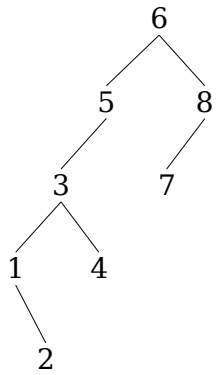
Definition:

Ein Binärbaum heißt binärer Suchbaum, wenn für alle Knoten k gilt:

a. Alle Knoten des linken Unterbaums sind kleiner als k.

b. Alle Knoten des rechten Unterbaums sind größer als k. ■

Beispiel:



Inorder-Durchlauf liefert sortierte Sequenz.

6.5.2 Grundlegende Operationen

Suchen

Rekursiv implementiert:

```
public class BinSearchTree {
     private int root;            //Nutzdaten
     private BinSearchTree left;  //linker UB
     private BinSearchTree right; //rechter UB
     ... //ctor

     public boolean search (int Key) {
         if (Key < value) {
             if (left == null)
                 return false;
             return left.search(Key); //Suche nach links forts
         }
         if (Key > value) {
             if (right == null)
                 return false;
             return right.search(Key);
         }
         // Key == value
         return true;
     }

     ...
} //class
```

Einfügen

Kann mit modifizierter Suche implementiert werden. Wird das Element nicht gefunden, weil Suche erfolglos an Knoten endet (z.B. Key < value und aber left == null), dann das Element hier richtig anhängen.

Löschen / Entfernen

Zuerst den zu entfernenden Knoten suchen. Fallunterscheidung:

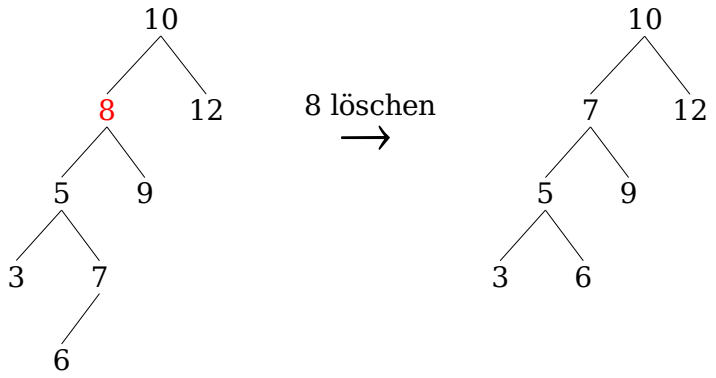
a. Knoten wird nicht gefunden: fertig

b. Knoten ist ein Blatt: Streichen

c. Knoten hat genau einen nicht-leeren UB: Ersetze Knoten durch UB

d. Knoten hat zwei nicht-leere UB: Ersetze Knoten durch das größte Element im linken UB oder das kleinste Element im rechten UB.

Beispiel:



Bemerkungen:

- Löschoperation nach d. zieht eine Löschoperation nach Schema b. oder c. nach sich.
- Größtes Element im linken UB: das am weitesten rechts befindliche

Komplexitäten für bin. Suchbaum T:

Suchen
Einfügen $O(h(T))$ (wobei h(T) = Höhe des Baumes T)
Entfernen

- Worst Case: Entartung des binären Suchbaums zu Liste
  $h(T)=n$ (n = Anzahl der Knoten)
- $O(\log n)$ im Average Case (z.B. Aufbauen eines binären Suchbaums mit Zufallszahlen)

6.6 AVL-Bäume

Adelson, Velski, Landis

„balancierte“ oder „ausgewogene“ Bäume

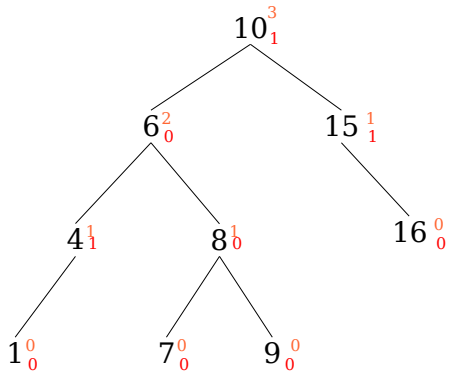
Bezeichnung: Knoten k → $L(k)$: linker UB von k
$R(k)$: rechter UB von k

6.6.1 Definition

Definition:

Ein binärer Suchbaum heißt AVL-Baum, wenn das Balancierungs-Kriterium (BK) für alle Knoten k erfüllt ist:
$|h(L(k))-h(R(k))|\leq 1$
(Dabei gelte per Konvention $h(0)=-1$) ■

Beispiel:



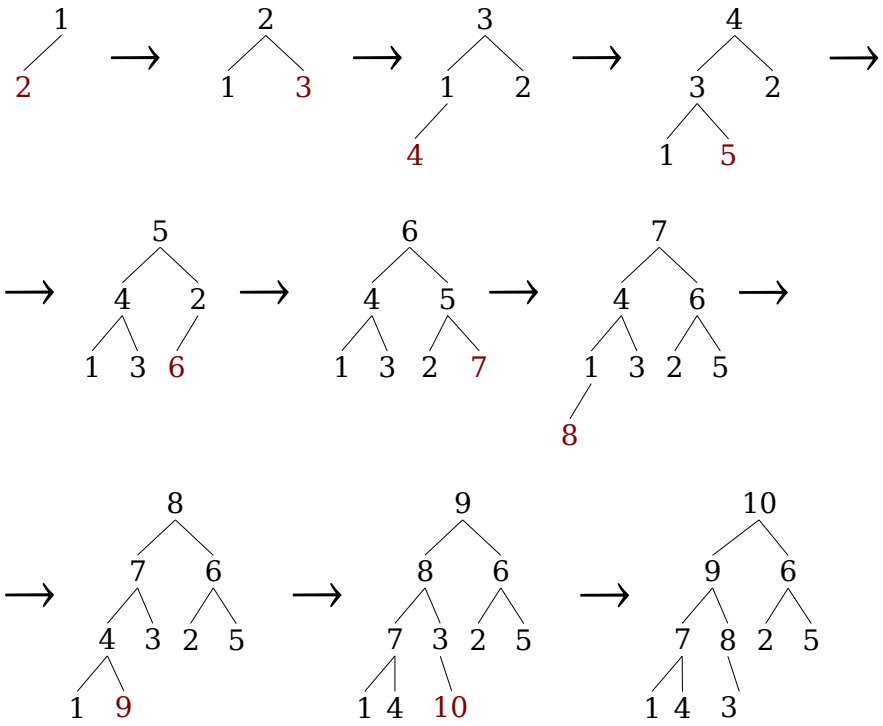
Höhe
|h(L(k)) - h(R(k))|

Ohne Beweis:

Satz

Ein AVL-BAum ist höchstens 45% höher als der entsprechende vollständige binäre Baum ($\rightarrow O(\log n)$ für die Suche, n = Anzahl Knoten). ■

Übung 6.4



6.6.2 Grundlegende Operationen

Suchen: wie beim binären Suchbaum

Einfügen: Zunächst wird neues Element wie beim binären Suchbaum eingefügt. Allerdings kann dies das BK verletzen

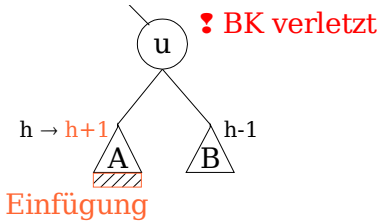
Korrektur der Balanzierungsschäden durch Rotationen:

– Nur die Knoten, die auf dem Weg von der Wurzel zum eingefügten Knoten liegen, verletzen evtl. das BK.

– BK wiederherstellen

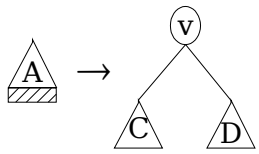
– Suchbaumeigenschaft erhalten

Vorgehen: Suche den Knoten mit der größten Tiefe, an dem das BK verletzt ist:



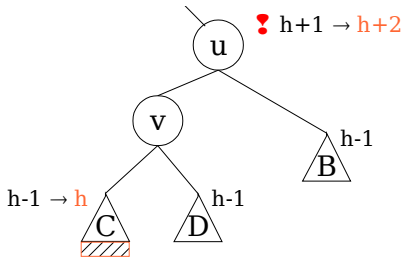
(Spiegelsymmetrischer Fall völlig analog)

Bei dieser Detaillierung keine Korrekturmöglichkeit. Zerlegung UBA:



Schadensfall 1:

Es ist der linke UB C von A verlänger worden:



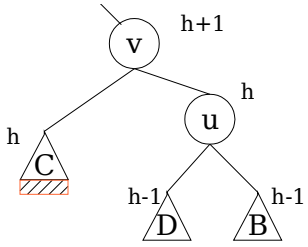
Dieser Fall (linker UB des linken UB wurde verlängert) läßt sich durch eine einfache Rotation beheben (hier: Rechtsrotation).

Sortierung

Reihenfolge ≠ Sortierung; Klammerung ≠ Baumstruktur

vorher: $(C \vee D) \wedge B$

nachher: $C \vee (D \wedge B)$

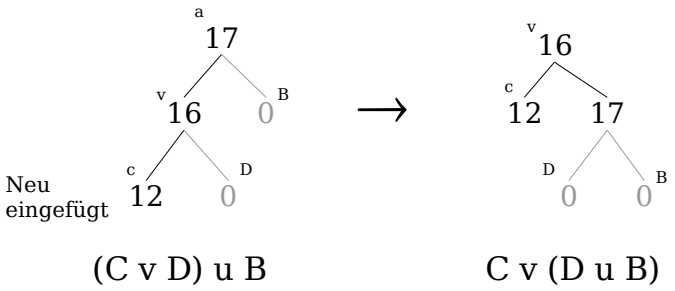


Einfügen und Rotation ändert Höhe nicht.

Suchbaumeigenschaft ✓

BK ✓

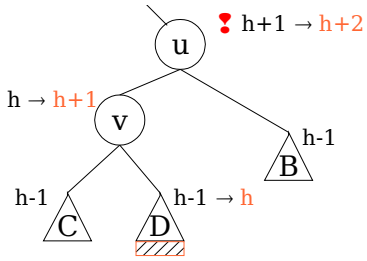
Beispiel



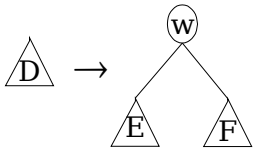
(C v D) u B C v (D u B)

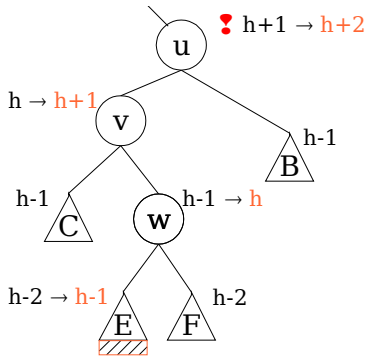
Schadensfall 2:

Es ist der rechte UB D von A verlängert worden:



Keine Korrekturmöglichkeit bei dieser Detaillierung. Zerlegung von D:



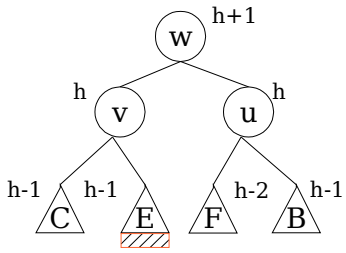


Sortierung

vorher: (C v (E w F)) u B

nachher: (C v E) w (F u B)

(Es ist egal, ob E oder F verlängert wurde.)



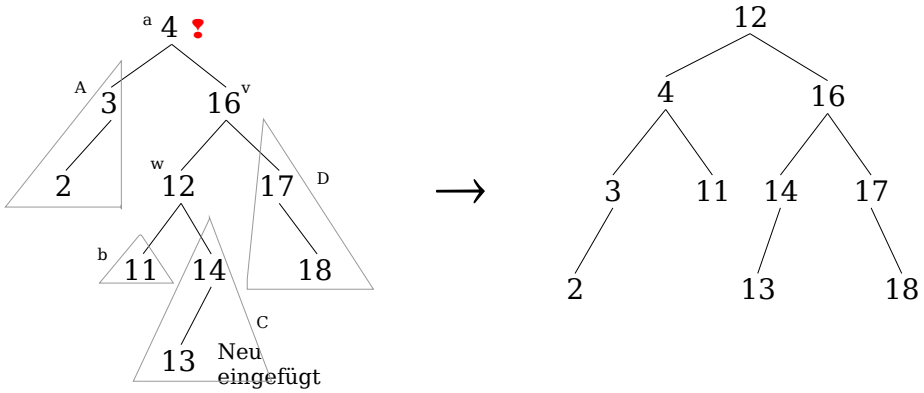
Doppelrotation (hier: Links-Rechts-Rotation)

Einfügen + Rotation ändert Höhe des Baums nicht.

Suchbaum ✓

BK ✓

Beispiel

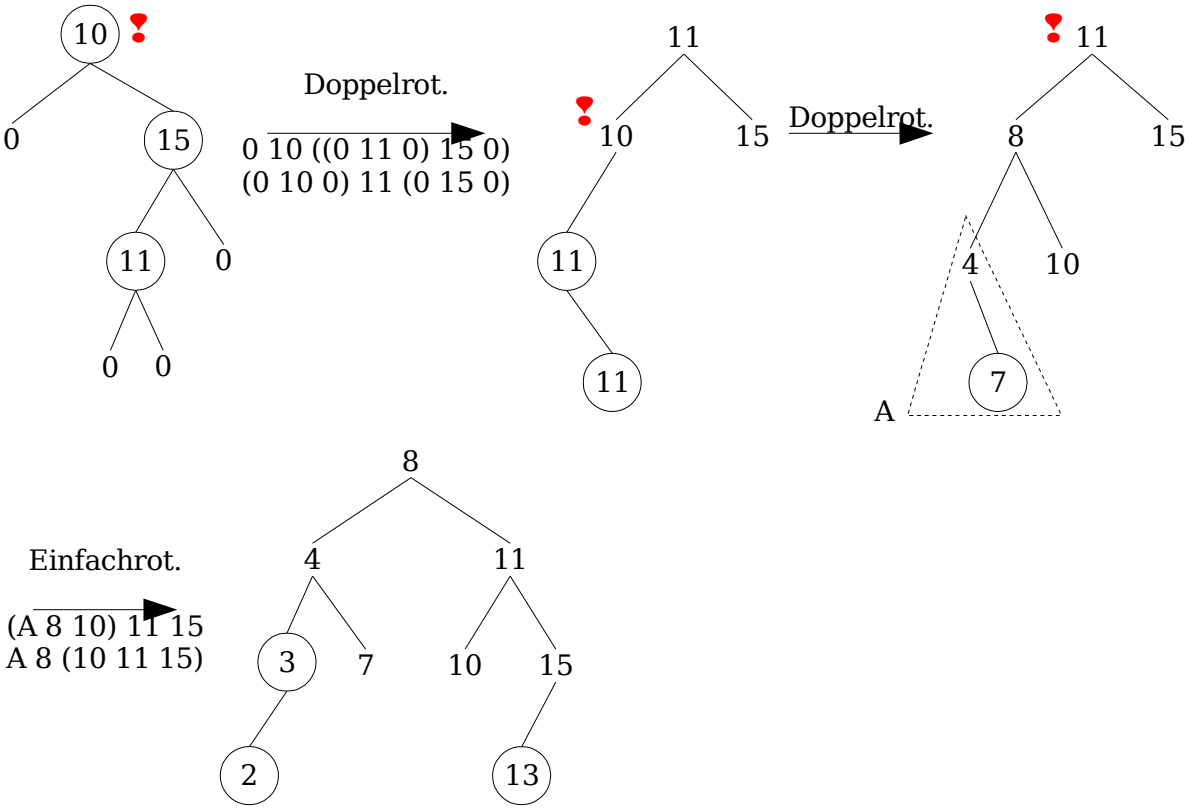


linker UB des rechten UB verlängert → Doppelrotation

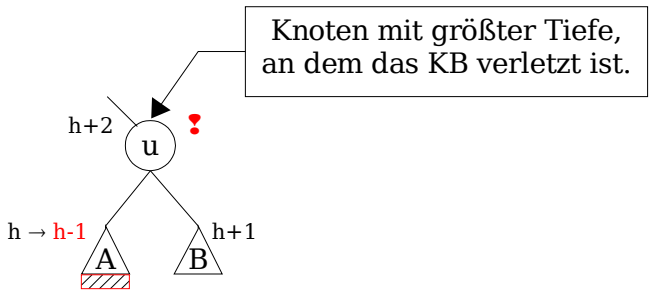
$A\,u\,((B\,w\,C)\,v\,D)$
$(A\,u\,B)\,w\,(C\,v\,D)$

Übung 6.6

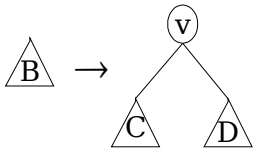
Entfernen



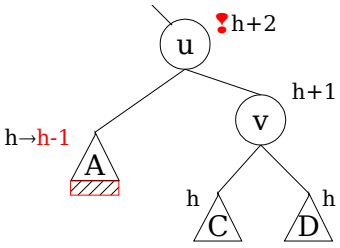
Auch hier wird zunächst wie beim binären Suchbaum verfahren. Wie beim Einfügen kann dadurch BK verletzt werden.



Bei dieser Detaillierung keine Korrekturmöglichkeit. Wir zerlegen UB B



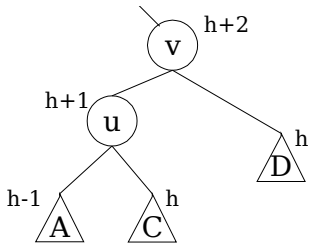
Situation:



Läßt sich durch Einfachrotation (Linksrotation) beheben:

A u (C v D)

(A u C) v D



Balanzierung ✓

Suchbaumeigenschaft ✓

Entfernen + Rotation ändert Höhe des UB nicht (keine weiteren Rotationen nötig)

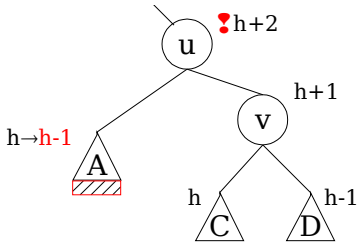
Schadensfall 2

Höhe von C sei h-1
Höhe von D sei h

auch hier reicht eine Einfachrotation (wie Schadensfall 1). Allerdings hat der resultierende Unterbaum die Höhe h+1 statt h+2. U.U. weitere Rotationen nötig.

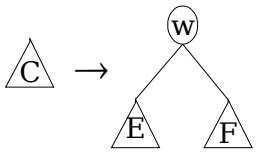
Schadensfall 3

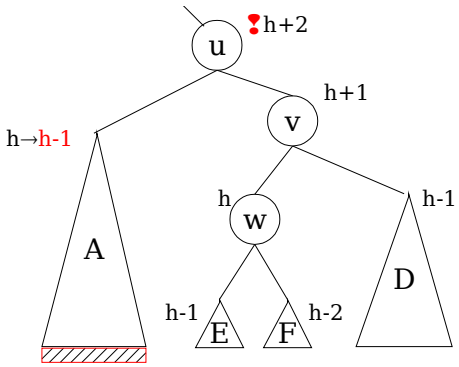
Höhe von C sei h
Höhe von D sei h-1



Linksrotation hilft hier nicht!

Keine Korrekturmöglichkeit bei dieser Detaillierung. Zerlegung von C



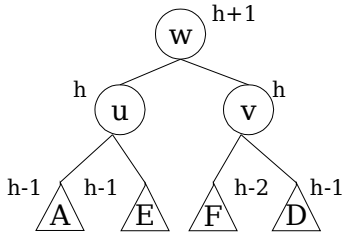


(Man kommt auf gleiches Ergebnis, wenn E Höhe h-2 und F Höhe h-1 hat, oder beide Höhe h-1.)

A u ((E w F) v D) vorher

(A u E) w (F v D) nachher

Doppelrotation (hier: Rechts-Links-Rotation)

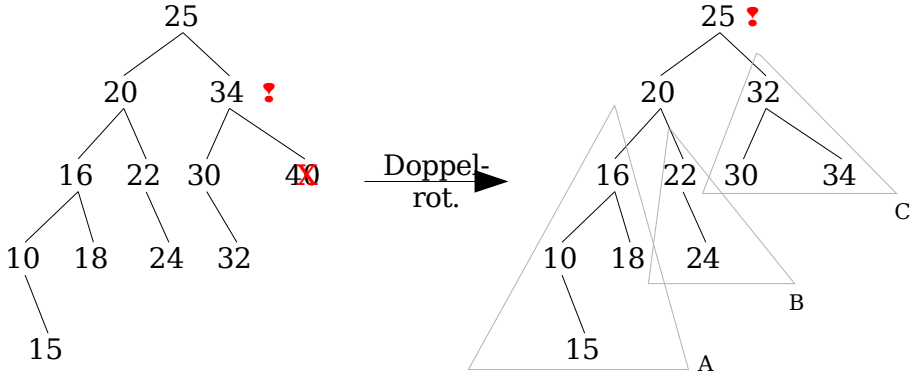


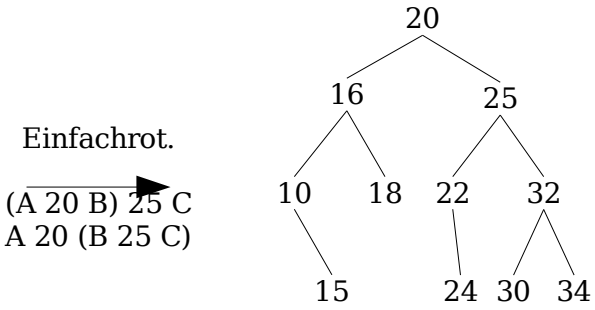
Entfernen + Rotation ändert die Höhe des UB h+2 → h+1 (evtl. weitere Rotationen weiter oben nötig).

Balanzierung ✓

Suchbaum ✓

<u>Beispiel:</u>





Empirische Untersuchungen: 1 Rotation pro 5 Löschungen im Mittel

AVL-Bäume garantieren Worst-Case $O(\log n)$ für Suchen, Einfügen und Entfernen (n = Anzahl Knoten).

6.7 B-Bäume

s. Datenbankanwendungen Prof. Hunsinger

7 Hash-Tabellen

7.1 Problemstellung, Definitionen

Aufgabe: Gegeben sei eine Menge von Schlüsseln $K=\{k_i\}, i=0,\ldots,L-1$. Die Menge aller potentiellen Schlüssel sei U ($K \subseteq U$, „Universum“). Möglichst effiziente Suche nach dem Schlüssel (dem Nutzinformationen zugeordnet sein können).

Grundidee:

- Speichere Schlüssel in Feld $F[0\ldots M-1]$, der Hash-Tabelle.
- Berechne aus gesuchtem Schlüssel k mit der Hash-Funktion h direkt den Index in F („Schlüsseltransformation“).
  $h: U \to \{0,\ldots,M-1\}$

dt. „Streuspeicherverfahren“

Grundlegende Operationen

Einfügen neuer Schlüssel k: $F[h(k)]=k$
Suchen eines Schlüssels k: $F[h(k)]$ inspizieren
Entfernen eines Schlüssels k: $F[h(k)]$ löschen

Beispiel: Sei $M=5, U=\{0,\ldots,99\}, K=\{3,15,22,24\}$
$h(k)=k \bmod 5$

F:

0	15
1	
2	22
3	3
4	24

Probleme:

1. Geeignete Wahl von M?
2. Geeignete Wahl von h?
3. Im Idealfall ist h, eingeschränkt auf K, injektiv. Sonst gibt es k, l mit $h(k)=h(l)$
   → Kollision. Wie behandelt man Kollisionen?

7.2 Wahl der Hashfunktion h

h soll

- schnell und einfach berechenbar sein
- die Schlüssel möglichst gleichmäßig über die Hashtabelle streuen

Bewährt: Divisionsmethode
Falls $U \subseteq \mathbb{Z}, k \in U : h(k) = k \bmod M$
Sonst: Finde Abbildung von U nach Z, und bilde danach Modulo-Rest

M am besten als Primzahl wählen.

7.3 Behandlung von Kollision

7.3.1 Offenes Hashing

Falls Kollision beim Einfügen, berechne Ersatzadresse in Hashtabelle F (Rehashing).

Nur geeignet, falls $L \leq M$.

Modifizierte Suche nach k: Berechne Ersatzadressen $h_i(k), i \geq 0$, solange bis
$F[h_i(k)] = k \rightarrow$ gefunden!
- oder -
$F[h_i(k)]$ unbesetzt bzw. alle Zellen durchsucht → nicht gefunden.

Definition:

Sei $\{h_i\}, i \geq 0$ eine Folge von Hashfunktionen, und $h_0(k), h_1(k), \ldots$ die durchsuchte Sequenz der Feldadressen.

a. $h_0(k) = h_0(l)$ für Schlüssel k, l heißt Primärkollision

b. $h_i(k) = h_j(l)$ für Schlüssel $k, l, i \neq j$ heißt Sekundärkollision

c. Kollisionen der Form $h_i(k) = h_i(l)$ für alle $i \geq 0$ heißen Häufung von Primärkollisionen

d. Kollisionen der Form $h_i(k) = h_{ji}(l_i)$ für alle $i \geq 0$ heißen Häufung von Sekundärkollisionen.

Modifiziertes Entfernen von k: Schlüsselwert darf nicht entfernt werden!
Würde Rehashfolge unterbrechen. Statt zu entfernen, wird der Tabellenplatz als belegt für die Suche und frei für das Einfügen markiert

0	15
1	22
2	Belegt für Suche frei für Einfügen
3	3
4	24

$h_0(57) = 2$
$h_1(57) = 1$

32 einfügen

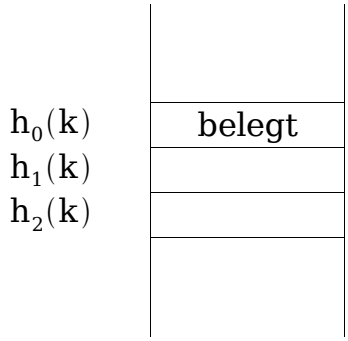
0	15
1	57
2	32
3	3
4	24

Wahl von $h_1, h_2, \ldots$

Lineare Sondierung:

$h_0(k)=h(k)$ (z.B. Div.methode)

$h_i(k)=(h_0(k)+i) \bmod M, i>0$



Vorteile:

- h_i ist für jedes $i>0$ einfach zu berechnen
- alle Adressen werden generiert

Nachteil:

Häufung von Primär- und Sekundärkollisionen, es bilden sich Cluster.

Quadratische Sondierung

$h_0(k)=h(k)$

$h_i(k)=(h_0(k)+i^2) \bmod M, i>0$

Vorteile:

- h_i schnell berechenbar
- Häufung von Sekundärkollisionen tritt nicht auf

Nachteile:

- Häufung von Primärkollisionen tritt auf
- $h_i(k)$ durchläuft i.A. nicht alle Tabellen-Adressen

Sondierung mit Double Hashing

$h_0(k)=h(k)$

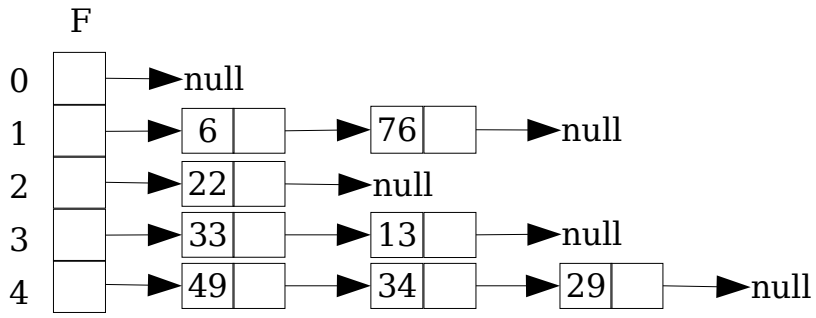
$h_0(k)=(h_0(k)+i\,h'(k)) \bmod M$

Hier ist h' eine von h verschiedene, geeignet gewählte Hashfunktion.
→ keine Häufung von Primärkollisionen mehr.

7.3.2 Separate Kollisionsklassen / Verkettung

$L \geq M$

Beispiel: $K=\{49,22,6,76,33,34,13,29\}, M=5, h(k)=k \bmod 5$



Vorteil: Kein Problem mit Tabellenüberlauf

Nachteil: extra Speicher, aufwändigere Algorithmen

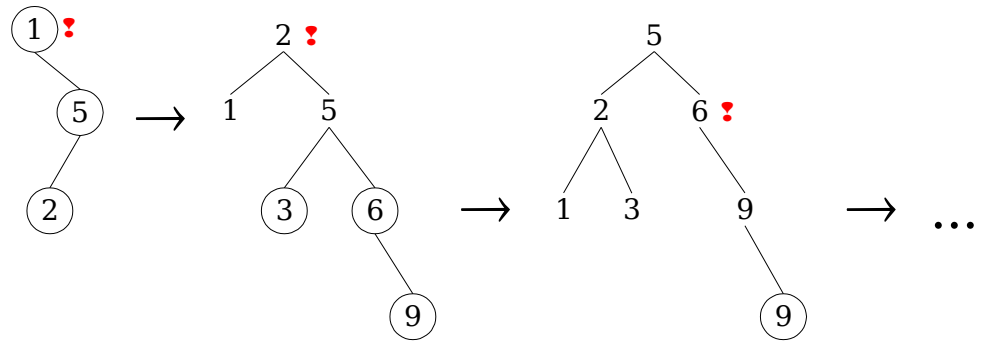
7.3.3 Komplexität

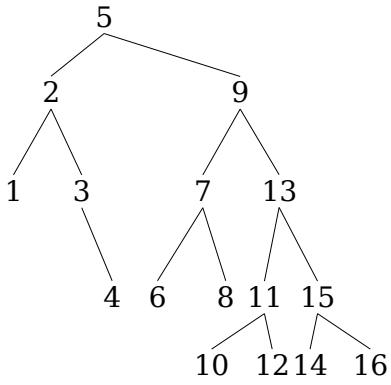
Bei geeigneter Wahl von M, h, und Kollisionsstrategie sehr schneller Zugriff!

Effizienz hängt vom Belegungsfaktor $\beta := \frac{L}{M}$ ab.

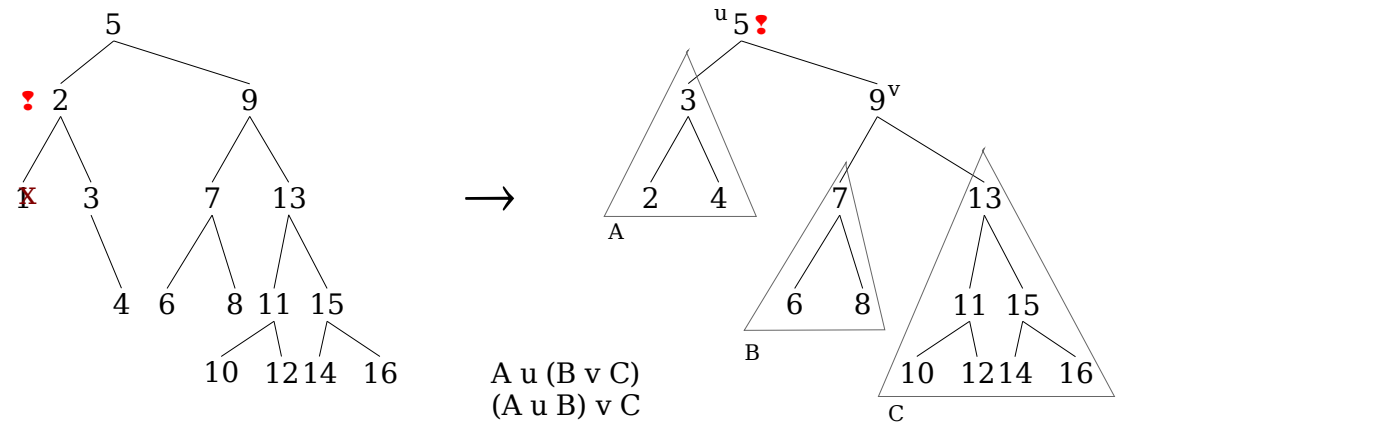
Aufgabe 6.9

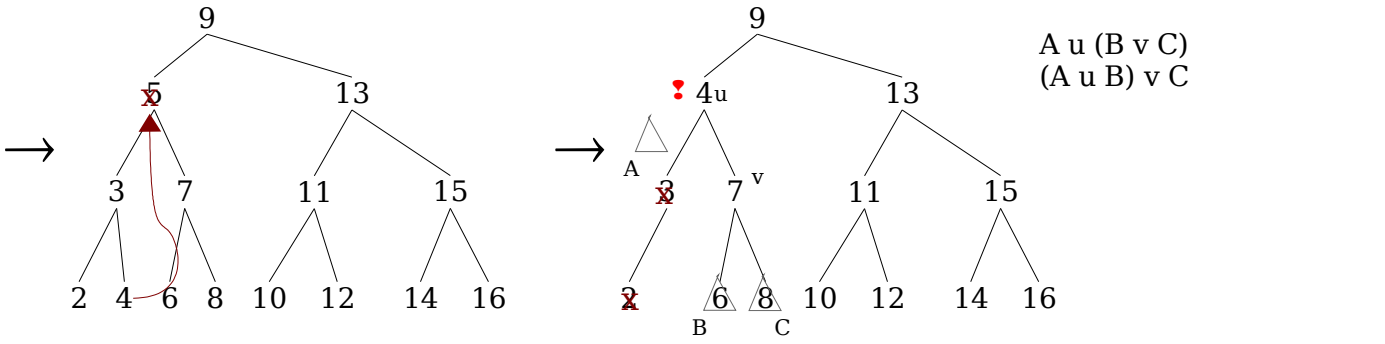
Einfügen:

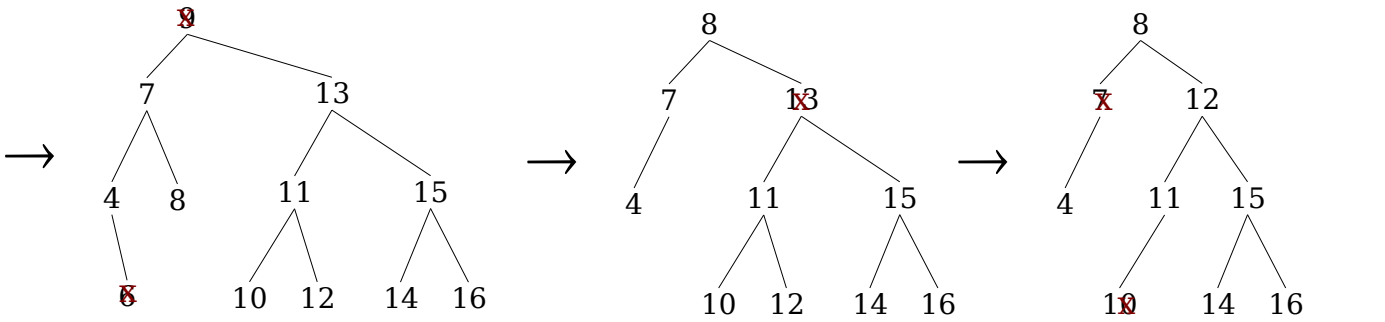


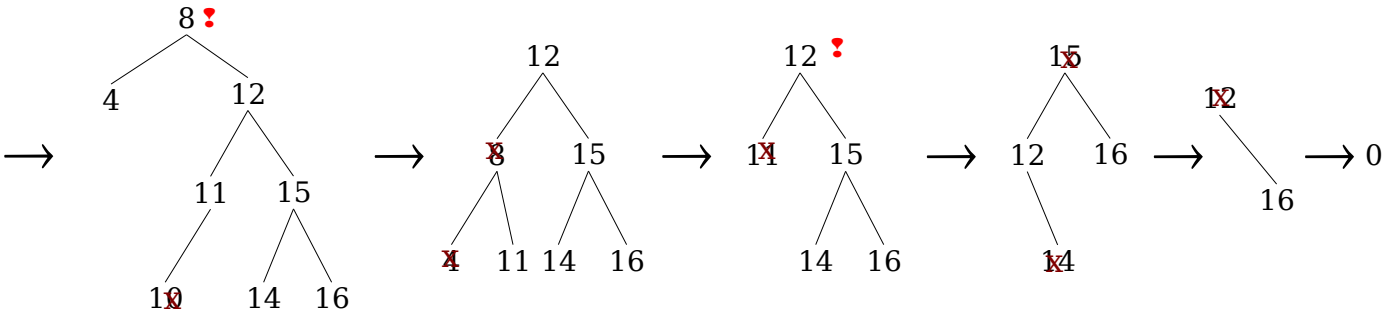


Entfernen


A u (B v C)
(A u B) v C


A u (B v C)
(A u B) v C

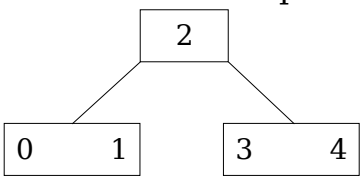




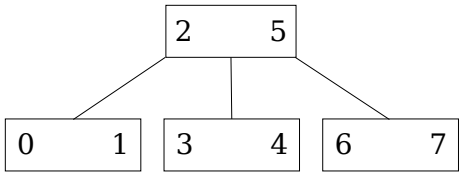
Aufgabe 6.12

Zunächst Befüllen der Wurzel:

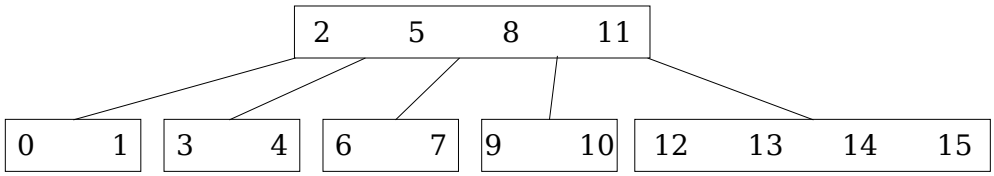
Einfügen der 4 führt zur Spaltung



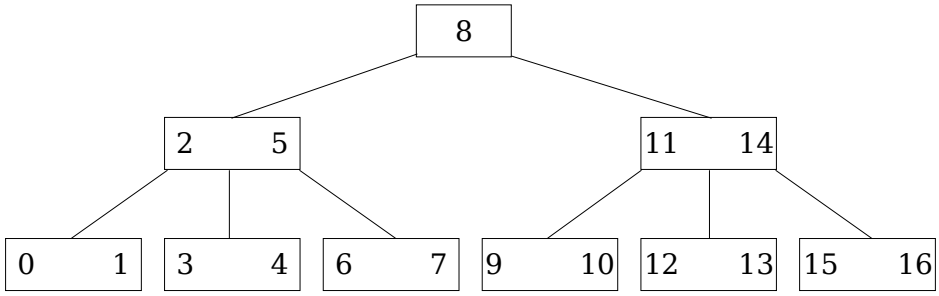
5, 6 können in rechten Blatt eingefügt werden. 7 führt zur Spaltung dieses Blattknotens



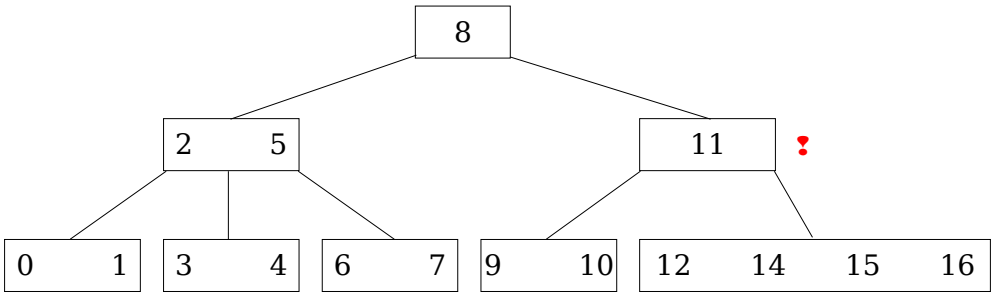
Dies geht so weiter, bis schließlich alle Zahlen bis einschließlich 15 eingefügt sind



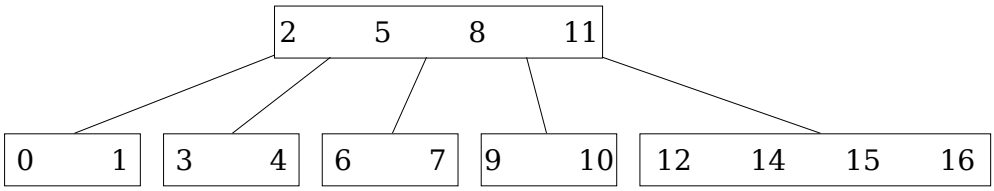
16 spaltet Blatt und führt zur Spaltung der Wurzel



c) Entfernen von 13. Verschmelzen der verbleibenden 12 mit einer der Nachbarseiten



Führt zu Unterlauf in der Seite darüber



Löschen von 16 trivial.

Aufgabe 7.2

Wert	**Adressberechnungen**	
25	25 mod 10 = 5	
17	17 mod 10 = 7	
29	29 mod 10 = 9	
21	21 mod 10 = 1	
11	11 mod 10 = 1	Kollision
	(1 + 1 (11 mod 7) mod 11 = 6	
30	30 mod 10 = 0	
40	40 mod 10 = 0	Kollision
	(0 + 1 (1 + 40 mod 7) mod 11 = 6	Kollision
	(0 + 2 (1 + 40 mod 7) mod 11 = 1	Kollision
	(0 + 3 (1 + 40 mod 7) mod 11 = 7	Kollision
	(0 + 4 (1 + 40 mod 7) mod 11 = 2	
22	22 mod 10 = 2	Kollision
	(2 + 1 (1 + 22 mod 7) mod 11 = 4	
31	31 mod 10 = 1	Kollision
	(1 + 1 (1 + 31 mod 7) mod 11 = 5	Kollision
	(1 + 2 (1 + 31 mod 7) mod 11 = 9	Kollision
	(1 + 3 (1 + 31 mod 7) mod 11 = 2	Kollision
	(1 + 4 (1 + 31 mod 7) mod 11 = 6	Kollision
	(1 + 5 (1 + 31 mod 7) mod 11 = 10	

		# Adressber.
0	30	1
1	21	1
2	40	5
3		
4	22	2
5	25	1
6	11	2
7	17	1
8		
9	29	1
10	31	6

b) Anzahl Adressberechnungen für eine erfolgreiche Suche im Schnitt:

$(1+1+5+2+1+2+1+1+6)/9=20/9\approx 2{,}2$

Seien $f_1(\beta)$ die Anzahl der Operationen, die man für Einfügen oder erfolglose Suche braucht. $f_2(\beta)$ sei Anzahl der Operationen für erfolgreiche Suche.

Für offenes Hashing gilt dann

$$\lim_{\beta\to 1} f_1(\beta)=c\cdot M$$
$$\lim_{\beta\to 1} f_2(\beta)=c\cdot M$$

Für manche Fälle ist eine analytische Darstellung bekannt.

Beispiel: Lineares Sondieren

$$f_1(\beta)=\frac{1}{2}\left(1+\frac{1}{(1-\beta)^2}\right)$$
$$f_2(\beta)=\frac{1}{2}\left(1+\frac{1}{1-\beta}\right)$$

Interpretation: Für $\beta\to 1$ gehen Ausdrücke formal gegen ∞. Kosten steigen schnell gegen Worst-Case $O(M)$.

Faustregel: $\beta\in[0.7,0.9]$ meist günstiger Kompromiß

Für Hashtabelle mit Verkettung kann Belegungsfaktor $\beta>1$ werden. Degeneriert im Worst-Case (alle Einträge in einer Liste) zu $O(L)$.

Anzahl der Adreßberechnungen ($\beta=0.9$)

Hash-Methoden	**Suche**	
	erfolgreich	erfolglos
Lineare Sondierung	5,5	50,5
Verkettung	1,45	1,31

⇒ Empfehlung: Verkettung

Anmerkung:

- virtuelle Speicher dann effektiv, wenn große Lokalität in Daten und Programmen
- Hashing versucht bestmöglich zu streuen, d.h. Lokalität zu zerstören → prinzipielle Unverträglichkeit

8 Sortieren

Web-Tip: http://sortieralgorithmen.de

8.1 Problemstellung

Gegeben

1. Sequenz von n Schlüsseln $a_0, a_1, \ldots, a_{n-1}$
2. Ordnungsrelation $\leq$ zwischen Schlüsseln

Aufgabe:
n Schlüssel durch Permutation π der Indizes in geordnete Reihenfolge bringen:

$$a_{\pi(0)} \leq a_{\pi(1)} \leq \ldots \leq a_{\pi(n-1)}$$

8.2 Internes Sortieren

n Elemente passen in Hauptspeicher und werden als Array dargestellt.

8.2.1 Internes Sortieren durch Vergleichen

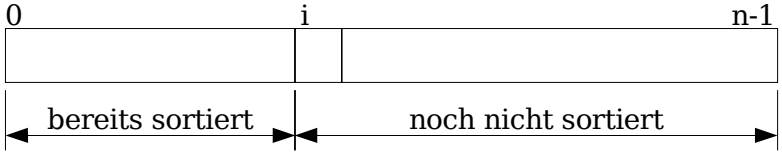
Basisoperationen:

- Vergleich zweier Schlüssel
- Austausch zweier Schlüssel

Es kann gezeigt werden, daß optimale asymptotische Komplexität für Anzahl Vergleiche und Austauschoperationen von der Ordnung $O(n \log n)$ ist.

Sortieren durch Einfügen (insertion sort)

Baue sortierten Anfangsteil auf



Füge Schlüssel an Position i an der richtigen Stelle im sortierten Teil ein. Damit wächst sortierter Teil um eins nach rechts.

$$\text{Worst Case} = \text{Avg Case} = O(n^2)$$

U.U. $O(n)$ für „fast sortierte Sequenzen“

Sortieren durch Auswählen (selection sort)

Wieder Aufteilung in sortierten und unsortierten Teil.

Füge nun das kleinste Element des unsortierten Teils am Ende des sortierten Teils ein.

$$\text{Worst Case} = \text{Avg Case} = O(n^2)$$

Meist etwas schneller als insertion sort (weniger Austauschoperationen).

Bubblesort

Vergleiche aufeinanderfolgende Paare und vertausche, wenn sie nicht in der richtigen Reihenfolge sind.

Es wird verglichen: a[0] mit a[1], a[1] mit a[2], ..., a[n-2] mit a[n-1]. Größtes Element steigt zum Array-Ende.
(Steigt auf wie Luftblase (bubble) im Wasser). $n-1$ Durchläufe, wobei sortierter Teil am Ende ausgenommen wird.

$$\text{Worst Case} = \text{Avg Case} = O(n^2)$$

Langsamster der drei einfachen Algorithmen.

Quicksort (1962 Haare)

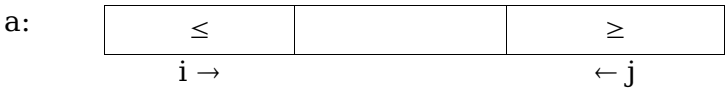
Sei a[0 ... n-1] zu sortierendes Feld

Grundideen:

- Vertauschung von Schlüsseln über größere Distanzen
- Divide & Conquer (Aufteilung in kleinere Teilprobleme)

1. Wähle (zufällig) einen Trennschlüssel (Pivot, Median) a[k] und teile a[0 ... n-1] in zwei Teilfelder auf:
   a. $a_{\leq} := a[0 \ldots i] : a[l] \leq a[k] \text{ für } 0 \leq l \leq i$
   b. $a_{\geq} := a[i+1 \ldots n-1] : a[l] \geq a[k] \text{ für } i+1 \leq l \leq n-1$
   „Partitionierung“, „Split“
2. Sortiere $a_{\leq}$ und $a_{\geq}$ ebenso (rekursiv)

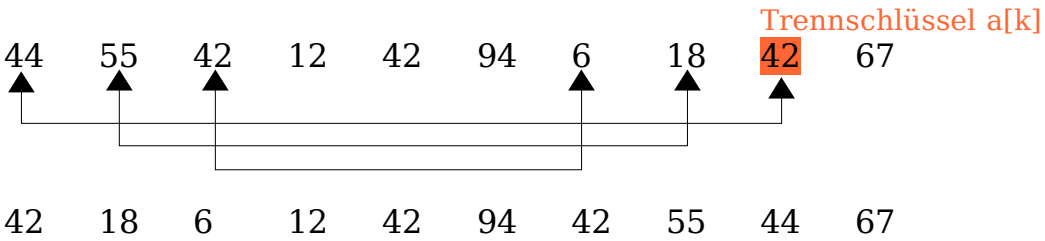
Skizze des split-Algorithmus:



Laß i, j laufen in angegebener Richtung:

- erst i bis $a[i] \geq a[k]$
- dann j bis $a[j] \leq a[k]$
- dann vertausche a[i] und a[j]
- stop, wenn j<i

Beispiel:



In Java:

```
/* a ist zu sortierendes (Teil-)Feld
   low ... untere Grenze
   high ... obere Grenze */

public void qsort (int a[], int low, int high) {
     if (low >= high)
         return;
     int b = split (a, low, high);          //Partitionierung
     qsort (a, low, b);
     qsort (a, b+1, high);
}

int split (int a[], int low, int high) {
     // Wähle mittleren Schlüssel als Trennschlüssel
     int ak = a[(low+high)/2];
     int i = low;
     int j = high;
     while (i < j) {
         while (a[i] < ak)
             i++;
         while (a[j] > ak)
             j++;
         if (i < j) {
             swap (a, i, j);            //ohne Implementation
             j++;                       //nötig z.B. für 1, 1
         }
     }
     return i;          //links von i ist a≤, rechts davon a≥
}
```

Komplexität: Worst Case $O(n^2)$ (Trennschlüssel wird in Sortierreihenfolge gewählt), Average Case $O(n \log n)$

Prüfungsaufgabenbuchtip: Bentley, Programming Pearls

Heapsort

siehe Abschnitt 6.4.2

immer $O(n \log n)$

Mergesort

wieder mit Divide-and-Conquer

Zerteile Folge a[0 ... n-1] in möglichst gleich große Teilfolgen

$a_1[0 \ldots \frac{n-1}{2}]$ und $a_2[\frac{n-1}{2}+1 \ldots n-1]$. Sortiere jede der Teilfolgen rekursiv durch Mergesort. Die sortierte Folge ergibt sich jetzt durch Verschmelzen der beiden sortierten Teilfolgen.

Beispiel: Verschmelzen der sortierten TF $a_1 = 1,2,3,5,9$ und $a_2 = 4,6,7,8,10$

a_1					a_2					Resultatfolge
1	2	3	5	9	4	6	7	8	10	-
↑					↑					1
	↑				↑					1 2
		↑			↑					...
			↑		↑					
			↑			↑				1 2 3 4 5
				↑		↑				
				↑			↑			...
				↑				↑		
				↑					↑	1 2 3 4 5 6 7 8 9 10

↑ ... zu vergleichende Schlüssel

Komplexität: Worst Case $O(n \log n)$ = Average Case

Wertung: Für kleines $n (\leq 20)$ einfaches Sortierverfahren (z.B. Selection Sort) verwenden.
Für große n Quicksort am schnellsten, aber auch riskant. Falls Risiko inakzeptabel, Mergesort / Heapsort verwenden.

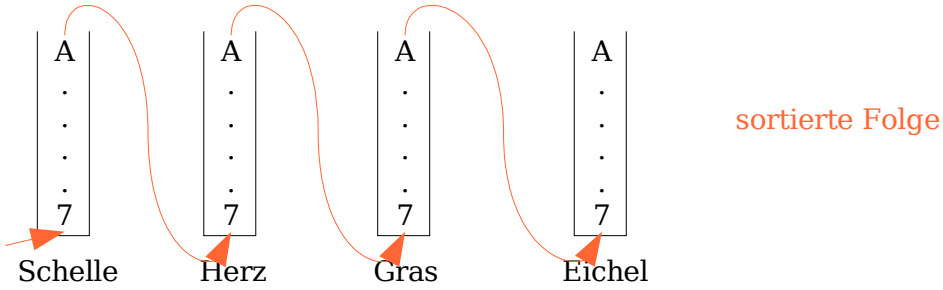
8.2.2 Internes Sortieren durch Verteilen

Methode: Ausnutzung der „Feinstruktur“ der zu sortierenden Schlüssel

Beispiel: Sortieren von Spielkarten
Karte = (Farbe, Wert)
Sortierfolgen:
Folgen: Schelle < Herz < Gras < Eichel
Werte: 7 < 8 < 9 < 10 < U < O < K

Sortierung durch Schubfach-Verteilung:

1. Bilde 8 Fächer $W_7, W_8, \ldots, W_A$
2. Werfe Karten in Fächer gemäß ihrem Wert
3. Bilde 4 Fächer F_S, F_H, F_G, F_E
4. Nimm nacheinander $W_7, W_8, \ldots W_A$ und verteile Inhalt gemäß Farbe auf F-Fächer



Anwendungen:

Suchen von Integern: n Zahlen im Bereich $[1 \ldots m]$. Bilde m Fächer und verteile Zahlen entsprechend ihrem Wert (bucket-sort).

Bit-Sort: Sortieren nach Zahlen; nutze als Feinstruktur die Bits; Voraussetzung

ist effizienter Zugriff auf einzelne Bits

Lexikographisches Sortieren: Gegeben Alphabet $\Sigma, |\Sigma| = m$. Sortiere n Strings konstanter Länge k in m Fächer, beginnend mit der am wenigsten signifikanten Stelle.
Beispiel: $\Sigma = \{a, b, c\}$, zu sortieren: $\{abc, bac, cab, bca\}$. Sortiere zunächst nach letzter Stelle in Fächer:

a-Fach	bca
b-Fach	cab
c-Fach	abc bac

Entnimm Fächerinhalte und sortiere anhand zweiter Stelle:

a-Fach	cab bac
b-Fach	abc
c-Fach	bca

Gleiches mit erster Stelle:

a-Fach	abc
b-Fach	bac bca
c-Fach	cab

Verteilungs-Sortieren ist unter der Annahme, daß Schubfachwahl kostenlos ist, von linearer Komplexität.

– Zugriff auf Schlüssel-Feinstruktur muß effizient möglich sein

– Fächeranzahl muß im Voraus bekannt sein

8.3 Externes Sortieren

Bisher: Zu sortierende Daten passen in Hauptspeicher. Zugriff auf ein Datum benötigt konstante Zeit.

Jetzt: Daten auf Sekundärspeicher (Festplatte, Magnetband, ...), passen nicht alle zugleich in Hauptspeicher. Zugriff auf Datum auf Sekundärspeicher 10^6 – 10^{10} mal langsamer.

Nutze Modifikationen von Mergesort.

Teile Datenbestand auf, daß einzelne einzelne Stücke in Hauptspeicher passen, sortiere sie dort (mit beliebigen Algorithmen). Schreibe sortierte Teilsequenzen auf Sekundärspeicher. Danach werden diese in mehreren Durchläufen unter Verwendung von Sekundärspeicher verschmolzen.

9 Suche in Texten

9.1 Textmustersuche

Gegeben sei Zeichenfolge $s := s_0 s_1 \ldots s_{n-1}$ (String) und Muster $p = p_0 p_1 \ldots p_{m-1}$ (en. Pattern) über einem Alphabet Σ.

Ein Vergleichsfenster w^i (en. Window) ist eine Teilzeichenfolge $s_i s_{i+1} \dots s_{i+m-1}$ der Länge m von s.

Gesucht: Erste Position $i, 0 \le i \le n-m$ in s, für die Vergleichsfenster mit Muster übereinstimmt: $w^i = p$.

Finden aller Vorkommnisse des Musters kann darauf zurückgeführt werden.

9.1.1 Anwendungen

– Texteditoren: Schlüsselwörter suchen, Variablennamen in Programmen

– Tools: z.B. grep unter UNIX

– Virenscanner: Auffinden von Virensignaturen in Dateien

– Biochemie: Suche von Basensequenzen in Genom

9.1.2 Direkte Suche

Methode: Vergleiche die Zeichen des Musters p_k mit den Zeichen w_k^i des Vergleichsfensters von links nach rechts, $k = 0, \dots, m-1$.
Sobald ein Zeichenpaar $w_k^i \neq p_k$ gefunden ist (Mismatch), wird Vergleichsfenster um eine Position nach rechts verschoben: $w^i \to w^{i+1}$
Falls alle Zeichen von p und w^i übereinstimmen: Vorkommnis an Position i

Beispiel:

String	a	b	a	c	a	b	a	b	c	
Muster	a	b	a	b						Mismatch
		a	b	a	b					Vorkommnis
			a	b	a	b				
				a	b	a	b			
					a	b	a	b		

Analyse: Algorithmus hat zwei geschachtelte Schleifen mit maximaler Lauflängen n bzw. m.
Worst-Case-Komplexitäten $O(n \cdot m)$ Vergleiche, z.B. für $s = a^n, p = a^{m-1}b$

a a a a a a a a a a
a a a b

Mismatch wird meist früher entdeckt (insbesondere für große Alphabete).

Durchschnittliche Anzahl von Vergleichen je Zeichen ist kleiner als

$$\frac{1}{1-h_{max}}.$$

h_j ist die Auftrittswahrscheinlichkeit des Zeichens p_j in s, und $h_{max} := \max_j h_j$.

Für deutschsprachige Text ist die durchschnittliche Anzahl der Vergleiche je Zeichen < 1,15, da „e“ eine Häufigkeit von 13% hat.

→ durchschnittliche Zeitkomplexität von direkter Suche $O(n)$.

Übung 8.1 A L G O R I T H M

a) Sortieren durch Einfügen (ohne Verschiebungen, um Platz für Einfügen zu schaffen)

A L G O R I T H M Grenze sortiert – unsortiert
A L G O R I T H M
A G L O R I T H M
A G L O R I T H M
A G L O R I T H M
A G I L O R T H M
A G I L O R T H M
A G H I L O R T M
A G H I L M O R T

b) Sortieren durch Auswählen

A L G O R I T H M Vertausche stets „Diagonalelement“ mit
A L G O R I T H M kleinstem Schlüssel des unsortierten Arrays
A G L O R I T H M
A G H O R I T L M
A G H I R O T L M
A G H I L O T R M
A G H I L M T R O
A G H I L M O R T
A G H I L M O R T

c) Bubblesort

A L G O R I T H M Vergleiche
A L G O R I T H M
A G L O R I T H M
A G L O R I T H M
A G L O R I T H M
A G L O I R T H M
A G L O I R T H M
A G L O I R H T M
A G L O I R H M T
A G L O I R H M T
A G L O I R H M T
A G L O I R H M T
A G L I O R H M T
A G L I O H R M T
A G L I O H M R T
A G I L O H M R T
A G I L H O M R T
etc.

9.1.3 Verbesserungsstrategien

1. Seltene Zeichen zuerst vergleichen. Z.B. im Deutschen Vergleiche auf x (0,2%) vor Vergleichen auf e (13%).
2. Bereits gewonnene Vergleichsinformation nutzen (KMP-A., s.u.)
3. Struktur des Musters bei Mismatch ausnutzen, um Vergleichsfenster weiter verschieben zu können (Sunday-A., s.u.)

Alternative Idee: Berechnung einer Signatur (eines Hashwertes) des Vergleichsfensters und Vergleich mit Signatur des Musters. Bei Gleichheit zeichenweiser Vergleich. Voraussetzung: Signatur von w^i läßt sich einfach aus Signatur von w^{i-1} berechnen (Rabin-Karp).

9.1.4 Knuth-Morris-Pratt-Algorithmus

Vergleich Vergleichsfenster – Muster liefert Mismatch „in der Mitte“ → Zeichen zuvor sind gleich!

Wie weit darf Vergleichsfenster unter Ausnutzung dieser Info verschoben werden, ohne ein Vorkommnis zu übersehen?

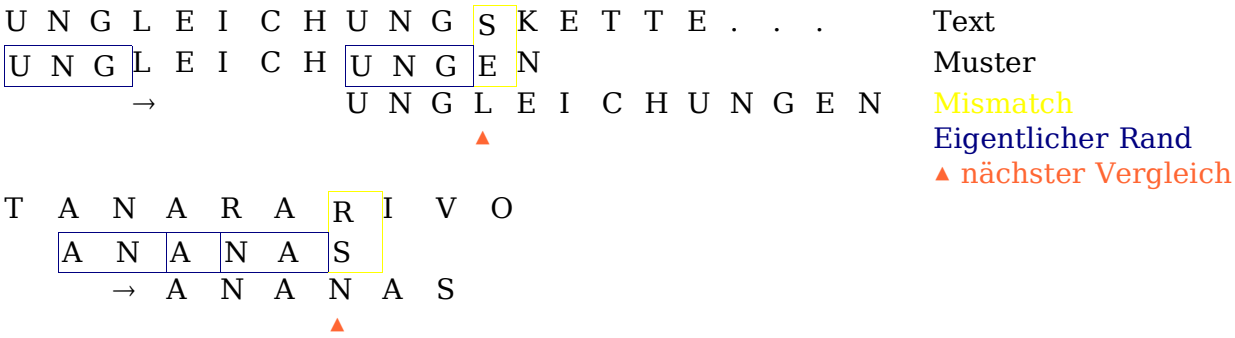
Definition:

Wort r heißt Rand von s, wenn es Präfix und Suffix ist. Der eigentliche Rand $r_e(s)$ ist der längste echte Rand von s. ■

Für das leere Wort ϵ definiert man $\underbrace{|r_e(\epsilon)|}_{\text{Länge eines Wortes}} = -1$.

Beispiel: ungleichung hat nicht-echte Ränder ϵ und ungleichung sowie den eigentlichen Rand ung.

Suche:



Idee von KMP: Bei Mismatch $w_k^i \neq p_k$ brauchen wir die Länge des eigentlichen Randes $r := |r_e(p_0 \ldots p_{k-1})|$:

- Vergleichsfenster von i nach $l := i + k - r$
- Nächster Vergleich: $w^l_{\max\{0,r\}}$ mit $p_{\max\{0,5\}}$

Zwei Phasen:

1. Setup: Berechne Shift-Tabelle: Enthält zu jedem Präfix von p Länge des eigentlichen Randes. Rekursiv in $O(m)$ Operationen möglich.
2. Eigentliche Suche:
   - „Erfolgreicher“ Vergleich: Vergleichsposition wandert 1 nach rechts.
   - „Erfolgloser“ Vergleich: Vergleichsfenster wandert um mindestens eine Position nach rechts.

→ nie mehr als $2n$ Vergleichsoperationen

- Gut für Streaming, da kein Rücksetzen nötig
- Gesamtkomplexität (mit Shift-Tabelle): $O(n+m)$

2. Algorithmus, der Musterstruktur ausnutzt: Boyer-Moore-Algorithmus

9.1.5 Sunday-Algorithmus

D. M. Sunday (1990), Quick Search Variante.

Methode

```
a a b a c b a b a a b a b      mismatch
a b a b                        Zeichen hinter Vergleichsfenster
      a b a b
        a b a b
          a b a b
```

Annahme: Vergleich $w^i - p$ liefert Mismatch

Betrachte Zeichen s_{i+m} hinter Vergleichsfenster. Kommt es im Muster nicht vor → Vergleichsfenster komplett hinter dieses Zeichen verschieben. Nächster Vergleich zwischen s_{i+m+1} und p_0.

Kommt s_{i+m} im Muster vor, so kann Vergleichsfenster nur um so viele Stellen verschoben werden, bis erstes Zeichen von rechts in p mit s_{i+m} übereinstimmt.

1. Berechnung der Shift-Tabelle. Ein Eintrag je Zeichen des Alphabets, der angibt, wie weit Vergleichsfenster im Falle eines Mismatches verschoben werden kann, wenn sich dieses Zeichen dahinter befindet.
2. Eigentliche Suche.

Analyse

- Worst Case $O(n \cdot m)$ (Text: b^n, Suchmuster: bbbbab)
- Mit Modifikationen: Worst Case $O(n)$
- Average Case $O(\frac{n}{m})$
- Natürlichsprachige Texte: ca. 10 mal schneller als direkte Suche
- besser für große Alphabete

Erstellung Shift-Tabelle:

1. Initialwert für alle Zeichen $m+1$

2. Durchlauf im Muster von links nach rechts
   - An Position i befinde sich Zeichen a
   - Setze für a den Shift auf $m-i$

9.2 Duplikatsuche

Finde in einem Text die längste Zeichenkette, die mindestens zweimal vorkommt.

Beispiel:
Nicht für die Schule, für das Leben lernen wir!

Anwendungen:

- Plagiate erkennen
- Code, der mit Copy & Paste erstellt wurde, identifizieren
- ...

Folgende Methode funktioniert effizient nur in Programmiersprachen, die ein String-Konzept ähnlich dem von C haben.

Wir nutzen ein Suffix-Array

Für Text der Länge $n \rightarrow n$ Zeiger; Zeiger i zeigt auf i-tes Zeichen.

Sei z.B.

```
char s[n+1]; /* Text + \0 */
char *p[n];  /* Suffix-Array */
```

Dann ist

```
p[i] = &s[i];
```

Jedes p[i] zeigt auf Suffix.

Beispiel: Text Banane
Suffix-Array:

```
p[0]  b  a  n  a  n  e
 ⋮    a  n  a  n  e
 ⋮    n  a  n  e
 ⋮    a  n  e
 ⋮    n  e
p[5]  e
```

In C können Einträge im Suffix-Array p als Strings interpretiert und sortiert werden (z.B. qsort).

Im Beispiel:

anane	längstes Duplikat
ane	
banane	
e	
nane	
ne	p in-place sortieren!

Durchlauf über sortiertes Array:

- Vergleich benachbarter Einträge auf übereinstimmende Präfixe
- Merke Position & Anzahl, wo Übereinstimmung am größten → längstes Duplikat

Komplexität: Worst Case $O(n^2 \log n)$

10 Verlustfreie Datenkompression

Kompression:
Verlustbehaftet: Komprimiert besser, nimmt Informationsverlust in Kauf.
Verlustfrei: Information kann vollständig wiederhergestellt werden (Texte, Grafiken geringer Farbtiefe)

Zu jedem Kompressionsverfahren und jeder Textlänge n gibt es mindestens einen Text, der sich nicht komprimieren läßt.
Betrachtung auf Bit-Ebene: n Bits → 2^n Texte
Mit $n-1$ Bits oder weniger lassen sich aber nur

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

verschiedene Binärwerte bilden → mindestens ein Text wird auf Text mit Länge $\geq n$ abgebildet.

10.1 Huffman-Kodierung

ASCII: immer 8 Bit / Zeichen

Mit variablen Codewort-Längen kann man ausnutzen, daß Zeichen unterschiedlich häufig sind. Häufige Zeichen → kurze Codewörter, seltene Zeichen → lange Codewörter.

Für eindeutige Dekodierung muß Präfix- oder Fano-Bedingung erfüllt sein: Kein Codewort ist Präfix eines anderen Codeworts.

Beispiel: Codes 0, 10, 110 erfüllen Präfix-Bedingung
10|0|10|110|0 läßt sich eindeutig zerlegen

Codes 0, 10, 101 erfüllen Präfix-Bedingung nicht
0||10|1|0||10 ... mehrere Zerlegungen möglich

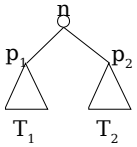
Huffman-Kodierung erfüllt die Präfix-Bedingung.

Huffman-Kodierung läßt sich als Binärbaum darstellen mit Zeichen des Alphabets in Blättern. Weg von Wurzel zum Blatt ergibt Codewort des Zeichens:
Verzweigung nach links → 0
Verzweigung nach rechts → 1

Baum minimiert durchschnittliche gewichtete Weglänge und wird von Blättern her konstruiert:

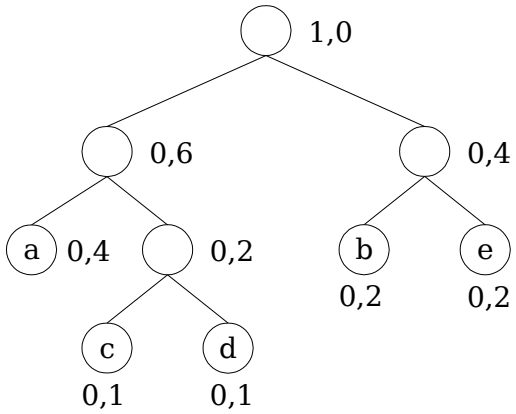
- Bilde für jedes Zeichen des Alphabets einelementigen Baum; versuche ihn mit Wahrscheinlichkeit des Zeichens. Menge der Bäume sei M.
- Solange M mehr als einen Baum enthält:

- Entnehme M die beiden Bäume T_1 und T_2 mit kleinsten Wahrscheinlichkeiten p_1 und p_2.
- Konstruiere Baum T mit neuem Wurzelknoten n und T_1 und T_2 als Unterbäumen
- Füge T zu M hinzu



Beispiel: Alphabet $\{a,b,c,d,e\}$

$p(a)=0,4$
$p(b)=0,2$
$p(c)=0,1$
$p(d)=0,1$
$p(e)=0,2$



Entstehende Kodierung:
a=00
b=10
c=010
d=011
e=11

Alternative Lösungen sind möglich.
Durchschnittliche gewichtete Weglänge:

$$\underbrace{0,4\cdot 2}_{p(a)\cdot\text{Länge Kodierung von a}} +0,2\cdot 2+0,1\cdot 3+0,1\cdot 3+0,2\cdot 2=2,2$$

im Mittel 2,2 Bit/Zeichen

Optimalität: Huffman-Kodierung ist Präfix-Code mit kürzester mittlerer Codewortlänge.

Anmerkung: Huffman-Kodierung kann auf Zeichenpaare, -tripel, usw. angewandt werden, wenn Häufigkeitsverteilung bekannt ist (Order-n) → höhere Kompressionsraten.

Theoretisches Limit für deutsche Texte (vermutet): 1,3 Bit/Zeichen

– Huffman-Kodierung ergibt sehr gute Kompressionsraten

– Häufigkeitsverteilung muß a-priori bekannt sein

Folgende Algorithmen sind Wörterbuch- (Dictionary-)Algorithmen ohne diesen Nachteil.

10.2 Lempel-Ziv-77 (LZ77)

Kodiert wiederkehrende Textstücke durch Referenzen im „Wörterbuch“.

Dazu wird Schiebefenster W über zu kodierenden Text geschoben. W besteht aus dem Such-Puffer S, der Suffix des bereits kodierten Textes ist, und dem Vorschau-Puffer L, der Präfix des noch zu kodierenden Textes ist. (Längen von S und L implementierungsabhängig.)

Suche nun längstes in S vorkommende Präfix p von L, und gib Tripel (o,l,x) aus mit

o : Offset von p in S
l : Länge von p

x : Zeichen, das den Mismatch ausgelöst hat

Falls keine Übereinstimmung, gib $(0,0,x)$ aus mit x dem ersten Zeichen von L.

Verschiebe nun W um $l+1$ (Hinter das Zeichen x.)

Beispiel:

bereits kodiert S	unkodiert L	Übereinstimmung	o	l	x
FISCHERS_FRITZ_FISCHT_FRISCHE_	FISCHE		0	1	T
FISCHERS_FRITZ_FISCHT_FRISCHE_	FISCHE		2	4	S
FISCHERS_FRITZ_FISCHT_FRISCHE_	FISCHE		7	2	E
FISCHERS_FRITZ_FISCHT_FRISCHE_	FISCHE		0	6	E

Anmerkungen:

- Beim Start hat Such-Puffer Länge 0, dann 1, ... bis zur vollen Länge.
- Durchsuchen von S mit schnellem Verfahren zur Textmustersuche.
- Dekodierung einfach: Beim Lesen ein Tripels hat man den Such-Puffer bereits dekodiert und kann das Tripel zu einem Textstück auflösen.

gzip, PKZip, Lharc, ARJ

10.3 Lempel-Ziv 78 (LZ78)

Verwendet explizites Wörterbuch. Tauchen Textstücke bei der Kodierung auf, die im Wörterbuch stehen, so gibt man statt dieser Referenzen auf Wörterbucheinträge aus.

Lese vom unkodierten Teil des Textes ein möglichst langes Präfix w, das im Wörterbuch D ist:

1. Präfixsuche: Starte mit leerem Wort $w=\epsilon$. Nächstes Zeichen des unkodierten Teils sei z.
   - Ist $w \cdot z \in D$, setze $w = w \cdot z$ und lesen nächstes Zeichen aus unkodiertem Bereich.
   - Andernfalls ($w \cdot z \notin D$) speichere $w \cdot z$ in D ab und gib Tupel $(i(w), z)$ aus, mit $i(w)$ dem Index des Wortes w in D.
     (Dabei gelte $i(\epsilon)=0$.)
2. Verschiebe Grenze zwischen kodiertem und unkodiertem Bereich um die Länge von $w \cdot z$, und beginne erneut mit Präfixsuche.

Beispiel:

```
                          ↓
FISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE...        Text
0000000 3 1 200 1 3 5 12 7  14 6  13 46        i(w)
FISCHER _ R TZ_ I C T F I  H _  S H            z
```

Entstehendes Wörterbuch: Ausgabe:0F0I0S0C...

1: F	2: I	3: S	4: C	5: H	6: E	7: R
8: S_	9: FR	10: IT	11: Z	12: _	13: FI	14: SC
15: HT	16: _F	17: RI	18: SCH	19: E_	20: FIS	21: CH

z.B. Ausgabe für Grenze kodiert / unkodiert hinter FISCHT:
„_“ ist in D enthalten, i(„_“)=12. „_F“ ist nicht enthalten
→ Ausgabe 12,F
Einfügen von „_F“ in D, i(„_F“)=16

Dekodierung einfach, da Wörterbuch einfach rekonstruiert werden kann.

10.4 Lempel-Ziv-Welch (LZW)

Zuerst wird Wörterbuch mit allen Zeichen des Alphabets initialisiert. Dann ähnlich wie LZ78 mit zwei Unterschieden:

1. Wir geben nur noch Referenzen auf das Wörterbuch aus.
2. Wir verschieben nach jeder Eintragung ins Wörterbuch nicht mehr um die Länge von $w \cdot z$, sondern nur noch um die Länge von w.

Beispiel:

FISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE	Text
24603156925478 18 11 137 18 20 12 14 24 301	Ausgabe

Initiales Wörterbuch:

0: C	5: R	10: FI	15: ER	20: RI	25: ISC	30: SCH
1: E	6: S	11: IS	16: RS	21: IT	26: CHT	31: HE_
2: F	7: T	12: SC	17: S_	22: TZ	27: T_	32: _FIS
3: H	8: Z	13: CH	18: _F	23: Z_	28: FR	33: SCHE
4: I	9: _	14: HE	19: FR	24: _FI	29: RIS	

Bei Kodierung entstehende Einträge:

Am Beginn der Dekodierung hat man nur initiales Wörterbuch. Damit soweit wie möglich dekodieren. Aus dekodiertem Teil kann man nach selbem Prinzip wie bei der Kodierung Rest des Wörterbuchs erschließen.

U.u. muß ein Wörterbucheintrag, von dem erst der Anfang bekannt ist, zur eigenen Rekonstruktion verwendet werden.

Beispiel:

Initiales Wörterbuch:

1: A	6: S
2: B	7: T
3: E	8: -
4: F	
5: N	

Komprimierter Text: 2 1 5 10 3 5 8 10 16 6 8 6 1 4 7

9: BA	10: AN	11: NA
12: ANE	13: EN	14: N-
15: -A	16: ANA	...

Lösung: BANANEN-ANANAS-SAFT

Anwendungen (LZW)

- GIF
- Datenübertragung per Modem

Wörterbuchbasierte Kompressionsalgorithmen

- Stark, wenn lange gleiche Zeichenketten oft vorkommen
- Schwach, wenn sich Eigenschaften des Eingabestroms häufig ändern

10.5 Burrows-Wheller-Transformation

Lange Texte werden in Blöcke (100KB – 900KB) aufgeteilt, einzeln transformiert und komprimiert.

Sei $t=t_0 \ldots t_{n-1}$ der Text. $(t \ll i)$ Sei der um i Positionen nach links rotierte Text. Die Matrix M habe $(t \ll i)$ als i-te Zeile. Zeile von M werden dann sortiert.

Beispiel: t = „BANANE“

M:						
	B	A	N	A	N	E
	A	N	A	N	E	B
	N	A	N	E	B	A
	A	N	E	B	A	N
	N	E	B	A	N	A
	E	B	A	N	A	N

						L
M sortiert:	A	N	A	N	E	B
≙ M'	A	N	E	B	A	N
$\alpha \rightarrow$	B	A	N	A	N	E
	E	B	A	N	A	N
	N	A	N	E	B	A
	N	E	B	A	N	A

Behauptung: t läßt sich rekonstruieren aus:

- der letzten Spalte L von M' (hier: BNENAA), und
- der Zeilennummer α, in der sich t in M' befindet (hier: $\alpha=2$)

Rekonstruktion:

- Aus L läßt sich Vertauschungsvektor π errechnen
  - Zuerst alle A's in L von oben nach unten durchnummerieren
  - dann alle B's, ... usw

L:	#n0		π
	0	B	2
	1	N	4
$\alpha \rightarrow$	2	E	3
	3	N	5
	4	A	0
	5	A	1

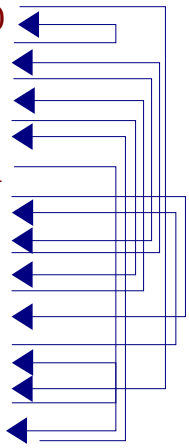
- „Dekodierung“ von hinten nach vorne (mit $\pi(i):=\pi_i$):
  - letztes Zeichen ist $L_\alpha=L_2=E$

- Vorletztes Zeichen ist $L_{\pi(\alpha)}=L_{\pi(2)}=L_3=N$
- Vorvorletztes Zeichen ist $L_{\pi(\pi(\alpha))}=L_{\pi(3)}=L_5=A$
- ...
- k-letztes Zeichen ist $L_{\pi^{(k)}(\alpha)}$

Beispiel:

Dekodiere: mit ('-' < 'a')

	#n0		π
L=	0	S	10
	1	-	0
	2	N	6
	3	N	7
$\alpha = 4 \rightarrow$	4	S	11
	5	O	8
	6	A	1
	7	A	2
	8	K	4
	9	K	5
	10	O	9
	11	A	3



KOKOS-ANANAS

Statt t komprimiere L!
L hat auch n Zeichen → wo liegt Vorteil?

Kontexteigenschaft: L enthält lange Sequenzen gleicher Zeichen. Z.B. landen durch Sortierung englischer Texte Zeilen, die mit „he“ beginnen, in M' nebeneinander.
Ein möglicher Ausschnitt von L wäre dann:
tttt_tt_sttt (the, _he, she)

L läßt sich gut mit Move-To-Front-Encoding (MTF) + Huffman komprimieren.

MTF: Alle Zeichen befinden sich in Liste. Für jedes Zeichen aus L geben wir dessen Position in der Liste aus, und verschieben es an Position 0.
→ Strom überwiegend kleiner Zahlen.

Anmerkung: Repräsentation von M durch Suffix-Array von tt (vgl. Duplikatsuche).
Keinesfalls Matrix explizit erstellen!

Anwendungen: bzip, bzip2, Mustersuche