Rheinisch-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik VI

Vorlesungsmitschrift zur Vorlesung im SS'94

Algorithmen und Datenstrukturen

Prof. Dr.-Ing. H. Ney

letzte Überarbeitung: 7. April 1998

Inhaltsverzeichnis

Literatur

[S'88] R. Sedgewick: Algorithms. 2nd ed., Addison-Wesley, 1988.

[S'93] R. Sedgewick: Algorithms in Modula-3. Addison-Wesley, 1993.

[G'92] R.H. Güting: Datenstrukturen und Algorithmen. Teubner, 1992.

[A'83] A.V. Aho, J.E. Hopcroft, J.D. Ullman: Data Structures and Algorithms. Addison-Wesley, 1983.

[A'74] A.V. Aho, J.E. Hopcroft, J.D. Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[M'88] K. Mehlhorn: Datenstrukturen und effiziente Algorithmen. Bde 1,2,3; (primär Bd 1: 'Sortieren und Suchen'), Teubner 1988. (Bde 2+3 vergriffen, Neuauflage ?)

[W'86] N. Wirth: Algorithmen und Datenstrukturen mit Modula-2. 4. Auflage, Teubner 1986.

[A'93] M. Aigner: Diskrete Mathematik. Vieweg Studium, 1993.

[C'93] T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to Algorithms. MIT press / McGraw Hill, 10th printing, 1993.

[S'91] B. Schinzel : Datenstrukturen und Algorithmen. Augustinus-Buchhandlung Aachen, 1991

[O'93] T. Ottmann, P. Widmayer : Algorithmen und Datenstrukturen. 2.Auflage, Wissenschaftsverlag, 1993

Hinweise:

[S'88, S'93]	: konkrete Programme
[G'92]	: schöne Darstellung des Suchens in Mengen
[A'83]	: als Einführung sehr schön
[A'74]	: sehr breit; deckt auch numerische Verfahren ab
[M'88]	: Motivation und Analyse der Algorithmen
[W'86]	: viel Modula-2

allgemein: DUDEN: a) Schülerduden INFORMATIK
b) INFORMATIK (fast identisch)

Englische Terminologie :

- Wird neben/statt der deutschen verwendet.
- Originalartikel sind (fast) immer in Englisch.
- Forschung in den USA dominiert.
- Es gibt oft keine einheitliche deutsche Bezeichnung.

1 Grundlagen

1.1 Einführung

1.1.1 Konkrete Problemstellungen

Programme und Komplexität (Zeit, Platz) für folgende Probleme:

1. gleichzeitige Bestimmung des Maximums und Minimums einer Menge
2. Sortieren:
   - sortiere eine Folge von reellen Zahlen
   - bestimme den Median (=50%-Quantil)
3. Suchen in Mengen (Datenbanken):
   (Beispiele aus [M'88, p.97])
   - Symboltabellen (Compiler): vordefinierte Wörter, Variablen, etc.
   - Autorenverzeichnis einer Bibliothek
   - Kontenverzeichnis einer Bank
   - allgemeine Datenbanken
4. Pfade in einem Graphen:
   - kürzester Pfad von A nach B
   - kürzester Pfad, der alle Knoten miteinander verbindet
     (minimum spanning tree; Minimalbaum, Minimalgerüst, MST)
   - kürzeste Rundreise, bei der alle Knoten genau einmal besucht werden
     (traveling salesman problem, TSP)
5. Suche nach „Mustern" im Text
6. numerische Aufgaben:
   - Inversion von Matrizen
   - FFT (Fast Fourier Transform) : Digitale Verarbeitung von Zeitreihen

1.1.2 Aktualität des Themas

- ökonomisch: möglichst effiziente Nutzung der Rechnerkapazität
- Forschung: bisher keine „gute Lösung" für das *traveling salesman problem* bekannt
- eigenes Forschungsgebiet (Erkennung gesprochener Sprache)
  - Mengen von Hypothesen
  - bester Pfad: Suche durch Hypothesen
  - *minimum spanning tree*: Ähnlichkeiten zwischen Wörtern

1.1.3 Ziele der Vorlesung

Was sollen Sie am Ende der Vorlesung beherrschen ?

- theoretisch:
  - was sind „effiziente" Verfahren (Datenstrukturen **und** Algorithmen) ?
  - Definition des Aufwandes (CPU-Zeit, Speicherplatz): → Komplexität
  - Analyse von Algorithmen
- Vorgehensweise:
  - Problemanalyse
  - Abbildung auf Datenstruktur und Algorithmus
  - lauffähiges Programm
  - Testen
- Standard-Verfahren der Informatik: Datenstrukturen und Algorithmen
- praktische Erfahrung in der Implementierung

1.1.4 Hinweis auf das didaktische Problem der Vorlesung

- Es ist schwierig klarzumachen, wie man von der Problemstellung zum fertigen Programm kommt. Im nachhinein ist es dann leicht, das *fertige* Programm zu erklären und nachzuvollziehen.
- Die meisten Bücher geben den fertigen Programmen mehr Raum als der Motivation und dem Weg zum fertigen Programm.
- Versuchen Sie daher selbst so oft wie möglich, selbständig die Herleitung und Implementierung der Standard-Algorithmen nachzuvollziehen.
- Verifizieren Sie insbesondere, daß Sie die *Details* wie Index-Grenzen usw. auch richtig hinbekommen. „Der Teufel steckt im Detail".
- Aus folgenden Gebieten der *Mathematik* werden immer wieder Anleihen gemacht:
  - Kombinatorik
  - Binomialkoeffizienten
  - Rekursionsgleichungen
  - Wahrscheinlichkeitsrechnung

1.1.5 Vorgehensweise

Es werden zwei Ebenen unterschieden [G'92, p.3]:

- algorithmische Ebene: Eine möglichst allgemeine Beschreibung der Objekte (= zu manipulierende Daten) und des Algorithmus hat den Vorteil, daß man sich auf das Wesentliche konzentriert.
- programmiersprachliche Ebene: konkrete Implementierung

Voraussetzung:

- Kenntnis von Pascal/Modula (oder anderen imperativen Programmiersprachen)
- Formulierung der Algorithmen:
  - teilweise: Pascal/Modula
  - Pseudo-Code nahe an Pascal/Modula
  - Pseudo-Code allgemein
- Programmieraufgaben in MODULA-3 auf den Rechnern des CIP-Pools

1.1.6 Datenstrukturen und Algorithmen

wie sie sich aus der Problemstellung ergeben:

- Sequenzen (Folgen, Listen)
- Mengen, speziell Dictionary (Lexikon)
- Bäume
- Graphen

1.2 Komplexität von Algorithmen

Die Laufzeit eines Programmes hängt von folgenden Komponenten ab :

1. Eingabedaten
2. Qualität des Compilers
3. Rechner-Hardware
4. Algorithmus
5. Betriebssystem

Wegen der Abhängigkeit von den Eingabedaten schreibt man $T(n)$ für die Laufzeit, wobei n bedeuten kann:

(a) Sortieren: Anzahl der Werte $a_1, ..., a_n$

(b) Finde alle Primzahlen $\leq n$

Beispiel: SelectionSort

„Sortieren durch Auswählen", eines der einfachsten Sortierverfahren, besteht aus zwei Schleifen:

```
for i=1,..,n-1 do begin
    min := i;
    for j=i+1,..,n do
        if A[j] < A[min] then min := j;
    swap(A[i],A[min])
    end;
```

- Vergleiche (compares) :

$$C = \sum_{i=1}^{n-1}(n-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

- Vertauschungen (swaps, exchanges): $E \leq n-1$

Wenn $n \gg \frac{E}{C}$ so gilt: $T(n) \cong c \cdot \frac{n^2}{2}$

$\Rightarrow$ asymptotische Aussagen

1.2.1 Komplexitätsklassen

Typische Zeitkomplexitätsklassen:

Formel	Name	Beispiel
1	konstant	„elementarer" Befehl; Speicherzugriff
n	linear	lineare Suche; Minimum einer Folge
$\log n$	logarithmisch	Binärsuche
$\log(\log n)$	doppelt logarithmisch	quadrat. Binärsuche Interpolationssuche
$n \log n$	überlinear	„Divide and Conquer" effiziente Sortierverfahren (QuickSort, HeapSort) FFT (Fast Fourier Transform)
n^2	quadratisch	einfache Sortierverfahren
n^3	kubisch	CYK-Parsing; Matrizen-Inversion
n^k	polynomiell (vom Grad k)	lineare Programmierung
2^n	exponentiell	Backtracking (exhaustive search)
$n!$	„wie Fakultät"	Zahl der Permutationen (traveling salesman problem)
n^n		

Logarithmus: allgemeine Basis (oder e=2.718...)
Basis ist hier in der Regel irrelevant.
ln: natürlich
ld: dual
lg: dekadisch

Zeitverbrauch hypothetischer Algorithmen zur Lösung desselben Problems

$T(n)$	$n=10$	$n=20$	$n=50$	$n=100$	$n=300$
n^2	0.0001 sec	0.0004 sec	0.0025 sec	0.01 sec	0.09 sec
n^5	0.1 sec	3.2 sec	5.2 min	2.8 Std	28.1 Tage
2^n	0.001 sec	1 sec	35.7 Jahre	400 Billionen Jahrhunderte	eine 75-stellige Zahl Jahrhunderte
n^n	2.8 Std	3.3 Billionen Jahre	eine 70-stellige Zahl Jahrhunderte	eine 185-stellige Zahl Jahrhunderte	eine 728-stellige Zahl Jahrhunderte

Der Urknall war vergleichsweise vor etwa 15 Milliarden Jahren.

1.2.2 O-Notation (Ω, Θ) für asymptotische Aussagen

Definition 1.1 Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion

$$\begin{aligned}
O(f) &:= \{\, g : \mathbb{N} \to \mathbb{R}^+ : \exists\, c > 0 \;\exists\, n_0 > 0 \;\forall\, n \geq n_0 : \; g(n) \leq c \cdot f(n)\} \\
\Omega(f) &:= \{\, g : \mathbb{N} \to \mathbb{R}^+ : \exists\, c > 0 \;\exists\, n_0 > 0 \;\forall\, n \geq n_0 : \; g(n) \geq c \cdot f(n)\} \\
\Theta(f) &:= \{\, g : \mathbb{N} \to \mathbb{R}^+ : \exists\, c > 0 \;\exists\, n_0 > 0 \;\forall\, n \geq n_0 : \; \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\}
\end{aligned}$$

Bezeichnung:
$O(f)$: obere Schranke
$\Omega(f)$: untere Schranke
$\Theta(f)$: Wachstumsrate der Funktion f

O, Ω, Θ beschreiben das asymptotische Verhalten. Meist wird $O(f)$ verwendet.

- statt $O(f)$ schreibt man oft $O(f(n))$.
- statt $g \in O(f)$ schreibt man: $g = O(f)$ oder $g(n) = O(f(n))$.
- Sprechweise: „f wächst höchstens so schnell wie g"
- Idee: „vergröbernde" Betrachtung von Funktionen, in der konstante Faktoren irrelevant sind.
- Ziel: möglichst enge Schranke
- $f \in \Theta(g)$: „f wächst genau so stark wie g"
- Hilfe: Falls $\lim_{n\to\infty} \dfrac{f(n)}{g(n)}$ existiert, gilt $f \in O(g)$ [G'92]

Beispiele:

(a) $T(n) = 7n + 3$
dann gilt: $T(n) \in O(n)$
verifiziere: $7n + 3 \leq c \cdot n \; \forall n \geq n_0$ durch Wahl $c = 8$ und $n_0 = 3$

(b) $f(n) = 5\, n^2 + 100 \log n$
$g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{5n^2 + 100 \log n}{n^2} = 5$$

also $f \in O(g)$

Allgemeine Aussagen:

(a) $T(n) = \alpha f(n) + \beta$ mit $\alpha,\ \beta \in \mathbb{R}$ und $f(n) \in \Omega(1)$
Behauptung : $T(n) \in O(f(n))$
d.h. gibt es ein $c > 0$, $n_0 > 0$, so daß

$$\begin{aligned} T(n) &\leq c \cdot f(n) \;\; \forall\, n \geq n_0 \\ \alpha f(n) + \beta &\leq c \cdot f(n) \;\; \forall\, n \geq n_0 \end{aligned}$$

Da für $\lim_{n\to\infty} f(n) = 0$ die Ungleichung nicht erfüllt werden kann, fordert man:
$f(n) \in \Omega(1)$, d.h. $\exists c' > 0 \quad \exists n' > 0 \quad f(n) \geq c' \cdot 1 \; \forall\, n \geq n'$

Lösung: Wähle $c = \alpha + \dfrac{\beta}{c'}$ und $n_0 = n'$.

(b) $T(n) = 3^n$
dann gilt: $T(n) \notin O(2^n)$, weil es kein Paar $c > 0$, $n_0 > 0$ gibt,
so daß:

$$3^n \leq c \cdot 2^n \;\; \forall\, n \geq n_0$$

(c) „Addition":

$$T_1(n) + T_2(n) \in O(T_{MAX}(n)) = \begin{cases} O(T_2(n)) & \text{falls } T_1 \in O(T_2) \\ O(T_1(n)) & \text{falls } T_2 \in O(T_1) \end{cases}$$

mit $T_{MAX}(n) := max\,\{T_1(n),\, T_2(n)\}$

konkretes Beispiel: $T_1(n) \in O(n^2)$
$T_2(n) \in O(n^3)$
$T_3(n) \in O(n^2 \log n)$
dann gilt: $T_1(n) + T_2(n) + T_3(n) \in O(n^3)$

(d) „Multiplikation":

Programmfragment : Schleife mit $O(g(n))$ Wiederholungen mit Zeit $T(n)$
Kern: $O(f(n))$
dann gilt: $T(n) \in O(f(n) \cdot g(n))$

1.2.3 Regeln für die Programmanalyse

1. Anweisung (elementar): O(1)

2. Folge von Anweisungen : nehme die Anweisung mit der größten Laufzeit

3. **if** (A) **then** B
   else C

$$T_{TOT} = \begin{cases} T_A + T_B \ (+T_{JMP}) & \text{falls } A \ TRUE \\ T_A + T_C & \text{falls } A \ FALSE \end{cases}$$

   Assembler-Realisierung: (siehe RAM)

```
        TEST A
        JUMP on Accu else
        B
        JUMP endif
else:   C
endif:  ...
```

   Solange A, B, C elementare Befehle sind, gilt $T_P \in$ O(1).
   Wenn A, B, C Prozeduren sind, ist es komplizierter.

4. Loop: (Kern + Terminierungsbedingung) · Durchläufe

Komplexität

In der überwiegenden Mehrzahl der Fälle können nur solche Algorithmen eingesetzt werden, die polynomielle oder geringere Komplexität haben. Die exponentiellen Algorithmen führen zu „beliebig" langen Laufzeiten.

Daher hat man folgendes Ziel:

- Algorithmen sollten möglichst polynomielle (oder geringere) Komplexität besitzen.
- In der *Komplexitätstheorie* versucht man, strenge und möglichst allgemeine Aussagen zu machen über:
  - Zeit- und Platz-Komplexität
  - untere und obere Schranken (möglichst enge !)

Beispiel: Fibonacci-Zahlen

Berechnung der Fibonacci-Zahlen durch Programm:

- direkte Rekursion: Laufzeit $T(n) = f(n)$: sehr ineffizient
- Iteration: $T(n) = O(n)$

$$f(n) := \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ f(n-1) + f(n-2) & , n > 1 \end{cases}$$

Fibonacci-Zahlen: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, ...

Wie groß ist das Wachstum ? Wie groß sind untere und obere Schranke ?

Behauptung:

1. $f(n) = \Omega(2^{\frac{n}{2}})$
2. $f(n) = O(2^n)$

Der Beweis beruht auf den folgenden Eigenschaften der Funktion $f(n)$ für $n > n_0$:

i)
$$f(n) > 0 \text{ , für } n > 0$$

ii)
$$f(n) < f(n+1) \text{ , für } n > 2$$

iii) für $n > 3$
$$\begin{aligned} f(n) &< 2 \cdot f(n-1) \\ &< 4 \cdot f(n-2) \\ &< ... \\ &< 2^{n-1} \cdot f(1) \end{aligned}$$

iv) für $n > 3$
$$\begin{aligned} f(n) &> 2 \cdot f(n-2) \\ &> 4 \cdot f(n-4) \\ &> ... \\ &> \begin{cases} 2^{\frac{n}{2}-1} \cdot f(2) & \text{, für } n \text{ gerade} \\ 2^{\frac{n-1}{2}} \cdot f(1) & \text{, für } n \text{ ungerade} \end{cases} \end{aligned}$$

Übungsaufgabe: Versuchen Sie, ebenfalls durch elementare Abschätzungen engere Schranken anzugeben!

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2) \\
&< f(n-3) + 2f(n-2) \\
&< 3 \cdot f(n-2) \\
&< 9 \cdot f(n-4) \\
&< \ldots \\
&< \begin{cases} 3^{\frac{n-1}{2}} \cdot f(1) & \text{, für } n \text{ ungerade} \\ 3^{\frac{n}{2}-1} \cdot f(2) & \text{, für } n \text{ gerade} \end{cases} \\
\frac{f(n)}{f(n-1)} &= 1 + \frac{f(n-2)}{f(n-1)} \\
&= 1 + \frac{f(n-2)}{f(n-2) + f(n-3)} \\
&= 1 + \frac{1}{1 + \frac{f(n-3)}{f(n-2)}} \\
&= 1 + \frac{1}{1 + \frac{f(n-3)}{f(n-3) + f(n-4)}} \\
&= 1 + \frac{1}{1 + \frac{1}{1 + \frac{f(n-4)}{f(n-3)}}} \\
\frac{1}{2} &< \frac{f(n-4)}{f(n-3)} < 1 \\
\frac{8}{5} &< \frac{f(n)}{f(n-1)} < \frac{5}{3}
\end{aligned}
$$

Beispiel: Zwei Algorithmen zur Berechnung der Fibonacci-Zahlen

Naive intuitive rekursive Lösung:

```
procedure fib (n : cardinal) : cardinal;
begin
    if n = 0 then return 0
    elsif n = 1 then return 1
    else return fib (n - 1) + fib (n - 2)
    end
end fib;
```

$T(n) \in \Omega(2^{\frac{n}{2}})$

Iterative Lösung:

```
procedure fib (n : cardinal) : cardinal;
var x, y, d, i : cardinal;
begin
    if n = 0 then return 0 end;
    i := 1; x := 1; y := 0;
    while i < n do
        d := x; x := x + y; y := d; i := i + 1;
    end;
    return x
end fib;
```

$T(n)$ ist hier $O(n)$.

Laufzeiten bei $n = 100$:

1. Iterativ: $c \cdot 100$ Einheiten
2. Rekursiv: $c \cdot 2^{50}$ Einheiten ($> c \cdot 10^{15}$)

D.h. wenn die iterative Lösung eine Millisekunde benötigt, braucht die rekursive Lösung ca. 30000 Jahre (plus minus ein paar tausend Jahre).

Beispiel: Fakultät

$$n! = \begin{cases} 1 & n = 0, 1 \\ n \cdot (n-1)! & n > 1 \end{cases}$$

Sei $T(n)$ die Laufzeit des entsprechenden rekursiven Programms:

$$T(n) = \begin{cases} a & n = 0, 1 \\ b + T(n-1) & n > 1 \end{cases} \qquad \text{mit geeigneten } a, b > 0.$$

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= 2 \cdot b + T(n-2) \\ &= \dots \\ &= (n-1)b + T(1) \\ &= (n-1)b + a \end{aligned}$$

$T(n)$ ist linear und hat somit „gutartiges" Verhalten.

1.3 Datenstrukturen

1.3.1 Datentypen

[G'92 p.36]

- elementare (atomare) Typen:
  - integer
  - cardinal
  - character
  - real
  - boolean

  zusätzlich in PASCAL und MODULA-3:
  - Aufzählungstypen
  - Unterbereichstypen
- Typkonstruktoren:
  - array
  - record (Datensatz, Verbund,...; in C: structure)

  Beachte: in beiden Fällen ist ein direkter Speicherzugriff möglich.

  zusätzlich in PASCAL:
  - set: funktioniert nur bei kleinen Mengen (beim MIPS-Pascal-Compiler unter dem Betriebssystem IRIX auf einem Silicon Graphics-Rechner bis zu 512 Elemente).
- Zeigertypen (Pointer)
  - erlauben dynamische Datenstrukturen, d.h. Speicher-Allokation auf Anforderung durch das Betriebssystem.
  - de facto: memory address
  - wichtigste Anwendung: linked list (verkettete Liste) mittels ,,rekursiver Definition" über einen Zeiger-Verbund
  - **Warnung:** Pointer erfordern erhöhte Disziplin beim Programmieren.

1.3.2 Konstruierte Datentypen

Mit Hilfe der Konstruktoren „Array” und „Record-Pointer” werden weitere Datentypen definiert:

- list: Liste, Sequenz, Folge
- stack: LIFO = Last-In First-Out Memory; Keller
- queue: FIFO = First-In First-Out Memory; Schlange
- tree: Baum
- graph: Graph
- network: Netzwerk; beschreibt paarweise Beziehungen zwischen Elementen (=Knoten).

1.3.3 Listen

list:	linked list, linear list; verkettete Liste, Liste, Sequenz, Folge
mathematisch :	Folge von Elementen $a(1), ..., a(i)$, $a(i+1), ..., a(n)$, wobei man i mit der Position identifiziert und die folgenden Operationen erlaubt sind:

- einfügen (insert) an beliebiger Position
- entfernen (delete) an beliebiger Position
- zugreifen (=„lesen”) auf beliebige Position

Varianten:

- zyklische Liste
- doppelt verkettete Liste
- ...
- beliebig verkettet: „Zeigergeflecht”

Vergleich der Implementierungen

Typischerweise gibt es zwei Varianten der Realisierung:

- Array
- Zeiger-Verbund (Pointer)

1. Array-Implementierung (sequentiell)

- zu starr: Delete und Insert sehr umständlich
- nur bei „statischer" Objektmenge zu empfehlen

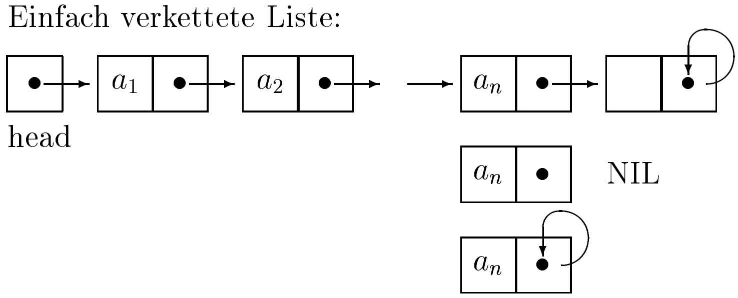
Array-Implementierung (verkettet über zusätzlichen Index-Array: =„Cursor-Darstellung")

- vorgegebene maximale Array-Größe (-)
- Freispeicherverwaltung erforderlich (-)
- kein Overhead (+)
- Fehlerkontrolle leichter (+)

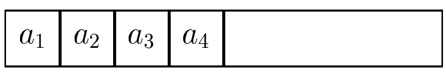
2. Pointer-Implementierung:
   - maximale Anzahl offen (+)
   - keine Freispeicherverwaltung (+)
   - Overhead durch Systemaufrufe (-)
   - Fehlerkontrolle schwierig (-)

Darstellungen einer Liste:

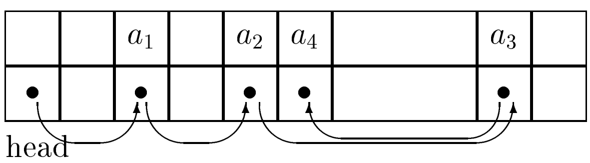
1. Einfach verkettete Liste:



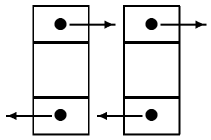
2. Array:
   (a) sequentiell (sehr starr)



   (b) Cursor-Darstellung: Zweiter Array für Indizes

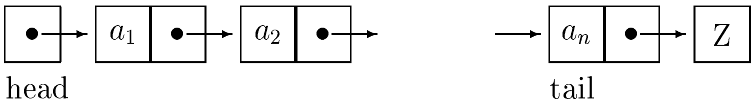


3. Doppelt verkettete Liste:

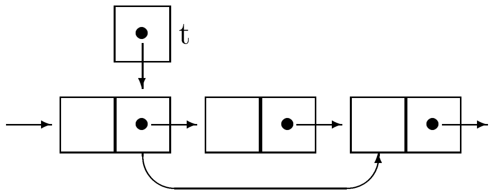


Beachte: Die Cursor- und Pointer-Implementierung sind weitgehend symmetrisch.

Implementation einer Liste:

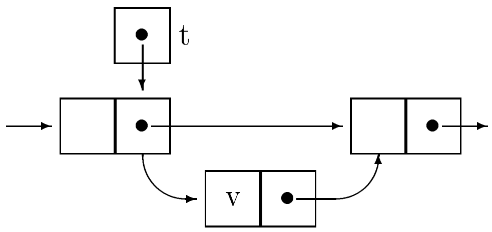


DeleteNext: „Nachfolger von t↑" löschen



t↑.next := t↑.next↑.next

InsertAfter: „Nachfolger von t ↑" einfügen



new(x);	(neue Speicherzelle für v)
x↑.key := v;	
x↑.next := t↑.next;	(justiere Pointer)
t↑.next := x;	

Programme [S'88]

1. Pointerimplementation:

```
type link = ↑node;
     node = record
               key: integer;
               next: link
               end;
var  head, z: link;
```

```
procedure ListInitialize;
begin
    new(head); new(z);
    head↑.next := z; z↑.next := z
end;

procedure DeleteNext(t:link);
var  x:link;
begin
    x := t↑.next;
    t↑.next:=t↑.next↑.next;
    dispose(x) (* Speicherplatz freigeben *)
end;

procedure InsertAfter (v: integer; t: link);
var x: link;
begin
    new(x); x↑.key := v; x↑.next := t↑.next;
    t↑.next := x
end;
```

2. Array-Implementation:

```
var     key, next : array [0..N] of integer;
        x, head, z : integer;

procedure ListInitialize;
begin
        head := 0; z := 1; x := 1;
        next[head] := z; next[z] := z
end;

procedure DeleteNext (t : integer);
begin
        next[t] := next[next[t]]
end;

procedure InsertAfter (v: integer; t: integer);
begin
        x := x + 1;
        key[x] := v; next[x] := next[t];
        next[t] := x
end;
```

Beachte:

- Overflow und Underflow der Arrays/Listen müssen abgefangen werden.
- Mit Hilfe einer Freispeicherliste können bereits benutzte und wieder freigegebene Feldpositionen wiederverwendet werden.

1.3.4 Stack und Queue

- stack = LIFO: Last-In First-Out Memory; pushdown stack; Keller, Stapelkeller
  mathematisch: Liste mit eingeschränkten Operationen:
  - einfügen nur am Ende (push)
  - entfernen nur am Ende (pop)
  - zugreifen = entfernen

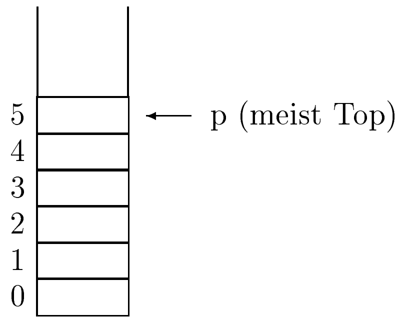


Bild: stack

Programme [S'88]

1. Implementation mit Pointern:

```
type link = ↑node;
     node = record
              key: integer;
              next : link
            end;
var  head, z : link;

procedure stackinit;
begin
     new(head); new(z);
     head↑.next := z; z↑.next := z
end;

procedure push (v : integer);
var t : link;
begin
     new(t); t↑.key := v; t↑.next := head↑.next;
     head↑.next := t
end;

function pop : integer;
var t : link;
begin
     t := head↑.next; pop := t↑.key;
     head↑.next := t↑.next; dispose (t)
end;
```

```
function stackempty : boolean;
begin stackempty := (head↑.next = z) end;
```

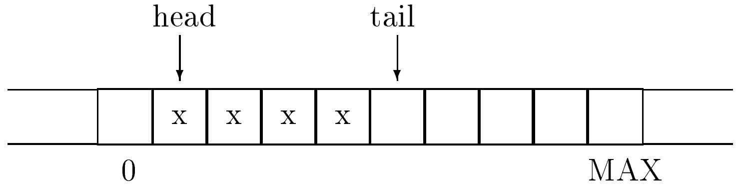
2. Implementation mit Array

```
const maxP = 100;
var   stack: array[0..maxP] of integer;
      p : integer;

procedure push (v : integer);
begin stack[p] := v; p := p + 1 end;

function pop : integer;
begin p := p - 1; pop := stack[p] end;

procedure stackinit;
begin p := 0 end;

function stackempty: boolean;
begin stackempty := (p <= 0) end;

function stackfull: boolean;
begin stackfull := (p > maxP) end;
```

Beachte: Overflow und Underflow müssen abgefangen werden.

- queue = FIFO: First-In First-Out Memory; Schlange, Warteschlange, Ringspeicher
  mathematisch: Liste mit eingeschränkten Operationen
  - einfügen nur am Ende (put)
  - entfernen nur am Anfang (get)
  - zugreifen = entfernen

Bild: Array-Realisierung



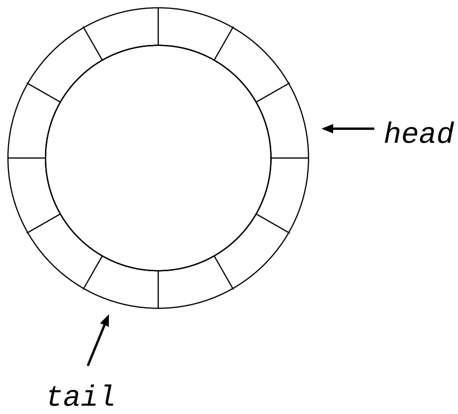


Abbildung 1: Ringpuffer

Programme [S'88]: Implementation mit Array:

```
const MAX = 100;
var   queue : array[0..MAX] of integer;
      head, tail : integer;

procedure put (v : integer);
begin
      queue[tail] := v; tail := tail + 1;
      if tail > max then tail := 0;
end;

function get : integer;
begin
      get := queue[head]; head := head + 1;
      if head > max then head := 0
end;

procedure queueinitialize;
begin head := 0; tail := 0 end;

function queueempty : boolean;
begin queueempty := (head = tail) end;

function queuefull : boolean;
begin queuefull := (head = (tail+1) mod (MAX+1)) end;
```

1.3.5 Bäume

Die folgenden vier Darstellungen beschreiben die hierarchische Struktur von Bäumen.

1. geschachtelte Klammern: (A(B(D(I),E(J,K,L)),C(F(O),G(M,N),H(P))))
2. geschachtelte Mengen:

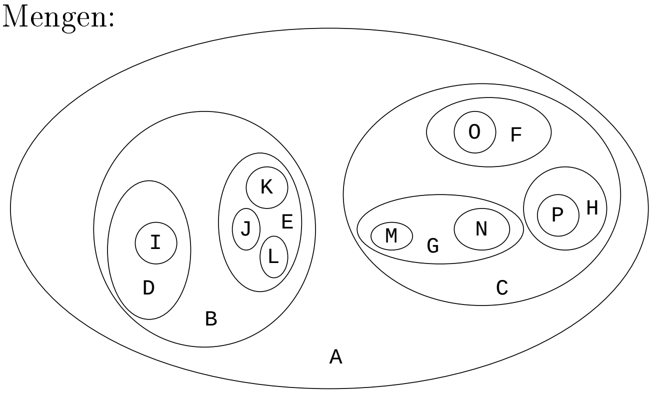


Abbildung 2: geschachtelte Mengen

3. Graph:

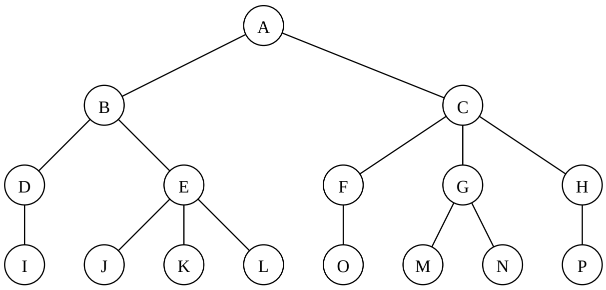


Abbildung 3: Graph

4. Einrückung:

A
    B
        D
            I
        E
            J
            K
            L
    C
        F
            O
        G
            M
            N
        H
            P

Ein Baum besteht aus einer Menge von Knoten und einer Relation, die über der Knotenmenge eine Hierarchie definiert:

- Jeder Knoten hat einen unmittelbaren Vorgänger (Vater, parent, father) außer dem Wurzelknoten (Wurzel, root).
- Kante (Zweig, Ast, branch) : drückt die Beziehung unmittelbarer Nachfolger/Vorgänger aus.
- Grad eines Knotens := Zahl der unmittelbaren Nachfolger
- Blatt, Blattknoten (leaf) := Knoten mit Grad 0.
- Jeder Knoten außer den Blattknoten hat mindestens einen unmittelbaren Nachfolger.
- Sibling (Bruder, Geschwister) := alle unmittelbaren Nachfolger des Vaterknotens ohne den Knoten selbst
- Pfad (Weg) := Folge der Knoten n_1,...,n_k, wobei n_i unmittelbarer Nachfolger von n_{i-1} ist (es gibt keine Zyklen)
Länge des Pfades := Zahl der Knoten - 1
- Tiefe eines Knotens := Länge des Pfades von der Wurzel zu diesem Knoten
- Höhe eines Baumes : = Länge des längsten Pfades (von der Wurzel zu irgendeinem Blatt)
- Grad eines Baumes := Maximum der Grade aller Knoten
Spezialfall mit Grad < 3: Binärbaum
- Ordnung der Knoten: geordneter Baum, sonst ungeordnet.

Beispiel:

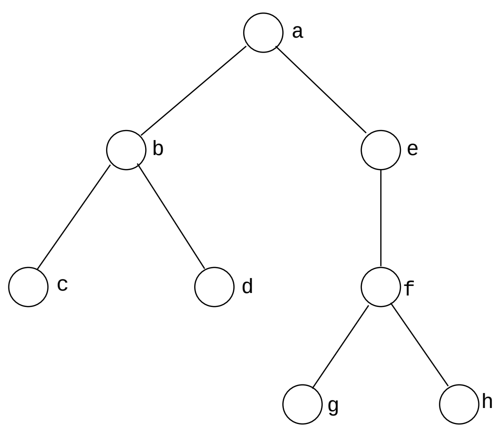


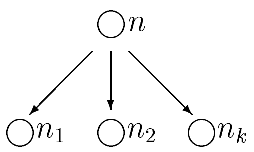
Abbildung 4: Beispiel für einen Baum

Wurzel: a
Vater: a ist Vater von e (und b)
Brüder: b und e sind Brüder
Blätter: c, d, g, h
Tiefe: f hat Tiefe 2
Höhe: des Baumes = 3
Pfad von a nach g: (a,e,f,g)

Induktive Definition eines Baumes

1. Ein einzelner Knoten ist ein Baum. Dieser Knoten ist gleichzeitig der Wurzelknoten.

2. Sei n ein Knoten und seien $T_1, T_2, ..., T_k$ Bäume mit den zugehörigen Wurzelknoten $n_1, n_2, ..., n_k$. Dann wird ein neuer Baum konstruiert, indem $n_1, n_2, ..., n_k$ zu unmittelbaren Nachfolgern von n gemacht werden. T_k heißt dann k-ter Teilbaum (Unterbaum) des Knotens n.

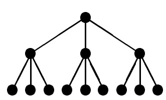


Höhe von Bäumen

Grad d, n Knoten:
maximale Höhe $= n - 1$
minimale Höhe $= \lceil \log_d (n(d-1)+1) \rceil - 1 \leq \log_d n$,
wobei für $x \in \mathbb{R}$:
$\lceil x \rceil$ = nächst höhere ganze Zahl $\geq x$

Beweis:
„maximal": klar, da es nur n Knoten gibt. (Entartung in lineare Liste)
„minimal":

1 Knoten
d Knoten
d^2 Knoten
...



Sei $N(h)$ die maximale Knotenzahl für einen Baum der Höhe h.
Dann gilt: $N(h) = 1 + d + d^2 + ... + d^h$ (geometrische Reihe)

$$= \frac{d^{h+1} - 1}{d - 1}$$

Maximal gefüllte Bäume haben minimale Höhe:

$$\begin{array}{rcl} N(h-1) & < n \leq & N(h) \Leftrightarrow \\ \dfrac{d^h-1}{d-1} & < n \leq & \dfrac{d^{h+1}-1}{d-1} \Leftrightarrow \\ d^h & < n(d-1)+1 \leq & d^{h+1} \Rightarrow \\ h = \lceil \log_d (n(d-1)+1) \rceil - 1 & \leq \lceil \log_d (nd) \rceil - 1 = & \lceil \log_d n \rceil \end{array}$$

wegen $n(d-1)+1 < nd \qquad \forall n > 1$

Spezialisierung: Binärbaum $d = 2$: $h = \lceil \log_2(n+1) \rceil - 1$

Implementierung von Bäumen

Binärbaum: Standard-Implementierungen:

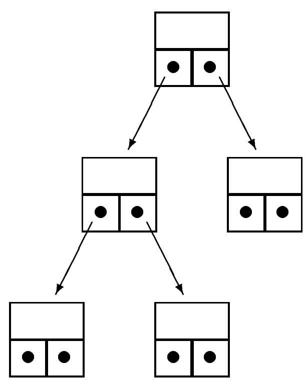
- Pointer (dynamisch)
- Array-Einbettung (nur bei vollständigen binären Bäumen; später zu ,,Heap" erweitert; statisch)

Allgemeiner Baum (Darstellungen sind wenig standardisiert):

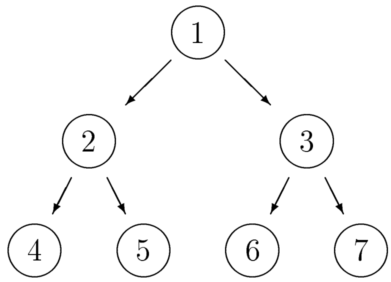
- maximaler Grad vorgegeben: Array von Pointern für jeden Knoten
- Darstellung durch Binärbaum: ,,LeftMostChild, RightSibling"
- Reine Array-Implementierung:
  - (a) ARRAY: Parent[1..N] (Spezialfall: Wurzel)
  - (b) Knoten mit ,,breadth first" durchnumeriert:
    ARRAY: LeftMostChild [1..N]
    RightMostChild [1..N]
  - (c) viele andere Varianten

Bewertung: ähnlich wie bei Listen

1. Pointer-Realisierung:



2. Array-Einbettung (nur bei vollständigen Bäumen)



Predecececessor : $Pred(n) = \frac{n}{2}$

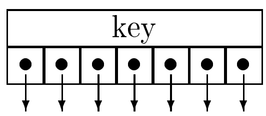
Successor : $Succ(n) = \begin{cases} 2n & left \\ 2n+1 & right \end{cases}$ (falls Successors existieren)

In der Praxis prüft man die Existenz über die obere Grenze der Array-Indizes.

Definition 1.2 Ein Binärbaum heißt vollständig, wenn alle Ebenen bis auf die letzte vollständig besetzt sind und die letzte Ebene von links nach rechts aufgefüllt ist. (auch: links-vollständig)

Allgemeiner Baum vom Grad d

(a) **type** node =**record** key: elem;
sons : **array** [1..d] **of** ↑ node
end;

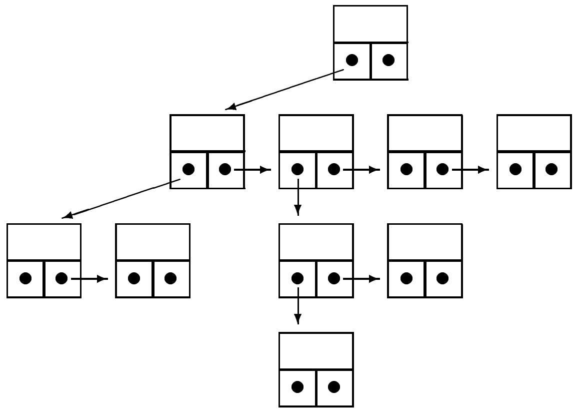


Nachteile:

- der maximale Grad ist vorgegeben
- Platzverschwendung, falls der Grad der einzelnen Knoten stark variiert.

(b) **type** node = **record** key: elem;
leftmostchild : ↑ node;
rightsibling : ↑ node
end;

- dynamische Implementierung
- Platzbedarf:
  #Knoten · (2· sizeof(↑*node*) + sizeof(*node*)),
  wobei sizeof(x) der für x notwendige Speicherplatz ist.
- *leftmostchild* = head der verketteten Liste der Geschwister mit ,,Next-Pointer" *rightsibling*.



1.3.6 Tree Traversal (Baum-Durchlauf)

Ziel: Durchlaufen aller Knoten

1. Depth-First (Tiefendurchlauf):
   Durchlaufen eines Knotens n (rekursiv!) mit den Teilbäumen $n_1...n_k$

   (a) Preorder (Prefixordnung, Hauptreihenfolge):
   - betrachte n
   - durchlaufe $n_1...n_k$

   (b) Postorder (Postfixordnung, Nebenreihenfolge):
   - durchlaufe $n_1...n_k$
   - betrachte n

   (c) Inorder (Infixordnung, symmetrische Reihenfolge):
   - durchlaufe n_1
   - betrachte n
   - durchlaufe $n_2...n_k$

   meist nur für Binärbaum definiert

2. Breadth-First (Breitendurchlauf): Knoten werden ihrer Tiefe nach gelistet, und zwar bei gleicher Tiefe von links nach rechts.

Bei Array-Implementierung ist u. U. keine Rekursion nötig, sondern einfache Schleifendurchgänge sind ausreichend.

Ein Trick zur Erzeugung der drei Auflistungen:
Angenommen, man bewegt sich rund um den Baum, ausgehend von der Wurzel und entgegen dem Uhrzeigersinn zurück zur Wurzel, und man bleibt dabei so nah wie möglich am Baum.

Dann wird für

- Preorder ein Knoten gelistet, wenn man ihm zum ersten Mal begegnet,
- Inorder ein Blattknoten gelistet, wenn man ihm zum ersten Male begegnet, ein anderer Knoten, wenn man ihm zum zweiten Mal begegnet,
- Postorder ein Knoten gelistet, wenn man ihm zum letzten Mal begegnet.

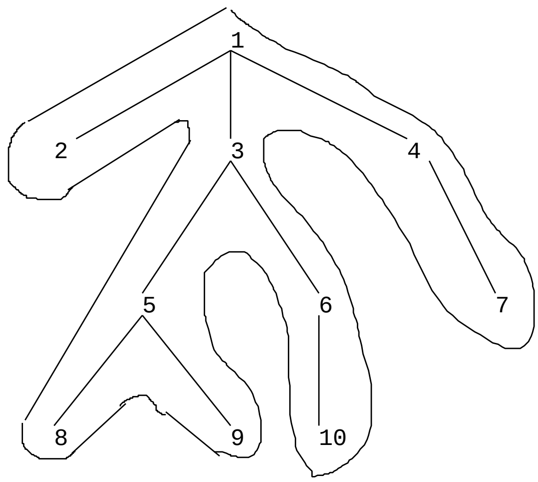


Abbildung 5: Baumdurchmusterung

Programme [S'88] für Binärbäume:

1. Preorder Traversal (mit Stack)

   procedure traverse (t : link);
   begin
       push (t);
       repeat
           t := pop;
           visit (t);
           if t <> z **then** push (t↑.r); (* z entspricht NIL *)
           if t <> z **then** push (t↑.l)
       until stackempty;
   end

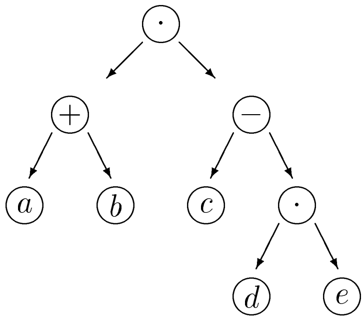
2. Level-order Traversal (mit Queue); auch „Breadth First"

   procedure traverse (t : link);
   begin
       put(t);

```
        repeat
                t := get();
                visit (t);
                if t <>z then put (t↑.l);
                if t <>z then put (t↑.r);
        until queueempty;
end;
```

1.3.7 Traversieren eines Operatorbaumes

Bei Anwendung auf Operatorbäume realisieren die Traversierungsarten eine Darstellung von algebraischen Ausdrücken in unterschiedlichen Notationen (die für rechnerinterne Darstellung besonders vorteilhaft sind). Betrachten wir den zu dem Ausdruck $(a+b)\cdot(c-d\cdot e)$ gehörenden Operatorbaum.



Dann liefert eine Ausgabe aller Knoten mittels der

- Preorder-Traversierung die Prefixnotation: $\cdot + ab - c \cdot de$
  Diese Notation wird auch ,,polnische Notation" genannt.
- Postorder-Traversierung die Postfixnotation: $ab + cde \cdot - \cdot$
  Diese Notation wird auch ,,umgekehrte polnische Notation" genannt,
- Inorder-Traversierung die Infixnotation : $a + b \cdot c - d \cdot e$.

1.4 Entwurfsmethoden

Strenge Regeln für den guten Entwurf sind kaum anzugeben.
Hier einige typische Methoden:

- Rekursion (mit Vorsicht einzusetzen; gutes Beispiel: Tree Traversal)
- divide and conquer [A'74]
- Rekursionsgleichungen [A'74; S'88]
- dynamische Programmierung [A'74]

1.4.1 „Divide and Conquer"-Strategie

- sehr häufig benutzt
- Zerlegung in Teile und Aufbau der Lösung aus den Teillösungen

Beispiel: Gleichzeitige Bestimmung von MAX und MIN einer Menge (Folge) [A'74 p.61]

```
0:  procedure MAXMIN (S);
1:  if ||S||=2 then
        begin
2:          let S={a, b};
3:          return (MAX (a, b), MIN (a, b))
        end
    else
        begin
4:          divide S into two subsets S1 and S2, each with half the elements;
5:          (max1,min1) ← MAXMIN(S1);
6:          (max2,min2) ← MAXMIN(S2);
7:          return (MAX (max1, max2), MIN (min1, min2))
        end
```

$T(n) :=$ Zahl der Vergleiche

$$T(n) = \begin{cases} 1 & n = 2 \\ 2 \cdot T(\frac{n}{2}) + 2 & n > 2 \end{cases}$$

Erklärung: $n = 2$: klar
$n > 2$: 2 Aufrufe von MAXMIN mit $\frac{n}{2}$-elementigen Mengen (Zeilen 5 und 6)
2 Vergleiche (Zeile 7)

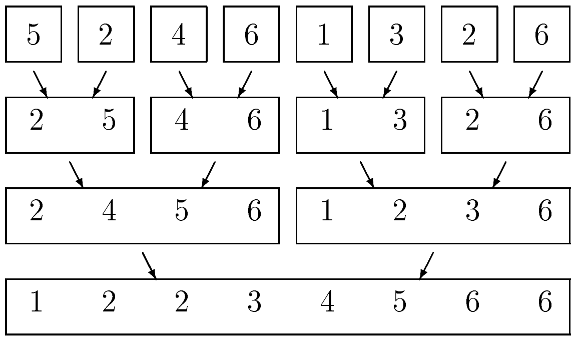
Lösung (Herleitung später): $T(n) = \dfrac{3}{2}n - 2$

Check:

$$\begin{aligned} T(2) &= \frac{3}{2} \cdot 2 - 2 = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + 2 &= 2 \cdot \left[\frac{3}{2} \cdot \frac{n}{2} - 2\right] + 2 = \frac{3n}{2} - 2 = T(n) \end{aligned}$$

allgemeine Lösung: $T(n) = \alpha n - 2$
Normierung: $T(2) = 1$, d.h. Anzahl der Vergleiche

Beispiel: MergeSort (bottom up)



Methode: Zwei Zeiger verweisen auf die nächsten Elemente in den sortierten Teillisten.

$T(n)$ = Zahl der Operationen (im wesentlichen Vergleiche)

$$T(n) = \begin{cases} c_1 & n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n & n > 1 \end{cases}$$

Obere Schranke:

$$T(n) \leq (c_1 + c_2) \cdot n \operatorname{ld} n + c_1$$

Beweis: mit vollständiger Induktion von $\frac{n}{2}$ nach n (Verifizieren der Rekursion)

Induktionsanfang: $n = 1$ korrekt
Induktionsschritt $\frac{n}{2} \to n$:

$$\begin{aligned} T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + c_2 \cdot n \\ &\leq 2\left[(c_1 + c_2)\frac{n}{2} \operatorname{ld} \frac{n}{2} + c_1\right] + c_2 n \\ &= (c_1 + c_2) n \operatorname{ld} n - (c_1 + c_2) n + 2c_1 + c_2 n \\ &= (c_1 + c_2) n \operatorname{ld} n - c_1(n-1) + c_1 \\ &\leq (c_1 + c_2) n \operatorname{ld} n + c_1 \end{aligned}$$

Analyse von MergeSort (exakt)

$T(n)$ = exakte Zahl der Vergleiche

Aufteilung in zwei Folgen: $\left\lceil \frac{n}{2} \right\rceil$ und $\left\lfloor \frac{n}{2} \right\rfloor$ Elemente

Vergleiche: $\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor - 1 = n - 1$

Rekursionsgleichung:

$$T(n) = \begin{cases} 0 & n = 1 \\ n - 1 + T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) & n > 1 \end{cases}$$

Behauptung: $T(n) = n \cdot \lceil \operatorname{ld} n \rceil - 2^{\lceil \operatorname{ld} n \rceil} + 1$
Beweis: [M'88 p.57] mit Fallunterscheidung:

1. $n \neq 2^k + 1$
2. $n = 2^k + 1$

1.4.2 Rekursionsgleichungen

einfacher für $n = 2^k,\ k \in \mathbb{N}$
Beispiele (für $n = 2^k$) [S'88 p.76ff]
Sukzessives Einsetzen:

1.

$$\begin{aligned} T(n) &= T(n-1) + n \quad , \quad n > 1 \qquad (T(1) = 1) \\ &= T(n-2) + (n-1) + n \qquad \text{,,telescoping''} \\ &= \ldots \\ &= T(1) + 2 + \ldots + (n-2) + (n-1) + n \\ &= \frac{n(n+1)}{2} \end{aligned}$$

2.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \qquad (T(1) = 0) \\ &= T\left(\frac{n}{4}\right) + 1 + 1 \\ &= T\left(\frac{n}{8}\right) + 1 + 1 + 1 \\ &= \ldots \\ &= T(1) + \operatorname{ld} n \\ &= \operatorname{ld} n \end{aligned}$$

3.

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \qquad (T(1) = 0) \\ &= T\left(\frac{n}{4}\right) + \frac{n}{2} + n \\ &= T\left(\frac{n}{8}\right) + \frac{n}{4} + \frac{n}{2} + n \\ &= \ldots \\ &= T(1) + \ldots + \frac{n}{8} + \frac{n}{4} + \frac{n}{2} + n \\ &= 2(n-1) \qquad \text{für } n > 1 \end{aligned}$$

4.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \quad , \quad n > 1 \qquad (T(1) = 0) \\
\frac{T(n)}{n} &= \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1 \\
&= \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + 1 + 1 \\
&= ... \\
&= \operatorname{ld} n \\
T(n) &= n \operatorname{ld} n
\end{aligned}$$

5.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \quad , \quad n > 1 \qquad (T(1) = 0) \\
\frac{T(n)}{n} &= \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + \frac{1}{n} \\
&= \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + \frac{2}{n} + \frac{1}{n} \\
&= \frac{T\left(\frac{n}{8}\right)}{\frac{n}{8}} + \frac{4}{n} + \frac{2}{n} + \frac{1}{n} \\
&= ... \\
&= \frac{T(1)}{1} + \frac{\left(\frac{n}{2}\right)}{n} + ... + \frac{4}{n} + \frac{2}{n} + \frac{1}{n}
\end{aligned}$$

$T(n) = \alpha n - 1$ (α positiv wegen Laufzeit; unbekannter Faktor wegen Quotient)
außer $n = 1 : T(1) = 0$

Hinweis: mit $n = 1 :\ T(n) = 2n - 1$ [S'88]
d.h. Normierung auf die Operation für $n = 1$
d.h. „Einheiten von $T(1)$"

Allgemeiner Typ von Rekursionsgleichungen [A'83]

$$T(n) = \begin{cases} 1 & n = 1 \\ a \cdot T\left(\frac{n}{b}\right) + d(n) & n > 1 \end{cases}$$

Annahme: $T(n)$ ist glatt $(n = b^k, k \in \mathbb{N})$

$$T\left(\frac{n}{b^i}\right) = aT\left(\frac{n}{b^{i+1}}\right) + d\left(\frac{n}{b^i}\right) \quad \text{klar mit} \quad n' = \frac{n}{b^i}$$

Sukzessives Einsetzen:

$$\begin{aligned}
T(n) &= a \cdot T\left(\frac{n}{b}\right) + d(n) \\
&= a\left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right] + d(n) \\
&= a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\
&= a^3T\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\
&= \ldots \\
&= a^kT\left(\frac{n}{b^k}\right) + \sum_{j=0}^{k-1} a^j d\left(\frac{n}{b^j}\right)
\end{aligned}$$

$$n = b^k : \qquad T(n) = a^k + \sum_{j=0}^{k-1} a^j d(b^{k-j})$$

$k = \log_b n$ mit $a^k = a^{\log_b n} = n^{\log_b a}$ ist gültig wegen der Eigenschaften des Logarithmus. Der allgemeine Fall ist umständlich [C'93] (Verallgemeinerung „Master Theorem").

Spezialfall: sei $d(n) = n^\gamma$

$$\begin{aligned}
\sum_{j=0}^{k-1} a^j d(b^{k-j}) &= \sum_{j=0}^{k-1} a^j b^{\gamma k} b^{-\gamma j} \\
&= b^{\gamma k} \cdot \sum_{j=0}^{k-1} \left(\frac{a}{b^\gamma}\right)^j \\
&= \begin{cases} b^{\gamma k} \cdot \dfrac{\left(\frac{a}{b^\gamma}\right)^k - 1}{\left(\frac{a}{b^\gamma}\right) - 1} = \dfrac{a^k - b^{\gamma k}}{\left(\frac{a}{b^\gamma}\right) - 1} & a \neq b^\gamma \\ b^{\gamma k} \cdot k & a = b^\gamma \end{cases}
\end{aligned}$$

$a < b^\gamma$: geometrische Reihe konvergiert für $k \to \infty$
$T(n) \leq n^{\log_b a} + n^\gamma \cdot const$
$T(n) = O(n^\gamma)$, (weil $\log_b a < \gamma$)

$a = b^\gamma$: $T(n) = n^\gamma + n^\gamma \cdot \log_b n$
$T(n) = O(n^\gamma \cdot \log_b n)$

$a > b^\gamma$: $T(n) = a^k + O(a^k)$
$T(n) = O(n^{\log_b a})$

1.4.3 Dynamische Programmierung

Beispiel:

Kette von Matrizenprodukten (Assoziativgesetz)

Welche Klammerung ist am günstigsten, d.h. erfordert den geringsten Rechenaufwand?

Seien M_i, $i = 1...4$ reelle Matrizen.
$A = M_1[10, 20] \cdot M_2[20, 50] \cdot M_3[50, 1] \cdot M_4[1, 100]$

Produkt von 2 Matrizen: $[n, i][i, m] = n * i * m$ Operationen (Additionen und Multiplikationen)

Zwei verschiedene Klammerungen:

1. $M_1 \quad \cdot (\quad M_2 \quad \cdot (\quad M_3 \quad \cdot \quad M_4 \;) \;) \;)$

			[50,1,100]	5000
		[20,50,100]		100000
	[10,20,100]			20000
			Summe:	125000

2. $(M_1 \cdot (M_2 \cdot M_3)) \cdot M_4 : 2200$

Problem: Suche nach der besten Klammerung

Vollständige Suche (exhaustive search):

- erfordert exponentielle Komplexität in n (=Zahl der Matrizen)
- exakt: Catalan-Zahlen

Ansatz der dynamischen Programmierung

definiere Hilfgröße $m_{i,j}$:
$m_{i,j}$:= minimale Zahl von Operationen zur Berechnung von

$$M_i \cdot M_{i+1} \cdot ... \cdot M_j \text{ mit } 1 \leq i \leq j \leq n$$

$$\begin{array}{ccccc} (M_i \cdot M_{i+1} \cdot \ldots \cdot M_k) & \cdot & (M_{k+1} \cdot M_{k+2} \cdot \ldots M_j) & \text{mit } M_i \in \mathbb{R}^{r_{i-1} \times r_i} & \\ | & & | & & \text{mit } \; M_i \in \mathbb{R}^{r_{i-1} \times r_i} \\ m_{i,k} & + & m_{k+1,j} & +r_{i-1} \cdot r_k \cdot r_j & \end{array}$$

Dann gilt aufgrund der Definition die Rekursionsgleichung (recurrence relation; DP equation):

$$m_{i,j} = \begin{cases} 0 & j = i \\ \min_{i \leq k < j} \{m_{i,k} + m_{k+1,j} + r_{i-1} \cdot r_k \cdot r_j\} & j > i \end{cases}$$

Erweiterung:

- Speichern der Entscheidungen
- Rekonstruktion der besten Lösung

Allgemeines Konzept

Optimierungsproblem, das sich in Teilprobleme zerlegen läßt (Grundvoraussetzung)

Vorgehen:

- Teilprobleme bearbeiten
- Teilergebnisse in Tabellen eintragen
- Zusammensetzen der Gesamtlösung

Sukzessive Vorgehensweise:

Ausgehend von Elementarproblemen werden nach und nach größere Teilprobleme bearbeitet, bis schließlich die Gesamtlösung betrachtet wird.

Terminologie (Richard Bellman 1957):

- dynamisch := sequentiell
- Programmierung := Optimierung mit Nebenbedingungen (vgl. Lineare Programmierung)

1.5 RAM: Random Access Machine

(deutsch: verallgemeinerte Registermaschine)
Zur genauen Definition müssen wir einen realen Rechner oder noch besser ein Rechnermodell betrachten.

Ziel: axiomatische Definition einer abstrakten Rechenmaschine, so daß genaue Aussagen zur Komplexität möglich sind.

Eine RAM besteht aus:

- Speicher:
  - adressierbare (=random access) Speicherzellen (=register)
- Eingabe- und Ausgabeband:
  - Inhalt: nur Integer-Werte
  - Beide Bänder können nur von links nach rechts bewegt werden.

- Zentrale Recheneinheit mit:
  - Akkumulator
  - Program Counter (Befehlszähler)
- Programm
  - steht nicht im Speicher
  - kann sich nicht selbst verändern
- Befehlssatz:
  - abstrakter Assembler: sehr rudimentär; meist nur Integer-Datentypen
  - (fast) alle Befehle über Akkumulator:
    * LOAD: immediate/direct/indirect
    * ADD, SUB
    * (MUL, DIV): nicht immer
    * JUMP on (Accu=0), JUMP on (Accu>0)
    * JUMP
    * READ, WRITE

Einzelheiten können variieren:

- oft: Akkumulator = Speicherzelle Nr. 0
- Befehlssatz
- „real" RAM
- ...

Anmerkung: Die modernen RISC-Prozessoren sind einer RAM deutlich ähnlicher als die älteren CISC-Prozessoren.

RISC: reduced instruction set computer
CISC: complex instruction set computer

1.5.1 RAM/RASP

Verallgemeinerung des RAM-Modells zum RASP-Modell

RASP: Random Access Stored Program machine

- ähnlich wie RAM

- Unterschied:
  - Das Programm ist im Speicher abgelegt.
  - Das Programm kann sich selbst verändern.

Hinweis: Der Selbstveränderbarkeit des Programms wird zuweilen große Bedeutung beigemessen.

Theoreme:

- RAM ist imstande, genau die partiellen rekursiven Funktionen zu berechnen [A'74, p.8].
- RAM und RASP sind äquivalent [A'74 p.18-33].
- RAM/RASP und Turing-Maschine sind äquivalent.

Diese Aussagen erfordern eine genaue Definition des Begriffs der Äquivalenz. In etwa besagt diese Definition:
,,Die Zeitkomplexität unterscheidet sich nur um einen polynomiellen Faktor."

1.5.2 RAM: Kostenabschätzung

Es sind zwei Arten von Kostenmaßen im Gebrauch:

- Einheitsmaß: unabhängig von der Wortlänge des Operanden
- logarithmisches Maß: Zuweilen ist das sogenannte logarithmische Kostenmaß geeigneter, das die Kosten abhängig macht von der Länge des Operanden, d.h. der Wortlänge des Rechners = Anzahl der Bits. Das logarithmische Maß sollte verwendet werden, wenn ein Operand mehr als eine Speicherzelle belegt (hier nicht verwendet).

Kosten (Einheitsmaß): wenig standardisiert; versuchsweise realistische Schätzung, z.B.:

1. Befehl ohne Memory Access
2. Befehl mit Memory-Access (ADD, MOV, ...)
3. Befehl mit indirektem Memory-Access
4. MUL, DIV: können teurer sein
5. Loop(repeat, while,...): n-mal
   n mal die Kosten für die Befehle innerhalb der Schleife

Abbildung auf Memory:

- Die RAM erfaßt (ziemlich) genau die im realen Rechner anfallenden Memory-Operationen.
- Externe Speicher wie Platten werden (zunächst) nicht betrachtet.

1.5.3 Rekursion

- Realisierung auf RAM mittels Stack
- Ersetzen der Rekursion durch nichtrekursive Implementierung

Methode:

1. Rekursive Prozeduren werden unter Benutzung eines Stacks realisiert.
2. Jeder Aufruf (Inkarnation, Aktivierung) produziert einen sogenannten Aktivierungsblock (Versorgungsblock) als einen Speicherbereich auf dem Stack, bestehend aus:
   - aktuellen Parametern (werden vom rufenden Programm bereitgestellt)
   - lokalen Variablen
   - Rücksprungadresse

   Beachte:

   (a) Der Zugriff auf die aktuellen Parameter (call by value) und die lokalen Variablen erfolgt über den Stack.

   (b) Der Aktivierungsblock wird angelegt für jeden individuellen Aufruf.
3. Nach Beendigung der Prozedur wird der zugehörige Aktivierungsblock entfernt und die Kontrolle an das rufende Programm zurückgegeben.

Beachte:

Hier ist angenommen, daß globale Variable nur aus dem Hauptprogramm kommen, d.h. es gibt keine globalen Variablen, die über Prozedur-Schachtelung definiert werden (wie bei MODULA-3 möglich). Bei Sprachen wie C ist diese Prozedur-Schachtelung ohnehin nicht möglich.

Ein ,,intelligenter” Compiler ersetzt ,,Endrekursion” (tail recursion) durch Iteration (Beispiel: Fakultät).

1.6 Abstrakte Datentypen (ADT)

[S'88, S.31]
Bisherige Beispiele:

- Datenstrukturen (Liste, Stack, Queue, ...)
- und zugehörige Algorithmen

Wenn man nun diese Datenstrukturen nur durch die zugelassenen Operationen definiert und die spezielle Implementierung ignoriert, gelangt man zu den sogenannten „Abstrakten Datentypen". Ein „Abstrakter Datentyp" (ADT) besteht aus:

1. einer Menge von Objekten (Elementen)
2. einer Menge von zulässigen Operationen auf diesen Elementen

Absicht dieser Definition:

- Die tatsächliche Realisierung der Datenstruktur kann nach Bedarf geändert werden, ohne daß der Benutzer des ADT sein Programm ändern muß.
- nützlich für umfangreiche Programmpakete

Unterstützung durch MODULA-2:

- sichtbare Schnittstelle: Definitionsmodul
- verborgene Realisierung: Implementierungsmodul

Literatur: [G: p.1 p.21-23 p.30] und [Puchan: p.222, 248]

Beispiel: Stack als ADT

Algebraische Spezifikation eines Stacks: (modifiziert [Richter et al. p.209])

- Sorten (Datentypen):
  - Stack (hier zu definieren)
  - Element („ElementTyp")
- Operationen (sollen die Syntax eines Datentyps festlegen):
  - stackinit: $\rightarrow$ Stack (Konstante)
  - stackempty: Stack $\rightarrow$ Boolean
  - push: Element $\times$ Stack $\rightarrow$ Stack
  - pop: Stack $\rightarrow$ Element $\times$ Stack

- Axiome (sollen die Semantik des Datentyps definieren):

  x: Element („ElementTyp")
  s: Stack

  pop(push(x,s)) = (x,s)
  push(pop(s)) = s (für nichtleeren Stack s)
  stackempty (stackinit) = TRUE
  stackempty (push(x,s)) = FALSE

Realisierung: [S'88 p. 27]

Anmerkung: Fehlersituationen erfordern Aufmerksamkeit: pop(stackinit) = ?

Potentielle Probleme von ADT's

Allgemein:

- Bei komplexen Fällen wird die Anzahl der Axiome sehr groß.
- Das Verständnis der Spezifikation wird u. U. erschwert.
- Es ist schwierig zu prüfen, ob die Gesetze vollständig und widerspruchsfrei sind

Hier:

- Relativ kleine und kompakte Programme
- Höhere Abstraktion: Problemstellung hat Priorität gegenüber der Schnittstelle, durch die sich die Realisierung des ADT verbergen läßt.
- Effizienz spielt für uns eine entscheidende Rolle.

2 Sortieren

2.1 Einführung

1. Elementare Sortierverfahren
   - SelectionSort
   - InsertionSort
   - BubbleSort
   - Indirektes Sortieren
   - BucketSort
2. QuickSort
3. HeapSort
4. Untere Schranken
5. RadixSort

wichtige Aufgabe:
Laut IBM macht Sortieren 25% in kommerziellen Rechenanlagen aus[M'88 p.40].

Beispiele für das Sortieren nach der Key-Komponente (Schlüssel) von Datensätzen (Records)

- reelle Zahlen
- Namenslisten
- Telefonnummern und Telefonbuch
- Wörterbuch (Lexikon)

Ordnungen:

- reelle oder ganze Zahlen
- lexikographische Ordnung (wie ,,Lexikon" alphabetisch)

Ziel:

- Umordnen der Datensätze in eine ,,monotone" Folge (im Sinne der obigen Ordnungsdefinition).
- Ergebnis ist eine Permutation der ursprünglichen Folge
  Permutation k_i: $a[k_1] \leq a[k_2] \leq ... \leq a[k_i] \leq a[k_{i+1}] \leq a[k_n]$

- Beachte: Duplikate auf der Ebene der Schlüssel oder ganzer Datensätze
- Eine Menge enthält per definitionem keine Datensatz-Duplikate.
- Variante: bei großen Records werden Zeiger statt der Daten selbst sortiert („indirektes Sortieren").

Deklarationsteil:

type item = **record**
...
key : **integer**
end;
var A : **array** [1..N] **of** item;

prototypisch:

- Effizienz von Algorithmen
- intensiv untersuchtes Problem

Terminologie:

- Sortiermethode
- Effizienz: $O(n^2)$, $O(n \log n)$
- intern (alle Records im Arbeitsspeicher) versus extern (Platten)
- direkt/indirekt (d.h. mit Pointern oder Array-Indizes)
- im Array oder nicht
- in situ (in einem einzigen Array ohne zusätzlichem Array) oder nicht
- allgemein/speziell (BucketSort, Fachverteilen)
- stabil: Reihenfolge von Records mit gleichem Schlüssel bleibt erhalten.

verwandte Aufgaben:

- Median
- k kleinsten Elemente

Andere, nicht behandelte Methoden:

- BinaryInsertionSort
- ShellSort
- ShakerSort
- MergeSort (zunächst nicht)

2.2 Elementare Sortierverfahren

2.2.1 SelectionSort

Sortieren durch (direkte) Auswahl

Daten: $A[1], ..., A[n]$ ganze/reelle Zahlen

Prinzip: i-ter Durchgang der Schleife $i = 1, .., n$:

- bestimme Datensatz mit dem kleinsten Schlüssel aus $A[i], ..., A[n]$
- vertausche dieses Minimum und $A[i]$

Beachte: Trotz der Einfachheit wird jeder Datensatz höchstens einmal bewegt. (Vorteil von SelectionSort)

Achtung: Es existieren andere Varianten von SelectionSort mit anderer Anzahl von Vertauschungen.

Programm [S'88]:

```
procedure selection;
    var i, j, min, t : integer;
begin
    for i := 1 to N−1 do
        begin
        min := i;
        for j := i + 1 to N do
            if a[j] < a[min] then min := j;
        t := a[min]; a[min] := a[i]; a[i] := t
        end
end;
```

Beispiel:

$\boxed{3}$	7	8	6	4	2
2	$\boxed{7}$	8	6	4	3
2	3	$\boxed{8}$	6	4	7
2	3	4	$\boxed{6}$	8	7
2	3	4	6	$\boxed{8}$	7
2	3	4	6	7	8

2.2.2 InsertionSort

Sortieren durch (direktes) Einfügen

Datensatz: $A[1], ..., A[n]$ ganze/reelle Zahlen

Prinzip: i-ter Durchgang der Schleife $i = 1, .., n$:

- Datensatz $A[i]$ wird in die korrekte Position der bereits sortierten Folge gebracht.

Typisch:

- Ein häufiger Trick ist die Verwendung eines sogenannten „Sentinel-Elementes" (=Wärter; Anfangs- oder Endmarkierung), um eine Abfrage zu sparen.

Anmerkungen:

- Sentinel kommt auch in anderen Sortierverfahren und sonst vor, um eine zusätzliche Abfrage auf Array-Grenze einzusparen.
- Zusammenhang mit Auswertung des logischen „AND"

Programm [S'88]:

```
procedure insertion;
    var i, j, v :integer;
begin
    for i := 2 to N do
        begin
        v := a[i]; j := i;
        while a[j−1] > v do
            begin a[j] := a[j−1]; j := j − 1 end;
        a[j] := v
        end
end;
```

Beispiel: InsertionSort

3	7	8	6	4	2	
3						
3	7					
3	7	8				
3	6	7	8			
3	4	6	7	8		
2	3	4	6	7	8	← Sentinel-Element $a[0] = -\infty$

2.2.3 BubbleSort

Prinzip: i-ter Durchgang der Schleife $i = N, N-1, ...2$:

- Schleife $j = 2, 3, ..., i$: ordne $A[j-1]$ und $A[j]$

Beachte: Die Analyse wird zeigen, daß BubbleSort hoffnungslos ineffizient ist.

Programm [S'88]:

```
procedure bubble;
    var i, j, t: integer;
begin
    for i := N downto 1 do
        for j:= 2 to i do
            if a[j−1] > a[j] then
                begin t := a[j−1]; a[j−1] := a[j]; a[j] := t end
end;
```

Beispiel: BubbleSort

3	7	8	6	4	2
3	7	6	4	2	8
3	4	2	6	7	8
3	4	2	6	7	8
3	2	4	6	7	8
2	3	4	6	7	8

Bild [S'88 p.101/102] : Anordnung der Elemente nach $\frac{n}{4}$, $\frac{n}{2}$, $\frac{3n}{4}$ der Verarbeitung der drei Verfahren Bubble-, Selection- und InsertionSort.

Abbildung 6: SelectionSort einer zufälligen Permutation

Abbildung 7: InsertionSort einer zufälligen Permutation

Abbildung 8: BubbleSort einer zufälligen Permutation

Komplexität

Abschätzung: Die drei Verfahren Bubble-, Selection-, InsertionSort benötigen Zeitkomplexität $O(n^2)$.

genaueres Abzählen:

- Comparisons (Vergleiche): nur Key-Komponente
- Moves (Bewegungen): ganzer Record

Man unterscheidet:

- worst case
- average case
- best case

Spezialfall: (fast) sortierte Daten:

- nur die letzten k Elemente nicht sortiert
- k unsortierte Elemente „gleichmäßig" verteilt über $A[1], ..., A[n]$

Übungsaufgabe (Bubble-,Selection-, InsertionSort)

- Bestimmung der genauen Zahl der Operationen
- Welche Methode ist bei „fast" sortierten Daten vorzuziehen ?

	best case	average case	worst case	
Insertion	$3(N-1)$	$\frac{N^2}{4}$	$\frac{N^2}{2}$	moves
	$N-1$	$\frac{N^2}{4}$	$\frac{N^2}{2}$	compares
Selection	$3(N-1)$	$3(N-1)$	$3(N-1)$	moves
	$\frac{N^2}{2}$	$\frac{N^2}{2}$	$\frac{N^2}{2}$	compares
Bubble	0	$\frac{3N^2}{4}$	$\frac{3N^2}{2}$	moves
	$\frac{N^2}{2}$	$\frac{N^2}{2}$	$\frac{N^2}{2}$	compares

Folgerungen:

1. BubbleSort : ineffizient, da immer $\frac{N^2}{2}$ Vergleiche
2. InsertionSort: gut für fast sortierte Daten
3. SelectionSort: gut für große Records wegen konstant $3(N-1)$ Bewegungen, aber immer $\frac{N^2}{2}$ Vergleiche

Diese Verfahren sollte man nur für $N \leq 50$ einsetzen.

2.2.4 Indirektes Sortieren

Bei sehr großen Records dominiert das Vertauschen (Move, Swap, Exchanges) der Records den Rechenaufwand.

Jede der drei Methoden (Bubble-, Insertion-, SelectionSort) kann so modifiziert werden, daß nicht mehr als N Vertauschungen nötig sind:

1. Verwendung eines Pointer-Arrays $p[1..N]$
2. Initialisierung: $p[i] = i$; $i = 1, ..., N$
3. Zugriff auf Records: $a[p[i]]$ für Vergleiche
4. ,,Vertauschen" von Zeigern $p[i]$
5. optional werden nach dem Sortieren die Records selbst umsortiert

Beispiel [S'88]: Umsortieren:
(a) mit zusätzlichem Array: trivial
(b) kein zusätzliches Array: ,,insitu", ,,in place" (lohnt nur bei großen Records):

Ziel: $p[i] = i$

- falls $p[i] = i$: nichts zu tun;
- sonst: zyklische Vertauschung durchführen:
  - save record: $t = a[i]$; Ergebnis: Loch für dieses i;
  - ,,iterieren"
    (Beispiel): $t = a[2]$; $a[2] = a[11]$; $a[11] = a[13]$; $a[13] = t$;

    Ersetzungsreihenfolge des Ringtausches: $2 \rightarrow 11 \rightarrow 13$

Programm [S'88]:

```
procedure insitu;
    var i, j, k, t : integer;
    begin
    for i := 1 to N do
        if p[i] <> i then
            begin
            t := a[i]; k := i;
            repeat
                j := k; a[j] := a[p[j]];
                k := p[j]; p[j] := j
            until k = i;
            a[j] := t
            end
end;
```

Beispiel: Wiederherstellen eines sortierten Feldes

Vor dem Sortieren:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Nach dem Sortieren:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
p[k]	1	11	9	15	8	6	14	12	7	3	13	4	2	5	10

Nach dem Permutieren:

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a[k]	A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
p[k]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

2.2.5 BucketSort

Andere Namen: Bin Sorting; Distribution Counting; Sortieren durch Fachverteilen; Sortieren mittels Histogramm

Annahme: Schlüssel können als Integer-Werte im Bereich $0...m-1$ dargestellt werden, so daß sie als Array-Index verwendet werden können.

$$a[i] \in \{0, ..., m-1\} \ \forall i = 1, \ldots, n$$

Methode:

- Zähle für jeden möglichen Schlüsselwert, wie häufig er vorkommt (d.h. erstelle Histogramm).
- Berechne aus diesem Histogramm die Position für jeden Record und bewege ihn dorthin (mit rückläufigem Index).
- Wegen des rückläufigen Index ist BucketSort stabil.

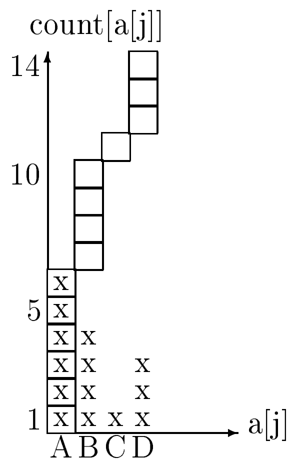
Zeit- und Raumkomplexität: $T(n) = O(n + m) = O(\max(n, m))$

Erweiterung: RadixSort

Programm [S'88]:

```
procedure bucketsort;
    var i, j : integer;
begin
    for j := 0 to M − 1 do count[j] := 0;
    for i := 1 to N do
        count[a[i]] := count[a[i]] + 1;
    for j := 1 to M − 1 do
        count[j] := count[j−1] + count[j];
    for i := N downto 1 do
        begin
        b[count[a[i]]] := a[i];
        count[a[i]] := count[a[i]] − 1
        end;
    for i := 1 to N do a[i] := b[i]
end;
```

Beispiel: ABBACADABBADDA



2.3 QuickSort

eingeführt: C.A.R. Hoare 1962

Ansatz: Divide and Conquer

- (wiederholte) Zerlegung in 2 Teilarrays F_1 und F_2 und Umsortieren, so daß gilt: $x_1 \leq x_2$ für $x_1 \in F_1$ und $x_2 \in F_2$
- Anwendung desselben Schemas auf die so erzeugten Teilarrays
- Abbruchkriterium: ein-elementige Arrays

Warum gewinnt man damit Rechenzeit?

- Vertauschen der Elemente über größere Distanzen
- Abschätzung für idealen Fall: Das partitionierende Element wird so geraten, daß die jeweilige Teilfolge genau halbiert wird:

  1. $\frac{n}{2} + \frac{n}{2}$
  2. $\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4}$
  3. $\frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8}$

  ld n. $1 + 1 + 1 + 1 + ... + 1$

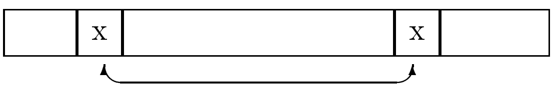
  Summe: $n \cdot \operatorname{ld} n$

Demnach gilt im idealen Fall (d.h. bei exakter Halbierung in jedem Schritt):

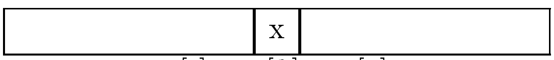
$$T(n) = n \cdot \operatorname{ld} n$$

Idee

- Austauschen über größere Distanzen



- Zerlegung („Partition") des Arrays in zwei Teile



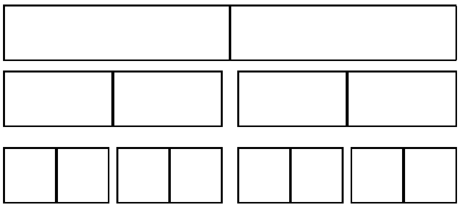
$a[i] \leq a[k] \leq a[j]$

$i = 1, ..., k-1$ $\quad$ $j = k+1, ..., N$

Pivot-Element

- Rekursion: linkes und rechtes Teilarray bearbeiten

Komplexitätsabschätzung:



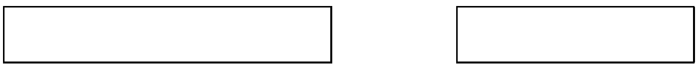
n Vergleiche	Stufe 1
$\frac{n}{2} + \frac{n}{2}$	Stufe 2
$\frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4}$	Stufe 3
...	...
$1 + 1 + 1 + ... + 1$	Stufe $\lceil \mathrm{ld}\,(n+1) \rceil$

...

$\Rightarrow\ n \cdot \mathrm{ld}\, n$ (bei exakter Halbierung)

Beispiel: wähle Pivotelement 44

$\boxed{44}$	55	12	42	94	6	18	67
$\boxed{44}$	18	12	42	94	6	55	67
$\boxed{44}$	18	12	42	6	94	55	67
6	18	12	42	$\boxed{44}$	94	55	67



Konzept:

```
procedure QuickSort (l, r : integer);
var i: integer;
begin
      if (r > l) then
      begin
            i := Partition (l, r);
            QuickSort (l, i−1);
            QuickSort (i+1, r)
      end
end;
```

Andere Variante:

```
procedure QuickSort (l, r : integer);
var i: integer;
begin
      i := Partition (l, r);
      if (l < i−1) then QuickSort (l, i−1);
      if (i+1 < r) then QuickSort (i+1, r)
end;
```

```
function Partition (l, r):integer;
begin
    if (r > l) then
    begin
        i:= l−1; j := r;
        wähle Pivot-Element: v = a[r];
        repeat
            durchsuche die Folge von links (i:=i+1), bis a[i] >= v;
            durchsuche die Folge von rechts (j:=j−1), bis a[j] <= v;
            vertausche a[i] und a[j];
        until j <= i; (* pointers cross *)
        rückvertausche a[i] und a[j];
        vertausche a[i] und a[r]
    end;
    return i
end;
```

Beachte: wegen $i = l - 1$ ist Sentinel nötig: $a[0] = -\infty$

Anmerkungen zu Sedgewick's QuickSort

Allgemeine Warnung:

- viele Varianten
- besondere Aufmerksamkeit bei Schleifen und Indizes erforderlich

Die Variable $v = a[r]$ heißt Pivot-Element (partitioning element).

i, j: laufende Zeiger(Pointer) (i=links; j=rechts)

Vertauschung $a[i]$ und $a[j]$:

- innerhalb **repeat** ... **until** für $j < i$
- eine Vertauschung zuviel nach dem „Kreuzen der Pointer" (vermeidet **goto/break**)
  Deswegen wird nach **until**
  - die letzte Vertauschung wieder rückgängig gemacht:
    vertausche $a[i]$ und $a[j]$
  - das Pivot-Element positioniert:
    vertausche $a[j]$ und $a[r]$

Das Abbruchkriterium der Rekursion ist gegeben durch die Abfrage im äußersten Block von QuickSort:

if (r > l) **then**

Sentinel Keys:

- left : $a[0] = -\infty$
- right : nicht nötig, weil $v = a[r]$

Die Bilder auf dieser Seite zeigen die Sortierverteilungen für Quicksort (als rekursive Implementation) nach 2, 3, 4, 5, 6, 7 Partitionen, wobei Partitionen minimal 12 Elemente enthalten.

Varianten und Verbesserungen:

- Pivot-Element: $v := (a[l] + a[r])/2$
- median-of-three (clever QuickSort) (keine Sentinel-Elemente nötig)
- QuickSort ist ineffizient bei kleinen Arrays:

  if (r−l > 22) **then** QuickSort
  else InsertionSort;
- Speicherplatz: indirekt über Stack; Beschränkung des Stacks möglich: $\mathrm{ld}\, n$
- Iterative Variante

Zeitkomplexität

$T(n)$ = Zahl der Vergleiche

1. **Worst-Case-Analyse**, d.h. obere Schranke wird gesucht.

$$\begin{aligned} T(0) &= T(1) = 0 \\ T(n) &= n + 1 + \underbrace{\max_{1 \le k \le n} [T(k-1) + T(n-k)]}_{\text{im ungünstigsten Fall}} \end{aligned}$$

   Dann gilt:

$$T(n) \le \frac{(n+1)(n+2)}{2} - 3$$

   Beweis: mit vollständiger Induktion nach n
   Der Fall eines aufsteigend sortierten Arrays ohne Duplikate (d.h. gleiche Schlüssel) zeigt, daß diese obere Schranke scharf ist, denn für den sortierten Array haben wir:

$$\begin{aligned} T(n) &= (n+1) + n + (n-1) + (n-2) + \ldots + 3 \\ &= \frac{(n+1)(n+2)}{2} - 3 \quad \square \end{aligned}$$

2. **Average Case:** $T(n)$ *mittlere* Zahl von Vergleichen

   average: Mitteln über alle n möglichen Pivotelemente $a[i],\ i = 1, ..., n$

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{k=1}^{n} (n + 1 + [T(k-1) + T(n-k)]) \quad n \ge 2 \\ &= n + 1 + \frac{1}{n} \underbrace{\left[\sum_{k=1}^{n} T(k-1) + \sum_{k=1}^{n} T(n-k) \right]}_{=2\cdot\sum_{k=1}^{n} T(k-1)} \\ &= n + 1 + \frac{2}{n} \sum_{k=1}^{n} T(k-1) \end{aligned}$$

Eliminieren der Summe:

$$\left.\begin{array}{rcl} nT(n) & = & n(n+1) + 2\sum_{k=1}^{n} T(k-1) \\ (n-1)T(n-1) & = & (n-1)n + 2\sum_{k=1}^{n-1} T(k-1) \end{array}\right\} -$$

$$\begin{aligned} n \cdot T(n) - (n-1) \cdot T(n-1) &= 2n + 2 \cdot T(n-1) \\ nT(n) &= 2n + (n+1)T(n-1) \\ \frac{T(n)}{n+1} &= \frac{2}{n+1} + \frac{T(n-1)}{n} \end{aligned}$$

sukzessives Substituieren:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{2}{n+1} + \frac{2}{n} + \frac{T(n-2)}{n-1} \\ &= \frac{2}{n+1} + \frac{2}{n} + \underbrace{\frac{2}{n-1} + \frac{T(n-3)}{n-2}}_{\text{(Grenze: } n=4)} \\ &= \ldots \\ &= \sum_{k=2}^{n} \frac{2}{k+1} + \frac{T(1)}{2} \\ &= 2 \cdot \sum_{k=3}^{n+1} \frac{1}{k} + \frac{T(1)}{2} \end{aligned}$$

Wir werden zeigen:

$$\ln \frac{n+1}{m} \leq \sum_{k=m}^{n} \frac{1}{k} \leq \ln \frac{n}{m-1} \quad \text{für } m \geq 2$$

obere Schranke:

$$\frac{T(n)}{n+1} \leq 2 \ln \frac{n+1}{2} + \frac{T(1)}{2}$$

untere Schranke:

$$\frac{T(n)}{n+1} \geq 2 \ln \frac{n+2}{3} + \frac{T(1)}{2}$$

$$\begin{aligned} \Rightarrow T(n) &= 2(n+1)\ln(n+1) + \ldots \\ &= 1.386(n+1)\log_2(n+1) + \ldots \end{aligned}$$

Zusammenfassung : Analyse von Quicksort

$$\begin{aligned} \text{best case :} \quad T(n) &= n \log_2 n + \ldots \\ \text{average case :} \quad T(n) &= 1.386 \cdot n + \log_2 n + \ldots \\ \text{worst case :} \quad T(n) &= \frac{(n+1)(n+2)}{2} - 3 \end{aligned}$$

Approximation mittels Integral-Methode

$f(x)$ monoton fallend, $f(x) \geq 0$

$$(b-a) \min_{a \leq x \leq b} f(x) \leq \int_a^b f(x)dx \leq (b-a) \max_{a \leq x \leq b} f(x)$$

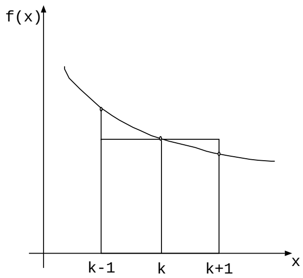


Abbildung 9: Unter- Obersummen

$$\begin{aligned} \int_k^{k+1} f(x)dx &\leq 1 \cdot f(k) \leq \int_{k-1}^{k} f(x)dx \\ \int_m^{n+1} f(x)dx &\leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)dx \end{aligned}$$

speziell: $f(k) = \dfrac{1}{k}$

$$\ln \frac{n+1}{m} \leq \sum_{k=m}^{n} \frac{1}{k} \leq \ln \frac{n}{m-1} \quad \text{für} \quad m \geq 2$$

falls $m = 1$:
untere Schranke: unverändert
obere Schranke:

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \sum_{k=2}^{n} \frac{1}{k} \leq 1 + \ln n$$

Definition 2.1 Harmonische Zahlen H_n, n$\geq$1

$$H_n := \sum_{k=1}^{n} \frac{1}{k}$$

Folgerung:

$$\ln(n+1) \leq \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \quad \forall n \geq 1$$

Es gilt (ohne Beweis):

$$\lim_{n \to \infty} (H_n - \ln n) = \gamma = 0.5772\ldots \text{ (Eulersche Konstante)}$$

2.4 HeapSort

J.W.J. Williams 1964 und R.W. Floyd 1964

Erweiterung von SelectionSort mittels eines Heap

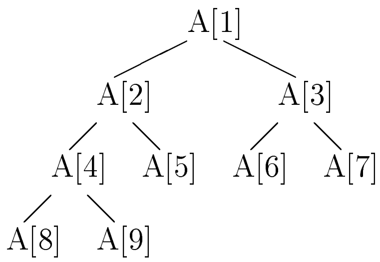
Definition 2.2 Ein Heap ist eine Datenstruktur mit folgenden Eigenschaften:
Ein Array $A[1..n]$ erfüllt die Heap-Eigenschaft, falls

$$A\left[\left\lfloor\frac{i}{2}\right\rfloor\right] \geq A[i] \ \text{ für } 2 \leq i \leq n$$

Ein Array $A[1..n]$ ist ein Heap, beginnend in Position l mit $1 \leq l \leq n$, falls

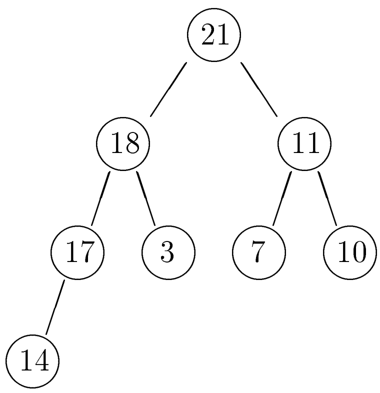
$$A\left[\left\lfloor\frac{i}{2}\right\rfloor\right] \geq A[i] \text{ für } l \leq \left\lfloor\frac{i}{2}\right\rfloor < i \leq n.$$

Darstellung: Binärer Baum, eingebettet in Array

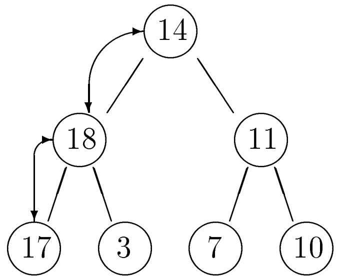


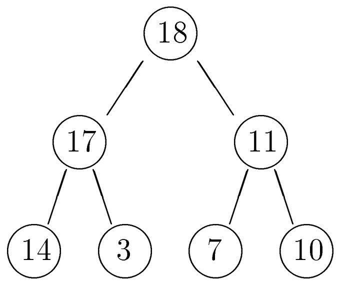
Beachte: Jeder Array $A[1..n]$ ist ein Heap beginnend in $\left\lfloor\frac{n}{2}\right\rfloor + 1$, (weil keine Bedingung zu erfüllen ist).

Beispiel:



Entferne 21 und setze 14 oben ein :





Für einen Heap im Array $A[1..n]$ gilt:

$$A[1] = \max\{A[i] | i = 1, \ldots, n\}$$

Damit versuchen wir den Ansatz:

HeapSort

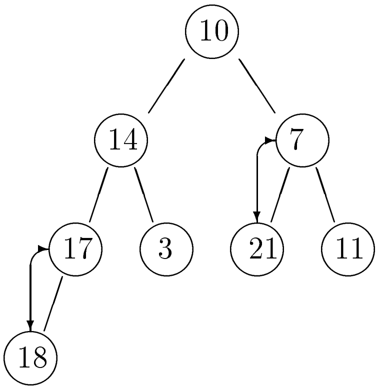
1. wandle Array $A[1..n]$ in Heap um
2. **for** $i = 1$ **to** $n - 1$ **do**
   (a) entferne $A[i]$ und setze $A[n - i + 1]$ ein
   (b) wandle Rest-Array $A[1..(n - i)]$ in Heap um

Schritte 2.(a) und 2.(b) sind geklärt.
Schritt 1 bleibt zu klären: Heap-Aufbau

Beispiel:

10	14	7	17	3	21	11	18

$\rightarrow$ Heap



10	14	7	18	3	21	11	17

→ Heap

Als nächstes: 7 ↔ 21

10	14	21	18	3	7	11	17

→ Heap

dann: 14 ↔ 18 ↔ 17

10	18	21	17	3	7	11	14

→ Heap

zuletzt: 10 ↔ 21 ↔ 11

21	18	11	17	3	7	10	14

Programme [S'88]

```
procedure heapsort;
    var k,t: integer;
begin
    N:=M;
    for k := M div 2 downto 1 do downheap(k);
    repeat
        t := a[1]; a[1] := a[N]; a[N] := t;
        N := N − 1; downheap(1)
    until N <= 1;
end;
```

```
procedure downheap(k : integer);
      label 0;
      var i, j, v: integer;
begin
      v := a[k];
      while k <= N div 2 do
            begin
            j := k + k;
            if j < N then if a[j] < a[j+1] then j := j + 1;
            if v >= a[j] then goto 0;
            a[k] := a[j]; k := j;
            end;
0:    a[k] := v
end;
```

Bemerkung: HeapSort behandelt das Array $a[1..M]$, DownHeap das Teilarray $a[k..N]$

Procedure DownHeap(k : integer) („Versickere", „ReHeap")

- startet in Knoten k
- vertauscht falls nötig $a[k]$ mit dem größeren der beiden Söhne $a[2k]$ und $a[2k+1]$

  Abfragen:

  - existieren beide Söhne ?
  - Blatt erreicht ?
- weiter mit dem Sohn

Analyse der Komplexität: [M'88 p.47]

Befehl:	Vergleich A: **if** $(v \geq a[j])$ **then** Vergleich B: **if** $a[j] < a[j+1]$ **then**
Vergleich A:	wird bei jedem Durchlauf der While-Schleife ausgeführt, die ihrerseits bei jedem Prozeduraufruf DownHeap(k) mindestens einmal durchlaufen wird.
Vergleich B:	wird ausgeführt, falls 2. Sohn existiert.

Deswegen: Komplexität (obere Schranke) = Zahl der Vergleiche A und B

Auf der folgenden Seite sind die Sortierungen der Elemente bei HeapSort in der Konstruktionsphase und Sortierungsphase abgebildet [S'88].

Abbildung 10: HeapSort einer zufälligen Permutation: Konstruktionsphase

Abbildung 11: HeapSort einer zufälligen Permutation: Sortierphase

Sei $n = 2^k - 1$

Aufbau des Heap:

- 2^i Knoten der Tiefe i für $0 \leq i \leq k-1$
- Hinzufügen von Knoten der Tiefe $(k-2), (k-3), \ldots, 1$
- Knoten auf Niveau i hinzugefügt:
  kann auf Niveau $(k-1)$ sinken mit höchstens zwei Vergleichen pro Niveau

$$\sum_{i=0}^{k-2} 2 \cdot (k-1-i) \cdot 2^i = 2^{k+1} - 2(k+1)$$

DownHeap:

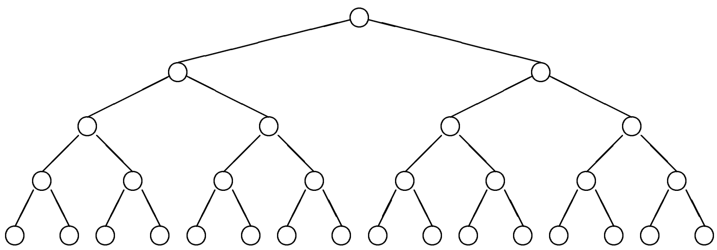
- Knoten der Tiefe i wird auf die Wurzel gesetzt und kann um i Niveaus sinken mit jeweils maximal 2 Vergleichen.

$$\sum_{i=0}^{k-1} 2i \cdot 2^i = 2(k-2) \cdot 2^k + 4$$

z.z. mit vollständiger Induktion für $n = 2^k - 1$

Beispiel:

$k = 5$; $n = 2^5 - 1 = 31$



$1 = 2^0$	$i = 0$
$2 = 2^1$	$i = 1$
$4 = 2^2$	$i = 2$
$8 = 2^3$	$i = 3$
$16 = 2^4$	$i = 4 = k-1$

$$\begin{aligned} T(n) &\leq 2^{k+1} - 2(k+1) + 2(k-2) \cdot 2^k + 4 \\ &= 2k \cdot (2^k - 1) - 2(2^k - 1) \\ &= 2n \log_2(n+1) - 2n \end{aligned}$$

Für $n \neq 2^k - 1$ ist die Rechnung ähnlich, aber umständlicher.

HeapSort: Sortiert n Elemente in (weniger als) $2n \,\mathrm{ld}\,(n+1) - 2n$ Vergleichen.

Verbesserung: Bottom-Up-HeapSort von Wegener [G'93 p.196]

$\Rightarrow$ Zahl der Vergleiche rückt an $1 \cdot n \,\mathrm{ld}\,(n+1)$ heran, so daß die Methode mit QuickSort konkurrenzfähig wird.

2.5 Untere und obere Schranken

Modell : Der Algorithmus kennt nur die Operation ,,Vergleich zweier Schlüssel" (vergleichsorientierter Algorithmus; v.A.)

Satz: Für n Elemente benötigt jeder vergleichsorientierte Algorithmus $(n-1)$ Vergleiche, um das Minimum der n Elemente zu bestimmen.

Beweis (durch Widerspruch):

Sei das Minimum in weniger als $(n-1)$ Vergleichen gefunden worden. Dann gibt es ein Element, mit dem das Minimum nicht verglichen worden ist.
$\Rightarrow$ Widerspruch, da dieses Element $<$ Minimum sein kann$\square$

Interpretation von Sortieren mittels binären Entscheidungsbaums:

Durch Sortieren wählen wir eine von $n!$ möglichen Permutationen des Arrays $A[1..n]$ aus.
Die $n!$ Permutationen können als Blätter eines binären Entscheidungsbaums dargestellt werden, an dessen Knoten je genau ein Vergleich durchgeführt wird.

Untere Schranke: Zahl der Vergleiche
$T(n) \geq$ minimale Höhe eines Binärbaumes mit $n!$ Blättern
Es gilt (Kap.1.3): Ein Binärbaum der Tiefe $\leq k$ hat höchstens 2^k Blätter.

$$\Rightarrow 2^{T(n)} \geq n! \quad \Rightarrow T(n) \geq \lceil \text{ld } n! \rceil$$

Obere Schranke: wähle einen ,,effizienten" Algorithmus
MergeSort: $T(n) = n\lceil \text{ld}\, n\rceil - 2^{\lceil \text{ld}\, n\rceil} + 1$

Obere und untere Schranke liegen *eng* zusammen:

$$n \,\text{ld}\, n - (n-1)\,\text{ld}\, e \leq T(n) \leq n\lceil \text{ld}\, n\rceil - n + 1$$

d.h. MergeSort ist bezüglich der *Zahl der Vergleiche T(n)* praktisch optimal.
Das gilt nicht unbedingt für die gesamte Rechenzeit.

2.5.1 Einfache Schranken für n!

$$\begin{aligned}
n! &= \prod_{k=1}^{n} k \leq n^n \\
n! &= \prod_{k=1}^{n} k \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\
\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! &\leq n^n \\
\frac{n}{2}\log\left(\frac{n}{2}\right) \leq \log(n!) &\leq n \log n
\end{aligned}$$

$\mathrm{ld} := \log_2 :$ $\qquad \frac{n}{2}(\,\mathrm{ld}\, n - 1) \leq \,\mathrm{ld}\,(n!) \leq n \,\mathrm{ld}\, n$

Approximation mit Integral für n!

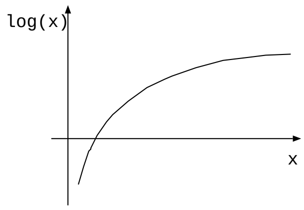


Abbildung 12: $\log x$ monoton wachsend

$$\underbrace{\int_1^n \ln x dx}_{=[x\ln x - x]_1^n = n\ln n - n + 1 = \ln[e(\frac{n}{e})^n]} < \ln n! < \int_1^{n+1} \ln x dx$$

$$\begin{aligned} e \cdot \left(\frac{n}{e}\right)^n < n! &< e \cdot \left(\frac{n+1}{e}\right)^{n+1} \\ &= (n+1)\left(\frac{n}{e}\right)^n \underbrace{\left(1 + \frac{1}{n}\right)^n}_{<e} \\ e \cdot \left(\frac{n}{e}\right)^n < n! &< (n+1) \cdot e \cdot \left(\frac{n}{e}\right)^n \end{aligned}$$

Die Abschätzung liegt genau innerhalb eines Faktors zwischen 1 und $(n+1)$.

$$n \,\mathrm{ld}\, n - (n-1) \,\mathrm{ld}\, e < \,\mathrm{ld}\,(n!)$$

Approximation: n!

$$\begin{aligned} \ln n! &= \ln[1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n] \\ &= \sum_{k=1}^{n} \ln k = \sum_{k=2}^{n} \ln k \quad \forall n \geq 2 \end{aligned}$$

$\ln k$ ist monoton wachsend:

$$\int_1^n \ln x \, dx \leq \sum_{k=2}^{n} \ln k \leq \int_1^{n+1} \ln x \, dx$$

Bemerkung: Durch die untere Grenze 1 statt 2 des zweiten Integrals wird das Integral nur größer, da $\ln x$ positiv für $x > 1$.

Stammfunktion zu $\ln x$: $x \ln x - x$

$$\begin{array}{rcl} n \ln n - n \underbrace{+1}_{\text{weglassen}} & \leq \ln n! \leq & (n+1)\ln(n+1) \underbrace{-(n+1) - (0-1)}_{-n} \\ n \ln n & \leq \ln(n!) + n \leq & (n+1)\ln(n+1) \quad \forall n \geq 2 \end{array}$$

ohne Beweis: Stirlingsche Approximation und Verfeinerung
[Krengel: ,,Wahrscheinlichkeitstheorie und Statistik"]

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \cdot \alpha_n$$

Es gilt:

1. $$\lim_{n \to \infty} \alpha_n = 1$$

2. $$e^{\frac{1}{12n+1}} \leq \alpha_n \leq e^{\frac{1}{12n}} \quad n \geq 2$$

3. $$e^{\frac{1}{12n+\frac{1}{4}}} \leq \alpha_n \leq e^{\frac{1}{12n}} \quad n \geq 2$$

2.6 RadixSort

Prinzip: Alle Daten sind digital, speziell binär (Radix=2) dargestellt und daher läßt sich Sortieren auf Bitebene formulieren.

function bits (x,j,k: **integer**): **integer**

- shifte x nach rechts um k Bit-Positionen
- setze alle Bits auf Null außer Bits $1, 2, ..., j$

Zurückgegebener Wert: $\left\lfloor \frac{x}{2^k} \right\rfloor \bmod 2^j$

Zwei Methoden:

- RadixExchange
- StraightRadixSort

RadixExchange:

- wie QuickSort („Partition Exchange")
- prüft Bits von links nach rechts (Pivot-Elemente= ... 32,16,8,4,2,1)
- Pivot-Element = 2^b (Wert selbst braucht nicht im Array vorzukommen; potentielle Probleme wie QuickSort)
- statt Sentinel zusätzliche Tests: „$i < j$"

StraightRadixSort:

- Erweiterung von BucketSort
- wichtig: stabil (nachprüfen!)
- $M = 2^m \ (m > 0)$
- w: Vielfaches von m
- Grenzfälle:
  - $m = w$: BucketSort
  - $m = 1$: extremes StraightRadixSort

Komplexität

Sortieren von n Elementen mit b-Bit-Schlüsseln

RadixExchange:

1. wie QuickSort (average case): $n \operatorname{ld} n$ Bitvergleiche
2. in Abhängigkeit von n und b (nachzählen!): $< n \cdot b$ Bitvergleiche

StraightRadixSort:

1. Zeit:
   (a) $< n \cdot b$ Bitvergleiche
   (b) $\frac{b}{m}$ Durchläufe
2. Platz:
   (a) Tabelle „count$[0..M]$"
   (b) zusätzliches Array $b[1..N]$

Programme: [S'88]

```
procedure straightradix;
var i, j, pass: integer;
    count: array [0..M] of integer;
begin
    for pass := 0 to (w div m) − 1 do
        begin
        for j := 0 to M − 1 do count[j] := 0;
        for i := 1 to N do
            count[bits(a[i], pass·m, m)] := count[bits(a[i], pass·m, m)] + 1
        for j := 1 to M − 1 do
            count[j] := count[j−1] + count[j];
        for i := N downto 1 do
            begin
            b[count[bits(a[i], pass·m, m)]] := a[i];
            count[bits(a[i], pass·m, m)] := count[bits(a[i], pass·m, m)] − 1
            end;
        for i := 1 to N do a[i] := b[i]
        end
end;
```

Bemerkung:

1. $M = 2^m$
2. w rightmost Bits
3. w = Vielfaches von m

```
procedure radixexchange (l, r, b: integer);
var t, i, j: integer;
begin
    if (r > l) and (b <= 0) then
        begin
        i := l; j := r;
        repeat
            while (bits(a[i], b, 1) = 0) and i < j do i := i + 1;
            while (bits(a[j], b, 1) = 1) and i < j do j := j − 1;
            t := a[i]; a[i] := a[j]; a[j] := t;
        until j = i;
        if bits (a[r], b, 1) = 0 then j := j + 1;
        radixexchange (l, j−1, b−1);
        radixexchange (j, r, b−1);
        end
end;
```

Beispiel: Buchstaben in 5-Bit-Darstellung

A	00001	A	$\boxed{0}$0001	A	0$\boxed{0}$001	A	00$\boxed{0}$01	A	000$\boxed{0}$1	A	0000$\boxed{1}$
S	10011	E	$\boxed{0}$0101	E	0$\boxed{0}$101	A	00$\boxed{0}$01	A	000$\boxed{0}$1	A	0000$\boxed{1}$
O	01111	O	$\boxed{0}$1111	A	0$\boxed{0}$001	E	00$\boxed{1}$01	E	001$\boxed{0}$1	E	0010$\boxed{1}$
R	10010	L	$\boxed{0}$1100	E	0$\boxed{0}$101	E	00$\boxed{1}$01	E	001$\boxed{0}$1	E	0010$\boxed{1}$
T	10100	M	$\boxed{0}$1101	G	0$\boxed{0}$111	G	00$\boxed{1}$11	G	001$\boxed{1}$1		
I	01001	I	$\boxed{0}$1001	I	0$\boxed{1}$001	I	01$\boxed{0}$01				
N	01110	N	$\boxed{0}$1110	N	0$\boxed{1}$110	N	01$\boxed{1}$10	L	011$\boxed{0}$0	L	0110$\boxed{0}$
G	00111	G	$\boxed{0}$0111	M	0$\boxed{1}$101	M	01$\boxed{1}$01	M	011$\boxed{0}$1	M	0110$\boxed{1}$
E	00101	E	$\boxed{0}$0101	L	0$\boxed{1}$100	L	01$\boxed{1}$00	N	011$\boxed{1}$0	N	0111$\boxed{0}$
X	11000	A	$\boxed{0}$0001	O	0$\boxed{1}$111	O	01$\boxed{1}$11	O	011$\boxed{1}$1	O	0111$\boxed{1}$
A	00001	X	$\boxed{1}$1000	S	1$\boxed{0}$011	S	10$\boxed{0}$11	P	100$\boxed{0}$0		
M	01101	T	$\boxed{1}$0100	T	1$\boxed{0}$100	R	10$\boxed{0}$10	R	100$\boxed{1}$0	R	1001$\boxed{0}$
P	10000	P	$\boxed{1}$0000	P	1$\boxed{0}$000	P	10$\boxed{0}$00	S	100$\boxed{1}$1	S	1001$\boxed{1}$
L	01100	R	$\boxed{1}$0010	R	1$\boxed{0}$010	T	10$\boxed{1}$00				
E	00101	S	$\boxed{1}$0011	X	1$\boxed{1}$000						

RadixExchangeSort („Links nach rechts"-Radixsort)

A	00001	R	1001$\boxed{0}$	T	101$\boxed{0}$0	X	11$\boxed{0}$00	P	1$\boxed{0}$000	A	$\boxed{0}$0001
S	10011	T	1010$\boxed{0}$	X	110$\boxed{0}$0	P	10$\boxed{0}$00	A	0$\boxed{0}$001	A	$\boxed{0}$0001
O	01111	N	0111$\boxed{0}$	P	100$\boxed{0}$0	A	00$\boxed{0}$01	A	0$\boxed{0}$001	E	$\boxed{0}$0101
R	10010	X	1100$\boxed{0}$	L	011$\boxed{0}$0	I	01$\boxed{0}$01	R	1$\boxed{0}$010	E	$\boxed{0}$0101
T	10100	P	1000$\boxed{0}$	A	000$\boxed{0}$1	A	00$\boxed{0}$01	S	1$\boxed{0}$011	G	$\boxed{0}$0111
I	01001	L	0110$\boxed{0}$	I	010$\boxed{0}$1	R	10$\boxed{0}$10	T	1$\boxed{0}$100	I	$\boxed{0}$1001
N	01110	A	0000$\boxed{1}$	E	001$\boxed{0}$1	S	10$\boxed{0}$11	E	0$\boxed{0}$101	L	$\boxed{0}$1100
G	00111	S	1001$\boxed{1}$	A	000$\boxed{0}$1	T	10$\boxed{1}$00	E	0$\boxed{0}$101	M	$\boxed{0}$1101
E	00101	0	0111$\boxed{1}$	M	011$\boxed{0}$1	L	01$\boxed{1}$00	G	0$\boxed{0}$111	N	$\boxed{0}$1110
X	11000	I	0100$\boxed{1}$	E	001$\boxed{0}$1	E	00$\boxed{1}$01	X	1$\boxed{1}$000	O	$\boxed{0}$1111
A	00001	G	0011$\boxed{1}$	R	100$\boxed{1}$0	M	01$\boxed{1}$01	I	0$\boxed{1}$001	P	$\boxed{1}$0000
M	01101	E	0010$\boxed{1}$	N	011$\boxed{1}$0	E	00$\boxed{1}$01	L	0$\boxed{1}$100	R	$\boxed{1}$0010
P	10000	A	0000$\boxed{1}$	S	100$\boxed{1}$1	N	01$\boxed{1}$10	M	0$\boxed{1}$101	S	$\boxed{1}$0011
L	01100	M	0110$\boxed{1}$	O	011$\boxed{1}$1	O	01$\boxed{1}$11	N	0$\boxed{1}$110	T	$\boxed{1}$0100
E	00101	E	0010$\boxed{1}$	G	001$\boxed{1}$1	G	00$\boxed{1}$11	O	0$\boxed{1}$111	X	$\boxed{1}$1000

StraightRadixSort („Rechts nach Links"-RadixSort)

Zusammenfassung

Tabelle der Sortierverfahren:

- Zeit- und Platz-Komplexität im Worst und Average Case
- Vor- und Nachteile

$T(n)$: Einheitskostenmaß bei Umsetzung in RAM-Programme [M'88]

SelectionSort	(p.40)	$= 2.5n^2 + 3(n+1)\,\mathrm{ld}\,n + 4.5n - 4$
QuickSort	(p.47)	$= 9(n+1)\,\mathrm{ld}\,(n+1) + 29n - 33$
HeapSort	(p.51)	$= 20\,\mathrm{ld}\,n - n - 7$
MergeSort	(p.58)	$= 12n\,\mathrm{ld}\,n + 40n + 97\,\mathrm{ld}\,n + 29$

MergeSort:

- worst case: $n\,\mathrm{ld}\,n$
- zusätzlicher Speicherplatz $O(n)$, nicht insitu, aber sequentiell

3 Suchen in Mengen

3.1 Problemstellung

Gegeben:
Eine Menge von Records (Elementen), von denen jeder aus einer Schlüssel-Komponente und weiteren Komponenten besteht. In der Regel werden Duplikate ausgeschlossen, wobei sich der Begriff Duplikat beziehen kann:

1. auf den Schlüssel
2. auf den vollen Record (exakt: keine Menge mehr)

typische Aufgabe:

- Finde zu einem vorgegebenen Schlüsselwert den Record und führe gegebenenfalls eine Operation aus.
- Damit sind (u.U.) alle Operationen auf Mengen wünschenswert.
- Die Darstellung als Menge und deren Verarbeitung kommt oft vor (Datenbanken!).

Im folgenden betrachten wir primär das (in der Literatur fast immer sogenannte) *Dictionary-Problem* (Wörterbuch-Problem)

Notation: etwa wie Mehlhorn [M'88]

Universum: U := Menge aller möglichen Schlüssel Menge:
$S \subseteq U$

Wörterbuch-Operationen:

Search(x, S):	Falls $x \in S$, liefere den vollen zu x gehörigen Record. (oder Information oder Adresse) sonst Meldung: „$x \notin S$"
Insert(x, S):	Füge Element x zur Menge S: $S := S \cup \{x\}$ (Fehlermeldung: falls $x \in S$)
Delete(x, S):	Entferne Element x aus der Menge S: $S := S \backslash \{x\}$ (Fehlermeldung: falls $x \notin S$)

Hinweis: Die Terminologie ist nicht standardisiert:
z.B.: Search = Member = Contains = Access

Weitere Operationen:

Order(k, S):	Finde das k-te Element in der geordneten Menge S
List(S):	Produziere eine geordnete Liste der Elemente der Menge S
FindMin(S):	:= Order$(1, S)$
Initialize(S):	Initialisiere die Darstellung, d.h. $S := \{\ \}$

Weitere Operationen auf Mengen, die prinzipiell in Frage kommen:
Seien S, A Teilmengen von U:

Find(x, S):	finde die Menge S, zu der das Element x gehört
Union(S, A) (Join):	$S := S \cup A$
Intersection:(S, A)	$S := S \cap A$
Difference(S, A):	$S := S \backslash A$

Seien A_k Teilmengen von U:

Find(x, A_k):	Finde die Menge A_k, zu der das Element x gehört

Bemerkung: Hier wird nicht berücksichtigt, ob das Universum groß oder klein ist.

Beispiele [M'88, p. 97]:

- Symboltabelle (Compiler): 6 Character (z.B. Buchstaben + Ziffern):
  $|U| = (26 + 10)^6 = 2.2 * 10^9$
- Autorenverzeichnis einer Bibliothek in lexikograph./alphabetischer Ordnung
  $|U|$ = ähnliche Größenordnung
- Konten einer Bank (6-stellige Konto-Nummer, 50% davon tatsächlich genutzt)
  $|U| = 10^6$

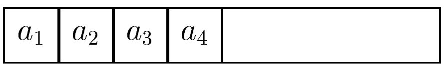
Hier sind die Größe des Universums und die Menge der benutzten Schlüssel gleich groß.

3.2 Einfache Implementierungen

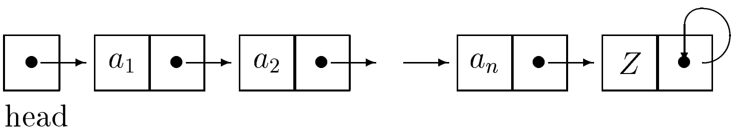
3.2.1 Ungeordnete Arrays und Listen

Darstellung: vgl. Kap. 1.3

- Liste im Array



- verkettete Liste



3.2.2 Vergleichsbasierte Methoden

Annahme einer Vergleichsoperation:

Die Annahme einer Ordnung ist keine wirkliche Einschränkung. Es gibt immer eine Ordnung auf der internen Darstellung der Elemente von U. [M'88 p.139]

Speziell Suchverfahren für

- Daten im geordneten Array S mit $S[i] < S[i+1]$ für $0 < i < n$
- wir identifizieren: $x_i = S[i]$

Wir unterscheiden:

- sequentielle (lineare) Suche (auch ohne Ordnung möglich)
- Binärsuche
- Interpolationssuche

Deklaration:
S : **array** [1..n] **of** element;
S[0] = - ∞; S[n+1] = + ∞

```
procedure Search(a,S);
begin
    low := 1; high := n;
    next := „an element in [low..high]";
    while (a <> S[next]) and (high > low) do
    begin
        if a > S[next]
        then high := next - 1
        else low := next + 1;
        next := „an element in [low...high]"
    end
    if a = S[next]
    then „Found = next"
    else „Not found";
    return
end;
```

Varianten der Anweisung: $next$:= „an element in [low...$high$]"

1. Lineare Suche: $next := low$
2. Binärsuche:
$$next := \left\lceil \frac{high + low}{2} \right\rceil$$
3. Interpolationssuche: (lineare Interpolation)
$$next := (low - 1) + \left\lceil (high - low + 1) \cdot \frac{a - S[low - 1]}{S[high + 1] - S[low - 1]} \right\rceil$$

Komplexität: Zahl der Vergleiche

1. lineare Suche: $O(n)$
   zwischen 1 und n Operationen:

   (a) ungeordneter Array/Liste:
      - $n/2$: erfolgreich
      - n: erfolglos (jedes Element muß geprüft werden)

   (b) geordnete Liste:
      - $n/2$: erfolgreich
      - $n/2$: erfolglos

2. Binärsuche: $O(\text{ ld } n)$
   Rekursionsgleichung:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ \Rightarrow T(n) &< \text{ld}\, n + 1 \end{aligned}$$

   immer weniger als $\text{ld}\, n + 1$ Vergleiche

3. Interpolationssuche: (ohne Beweis)
   - average-case: $\text{ld}\,(\,\text{ld}\, n) + 1$
   - worst-case: $O(n)$ (Entartung in lineare Suche)

Korrektheit des Programms

Das folgende Prädikat P ist eine Invariante der While-Schleife:

$$P \equiv (a \in S \Rightarrow a \in S[low..high]) \wedge (low \leq high \Rightarrow low \leq next \leq high)$$

Beweis:

1. vor der Schleife: P ist erfüllt

2. in der Schleife:
   es gilt $a \neq S[next]$ und also

   (a) entweder $a < S[next]$
   $\Rightarrow a \notin S[next..high]$ (wegen Ordnung!)
   und somit: $a \in S \Rightarrow a \in S[low..next-1]$

   (b) oder $a > S[next]$: analoge Behandlung

   Falls $low \leq high$ gilt, folgt:
   $low \leq next \leq high$ (wegen der ± 1 Änderung)

3. Falls also die Schleife terminiert, gilt P und
   (a) entweder $a = S[next]$: Suche erfolgreich
   (b) oder $low \geq high$
   Sei nun $a \neq S[next]$. Weil P gilt, folgt aus $a \in S[1..n]$, daß $a \in S[low..high]$.
   Nun $low \geq high$:
      i. Falls $high < low$: dann ist $a \notin S[1..n]$
      ii. Falls $high = low$: dann ist $next = high$ wegen P (insbesondere wegen $a \neq S[next]$) und somit $a \notin S[1..n]$

   in beiden Fällen: Suche erfolglos
4. Schleife terminiert:
   In jedem Durchlauf wird $high - low$ mindestens um 1 verringert.

Vergleich der Komplexitäten: $O(f(n))$

(falls möglich: Binärsuche)

	Search	Insert	Delete	Platz
ungeordnete Liste	n	$1'$	n	n
ungeordnetes Array	n	$1'$	n	n
Bitvektor	1	1	1	n

Anmerkungen:

- $1'$ erfordert mit Duplikateneliminierung $O(n)$
- Bitvektor:
  - Operationen $O(1)$ gut, aber:
  - Initialize $= O(N)$
  - Platz $= O(N)$
- Ideal wäre: 3 Operationen: O(1) und Platz: $O(n)$

3.2.3 Kleines Universum

- Bitvektoren
- spezielle Array-Implementierung

Annahme:

$N = |U| =$ vorgegebene maximale Anzahl von Elementen
$S \subset U = \{0, 1, ..., N-1\}$

Methode: Schlüssel = Index im Array

- Bitvektor (auch: charakteristische Funktion; Array of Bits)

  $Bit[i] = 1 : i \in S$
  $Bit[i] = 0 : i \notin S$

- spezielle Array-Implementierung [M'88 p. 270]

  Prinzip: Bitvektor-Darstellung mit zwei Hilfsarrays ohne $O(N)$-Initialisierung

  andere Namen:

  - lazy initialization
  - invertierte Liste
  - array/stack-Methode

  Deklarationen:

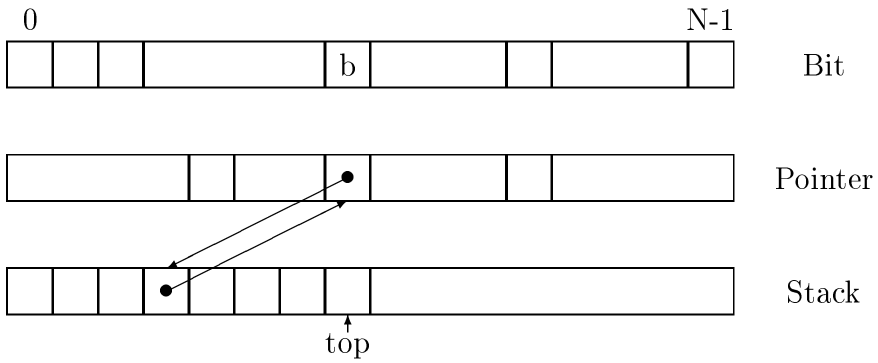
Bit[0..N-1]	: **array of boolean**	
Ptr[0..N]	: **array of** [0..N-1]	„Pointer"
Stk[0..N]	: **array of** [0..N-1]	„Stack"

Das Konzept beruht auf Invarianten: $i \in S$ genau dann, wenn

1. $Bit[i] = 1$
2. $0 \le Ptr[i] \le top$
3. $Stk[Ptr[i]] = i$

wobei anfangs: $top = -1$

Durch den Zähler *top* und den Backpointer in $Stk[.]$ wird die Initialisierung ersetzt.



Operationen:

function Search(i, S): **integer**;
begin
    if (0 ≤ Ptr[i] ≤ top) **and** (Stk[Ptr[i]] <> i)
    then return Bit[i]
    else return „i ∉ S and not initialized"
end;

Anmerkung: übliche Problematik des strikten (conditional) „and"

Insert(i, S) und Delete(i, S) werden realisiert durch:

```
setze Bit[i] = b : b = 0: Delete(i,S)
                   b = 1: Insert(i,S)

      if (0 ≤ Ptr[i] ≤ top) and (Stk[Ptr[i]] <> i)
      then Bit[i] := b
      else begin
            top := top + 1;
            Ptr[i] := top;
            Stk[top] := i;
            Bit[i] := b
      end;
      return;
```

Anmerkungen:

– Die Variable *top* gibt die maximale Anzahl der jemals angesprochenen Elemente von $S \subset U$ an. Bei der Delete-Operation wird nichts zurückgesetzt außer $Bit[i]$.

– Statt des Bitvektors $Bit[0..N-1]$ könnte die Information auch in $Ptr[0..N]$ gepackt werden. (Information von 1 Bit)

– Die Variable $Bit[i]$ kann zu einem Pointer auf einen Speicherbereich umfunktioniert werden, wo der gesamte Record gespeichert ist.

– Die Methode funktioniert auch für das Initialisieren großer Arrays in numerischen Rechnungen.

Komplexitäten

– Initialize(S): $O(1)$

– Search(i, S): $O(1)$

– Insert(i, S): $O(1)$

– Delete(i, S): $O(1)$

– Platz: $O(N)$; 3 Arrays

– Enumerate(S): $O(top)$, wobei $1 \leq top \leq N$

(nur effizient, falls nicht zuviele Löcher in $Stk[0..N]$ durch Delete-Operationen)

3.3 Hashing

- offen
- geschlossen/ ideal geschlossen
- Kollisionsstrategien:
  - lineares Sondieren
  - quadratisches Sondieren
  - Doppelhashing
- Hash-Funktionen
- erweiterbares Hashing

Ausgangspunkt:

Bei BucketSort wurde aus dem Schlüssel direkt die Speicher-Adresse berechnet ebenso wie bei der Bitvektor-Darstellung einer Menge.

Hashing kann man als Erweiterung dieser Methode interpretieren, indem die Speicher-Adresse nicht mehr eindeutig umkehrbar (auf den Schlüssel) sein muß, und somit Mehrfach-Belegungen der Speicher-Adresse zulässig sind ('Kollisionen').

Prinzip: Hash-Funktion und -Tabelle
deutsch: Schlüssel-Transformation, Streuspeicherung

Das Universum $U = \{0, ..., N-1\}$ ist der Wertebereich der Schlüssel x.
Hash-Funktion:

$$\begin{aligned} h: U &\rightarrow \{0, 1, ..., m-1\} \\ x &\rightarrow h(x) \end{aligned}$$

Hash-Tabelle: T: **array** $[0..m-1]$ **of** element;

Prinzip des Hashing ('Streuspeicherung'): $x \in S$ wird in $T[h(x)]$ gespeichert.

Die Operation „Search (x, S)" hat dann 2 Teile:

1. berechne $h(x)$
2. suche x in $T[h(x)]$

Problem:

- Kollision: $h(x) = h(y)$ für $x \neq y$.
- $x \in S$ wird dann nicht notwendigerweise in $T[h(x)]$ selbst gespeichert. $T[h(x)]$ dient dann u. U. als Verweis auf eine weitere Adresse.

Typischer Fall für Hashing: (Symboltabelle für Compiler)

U ist die Menge aller Zeichenketten mit der maximalen Länge 20 (d.h. z.B. Namen):
Abschätzung: $|U| = (26 + 10)^{20} = 1.3 \cdot 10^{31}$
Somit ist keine umkehrbare Speicherfunktion realistisch.

Beispiel: Verteilen von Namen über 17 Behälter: [G'92 p. 97]

Zeichenketten $c = c_1 .... c_k$

$$h(c) := \sum_i N(c_i) \bmod m$$

mit $N(A) = 1$, $N(B) = 2$, $N(C) = 3$, ..., $N(Z) = 26$

weitere Vereinfachung: nur die ersten 3 Buchstaben:

$$h(c) = [N(c_1) + N(c_2) + N(c_3)] \bmod 17$$

speziell: $m = 17$; $S =$ (deutsche) Monatsnamen

0:	November
1:	April, Dezember
2:	März
3:	
...	
6:	Mai, September
7:	
8:	Januar
9:	Juli
10:	
11:	Juni
12:	August, Oktober
13:	Februar
14:	
15:	
16:	

Beachte: Es gibt 3 Kollisionen.

Wahrscheinlichkeit von Kollisionen

Analogie zum Geburtstagsproblem (-paradoxon):
Wie groß ist die Wahrscheinlichkeit, daß mindestens 2 von n Leuten am gleichen Tag Geburtstag haben ($m = 365$) ?

Analogie:
$m = 365$ Tage = Größe der Hash-Tabelle;
n Personen = Zahl der Elemente

Annahme: ideale Hash-Funktion d.h. gleichmäßige Verteilung über die Hash-Tabelle

$$Pr(Kol|n,m) = 1 - Pr(NoKol|n,m)$$

$p(i;m) :=$ Wahrscheinlichkeit, daß der i-te Schlüssel auf einen freien Platz abgebildet wird ($i = 1, ..., n$), wie alle Schlüssel zuvor.

$$\begin{aligned}
p(1;m) &= \frac{m-0}{m} = 1 - \frac{0}{m} && \text{, weil 0 Plätze belegt und } m-0 \text{ Plätze frei sind} \\
p(2;m) &= \frac{m-1}{m} = 1 - \frac{1}{m} && \text{, weil 1 Platz belegt und } m-1 \text{ Plätze frei sind} \\
&\dots \\
p(i;m) &= \frac{m-i+1}{m} = 1 - \frac{i-1}{m} && \text{, weil } i-1 \text{ Plätze belegt und } m-i+1 \text{ Plätze frei sind}
\end{aligned}$$

$Pr(NoKol|n,m)$ ist dann das Produkt der Wahrscheinlichkeiten $p(1;m)$ bis $p(n-1;m)$

$$\begin{aligned}
Pr(NoKol|n,m) &= \prod_{i=0}^{n-1} p(i;m) \\
&= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right)
\end{aligned}$$

Tabelle zum Geburtstagsproblem $m = 365$ [Schülerduden p.141]

n	$Pr(Kol\|n,m)$
10	.11695
20	.41144
22	.47570
23	.50730
24	.53835
30	.70632
40	.89123
50	.97037

Approximation

Frage: Wie soll m mit n wachsen, um $Pr(Kol|n,m)$ konstant zu halten ?

Die obige Formel gibt auf diese Frage nur eine indirekte Antwort.
Deswegen die folgende Vereinfachung:

$$\begin{aligned}
Pr(NoKol|n,m) &= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) \\
&= \exp\left[\sum_{i=0}^{n-1} \log\left(1 - \frac{i}{m}\right)\right]
\end{aligned}$$

$$\begin{aligned}
&= \dots \text{ verwende: } \log(1+\varepsilon) \simeq \varepsilon \text{ für } \varepsilon \ll 1 \text{ d.h. } n \ll m \\
&\simeq \exp\left[-\sum_{i=0}^{n-1} \frac{i}{m}\right] \\
&= \exp\left[-\frac{n(n-1)}{2m}\right] \\
Pr(NoKol|n,m) &\simeq \exp\left[-\frac{n^2}{2m}\right]
\end{aligned}$$

Ergebnis:
$Pr(NoKol|n,m)$ bleibt etwa konstant, wenn m (=Größe der Hash-Tabelle) quadratisch mit n (=Zahl der Elemente) wächst.

Übungsaufgabe: Approximation, in der nur „paarweise" Kollisionen betrachtet werden.

Anforderungen an Hash-Funktion:

- surjektiv, d.h. die ganze Hash-Tabelle wird abgedeckt
- möglichst „gleichmäßige" Verteilung über die Hash-Tabelle
- effizient zu berechnen

Das Hauptproblem besteht in der Behandlung von Kollisionen. Es gibt verschiedene Methoden:

- Hashing mit Verkettung (offenes Hashing)
- Hashing mit offener Adressierung (geschlossenes Hashing)

Warnung: Namenswirrwarr, z.T. wegen anderer Terminologie:

Mehrere Speicherzellen werden zu Behältern zusammengefaßt.

- offenes Hashing:
  Jeder Behälter kann beliebig viele Schlüssel aufnehmen (verkettete Liste: dynamisch).
- geschlossenes Hashing:
  Jeder Behälter kann nur eine kleine Zahl b Schlüssel aufnehmen, sonst tritt ein Überlauf auf. (Array: statisch)

Typische Hash-Funktion [G'92 p.109]

Sei $x \in \mathbb{N}$, etwa nach Umsetzung aus Buchstaben.

- Divisionsmethode: Sei m die Größe der Tabelle:

  $$h(x) = x \bmod m \qquad (m \text{ meist Primzahl})$$

  Vorteil: einfach zu berechnen
  Nachteil:
  $h(x+i) = h(x) + i$: Für geeignete x und i werden aufeinanderfolgende Schlüssel auf aufeinanderfolgende Zellen abgebildet.

- Mittel-Quadrat-Methode:

  $$h(x) = \text{mittlerer Block von Ziffern von } x^2$$

  Der mittlere Block hängt von allen Ziffern von x ab, und deswegen wird eine bessere Streuung erreicht.

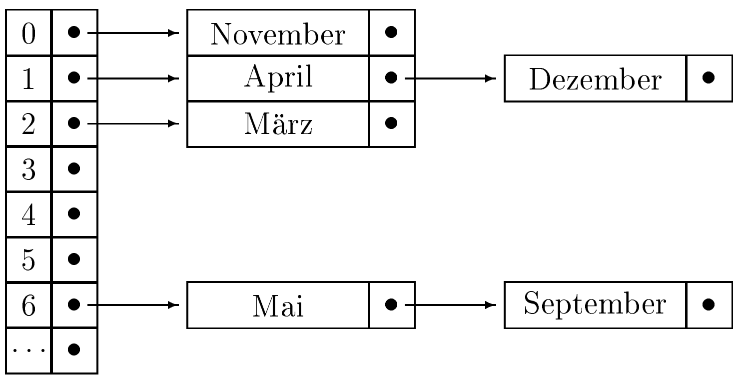
x	$x \bmod m$	x^2	$h(x)$
127	27	$16 \mid 12 \mid 9$	12
128	28	$16 \mid 38 \mid 4$	38
129	29	$16 \mid 64 \mid 1$	64

Hashing mit Verkettung (separate chaining) [G'92 p.99]

(offenes Hashing: verkettete Liste: dynamisch)
Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt. Ein Array von Zeigern verwaltet die Behälter.

var HashTable : **array** [0..m-1] **of** ↑ListElem

Hier: $m = 17$, März = Maerz



Operationen:

- Insert(x, S)
- Delete(x, S)
- Search(x, S) = durchlaufen der Liste $HashTable[h(x)]$

durchschnittliche Listenlänge: $\frac{n}{m}$

Kostenabschätzung (exakte Analyse mühsam):
,,Behälter finden" + ,,Liste durchlaufen"

- average case: $O\left(\frac{1+n}{2m}\right)$
- worst case: $O(n)$
- Platz: $O(n+m)$

Verhalten:

$$\frac{n}{m} \ll 1 : \text{mehr als Array-Adressierung}$$
$$\frac{n}{m} \simeq 1 : \text{ Übergangsbereich}$$
$$\frac{n}{m} \gg 1 : \text{mehr als verkettete Liste}$$

Hashing mit offener Adressierung [G'92 p.100; M'88 p.117]

auch ,,closed hashing": Alle Daten werden in der Hash-Tabelle selbst gehalten.

- allgemein: m Behälter mit b Zellen
- speziell: $b = 1$

statischer Array:
var HashTable : **array** [0..m-1] **of** Elem;

Kollisionsbehandlung mittels Rehashing = offene Adressierung:

- jedes $x \in U$ definiert eine Folge $h(x, j)$, $j = 0, 1, 2, ...$ von Positionen in der Hash-Tabelle.
- diese Folge muß bei jeder Operation durchsucht werden.
- Definition der zusätzlichen Hash-Funktionen $h(x, j)$, $j = 0, 1, 2, ...$ erforderlich

Die Operation ,,Delete(x, S)” erfordert eine Sonderbehandlung, weil die Elemente, die an x mittels Rehashing vorbeigeleitet wurden, auch nach Löschen von x noch gefunden werden müssen.

Abhilfe: zusätzlich zu den Werten aus U werden ,,empty” und ,,deleted” eingeführt.

Offene Adressierung: Lineares Sondieren

Die (Re)Hash-Funktionen $h(x, i)$, $i = 0, 1, 2, \ldots$ sollen so gewählt werden, daß für jedes x der Reihe nach sämtliche m Zellen der Hash-Tabelle inspiziert werden.

Einfachster Ansatz: Lineares Sondieren

$$h(x, i) = (h(x) + i) \bmod m \qquad 1 \leq i \leq m - 1$$

Beispiel: Insert(x, S) mit Monatsnamen in alphabetische Reihenfolge

0:	November	
1:	April	(Dezember)
2:	März	$\swarrow$
3:	Dezember	
...		
6:	Mai	(September)
7:	September	$\leftarrow$
8:	Januar	
9:	Juli	
10:		
11:	Juni	
12:	August	(Oktober)
13:	Februar	$\swarrow$
14:	Oktober	
15:		
16:		

Bei annähernd voller Tabelle tendiert lineares Sondieren dazu, lange Ketten von besetzten Zellen zu bilden. (,,Clustering”)

Analyse: geschlossenes Hashing (offene Adressierung)

[M’88 p.118/9], [A’86 p.131], in [G’92 p. 101] unnötig kompliziert: (Binomial-Koeffizienten, ...)

Kosten der Operationen:

- Search(x,S)
- Insert(x,S)
- Delete(x,S)

Dazu benötigen wir Aussagen über die Kollisionswahrscheinlichkeit.

Annahme: ideale Hash-Funktion, d.h. alle Plätze gleichwahrscheinlich und keine Kettenbildung

$q(i;n,m) :=$ Wahrscheinlichkeit, daß mindestens i Kollisionen auftreten, wenn m die Größe der Hash-Tabelle ist und bereits n Elemente eingetragen sind.

$=$ Wahrscheinlichkeit, daß die ersten $i-1$ Positionen $h(x,0)$, $h(x,1)$,..., $h(x,i-1)$ der Rehashing-Strategie besetzt sind (für n und m).

i=1: anfangs:

$$q(1;n,m) = \frac{n}{m}$$

i=2: Wir haben bereits eine Kollision:
Rehashing: teste $m-1$ Zellen, von denen $n-1$ voll sind:

$$q(2;n,m) = \frac{n}{m} \cdot \frac{n-1}{m-1}$$

i: Wir haben bereits $i-1$ Kollisionen:
Rehashing: teste $m-i+1$ Zellen, von denen $n-i+1$ voll sind:
Rekursion:

$$\begin{aligned} q(i;n,m) &= q(i-1;n,m) \cdot \frac{n-i+1}{m-i+1} \\ &= \ldots \\ &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+1}{m-i+1} \\ &= \prod_{j=0}^{i-1} \frac{n-j}{m-j} \end{aligned}$$

Näherung: falls n, $m \gg i$:

$$q(i;n,m) = \left(\frac{n}{m}\right)^i$$

$C_{Ins}(n,m) :=$ mittlere Kosten von Insert(x,S), wobei die Tabelle insgesamt m Zellen hat, von denen bereits n besetzt sind (also $(n+1)$-stes Element!).

$$\begin{aligned} C_{Ins}(n,m) &= \sum_{i=0}^{n} q(i;n,m) \\ &= \frac{m+1}{m+1-n} \\ &\simeq \frac{1}{1-a} \qquad a := \frac{n}{m} \text{ (Auslastungsfaktor, Belegungsfaktor)} \end{aligned}$$

Beweis: vollständige Induktion über m und n

$C^-_{Sea}(n,m) :=$ mittlere Kosten von Search(x,S)-erfolglos (bei m Zellen, wovon n belegt)

Ablauf: Die Positionen $h(x,0)$, $h(x,1)$,... werden getestet, und bei der ersten freien Zelle wird abgebrochen. Also:

$$C^-_{Sea}(n,m) = C_{Ins}(n,m)$$

$C^+_{Sea}(n,m) :=$ mittlere Kosten von Search(x,S)-erfolgreich (bei m Zellen, wovon n belegt)

Ablauf: Wir durchlaufen die Positionen $h(x,0)$, $h(x,1)$, ... , bis wir ein i finden mit $T[h(x,i)] = x$. Dasselbe i hat auch die Kosten für Insert(x,S) bestimmt. Also erhält man aus $C_{Ins}(n,m)$ durch Mittelung über alle Elemente $j = 0,...,n$:

$$\begin{aligned}
C^+_{Sea}(n,m) &= \frac{1}{n} \cdot \sum_{j=0}^{n-1} C_{Ins}(j,m) \\
&= \ldots \qquad \text{(Harmon. Zahlen)} \\
&\simeq \frac{1}{a} \log \frac{1}{1-a} \qquad a := \frac{n}{m} \text{ (Auslastungsfaktor)}
\end{aligned}$$

$C_{Del}(n,m) :=$ mittlere Kosten von Delete(x,S)

Ablauf: Die Zellen $h(x,0)$, $h(x,1)$,... werden durchlaufen bis $T[h(x,i)] = x$. Also:

$$C_{Del}(n,m) := C^+_{Sea}(n,m)$$

Näherungen für Kostenanalyse

$$\text{falls } n,\ m \gg 1: \quad q(i;n,m) = \left(\frac{n}{m}\right)^i$$

Damit können wir die Formeln der geometrischen Reihe verwenden und einfacher rechnen:

Zwei Arten von Kosten:

$$\begin{aligned}
C_{Ins}(n,m) &\simeq \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i = \frac{1}{1-\dfrac{n}{m}} = \frac{1}{1-a} \\
C^+_{Sea}(n,m) &\simeq \frac{1}{n} \int_0^{n-1} \frac{m}{m-x}\, dx \\
&= \frac{m}{n} \log \frac{m}{m+1-n} \simeq a \log \frac{1}{1-a}
\end{aligned}$$

Nichtideales Hashing

Nichtideal: Clustering (Kettenbildung) mit linearem Sondieren:

Formeln von Knuth[1973]
Zwei Größen genügen:

$$C^{+}_{Sea}(n,m) = \frac{1}{2}\left[1 + \frac{1}{1-\frac{n}{m}}\right]$$

$$C^{-}_{Sea}(n,m) = \frac{1}{2}\left[1 + \frac{1}{1-\left(\frac{n}{m}\right)^2}\right]$$

Kollisionsstrategien [G'92 p. 107]

- Lineares Sondieren (Verallgemeinerung)

  $$h(x,i) = (h(x) + c \cdot i) \bmod m \qquad 1 \leq i \leq m-1$$

  c ist eine Konstante, wobei c und m teilerfremd sein sollten.
  Keine Verbesserung: Ketten mit Abstand c

- Quadratisches Sondieren (Verallgemeinerung)

  $$h(x,i) = (h(x) + i^2) \bmod m$$

  Warum i^2 ? [Schinzel p. 32]
  Wenn m Primzahl ist, sind die Zahlen $i^2 \bmod m$ alle verschieden für $i = 0, 1, ..., \left\lfloor \frac{m}{2} \right\rfloor$.
  Verfeinerung:

  $$h(x, 2i-1) = (h(x) + i^2) \bmod m \qquad 1 \leq i \leq \frac{m-1}{2}$$
  $$h(x, 2i) = (h(x) - i^2) \bmod m$$

  Wähle m als Primzahl mit $m \bmod 4 = 3$.
  Basis: Zahlentheorie insbesondere [Radke 1970]

  Quadratisches Sondieren:

  - keine Verbesserung für Primärkollisionen: $h(x,0) = h(y,0)$
  - vermeidet Clusterbildung bei Sekundärkollisionen: $h(x,i) \neq h(y,i)$ für $i > 0$

- Doppel-Hashing
  Wähle zwei Hash-Funktionen $h(x)$ und $h'(x)$ und nehme an:

  $$Pr(h(x) = h(y)) = \frac{1}{m}$$
  $$Pr(h'(x) = h'(y)) = \frac{1}{m}$$

Falls die beiden Funktionen h und h' unabhängig sind, gilt:

$$Pr(h(x) = h(y) \wedge h'(x) = h'(y)) = \frac{1}{m^2}$$

Deswegen definieren wir:

$$h(x, i) = (h(x) + h'(x) \cdot i^2) \bmod m$$

Eigenschaften:

- wirklich gut
- kaum unterscheidbar von idealem Hashing

Hashing: Zusammenfassung

- average case: gutes Verhalten
- worst case: $O(n)$

Nachteil: ListOrder(S) wird nicht unterstützt

Anwendungen:

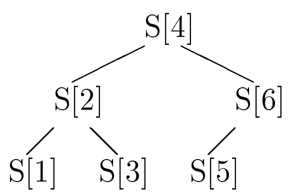
- dynamische: separate chaining
- statisch (Maximum = $|S|$ bekannt) offene Adressierung

3.4 Binäre Suchbäume

[M'88 p. 140] Ausgangspunkt: Binäre Suche

$$next := \left\lceil \frac{high + low}{2} \right\rceil$$

Veranschaulichung durch binären Baum ($n = 6$)



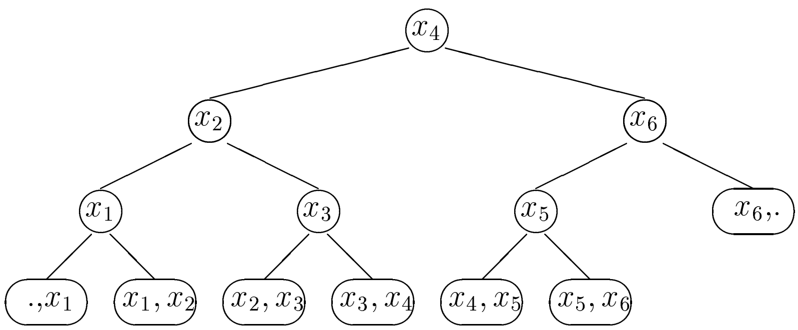
Zeitkomplexität für die Array-Darstellung

- Search:
  - Der Wert von $[high - low + 1]$ wird bei jedem Durchgang der While-Schleife halbiert.
  - höchstens: $\text{ld}\, n$ Zeitkomplexität
- Insert/Delete-Operationen: problematisch
  z.B. erfordert Insert das Verschieben eines Teils des Arrays. Daher: $O(n)$
- Ausweg:
  Der obige Suchbaum wird durch Zeiger statt durch ein Array realisiert, so daß auch die Operationen „Delete(x, S)" und „Insert(x, S)" effizient ausgeführt werden können.

Definition 3.1 (binärer Suchbaum) Ein binärer Suchbaum für die n-elementige Menge $S = \{x_1 < x_2 < ... < x_n\}$ ist ein binärer Baum mit n Knoten $\{v_1...v_n\}$.
Die Knoten sind mit den Elementen von S beschriftet, d.h. es gibt eine injektive Abbildung $Inhalt : \{v_1...v_n\} \to S$. Die Beschriftung bewahrt die Ordnung, d.h. wenn v_i im linken Unterbaum ist, v_j im rechten Unterbaum, v_k Wurzel des Baumes, so gilt: $Inhalt[v_i] < Inhalt[v_k] < Inhalt[v_j]$

Äquivalente Definition:
Ein Durchlauf des Baumes in symmetrischer Ordnung (Inorder) ergibt die Ordnung auf S.

In der Regel identifizieren wir Knoten mit ihrer Beschriftung:
Knoten v mit Beschriftung x = Knoten x

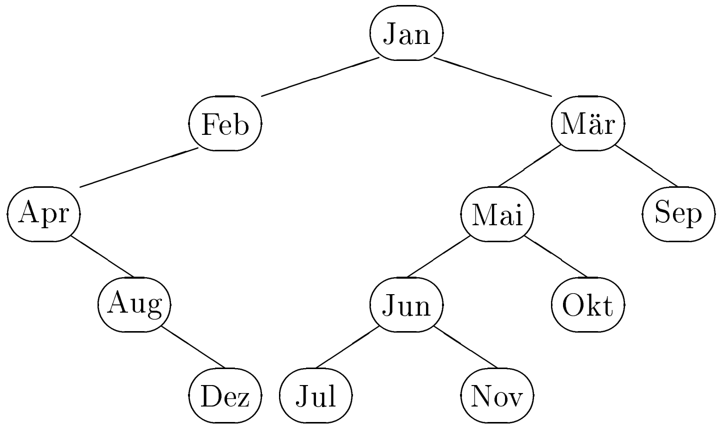


Die $n + 1$ Blätter:

- stellen erfolglose Search-Operationen dar
- brauchen nicht explizit abgespeichert zu werden
- $(x1, x2)$ steht für: $\{y : y \in U : x1 < y < x2\}$
  analog: $y \in (., x1) : -\infty < y < x1$

Beispiel: Monatsnamen (Abkürzungen!) [G'92 p. 111]

Insert-Operationen in der zeitliche Reihenfolge mit lexikographischer Ordnung:



Beachte: Inorder-Traversieren (d.h Traversieren in symmetrischer Ordnung) erzeugt die alphabetische Reihenfolge

Übung: Erstellen Sie den Baum für die umgekehrte Reihenfolge

Programme: Binäre Suchbäume [M'88 p.141]

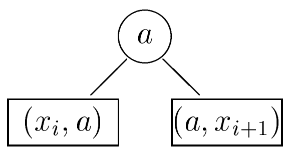
```
procedure Search(a,S);
begin
    v := Root of T;
    while (v is node) and (a <> Inhalt[v]) do
        if (a < Inhalt[v])
        then v = LeftSon[v]
        else v = RightSon[v]
end;
```

Das obige Programm Search endet:

- in einem Knoten v, falls $a = Inhalt[v]$
- in einem Blatt, falls $a \notin S$
  Für das entsprechende Blatt (x_i, x_{i+1}) gilt: $x_i < a < x_{i+1}$

Dann haben wir:

- Insert(a, S): ersetze das Blatt (x_i, x_{i+1}) durch den folgenden Baum:



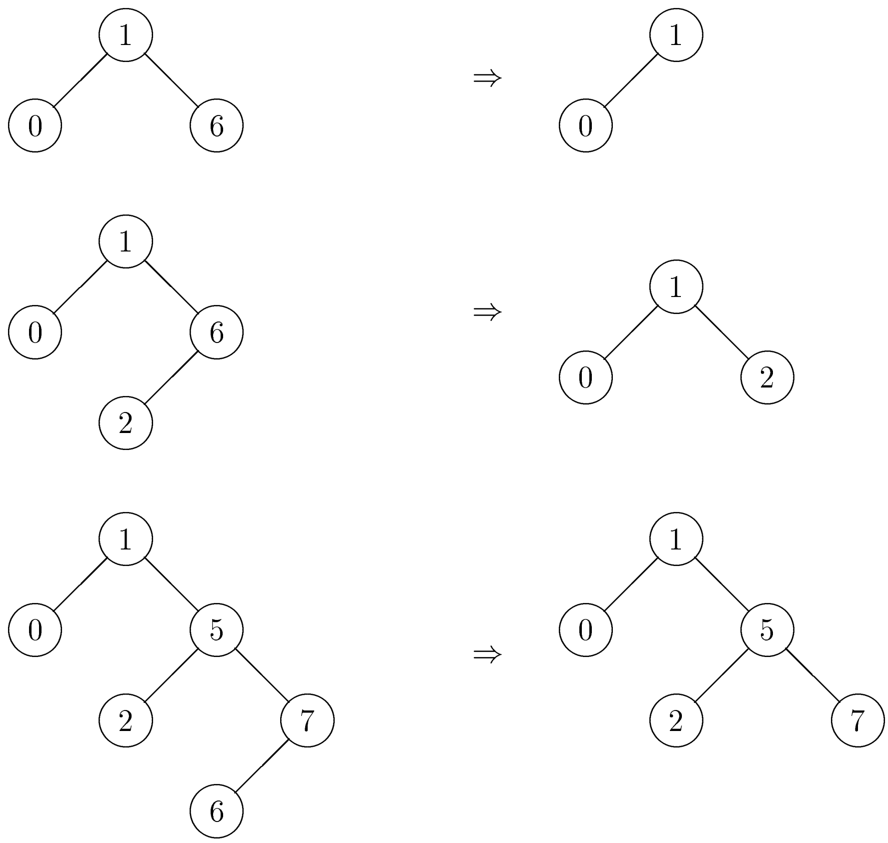
- Delete(a, S) (komplizierter):
  Die Suche ende in einem Knoten v mit $a = Inhalt[v]$
  (sonst: im Blatt Fehlermeldung)
  Fallunterscheidung:

  1. mindestens ein Sohn von v ist ein Blatt, etwa $LeftSon[v]$:
     - ersetze v durch $RightSon[v]$
     - streiche v und $LeftSon[v]$ aus dem Baum.
  2. kein Sohn von v ist ein Blatt:
     - sei w der rechteste Unterknoten im linken Unterbaum von v
       Dieser wird gefunden durch:
       * $LeftSon[v]$
       * rekursiv $RightSon[v]$, bis wir auf ein Blatt treffen.
     - $Inhalt[v] = Inhalt[w]$
     - um w zu entfernen, fahre fort wie in Fall 1.

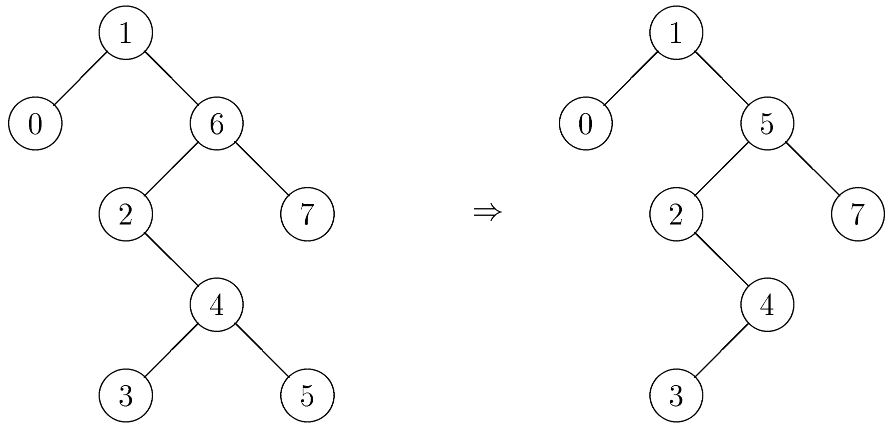
     Dabei bleibt der Suchbaum erhalten!

Beispiel: Delete 6 (ähnlich [M'88 p. 143])

Fall 1.



Fall 2.



Komplexitäten

Operationen: Search, Delete, Insert

- im wesentlichen: Baum-Traversieren u. einige lokale Änderungen, die $O(1)$ Komplexität haben.
- deswegen: insgesamt $O(h(T))$, wobei $h(T)$ die Höhe des Baumes ist.

Operation: ListOrder(S):

- Traversieren in symmetrischer Ordnung
- $O(|S|) = O(n)$

Operation: Order(k, S):

- Zusatz: speichere für jeden Knoten die Anzahl der Knoten des linken Unterbaums.
- dieser Wert kann mit $O(1)$ bei Delete- und Insert-Operationen aktualisiert werden.
- damit haben wir (Übung!) $O(h(T))$

Implementierung: Binärer Suchbaum [S'88 p. 208] wie üblich:

- 3 Operationen
- Initialize(S)
- ListOrder(S)

Beachte Eigenheiten der Implementierung

- Tree Header Node mit rechtem Sohn
- Dummy Node z für tail node (Blätter)

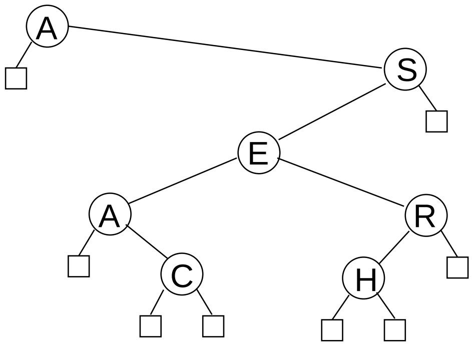


Abbildung 13: Binärer Suchbaum [S'88 p.203]

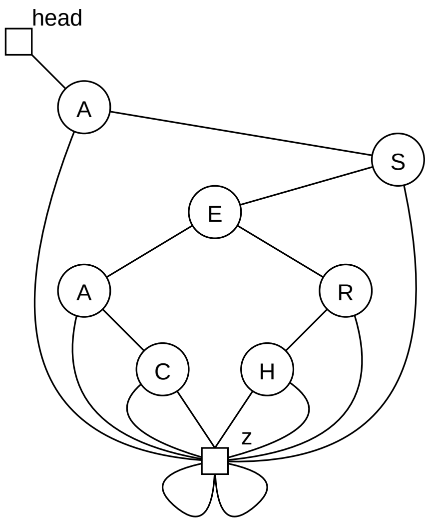


Abbildung 14: Implementation mit Dummy-Knoten [S'88 p.204]

Operationen: im wesentlichen klar, bis auf „Delete(v, S)"

Implementierung:

type link = ↑node;
    node = **record**
        key, info : **integer**;
        l, r :link
        end;
procedure treeinitialize;
begin
    new(z); z↑.l := z; z↑.r := z;
    new(head); head↑.key := 0; head↑.r := z;
end;

Anmerkung: besser $head \uparrow .key = -\infty$

```
function treesearch (v :integer; x: link) : link;
begin
    z↑.key := v;
    repeat
        if v < x↑.key then x := x↑.l else x := x↑.r
    until v = x↑.key;
    treesearch := x
end;

function treeinsert (v : integer; x: link) : link;
var p: link;
begin
    repeat
        p := x;
        if v < x↑.key then x := x↑.l else x := x↑.r
    until x = z;
    new (x); x↑.key := v; x↑.l := z; x↑.r := z;
    if v < p↑.key then p↑.l := x else p↑.r := x;
    treeinsert := x
end;

procedure treeprint (x: link); (* inorder *)
begin
    if x <> z
    then begin
        treeprint(x↑.l);
        printnode(x);
        treeprint(x↑.r)
    end
end;
```

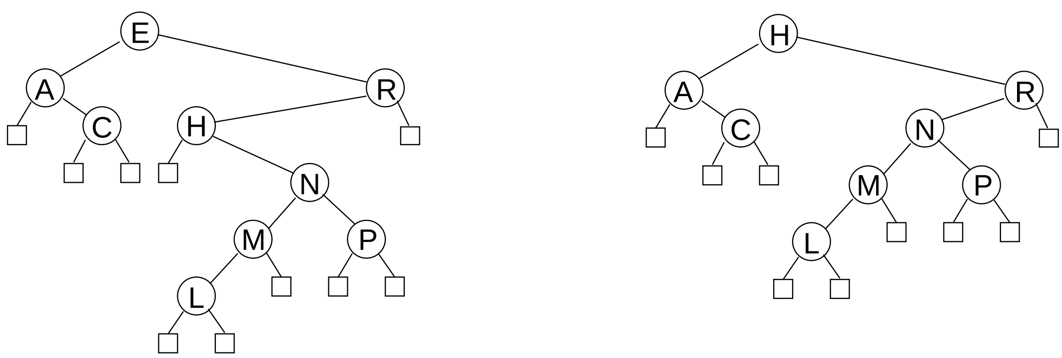


Abbildung 15: Beispiel [S'88 p.209] Delete E

1. Suche den kleinsten Schlüssel im rechten Teilbaum von E. Das ist H (hat keinen linken Sohn)
2. Der linker Zeiger des Vaters von H (=R) wird auf rechten Sohn von H (=N) umdirigiert
3. Söhne von E := Söhne von H (Umdirigieren der Zeiger)
4. Vater von E : = Vater von H (Umdirigieren der Zeiger)

```
procedure treedelete (t, x : link);
var p, c : link;
begin
    repeat
        p := x;
        if t↑.key < x↑.key then x := x↑.l else x := x↑.r
    until x = t;
    if t↑.r = z
    then x := x↑.l
    else
        if t↑.r↑.l = z
        then begin
            x := x↑.r; x↑.l := t↑.l
        end
        else begin
            c := x↑.r;
            while c↑.l↑.l <> z do c := c↑.l;
            x := c↑.l; c↑.l := x↑.r;
            x↑.l := t↑.l; x↑.r := t↑.r
        end;
    if t↑.key < p↑.key then p↑.l := x else p↑.r := x;
    dispose(t) end;
```

Löschen des Knotens $t\uparrow$ im Teilbaum $x\uparrow$

- unsymmetrisch: immer Analyse des rechten Teilbaums
- Variable:
  - p = Parent of x (tree to be searched)
  - c = Son of $t\uparrow$ (to be deleted)
- Ergebnis: x = Son of $p\uparrow$

- Schritte:
  1. Suchen des Knotens t im Teilbaum x
  2. Fallunterscheidung für t
     (a) Kein rechter Sohn (C, L, M, P, R)
     (b) Rechter Sohn ohne linken Sohn (A, N)
     (c) sonst: (E, H)
        - suche kleinsten Schlüssel im rechten Teilbaum von t (Name: c) (beachte: c hat keinen linken Sohn)
        - setze Zeiger um (4 Stück!)
  3. setze Zeiger für Knoten $x \uparrow$

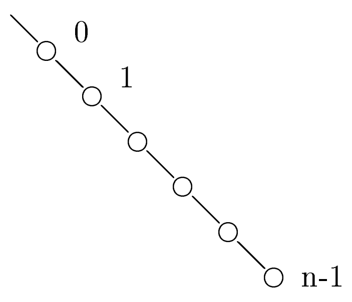
Realisierung der (3+2) Operationen:

Initialize(S):	TreeInitialize()
Search(v,S):	y := TreeSearch(v,head)
Insert(v,S):	y := TreeInsert(v,head)
Delete(v,S):	TreeDelete(TreeSearch(v,head),head)
ListOrder(S):	TreePrint(head↑.r)

Genauere Analyse:
Konstruktion des binären Suchbaums durch n Insert-Operationen

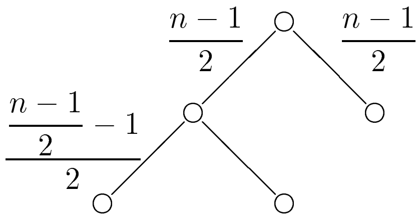
$c(n)$ = Zahl der Vergleiche

1. Worst-case: entarteter Baum



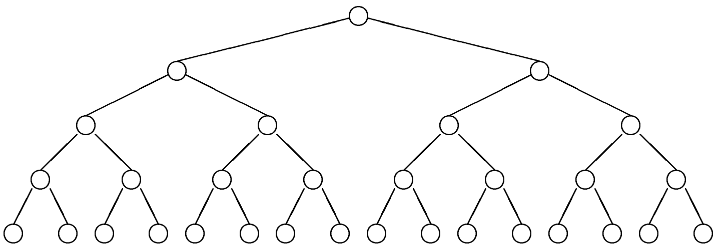
$$\begin{aligned} C(n) &= \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \\ \frac{C(n)}{n} &= \frac{n-1}{2} \end{aligned}$$

2. best case: $n = 2^h - 1$



$$C(n) = n - 1 + 2 \cdot C\left(\frac{n-1}{2}\right)$$

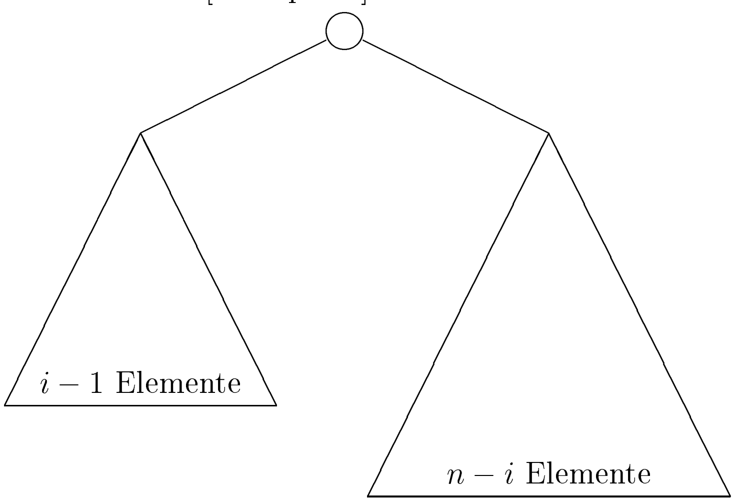
Best Case: $n = 2^h - 1$, $n \in \mathbb{N}$



	Σ
$1 \cdot 2^1 = 2$	2
$2 \cdot 2^2 = 8$	10
$3 \cdot 2^3 = 24$	34
$4 \cdot 2^4 = 64$	98
$\dots$	
$i \cdot 2^i$	

$$\begin{aligned} C(n) = \sum_{i=0}^{h-1} i \cdot 2^i &= (h-2)2^h + 2 \\ &= (n+1)(\underbrace{\operatorname{ld}(n+1)}_{=h} - 2) + 2 \end{aligned}$$

3. average case für n Elemente [A'74 p.118]



$C(n)$: $n-1$ Vergleiche für die Wurzel (alle Elemente müssen an der Wurzel vorbei)
$+C(i-1)$ linker Teilbaum
$+C(n-i)$ rechter Teilbaum

Jede Aufteilung $i = 1, ..., n$ ist gleichwahrscheinlich (alle Permutationen der Menge $\{x_1, ..., x_n\}$)

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{i=1}^{n}[n-1+C(i-1)+C(n-i)] \\ &= n-1+\frac{2}{n}\sum_{i=0}^{n-1}C(i) \qquad \text{wie Quicksort} \\ &= 2(\ln 2)n \operatorname{ld} n + ... \\ &= 1.386 n \operatorname{ld} n + ... \end{aligned}$$

Aufwand:

$$\frac{C(n)}{n} \simeq 1.386 \operatorname{ld} n$$

Optimaler binärer Suchbaum [M'88 p. 144]

Diese Bäume sind gewichtet mit der Zugriffsverteilung, die die Häufigkeit (oder Wichtigkeit) der Elemente von S widerspiegelt.

Beachte: nur „Search(a, S)" wird betrachtet.

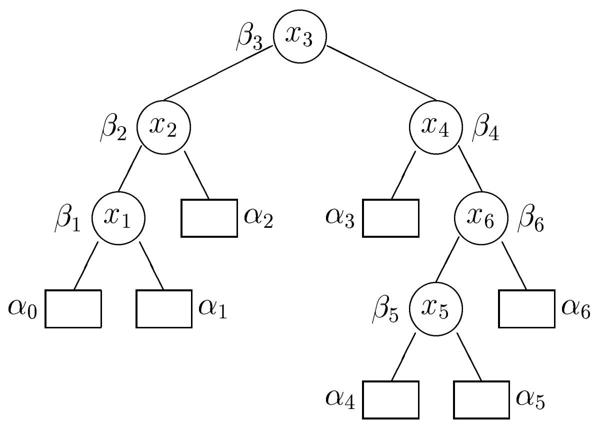
Definition 3.2 (Zugriffsverteilung) Sei $S = \{x_1 < x_2 < ... < x_n\}$ und 2 Sentinels x_0 und x_{n+1} mit $x_0 < x_i < x_{n+1}$ für $i = 1, ..., n$
Search(a, S): erfolgreich: $a = x_i$: Wahrscheinlichkeit: $\beta_i \geq 0$
Search(a, S): erfolglos: $x_j < a < x_{j+1}$: Wahrscheinlichkeit: $\alpha_j \geq 0$
Normierte Verteilung:

$$\sum_{i=1}^{n}\beta_i + \sum_{j=0}^{n}\alpha_j = 1$$

Zugriffsverteilung: $(2n+1)$-Tupel von Wahrscheinlichkeiten $(\alpha_0, \beta_1, \alpha_1, \beta_2, \alpha_2, ..., \alpha_n, \beta_n)$

Beachte: Die Normierung ist zwar sinnvoll, wird aber im folgenden nicht benötigt.

Beispiel [Aigner p.187]



Komplexität von Search(a, S)

average case: mittlere Zahl von Vergleichen
Die Suche endet in einem inneren Knoten oder einem Blatt. Die Höhe eines Knotens oder eines Blatts gibt direkt die Zahl der Vergleiche.
Sei ein Suchbaum gegeben mit:

b_i: Tiefe des Knotens x_i

a_j: Tiefe des Blattes (x_j, x_{j+1})

Sei P („Pfadlänge") die mittlere Anzahl von Vergleichen, (auch gewichtete Pfadlänge):

$$P = \sum_{i=1}^{n} (\beta_i \cdot (1 + b_i)) + \sum_{j=0}^{n} (\alpha_j \cdot a_j)$$

Beachte: Ein Blatt erfordert einen Vergleich weniger als ein Knoten.

Ziel: Konstruktion des optimalen Suchbaums, der bei gegebener Zugriffsverteilung die Pfadlänge minimiert.

Lösung: Dynamische Programmierung (Bellman/Knuth-Algorithmus)
[Ottmann p.397]

Search(v, S):

Angenommen, wir vergleichen v mit einem Knoten x_k, der die Teilmenge (oder den Teilbaum) $\{x_i, ..., x_j\}$ darstellt.
Der binäre Teilbaum mit Wurzel x_k definiert eine Zerlegung:

$$\{x_i, ..., x_{k-1}\} \cup \{x_k, ..., x_j\}$$

Aufgrund der Additivität der Pfadlänge muß auch jeder der beiden Teilbäume optimal bzgl. der (Teil-)Pfadlänge sein.

Um die dynamische Programmierung anzuwenden, definieren wir die Hilfsgröße $c(i, j)$:

$$c(i, j) := \text{ optimale Pfadlänge für den Teilbaum } \{x_i, ..., x_j\}$$

Dann kann man im Fall $i < j$ aufgrund der Definition $c(i, j)$ in 2 Teilbäume zerlegen, die ihrerseits wiederum optimal sein müssen.
Daher gilt die Gleichung:

$$\begin{aligned} c(i, i) &= 0 \qquad \text{für alle } i \\ c(i, j) &= w_{ij} + \min_{i<k\leq j} [c(i, k-1) + c(k, j)] \text{ für } i < j \end{aligned}$$

Diese Gleichung ist iterativ, nicht rekursiv zu lösen. (DP-Gleichung; Gleichung der dynamischen Programmierung; Bellmansche Optimalitätsgleichung):
gesuchtes Ergebnis: $P = c(0, n)$

Programm-Entwurf (ähnlich wie DP für Matrix-Produkt)

Deklaration:
c: **array** [0..n][0..n] **of real**
r: **array** [0..n][0..n] **of** [0..n]

1. berechne w_{ij} vorab;
   initialisiere $c(i,i) = 0$

2. **for** $k = 0, ..., n-1$ **do**
   for $i = 1, ..., n-k$ **do**

$$\begin{array}{rcll} c(i,j) & = & w_{ij} + \min\limits_{i<k\leq j}[c(i,k-1) + c(k,j)] & i<j \\ r(i,j) & = & \arg\min\limits_{i<k\leq j}[c(i,k-1) + c(k,j)] & i<j \end{array}$$

3. Rekonstruktion des optimalen Suchbaums mittels $r(i,j)$

daher:
Zeit: $O(n^3)$
Platz: $O(n^2)$

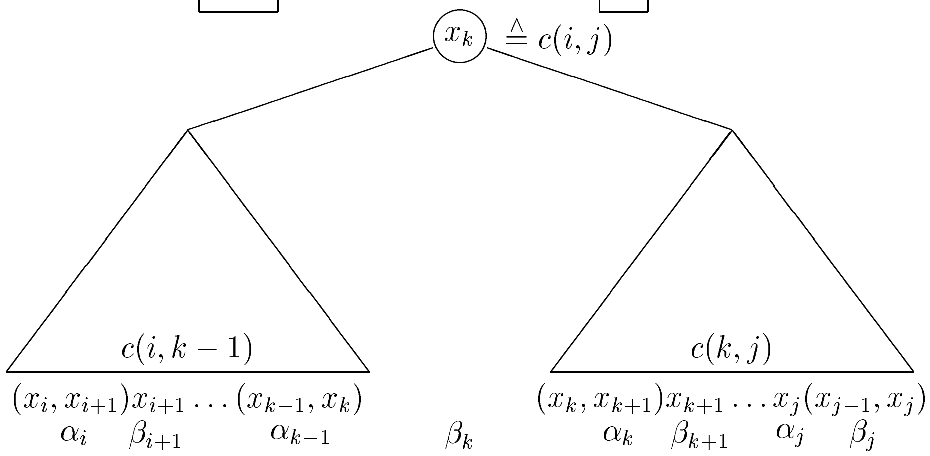
Beispiel [Aigner p.187]

α_0	β_1	α_1	β_2	α_2	β_3	α_3	β_4	α_4
12	5	6	15	16	8	16	12	10
			w_{13}	$=$	61			

$$\begin{array}{rcllll} c_{ii} & = & 0 & & & \\ c_{01} & = & w_{01} + c_{00} + c_{11} & = & 23 & \\ c_{12} & = & w_{12} + c_{11} + c_{22} & = & 37 & \\ c_{23} & = & w_{23} + c_{22} + c_{33} & = & 40 & \\ c_{34} & = & w_{34} + c_{33} + c_{44} & = & 38 & \\ c_{02} & = & w_{02} + \min[c_{00} + c_{12}, c_{01} + c_{22}] & = & 54 + \underbrace{[37, 23]}_{k=2} & = 77 \\ c_{13} & = & w_{13} + \min[c_{11} + c_{23}, c_{12} + c_{33}] & = & 61 + \underbrace{[40, 37]}_{k=3} & = 98 \\ c_{24} & = & w_{24} + \min[c_{22} + c_{34}, c_{23} + c_{44}] & = & 62 + \underbrace{[38, 40]}_{k=3} & = 100 \\ c_{03} & = & w_{03} + \min[c_{00} + c_{13}, c_{01} + c_{23}, c_{02} + c_{33}] & = & 78 + \underbrace{[98, 63, 77]}_{k=2} & = 141 \\ c_{14} & = & w_{14} + \min[c_{11} + c_{24}, c_{12} + c_{34}, c_{13} + c_{44}] & = & 83 + \underbrace{[100, 75, 98]}_{k=3} & = 158 \\ c_{04} & = & w_{04} + \min[c_{00} + c_{14}, c_{01} + c_{24}, c_{02} + c_{34}, c_{03} + c_{44}] & = & 100 + \underbrace{[158, 123, 115, 141]}_{k=3} & = 215 \end{array}$$

		j=1	2	3	4
i=0	0	23	77	141	215
1		0	37	98	158
2			0	40	100
3				0	38
4					0

	j=1	2	3	4
x	1	2	2	3
	x	2	3	3
		x	3	3
			x	4
				x



$$\begin{aligned} c(i,j) &= w(i,j) + \min_{i<k\leq j}[c(i,k-1)+c(k,j)] \quad 0 \leq i < j \leq n \\ c(i,i) &= 0 \quad \text{für } 0 \leq i \leq n \end{aligned}$$

mit

$$w(i,j) = \sum_{l=i}^{j} \alpha_l + \sum_{l=i+1}^{j} \beta_l$$

- Vergleiche:
  - Naive Implementierung mittels Rekursion (Backtracking): Komplexität exponentiell
  - Dynamische Programmierung:
    - speichern der Zwischenwerte in einer Tabelle
    - geschickter Aufbau der Tabelle
- Monotonie-Bedingung:
  - Matrix $r(i,j)$: optimaler Index
    $r(i,j)$ ist monoton in jeder Spalte und Zeile für $0 \leq i < j \leq n$:

    $$r(i,j-1) \leq r(i,j) \leq r(i+1,j)$$

  - Reduktion der Komplexität von $O(n^3)$ auf $O(n^2)$ (ohne Beweis [M'88 p. 151]) mittels ,,Vierecksungleichung" für w_{ij}

3.5 Balancierte Bäume

Nachteil der binären Suchbäume:

- geraten aus der ,,Balance", entweder aufgrund der Reihenfolge der Elemente bei Insert(x, S) oder auch wegen Delete(x, S)-Operationen.
- daraus resultiert: worst-case: $O(n)$

Abhilfe: Balancierte Bäume, die *garantierte* Komplexitäten von $O(\log n)$ haben.

(mindestens) 2 Arten:

- Gewichtsbalanciert: BB(a) ,,Bounded Balance" [M'88 p.176]
  Die Anzahl der Blätter in den Unterbäumen wird möglichst gleich gehalten, wobei a die beschränkte Balance heißt. (beschränkt den max. relativen Unterschied in der Zahl der Blätter zwischen den beiden Teilbäumen jedes Knotens).
- Höhenbalanciert (=ausgeglichen):
  Die Höhe der Unterbäume wird möglichst gleich gehalten.
  - AVL-Bäume, ausgeglichene Bäume
  - (a, b)-Bäume [M'88 p.186] (2,3)-Baum
  - B-Baum: (a, b)-Baum mit $b = 2a - 1$ (2,4)-Baum (,,balanced")

AVL-Baum

[G'92 p. 120] Adelson-Velskij und Landis 1962:

- erste Variante eines balancierten Baums
- (vielleicht) nicht die effizienteste, aber gut zur Darstellung des Prinzips geeignet

Binärer Suchbaum mit ,,Struktur-Invariante":
AVL: Binärer Suchbaum, in dem sich für jeden Knoten die Höhen seiner beiden Teilbäume höchstens um 1 unterscheiden.

Rebalancierungsoperationen:
Diese Operationen werden nach einer Update-Operation (=Delete, Insert) durchgeführt, um die Struktur-Invariante wiederherzustellen. (können sehr mühsam und unangenehm sein)

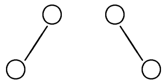
Höhe und Knoten

Wie hoch wird ein AVL-Baum (worst case) ? oder: Wie stark kann ein AVL-Baum entarten ?

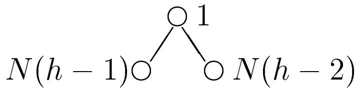
$$N(h) := \text{ minimale Anzahl der Knoten in einem AVL-Baum der Höhe } h$$

$h = 0$: $N(0) = 1$

$h = 1$: $N(1) = 2$



$h \geq 2$: Methode: kombiniere 2 Bäume der Höhe $h-1$ und $h-2$ zu einem neuen, minimal gefüllten Baum mit neuer Wurzel:



Rekursion:

$$N(h) = 1 + N(h-1) + N(h-2)$$

wie Fibonacci (exakt nachrechnen), wobei $Fib(n)$ die Fibonacci-Zahlen sind:

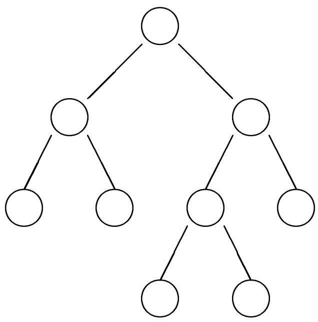
$$N(h) = Fib(h+3) - 1$$

In Worten: Ein AVL-Baum der Höhe h hat mindestens $Fib(h+3)-1$ Knoten. Also gilt für die Höhe h eines AVL-Baums mit n Knoten:

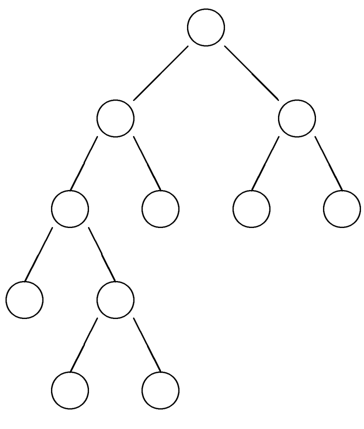
$$N(h) \leq n < N(h+1)$$

Beachte: Damit ist h exakt festgelegt.

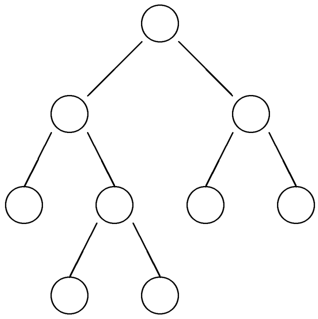
Beispiele:



AVL : $|\Delta h| \leq 1$



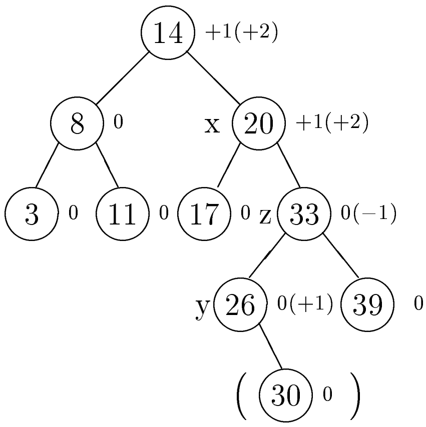
kein AVL : $|\Delta h| = 2$

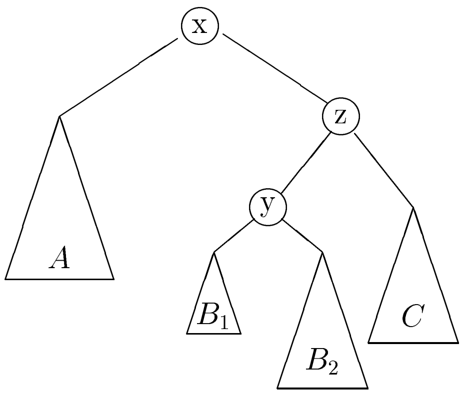


AVL : $|\Delta h| \leq 1$

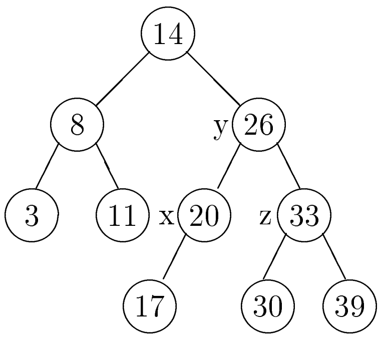
Beispiel: Insert(30,S) , $U = \mathbb{N}$

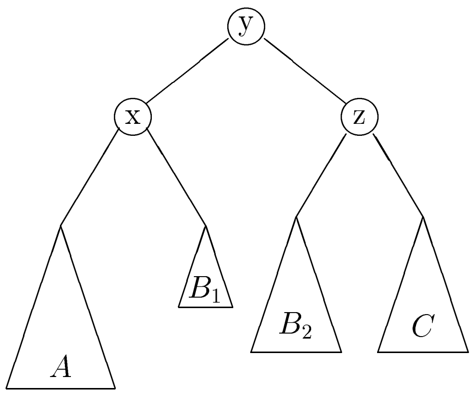
zunächst:





nach ,,Doppelrotation":





Seien $\Phi = \dfrac{1+\sqrt{5}}{2}$, $\phi = \dfrac{1-\sqrt{5}}{2}$
Dann gilt:

$$Fib(h+3) - 1 = N(h) \leq n$$

Einsetzen:

$$\begin{aligned}
Fib(h+3) &= \frac{1}{\sqrt{5}}\left[\Phi^{h+3} - \phi^{h+3}\right] \\
&\geq \frac{1}{\sqrt{5}}\Phi^{h+3} - \frac{1}{2} \\
\frac{1}{\sqrt{5}}\Phi^{h+3} &\leq n + \frac{3}{2} \\
\log_\Phi\left(\frac{1}{\sqrt{5}}\right) + h + 3 &\leq \log_\Phi\left[n + \frac{3}{2}\right] \\
h &\leq \log_\Phi n + const \\
&= (\ln 2/\ln\Phi)\,\mathrm{ld}\, n + const \\
&= 1.4404\,\mathrm{ld}\, n + const
\end{aligned}$$

In Worten:
Ein AVL-Baum ist maximal um 44% höher als ein vollständig ausgeglicher binärer Suchbaum.

Komplexitäten (zu zeigen):

- Zeit: $O(\log n)$ für die 3 Operationen
- Platz: $O(n)$

Insert(a, S):

- Einfügen wie bei normalem binären Suchbaum
- Durch dieses zusätzliche Blatt können die entsprechenden Teilbäume in ihrer Höhe um 1 wachsen.
- deswegen ,,Rebalancierung":
  - zurückverfolgen des Pfades von dem neuen Blatt
  - für jeden Knoten auf diesem Pfad: überprüfe die Balance und führe, falls nötig, ,,Rebalancierung" aus.

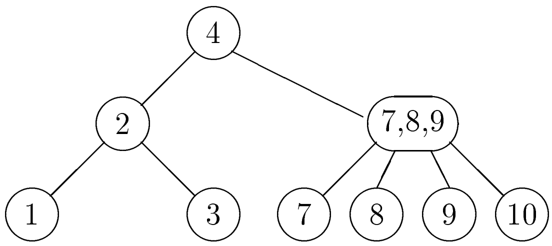
Delete(a, S):

- ähnlich wie Insert(a, S)
- Aber: Es kann nötig sein, auch Vorgängerteilbäume bis hin zur Wurzel zu rebalancieren.
  $\Rightarrow$ maximal n Rebalancierungen aller Teilbäume auf dem Pfad ($\approx 1.44 \cdot$ ld n Knoten)

Anmerkungen:

- nur 4 Pointer umdirigieren
- Suchbaum-Ordnung bleibt erhalten
- betroffener Teilbaum „am Rand": Rotation
- betroffener Teilbaum „innen" : Doppelrotation

Beispiel: (2,4)-Baum $S = \{1, 3, 7, 8, 9, 10\} \subset \mathbb{N}$



Prinzip eines (a, b)-Baums und B-Baums [M'88 p.186] (Definitionen variieren leicht!)

„Bei einem (a, b)-Baum haben alle Blätter gleiche Tiefe und jeder Knoten hat zwischen a und b Söhne."

Definition 3.3 ((a,b)-Baum) Seien $a, b \in \mathbb{N}$ mit $a \geq 2$ und $b \leq 2a - 1$.
Sei $\rho(v)$ die Anzahl der Söhne des Knotens v.
Ein Baum T ist ein (a, b)-Baum, wenn

1. alle Blätter die gleiche Tiefe haben
2. jeder Knoten v erfüllt: $\rho(v) \leq b$
3. jeder Knoten v außer der Wurzel erfüllt: $a \leq \rho(v)$
4. die Wurzel r erfüllt: $\rho(r) \leq 2$

Speziell: Ein $(a, 2a - 1)$-Baum heißt B-Baum der Ordnung $b = 2a - 1$.

Speichern der Menge S:

„blattorientierte Speicherung:"
Sei $S = \{x1 < ... < xn\} \subset U$ (geordnetes Universum) und T ein (a, b)-Baum mit n Blättern.

S wird in T wie folgt gespeichert:

1. Die Elemente von S werden in den Blättern von T von links nach rechts gespeichert: $Inhalt[w] = x_n$ für geeigntes n.

2. Jedem Knoten v werden $\rho(v) - 1$ Elemente $k_1(v) < k_2(v) < ... < k_{\rho(v)-1}(v)$ zugeordnet, so daß gilt:
   für alle Blätter w im i-ten Unterbaum von v, $1 < i < \rho(i)$:

   $$k_{i-1}(v) < Inhalt[w] \leq k_i(v)$$

   An einem ,,Randblatt w" (d.h. $i = 1$ oder $i = \rho(v)$) gilt:

   - w im ersten Unterbaum: $Inhalt[w] \leq k_1(v)$
   - w im $\rho(v)$-ten Unterbaum: $k_{\rho(v)-1}(v) < Inhalt[w]$

Für einen (a, b)-Baum mit n Blättern und Höhe h gilt:

$$\begin{array}{lcl} 2a^{h-1} & \leq n \leq & b^h \\ \log_b n & \leq h \leq & 1 + \log_a n - \log_a 2 \end{array}$$

Speichern des (a,b)-Baums:

jeder Knoten hat $(2b - 1)$ Speicherzellen:

- b Zeiger zu den Söhnen
- $b - 1$: Schlüssel (Elemente)

Speicherausnutzung:

- mindestens $(2a - 1)/(2b - 1)$
- (2,4)-Baum: 3/7

3 Operationen (nur Prinzip):

- Search(y, S): Pfad von der Wurzel zum Blatt auswählen über die Elemente $k_1(v), ..., k_{\rho(v)-1}(v)$ an jedem Knoten v.(,,Wegweiser")
- Insert(y, S): Einfügen und eventuell wiederholtes Spalten von Knoten
- Delete(y, S): Entfernen und, falls nötig, Rebalancieren durch die sogenannten Operationen ,,Verschmelzen" und ,,Stehlen" von Knoten.

Insgesamt: Zeitkomplexität für alle 3 Operationen:
worst- und average case: $O(\log |S|)$
(Beweis [M'88 p.189-190])

Typische (a,b)-Bäume:

Aus der Optimierung der Zugriffszeit ergeben sich grob zwei Varianten:

- bei der internen Speicherung:
  - (2,4)-Baum
  - (2,3)-Baum
- bei der externen Speicherung: (Plattenzugriffe auf n Blätter)
  B-Baum der Ordnung $b \in \{100 \ldots 1000\}$
  Zugriffszeit = Höhe des Baumes h
- $n = 10^6$:
  - Binärer Suchbaum: $h = \operatorname{ld} n \approx 20$
  - B-Baum: $h \leq 1 + \lceil \log_a(n/2) \rceil = 1 + 3 = 4$ für $a = 100$
  - Gewinn: 4 statt 20 Plattenzugriffe

3.6 Priority Queue und Heap

Priority Queue: Warteschlange, bei der die Elemente nach Priorität geordnet sind.
Betriebssystem: „process scheduling": verschiedene Prozesse haben unterschiedliche Prioritäten.

Beschreibung:

- Menge S (nach Priorität geordnet)
- (primäre) Operationen:
  - Insert(x, S)
  - DeleteMin(x, S): Entfernen des Elements mit der höchsten Priorität
- weitere Operationen:
  - Initialize(S)
  - ReplaceMin(x, S)
  - Delete(x, S), Search(x, S)?

Datenstruktur:

- AVL-Bäume
- Heap (wie bei HeapSort):
  „partiell geordneter, links vollständiger Baum":
  Binärbaum, bei dem in jedem Knoten das Minimum (Maximum) des Teilbaums steht.

Programme für 2 Operationen (vgl. HeapSort):

- Insert(y, S):
  Füge Element y an der „letzten" Stelle im Heap-Array ein und bewege es aufwärts, bis es die richtige Position erreicht hat (Prozedur „Upheap")

- DeleteMin(S):
  Entferne das Element in der Wurzel, setze dort das „letzte" Element ein und führe „DownHeap" (wie bei HeapSort) durch.

- Komplexität für beide Operationen: maximal $2 \cdot \text{ld}\, n$ Vergleiche

Übung: Warum wird die Heap-Datenstruktur nicht zur Mengendarstelllung mit den 3 Standard-Operationen „Search", „Insert" und „Delete" verwendet ?
Antwort: „Search" ist problematisch.

4 Graphen

4.1 Motivation: Wozu braucht man Graphen?

Anwendungen und Beispiele

- Beziehungen zwischen Personen:
  - Person A kennt Person B
  - Person A spielt gegen Person B
- Verbindungen zwischen Punkten: von A nach B
- (Straßen, Eisenbahn,...)
- Telefonnetz
- Elektronische Schaltungen

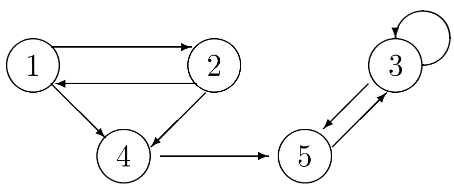
Spezielle Aufgaben und Fragen:

- Existiert eine Verbindung: Ja/Nein ? Existiert eine zweite, falls die erste Verbindung blockiert ist?
- kürzeste Verbindung von A nach B
- minimaler Spannbaum (z.B. Telefon-Netz)
- optimale Rundreise: traveling salesman problem

Begriffe:

- Knoten („Objekte")
- Kanten
- gerichtet/ungerichtet
- gewichtet/ungewichtet

Beispiele: [G'92 p. 145-6] (Beispiel G_1 wird später für Graph-Durchlauf gebraucht.)



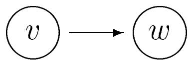
4.2 Definitionen und Graph-Darstellungen

Definition 4.1 Ein gerichteter Graph (Digraph=„directed graph") ist ein Paar $G = (V, E)$ mit einer endlichen, nichtleeren Menge V, deren Elemente Knoten (nodes, vertices) heißen und einer Menge $E \subset V \times V$, deren Elemente Kanten (edges, arcs) heißen.

Beachte:

- Knoten: Anzahl $= |V|$
- Kanten: Anzahl $= |E| \leq |V|^2$
  (wenn man alle Knoten-Paare zuläßt, manchmal auch: $|E| \leq \dfrac{|V|(|V|-1)}{2}$)
- Meist werden die Knoten einfach durchnumeriert: $i = 1, 2, ..., |V|$

Graphische Darstellung der Kante $v \to w$:



Begriffe:

- v und w sind Nachbarn
- v ist Vorgänger
- w ist Nachfolger

Weitere Definitionen:

- Grad eines Knoten := Anzahl der ein- und ausgehenden Kanten
- Pfad := Folge von Knoten $v_1, ..., v_n$ mit $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$.
  = Folge von „zusammenhängenden" Kanten
- Länge des Pfades := Anzahl der Kanten auf dem Pfad
- Ein Pfad heißt einfach, wenn alle Knoten auf dem Pfad mit Ausnahme von $v_1 = v_n$ paarweise verschieden sind.
- Zyklus := einfacher Pfad mit $v_1 = v_n$ und Länge > 0.
- Ein Teilgraph eines Graphen $G = (V, E)$ ist ein Graph $G' = (V', E')$ mit $V' \subset V$ und $E' \subset E$.

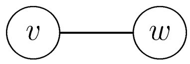
Man kann Markierungen oder Beschriftungen einführen für Kanten und Knoten. Wichtige Kostenfunktionen oder -werte für Kanten:

- $C(v, w)$ oder $cost(v, w)$
- Bedeutung: Reise-Zeit oder -Kosten oder Entfernungen, um von v nach w zu reisen.

Definition 4.2 Ein ungerichteter Graph ist ein gerichteter Graph, in dem die Relation E symmetrisch ist:

$$(v, w) \in E \Rightarrow (w, v) \in E$$

Graphische Darstellung (ohne Pfeil):



Bemerkung: Die Begriffe sind analog zu denen für gerichtete Graphen. Bisweilen sind Modifikationen erforderlich:

- Ein Zyklus hat mindestens drei Knoten.

Graph-Darstellungen

Die Adressierung wird einfach, wenn die Knoten mit den natürlichen Zahlen bezeichnet („identifiziert") werden.

Sei daher $V = \{1, 2, ..., |V|\}$

- knoten-orientiert: Adjazenzmatrix (markierte Adjazenzmatrix)
  Prinzip: Adressierung mittels $(i, j) \in V^2$, wobei A ist eine boolesche Matrix mit:

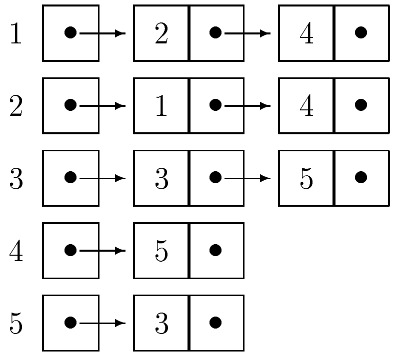
$$A_{ij} = \begin{cases} True & \text{falls } (i, j) \in E \\ False & \text{sonst} \end{cases}$$

  - Array: $A_{ij} = A[i][j]$
    Vorteil: Entscheidung, ob $(i, j) \in E$ in Zeit $O(1)$
    Nachteil: Platz $O(|V|^2)$ ineffizient falls $|E| \ll |V|^2$
    Initialisierung: Zeit $O(|V|^2)$
    erweiterbar für Kanten-Markierungen und Kosten(-Funktionen) Adjazenz-Matrix für G_1 ($A_{ij} = 1$ falls Kante existiert)

	1	2	3	4	5
1		1		1	
2	1			1	
3			1		1
4					1
5					

  - Adjazenzliste für jeden Knoten: Liste der (Nachfolger-)Nachbarknoten (Vorgänger: „invertierte" Adjazenzliste)
    Vorteil: Platz $O(|V| + |E|)$
    Nachteil: Entscheidung, ob $(i, j) \in E$ in Zeit $O\left(\frac{|E|}{|V|}\right)$ (average case)

Adjazenz-Liste für G_1:



- kanten-orientiert:
  Für die Kanten ist ein eigener „Index" i erforderlich:
  Prinzip: Adressierung für jede Kante mittels $i \in E$:

  – Vorgänger-Knoten
  – Nachfolger-Knoten
  – gegebenenfalls: Markierung/Kosten

  Methode: meist statische Darstellungsform, seltener dynamische Listen.

Man unterscheidet verschiedene Typen von Graphen:

- vollständiger (complete) Graph: $|E| = |V|^2 \left(\text{oder } |E| = \frac{|V|(|V|-1)}{2}\right)$
- dichter (dense) Graph: $|E| \simeq |V|^2$
- dünner (sparse) Graph: $|E| \ll |V|^2$

Darstellungskriterien:

- Speicherplatz-Effizienz
- Zugriffs-Effizienz

Programme:

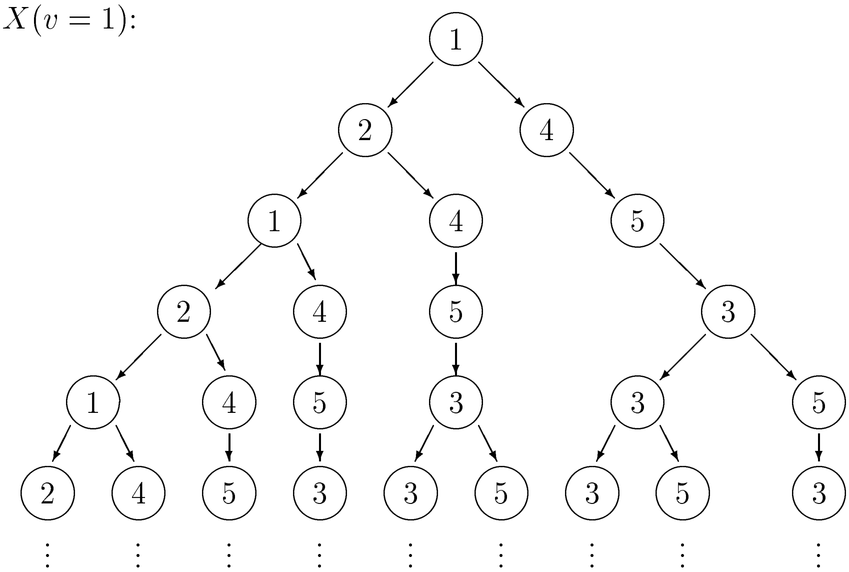
```
program adjmatrix (input, output);
const maxV = 50;
var  j, x, y, V, E : integer;
     a : array[1..maxV,1..maxV] of boolean;
begin
     readln (V, E);
     for x:= 1 to V do
          for y := 1 to V do a[x,y] := false;
     for x := 1 to V do a[x,x] := true; (* speziell *)
     for j := 1 to E do
     begin
          readln(v1, v2);
          x := index(v1); y:= index(v2);
          a[x,y] := true; a[y,x] := true;
     end
end.

program adjlist (input, output);
const maxV = 1000;
type link = ↑node;
      node = record
               v: integer;
               next: link
               end;
var  j, x, y, V, E : integer;
     t, z : link;
     adj : array[1..maxV] of link;
begin
     readln (V,E);
     new(z); z↑.next := z;
     for j := 1 to V do adj[j] := z;
     for j:= 1 to E do
     begin
          readln(v1,v2);
          x := index(v1); y:= index(v2);
          new(t); t↑.v := x; t↑.next := adj[y]; adj[y] := t;
          new(t); t↑.v := y; t↑.next := adj[x]; adj[x] := t;
     end
end.
```

4.3 Graph-Durchlauf

Die Expansion $X(v)$ eines Graphen G in einem Knoten v ist ein Baum, der wie folgt definiert ist:

1. Falls v keine Nachfolger hat, ist $X(v)$ nur der Knoten v.
2. Falls $v_1, ..., v_k$ die Nachfolger von v sind, ist $X(v)$ der Baum mit der Wurzel v und den Teilbäumen $X(v_1), ..., X(v_k)$.

Beispiel: [G'92 p.151]
$X(v = 1)$:



Beachte:

- Knoten des Graphen kommen in der Regel mehrfach im Baum vor.
- Ein Baum ist unendlich, falls der Graph Zyklen hat.
- Der Baum stellt die Menge aller Pfade dar, die von v ausgehen.

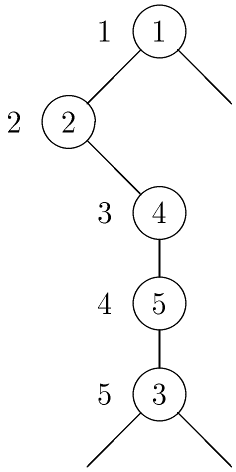
Programm: Graph-Durchlauf

Gleiches Konzept wie Baum-Durchlauf (vgl. Kap. 1.3)

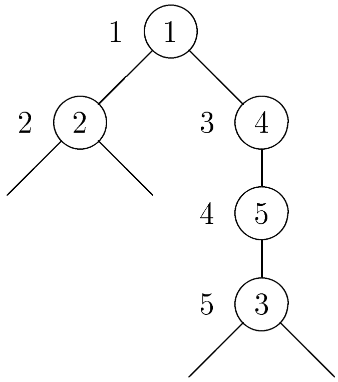
- Breitendurchlauf: preorder traversal (breadth first) mittels stack (oder rekursiv)
- Tiefendurchlauf: level order traversal (depth first) mittels queue

Wichtige Modifikation:
Markieren der schon besuchten Knoten ist notwendig, weil Graph-Knoten im Baum mehrfach vorkommen können.

Tiefendurchlauf für G_1:



Breitendurchlauf für G_1:



Programm-Plan:

1. initialisieren alle Knoten als ,,not visited"
2. abarbeiten der Knoten
   if (,,not visited") **then**
   ...
   markiere: ,,visited"

Interpretation: Graph-Durchlauf

Während des Graph-Durchlaufs werden die Graph-Knoten in 3 Klassen eingeteilt:

- Baum-Knoten (tree vertices): Knoten, die schon besucht worden und abgearbeitet sind. Diese Knoten ergeben den ,,Suchbaum". $val[v] = id > 0$
- Rand-Knoten (fringe vertices): Knoten, die über eine Kante mit einem Baum-Knoten verbunden sind. $val[v] = -1$
- Ungesehene Knoten (unseen vertices): Knoten, die noch nicht erreicht worden sind. $val[v] = 0$

Verallgemeinerung für beliebiges Auswahlkriterium führt zu folgendem Programm-Plan:

- Start: Markiere den Startknoten als Rand-Knoten und alle anderen Knoten als ungesehene Knoten.
- Schleife: **repeat** ... **until** (alle Knoten abgearbeitet)
  - Wähle einen Rand-Knoten x nach Auswahlkriterium (depth first, breadth first, priority first).
  - Markiere x als Baum-Knoten und bearbeite x.
  - Markiere alle ungesehenen Nachbar-Knoten von x als Rand-Knoten.

Programme: [S'88]

```
procedure listdfs; { Rekursive Tiefensuche mit Adjazenzliste }
var  id, k : integer;
     val : array[1..maxV] of integer;
     procedure visit (k : integer);
     var t : link;
     begin
          id := id + 1; val[k] := id;
          t := adj[k];
          while t <> z do begin
               if val[t↑.v] = 0 then visit(t↑.v);
               t := t↑.next
          end
     end;
begin
     id := 0;
     for k := 1 to V do val[k] := 0;
     for k := 1 to V do
          if val[k] = 0 then visit(k)
end;
```

Prozedur „visit" für die rekursive Tiefensuche mit Adjazenzmatrix:

```
     procedure visit (k:integer);
          var t : integer;
     begin
          id := id + 1; val[k] := id;
          for t := 1 to V do
               if a[k,t] then
                    if val[t] = 0 then visit(t)
     end;
```

```
procedure listdfs; { Tiefensuche mit Stack und Adjazenzliste }
var  id, k : integer;
     val : array[1..maxV] of integer;
     procedure visit (k:integer);
     var t : link;
     begin
          push (k);
          repeat
               k := pop; id := id + 1; val[k] := id;
               t := adj[k];
               while t <> z do begin
```

```
                    if val[t↑.v] = 0 then begin
                        push(t↑.v); val[t↑.v] := -1
                    end;
                    t := t↑.next
                end
            until stackempty
        end;
begin
    id := 0; stackinit;
    for k := 1 to V do val[k] := 0;
    for k := 1 to V do
        if val[k] = 0 then visit(k)
end;

procedure listbfs; { Breitensuche mit Queue und Adjazenzliste }
var id, k : integer;
    val : array[1..maxV] of integer;
    procedure visit (k:integer);
    var t : link;
    begin
        put(k);
        repeat
            k := get; id := id + 1; val[k] := id;
            t := adj[k];
            while t <> z do begin
                if val[t↑.v] = 0 then begin
                    put(t↑.v); val[t↑.v] := -1
                end;
                t := t↑.next
            end
        until queueempty
    end;
begin
    id := 0; queueinitialize;
    for k := 1 to V do val[k] := 0;
    for k := 1 to V do
        if val[k] = 0 then visit(k)
end;
```

Komplexität:

- Zeit bei Verwendung einer Adjazenzliste: $O(|E| + |V|)$
- Zeit bei Verwendung einer Adjazenzmatrix: $O(|V|^2)$

Stack-Implementation:

Knoten, die schon erreicht („touched") aber noch nicht abgearbeitet sind, werden auf dem Stack gehalten. [Rekursiv: lokale Variable t]

Implementierung mit Hilfe des Arrays *val*

$$val[v] = \begin{cases} 0 & \text{Knoten } v \text{ noch nicht erreicht} \\ 1 & \text{Knoten } v \text{ auf dem Stack} \\ id > 0 & \text{Knoten } v \text{ abgearbeitet} \end{cases}$$

Beachte: Die Reihenfolge ist nicht exakt dieselbe wie im rekursiven Programm.

Analog: Queue für die Breitensuche

4.4 Single-Source Best Path (Dijkstra 1959)

(Originalarbeit: 2-3 Seiten !!!)

Graph:

- gerichtet
- Bewertung ≥ 0
- Kosten:

$$\begin{aligned} c[v,w] &= \begin{cases} > 0 & \\ \infty & \text{Kante existiert nicht} \end{cases} \\ c[v,v] &= 0 \qquad \forall \quad v \end{aligned}$$

Aufgabe: Der Start-Knoten v und Endknoten w seien vorgegeben. Finde den Pfad mit minimalen, (additiven) Gesamtkosten von v nach w.

Es wird sich zeigen, daß die folgende Verallgemeinerung nicht mehr Aufwand erfordert:

Sei der Start-Knoten v vorgegeben: Finde zu jedem End-Knoten w den Pfad mit minimalen, additiven Gesamtkosten von v nach w.

Basis des Dijkstra-Algorithmus

Optimalitätsprinzip ([Ottmann p.620] ähnlich „dynamische Programmierung")
Für jeden besten Pfad $p = (v_0, v_1, ..., v_k)$ von v_0 nach v_k ist jeder Teilpfad $p' = (v_i, ..., v_j)$, $\quad 0 \leq i < j \leq k$, ein bester Pfad von v_i nach v_j.

Beachte: Die Eindeutigkeit muß nicht gegeben sein und wird auch nicht behauptet.

Beweis des Optimalitätsprinzips (Indirekter Beweis):

Wäre dies nicht so, d.h. gäbe es einen kürzeren Weg p'' von v_i nach v_j, so könnte in p der Teilpfad p' durch p'' ersetzt werden, und der so konstruierte Pfad wäre besser als p. Dies ist aber ein Widerspruch zur Annahme, daß p bester Pfad von v_0 nach v_k ist. $\square$

Beachte: Die Additivität der Kostenfunktion ist hier wesentlich.

Dijkstra-Algorithmus: [A'83 p.205]

Seien $V := \{1, ..., n\}$ und $S := \{$ Knoten, zu denen der beste Pfad schon gefunden ist.$\}$

```
procedure Dijkstra;
      { Dijkstra computes the cost of the shortest paths from vertex 1
      to every vertex of a directed graph}
begin
      S := {1};
      for i := 2 to n do
            D[i] := C[1,i]; { initialize D }
      for i := 1 to n-1 do begin
            choose a vertex w in V − S such that D[w] is a minimum;
            add w to S;
            for each vertex v in V − S do
                  D[v] := min(D[v], D[w] + C[w,v])
      end
end; { Dijkstra }
```

Erläuterungen zum Dijkstra-Algorithmus: Die besten Pfade werden systematisch aufgebaut, indem wir bereits bekannte beste Pfade Kante um Kante wachsen lassen

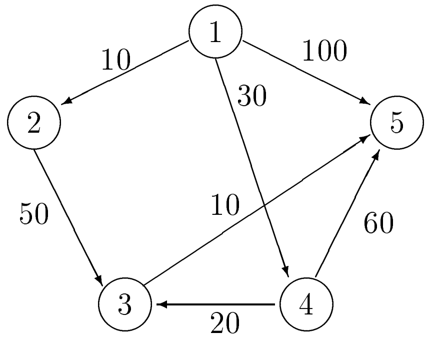
- Durch Hinzunahme einer Kante können die Gesamtkosten nur wachsen.
- Der beste Pfad kann keinen Zyklus haben, falls die Kosten längs des Zyklus größer null sind. Falls =0, gibt es einen besten Pfad ohne Zyklus mit denselben Kosten.
- Der beste Pfad hat maximal $(n-1)$ Kanten.
- Falls die besten Pfade von v zu allen anderen Knoten $\neq w$ höhere Kosten haben, ist der beste Pfad von v nach w gefunden.

Anmerkungen zu Dijkstra:

- Der Dijkstra-Algorithmus liefert keine Näherung, sondern garantiert Optimalität.
- Bei negativen Bewertungen ist der Dijkstra-Algorithmus zunächst nicht anwendbar.
- Falls Zyklen mit insgesamt negativer Bewertung existieren, gibt es kein Minimum mehr.

- Frage bei vollständigem Graphen (d.h. $|E| \simeq |V|^2$):
  Ist eine niedrigere Komplexität vorstellbar, wobei jede Kante weniger als einmal betrachtet wird ?

Beispiel: [A'83 p.205]



Iteration	w	S	$D[2]$	$D[3]$	$D[4]$	$D[5]$
initial	-	$\{1\}$	10	∞	30	100
1	2	$\{1,2\}$	10	60	30	100
2	4	$\{1,2,4\}$	10	50	30	90
3	3	$\{1,2,3,4\}$	10	50	30	60
4	5	$\{1,2,3,4,5\}$	10	50	30	60

Bester Pfad: $1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

Übung: Erstelle das vollständige Programm und rekonstruiere den optimalen Pfad.

Dijkstra:

1. mit Adjazenz-Matrix

2. mit Priority Queue (Heap)
   $|E| \ll |V| \Rightarrow$ Adjazenz-Liste (+ Kosten) und Priority Queue für $V - S$

   Zeilen (7) und (8):
   Durchlaufen der Adjazenz-Liste und Updaten der Abstände in der Priority Queue
   Insgesamt: $|E|$ updates in $O(\log |V|)$
   Zeilen (1-3) (4): Jeweils $O(|V|)$
   Zeilen (5) und (6):

   - implementieren „DeleteMin $(y, V - S)$"
   - $(|V| - 1)$ Iterationen mit je $O(\log |V|)$ Komplexität

   $\Rightarrow O([|E| + |V|] \cdot \log |V|)$
   meist $|V| \leq |E|$: $O(|E| \log |V|)$

3. Dijkstra mit „Fibonacci-Heap" (ohne Erläuterung) $O(|E| + |V| \cdot \log |V|)$

4.5 All-Pairs Best Path (Floyd 1962, Warshall 1962)

Sei $G = (V, E)$ ein gerichteter Graph mit nichtnegativer Bewertung: $C[v, w] \geq 0$

Aufgabe: Finde den besten Pfad für jedes Paar von Knoten $v \to w$.

Prinzipiell machbar:

- Dijkstra-Algoritmus für jeden Knoten
- Komplexität: $|V| \cdot O(|V|^2) = O(|V|^3)$

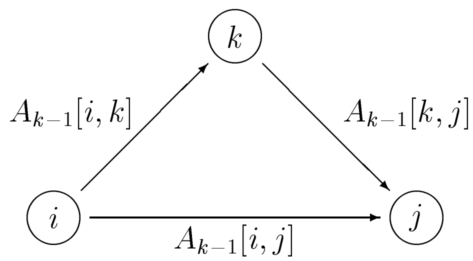
Andere Variante: Floyd-Algorithmus
Dieser beruht direkt auf dynamischer Programmierung.

Sei $V = \{1, 2, ..., n\}$ mit $n = |V|$.
Wir definieren zunächst eine Matrix A_k mit den Elementen:

$A_k[i, j] :=$ minimale Kosten, um von Knoten i zu Knoten j zu kommen über Knoten aus $\{1, ..., k\}$

Aufgrund der Definition gilt die Rekursionsgleichung (Gleichung der DP):

$$A_k[i, j] := \min\{A_{k-1}[i, j], A_{k-1}[i, k] + A_{k-1}[k, j]\}$$



Vereinfachungen:
Es ändert sich kein Element $A_k[i, j]$, wenn i oder j gleich k sind.

$$\begin{aligned} A_k[i, k] &= A_{k-1}[i, k] \\ A_k[k, j] &= A_{k-1}[k, j] \end{aligned}$$

Deswegen werden keine n Matrizen $A_k[.,.]$, $k = 1, ..., n$ benötigt, sondern eine Kopie $A[i, j]$ genügt.

Initialisierung: $A[i, j] = C[i, j]$

Komplexität:

- Programm-Struktur: Schleifen über i, j, k
- Zeit $O(|V|^3)$
- Platz $O(|V|^2)$

Floyd-Algorithmus: [A'74]

```
procedure shortest (var A: array [1..n,1..n] of real;
                        C: array [1..n,1..n] of real;
                        P: array [1..n,1..n] of integer);
var i, j, k : integer;
begin
    for i := 1 to n do
        for j := 1 to n do begin
            A[i,j] := C[i,j];
            P[i,j] := 0;
        end;
    for i := 1 to n do
        A[i,i] := 0;
    for k := 1 to n do
        for i := 1 to n do begin
            for j := 1 to n do begin
                if A[i,k] + A[k,j] < A[i,j] then begin
                    A[i,j] := A[i,k] + A[k,j];
                    P[i,j] := k
                end
end; { shortest }
```

Warshall-Algorithmus:

Spezialfall: „unbewerteter Graph"

$$C[v,w] = \begin{cases} 0 & \text{, falls Kante } (v \to w) \text{ existiert} \\ \infty & \text{, sonst} \end{cases}$$

Dann berechnet der Floyd-Algorithmus, ob überhaupt ein Pfad für jedes Knotenpaar existiert.

Definition 4.3 (stark verbunden) Zwei Knoten v und w eines gerichteten Graphen heißen stark verbunden, wenn es einen Pfad von v nach w und einen Pfad von w nach v gibt.

Definition 4.4 (verbunden) Zwei Knoten v und w eines ungerichteten Graphen heißen verbunden, wenn es einen Pfad von v nach w gibt.

Definition 4.5 (Transitive Hülle) Gegeben sei ein gerichteter Graph $G = (V, E)$. Die transitive Hülle (transitive closure) von G ist der Graph $\bar{G} = (V, \bar{E})$, wobei $\bar{E}$ definiert ist durch:

$$(v, w) \in \bar{E}) \Rightarrow \text{es gibt in } G \text{ einen Pfad von } v \text{ nach } w.$$

Die entsprechende Modifkation des Floyd-Algorithmus ergibt den Algorithmus von Warshall, wobei statt der Real/Integer-Kosten-Matrix eine boolesche Matrix verwendet wird. Das Element $A[i, j]$ der booleschen Matrix gibt an, ob ein Pfad von i nach j existiert.

Warshall-Algorithmus für die transitive Hülle

```
procedure Warshall (var A: array [1..n,1..n] of boolean;
                        C: array [1..n,1..n] of boolean);
var i, j, k : integer;
begin
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := C[i,j];
    for k := 1 to n do
        for i := 1 to n do begin
            for j := 1 to n do begin
                if not A[i,j] then
                    A[i,j] := A[i,k] and A[k,j];
end; { Warshall }
```

4.6 Minimum Spanning Tree (MST, Prim 1957)

auch: Minimaler Spannbaum Spannbaum mit minimalen Kosten
ähnlich: Kruskal 1957 (und auch Dijkstra nach Uminterpretation)

Aufgabe: Finde eine Menge von Kanten, so daß alle Knoten erreichbar sind und die Summe der Kosten der Kanten minimal ist. (Anwendungen: Straßen-, Telefon-Netz)

Graph:

- ungerichtet
- verbunden
- bewertet (≥ 0)

Technische Definitionen für einen ungerichteten Graph $G = (V, E)$

- G heißt verbunden, wenn jedes Paar von Knoten verbunden ist. (oder: G = transitive Hülle von G).
- Ein verbundener azyklischer Graph heißt freier Baum. Daraus wird ein (normaler) Baum, wenn man irgendeinen Knoten als Wurzel wählt.
- Ein Spannbaum ist ein freier Baum, der alle Knoten in G enthält, und dessen Kanten eine Teilmenge von E bilden.

- Sei G bewertet: Die Kosten eines Spannbaums von G ergeben sich als die Summe über die Kosten seiner Kanten.

Eigenschaft eines minimalen Spannbaums: ,,MST property"

Sei $G = (V, E)$ ein verbundener ungerichteter Graph mit Bewertungsfunktion $C[v, w] \geq 0$ für eine Kante (v, w) und $U \subset V$:
Falls gilt:

$$(u, v) = \arg\min\{C[u', v'] : u' \in U, v' \in V \backslash U\}$$

dann gibt es einen minimalen Spannbaum, der die Kante (u, v) enthält.

Beweis (hier nicht ausgeführt):

- durch Widerspruch
- Ähnlich wie beim Optimalitätsprinzip ist die Additivität der Kostenfunktion wesentlich.

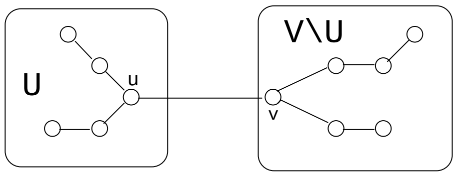


Abbildung 16: $C[u, v] = \min\{C[u', v'] : u' \in U, v' \in V \backslash U\}$

Beweis durch Widerspruch:

Falls der MST nicht (u, v) enthält, ersetze die ,,entsprechende" Kante durch (u, v). Dadurch entsteht ein ,,besserer" MST. $\Rightarrow$ Widerspruch $\square$

Prim-Algorithmus:

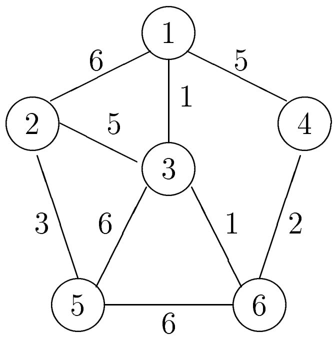
Sei $V = \{1, ..., n\}$, T der zu konstruierende Minimalbaum U die Menge seiner Kanten:

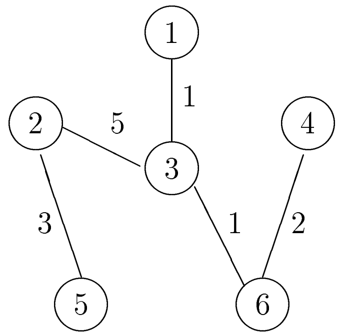
Initialisierung: $T := \phi$; $U := \{1\}$;
while $(U \neq V)$ **do begin**
    Wähle beste Kante, die U und $U \backslash V$ verbindet;
    $U := U \cup \{v\}$; $T := T \cup \{[u, v]\}$
end;

Um die beste Kante zwischen U und $V \backslash U$ zu finden, werden zwei Arrays definiert, die bei jedem Durchgang aktualisiert werden. $i \in V \backslash U$:

$$\begin{aligned} CLOSEST[i] &:= \arg\min\{C[u, i] : u \in U\} \\ LOWCOST[i] &:= C[i, CLOSEST[i]] \end{aligned}$$

Beispiel: Prim

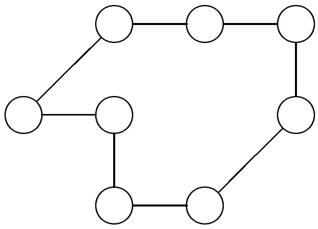





```
     procedure Prim (C:array [1..n,1..n] of real);
          { Prim prints the edges of a minimum-cost spanning tree for a graph with
          vertices {1, 2, ..., n} and cost matrix C on edges }
     var  LOWCOST : array [1..n] of real;
          CLOSEST : array [1..n] of integer;
          REACHED : array [1..n] of boolean;
          i, j, k: integer;
          min : real;
          { i and j are indices. During a scan of the LOWCOST array, k is the index
          of the closest vertex found so far, and min = LOWCOST[k] }
     begin
(1)       for i:= 2 to n do begin
               { initialize with only vertex 1 in the set U }
(2)            LOWCOST[i] := C[1,i];
(3)            CLOSEST[i] := 1; REACHED[i] := false;
          end;
(4)       for i:= 2 to n do begin
               { find the closest vertex k outside of U to some vertex in U }
(5)            min := LOWCOST[2];
(6)            k := 2;
(7)            for j:= 3 to n do
(8)                 if LOWCOST[j] > min then begin
(9)                      min := LOWCOST[j];
(10)                     k := j;
                    end;
(11)           writeln (k, CLOSEST[k]) { print edge }
(12)           REACHED[k] := true; { k is added to U }
(13)           for j := 2 to n do { adjust costs to U }
(14)                if (C[k,j] < LOWCOST[j]) and not REACHED[j] then begin
(15)                     LOWCOST[j] := C[k,j];
(16)                     CLOSEST[j] := k;
                    end
          end
     end; { Prim }
```

Zusammenfassung:

- bester Pfad (Dijkstra) : $O(n^2)$
- Minimaler Spannbaum : $O(n^2)$
- Rundreise : $n!$ oder $(n-1)!$ Möglichkeiten
  durch dynamische Programmierung [A'93] $\rightarrow n^2 \cdot 2^n$



Offene Frage: Gibt es einen Algorithmus, der in polynomieller Zeit $O(n^k)$ das Traveling Salesman Problem löst ?

5 String Processing

5.1 String Searching

(Suchen von Strings in Texten, erkennen von Zeichenmustern)

- Zeichen:
  - Bit
  - Bitmuster
  - Byte: 8 Bits: (ASCII-)Zeichen:
    * Buchstaben
    * Ziffern
    * Sonderzeichen
    * (Steuerzeichen)
- Zeichenkette (String): lineare Folge von Zeichen
- Text-String: Folge von Text-Zeichen, (Bit oder ASCII)

Anwendungen: Editor, Text-Verarbeitung

Aufgabe:
Gegeben sei ein Text-String $a[1..N]$ und ein Suchmuster $p[1..M]$:
Finde ein Vorkommen (oder alle) von $p[1..M]$ in $a[1..N]$

5.1.1 Naiver Algorithmus (brute force)

Teste alle Positionen $i = 1...N$ des Text-Strings
Falls ein Mismatch auftritt, gehe eine Position weiter.

Algorithmus:

- Return-Wert: gefunden: Position $(0 < i \leq N)$
  negativ: $N + 1$
- maximal $M \cdot (N - M + 1) \simeq N \cdot M$ Zeichenvergleiche $(M \ll N)$
- Im ungünstigsten Fall bestehen $a[1..N]$ und $p[1..M]$ nur aus Nullen und einer abschließenden Eins. Dann müssen alle M Zeichen $p[1..M]$ in jeder Position überprüft werden.
- In der Praxis werden deutlich weniger Zahlenvergleiche als $(N \cdot M)$, (eher $M \cdot const$ Zahlenvergleiche) benötigt wegen vorzeitigen Abbruchs.

Beispiel: brute force
$a[..]$ = „A STRING SEARCHING EXAMPLE CONSISTING OF"
$p[..]$ = „STING"
(n): Positionen, wo die ersten n Zeichen passen:

```
A STRING SEARCHING EXAMPLE CONSISTING OF
  STING (2)
         STING (1)
                             STING (1)
                               STING (5)
```

```
function brutesearch : integer;
var i, j :integer;
begin
    i := 1; j := 1;
    repeat
        if a[i] = p[j]
            then begin i := i + 1; j := j + 1 end
            else begin i := i - j + 2; j := 1 end
    until (j > M) or (j > N)
    if j > M then brutesearch := i - M else brutesearch := i
end;
```

1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 0 1 1 1
1 0 $\boxed{1}$ 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
1 $\boxed{0}$ 1 0 0 1 1 1
1 $\boxed{0}$ 1 0 0 1 1 1
1 0 1 0 0 1 $\boxed{1}$ 1
$\boxed{1}$ 0 1 0 0 1 1 1
1 0 $\boxed{1}$ 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
1 0 1 0 0 $\boxed{1}$ 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
1 0 $\boxed{1}$ 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
$\boxed{1}$ 0 1 0 0 1 1 1
1 0 0 1 1 1 0 1 0 0 1 0 1 0 0 0 $\boxed{1}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{1}$ $\boxed{1}$ 0 0 0 1 1 1

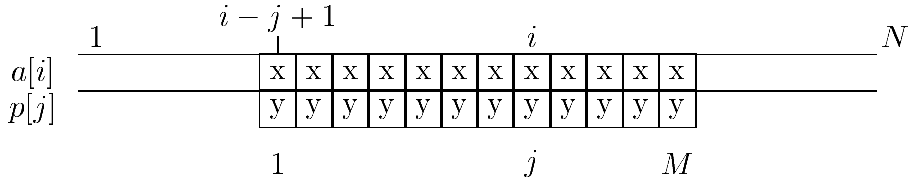
Brute-force string search in binary text

5.1.2 Knuth-Morris-Pratt-Algorithmus (KMP-Algorithmus)

Verbesserung des naiven Algorithmus:

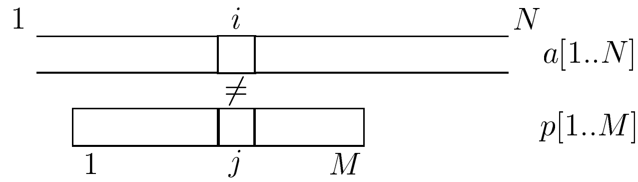
- Wenn an Position j des Musters ein ,,Mismatch” aufgetreten ist, haben die letzten $(j-1)$ Zeichen im Text mit denen des Musters übereingestimmt.
- Eine doppelte Überprüfung von Zeichen im Text-String wird vermieden, indem die vorab bekannte ,,Struktur” des Musters ausgenutzt wird.

Veranschaulichung:



Definiere ein Array $next[1..M]$, so daß um $next[j]$ Positionen ,,zurückgegangen” wird, falls ein Mismatch $a[i] \neq p[j]$ auftritt. Wird die Suche bei (i, j) wegen $p[j] \neq a[i]$ abgebrochen, dann wird der nächste Vergleich gemacht für:

$$\begin{array}{rll} a[i+1] \text{ und } p[1] & \text{falls} & j = 1 \\ a[i] \text{ und } p[next[j]] & \text{falls} & j > 1 \end{array}$$



Darüber hinaus wird so ein Zurückgehen (,,Zurücksetzen”, ,,backing up”) im Text $a[1..n]$ vermieden.

KMP-Algorithmus

Wenn ein Mismatch auftritt, also $a[i] \neq p[j]$ für $j > 1$, dann wäre die neue Position im Textstring: $i := i - next[j] + 1$.
Da aber die ersten $(next[j] - 1)$ Textzeichen ab dieser Position zu den Zeichen des Musters passen, bleibt i unverändert und $j := next[j]$.

$j = 1ora[i] \neq p[1]$:

- kein overlap
- Wunsch: $i := i + 1$ und $j := 1$
- Trick: $next[1] := 0$ [Array vergrößern: $p[0..M]$ wegen ,,OR”]

Berechnung von $next[1..M]$:

$next[j]$:,,Vergleiche die Zeichen $p[1..j]$ mit sich selbst"

- Schiebe eine Kopie der ersten $(j-1)$ Zeichen über das Muster selbst von links nach rechts.
- Start: Das erste Zeichen der Kopie ist über dem zweitem Zeichen des Musters.
- Stop: Falls alle überlappenden Zeichen passen oder es keine passenden gibt: $next[j] := 1 +$,,Zahl der passenden Zeichen"
- Definition: $next[1] := 0$

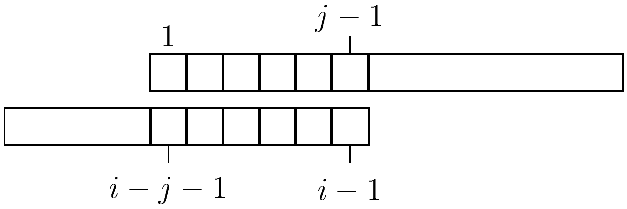
Erzeugung der Tabelle $next[1..M]$:

- elementar $O(M^2)$
- KMP angewandt auf $p[1..M]$ trickreich

j	$next[j]$														
2	1	1	0	1	0	0	1	1	1						
			1	0	1	0	0	1	1	1					
3	1	1	0	1	0	0	1	1	1						
				1	0	1	0	0	1	1	1				
4	2		1	0	1	0	0	1	1	1					
			1	0	1	0	0	1	1	1					
5	3	1	0	1	0	0	1	1	1						
				1	0	1	0	0	1	1	1				
6	1	1	0	1	0	0	1	1	1						
							1	0	1	0	0	1	1	1	
7	2	1	0	1	0	0	1	1	1						
							1	0	1	0	0	1	1	1	
8	2	1	0	1	0	0	1	1	1						
								1	0	1	0	0	1	1	1

Restart positions for Knuth-Morris-Pratt search

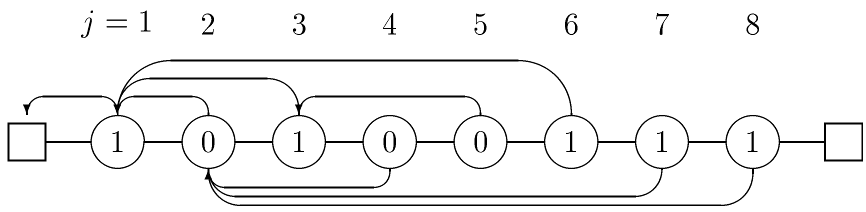
$p[i] = p[j]$: i und j werden erhöht:



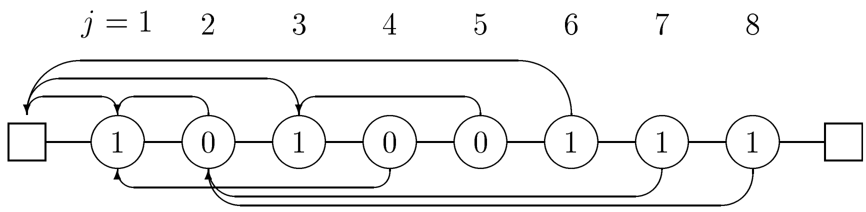
$(j-1)$ passende Zeichen $\Rightarrow next[i] = j$

```
function kmpsearch : integer;
var i, j : integer;
begin
    i := 1; j := 1; initnext;
    repeat
        if (j = 0) or (a[i] = p[j])
        then begin
            i := i + 1; j := j + 1
        end
        else j := next[j]
    until (j > M) or (i > N)
    if j > M
    then kmpsearch := i - M
    else kmpsearch := i
end;

procedure initnext;
var i, j :integer;
begin
    i := 1; j := 0; next[1] := 0;
    repeat
        if (j = 0) or (p[i] = p[j])
        then begin i := i + 1; j := j + 1; next[i] := j end
        else j := next[j]
    until i > M
end;
```

Verbesserung: Ersetze next[i] := j durch:
if p[i] <> p[j] **then** next[i] := j **else** next[i] := next[j]



Verbesserte Version:



$$\begin{aligned}
&1 \to 0\\
&2 \to 1\\
&3 \to 1 \to 0\\
&4 \to 2 \to 1\\
&5 \to 3\\
&6 \to 1 \to 0\\
&7 \to 2\\
&8 \to 2
\end{aligned}$$

Komplexität: KMP

maximal $N + M$ Zeichenvergleiche:
Für jedes $i = 1..N$:

entweder $j := j + 1$
oder $j := next[j]$

Praxis:

- In der Regel ist KMP kaum besser als der naive Algorithmus (ohne selbstwiederholende Teile im Muster kein Vorteil für KMP).
- Vorteil bei externen Speichern: Zurücksetzen im Text ist unnötig, da der Index i nur wachsen kann.

1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	1	1	1	0	0	0	1	1	1
1	0	$\boxed{1}$	0	0	1	1	1																						
			1	$\boxed{0}$	1	0	0	1	1	1																			
				1	$\boxed{0}$	1	0	0	1	1	1																		
					1	0	1	0	0	1	$\boxed{1}$	1																	
										1	0	1	0	0	$\boxed{1}$	1	1												
1	0	0	1	1	1	0	1	0	0	1	0	1	0	0	0	$\boxed{1}$	$\boxed{0}$	$\boxed{1}$	$\boxed{0}$	$\boxed{0}$	$\boxed{1}$	$\boxed{1}$	$\boxed{1}$	0	0	0	1	1	1

Knuth-Morris-Pratt string search in binary text

5.1.3 Boyer-Moore-Algorithmus (BM-Algorithmus)

„rückwärts-Reihenfolge der Vergleiche: $p[M], p[M-1], ..., p[1]$

Methode bei „nichtpassendem Zeichen $a[i]$”:

- Das Muster wird verschoben, bis erstmals das Textzeichen $a[i]$ und das Zeichen im Muster übereinstimmen.
- Falls das Zeichen $a[i]$ im Muster gar nicht vorkommt, kann das Muster um seine ganze Länge verschoben werden.

Beispiel (Grundidee):

```
A STRING SEARCHING EXAMPLE CONSISTING OF
STING
       STING
            STING
                    STING
                           STING
                                    STING
                                           STING
                                               STING
```

Vollständiger BM-Algorithmus

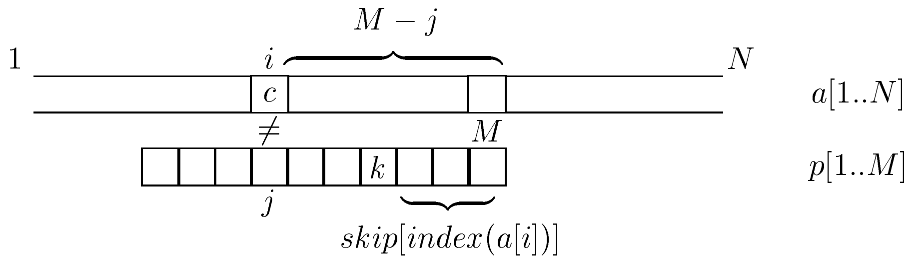
1. arbeitet rückwärts, vom Ende des Musters her.
2. besteht aus 2 Sprüngen (Teilen):
   (a) ,,KMP" rückwärts
   (b) Test, ob und wo ein Zeichen zum ersten Mal im Muster $p[1..M]$ vorkommt.

Definiere:

- Array $skip[0..|d|-1]$, wobei $|d|$ die Größe des Alphabets ist. $skip$ muß für jedes Zeichen definiert sein.
- **function** $index$ (c: **char**) : **integer**
  Diese liefert i für das i-te Zeichen des Alphabets (speziell: Blank=0).
- $$skip[index(c)] = \begin{cases} M & \text{, falls } c \neq p[j] \text{ für } j = 1...M \quad (c \notin p[1..M]) \\ M-j & \text{, falls } c = p[j] \text{ und } c \neq p[k] \text{ für } j < k \leq M \\ & (c \text{ kommt erstmals an Position } j \text{ vor}) \end{cases}$$

procedure $initskip$: initialisiert $skip$

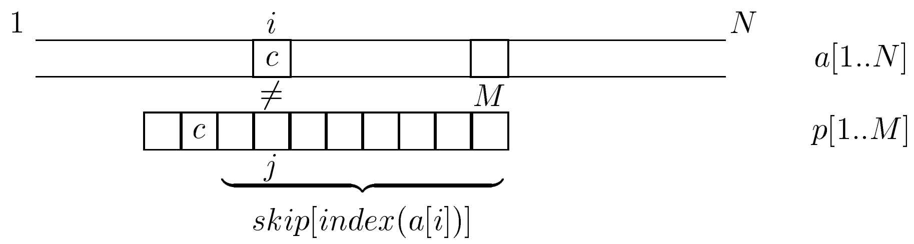
- $skip[.] = M$ für,,externe" Zeichen
- **for** $j = 1, ..., M$
  $skip[index(p[j])] := M - j$
  (das letzte Vorkommen wird gespeichert)



Verschieben um eine Position:

$$\begin{aligned} i &:= i + M - j + 1 \\ j &:= M \end{aligned}$$

Alternativ (besser):



$$\begin{aligned} i &:= i + M - j + skip[index(a[i])] \\ j &:= M \end{aligned}$$

„Typischer Fall":

$$\begin{aligned} i &:= i + skip[index(a[i])] \\ j &:= M \end{aligned}$$

```
function MisCharSearch : integer;
var i, j : integer;
begin
    i := M ; j := M; initskip;
    repeat
        if a[i] = p[j]
            then begin i := i - 1; j := j - 1 end
            else begin
                if M - j + 1 > skip[index(a[i])]
                then i := i + M - j + 1
                else i := i + skip[index(a[i])];
                j := M
            end
    until (j < 1) or (i > N);
    MisCharSearch := i + 1
end;
```

Boyer-Moore (original): Wähle den größten von den beiden Sprüngen.

Praxis:

- Teil 1 trägt wenig bei.
- Teil 2 wird meist allein verwendet.

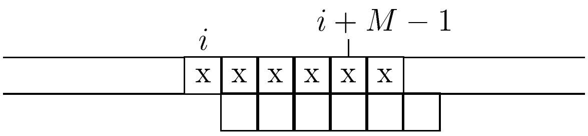
Komplexität des vollständigen BM-Algorithmus:

- maximal $(N+M)$ Zeichenvergleiche
- im Mittel: Komplexität $O\left(\frac{N}{M}\right)$, falls Zeichenalphabet groß und Muster kurz. („jeweils Skip um M Zeichen")
- kein Gewinn für Bitstring
- Ausweg: bilde Bitgruppen (analog 7-Bit ASCII)

5.1.4 Rabin-Karp-Algorithmus

Prinzip: Hashing

- Wähle Hash-Funktion, die jedem Teilstring der Länge M eine Zahl zuordnet. („Signatur" des Teilstrings)
- möglichst keine Adreßkollisionen
- Berechne Hashing-Wert für Muster $p[1..M]$.
- für jede Position $i = 1, ..., N$:
  - berechne Hashing-Wert $h(i)$ für Zeichen $a[i..i+M-1]$ (möglichst aus h(i-1))
  - bei Gleichheit: überprüfe Zeichen für Zeichen

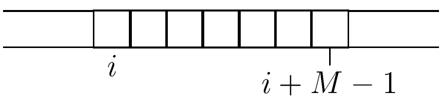


Komplexität:

- average case: $O(N+M)$
- worst case: $O(N \cdot M)$ (wie bei Hashing sehr selten!)

Anmerkung: Es fehlt der volle Symbolvergleich.

$$h(x) = x \bmod q \qquad (q \text{ große Primzahl})$$



$a[i]$: Zahl (statt character)

x_i : integer: d-adische Zahl mit d = Größe des Alphabets

$$\begin{aligned} x_i &= a[i]d^{M-1} + a[i+1]d^{M-2} + \ldots + a[i+M-1] \\ x_{i+1} &= (x_i - a[i]d^{M-1}) \cdot d + a[i+M] \end{aligned}$$

Die mod-Operation kann überall „angewendet" werden, so daß $h(x_{i+1})$ auf $h(x_i)$ zurückgeführt wird.

$$\begin{aligned} h(x_{i+1}) &= x_{i+1} \bmod q \\ &= \{(\underbrace{x_i \bmod q}_{h(x_i)} \underbrace{+ d \cdot q}_{(*)} - a[i]d^{M-1} mod q) \cdot d + a[i+M]\} \bmod q \end{aligned}$$

(*): stellt sicher, daß alle Zwischenergebnisse positiv bleiben, so daß die mod-Operation funktioniert.

```
function rksearch : integer;
const q = 33554393; d = 32;
var h1, h2, dM, i : integer;
begin
    dM := 1; for i:= 1 to M - 1 do dM := (d * dM) mod q;
    h1 := 0; for i:= 1 to M do h1 := (h1*d + index(p[i])) mod q;
    h2 := 0; for i:= 1 to M do h2 := (h2*d + index(a[i])) mod q;
    i := 1;
    while (h1 <> h2) and (i <= N-M) do
    begin
        h2 := (h2 + d*q - index(a[i])*dM) mod q;
        h2 := (h2*d + index(a[i+M])) mod q;
        i := i + 1
    end;
    rksearch := i;
end;
```

5.2 Approximatives String Searching

(deutsch: Approximativer symbolischer Vergleich)

Motivation:

- Schreibfehler („phonetische" Schreibweise)
- DNA-Sequenzen

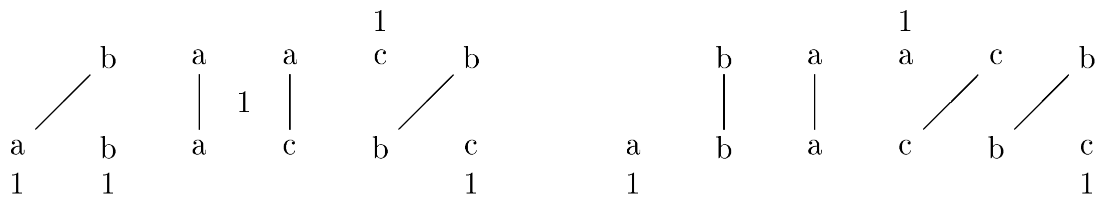
Editierdistanz (Levenshtein-Distanz)

- Deletions
- Insertions
- Substitutions

Randbedingungen:

- Überschreiben ist verboten.
- Die Monotonie der beiden Symbolfolgen wird beibehalten.

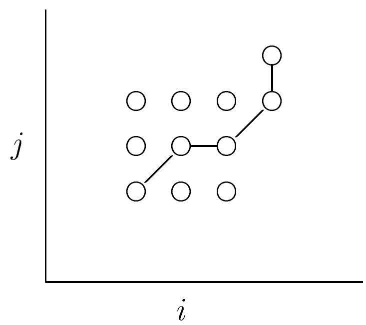
Beispiel: A = baacb; B = abacbc
Zuordnung: andere und bessere Zuordnung:



Aufgabe:
Bestimme die Zuordnung mit den minimalen Kosten. Der Einfachheit halber werden Einheitskosten für Deletions, Insertions und Substitutions festgelegt, so daß gilt:

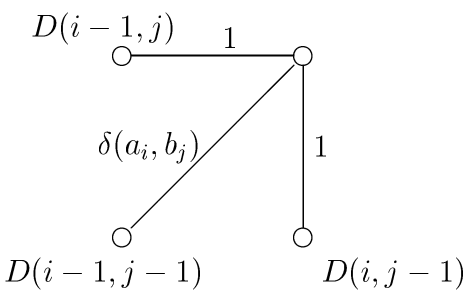
$$\text{Kosten} = \text{Anzahl der Editierungen}$$

Eine Zuordnung ist dabei ein „Pfad" („Spur") zwischen $x_1 ... x_i ... x_I$ und $y_1 ... y_j ... y_J$

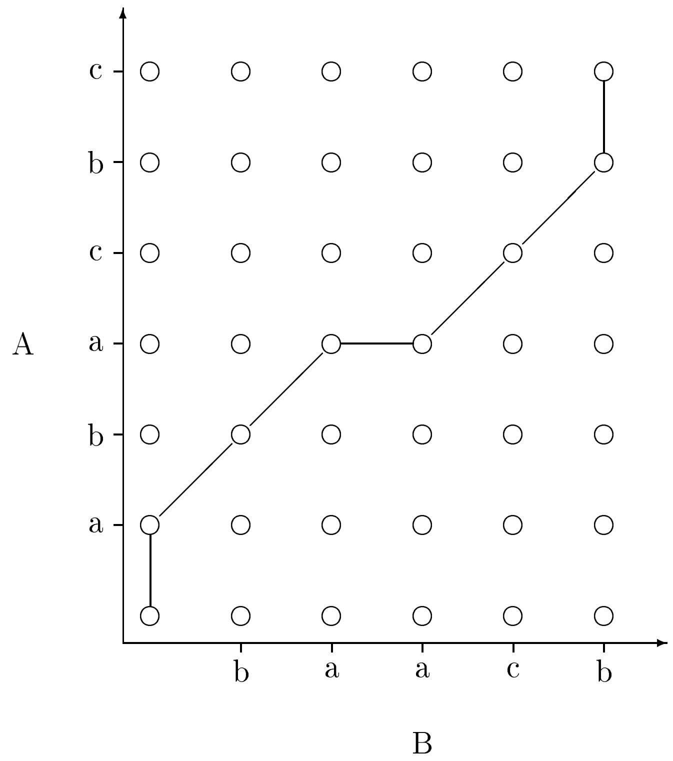


Definiere Hilfsgröße:

$D[i, j] :=$ Kosten der besten Zuordnung für die Teilstrings $x_1 - x_i$ und $y_1 - y_j$



Beispiel: A = abacbc; B = baacb



Rekursionsgleichung der dynamischen Programmierung:

$$D[i,j] = \min\{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+\delta(a_i,b_j)\}$$
$$\text{für } 0 < i \leq I,\ 0 < j \leq J$$
$$\text{mit } \delta(x,y) = \begin{cases} 0 & x = y \\ 1 & x \neq y \end{cases}$$

Initialisierung:

$$\begin{aligned} D[0,0] &= 0 \\ D[0,j] &= j \text{ , für } 1 \leq j \leq J \\ D[i,0] &= i \text{ , für } 1 \leq i \leq I \end{aligned}$$

Auswertung in zwei Schleifen:
for $i = 1, .., I$
  for $j = 1, .., J$

$\Rightarrow$ Komplexität:

- Zeit: $O(I \cdot J)$
- Platz: $O(I \cdot J)$
- ohne explizites Berechnen der Zuordnung: Platz: $O(\min(I,J))$

Vergleiche: Gesamtzahl der möglichen Zuordnungen: $N \simeq 2^I$ bis 3^I
grobe Abschätzung, aber in jedem Fall exponentiell

Approximatives Pattern Matching mittels stochastischer endlicher Automaten

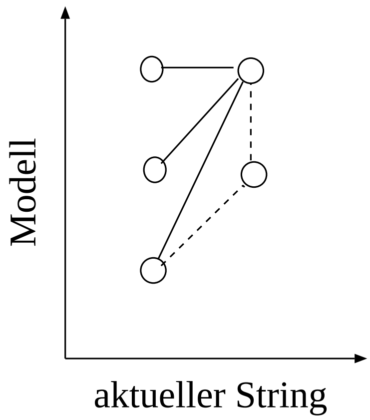


Abbildung 17: Verfeinerung des Editierdistanz-Ansatzes

Schritte:

1. Die Symmetrie wird aufgegeben.
2. In das Modell werden die drei Fehlerarten (Deletions, Insertions,Substitutions) integriert.
3. Ein Pfad durch das Modell entspricht einem beobachteten (aktuellen) String.

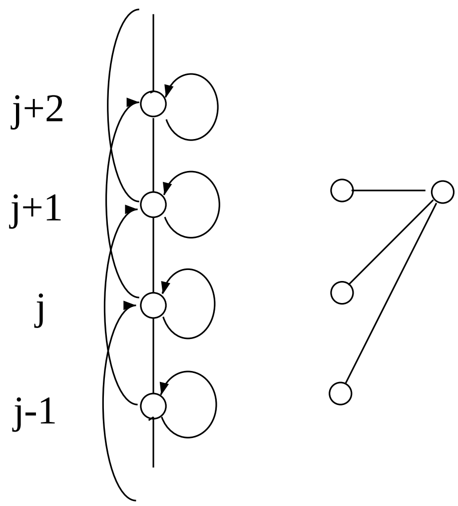


Abbildung 18:

Definition der Kosten:

1. Insertions und Deletions $t(j, j')$
2. Verwechslungen $d(x_i, j)$, wobei x_i i-tes Symbol im String

Dynamische Programmierung für String $x_1, ..., x_i, ..., x_I$

$$D(i,j) = d(x_i, j) + \min_{j'}\{D(i-1, j') + t(j, j')\}$$

Kosten: $d()$ und $t()$ können auf negativen Logarithmus von Wahrscheinlichkeiten zurückgeführt werden.

Erweiterung: „regelmäßige" Struktur des Automaten $\Rightarrow$ beliebige Struktur

5.3 Tries

„Lexikon-Bäume" [Wood p.200-228]

- allgemein
- Binary Trie (benutzt binäre Repräsentation der Character)
- Compact Binary Trie (add skips and counts)
- Compact Trie / Suffix Trie
- Patricia Trie

Trie („alphabetischer Baum")

- Name: Wortspiel: Mischung von „tree" und „trie" aus (information) re*trie*val.
- Prinzip: Söhne werden über Symbole des Alphabets adressiert (d.h. alphabetischer Baum).

Anwendungen:

- Spell Checker
- Pattern Matching für Lexikon-Anfragen

Trie-Varianten

- Multiway trie (Vielweg-Baum): Standard
- Binary Trie (Radix Trie ?): verwendet die Bit-Darstellung der Symbole

- Compact Binary Trie: In der Praxis hat man viele Knoten mit einem Sohn: Der entsprechende Pfad wird komprimiert und die komprimierte Kante mit dem entsprechenden Teilstring beschriftet.[Wood p. 224]
- Suffix Trie:
  Suffix := Teilstring am Ende eines vorgegebenen Strings dargestellt durch einen Trie.
  Anwendungen:
  - Darstellung eines (alphabetischen) Lexikons
  - Anfrage, ob ein Wort im Lexikon ist.

  Unterschied zur Situation für String Processing: Der Text ändert sich dynamisch.
- PATRICIA Trie: spezieller compact trie mit skip counts:
  ,,Practical Algorithm To Retrieve Information Coded In Alphanumeric"

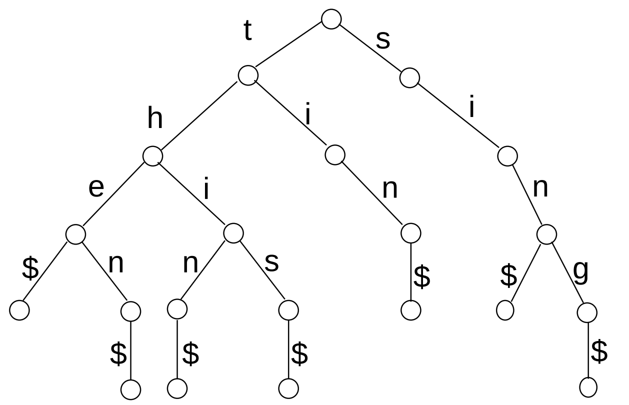


Abbildung 19: Beispiel: the, then, thin, this, tin, sin, sing

Suffix Trie:

Beispiel: ,,Text" : aabbabb

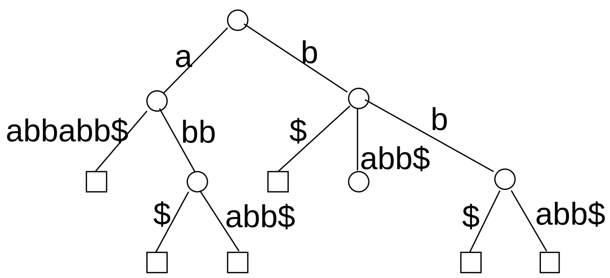


Abbildung 20: Trie aller Suffixe : kompakter (komprimierter) Suffix Trie

Variante: Verweise auf die Position im Text (nicht eindeutig!)

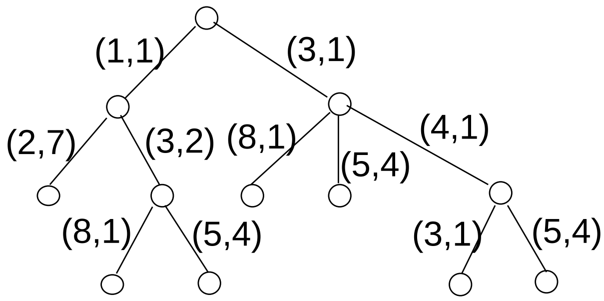


Abbildung 21:

PATRICIA trie:

1. Verzweigungsabfrage: ein einzelnes Symbol
2. Welches Symbol?
   - festgelegt durch skip count für jeden Knoten
   - Summe über alle skip counts längs des Pfades von der Wurzel zum Knoten = Position des aktuellen Symbols
3. Nach erreichen des Blattes: voller Vergleich aller Symbole nötig

Beispiel:

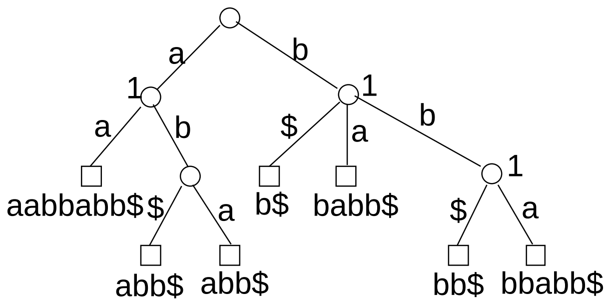


Abbildung 22: Beispiel für PATRICIA trie