

# Kap. 4: Suchen in Datenmengen



Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

9. VO *TEIL 2* DAP2 SS 2009 14. Mai 2009

# Suchen in Datenmengen

- **Motivation:** Suchen in Datenbanken, in Wörterbüchern, im WWW, ...

- **hier:** nur elementare Suchverfahren (nur Vergleichsoperationen erlaubt)
- **später:** auch arithmetische Operationen (um aus Suchschlüssel Speicheradresse zu berechnen)

# Kap 4.1: Suchen in sequentiell gespeicherten Folgen

- **Gegeben:** Datenelemente sind in Feld  $A[1], \dots, A[n]$  gespeichert
- Schlüssel sind ansprechbar über  $A[i].key$

- **Aufgabe:** Suche in  $A$  ein Element mit Schlüssel  $s$ , d.h. ein  $i$  mit  $A[i].key = s$

# Überblick

- Lineare Suche

- Binäre Suche
- Exkurs: Insertion-Sort

- Geometrische Suche

# Motivation

„Warum soll mich das interessieren?“

Suchen ist „das Wichtigste“ überhaupt!

„Warum soll ich heute hier bleiben?“

Beliebte Klausurfragen!!!

...und ein interessantes Rätsel...

# Ein kleines Rätsel

- Wie oft muss man ein Blatt Papier (0,1 mm dick) falten, bis es eine Dicke erreicht, die der Entfernung von Erde zu Mond entspricht?  
(363.258 km = 363.258.000.000 mm)

# 4.1.1 Lineare Suche

„Naives Verfahren“

- **Idee:** Durchlaufe  $A$  von vorn nach hinten und vergleiche jeden Schlüssel mit dem Suchschlüssel  $s$ , solange bis  $s$  gefunden wird.

- **Analyse:**
- **Best Case:**  $C_{\text{best}}(n)=1$
- **Worst Case:**  $C_{\text{worst}}(n)=n$

# 4.1.1 Lineare Suche ff

- **Average Case:** Annahme: jede Anordnung ist gleichwahrscheinlich:

$$C_{avg}(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

## Diskussion:

- lange Suchzeit → nur für kleine  $n$  empfehlenswert
- einfache Methode auch für einfach verkettete Listen geeignet



# 4.1.2 Binäre Suche / Idee

**Annahme:** die Liste ist bereits sortiert:

$$A[1].key \leq A[2].key \dots \leq A[n].key$$

**Divide-and-Conquer:** Vergleiche den Suchschlüssel  $s$  mit dem Schlüssel des Elements in der Mitte  $m$

- Falls  $A[m].key == s$ ? Treffer  $\rightarrow$  STOP
- Falls  $s$  ist kleiner:
  - Durchsuche Elemente links von  $m$
- Falls  $s$  ist größer:
  - Durchsuche Elemente rechts von  $m$

# BinarySearch (nicht-rekursiv)

**Procedure** BinarySearch(A,s,l,r)

(1) **var** Index m

(2) **while**  $l \leq r$  **do** {

(3)     m :=  $\lfloor (l+r)/2 \rfloor$      // Mitte bestimmen

(4)     **if** A[m].key==s **then** return m

(5)     **if** A[m].key>s **then** r:=m-1

(6)     **else**                     l:=m+1     // A[m].key<s

(7) }

(8) return 0

Aufruf: BinarySearch(A,s,1,n)

# Skript-Variante: BinarySearch (nicht-rek.)

**Procedure** BinarySearch(A,s,l,r)

(1) **var** Index m

(2) **repeat**

(3)     m :=  $\lfloor (l+r)/2 \rfloor$

(4)     **if** s < A[m].key **then** r := m-1

(5)     **else** l := m+1

(6) **until** s == A[m].key **or** l > r

(7) **if** s == A[m].key **then** return m

(8) **else** return 0

Aufruf: BinarySearch(A,s,1,n)

# Analyse von BinarySearch

Annahmen:

- Skript-Variante: nicht-rekursiv
- die Daten sind schon sortiert
- wir zählen nur die Vergleiche in Zeile (4)
- Annahme:  $n=2^k-1$  für geeignetes  $k$

Best Case:  $C_{\text{best}}(n)=1=\Theta(1)$

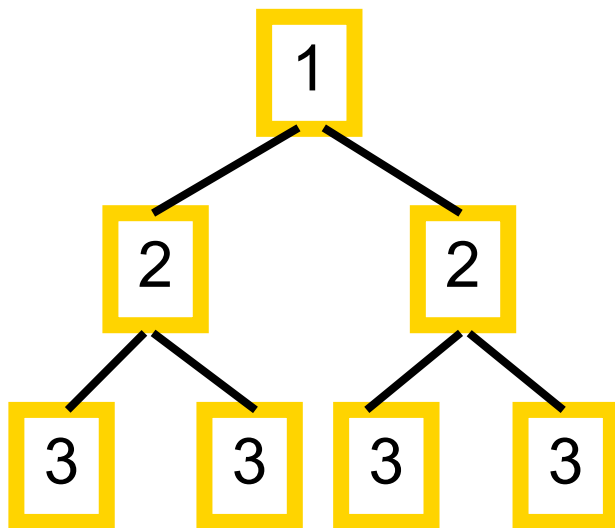
Worst Case:  $C_{\text{worst}}(n)=?$

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

für erfolgreiche  
und erfolglose Suche

**Worst Case:  $C_{\text{worst}}(n) = \log(n+1) = \Theta(\log n)$**

Position:	1	2	3	4	5	6	7
Anzahl Vergleiche:	3	2	3	1	3	2	3



weiter: nächste VO

Anzahl Schritte	Anzahl Positionen	Summe
1	$1=2^0$	1
2	$2=2^1$	3
3	$4=2^2$	7
k	$2^{k-1}$	$\sum 2^{i-1}$