

Kapitel 4: Suchen in Datenmengen

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

9. VO

2. Mai 2006

DAP 2

**Am Do., 4.Mai keine Vorlesung, sondern
Übungstest für aktive Übungsgruppen**

Falls es Serverprobleme mit
ls11-www.cs.uni-dortmund.de
gibt, dann finden Sie alle Unterlagen
(Übungsblätter, Folien, Skript) auf
ls2-www.cs.uni-dortmund.de/~droste/DAP2

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?“

Wir zaubern heute!

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)

- **var** Index m
- **repeat**
 - (1) $m := \lfloor (l+r)/2 \rfloor$
 - (2) **if** $s < A[m].key$ **then** $r := m - 1$
 - (3) **else** $l := m + 1$
 - (4) **until** $s == A[m].key$ **or** $l > r$
 - (5) **if** $s == A[m].key$ **then** return m
 - (6) **else** return 0

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

Analyse von BinarySearch

Annahmen:

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

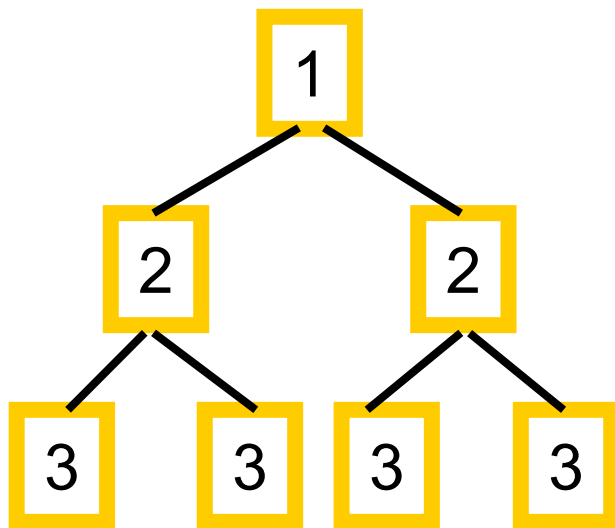
Best Case: $C_{\text{best}}(n)=1=\Theta(1)$

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

Worst Case Analyse von BinarySearch

Wie lange dauert es, das Element an Position $i \in \{1, 2, \dots, n\}$ zu finden?

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



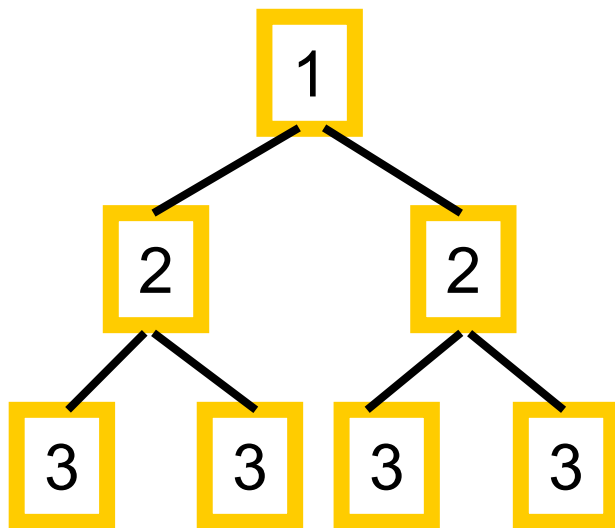
Anzahl Schritte	Anzahl Positionen	Summe
1	$1=2^0$	1
2	$2=2^1$	3
3	$4=2^2$	7
k	2^{k-1}	$\sum 2^{i-1}$

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

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

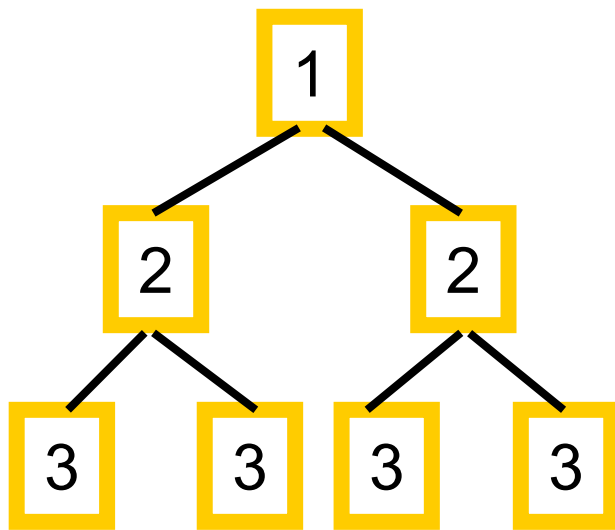


Anzahl Schritte	Anzahl Positionen	Summe
1	$1 = 2^0$	1
2	$2 = 2^1$	3
3	$4 = 2^2$	7
k	2^{k-1}	$\sum 2^{i-1}$

Average Case Analyse von BinarySearch

Durchschnittliche Zeit für erfolgreiche Suche:

$$\begin{aligned} C_{avg}(n) &= \frac{1}{n} \sum_{i=1}^k i 2^{i-1} = \dots = \\ &= \frac{n+1}{n} \log(n+1) - 1 = \theta(\log n) \end{aligned}$$



Anzahl Schritte	Anzahl Positionen	Produkt	Summe
<i>1</i>	$1=2^0$	<i>1</i>	<i>1</i>
<i>2</i>	$2=2^1$	<i>4</i>	<i>5</i>
<i>3</i>	$4=2^2$	<i>12</i>	<i>17</i>
<i>k</i>	2^{k-1}	$k 2^{k-1}$	$\sum i 2^{i-1}$

Binäre Suche / Diskussion

- Binäre Suche ist für große n sehr empfehlenswert!
- z.B. $\log 1000 = 10$; $\log 10^6 \approx 20$
- Binäre Suche ist nur sinnvoll, wenn sich die Daten nicht allzu oft ändern

- In der Praxis oft Interpolationssuche: wähle m als erwartete Position des Suchschlüssels
- hier ist $C_{\text{avg}}(n) = \Theta(\log \log n)$ z.B. $\log \log 10^6 \approx 5$
- aber $C_{\text{worst}}(n) = \Theta(n)$, z.B. „AAAA...AAABZ“

Binäre Suche bei InsertionSort

(1) **for** $k:=2,\dots,n$ {

(2) $s:=A[k].key$

(3) $i:=k$

(4) **while** $i>1$ and $A[i-1].key>s$ {

(5) $A[i].key:=A[i-1].key$

(6) $i:=i-1$

(7) }

(8) $A[i].key:=s$

(9) }

Ersetze die Suche nach
der richtigen Position für
 $A[k]$ durch
 $\text{BinarySearch}(A,s,1,k-1)$

jedoch: return l statt 0 ,
falls s nicht gefunden wird

bisheriges Insertion-Sort

- **Anzahl der Schlüsselvergleiche:**

$$C_{\text{best}}(n) = \Theta(n) \text{ und } C_{\text{avg}}(n) = C_{\text{worst}}(n) = \Theta(n^2)$$

- **Anzahl der Datenbewegungen:**

$$M_{\text{best}}(n) = \Theta(n) \text{ und } M_{\text{avg}}(n) = M_{\text{worst}}(n) = \Theta(n^2)$$

BinarySearch Insertion-Sort

- **Anzahl der Schlüsselvergleiche:**

$$C_{\text{best}}(n) = \Theta(n) \text{ und } C_{\text{worst}}(n) = \Theta(n \log n)$$

$$\begin{aligned} C_{\text{worst}}(n) &= \sum_{i=1}^{n-1} \lceil \log(i+1) \rceil = \sum_{i=2}^n \lceil \log i \rceil \\ &\leq \sum_{i=2}^n (\log i + 1) = \log(n!) + n - 1 \\ \text{Stirling-Formel} &\approx n \log n - n \log e + O(\log n) \\ &\approx n \log n - 1,4427n \end{aligned}$$

BinarySearch Insertion-Sort

- **Anzahl der Datenbewegungen:**

$$M_{\text{best}}(n) = \Theta(n) \text{ und } M_{\text{avg}}(n) = M_{\text{worst}}(n) = \Theta(n^2)$$

- **Eigenschaften:**

- in situ? 😊
- adaptiv? 😊
- stabil? ⚡

4.1.3 Geometrische Suche / Idee

Annahme: die Liste ist bereits sortiert und sei 2^k die größte Zweierpotenz mit $2^k \leq n$

Vergleiche die Daten an den Positionen $2^0, 2^1, 2^2, \dots, 2^k$ so lange mit dem Suchschlüssel s

1. bis s gefunden wurde (dann STOP) oder
2. ein Datum $A[2^m].\text{key} > s$ gefunden wurde für ein m

In diesem Fall: Starte BinarySearch auf den Plätzen $2^{m-1}+1, \dots, 2^m-1$

Falls $A[2^k] < s$, dann BinarySearch auf $2^{k+1}, \dots, n$

Analyse von Geometrischer Suche

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

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

- in Startphase $k+1 \leq \log n + 1$ Vergleiche
- jede der beiden Phasen mittels BinarySearch benötigt höchstens $\lceil \log(n+1) \rceil$ Vergleiche
- insgesamt sind dies höchstens doppelt so viele Vergleiche wie BinarySearch

Geometrischer Suche / Diskussion

- Geometrische Suche ist für Daten mit $s < A[2^m].key$, für m klein, sehr effizient.
 - Dann ist die Anzahl der Vergleiche durch $2m+1$ nach oben beschränkt
 - besser als BinarySearch, falls $m \leq n^{1/2}$
-
- Geometrische Suche auch gut geeignet, wenn der Suchbereich unbekannt ist:
suche einfach in $2^0, 2^1, 2^2, \dots$ bis s gefunden wurde oder ein Element mit $A[2^m].key > s$.
Dann BinarySearch auf $2^{m-1}+1, \dots, 2^m-1$