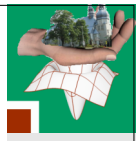


Informatik I
Suchen

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Problemstellung

- gegeben ist eine Menge von Datensätzen $\{A_1,...,A_n\}$
  - suche nach einem oder mehreren bestimmten Datensätzen
- Anmerkung:
  - häufig sind die Elemente **`A[i]`** komplexe Objekte bestehend aus vielen Attributen (= Members) (z.B. Autos),
  - sie enthalten ein Feld **`A[i].key`**, nach dem gesucht wird (z.B. Leistung); dieses Feld heißt Schlüssel (*Key*)
- Betrachte ab jetzt o.B.d.A. nur noch die Folge der Keys:
  - Folge F = $(A_1,...,A_n)$, die A_i sind die *Keys* der Datensätze
- Allgemeine Spezifikation für das Suchproblem:
  - Eingabe: Array A, gesuchtes Element x
  - Ausgabe: Index i mit A[i] = x, falls x in A

Lineare Suche

- wenn nichts über die (An-)Ordnung der Elemente im *Container* bekannt ist, kann nur die lineare Suche benutzt werden:
  - Entferne der Reihe nach die Elemente aus dem Container, bis dieser leer ist oder ein Element mit der gewünschten Eigenschaft gefunden wurde
- Beispiel: suche gegebene Telefonnummer in einem Telefonbuch
- zwei Grundoperationen
  - Prüfen, ob Container leer ist: isEmpty()
  - "Nächstes" Element aus dem Container herausgreifen: getNext()
- Aufwand
  - worst case: n Schleifendurchläufe
  - average case: $\frac{1}{n}(1+2+\ldots+n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} \approx \frac{n}{2}$

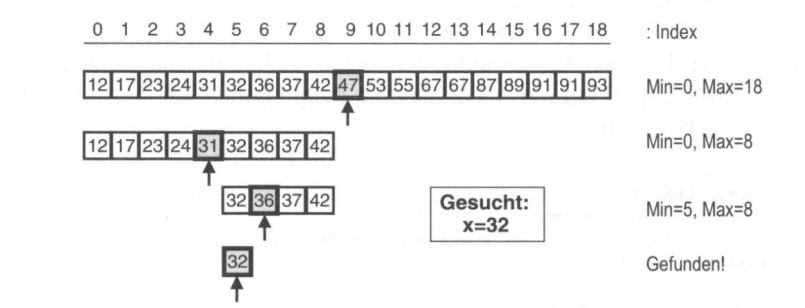
Binäre Suche

- Ordnung: die N Elemente seien nun aufsteigend sortiert (geordnet), d.h.

$$\forall i, 0 \leq i < n-1 : A_i \leq A_{i+1}$$

- Beispiel: suche nach einem Namen im Telefonbuch
- Lösungsstrategie: "Intervallhalbierung"
  - Schlage Telefonbuch in der Mitte auf, vergleiche gesuchten Namen mit einem Namen auf der Seite
  - Entscheide, in welcher Hälfte des Telefonbuches sich der gesuchte Name befindet (halbiert Suchraum grob gerechnet)
  - wiederhole Verfahren mit dieser Hälfte, bis richtige Seite gefunden ist

Beispiel



Algorithmus ...

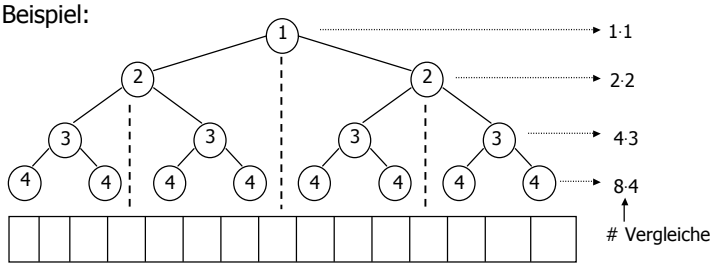
```
def binsearch( A, k, l = 0, r = -17 ):
    """ Return index i such that l <= i <= r and
        A[i] == k, or, if no such i exists,
        returns -1.
    """
    if r  < 0:
        return binsearch( A, k, 0, len(A) )
    m = (l + r) / 2
    if k < A[m]:
        return binsearch( A, k, l, m-1 )
    if k > A[m]:
        return binsearch( A, k, m+1, r )
    return m
```

… und nicht-rekursiv

```
def binsearch( A, k ):
    l = 1
    r = len(A) - 1
    while l <= r:
        m = (l + r) / 2
        if k < A[m]:
            r = m - 1
        elsif k > A[m]:
            l = m + 1
        else:
            return m
    return l-1
```


Analyse

- Worst case: $\log n$ Vergleiche (dito für Anzahl Mittelwertber.)
- Average case:
  - Annahme: $n + 1 = 2^k$
  - Beispiel:



  - Durchschnittlich: $\frac{1}{n}\sum_{i=1}^{k} i2^{i-1} = \frac{1}{n}((k-1)2^k + 1) \approx \log n$

Beweis für $\sum_{i=1}^{k} i2^{i-1} = (k-1)2^k + 1$

$$\sum_{i=1}^{k} i2^{i-1} = \left.\begin{array}{l} 2^0 + \\ 2^1 + 2^1 + \\ 2^2 + 2^2 + 2^2 + \\ \ldots \\ 2^{k-1} + \ldots \quad \ldots + 2^{k-1} \end{array}\right\} k$$

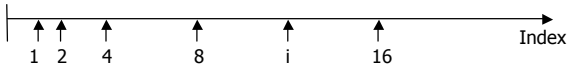
$$= \sum_{0}^{k-1} 2^i + \sum_{1}^{k-1} 2^i + \ldots + \sum_{k-1}^{k-1} 2^i$$

$$= \begin{array}{l} (2^k-1) - \\ (2^0-1) \end{array} + \ldots + \begin{array}{l} (2^k-1) - \\ (2^l-1) \end{array} + \ldots + \begin{array}{l} (2^k-1) - \\ (2^{k-1}-1) \end{array}$$

$$= k2^k - \sum_{i=0}^{k-1} 2^i = k2^k - (2^k - 1) = (k-1)2^k + 1$$

Exponentielle Suche

- Situation:
  - n sehr groß
  - Gesuchtes i, mit $A_i = k$, ist relativ klein
- Idee: suche zunächst "rechten Rand" r, so daß $k < A_r$
- Algo:




```
r = 1
while A[r] < k:
    r *= 2
binsearch( A, k, r/2, r )
```

- Analyse:
  - rechten Rand suchen: $\lceil \log i \rceil$ viele Schleifendurchläufe
  - Binärsuche: $\sim \lceil \log i \rceil$

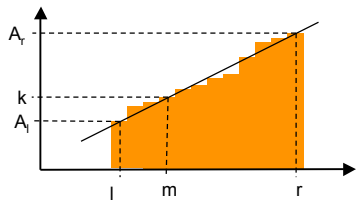
Interpolation search

- Idee:
  - Bei Suche nach "Dix" oder "Zachmann" im Telefonbuch schlagen wir es eher weiter vorne bzw. weiter hinten auf
  - Schätze Position des gesuchten Keys im Array
- Erinnerung: bei binärer Suche wird noch zu durchsuchendes Index-Intervall bei $m = l + \frac{1}{2}(r - l)$ unterteilt
- Ersetze Faktor 1/2 durch

$$m = l + (r - l)\frac{k - A_l}{A_r - A_l}$$

- Algo analog zu Binärsuche

- Klar ist: wenn Keys halbwegs linear aufsteigend sind, liegt dieses interpolierte m sehr dicht am gesuchten Key



- Voraussetzung:
  - Ordnungsrelation auf den Keys reicht nicht!
  - Man muß auf den Keys rechnen können
- Aufwand (o.Bew.)
  - Average case: ~ log log n
  - Worst case: ~ n (!)

Key-Indizierte Suche

- Triviale Lösung, falls Key-Menge klein
- Idee: Speichere Daten in einem Array "mit Lücken"
  - `A[k]` enthält Datensatz mit Key `k` (inkl. "Nutzdaten")
  - unbenutzte `A[i]` enthalten `None`
- Suche nach Key k ist trivial: liefere `A[k]`
- Bedingungen:
  - Wertebereich der Keys muß im voraus bekannt sein
  - Datensätze mit gleichen Keys klappt nicht so ohne weiteres
- Verallgemeinerung: Hash-Tables (später)

Anforderungen an Container

- die Elemente sind an Positionen gespeichert
- über die Positionen hat man direkten Zugriff auf die Elemente
- die Elemente in den Positionen sind entsprechend ihrer Ordnung im Behälter plaziert
- ein sortiertes Array erfüllt diese Bedingungen

Lineare Suche ↔ binäre Suche

- bei großen Datenmengen ist binäre Suche wesentlich effizienter
  - Verdoppelung der Datenmenge
    - lineare Suche: doppelter Aufwand
    - binäre Suche: ein weiterer Vergleich
- bei kleinen Datenmengen (n≈10) ist lineare Suche schneller
- nicht jeder Behälter ist für binäre Suche geeignet
- muß man häufig in einem unsortierten Behälter suchen, lohnt es sich, die Elemente einmal in ein Array zu kopieren und dieses zu sortieren. Dann kann man die binäre Suche wiederholt anwenden.