Suchen

lineare Suche, binäre Suche, divide and conquer, rekursive und iterative Algorithmen, geordnete Daten, *Comparable*

Welche Nummer hat Herr Meier ?

Telefonbuch Marburg

Enthält Einträge (Elemente) der Form :

Name, Vorname

Adresse

Tel.Nummer

- geordnet nach Name,

- bei Gleichheit nach Vorname

Name und Vorname sind Schlüssel

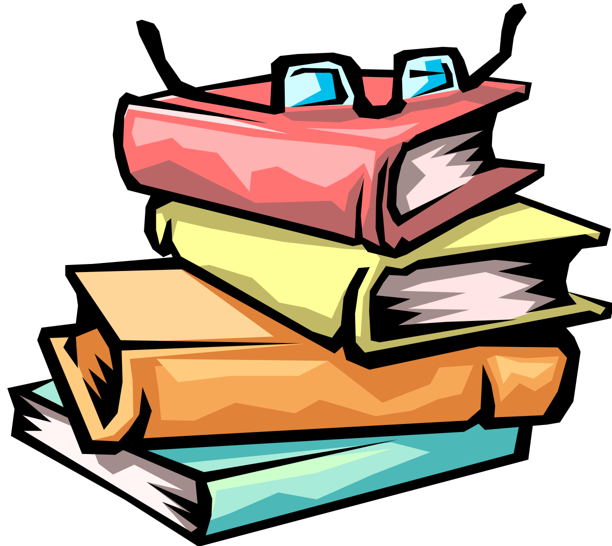
Wenn wir Namen und Vornamen wissen, finden wir den Eintrag von Herrn Meier sehr schnell.

- Wir schlagen das Telefonbuch etwa in der MItte auf
- Dann entscheiden wir uns, ob wir in der linken oder in der rechten Hälfte weitersuchen

Binäre Suche

© H. Peter Gumm, Philipps-Universität Marburg

Welche Nummer hat Herr Huber ?



Das Münchner Telefonbuch ist 10-mal so dick wie das Marburger.

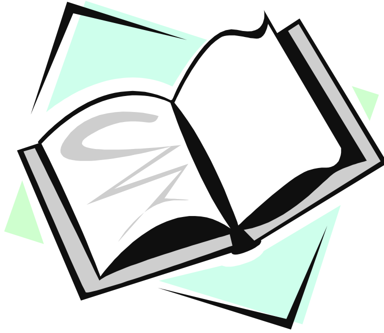
Brauchen wir 10-mal so lang, einen Teilnehmer aufzusuchen ?

Nein, wir brauchen nur unwesentlich länger – genau genommen brauchen wir im Schnitt nur 3 - 4 mehr Namen zu lesen und mit dem gesuchten zu vergleichen.

Binäre Suche ist sehr effizient

© H. Peter Gumm, Philipps-Universität Marburg

Wer hat die Nummer 1 32 13 ?



Telefonbuch Marburg

Die Information ist auch im Telefonbuch, ist aber nur schwer aufzufinden.

Einzige Möglichkeit :

Lineare Suche

- Im Telefonbuch von München dauert die Suche $\approx$ 10 mal so lange
- Der Aufwand für lineare Suche ist proportional der Anzahl N der Einträge
- Der Aufwand für binäre Suche ist proportional $\log_2 N$, dem binären Logarithmus von N

© H. Peter Gumm, Philipps-Universität Marburg

Divide et Impera

- Teile und Herrsche – *divide and conquer*
  - Zerlege das Problem in kleinere Probleme
  - löse die kleinen Probleme
  - aus den Lösungen erzeuge Lösung des ursprünglichen Problems
- Wenn dieses Prinzip anwendbar ist, erhält man effiziente Algorithmen
  - Beim Suchen des Teilnehmers mit Nummer 13213 können wir es nicht anwenden
  - Beim Suchen nach Namen im Telefonbuch ist es anwendbar.

© H. Peter Gumm, Philipps-Universität Marburg

Infrastruktur

- Für *lineare Suche* muss man die Einträge nur irgendwie aufzählen können
  - lineare Suche für (fast) alle Behälter möglich

- Für *binäre Suche* müssen
  - die Daten *geordnet* sein
  - *direkter Zugriff* auf einen mittleren Eintrag möglich sein
  - *Divide*:
    - Behälter muss sich in zwei analoge Behälter (logisch) zerlegen lassen – z.B.:
      - das Telefonbuch München von A – K,
      - das Telefonbuch München von L – Z.
    - Die Entscheidung, in welchem der beiden Teil-Behälter sich der gesuchte Eintrag befindet geht *schnell* - unabhängig von der Größe des Behälters

- Binäre Suche benötigt Infrastruktur :
  - *geordnete Daten*
  - *wahlfreier direkter Zugriff*

© H. Peter Gumm, Philipps-Universität Marburg

Ergebnisse von Suchalgorithmen

- Wenn ein gesuchtes Element gefunden wird kann das Ergebnis sein:
  - true – ja, das Element ist vorhanden
  - das Element selber
  - der Index (der Ort) wo das gesuchte Element gespeichert ist
- Wenn mehrere Elemente mit den Suchkriterien gefunden werden
  - die Anzahl
  - das erste Element
  - alle Elemente
- Wenn kein Element den Suchkriterien genügt
  - eine Ausnahme
  - ein Ersatzelement (dummy, sentinel), das anzeigt, dass kein richtiges Element vorhanden ist
    - -1 als Indexwert
    - Object null
    - double inf (infinity)
    - double NaN (not a number)
  - eine Fehlermeldung

Kein Eintrag mit Nummer 13213

© H. Peter Gumm, Philipps-Universität Marburg

Lineare Suche im Array

- Behältertyp: int [ ]
- Suchkriterium durch boolesche Methode
- Am Ende der Schleife zwei Möglichkeiten
  - Gefunden an Position i
    → Rückgabe i
  - Nicht gefunden,
    → Rückgabe -1

```
 9  // ein BeispielArray
10  private static int[] beispielArray = {2, 23, 16, 2, 17, 56, 2
11
12  // ein Beispielkriterium
13  private static boolean kriterium(int x){
14     return 10 < x & x < 20;
15  }
16
17  /** Lineare Suche
18   *  Sucht Index eines Elements, das das Kriterium erfüllt.
19   *  @param   zu durchsuchendes int-Array
20   *  @return  kleinster Index eines gefundenen Elementes,<br>
21   *  oder -1, falls kein solches Element vorhanden.
22   */
23  static int linSearch(int []a ){
24    int i=0;
25    while(i<a.length && !kriterium(a[i])) i++;
26    if (i<a.length) return i;
27    else return -1;
```

Klasse übersetzt - keine Syntaxfehler

gespeichert

© H. Peter Gumm, Philipps-Universität Marburg

Suche mit Exception

- Am Ende der Schleife die Möglichkeiten:
  - Gefunden an Position i
    - Rückgabe i
  - Nicht gefunden,
    - Exception

```
30  /** Lineare Suche
31   *  Exception, falls Element nicht gefunden.
32   */
33  static int exLinSearch(int[] a)
34                      throws NixDaException{
35    int i=0;
36    while(i<a.length && !kriterium(a[i])) i++;
37    if (i<a.length) // gefunden
38       return i;
39    else // nichts gefunden
40       throw new NixDaException();
41  }
```


© H. Peter Gumm, Philipps-Universität Marburg

Testen in BlueJ

- Rechte Maustaste auf Klassensymbol liefert statische Methode *`linSearch`* im Kontextmenü
- Eingabe eines Behälters als Parameter:
  - {1,13,19,25, 44}
  - SuchAlgorithmen.beispielArray
  - {1, 10, 100, 1000} { }
- Für die nächsten Aufrufe hat sich Bluej diese Eingabe gemerkt
  - Sie werden im Menü serviert

© H. Peter Gumm, Philipps-Universität Marburg

Verwandte der Suchalgorithmen

- Analog zum Suchen verlaufen
  - boolean ***exists()***
    - Gibt es ein Element, das das Suchkriterium erfüllt, oder nicht ?
  - int ***count()***
    - Anzahl der Elemente, die das Kriterium erfüllen
  - int[ ] ***filter()***
    - Erstelle Liste aller Elemente, die das Kriterium erfüllen
- All diese Algorithmen verlaufen analog zum Suchalgorithmus

SuchAlgorithmen

Klasse Bearbeiten Werkzeuge Optionen

Übersetzen Rückgängig Ausschneiden Kopieren Einfügen Suchen... Weitersuchen

```
54  /** Filtert die Elemente des Arrays,
55  * die das Kriterium erfüllen<br>
56  * @param  a zu filternder Array
57  * @return die Liste der erfüllenden Elemente
58  */
59  static int[] filter(int[] a){
60      int[] result = new int[count(a)];
61      int index=0;
62      for(int i=0; i < a.length; i++)
63        if (kriterium(a[i])){
64            result[index]=a[i];
65            index++;
66        }
67      return result;
68  }// end filter
```

Klasse übersetzt - keine Syntaxfehler

int length	3
[0]	11
[1]	17
[2]	19

BlueJ: Methodenaufruf

```
// Filtert die Elemente des Arrays,
// die das Kriterium erfüllen<br>
// @param  a zu filternder Array
// @return die Liste der erfüllenden Elemente
int[] filter(int[] a)
```

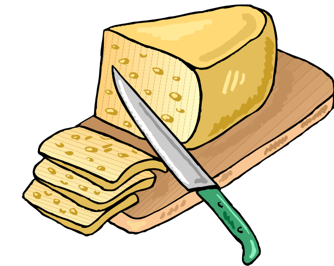
SuchAlgorithmen.filter ({1,2,5,11,17,19})

Ok Abbrechen

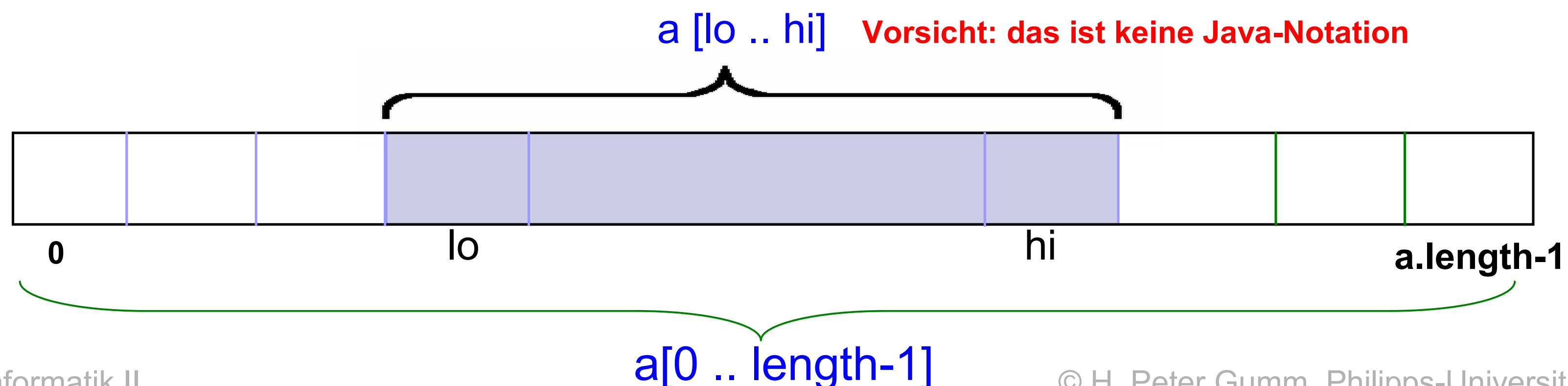
count benötigt, um Array-Größe festzulegen

© H. Peter Gumm, Philipps-Universität Marburg

Suche in Intervallen



- Rekursive Suchalgorithmen ...
  - zerlegen den Array in Teile (Abschnitte/Intervalle)
  - suchen (rekursiv) in den einzelnen Abschnitten
- Die Teilprobleme verlangen Suche in einem Intervall des Arrays
  - zwischen zwei Indexwerten lo und hi
  - Wir schreiben a [lo .. hi] für den Bereich (engl.: *slice*) des Arrays von lo bis hi.
- Daher verallgemeinern wir den Suchalgorithmus, so dass er in einem beliebigen Intervall eines Arrays suchen kann
  - jetzt sind die Teile (divide) von der gleichen Art wie der ursprüngliche Behälter
  - daher funktioniert „divide et impera“



© H. Peter Gumm, Philipps-Universität Marburg

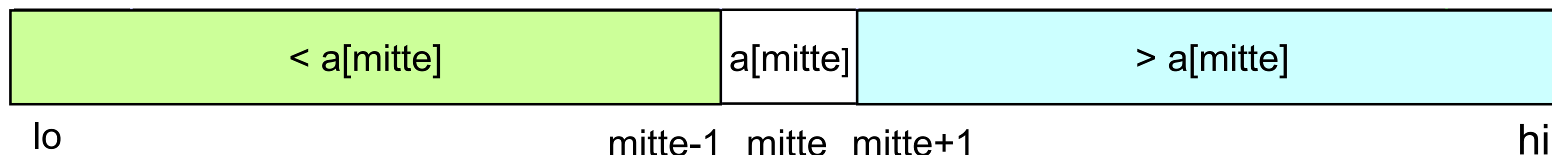
Binäre Suche

- *binSearch* :
  - wähle einen mittleren Index **i**
  - falls gesuchtes Element s
    - = a[i] ? fertig
    - < a[i] suche im vorderen Teil
    - > a[i] suche im hinteren Teil
- Implementierung
  - **Verallgemeinerung:**
    Suche im Intervall a[lo..hi]
    Hilfsfunktion auxBinSearchRec
  - Intervall leer ?
  - Bestimme (ungefähre) Mitte des Intervalls
  - Element s
    - gefunden ?
    - links von der Mitte ?
    - sonst: rechts der Mitte

```java
/** Sucht integer s in einer Liste a von integern
 *  Falls nicht vorhande ist Ergebnis -1, ansonsten
 *  der Index an dem s gefunden wurde */

static int binSearchRec(int[] a, int s){
  return auxBinSearchRec(a,0,a.length-1,s);
}

static int auxBinSearchRec(int[] a, int lo, int hi,int s){
  if (hi < lo) return -1;
  else{
      int mitte=(hi+lo)/2;
      if( s == a[mitte]) return mitte;
      else if (s < a[mitte])
          return auxBinSearchRec(a,lo,mitte-1,s);
      else return auxBinSearchRec(a,mitte+1,hi,s);
      }
  }
```



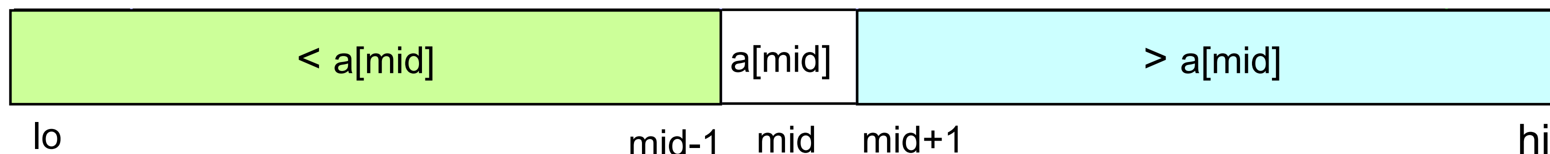
© H. Peter Gumm, Philipps-Universität Marburg

binSearch iterativ

-oder: Wie fängt man einen Tiger

- Suche in a[lo..hi]
  - Schiebe lo und hi zusammen, bis das Element gefangen ist
- Element in der Mitte gefunden?
  - Gebe Index zurück
- Sonst:
  - Links von der Mitte
    - Schiebe hi nach unten
  - Rechts von der Mitte
    - Schiebe lo nach oben

```
82  /** Iterative binäre Suche
83   * Sucht Index von s in int[]a
84   * Gibt -1 zurück, falls s nicht gefunden wird
85
86   static int binSearch(int[] a, int s){
87     int lo=0, hi=a.length-1; // Intervallgrenzen
88     while (lo <= hi){
89         int mid = (lo+hi)/2;
90         if(s==a[mid]) return mid;
91         if(s>a[mid]) lo=mid+1;
92         else hi=mid-1;
93       }
94     return -1; // Nicht gefunden
95  }
```

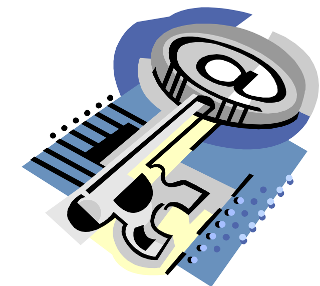


© H. Peter Gumm, Philipps-Universität Marburg

Der Schlüssel zu Frau Meier

- Meist sucht man komplexere Objekte als Zahlen
- Ein Telefonbucheintrag besteht aus
  - Name, Vorname, Adresse, Telefonnummer
- Geordnet ist das Telefonbuch aber nur nach
  - Name, und
  - Vorname (falls Namen gleich)
- Diese Kombination ist der *Such-Schlüssel*
- Ein *Schlüssel* soll
  - einen Datensatz möglichst eindeutig bestimmen
  - eine Ordnung tragen
- Die Daten werden nach Schlüsseln geordnet und aufgesucht.

Meier Brigitte
0 85 46 25 1 25
Am Kirchenfeld 15,
94113 Tiefenbach



© H. Peter Gumm, Philipps-Universität Marburg

Vergleich

- Binäre Suche erfordert
  - eine Sortierung der Daten
  - ein direkter Zugriff auf ein mittleres Element
- Sortierung erfordert,
  - zwei Datenelemente muss man vergleichen können
  - ersatzweise Schlüsselvergleich
- Vereinbarung
  - Daten implementieren Funktion *kleinerGleich*
  - Sortieralgorithmus verwendet zum Vergleich die Funktion *kleinerGleich*
- In Java
  - Schreibe *interface Ordnung*
    - verlangt boolesche Methode *kleinerGleich*
    - Klasse der Datenelemente *implements* Ordnung
    - ein Sortierprogramm funktioniert mit jeder Klasse, die Ordnung implementiert

```
public interface Ordnung{

  /** True falls <em>this</em> kleiner
   *  oder gleich Objekt o.
   *  Normalerweise muss man o auf den
   *  Typ von <em>this</em> casten.
   */
  public boolean kleinerGleich(Object o);

} // Ende des interface Ordnung
```


© H. Peter Gumm, Philipps-Universität Marburg

Comparable

- *Interface Comparable* im Java-API spezifiziert Methode
  - ***int compareTo(Object o)***
- Viele Java-Klassen implementieren dieses Interface, u.a.
  - ***String, Integer, Float, Character, Date, …***
- Dabei gilt immer:

$$x.\text{compareTo}(y) = \begin{cases} < 0 \ , & \text{falls } x < y \\ 0 \ , & \text{falls } x.\text{equals}(y) \\ > 0 \ , & \text{falls } x > y \end{cases}$$

- Java Methoden, die mit Comparable arbeiten, erwarten dieses Verhalten.
  - Manchmal beschränkt man sich auf die Werte -1, 0, +1,
  - Ansonsten gibt Comparable auch noch eine Art Differenz
    - bei **Integer**: **x.compareTo(y) = x- y**
    - Bei **Strings**: siehe API und Beispiel

```
"Anton".compareTo("Otto")
-14  (int)
"Anton".compareTo("Anna")
6  (int)
"Anton".compareTo("Antonia")
-2  (int)
```

- Leider kann man das geforderte Wohlverhalten einer Implementierung in Java nicht erzwingen.

© H. Peter Gumm, Philipps-Universität Marburg

Beispiel einer Implementierung

```
public class Eintrag implements Comparable{
  String name;
  String vorname;
  String ort;
  int telefonNummer;

  public Eintrag(String name,String vorname,String ort,int nummer){
        this.name=name;
        this.vorname=vorname;
        this.ort=ort;
        telefonNummer=nummer;
  }
  public int compareTo(Object o){
    String nameVorName=((Eintrag)o).name+((Eintrag)o).vorname;
    return (name+vorname).compareTo(nameVorName);
  }
}
```

Eintrag als Klasse mit einer Ordnung
Schlüssel: name + “ “ + vorname

compareTo erwartet ein *Object*
Wir *casten* es zuerst zu einem Eintrag

Hier benutzen wir die *compareTo*-Methode der Klasse String

© H. Peter Gumm, Philipps-Universität Marburg

Vorteile

- Eintrag implementiert Comparable
- Auf Eintrag[ ] sind Such- und Sortiermethoden aus java.util.Arrays anwendbar

Telefonbuch

Class Edit Tools Options

Compile Undo Cut Copy Paste Close Implementation

```
import java.util.Arrays; // enthält Sortiermethoden
                         // für Arrays
public class Telefonbuch{
  private String stadt;
  private Eintrag[] dasBuch;
  private int anzahl;

  public Telefonbuch(String stadt, int groesse){
      this.stadt = stadt;
      dasBuch = new Eintrag[groesse];
      anzahl=0;
  }

  private void sortiere(){
    Arrays.sort(dasBuch);  // sort ist Klassenmethode
  }
```

Class compiled - no syntax errors

saved

© H. Peter Gumm, Philipps-Universität Marburg

Binäre Suche - selbstgestrickt

- Allgemeine binäre Suche
- funktioniert für alle Objekttypen, die das *Interface Comparable* implementieren
  - wie z.B. mit Einträgen in einem Telefonbuch
  - Adressen
- **`a.compareTo(b)`**:
  - < 0 steht für a ◀ b
  - == 0 für a == b
  - > 0 für a ▶ b.
- Rückgabewert:
  - Index, an dem das Element gefunden wurde , oder
  - -1, falls nicht vorhanden

```
static int binSearch(Comparable[] liste,Comparable x){
  int lo = 0,mid=0;
  int hi=liste.length-1;
  while(lo <= hi){
    mid=(lo+hi)/2;
    int vergleich=x.compareTo(liste[mid]);
    if (vergleich==0) return mid;
    else if (vergleich < 0) hi=mid-1;
    else lo=mid+1;
  }
  // x nicht in liste :
  return -1;
}
```


© H. Peter Gumm, Philipps-Universität Marburg