



Suchen

lineare Suche, binäre Suche
geordnete Daten, interface
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**

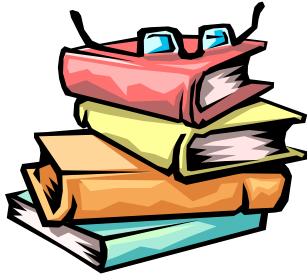
Wenn wir Namen und Vornamen wissen, finden wir sehr schnell den Eintrag von Herrn Meier sehr schnell.

- 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



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



Wer hat die Nummer 1 32 13 ?



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

Einzige Möglichkeit :

Lineare Suche

Telefonbuch Marburg

- Im Telefonbuch von München dauert die Suche 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



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 Telefonbuches ist es anwendbar.



Infrastruktur

- Für *lineare Suche* muss man die Einträge nur irgendwie aufzählen können
- Für *binäre Suche* müssen
 - die Daten geordnet sein
 - direkter Zugriff auf einen mittleren Eintrag möglich sein
 - Divide:
 - der Behälter – hier das Telefonbuch – sich in zwei analoge Behälter (logisch) zerlegen lassen – hier :
 - das Telefonbuch München von A – K,
 - das Telefonbuch München von L – Z.
 - Wir wissen, in welchem der beiden halb so großen Behälter sich der gesuchte Eintrag befindet
- Binäre Suche benötigt als Infrastruktur also
 - Geordnete Daten
 - Wahlfreier direkter Zugriff



Ergebnisse von Suchalgorithmen

- Wenn ein gesuchtes Element gefunden wird
 - true – das Element ist da
 - das Element selber
 - der Ort (Index) wo das gesuchte Element gespeichert ist
- Wenn mehrere Elemente mit den Suchkriterien gefunden werden
 - 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



Suche im Array

- Behälter: Ein int []
- Suchkriterium durch boolesche Methode
 - Am Ende der Schleife zwei Möglichkeiten
 - Nicht gefunden, d.h. i==a.length
 - Gefunden sonst

```
SuchAlgorithmen
Klasse Bearbeiten Werkzeuge Optionen
Übersetzen Zurück Ausschneiden Kopieren Einfügen Schließen Implementierung
static int[] beispielArray = {2,23,16,2,17,56,2,3,57,17};

// Ein Suchkriterium
private static boolean kriterium(int x){
    return x > 10 && x < 20;
}

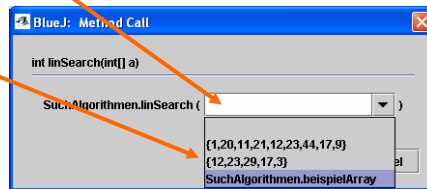
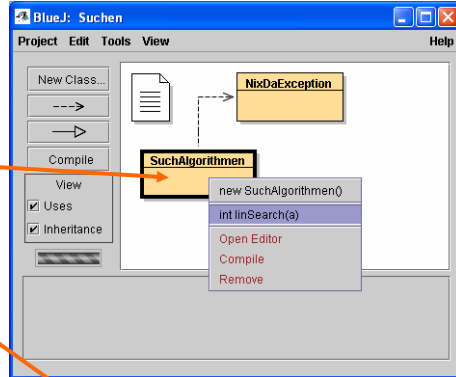
// finde ein Element, das die Kriterien erfüllt
static int linSearch(int [] a) throws NixDaException{
    int i=0;
    while (i < a.length && !kriterium(a[i])) i++;
    if (i==a.length) throw new NixDaException();
    else return a[i];
}

gespeichert
```



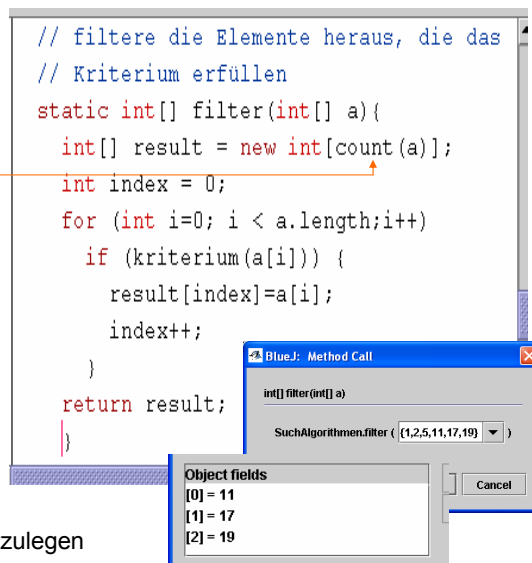
Test in BlueJ

- Rechte Maustaste auf Klassensymbol liefert statische Methode `linSearch` im Kontextmenü
- Eingabe eines Behälters als Parameter:
 - `{1,20,11,21,12,23,44,17,9}`
 - `SuchAlgorithmen.beispielArray`
 - `{12,20,29,17,3}`
 - `{}`
- Für die nächsten Aufrufe hat sich BlueJ diese Eingabe gemerkt
 - Sie werden im Menü serviert



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



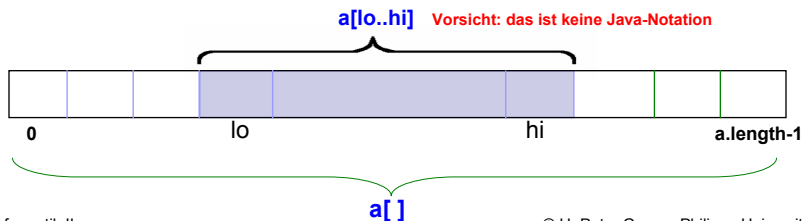
count benötigt um Array-Größe festzulegen



Intervalle



- Für die binäre Suche ...
 - ... wählen wir einen mittleren Index
 - ... vergleichen das mittlere Element mit dem gesuchten
 - Ist das gesuchte
 - kleiner, dann suchen wir im vorderen Teil
 - grösser, dann suchen wir im hinteren Teil des Arrays.
- Die Teilprobleme verlangen Suche in einem Intervall des Arrays
 - Die Elemente 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“



binSearchRec

- **binSearchRec**
 - ruft sofort Hilfsfunktion **auxBinSearchRec** auf
 - Diese arbeitet auf einem **Intervall lo..hi**
 - Ist das Intervall leer –
gehe -1 zurück
 - Bestimme die Mitte des
Intervalls
 - Element gefunden ?
 - Wenn links von der Mitte,
suche im unteren Teil,
 - sonst im oberen Teil

```
/** 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);
    }
}
```



binSearch ohne Rekursion

- 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

```
/** Iterative binäre Suche
 * Sucht Index von int s in int[] a
 * Gibt -1 zurück falls s nicht gefunden wird */

static int binSearch(int[] a, int s){
    int lo=0,mid=0; // lo : linke Grenze
    int hi=a.length-1; // hi : rechte Grenze
    while(lo <= hi){
        mid = (lo+hi)/2;
        if(s==a[mid]) return mid;
        if(s > a[mid]) lo=mid+1;
        else hi=mid-1;
    }
    return -1;
}
```



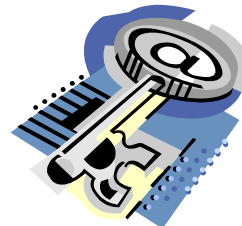
Der Schlüssel zu Frau Meier

- 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 Schlüssel_n geordnet und aufgesucht.

Meier Brigitte

0 85 46 25 1 25

Am Kirchenfeld 15,
94113 Tiefenbach





Vergleich



- Binäre Suche erfordert eine Sortierung der Daten
 - Sortierung erfordert, dass man zwei Datenelemente, bzw. deren Schlüssel, vergleichen kann
- Vereinbarung
 - Daten implementieren Funktion *kleinerGleich*
 - Sortieralgorithmus verwendet zum Vergleich die Funktion *kleinerGleich*
- In Java
 - Schreibe *interface Ordnung*
 - *Ordnung* enthält *boolesche Methode kleinerGleich*
 - Klasse der Datenelemente (z.B. `class Eintrag`) *implements* *Ordnung*
 - *ein* Sortierprogramm funktioniert mit *jeder* Klasse, die *Ordnung* implementiert



Comparable



- *Interface Comparable* im API vorhanden
- Spezifiziert nur eine Methode:
 - `int compareTo(Object o)`
- Viele eingebaute Java-Klassen implementieren dieses Interface, u.a.
 - String, Integer, Float, Character, Date, ...
- Dabei gilt immer:

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

- Jede Implementierung von *Comparable* sollte es analog machen:
 - Viele vorhandene Java Methoden, die auf mit *Comparable* Objekten arbeiten, erwarten ein solches Verhalten.
 - Manchmal beschränkt man sich auf die Werte -1, 0, +1,
 - Manchmal gibt *Comparable* auch noch eine Art Differenz zurück – z.B. bei Strings
- Leider kann man das in Java nicht erzwingen



Beispiel einer Implementierung

```
Eintrag
Class Edit Tools Options
Compile Undo Cut Copy Paste Close Implementation
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 Objekt
Wir casten es zuerst zu einem Eintrag

Hier benutzen wir die compareTo-Methode der Klasse String



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
    }
}
```



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)**:
 - -1 steht für a < b
 - 0 für a == b
 - 1 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;
}
```