Grundlagen der Informatik für Ingenieure I

14 Datenstrukturen

14 Datenstrukturen

- Unter Datenstrukturen versteht man in herkömmlichen Programmiersprachen Datentypen, deren einzelne Elemente primitive Datentypen **verschiedenen** Typs, arrays oder auch wieder Datenstrukturen sein können *(records in "Pascal”, struct in “C”)*.
- Falls die Sprache über *Pointertypen* verfügt, können einzelne Elemente auch vom Typ “*Pointer auf...”* sein, so dass es einfach ist, verkettete Datenstrukturen aufzubauen.
- In Java kennen wir nur die primitiven Typen als “pure” Datenelemente, selbst Zeichenketten und Datenfelder sind schon Objekte. So ist es auch nur möglich, komplexere Datenstrukturen, als verkettete Objekte aufzubauen. Java kennt zwar explizit keine Pointer, aber Referenzen auf Objekte, was nichts anderes ist, als dereferenzierte Pointer.

14.1 *Pointer* (Zeiger)

- Variablen- oder Methodennamen sind symbolische Bezeichnungen von Adressen. Im ersten Fall liegen diese im Adressraum eines Datensegments im zweiten Fall im Adressraum eines Codesegments. Die tatsächliche physikalische Adresse ist erst zum Zeitpunkt des Ladens in den Arbeitsspeicher bekannt (siehe auch Background Kapitel 3 - Rechnerarchitektur und Background Kapitel 6- Speicherverwaltung).
- Der Typ einer Variablen bzw. Konstanten ist entscheidend dafür, in welche Arbeitsregister des Prozessors die Daten geladen werden und welche Teile der Verknüpfungshardware (Functional Units) Operationen auf diese Daten ausführen.
  - Über einen Variablennamen referieren wir eine Adresse mit dem Effekt, dass mit dem Datum, welches in dieser Speicherzelle gespeichert ist, etwas geschieht. (In der Regel wird es zur weiteren Verarbeitung in ein Register geladen.)
    Selektiert wird die Speicherzelle dadurch, dass die Adresse in ein Adressregister des Prozessors geladen wird.

14.2 *Pointer* (Zeiger)

- Wird nun ein Typ *"Pointer"* eingeführt, so ist der Inhalt dieser Speicherzellen als **Adresse** zu interpretieren.
  - Wird der *Pointer* manipuliert, also zur Adressrechnung benutzt, wird er wie ein Integerwert behandelt, wenn auch mit eingeschränkter Arithmetik. (siehe später in diesem Kapitel)
  - Wird der *Pointer* zur Adressierung einer Speicherzelle benutzt, also dereferenziert verwendet, wird er in ein **Adressregister** geladen.
  - Zur Unterscheidung dieser "Modi" sind, z. B. in "C", weitere Operatoren notwendig.

14.2 *Pointer* (Zeiger)

- Wie schon mehrfach erwähnt, ist die Programmiersprache selbst nicht der "Kern" der Vorlesung. Trotzdem wirkt natürlich die Wahl der Beispielsprache auf die Vorlesung prägend.

  Eine wesentliche Eigenschaft von Java ist, dass sie *Pointer* nur als Referenzen auf Objekte kennt; eine direkte Manipulation von *Pointern* ist nicht möglich. Anders ist dies in C oder C++.

  Da aber gerade in der Systemprogrammierung der Umgang mit *Pointern* allgemein üblich ist, sollte auch eine einführende Vorlesung wenigstens ein grobes Verständnis vom Konzept eines *Pointers* vermitteln. Aus diesem Grund habe wir an dieser Stelle ein *"Pointer"* - Abschnitt vorgesehen, obwohl **das Konzept des *Pointers* und OO-Konzepte unvereinbar sind!**

- Basis ist die Sprache *"C"* und fast alle Beispiele beschränken sich auf den Typ *char* (= 1 Byte) in *"C"*!

14.2 *Pointer* (Zeiger)

- Vereinbarung einer *Character-Variablen*, eines *Character-Arrays*:

```
char c;
char stringarray[9];
```

  - *Strings* (Zeichenketten) sind *Characterfolgen (arrays)*, die als letztes Zeichen '\0' enthalten.

h	a	l	l	o	\0			

- Zeiger auf Variable vom Typ *char*

```
char *charpointer ; /* Zeiger auf Zeichenvariable! */
```

  Der *"*"* vor dem Variablennamen drückt aus, dass es sich hier um eine *Pointer*-Variable handelt.

  - Wie üblich, ist mit der Deklaration einer Variablen noch kein Wert zugewiesen worden. Wir müssen der *Pointer*-Variablen also als Wert eine Adresse zuweisen:

```
charpointer = &c;      /* Lies: Adresse von c */
```

  &: Referenzierungsoperator

14.2 *Pointer* (Zeiger)

- Wollen wir nun auf den Inhalt der Speicherzelle zugreifen auf die unser *Pointer* verweist, dann müssen wir den *Pointer* dereferenzieren:

```
char b;
b = *charpointer;
```

ist äquivalent zu:

```
b = c;
```

***: Dereferenzieroperator**

Da diese Art der Verwendung in Java die einzige für einen Pointer ist, brauchen wir in Java diesen Operator nicht. Außerdem sind in Java nur Referenzen auf Objekte und nicht auf primitive Typen möglich!

- Der Name eines *arrays* in *"C"* ist eine *Pointer*-Konstante, so dass es sehr einfach ist, einen *Pointer* auf den Anfang eines Feldes verweisen zu lassen:

```
charpointer = stringarray;
```

ist äquivalent zu:

```
charpointer = &stringarray[0];
```


14.2 *Pointer* (Zeiger)

❑ Auf dem Pointer selbst sind ebenfalls Operationen möglich. Diese beschränken sich auf Adressarithmetik

```
+; ++; - ;--
Beispiel:
c = *strinpointer++; ist äquivalent zu: c = stringarray[i++];
```

und Vergleichsoperationen

```
==; !=
Beispiel:
if ( stringpointer == NULL )
...
```

- Adressarithmetik: *Pointerincrement* ist abhängig vom *Pointertyp*! Zum Beispiel:
  - 1 Byte: *char*
  - 4 Byte: *int*
  - 8 Byte: *double*
  - Länge eines Feldes, wenn *Pointer* vom Typ "Zeiger auf ein Feld" ist.

14.2 *Pointer* (Zeiger)

- *Strings* (Zeichenketten) sind nur zeichenweise manipulierbar. Es gibt in C keine primitiven Operationen auf Zeichenketten, wohl aber eine Reihe von *Library*-Funktionen: Siehe *man string(3C)*!
- *Stringliterale* z. B. **"hello"** werden automatisch (vom Compiler) mit einem '\0' abgeschlossen.

```
char amessage[] = "now is the time"; /* array of char */
char *pmessage = "now is the time"; /* Pointer auf array
                                       (1. Element) */
```

- *Stringcopy* in *Pointer*- und *Array*-Version:

```
while ((*sp++ = *tp++) != '\0');
```

ist äquivalent zu:

```
while ((sarray[i] = tarray[i]) != '\0')
    i++;
```


14.2 *Pointer* (Zeiger)

- *Arrays* von *Pointern (Command-line Arguments)*

```
char *strings[10];
```

- *Pointer* auf *Pointer*

```
char **pp = strings;
```

- Pointer auf *array*

```
char (*parray)[10];
```

Inkrement: 10 Byte!

14.2 *Pointer* (Zeiger)

◆ Beispiel:

```
/* testpointer1.c */
main(int argc, char *argv[]) {
    int i;
    char string[] = "Hallo!\n";
    char *strings[] = {"Alle ","mein ","Entchen ","schwimmen "
                       ", "auf ","dem ","See.\n", NULL};
    char **p_on_strings = strings;
/* printf() erwartet Pointer als Parameter beim Typ "%s" */
    printf("%s", string);
    for (i = 0;  strings[i] != NULL; i++)
        printf("%s", strings[i]);

    while (*p_on_strings != NULL){
        printf("%s", *p_on_strings);
        p_on_strings++;
    }
}
```

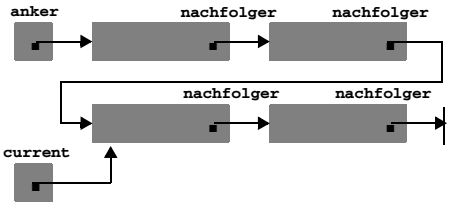

14.2 *Pointer* (Zeiger)

◆ Aufgabe: Bitte versuchen Sie sich mit einer Datenstrukturskizze klarzumachen was passiert (Lösung nächste Vorlesung):

```
/* testpointer2.c */
/* keine Programmier-Stil-Empfehlung! */
main()
{   int i;
    char *c[] = { "ENTER","NEW","POINT","FIRST"};
    char **cp[4];
    char ***cpp = cp;
    for (i=0; i< 4; i++)
        cp[i] = c + 3 -i;

    printf("%s", **++cpp);          /* v. rechts n. links*/
    printf("%s", *--*++cpp + 3);
    printf("%s", *cpp[-2]+3 );      /* [] staerkste Bindung */
    printf("%s\n", cpp[-1][-1]+1);  /* v. links n. rechts */
}
```

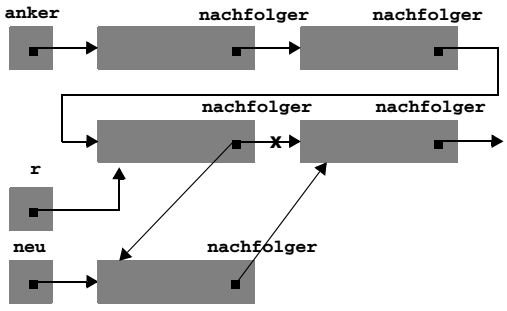
14.3 Verkettete Listen

- Bei vielen Problemfällen erweisen sich Felder als eine zu starre Datenstruktur mit u. a. folgenden Nachteilen:
  - Speicherplatz für alle Elemente muss vorab angefordert werden. Erweiterungen sind dynamisch nur mit erheblichem Aufwand (*Instantiieren*, Kopieren in Java; Speicheranforderung, Kopieren in C) möglich.
  - Einschieben von Elementen (etwa zur Erhaltung einer Sortierordnung) ist sehr aufwendig: Kopieren aller nachfolgenden Elemente.
  - Ungünstig bei Datensammlungen, die im Laufe der Zeit sehr stark wachsen und/ oder schrumpfen
- Allgemein: Wenn der Direktzugriff auf ein Element über einen Index weniger wichtig ist, dann sollte man überlegen, statt eines Feldes eine verkettete Liste zu verwenden.
  Einfache verkettete Liste:



14.3.1 Einfügen

- Einfügen
  - Ein neues Element kann an jeder beliebigen Stelle der Liste eingefügt werden.
    Dazu muss eine Referenz r auf den Vorgänger bereitgestellt werden.



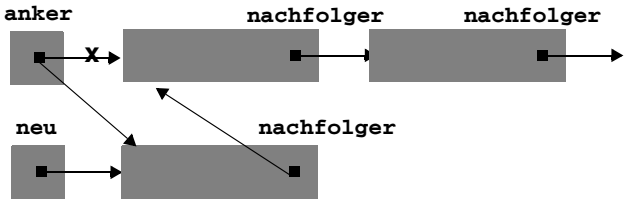

```
// Einfuegen hinter r
    neu.nachfolger = r.nachfolger
    r.nachfolder = neu
```

14.3.1 Einfügen

❏ Einfügen (cont)

Für den Sonderfall, dass das neue Listenelement ganz an den Anfang gehängt werden soll:

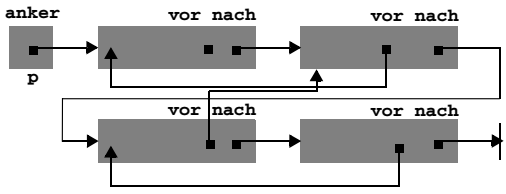
```
neu.nachfolger = anker
anker = neu
```



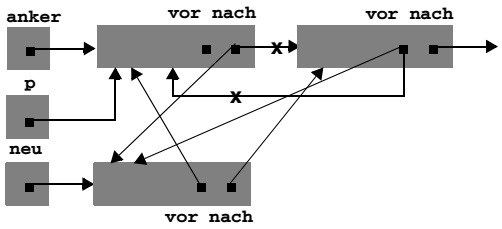
14.3.2 Doppelt verkettete Liste

❏ Doppelt verkettete Liste

- Bei der einfach verketteten Liste kommt man von einem Element nur sehr umständlich zu seinem Vorgänger.
- Idee: auch rückwärts verketten



❏ Einfügen
Wichtig ist die Reihenfolge der Einzeloperationen. Es ist darauf zu achten, dass nicht noch benötigte Verkettungsinformation überschrieben wird.



14.3.3 Realisierung in C

- Primitive Datentypen verschiedenen Typs lassen sich in “C” in sogenannte *structures* zusammenfassen:
  Definition:

```
struct node{
    char *name;
    char *vorname;
    int alter;
    struct node *next;
};
```

  Deklaration:

```
struct node a, b, c;
struct node *anchor;
```

  Zugriff:

```
a.name = "Meier";
meieralter = a.alter;
anchor = &a;
a.next = &b;
```


14.3.3 Realisierung in C

- Zugriff über *Pointer:*

```
(*anchor).vorname = "Erich";
```

  ist äquivalent zu:

```
anchor->vorname = "Erich";
```

  ist äquivalent zu:

```
a.vorname = "Erich";
```

  und

```
a.next->name = "Mueller";
```

  ist äquivalent zu:

```
b.name = "Mueller";
```

- Dynamische (zur Laufzeit) Erzeugung eines *"node"*:

```
...
if (b.next = (struct node *)malloc(sizeof(struct node)) == NULL)
    printf ("malloc: no more space available!";
...
```

  Dynamisch angelegte Struktur ist nur über *Pointer* zugänglich!

  "Verfahren" ist äquivalent dem in Java ablaufenden Vorgang bei der Instantiierung eines Objekts.

14.3.4 Realisierung in Java

- Datenstrukturen sind Objekte
- Datenelemente werden durch Instanzvariablen repräsentiert

14.3.4 Realisierung in Java am Beispiel Warteschlange

- Die Eigenschaft einer Warteschlange ist, dass man
  - Elemente an das Ende der Schlange anfügt und
  - Elemente von Beginn der Schlange entfernt.
- verwendet werden zwei “Anker”:
  - einer mit dem das 1. Element referenziert werden kann und
  - einer mit dem das letzte Element referenziert werden kann.
- Als Element implementieren wir einen universellen Knoten *(Node)*, mit dem man sowohl einfach verkettete Listen, als auch doppeltverkettete Listen aufbauen kann:

```
class Node{
   private String information;
   private Node precElement;
   private Node succElement;

// Konstruktor für einfach verkettete Listen

   public Node (String info, Node succ {
      information = info;
      succElement = succ;
   }
```

14.3.4 Realisierung in Java am Beispiel Warteschlange

```
  // Konstruktor fuer doppelt verkettete Listen

  public Node (String info, Node prec, Node succ) {
    information = info;
    precElement = prec;
    succElement = succ;
  }
// Access-Methods

  public String getInformation(){
    return information;
  }
  public Node getPrecessor(){
    return precElement;
  }
  public Node getSuccessor(){
    return succElement;
  }
  public void setPrecessor(Node element){
    this.precElement = element;
  }
  public void setSuccessor(Node element){
    this.succElement = element;
  }
}
```


14.3.4 Realisierung in Java am Beispiel Warteschlange

```
class Queue{

  private Node firstElement;
  private Node lastElement;

  public Queue() {
    firstElement = new Node("first", null);
    lastElement = new Node("last", null);
  }

  public void addElement(String info) {
    currentElement = new Node(info, null);
    if (lastElement.getSuccessor() == null){
      // newNode is first element in queue
      firstElement.setSuccessor(currentElement);
      lastElement.setSuccessor(currentElement);
    } else {
      // newNode is another node
      lastElement.getSuccessor().setSuccessor(currentElement);
      lastElement.setSuccessor(currentElement);
    }
  }
```

14.3.4 Realisierung in Java am Beispiel Warteschlange

```
  public String getNodeInfo(Node theNode){
    System.out.println("getNodeInfo(): not yet implemented");
    return "getNodeInfo(): not yet implemented";
  }

  public String getInfos() {

    Node currentElement;
    String information ="";

    currentElement = firstElement;
    while ((currentElement = currentElement.getSuccessor()) !=
                                                          null){
      information += currentElement.getInformation();
    }
    return information;
  }

  public Node removeElement() {
    System.out.println("removeElement(): not yet implemented");
    return null;
  }
}
```


14.3.4 Realisierung in Java am Beispiel Warteschlange

- Das Testprogramm

```
class QueueTest {

    public static void main(String args []) {

      String wsinfos;

      Queue ws = new Queue();

      ws.addElement("This ");
      ws.addElement("is ");
      ws.addElement("the ");
      ws.addElement("way ");
      ws.addElement("building ");
      ws.addElement("queues! ");
      wsinfos = ws.getInfos();
      System.out.println (wsinfos);
    }
}
```

14.3.4 Realisierung in Java am Beispiel Warteschlange

- Das vorgehende Beispiel zeigt, dass die Verwendung von “Access-Methods” eine sehr umständliche Angelegenheit sein kann.
  Sinnvoller (und kompakter) ist für die Knoten die Definition einer Hilfsklasse und der direkte Zugriff auf die Instanzvariablen der Objekte dieser Klasse:

```
//-------------------------------------------------------
// Hilfsklasse
class Node {
  protected Object information;
  protected Node succElement;

  protected Node ( Object info ) {
    information = info;
  }

  Object getInformation(){
    return information ;
  }

}
//-------------------------------------------------------
```


14.3.4 Realisierung in Java am Beispiel Warteschlange

```
public class Queue{

  private Node firstElement;
  private Node lastElement;

  public void addElement( String info ) {

    Node currentElement;
    currentElement = new Node( info );

    if ( firstElement == null ){
      // newNode is first element in queue

      firstElement = currentElement;
      lastElement = currentElement;
    } else {
      // newNode is another node

      lastElement.succElement = currentElement;
      lastElement = currentElement;
    }
  }
```

14.3.4 Realisierung in Java am Beispiel Warteschlange

```
  public String getInfos() {

    Node currentElement;
    String information ="";
    currentElement = firstElement;
    while ( currentElement != null ){
      information += currentElement.getInformation();
      currentElement = currentElement.succElement;
    }

    return information;
  }

}
```


14.3.5 Linked Lists in Java

- Java enthält im Package java.util u. a. auch eine Klasse LinkedList. Hierzu ein Beispiel:

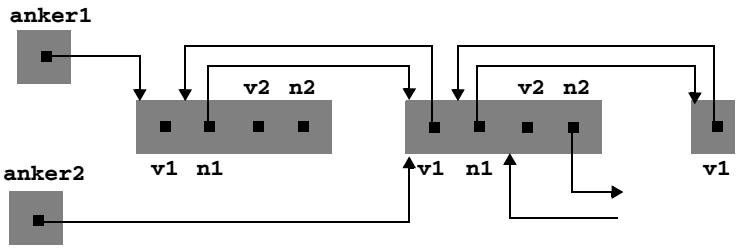
```
*// Author Cay Horstmann (Core-Java)

import java.util.*;

public class LinkedListTest {

  public static void main(String[] args) {

    List a = new LinkedList();
    a.add("Angela");
    a.add("Carl");
    a.add("Erica");

    List b = new LinkedList();
    b.add("Bob");
    b.add("Doug");
    b.add("Frances");
    b.add("Gloria");
```

14.3.6 Linked Lists in Java

```
// Einträge von b nach a mischen
     ListIterator aIter = a.listIterator();
     Iterator bIter = b.iterator();

     while (bIter.hasNext()) {
       if (aIter.hasNext()) aIter.next();
         aIter.add(bIter.next());
     }
     System.out.println(a);

      // Jeden zweiten Eintrag aus b entfernen

     bIter = b.iterator();
     while (bIter.hasNext()) {
       bIter.next(); // Ein Element überspringen
       if (bIter.hasNext()) {
         bIter.next(); // Nächstes Element überspringen
         bIter.remove(); // Dieses Element entfernen
       }
     }
     System.out.println(b);

   // Massenoperation: Alle Elemente in b aus a entfernen

     a.removeAll(b);
     System.out.println(a);
   }
}
```


14.4 Multi-Listen

- Multi-Listen (auch mehrdimensionale Listen genannt)
  - Eine Menge von Elementen gleichzeitig nach *mehreren Kriterien* organisiert; jede Organisation durch eine Verkettung dargestellt.
  - Beispiel: Liste: GDI-Studierende; Teillisten: MB, WW, ...”
  - Organisation nach zwei Kriterien:

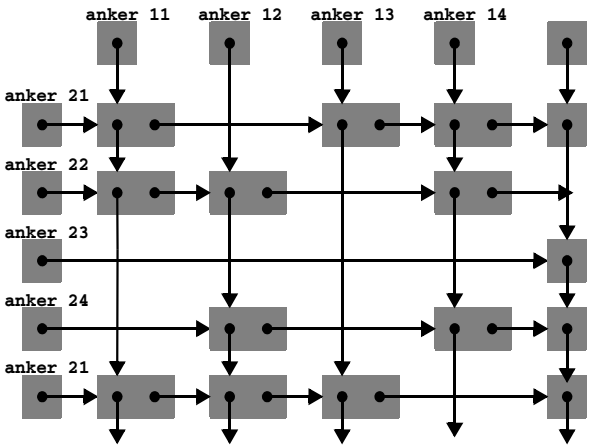


14.4 Multi-Listen

- Jede Liste kann für sich getrennt verwaltet werden.
- Ganz allgemein stellt eine Liste eine (beliebige) Teilmenge der Menge aller Elemente dar.
- Die durch die Listen realisierten Teilmengen können (zeitweise) sogar disjunkt sein.
- Aushängen aus einer Liste ist zu trennen vom Löschen!
- Gelöscht werden dürfen nur Elemente, die in keiner der beiden Listen mehr enthalten sind.

14.4.1 Spezialfall: Dünn besetzte Matrizen

- Mehrere Anker pro Dimension:
- Einen für jede Spalte in der einen und einen für jede Zeile in der anderen (jeweils disjunkte Listen)



(nur einfache Verkettung dargestellt)

14.5 Suchen und Sortieren

❑ Suchen und Sortieren sind die häufigsten Verfahren, die auf strukturierte Daten angewendet werden. Sie sind schon seit Anfang der Informatik ausführlich untersucht worden, so dass es sich kaum lohnt eigene Algorithmen “zu erfinden”.
Der Klassiker der Literatur ist **Knuth:** *“The Art of Programming”.*

14.5.1 Suchen

◆ Sehr häufiges Problem:
“Finde Element mit bestimmten Schlüsselwert”

◆ einfachste Lösung, Programmfragment:

```
....
String[] persons;
String search;

.......
search = "Vergleichswert";
while (int i < persons.length && persons[i] != search) {
    i++;
}
.....
```

14.5.1 Suchen

- Einführung eines “Wächters”; englisch ’sentinel’ oder ’stopper’
  - dem Feld wird ein weiteres Element angefügt, welches mit dem Suchbegriff vor Beginn der Suche vorbesetzt wird. Dadurch wird die bei jedem Durchlauf auszuwertende Abbruchbedingung vereinfacht.

```
....
String[] persons;
String search;

.......
search = "Vergleichswert";
persons[persons.length-1] = search;
while (persons[i] != search) {
    i++;
}
.....
```


14.5.2 Suche in sortierten Feldern

Die Suche kann erheblich schneller durchgeführt werden, wenn das Feld nach dem Schlüsselwort sortiert ist. (Siehe auch Abschnitt Sortieren)

- Binäre Suche
  - Springe in die Mitte des Feldes
  - Wenn Vergleichswert dort gefunden: fertig
  - Wenn Vergleichswert kleiner als dortiger Schlüsselwert nur noch in der ersten Hälfte Suchen, andernfalls in der 2. Hälfte
  - mit gleicher Technik wieder in die Mitte des verbleibenden Feldes springen, usf.
  - Rekursive Definition; logarithmischer Aufwand anstatt des linearen

14.5.2 Suchen in sortierten Feldern

- Beispiel:

```
public class MySearch {

   final boolean DEBUG = true;

   public void binSearch(int intArray[], int key) {

       int i = 0, left = 0, right = 0, center = 0;

        if(DEBUG) {
          System.out.println("Das Eingangsarray: ");
          printArray(intArray);
        }
        right = intArray.length - 1;
        center = (right + left) / 2;

        while ((left + 1 < right) && (intArray[center] != key)) {
          if(DEBUG)
            System.out.println("mitte: "+ center);
          if (intArray[center] < key)
            left = center;
          else
            right = center;
          center =  (right + left) / 2;
          i++; //Statistik
        }
```


14.5.2 Suchen in sortierten Feldern

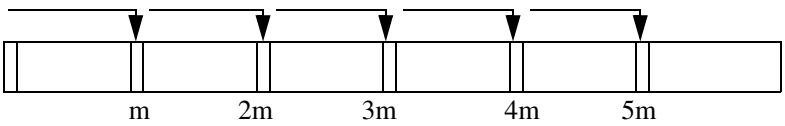
```
        if ((left < right) && (intArray[center] == key))
          System.out.println("key: " +key+" gefunden"+
                             " am Platz: "+center+" nach "+i+" Iterationen!");
        else
          if ((left < right) && (intArray[center + 1] == key))
            System.out.println("key: " +key+" gefunden"+
                             " am Platz: "+center+" nach "+i+" Iterationen!");
        else

          System.out.println("key: " +key+" nicht gefunden");
   }
    private void printArray(int intArray[]) {

        for (int i = 0; i< intArray.length; i++) {
          System.out.print(intArray[i] + " ");
        }
        System.out.println();
   }
}
```

14.5.2 Suchen in sortierten Feldern

- Fibonacci-Suche
  - ähnlich wie binäre Suche, jedoch keine Division sondern nur Addition und Subtraktion
  - Aufteilung der Suchbereiche nach den Fibonacci-Zahlen
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_k = F_{k-1} + F_{k-2}$ für $k > 1$

14.5.2 Suchen - in sortierten Feldern

- Sprungsuche
  - Aufteilung eines Feldes in Abschnitte der Länge m
  - Überprüfung der Elemente m, 2m, 3m, usw. zur Ermittlung des Abschnitts, in dem der gesuchte Schlüssel liegen muss.
  - Dann sequentielle Suche in dem Abschnitt (oder binäre Suche oder auch weiter Sprungsuche)



  - Aufwand etwa Wurzel aus N, d. h. schlechter als binäres Suchen, aber einfacher zu realisieren.
    Ist besonders dann eine sinnvolle Vorgehensweise, wenn der gesamte Suchraum nur abschnittsweise zur Verfügung steht; z. B. beim blockweises Lesen von einem Peripheriegerät.

14.5.3 Sortieren

- eine außerordentlich häufig benötigte Funktion
  - z. B. für bessere Lesbarkeit von Listen oder
  - Vorbereitung für Suchverfahren
- Aufwand zunächst N^2
  - Häufig notwendige Hilfsfunktion ist der Tausch zweier Elemente:

```
temp = array[i];
array[i] = array[j]
array[j] = temp;
```


14.5.3.1 Selection Sort (Sortieren durch Auswählen)

- Ermittle kleinstes Element und setze es an den Anfang
- Wiederhole dies für den Rest des Feldes

❑ Beispiel in FORTRAN95 - Code

```
SUBROUTINE ssort (feld, max, debug)
! Sortiert 'feld' nach der Methode 'Selection Sort'
! Formalparameter
INTEGER, INTENT (IN) :: max
LOGICAL, INTENT (IN) ::debug
INTEGER , DIMENSION (max) :: feld

! Lokale Variable:
INTEGER i, j, min

  IF (debug) PRINT *, feld
  DO i = 1, max
    min = i
    DO j = i+1, max
      IF (feld(j) < feld(min)) min = j
    END DO
    IF (i /= min) CALL tausche(feld,max,i,min)
  END DO
  IF (debug) PRINT *, feld
END SUBROUTINE ssort
```

14.5.3.2 Sortieren durch Einfügen

- Betrachte die N Elemente der Reihe nach
- Füge jedes an der richtigen Stelle in die links von ihm stehende Folge ein - durch Verschieben

❏ Beispiel in FORTRAN95 - Code

```
SUBROUTINE esort (feld, max, debug)
! Sortiert 'feld' nach der Methode 'Sortieren durch Einfuegen'
! Formalparameter:
INTEGER, INTENT (IN) :: max
LOGICAL, INTENT (IN) :: debug
INTEGER , DIMENSION (max) :: feld
! Lokale Variable:
INTEGER i, free, temp

  IF (debug) PRINT *, feld
  DO i = 2, max
    free = i
    temp = feld(free)
    DO WHILE ((free > 1) .AND. (feld(free-1) > temp))
      feld(free) = feld(free-1)
      free = free-1
    END DO
    feld(free) = temp
  END DO
  IF (debug) PRINT *, feld
END SUBROUTINE esort
```


14.5.3.3 Sortieren durch Vertauschen (Bubblesort)

- Feld durchlaufen, dabei benachbarte Elemente vertauschen, wenn sie nicht in Sortierreihenfolge stehen
- Wiederholen, bis keine Vertauschung mehr erfolgt ist
- große Werte steigen wir “Blasen” an das Ende des Feldes auf.

❏ Beispiel in Java - Code

```
public void bubbleSort(int intArray[]) {
  boolean change;
  int icount = 0, excount = 0;
  int temp;

  if(DEBUG) {
    System.out.println("Das Eingangsarray: ");
    printArray(intArray);
  }
```

14.5.3.3 Sortieren durch Vertauschen (Bubblesort)

```
    loop:for (int  i = intArray.length - 1; i > 0; i--) {
           icount++;
           change = false;
           for (int  j = 0; j < i; j++) {
             if (intArray[j] > intArray[j + 1]) {
               temp = intArray[j];
               intArray[j] = intArray[j + 1];
               intArray[j+1] = temp;
               change = true; // change happens
               excount++;
             }
           }
           if (!change) break loop; // No change in last run
         }
         if(DEBUG) {
           System.out.println("Das Ausgangsarray: ");
             printArray(intArray);
         }
         System.out.println("Vertauschungen: "+ excount);
         System.out.println("Durchlaeufe: "+ icount);
     }
      private void printArray(int intArray[]) {
           for (int i = 0; i< intArray.length; i++) {
                    System.out.print(intArray[i] + " ");
                }
      System.out.println();
      }
}
```


14.5.4 Sort & Search - Arbeitsumgebung

- Klassen: **MySearch; MySort; TestSortAndSearch**
- Zur Zeit implementiert: **MySearch.binSearch(...)**
  MySort.bubbleSort(...)
- **TestSortAndSearch:**

```
import java.io.*;

public class TestSortAndSearch {

     public static void main(String args[])throws java.io.IOException {
       int length, key;
       int iterationen;
       String input;
       BufferedReader inStream;

         inStream = new BufferedReader(new InputStreamReader(System.in));
         System.out.println("Eingabe: Länge des Feldes");
         length = Integer.parseInt(inStream.readLine());
         int array[] = new int[length];

         // Aufbau eines Zufallszahlen Integer-Arrays:

         for (int i = 0; i < array.length; i++) {
           array[i] = (int)(Math.random() * array.length);
         }
```

5 Sort & Search - Arbeitsumgebung

```
        // Test von MySort.bubbleSort(...):

          MySort sort = new MySort();
          sort.bubbleSort(array);

        // Test von MySearch.binSearch(...):

          System.out.println("Welcher Schlüssel soll gesucht werden?");
          key = Integer.parseInt(inStream.readLine());

          MySearch search = new MySearch();
          search.binSearch(array, key);
    }
}
```


5 Sort & Search - Arbeitsumgebung

```
> java TestSortAndSearch
Eingabe: Länge des Feldes
100
Das Eingangsarray:
5 73 10 66 74 99 81 58 97 42 50 79 71 18 14 96 34 94 86 29 84 51 29 81 94 68 38
29 19 22 82 95 71 10 35 96 33 12 87 34 11 91 53 90 91 96 45 56 14 40 19 47 60 92
67 87 82 65 66 41 15 87 68 25 26 73 55 24 0 66 75 9 57 88 6 27 65 70 95 51 16 39
87 16 67 47 6 30 38 69 73 45 13 76 94 9 39 78 55 46
Das Ausgangsarray:
0 5 6 6 9 9 10 10 11 12 13 14 14 15 16 16 18 19 19 22 24 25 26 27 29 29 29 30 33
34 34 35 38 38 39 39 40 41 42 45 45 46 47 47 50 51 51 53 55 55 56 57 58 60 65 65
66 66 66 67 67 68 68 69 70 71 71 73 73 73 74 75 76 78 79 81 81 82 82 84 86 87 87
87 87 88 90 91 91 92 94 94 94 95 95 96 96 96 97 99
Vertauschungen: 2660
Durchlaeufe: 91
Welcher Schlüssel soll gesucht werden?
87
Das Eingangsarray:
0 5 6 6 9 9 10 10 11 12 13 14 14 15 16 16 18 19 19 22 24 25 26 27 29 29 29 30 33
34 34 35 38 38 39 39 40 41 42 45 45 46 47 47 50 51 51 53 55 55 56 57 58 60 65 65
66 66 66 67 67 68 68 69 70 71 71 73 73 73 74 75 76 78 79 81 81 82 82 84 86 87 87
87 87 88 90 91 91 92 94 94 94 95 95 96 96 96 97 99
mitte: 49 links: 0 rechts: 99
mitte: 74 links: 49 rechts: 99
mitte: 86 links: 74 rechts: 99
mitte: 80 links: 74 rechts: 86
key: 87 gefunden am Platz: 83 nach 4 Iterationen!
```

14.5.6 Häufigkeitsgeordnete Listen

- ❏ Häugigkeitsgeordnete Listen
  - Sortierung *allein* zur Beschleunigung der Suche lohnt sich bei Listen kaum.
  - Wenn es keine anderen Gründe für die Erhaltung einer Sortierordnung gibt, kann die Suche in vielen Fällen dadurch beschleunigt werden, dass man die häufig benötigten Elemente an den Anfang der Liste stellt bzw. die Liste absteigend nach Zugriffshäufigkeit sortiert.
  - In der Praxis oft zu beobachten:

    80/20-Regel:

    - 80% der Zugriffe erfolgen auf 20% der Daten
    - von diesen 80% betreffen wiederum 80% (64% der gesamten Zugriffe) 20% von den ersten 20% (4% aller Daten) usw.

    Wenn das der Fall ist und die Liste nach Häufigkeit sortiert wurde, reduziert sich der Aufwand für die Suche auf ca. 0,122 N (statt N/2).

14.5.6 Häufigkeitsgeordnete Listen

- ❏ Häugigkeitsgeordnete Listen
  - Erforderliche Maßnahmen:
    - Zugriffszähler in jedem Element führen (“Frequency Count”) und von Zeit zu Zeit neu sortieren, oder
- ❏ selbstorganisierende Liste
  - Jedes gesuchte Element an den Anfang der Liste setzen (“Move to Front”)
    - Vorteil: kein Häufigkeitszähler zu verwalten
    - Nachteil: sehr oft Umhängen (als Seiteneffekt der Suche!)

14.6 Gestreute Speicherung (Hashing)

- Hash-Funktion
  Suche bisher im besten Fall - bei vorliegender Sortierung - mit logarithmischen Aufwand. Kann man das noch besser machen?

  Idee:
  Wenn man aus dem Vergleichswert (Schlüssel) den Speicherort **(einfach)** berechnen könnte, braucht man nur genau einen Zugriff unabhängig von der Größe der Datenbasis.

- Hash-Funktion:

```
Sei S die Menge aller möglichen Schlüsselwerte und
A = {0,1,.....m-1} das Interval der Zahlen von 0 bis m-1
dann ist h: S --> A eine Hash-Funktion
```

- HASH-Tabelle:
  Damit das Verfahren funktioniert, muss die Datenbasis geeignet aufbereitet werden, d. h. die Elemente der Datenbasis werden mit der Hashfunktion (Ergebnis = ganze Zahl) abgespeichert; es wird eine Hash-Tabelle aufgebaut.

- Es entsteht keine Ordnung der Einträge!

14.6 Gestreute Speicherung (Hashing)

- Kollisionen
  Von Kollisionen spricht man, wenn es mehrere Schlüssel gibt, die nach Anwendung der Hashfunktion zum gleichen Speicherort führen.

  Die betreffenden Schlüsselwerte nennt man Synonyme.
  Bei Kollisionen ist eine Sonderbehandlung erforderlich, die dazu führt, dass eines der beiden Elemente nicht mehr in einem einzigen Zugriff errechnet werden kann.

- Für eine Hash-Funktion soll gelten:
  - einfach und effizient zu berechnen
  - erzeugt gleichmäßige Belegung des Feldes
  - verursacht möglichst wenig Kollisionen

- Ihre Leistungsfähigkeit hängt ab von
  - dem Belegungsgrad des Feldes
  - der Methode zur Kollisionsauflösung
  - der Verteilung der aktuell vorkommenden Schlüsselwerte

14.6 Gestreute Speicherung (Hashing)

- Divisionsrest-Methode
  - interpretiere Binär- oder Dezimaldarstellung des Schlüssels s als ganze Zahl;
  - dividiere durch m (Länge der Hashtabelle)
  - verwende den Rest der Division als Index (Adresse)
  - Einfach bei Zahlen:

```
h = MOD(s, m)
```

  - Welche Eigenschaften sollten für m gelten:
    - größer als die Zahl der zu speichernden Elemente
    - ungerade; besser:
    - Primzahl

14.6 Gestreute Speicherung (Hashing)

- Beispiel für Zeichenketten:

```
int hashChar(String inputString, int buckets) {

    int sum = 0;
    char word[] = new char[100];
    inputString.getChars(0, inputString.length() -1, word,0);

    for (int i = 0; i < inputString.length(); i++) {
      Character c = new Character(word[i]);
      sum = sum + c.charValue();
    }
    return sum%buckets;
  }
// effizientere Implementierungen sind denkbar!!
```

(In der Klasse String steht natürliche eine Methode hashCode() zur Verfügung.)

14.6 Gestreute Speicherung (Hashing)

- Mid-Square-Methode
  - Schlüsselwort s wird quadriert
  - t Ziffern aus der Mitte des Ergebnisses werden als Index (Adresse) verwendet.
  - Wenn b die Basis des verwendeten Zahlensystems ist, muss also $m = b^t$ gelten.
  - Verfahren ist relativ gut, solange Schlüsselwerte wenig führende oder nachfolgende Nullen besitzen und sich die mittleren Stellen genügend ändern.

14.6 Gestreute Speicherung (Hashing)

- Faltung
  - Zerlegung des Schlüsselwerts in Teile, die bis auf das letzte die Länge einer Adresse besitzen.
  - “Faltung” und Addition (oder auch EXOR) der Teile
  - ggf. höchstwertige Ziffer abschneiden
  - Faltung entweder wie Papier vom Rand her oder durch Übereinanderschieben
  - Beispiel:

```
b = 10, t = 3, m = 1000; s = 12345678

Randfalten:          Schichtung:
          321                  123
           87                  456
          456                   78
          864                  657
```

14.6 Gestreute Speicherung (Hashing)

- Hash-Funktionen: Zusammenfassung
  - Es gibt noch viele andere (s. ggf. Literatur)
  - Verhalten hängt von der vorliegenden Schlüsselmenge ab; daher mit analytischen Modellen nur unzureichend zu erfassen.
  - Empirische Untersuchungen haben gezeigt:
    - Die Divisionsrest-Methode ist im Mittel das leistungsfähigste Verfahren; für bestimmte Schlüsselmengen können jedoch andere Techniken besser geeignet sein.
    - Keine Hash-Funktion ist **immer** besser als die anderen
    - Wenn über die Schlüsselverteilung nichts bekannt ist, fährt man mit der Divisionsrest-Methode am besten.

14.6 Gestreute Speicherung (Hashing)

- Kollisionsbehandlung
  - Hash-Funktion ermittelt für neu abzuspeichernden Schlüsselwert s den Index i, unter dem bereits ein Schlüsselwert gespeichert ist.
    Es muss also erkennbar sein, ob eine Position belegt ist oder nicht, z. B. getrennt geführte Bitliste; Belegtindikator im Feld, wenn Elemente sowieso Strukturen, ...
  - Den neu zu speichernden Schlüsselwert bezeichnet man als Überläufer. Zwei Lösungsmöglichkeiten:
    - Es wird für s ein anderer Platz im Feld verwendet, der noch frei ist. Diese Methode wird als “open addressing” bezeichnet und ist dann gut geeignet, wenn die Belegung des Feldes nicht in vornherein ’dicht’ ist.
    - s wird in einem separaten Überlaufbereich zusammen mit weiteren Überläufern verwaltet. Diese Methode wird als “separate overflow” bezeichnet.

14.6 Gestreute Speicherung (Hashing)

- ❏ Kollisionsbehandlung (cont)
  - Beides muss nach einer strengen Systematik erfolgen, damit der Schlüsselwert auch wiedergefunden werden kann.
  - Es wird also einen Folge von Hash-Funktionen h_i definiert, wobei h_{i+1} angewendet wird, wenn h_i auf einen besetzten Platz geführt hat.

14.6 Gestreute Speicherung (Hashing)

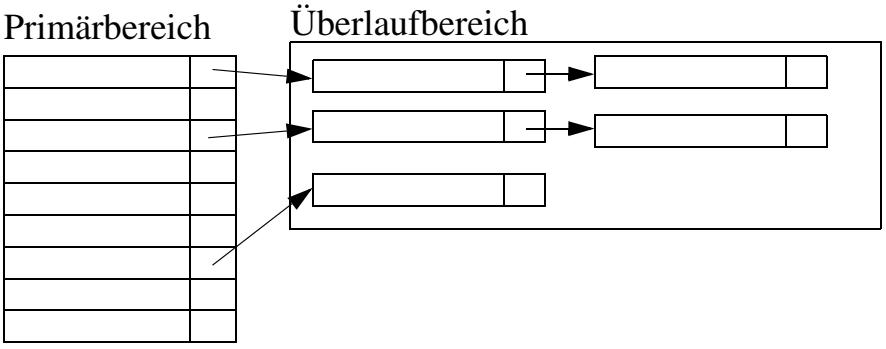
- ❏ Lineares Sondieren (linear probing)

$$h_0(s) = h(s)$$
$$h_i(s) = (h_0(s) + i) \bmod m$$
$$= (h_{i-1}(s) + 1) \bmod m$$

  - Von der sogenannten Hausadresse aus wird sequentiell (und ggf. zyklisch) nach einer freien Adresse gesucht.
  - dabei wird jeder freie Platz irgendwann gefunden; das Verfahren scheitert erst, wenn das Feld voll ist.
  - Suche und Einfügen sind einfach zu realisieren
- ❏ Wie wird ein Überläufer wieder aus dem System gelöscht?
  - Überläufer, die zuvor über die freiwerdende Position hinweg verschoben wurden, müssen “zurückgeholt” werden.
- ❏ Weitere Vertiefung: s. Literatur

14.6 Gestreute Speicherung (Hashing)

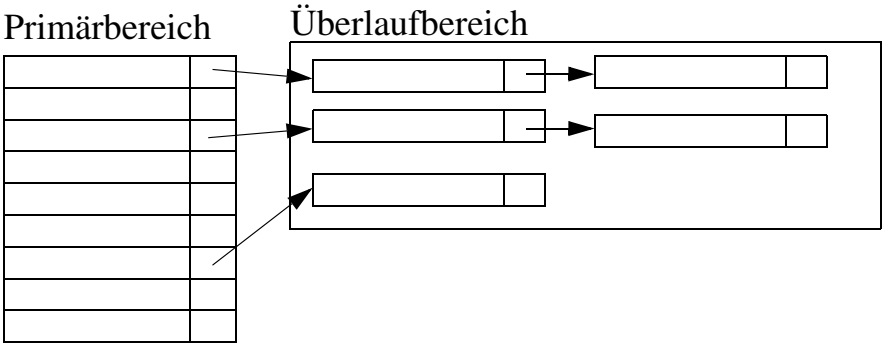
- Separater Überlaufbereich
  - zusätzlicher Speicherbereich
    - dort wieder Hashing oder
    - z. B. Verkettung der Synonyme



  - Suchen, Einfügen, Löschen relativ einfach zu Implementieren.

14.7 Gestreute Speicherung (Hashing) in Java

- Hashtabelle
  - Hashtabellen sind als Arrays verketteter Listen implementiert. Die einzelnen Listen werden als *buckets* bezeichnet.



- Hashfunktion(= Finden des Speicherorts)
  - Berechnung des Hashwerts (siehe später) dividiert modulo durch die Gesamtzahl der *buckets*.

14.7 Gestreute Speicherung (Hashing) in Java

- Kollisionen
  - Im Idealfall treten keine Kollisionen auf. Im Falle das Kollisionen auftreten, muss bei Set-Mengen (= Mengen ohne Duplikate) untersucht werden, ob das neueinzufügenen Objekt bereits in der Menge der Objekte dieses *buckets* vorhanden ist.
  - Treten Kollisionen zu häufig auf, muss an eine Vergrößerung der *bucket*-Anzahl gedacht werden um das System effizient zu halten.
    - Empfehlung:
      Wenn man weiss, wie viele Elemente maximal in einer Hashtabelle gehalten werden sollen, so sollte man die Länge der Tabelle etwa um 150% größer ansetzen und dafür eine geeignete Primzahl wählen.
      (also z. B. 500 Elemente --> 150% --> 750; "nächste gößere" Primzahl: 751)

14.7 Gestreute Speicherung (Hashing) in Java

14.7 Gestreute Speicherung (Hashing) in Java

- Kollisionen
  - Ist die Anzahl der zu speichernden Objekte im voraus nicht bekannt, kann Java mittels eines Lastfaktors die Restrukturierung der Hashtabelle anstoßen. Der Lastfaktor ist ein Maß für den Füllgrad einer Hashtabelle.
  - Gibt man keinen Lastfaktor an, so ist der (default) Lastfaktor 0,75, d.h. bei einem Füllgrad von 75 % wird die Länge der Tabelle verdoppelt.

14.7 Gestreute Speicherung (Hashing) in Java

- Eine Standardanwendung von Hashtabellen ist die Verwaltung von Set-Mengen.
- Klasse *HashSet*
  - Konstruktoren

Konstruktoren	Beschreibung
HashSet()	101 Elemente, Lastfaktor 0.75
HashSet(int initialCapacity)	initialCapacity Elemente, Lastfaktor 0.75
HashSet(int initialCapacity, float loadfactor)	initialCapacity Elemente, Lastfaktor loadfactor

14.7 Gestreute Speicherung (Hashing) in Java

- Methoden

Methoden	Beschreibung
boolean add(Object o)	Fügt das Element in den HashSet ein, falls noch nicht vorhanden, sonst: false
void clear()	Löscht sämtliche Element aus dem HashSet
boolean contains (Object o)	Gibt "true' zurück, falls Object Element vom HashSet
boolean isEmpty()	Gibt "true' zurück, falls HashSet leer
Iterator iterator()	Gibt den Iterator des Sets zurück
boolean remove	Löscht das Element in den HashSet , falls vorhanden, sonst: false
int size	

- Methoden des Interfaces Iterator

Methoden	Beschreibung
boolean hasNext()	Gibt "true" zurück, falls weitere Elemente vorhanden sind
Object next()	Gibt das nächste Element zurück

14.7 Gestreute Speicherung (Hashing) in Java

- Hashfunktionen
  - Wie wird nun der Hashcode berechnet?
    - zunächst einmal ist in der Klasse *Object* eine Methode **`hashCode()`** definiert, die aus der Speicheradresse!!! des Objekts einen Hashcode berechnet.
    - Diese Art der Berechnung ist nicht besonders nützlich, wenn man für gleiche Zustände von verschieden Objekten den gleichen Hashcode erhalten möchte.
    - Man sollte also die Methode **`hashCode()`** überschreiben, bzw. bei verwendeten Klassen überpüfen, ob in dieser Klasse die Methode **`hashCode()`** überschrieben worden ist, wie z. B. in *String*.
- Für die Berechnung von Hashcodes haben Sie im allgemeinen Teil einige Anregungen erhalten. Die einfachsten Lösungen sind meistens die besten! Ggf. Effizienz testen. Sie müssen als Hashwert nur eine beliebige ganze Zahl (auch negativ) ermitteln. Die Berechnung des *buckets* wird von der **`add()`**-Methode übernommen.
  - Tipp: Bei Warenhaltungssystemen gibt es meistens bereits eine Artikelnummer oder Teilenummer, die eindeutig ist. Nehmen Sie diese direkt als Hashwert.

14.7 Gestreute Speicherung (Hashing) in Java

```
// Beispiel von: Author Cay Horstmann (Core-Java)

import java.util.*;
import java.io.*;

public class SetTest

{  public static void main(String[] args) {

     Set words = new HashSet(59999);
     long totalTime = 0;

     try {
        BufferedReader in = new
                    BufferedReader(new InputStreamReader(System.in));
        String line;
        while ((line = in.readLine()) != null) {
           StringTokenizer tokenizer = new StringTokenizer(line);
           while (tokenizer.hasMoreTokens()) {
              String word = tokenizer.nextToken();
              long callTime = System.currentTimeMillis();
              words.add(word);
              callTime = System.currentTimeMillis() - callTime;
              totalTime += callTime;
           }
        }
     }
```

14.7 Gestreute Speicherung (Hashing) in Java

```
            catch (IOException e) {
               System.out.println("Fehler " + e);
            }
            Iterator iter = words.iterator();
            while (iter.hasNext()) {
               System.out.println(iter.next());
               System.out.println(words.size()
                  + " verschiedene Wörter. " + totalTime + " Millisekunden.");
            }
}
```


14.7 Gestreute Speicherung (Hashing) in Java

```
> cat inputfile.txt
Alle
meine
Entchen
schwimmen
auf dem See
Koefchen
in das
Wasser
Schwaenzchen
in die Hoeh
> java SetTest <inputfile.txt
die
das
dem
See
in
schwimmen
Koefchen
auf
Wasser
Hoeh
Alle
Schwaenzchen
meine
Entchen
14 verschiedene W?rter. 0 Millisekunden.
```