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.

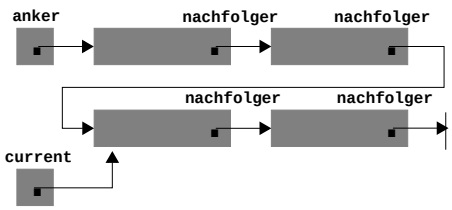
- So ist es auch nur möglich, komplexere Datenstrukturen, als verkettete Objekte aufzubauen. Java kennt zwar explizit keine Pointer, aber Referenzen auf Objekte, was nichts anderes ist, als dereferenzierten Pointer.

14.1 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 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 starkwachsen und/oder schrumpfen

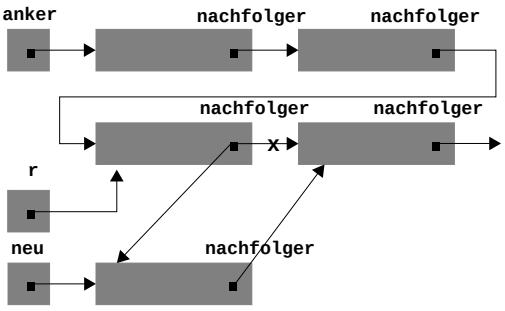
14.1 Verkettete Listen

- 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:



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.

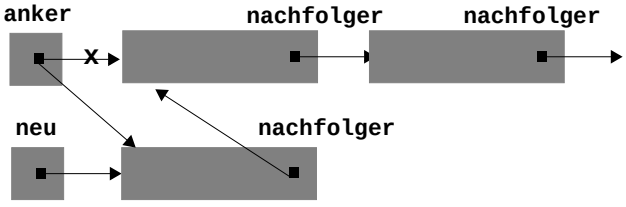



```
// Einfuegen hinter r
    neu.nachfolger = r.nachfolger
    r.nachfolder = neu
```


1 Einfügen

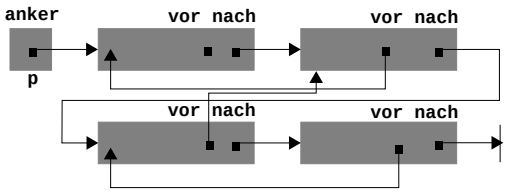
- Einfügen (cont):
  - Für den Sonderfall, dass das neue Listenelement ganz an den Anfang gehängt werden soll:

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



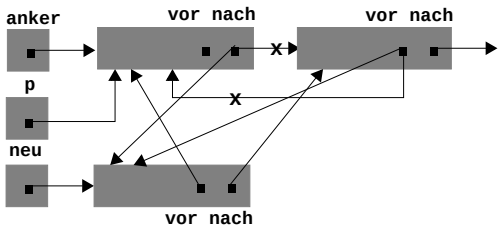
2 Doppelt verkettete Liste

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



2 Doppelt verkettete Liste

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



3 Realisierung in Java

- Datenstrukturen sind Objekte
- Datenelemente werden durch Objektvariablen repräsentiert

4 Beispiel Warteschlange

- Die Eigenschaft einer Warteschlange ist, dass man
  - Elemente an das Ende der Schlange anfügt und
  - Elemente vom Anfang 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.

4 Beispiel Warteschlange

◆ Als Element implementieren wir einen universellen Knoten (Node), mit dem man sowohl einfach verkettete Listen, als auch doppelt-verkettete Listen aufbauen kann:

```
// Hilfsklasse

class Node {

    String information;
    Node succElement;

    Node ( String info ) {
        information = info;
    }

    String getInformation() {
        return( information );
    }
 }
```


4 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;
        }
    }
```

4 Beispiel Warteschlange

```
    public String getInfos() {

        Node currentElement;
        String information ="";
        currentElement = firstElement;

        while ( currentElement != null ) {
            information += currentElement.getInformation();
            currentElement = currentElement.succElement;
        }

        return information;
    }
}
```

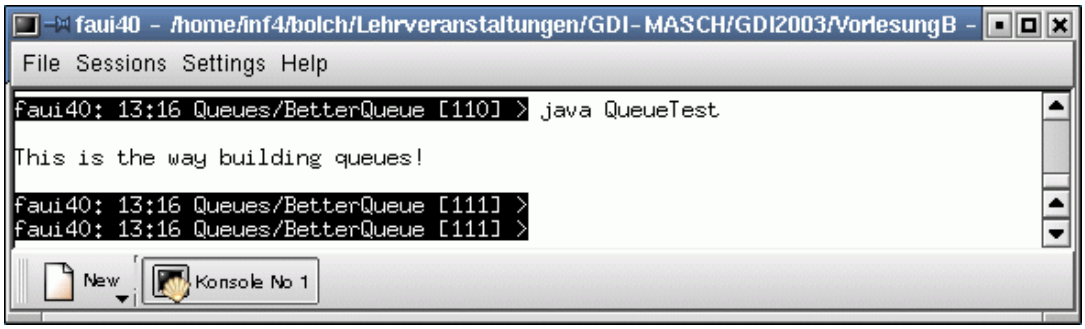

4 Beispiel Warteschlange

- 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 );
    }
}
```

4 Beispiel Warteschlange

◆ Ergebnis:



5 Linked Lists in Java

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

```
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" );

        System.out.println("Liste a: \n" + a + "\n" );
        System.out.println("Liste b: \n" + b + "\n" );
```

5 Linked Lists in Java

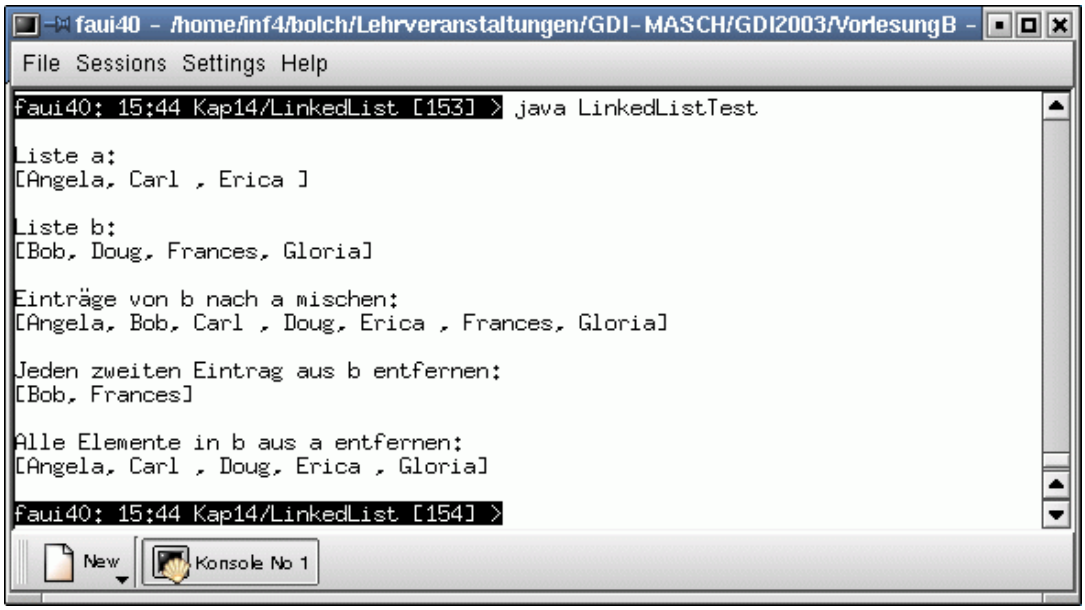
```
        System.out.println( "Einträge von b nach a mischen: " );

        ListIterator aIter = a.listIterator();
        Iterator bIter = b.iterator();

        while ( bIter.hasNext() ) {
            if ( aIter.hasNext() ) aIter.next();
            aIter.add( bIter.next() );
        }

        System.out.println( a + "\n" );
```


5 Linked Lists in Java

```
            System.out.println( "Jeden 2. 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 + "\n" );

            System.out.println( "Alle Elemente in b aus a entfernen:" );

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

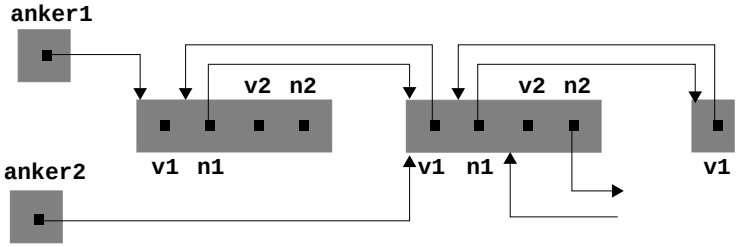
5 Linked Lists in Java

◆ Ergebnis:



14.2 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: alle Studenten; Teilliste “WRI”
  - Organisation nach zwei Kriterien:

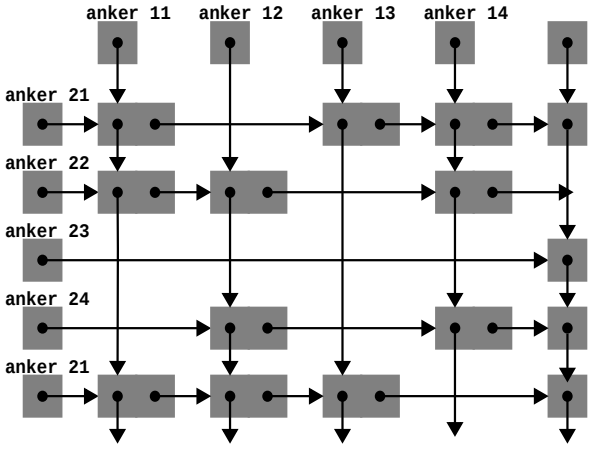


14.2 Multi-Listen

- 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.

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)

14.3 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”.*

1 Suchen

- Sehr häufiges Problem:

  “Finde Element mit bestimmten Schlüsselwert”

- Einfachste Lösung, Programmfragment:

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

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

1 Suchen

- Im folgenden werden sich alle Beispiele auf INTEGER - Schlüssel und -Felder beschränken.
- Das vereinfacht die Beispiele ohne auf das “Wesentliche” zu verzichten.
- 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.

1 Suchen

- Lösung:

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

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

2 Suchen 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

2 Suchen in sortierten Feldern

- Beispiel Binäre Suche:

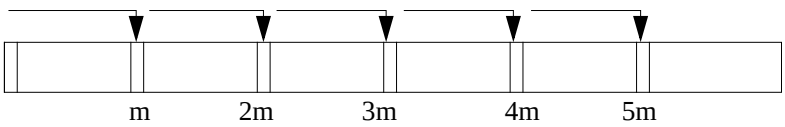
```
public class MySearch {

    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
    }
```

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();
    }
}
```


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.

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;
```


3 Sortieren

- Selection Sort (Sortieren durch Auswählen):
  - Ermittle kleinstes Element und setze es an den Anfang
  - Wiederhole dies für den Rest des Feldes
- Sortieren durch Einfügen:
  - Betrachte die N Elemente der Reihe nach
  - Füge jedes an der richtigen Stelle in die links von ihm stehende Folge ein
- 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 wie “Blasen” an das Ende des Feldes auf.

3 Sortieren

- Sortieren durch Vertauschen (Bubblesort):
  - Beispiel in Java - Code:

```
public class MySort {

    final boolean DEBUG = true;

    public void bubbleSort( int intArray[] ) {

        boolean change;
        int icount = 0, excount = 0;
        int temp;

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


3 Sortieren

- Sortieren durch Vertauschen (cont.):

```
        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
        }
```

3 Sortieren

- Sortieren durch Vertauschen (cont.):

```
            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();
        }
    }
```


4 Sort & Search - Arbeitsumgebung

- Klassen: **MySearch; MySort; TestSortAndSearch**

- 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( "\nEingabe: Länge des Feldes" );
        length = Integer.parseInt( inStream.readLine() );
        int array[] = new int[length];
```

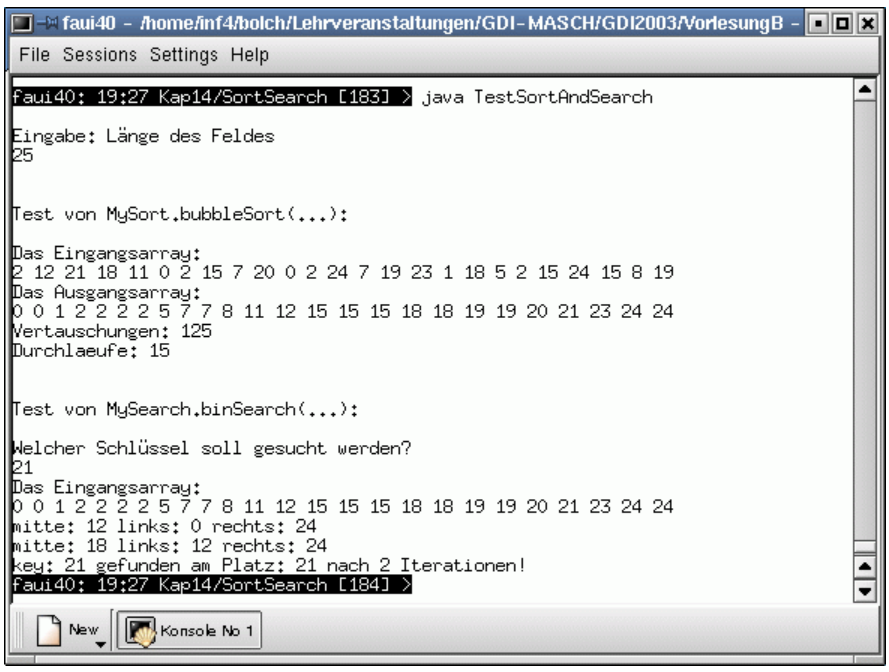
4 Sort & Search - Arbeitsumgebung

- TestSortAndSearch (cont.):

```
        // Aufbau eines Zufallszahlen Integer-Arrays:

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

        System.out.println( "\nTest von MySort.bubbleSort(...):" );

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

        System.out.println( "\nTest von MySearch.binSearch(...):" );

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

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


4 Sort & Search - Arbeitsumgebung

- Ergebnis:



5 Häufigkeitsgeordnete Listen

- Häugigkeitsgeordnete Listen
  - Sortierung *allein* zur Beschleunigung der Suche lohnt sich bei Listen kaum.
  - Die Suche kann 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).

5 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!)

14.4 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 Intervall der Zahlen von 0 bis m-1
  dann ist h: S --> A eine Hash-Funktion

14.4 Gestreute Speicherung (Hashing)

- HASH-Tabelle:
  - Damit das Verfahren funktioniert, muss die Datenbasis geeignet aufbereitet werden.
  - Die Elemente der Datenbasis werden mit der Hashfunktion (Ergebnis = ganze Zahl) abgespeichert.
  - Es wird eine Hash-Tabelle aufgebaut.
  - Es entsteht keine Ordnung der Einträge!

- 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.

14.4 Gestreute Speicherung (Hashing)

- 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