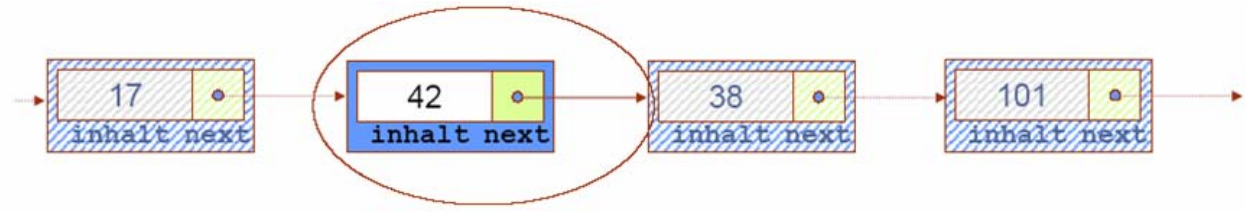
Kapitel 10: Algorithmen und Datenstrukturen

Algorithmen und Datenstrukturen wurden klassischerweise als „das“ Thema der praktischen Informatik betrachtet, in praktisch jedem Informatikstudiengang gibt es Spezialvorlesungen dazu. Andererseits ist die Bedeutung des Themas in der Praxis rückläufig, da es mittlerweise zu fast jedem Standard-Thema umfangreiche Bibliotheken gibt und man daher viele Algorithmen und Datenstrukturen nicht selbst implementieren, sondern einfach importieren wird. Für einen Informatiker ist es jedoch unerlässlich, zu wissen, wie die importierten Routinen prinzipiell funktionieren, sonst ist das Ergebnis vielleicht nicht optimal. Darüber hinaus gibt es auf dem Gebiet immer noch interessante Forschungsfragen und einige überraschende Effekte, besonders was die Komplexität gewisser Probleme betrifft.

10.1 Listen, Bäume, Graphen

Im Kapitel über abstrakte Datentypen waren diese durch ihre Signatur (Methodenköpfe) und Eigenschaften (algebraische Gesetze) definiert worden. In der Implementierung hatten wir gesehen, dass Klassen Datenfelder enthalten können. Jedes Objekt gehört zu einer bestimmten Klasse (Beispiel: int i; Thread f; PKW herbie usw.). Objekte können andere Objekte als Bestandteile enthalten (Beispiel: Auto enthält Tachostand). Frage ist, ob ein Objekt andere Objekte derselben Klasse enthalten kann (Beispiel: Schachtel enthält Schachtel)? Falls diese Möglichkeit in einer Sprache zugelassen ist, sprechen wir von *rekursiven Datenstrukturen*. (Sprachen wie C oder Delphi, die dieses Konzept nicht besitzen, greifen zur Realisierung meist auf maschinennahe Hilfsmittel wie Zeiger zurück.) In rekursiven Datenstrukturen ist es erlaubt, dass ein Objekt eine Komponente vom selben Typ enthält wie das Objekt selbst. Das Rekursionsende wird dann durch das „leere Objekt“ null gekennzeichnet.

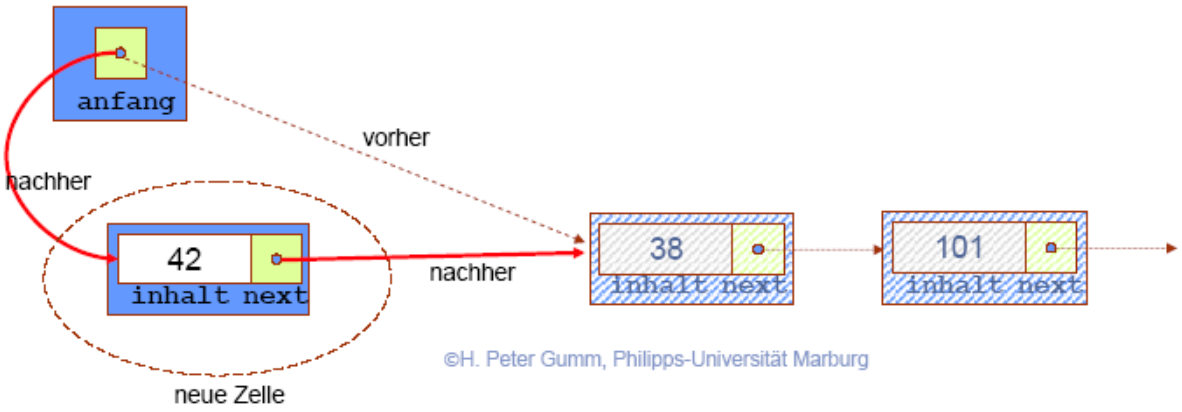
```
public class Zelle {
  int inhalt;
  Zelle next; // Verweis auf die nächste Zelle
  Zelle (int i, Zelle n){
    inhalt = i;
    next = n;
  }
}
```



Mit solchen Zellen lassen sich *verkettete Listen* von Zellen realisieren. Listen sind endliche Folgen von Elementen und dienen hier als Beispiel für eine *dynamische Datenstruktur*: Im Unterschied zu Arrays, bei denen bereits bei der Erzeugung eine feste Maximallänge angegeben werden muss, können Listen beliebig wachsen und schrumpfen. Neue Elemente werden einfach an die Liste angehängt, nicht mehr benötigte Elemente können wieder aus der Liste entfernt werden. Verkettete Listen kann man sich wie eine Perlenschnur vorstellen: von jeder Perle gibt es eine Verbindung zur folgenden. In den Perlen befindet sich die Information. Im konkreten Beispiel besteht die Liste aus einer Zelle, die den Anfang markiert. Jede Zelle hat einen Inhalt und einen Verweis auf die folgende Zelle. In der Klasse „Liste“ befinden sich darüber hinaus Methoden, um neue Elemente anzufügen, zu suchen, zu löschen usw. Vorne anfügen kann beispielweise dadurch erfolgen, dass man ein neues Element

erzeugt, das als Nachfolger den bisherigen Anfang hat, und dieses neue Element als neuen Anfang der Liste nimmt.

```
public class Liste {
  Zelle anfang;
  void prefix(int n){
    Zelle z = new Zelle (n,anfang);
    anfang = z;
  }
  void search(int n){...}
}
```



Zum Ausgeben einer Liste überschreiben wir die Objekt-Methode toString. Die Ausgabe erfolgt rekursiv gemäß der rekursiven Definition der Datenstruktur.

```
public class Zelle {
  ...
  public String toString(){
    return inhalt+((next==null)?"":   " -> " + next.toString());
  }
}

public class Liste {
  Zelle anfang;
  ...
  public String toString() { return anfang.toString(); }
}
```

Hier ist ein kleines Beispiel zum Testen der Klasse „Liste“.

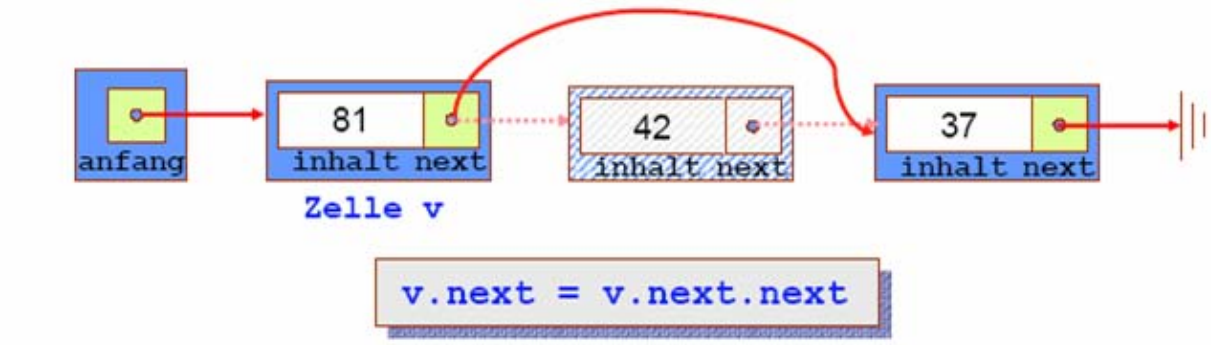
```
public class ListenAnwendung {
  static Liste l = new Liste();
  public static void main(String[] args) {
    l.prefix(101);
    l.prefix(38);
    l.prefix(42);
    System.out.println(l);
  }
}
```

Dieser Test druckt `42 -> 38 -> 101.`

Entfernen von Elementen einer Liste: Um das erste Element zu löschen, setzen wir den Anfang einfach auf das zweite Element:

```
void removeFirst () {
  anfang = anfang.next;
}
```

Der vorherige Anfang ist damit nicht mehr zugreifbar! Natürlich darf diese Methode nur ausgeführt werden, falls die Liste nicht leer ist. Um ein inneres Element zu löschen, verbinden wir die Vorgängerzelle mit der Nachfolgerzelle.



Man beachte, daß die Liste dadurch verändert wird! Das abgeklemmte Element ist vom Programm aus nicht mehr zugreifbar. Das Java-Laufzeitsystem sorgt dafür, dass dies irgendwann von der Speicherbereinigung entdeckt wird und der Platz wiederverwendet werden kann. Programmiersprachlich lässt sich das Entfernen des n-ten Elementes etwa wie folgt definieren:

```
void removeNth(int n) {
  if(anfang !=null) {
    Zelle v = anfang;
    while (v.next!=null && n>1) {
      v = v.next;
      n--;
    }
    if(v.next!=null) v.next = v.next.next;
  }
}
```

Achtung: removeFirst() ist nicht dasselbe wie removeNth(1). Auf ganz ähnlich Art lassen sich weitere Listenoperationen rekursiv definieren.

```
int laenge(){
  int i = 0; Zelle z = anfang;
  while (z!=null) {
    i++;
    z=z.next;
  }
  return i;
}

boolean enthaelt(int n) {
  boolean gefunden = false;
  Zelle z = anfang;
  while (z!=null && !gefunden) {
    if (z.inhalt==n) gefunden=true;
    z=z.next;
  }
  return gefunden;
}

Zelle suche(int n){
  Zelle z = anfang;
  while (z!=null) {
    if (z.inhalt==n) return z;
    z=z.next;
  }
  return z;
```

```
}
```

Eine Anwendung der Datenstruktur Liste ist die Realisierung von Kellern durch Listen.

```
class Stapel extends Liste {
  Stapel() {}
  Stapel(Zelle z) {
    anfang = z;
  }

  boolean isEmpty(){
    return anfang==null;
  }

  Stapel push(int i) {
    Zelle z = new Zelle(i, anfang);
    return new Stapel(z);
  }

  Stapel pop() {
    return new Stapel(anfang.next);
  }

  int top() {
    return anfang.inhalt;
  }
}
```

Ein Problem dieser Implementierung ist, dass bei jedem push und pop ein neuer Listenanfang generiert wird. Dieser wird zwar, wenn er nicht mehr benötigt wird, irgendwann von der Speicherbereinigung aufgesammelt. Trotzdem ergibt sich ein gewisser Effizienzverlust. Eine alternative Implementierung wäre etwa

```
Stapel popSE() {
  removeFirst();
  return this;
}
Stapel pushSE(int n) {
  prefix(n);
  return this;
}
```

Bei der Ausführung wird jetzt allerdings nur noch auf ein und derselben Liste gearbeitet! Dadurch ergeben sich weitere Seiteneffekte. Beispiel zur Demonstration:

```
public class StapelDemo {
  Stapel s1 = new Stapel();
  Stapel s2 = new Stapel();
  int sampleMethod() {
    s1=s1.push(111).push(222).push(333);
    s2=s1.pop();
    return s1.top();
  }
}
```

Es ergibt sich die Ausgabe 333, da s1 = `333 -> 222 -> 111`. Beim Austausch von pop durch popSE würde der Stapel s1 überschrieben, d.h. nach Aufruf von `s2=s1.popSE();` ist s1 = `222 -> 111` und die Ausgabe ist 222.

Schlangen

Mit verketteten Listen lassen sich auch Schlangen (queues) realisieren. Dazu braucht man einen zusätzlichen Zeiger auf den Anfang der Schlange. Um den Aufbau am Schlangenende und den Abbau am Anfang zu implementieren, wird die Verkettung sozusagen „umgedreht“

```
public class Schlange extends Liste {
  Zelle ende;

  boolean isEmpty(){
    return anfang==null;
  }

  int head() {
    return anfang.inhalt;
  }

  void tail() {
    anfang=anfang.next;
  }

  void append(int n) {
    Zelle z = new Zelle(n, null);
    if (isEmpty()) { anfang = z; ende = z; }
    else { ende.next = z; ende = z; }
  }
}
```

Ein Beispiel mit Schlangen:

```
class SchlangenDemo{
  static Schlange s = new Schlange();
  public static void main(String[] args) {
    s.append(100);
    System.out.println(s);
    s.append(200); s.append(300);
    System.out.println(s);
    System.out.println(s.head());
    s.tail();
    System.out.println(s.head());
  }
}
```

druckt

```
100
100 -> 200 -> 300
100
200
```

Doppelt verkettete Listen („Deque“, double-ended queue)
Für Listen, die auf beiden Seiten zugreifbar sein sollen, bietet sich eine symmetrische Lösung an. Für jede Zelle wird Nachfolger und Vorgänger in der Liste gespeichert. Beim Einfügen und Löschen müssen die doppelten Verkettungen beachtet werden.

```
public class Deque {
  class Item{
    int inhalt;
    Item links, rechts;
    public String printLR(){
      return inhalt + ((rechts==null)?"": "->" + rechts.printLR());}
    public String printRL(){
      return ((links==null)? "" :  links.printRL() + "<-") + inhalt;}
  }
  Item erstes, letztes;
  Deque() {}
  Deque(Item e, Item l) { erstes=e; letztes=l; }
```

```
  public void print(){
    if (! isEmpty()){
      System.out.println(erstes.printLR());
      System.out.println(letztes.printRL());
    }
  }
  boolean isEmpty() { return (erstes==null); }
  int first (){ return erstes.inhalt; }
  void rest() {
    erstes.rechts.links = null;
    erstes=erstes.rechts;
  }
  void prefix(int i) {
    Item neu = new Item();
    neu.inhalt = i;
    if (this.isEmpty()) {
      erstes = neu; letztes = neu; }
    else {
      neu.rechts = erstes;
      erstes.links = neu;
      erstes = neu; }
  }
  int last (){ return letztes.inhalt; }
  void lead() {
      letztes.links.rechts = null;
    letztes=letztes.links;
  }
  void postfix(int i) {
    Item neu = new Item();
    neu.inhalt = i;
    if (this.isEmpty()) {
      erstes = neu; letztes = neu; }
    else {
      neu.links = letztes;
      letztes.rechts = neu;
      letztes = neu; }
  }
}
```

Löschen eines inneren Knotens erfolgt durch Ersetzung zweier verschiedener Zeiger.
Man betrachte z.B. die Liste `erna <-> mary <-> hugo`, löschen von `mary` erfolgt durch

```
erna.rechts = erna.rechts.rechts; hugo.links = hugo.links.links;
```

oder, alternativ durch

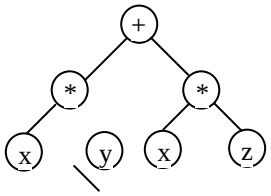
```
erna.rechts = erna.rechts.rechts; erna.rechts.links = erna;
```

Programmiersprachlich kann man das wie folgt realisieren:

```
void removeNth(int n){
  if (n==0) rest(); // erstes Element (n==0) ist zu löschen
  else {
    Item search = erstes;
    for (int i=1; i<n; i++) // gehe zu Element vor dem zu löschenden
      if (search == null || search.rechts == null){
        System.out.println("Liste zu kurz"); return; }
      else search = search.rechts;
    if (search.rechts == null){//search = letztes
        System.out.println("Liste zu kurz"); return; }
    else search.rechts = search.rechts.rechts;
    if (search.rechts == null) //letztes Element gelöscht
      letztes = search;
    else
      search.rechts.links = search;
  }
}
```

Bäume in Java
Syntaktisch sehen binäre Bäume genauso wie doppelt verkettete Listen aus.

```
class Bintree{
  Bintree left;
  char node;
  Bintree right;
...}
```



Die Verallgemeinerung auf n-äre Bäume ist offensichtlich:

```
class Ntree{
  String name;
  int children;
  Ntree [] child;
  Ntree (String s, int n) {
    name=s; children=n; child=new Ntree [n];}
  ...}
```

Anwendung von Binärbäumen:

- geordnete Binärbäume sind definiert durch die folgende Eigenschft:
  alle Knoten des linken Unterbaums < Wurzel < alle Knoten des rechten Unterbaums
- Das Einfügen geschieht daher je nach einzufügendem Inhalt
- Ebenso passiert das Suchen im entsprechenden Unterbaum
- Löschen ist etwas komplizierter
- Mit sortierten Binärbäumen erreicht man logarithmische Durchschnittskomplexität
- Problem: Bäume können entarten (d.h. nur wenige aber sehr lange Zweige haben)

Ein Beispiel für einen geordneten Binärbaum ist nebenstehend angegeben.

```
55
  11
    null
    null
  77
    66
      null
      null
    99
      null
      null
```

Einfügen in geordneten Binärbäumen macht man am besten rekursiv:

```
void insert (int i){ einfügen(i, this); }
private void einfügen (int i, Bintree b) {
 if (i<b.node)
  if (b.left==null) b.left=new Bintree(i);
  else einfügen(i, b.left);
 else if (i>b.node)
  if (b.right==null) b.right=new Bintree(i);
  else einfügen(i, b.right);
// else i==b.node, d.h. schon enthalten
}
```

Suchen in geordneten Binärbäumen ist sehr ähnlich zum Einfügen:

```
boolean find(int i) {
  return finde(i, this); }
private boolean finde(int i, Bintree b) {
  if (i<b.node)
    if (b.left==null) return false;
    else return finde(i, b.left);
  else if (i>b.node)
    if (b.right==null) return false;
    else return finde(i, b.right);
  else return true; //i==b.node
}
```

Löschen ist dagegen etwas komplizierter.?

- Beim Löschen eines Knotens muss ein Unterbaum angehoben werden
  - Wahlfreiheit (linker oder rechter Unterbaum?)
- Durch das Anheben entsteht eine Lücke, die wiederum durch Anheben eines Unterbaums gefüllt werden muss
  - Problem mit der Balance (Ausgewogenheit) des Baumes
  - Balancegrad beeinflusst die Komplexität des Suchens!
- Lösungsmöglichkeiten:
  - Abspeichern der Baumhöhe oder der Anzahl der Teilbäume für jeden Knoten
  - endlich verzweigte Bäume

Endlich verzweigte Bäume

```
class Ntree{
  String name;
  int children;
  Ntree [] child;
  Ntree (String s, int n) {
    name=s; children=n; child=new Ntree [n];} }

public class BeispielFamilie {
  private Ntree t;
  BeispielFamilie() {
    Ntree n28=new Ntree("Johanna",3);
    Ntree n55=new Ntree("Renate",0);
    Ntree n60=new Ntree("Angelika",2);
    Ntree n62=new Ntree("Margit",1);
    Ntree n89=new Ntree("Laura",0);
    Ntree n93=new Ntree("Linda",0);
    Ntree n98=new Ntree("Viktoria",0);
    n28.child[0]=n55; n28.child[1]=n60; n28.child[2]=n62;
    n60.child[0]=n89; n60.child[1]=n93;
    n62.child[0]=n98; t=n28;
  }
}
```

Suche in endlich verzweigten Bäumen erfolgt durch Iteration innerhalb einer Rekursion (!):

```
boolean search (String s) {
  return suche(s, this);
}
private boolean suche (String s, Ntree t) {
  if (t==null) return false;
  if (t.name==s) return true;
  for (int i=0; i<t.children; i++)
    if (suche(s, t.child[i])) return true;
  return false;
}
```

10.2 Graphalgorithmen

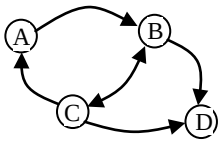
- Wdh.: Definition Graph
  - Darstellung einer binären Relationen über einer endlichen Grundmenge
  - Tupel (V,E), V endliche Menge von *Knoten*, E Menge von *Kanten*, zu jeder Kante genau ein *Anfangs*- und ein *Endknoten*
- Repräsentationsmöglichkeiten
  - als Relation (Menge von Paaren)

- Knotendarstellung, Verweise als Kanten
- Adjazenzmatrix

Knoten-Kanten-Darstellung

Syntaktisch lassen sich Graphen auch genauso wie endlich verzweigte Bäume darstellen:

```
class Graph{
  char node;
  int numberOfEdges;
  Graph [] edge;
  Graph (char c, int n) {
    node=c;
    numberOfEdges=n;
    edge=new Graph [n];
  }
}
```



Semantisch können Graphen Zyklen enthalten:

```
BeispielGraph() {
  Graph nodeA=new Graph('A',1);
  Graph nodeB=new Graph('B',2);
  Graph nodeC=new Graph('C',3);
  Graph nodeD=new Graph('D',0);
  nodeA.edge[0]=nodeB;
  nodeB.edge[0]=nodeD; nodeB.edge[1]=nodeC;
  nodeC.edge[0]=nodeA; nodeC.edge[1]=nodeB; nodeC.edge[2]=nodeD;
}
```

Anwendung von Graphen

- Beispiel: Verbindungsnetz der Bahn
- Suche Verbindung zwischen zwei Knoten
- Problem: Zyklen führen evtl. zu nicht terminierender Rekursion
- Lösung: Markieren bereits untersuchter Knoten (z.B. Eintragen in einer Menge)

Erreichbarkeit - fehlerhaft

```
boolean search (char c) {
  return suche(c, this); }
private boolean suche (char c, Graph g) {
  // Achtung fehlerhaft!!
  if (g==null) return false;
  if (g.node==c) return true;
  for (int i=0; i<g.edges; i++)
   if (suche(c, g.edge[i])) return true;
  return false;
}
// funktioniert nur falls Graph zyklenfrei
// d.h. wenn Graph n-fach verzweigter Baum ist
```

Erreichbarkeit – korrigiert

```
import java.util.*;
Set s;
boolean search (char c) {
  s = new HashSet();
  return suche(c, this); }
private boolean suche (char c, Graph g) {
  if (g==null) return false;
  if (s.contains(g)) return false;
  if (g.node==c) return true;
  s.add(g);
```

```
    for (int i=0; i<g.edges; i++)
      if (suche(c, g.edge[i])) return true;
    return false;
  }
```

Darstellung von Graphen als Adjazenzmatrix

- Angenommen, die Knoten seien $k_1 \ldots k_n$
- boolesche Matrix m der Größe n×n
  - m[i][j] gibt an, ob eine Verbindung von k_{i-1} zu k_{j-1} existiert oder nicht
- Vorbesetzung z.B.:

```
static void fillMatrixRandom
         (boolean matrix[][], float f) {
   for (int i = 0; i<n; i++) {
    for (int j = 0; j<n; j++) {
     matrix[i][j]=(Math.random()<f); } } }
```

transitive Hülle

Def. transitive Hülle:

$xR^*y \leftrightarrow xRy \vee \exists z\,(xR^*z \wedge zR^*y)$

- Algorithmus von Warshall:
  - starte mit $R^*=R$
  - für jeden möglichen Zwischenknoten z:
    - Berechne für alle x und y, ob $xR^*y \vee (xR^*z \wedge zR^*y)$
  - Reihenfolge der Schleifen ist wichtig!
- Algorithmus von Floyd (kürzeste Wege)
  - Entfernungsmatrix, min statt ∨, + statt ∧

```
static void warshall(boolean m[][], boolean t[][]) {
   for (int x = 0; x<n; x++) {
    for (int y = 0; y<n; y++) {
     t[x][y]=m[x][y]; }
   }
   for (int z=0; z<n; z++) {
    for (int x = 0; x<n; x++) {
     for (int y = 0; y<n; y++) {
      t[x][y]=t[x][y]||t[x][z]&&t[z][y]; }
     }
   }
 }
```

10.3 Suchen und Sortieren

- Gegeben eine (irgendwie strukturierte) Sammlung von Daten
  - Spezielles Suchproblem: entscheide ob ein gegebenes Element in der Sammlung enthalten ist
  - Allgemeines Suchproblem: finde ein (oder alle) Elemente mit einer bestimmten Eigenschaft
- Algorithmen hängen stark von der Struktur der Datensammlung ab!
  - im Folgenden: als Reihung (Array) organisiert

lineare Suche

- Wenn über den Inhalt der Reihung nichts weiter bekannt ist, muss sie von vorne bis hinten durchsucht werden

```
public class Suche {
```

```
 public static final int n = 10;
  static void printReihung(int reihung[]) {
   for (int i = 0; i<n; i++)
     System.out.print(reihung[i] + " ");
   System.out.println("");
  }
  static void fillReihungRandom(int reihung[]) {
   for (int i = 0; i<n; i++)
     reihung[i]=(int) (Math.random()*10);
  }
  static int sucheLinear(int suchreihung[], int suchinhalt) {
   for (int i=0;i<n;i++) {
    if (suchreihung[i]==suchinhalt) return(i);
   }
   return (-1);//oder exception
  }
 public static void main(String[] args) {
   int[]r=new int[n];
   fillReihungRandom(r);
   printReihung(r);
   System.out.println(sucheLinear(r,3));
 }
}
```

- Komplexität: mindestens 1, höchstens n Schleifendurchläufe ➔ O(n)

binäre Suche

- Wenn der Inhalt der Reihung aufsteigend sortiert ist, können wir es besser machen:

```
public class Suche {
 public static final int n = 10;
  static void printReihung(int reihung[]) {
   for (int i = 0; i<n; i++)
     System.out.print(reihung[i] + " ");
   System.out.println("");
  }
  static void fillReihungSorted(int reihung[]) {
   final int inc = 3;
   reihung[0]=(int) (Math.random()*inc);
   for (int i = 1; i<n; i++)
     reihung[i]= reihung[i-1] + (int) (Math.random()*inc);
  }
  static int sucheBZ(int sr[], int si, int lo, int hi) {
   if (lo > hi) return -1;
   int mitte = (hi + lo) / 2;
   if (si == sr[mitte]) return mitte;
   else if (si < sr[mitte])
    return sucheBZ(sr, si, lo, mitte - 1);
   else /* (si > sr[mitte])*/
    return sucheBZ(sr, si, mitte + 1, hi);
  }
  static int sucheBinaer(int suchreihung[], int suchinhalt) {
   return sucheBZ(suchreihung, suchinhalt, 0, n-1);
  }
 public static void main(String[] args) {
   int[]r=new int[n];
   fillReihungSorted(r);
   printReihung(r);
   System.out.println(sucheBinaer(r,7));
 }
}
```

Hashtabellen

- Wenn über den Inhalt der Reihung mehr bekannt ist, können wir es noch besser machen
  - Annahme: 10*i ≤ r[i] ≤ 10*i + 9,
    z.B. r[7] liegt zwischen 70 und 79,
    d.h. jedes Datenelement hat höchstens einen möglichen Platz

```
public class Suche {
 public static final int n = 10;
  static void printReihung(int reihung[]) {
   for (int i = 0; i<n; i++)
     System.out.print(reihung[i] + " ");
   System.out.println("");
  }
  static void fillReihungHashable(int reihung[]) {
   for (int i = 0; i<n; i++)
     reihung[i]= 10*i + (int) (Math.random()*10);
  }
  static int sucheHash(int suchreihung[], int suchinhalt) {
   int idx = suchinhalt / 10;
   return (suchreihung[idx]==suchinhalt)?idx:-1;
  }
 public static void main(String[] args) {
   int[]r=new int[n];
   fillReihungHashable(r);
   printReihung(r);
   System.out.println(sucheHash(r,77));
 }
}
```

Sortieren

- Oft lohnt es sich, Daten zu sortieren
  - einmalige Aktion, Abfragen häufig
  - oft gleich beim Eintrag möglich
- Wie sortiert man eine unsortierte Reihung?
  - selection sort: größtes Element an letzte Stelle, zweitgrößtes an zweitletzte Stelle, usw.
    - einfach aber ineffizient!
  - insertion sort: In absteigender Reihenfolge: das i-te Element in den sortierten Bereich [i+1, …, n] einsortieren
    - noch ineffizienter!
- Beispiele

Bubblesort

- Ineffizienz vorher: „weites“ Vertauschen mit Informationsverlust
- Idee: Tauschen von Nachbarn
  - Falls zwei Nachbarn in verkehrter Reihenfolge, vertausche sie
  - nach dem i-ten Durchlauf sind die obersten i Elemente sortiert

```
public class Sortieren {
 public static final int n = 5;
  static void printReihung(int reihung[]) {
   for (int i = 0; i<n; i++)
     System.out.print(reihung[i] + " ");
   System.out.println("");
  }
  static void fillReihungRandom(int reihung[]) {
   final int range = 100;
```

```
    for (int i = 0; i<n; i++)
      reihung[i]=(int) (Math.random()*range);
   }
  static void swap (int[] a, int i, int j) {
   int h = a[i]; a[i] = a[j]; a[j] = h; }

  static void bubbleSort (int[] a) {
   for (int k=n-1; k>0; k--) {
    for(int i=0; i<k; i++) {
     if (a[i]>a[i+1]) {
      swap(a,i,i+1);
      printReihung(a);
     }
    }
   }
  }
  public static void main(String[] args) {
   int[]r=new int[n];
   fillReihungRandom(r);
   printReihung(r);
   bubbleSort(r);
   printReihung(r);
  }
 }
```

Quicksort

- berühmter, schneller Sortieralgorithmus
- wähle ein „mittelgroßes“ Element w=a[k], alle kleineren nach links, alle größeren nach rechts
- rekursiv linke und rechte Teile sortieren

```
public class Sortieren {
 public static final int n = 5;
  static void printReihung(int reihung[]) {
   for (int i = 0; i<n; i++)
     System.out.print(reihung[i] + " ");
   System.out.println("");
  }
  static void fillReihungRandom(int reihung[]) {
   final int range = 100;
   for (int i = 0; i<n; i++)
     reihung[i]=(int) (Math.random()*range);
  }
 static void swap (int[] a, int i, int j) {
  int h = a[i]; a[i] = a[j]; a[j] = h; }

 private static int partition(int[]a, int lo, int hi) {
  swap(a,(lo+hi)/2, hi);
  int w = a[hi], k=lo;
  for (int i=k; i<hi; i++)
   if (a[i]<w) {swap(a,i,k); k++;}
  swap(a,k,hi);
  return k;
 }
 public static void qSort(int[]a, int lo, int hi) {
  if (lo<hi) {
   int pivIndex = partition(a,lo,hi);
   qSort(a,lo,pivIndex-1);
   qSort(a, pivIndex+1, hi);
  }
 }
 public static void quickSort(int[] a) {
```

```
    qSort(a,0,n-1);
   }
   public static void main(String[] args) {
    int[]r=new int[n];
    fillReihungRandom(r);
    printReihung(r);
    quickSort(r);
    printReihung(r);
   }
  }
```

Partitionierung (Methode `partition(int[]a, int lo, int hi)`)

- Idee: Pivotelement w irgendwo aus der Mitte wählen (eigentlich egal), am rechten Rand (hi) ablegen
- Dann 3 Bereiche bilden
  - lo..k-1 : Elemente kleiner als w
  - k..i-1: Elemente größer gleich w
  - i..hi-1: unsortierte Elemente