

```

*** UNdirekt1-1:
    class Parkplatz {
private String name;
Konstruktor und getName;
private Auto meinAuto;
void setAuto ( Auto a ){if ( a . parkt ( ) )return ;
if {( meinAuto != null )//Parkplatz ist b e r e i t s b e s e t z t return ;}
meinAuto = a ; a . parke ( true );}
void removeAuto () { {if ( meinAuto == null ) //      kein      Auto      park return ;}
meinAuto . parke ( false );
meinAuto = null ;}
public Auto getAuto () {      return meinAuto ;}    }
    class Auto {
private String name;
Konsturkot und getName;
private boolean parkt = false ;
public boolean parkt () {      return this . parkt ; }
public void parke ( boolean b ) {      parkt = b ;      }
** Bidirekt 1-1
    class Parkplatz ... private Auto meinAuto ;
public void setAuto ( Auto a ) {      if ( meinAuto != null )
return
if ( a . getParkplatz () != null && a . getParkplatz () != this ) return
meinAuto = a ; a . setParkplatz ( this );
void removeAuto () { if ( meinAuto == null ) return
Auto a = meinAuto ;      meinAuto = null ; a . removeParkplatz ()
    class Auto . private Parkplatz meinParkplatz
void setParkplatz ( Parkplatz p ) {      if ( meinParkplatz != null )
return ; // Auto parkt b e r e i t s
if ( p . getAuto () != null && p . getAuto () != this )      return ; // Parkplatz von  anderem Auto  be set zt
meinParkplatz = p ;      p . setAuto ( this )
void removeParkplatz () { if ( meinParkplatz == null ) eturn ; // Auto parkt nicht
Parkplatz p = meinParkplatz ;      meinParkplatz = null ;      p . removeAuto ( ) ;
}
}
}
-----
** Undir 1-n
    class Parkhaus { // ...
private List <Auto> meineAut = new LinkedList<Auto>();
public void addAuto ( Auto a ){if ( a . parkt ( ) ) return ; //Auto parkt b e r e i t s
meineAutos . add ( a ) ; a . parke ( true );}
public void rmAu( Auto a ){if ( meineAut . remove ( a ) )//Auto hat h i e r geparkt a . parke ( false );}
public List <Auto> getAuto ()      return meineAutos
    class Auto { // ...
private boolean parkt = false ;
public boolean parkt () { return this . parkt ;}
public void parke ( boolean b ) { parkt = b ;} }
** Bidirekt 1-n
    class Parkhaus . .
private List <Auto> meineAutos = new LinkedList<Auto>();
public void addAuto ( Auto a ) {      if ( meineAutos . contains ( a ) ) return
if ( a . getParkhaus () != null && a . getParkhaus () != this ) return
meineAutos . add ( a ) ; a . setParkhaus ( this );}
public void removeAuto ( Auto a ) {      if ( ! meineAutos . contains ( a ) ) return
meineAutos . remove ( a ) ; a . removeParkhaus ( ) ;}
public List <Auto> getAuto () {      return meineAutos ;
    class Auto ...
private Parkhaus meinParkhaus ;
public void setParkhaus ( Parkhaus p ) { if ( meinParkhaus != null ) return
meinParkhaus = p ; p . addAuto ( this );
void removeParkhaus () { if ( meinParkhaus == null ) return
Parkhaus p = meinParkhaus ;      meinParkhaus = null ; p . removeAuto ( this );
pubic Parkhaus getParkhaus () { return meinParkhaus ;
}
}
}

```

```

** Bidirekt      n-m
class Student { // ..
private List<Vorlesung> meineVor = new LinkedList<Vorlesung>();
public void addVorlesung ( Vorlesung v ) {
if (!meineVorl . contains ( v )) { meineVorl . add ( v ); v . addStudent ( t h i s ); } }
public void removeVor ( Vorlesung v ) { if ( meineVorl . remove ( v )) v . removeSt ( t h i s ); }
public List<Vorlesung> getVorlesungen () { return meineVorl ; }
class Vorlesung { // ..
private List<Student> meineSt = new LinkedList<Student>()
public void addStudent ( Student s ){ if (!meineSt . contains ( s )){ meineSt . add ( s ); s .
addVorlesung ( t h i s ); }
public void removeStudent ( Student s ) { if ( meineSt . remove ( s ))s . removeVorlesung
( t h i s ); }
public List<Student> getStudenten () { return meineStudenten ;

```

---

```

HaufigstesWort :
TreeMap<String , Integer > h = new ....; auf null pruefen h-put (w, h . get (w)+1)}
for ( String w: wortliste ){ if (! h . containsKey (w)) h . put (w, 1 ) e l s e
int max = 0 ; String l ;
for (Map . Entry<St , Inte> e : h . entrySet ( ) ) { if ( e . getValue ( ) > max ) l = e . getKey ( ) ;
max = e . getValue ( ) ; } return l ;

```

---

```

Liste add woerter sortiert klausur 13/14:
if ( head == null ) head = newNode( null , w );
else if ( w . compareTo ( head . wort ) < 0 ) head = newNode( head , w );
else if ( w . equals ( head . wort )) head . h++;
else {Node p = head ;
while ( p . next != null && w . compareTo ( p . next . wort ) > 0 ) p = p . next ;
if ( p . next != null && w . equals ( p . next . wort )) p . next . h++;
else p . next = new Node( p . next , w )

```

---

MESSEN in Worstcase :

$n \log n \rightarrow (n \cdot \text{vergrößerung}) \cdot \log(n \cdot \text{vergrößerung}) \cdot \text{msec}$  durch  $n \log n$

$n^2 \rightarrow (\text{vergrößerung}^2) \cdot \text{msec}$

---

MEHRFACHES löschen : Node p = head.n; while(p.n != null) if (p.data == p.n.d ) p.n = p.n.n; else p = p.n;

---

Doppelt verkettete : löschen p.prev.next = p.next; p.next.prev = p.prev;

Einfügen von r NACH q : Node r = new Node(x,q.next,q);r.next.prev=r;q.next=r;

Einfügen von r VOR q : Node r = new Node(x,q,q.prev);r.prev.next=r;q.prev=r;

Einfügen nach head : Node r = new Node();r.data = 5;r.next = head.next;r.prev = head;

head.next = r;r.next.prev = r;

---

```

abstract class Spiel implements Comparable<Spiel>{
public int compareTo(Spiel s) {
if (this.getAnzSpieler() < s.getAnzSpieler()) return -1; == return 0; > return 1;}

```

---

k <: S <: Comparable<S> <: Comparable<? super S> <: Comparable<? super K> For die Sort methode

---

kleinset Elem in Binary : if (p == null) return null;if (p.left == null)p = p.right;

else p.left = delMinR(p.left);return p; GRÖßte -> p.right p.left vertauschen

---

Anzahl Blätter Binär private static int numberOfLeaves(Node p) { if (p == null) return 0; else if (p.next == null && p.right == null) return 1; else return numberOfLeaves(p.left)+numberOfLeaves(p.right);

---

INSERT Sortiert Aufsteigend :Node p = head;while (p.next != null && p.next.data < x)

p = p.next;p.next = new Node(p.next,x);

---

MAX löschen if (head.next == null)throw new NullPointerException();

Node p = head;while (p.next.next != null)p = p.next;

int r = p.next.data;p.next = null;return r;

---

Falls A <: B, dann folgt

C<? extends A> <: C<? extends B> (Kovarianz) und

C<? super B> <: C<? extends A> (Kontravarianz)

---

Contains absteigend Sortiert Verkettete :

```
Node p = head.next; while (p != null && p.d > x) p = p.next;
```

```
if (p == null || p.d < x) false; else true;
```

```
remove : gleich while wie oben; if (p != null && p.d == x) p = p.next;
```

```
LISTE und MAP zusammen : Map<String, Integer> anzVorJeDozent = new TreeMap<String, Integer>();
```

```
for (LV lv : lvList) { if (!anzVorJeDozent.containsKey(lv.dozent))
```

```
anzVorJeDozent.put(lv.dozent, 1); else anzVorJeDozent.put(lv.dozent, anzVorJeDozent.get(lv.dozent)+1); }
```

```
List<String> flDoz = new LinkedList<String>(); for (String doz : anzVorJeDozent.keySet())
```

```
if (anzVorJeDozent.get(doz) > 3) flDoz.add(doz); return flDoz;
```

```
ADD und GET Liste : public List<Integer> getList() {
```

```
List<Integer> l = new LinkedList<Integer>(); addListR(l, root); return l; }
```

```
private void addListR(List<Integer> l, Node p) {
```

```
if (p != null) { addListR(l, p.left); l.add(p.data); addListR(l, p.right); }
```

Binär REKURSIV:

```
-public boolean contains(int x) { return containsR(x, root); }
```

```
-private boolean containsR(int x, Node p) { if (p == null) return false;
```

```
else if (x < p.data) return containsR(x, p.left);
```

```
else if (x > p.data) return containsR(x, p.right);
```

```
else return true;
```

```
-public void insert(int x) { root = insertR(x, root); }
```

```
-private Node insertR(int x, Node p) { if (p == null) p = new Node(x);
```

```
else if (x < p.data) p.left = insertR(x, p.left);
```

```
else if (x > p.data) p.right = insertR(x, p.right); return p; }
```

```
private int getMin(Node p) { assert (p != null); while (p.left != null) p = p.left; return p.data; }
```

