Datenstrukturen und Algorithmen

Grundlegende Datenstrukturen

Allgemeine Eigenschaften von Daten

- Es existieren Basisdaten wie Zeichen, Wahrheitswerte (true, false), ganze Zahlen oder Gleitpunktzahlen.
- Daten können Beziehungen untereinander haben, wie z. B. Listen, hierarchische Datenstrukturen wie Bäume usw.

Basis-Datentypen

char: Menge der Zeichen,
int: Menge der ganzen Zahlen, die im Rechner darstellbar,
float: Menge darstellbarer Gleitpktzahlen (einfach genau),
double: Menge darstellbarer Gleitpktzahlen (doppelt genau),
Array: Zusammenfassung zusammengehöriger Daten des gleichen Typs.

Datenstruktur = Daten + Operationen

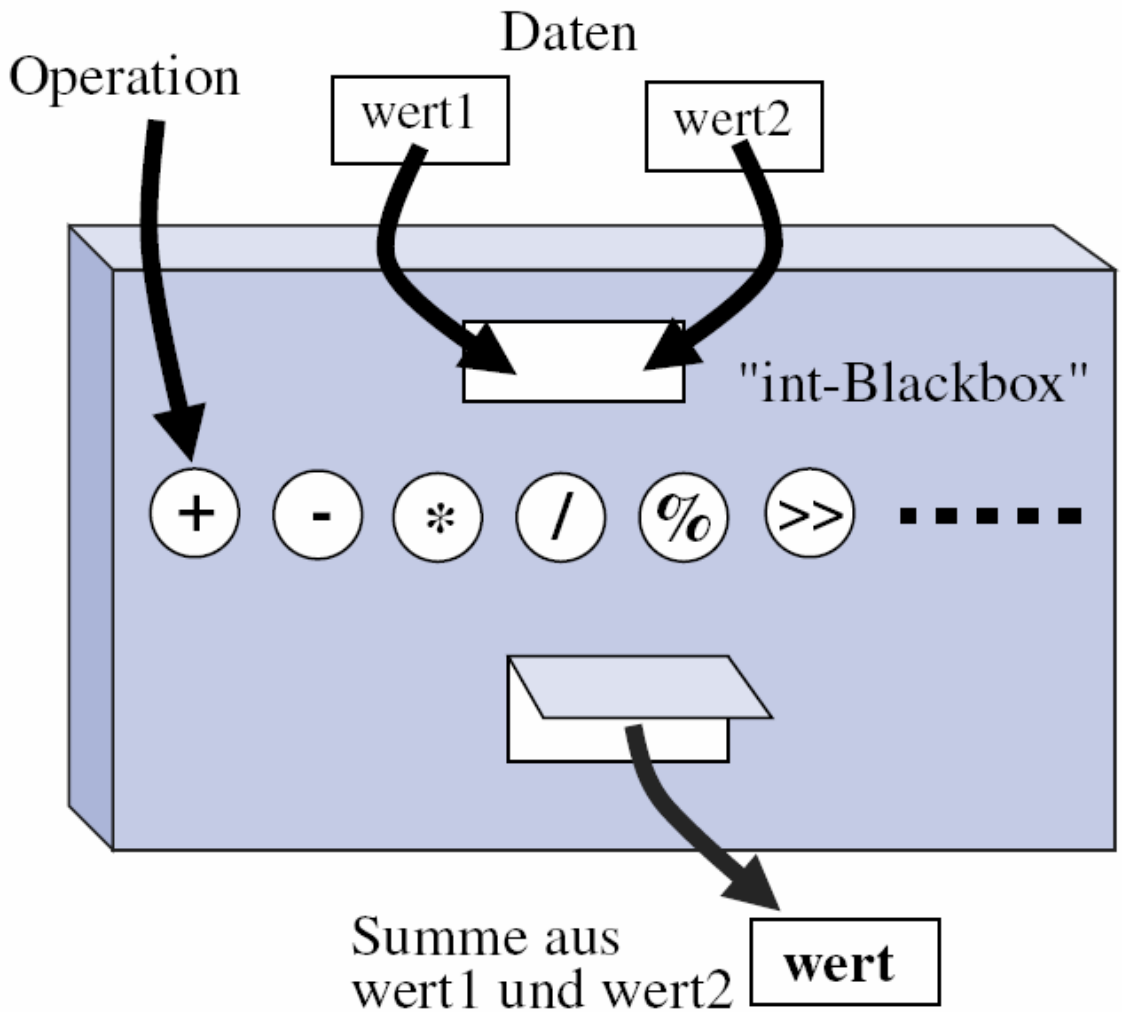
Unter Datenstruktur versteht man den Datentyp zusammen mit Operationen, die auf diesem Datentyp erlaubt sind.

Datentyp	Operationen	Bedeutung (Operandenzahl)
`short, int, float, double`	-	Negation des Werts (1)
`short, int, float, double`	+	Addition der Werte (2)
`short, int, float, double`	-	Subtraktion der Werte (2)
`short, int, float, double`	*	Multiplikation der Werte (2)
...	...	...

Für float und double sind nicht alle Operationen angeboten, die für ganzzahlige Datentypen wie short und int verfügbar.
→ z. B. kein Links-/Rechtsshift von Bits (<<, >>) und keine Modulo-Operation %

Datenstruktur = Daten + Operationen

Der Datentyp int und seine vordefinierten Operationen



Verkettete Listen

Einfach verkettete Listen

```
/* Rekursive Struktur in C */
struct elem {
    char          name[20];
    struct elem  *next;
};
```

```
/* Rekursives Objekt in Java */
class Elem {
    String   name;
    Elem     next;
     Elem(String s) { name = s; }
}
```

Beispiel:

Eintragen von Namen in einer Liste und dann rückwärts ausgeben

```
Namen:
Hans
Franz
Harald

..... Umgekehrt:
Harald
Franz
Hans
```

Einfach verkettete Listen

C-Programm

1) Zu Beginn **anfang** auf **NULL** gesetzt, da noch kein Eintrag in Liste vorhanden ist.

2) In while-Schleife mit **fgets() name** (z.B. „Hans“) einlesen. Bei Leerzeile wird while-Schleife verlassen und Eingabe der Namen beendet.

```
int   main(void) {
    struct elem
*anfang = NULL, *cursor;
    char name[20];
    printf("Namen:");
    while (1) {
        fgets (name, 20, stdin);
        if (strlen (name) == 1)
            break;
        /* Dynamisch Speicherplatz fuer
           Struktur 'elem' anfordern */
        cursor =malloc(sizeof(struct elem));
        if (cursor == NULL) {
            printf("Speicherplatzmangel\n");
            exit (1);
        }
        strcpy(cursor->name, name);
        cursor->next = anfang;
        anfang = cursor;
    }
    printf(" ..... Umgekehrt:\n");
    while (cursor != NULL) {
        printf("%s", cursor->name);
        cursor = cursor->next;
    }
```

Einfach verkettete Listen

Java-Programm

1) Zu Beginn **anfang auf null**, da noch kein Eintrag in Liste vorhanden.

2) In while-Schleife mit **ein.readLine() name** (z.B. „Hans“) einlesen. Bei Leerzeile while-Schleife verlassen und Eingabe der Namen beendet.

```
public class Liste1 {
  private static  Elem anfang;
  private static  Elem cursor;

  Liste1 ()  {  anfang = null;  }

  public static  void main(String arg[]) {
      String   name;
      Eingabe ein = new Eingabe();
      System.out.println("Namen:");
      while (true)  {
          name = ein.readLine("");
          if  (name.length() == 0)
              break;
          cursor = new Elem(name);
          cursor.next = anfang;
          anfang = cursor;
      }
      System.out.println(" ..... Umgekehrt:");
      while (cursor != null)  {
          System.out.println(cursor.name);
          cursor = cursor.next;
      }
  }
}
```

Einfach verkettete Listen

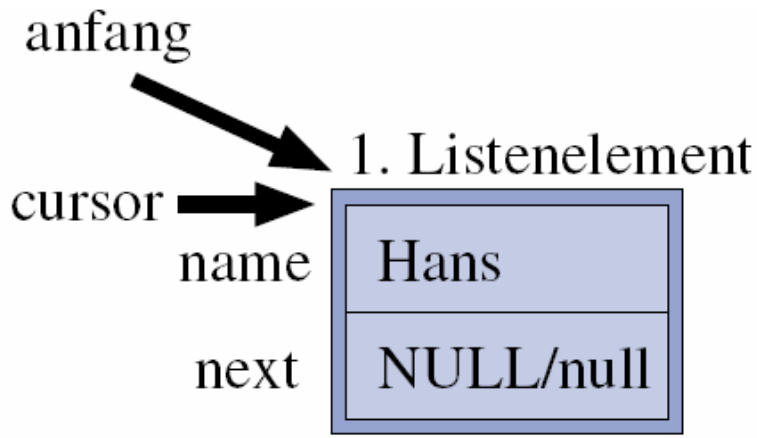
3) Für jeden Namen (solange keine Leerzeile) wird zusammenhäng. Speicher für elem angefordert und dessen Anfangsadr. in cursor abgelegt und dann wird dort aktuell eingegeb. Name abgelegt. cursor wird als Hilfszeiger bzw. Referenz auf innere Elemente in der Liste verwendet.

```
/* in liste1.c */
cursor =malloc(sizeof(struct elem));
if (cursor == NULL) {
    printf("Speicherplatzmangel\n");
    exit(1);
}
strcpy(cursor->name, name);
```

```
/* in Liste1.java */
cursor = new Elem(name);
```

Einfach verkettete Listen

4) Mit folg. Anweisungen wird beim 1. Durchlauf Komponente next von cursor auf NULL bzw. null gesetzt, da anfang mit NULL bzw. null vorbesetzt wurde.
→ noch mit „anfang = cursor“ Adr. von cursor in anfang abgelegt

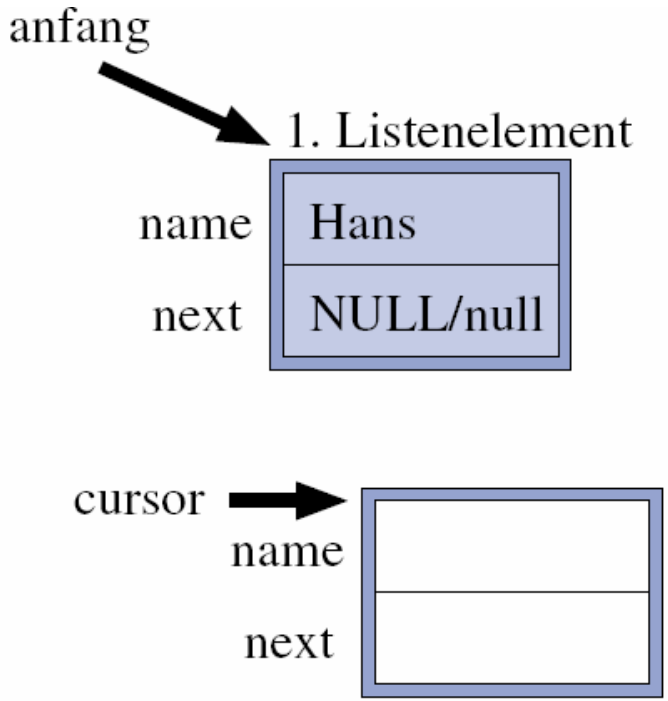



```
/* in liste1.c */
cursor->next = anfang;
anfang = cursor;
```

```
/* in Liste1.java */
cursor.next = anfang;
anfang = cursor;
```

Einfach verkettete Listen

5) Nun while-Schleife wieder von Beginn an. Nachdem wieder Name (z.B. „Franz“) eingelesen, wird mit malloc() bzw. new ein Speicher für elem angefordert und dessen Anfangsadresse in cursor festgehalten.



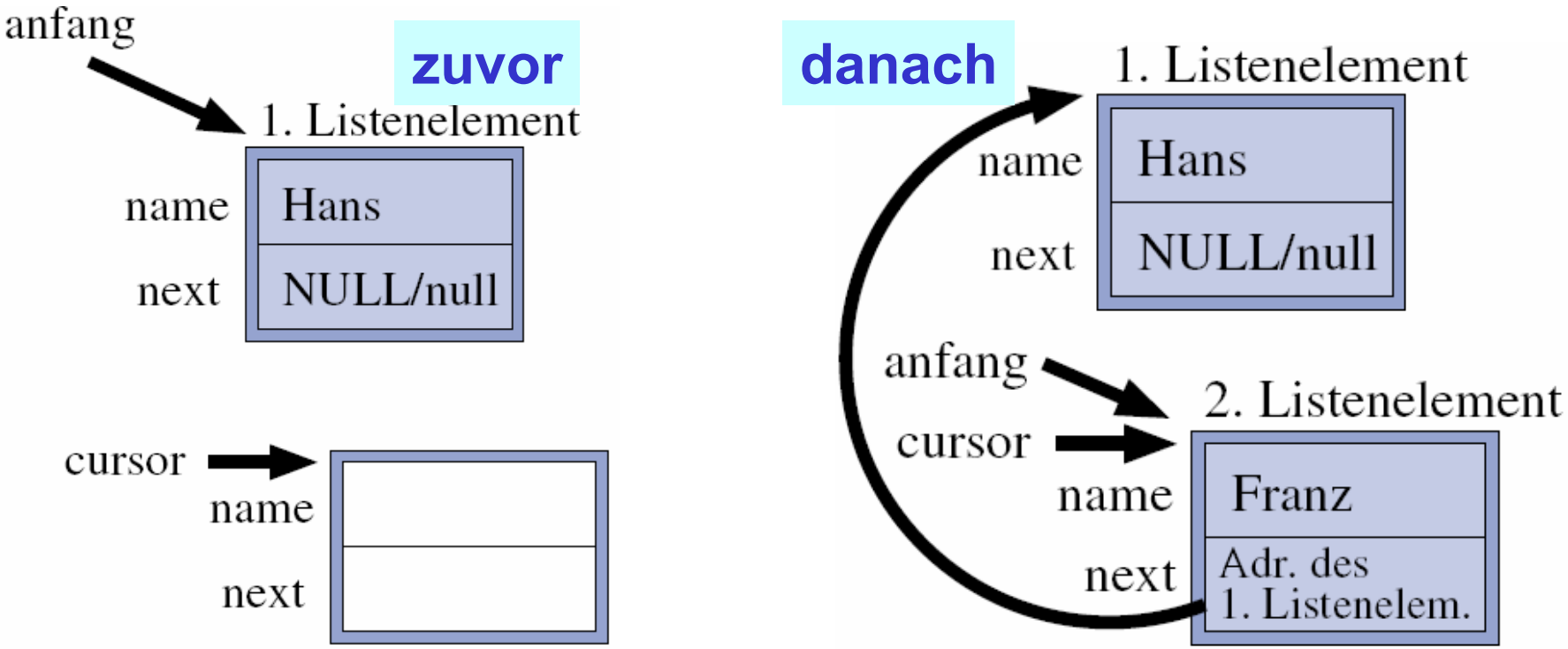

```
cursor =malloc(sizeof(struct elem));
cursor = new Elem(name);
```

Einfach verkettete Listen

6)

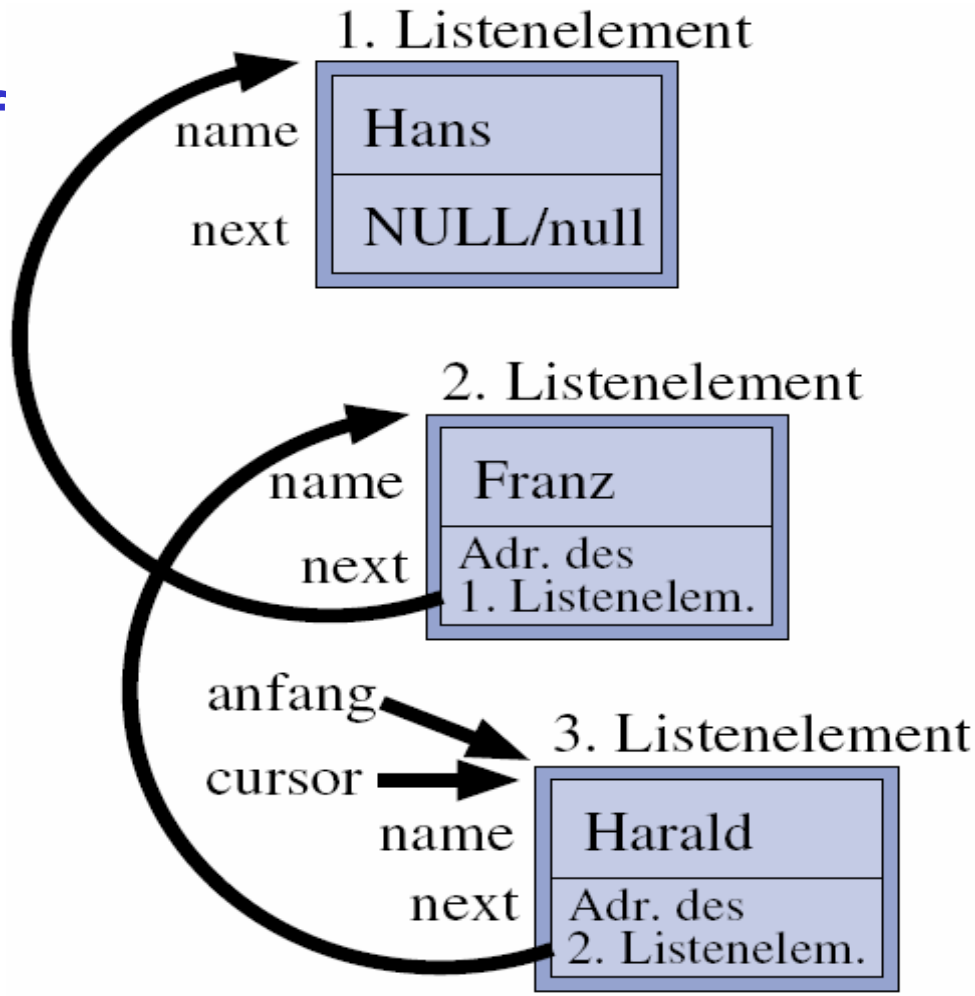
```
/* in liste1.c */
cursor->next = anfang;
anfang = cursor;
```

```
/* in Liste1.java */
cursor.next = anfang;
anfang = cursor;
```



Einfach verkettete Listen

7) Nach weiterem while-Schleifendurchlauf mit Eingabe des Namen „Harald“

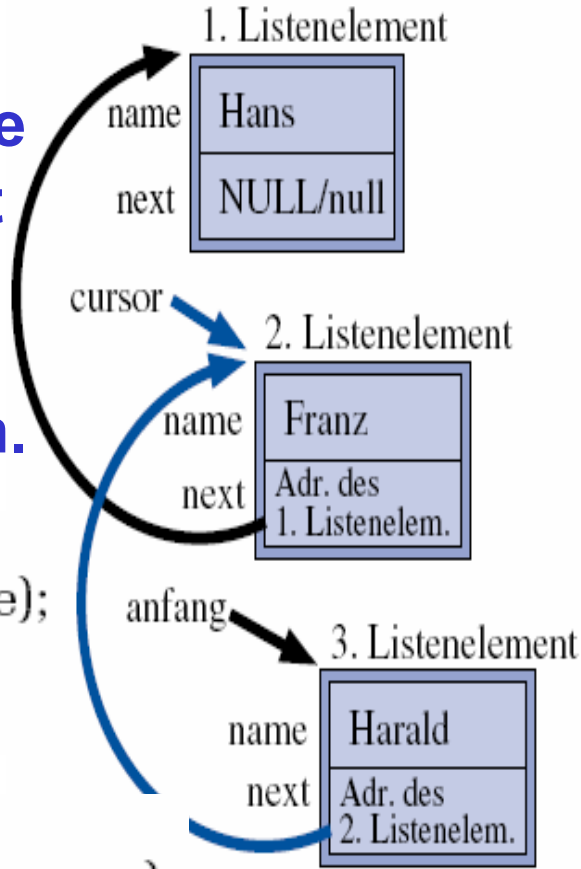


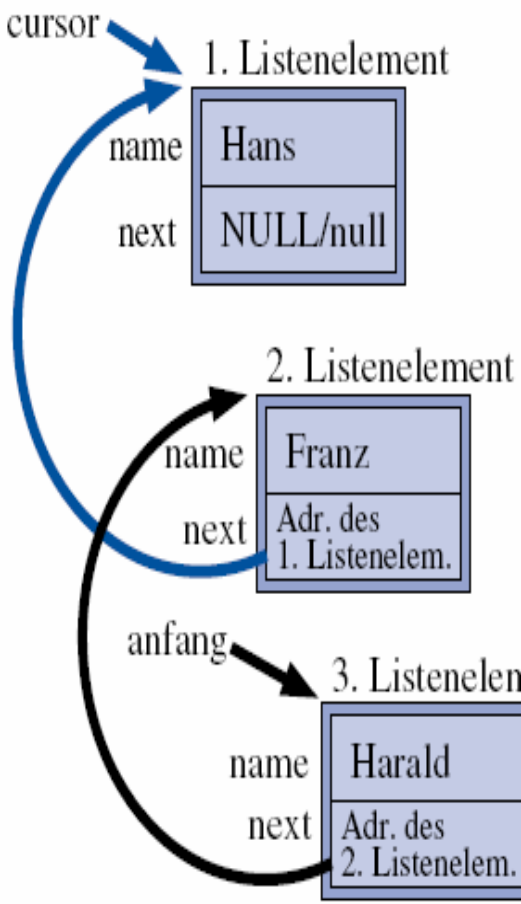
Einfach verkettete Listen

8) wird beim nächsten while-Durchl. Leerzeile eingegeb. → while mit break verlassen und in Liste vorh. Namen rückwärts ausgegeben.

```
while (cursor != NULL) {
    printf("%s", cursor->name);
    cursor = cursor->next;
}
```

```
while (cursor != null) {
    System.out.println(cursor.name);
    cursor = cursor.next;
}
```





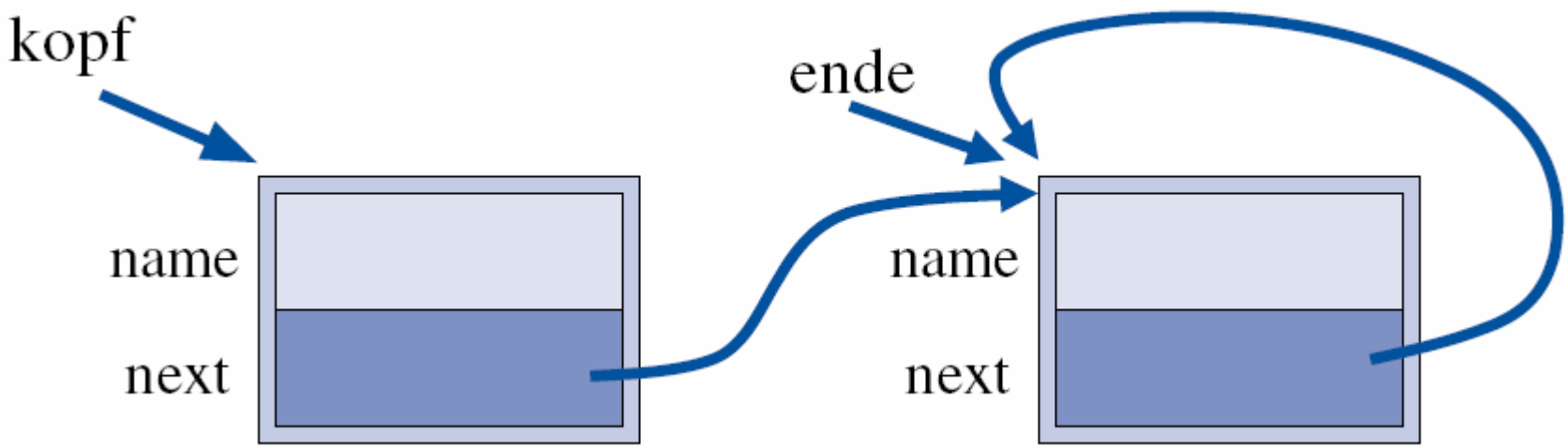
Sortierte Listen und Operationen für einfach verkettete Listen

```
public static  void main(String args[]) {
    kopf = new Elem(""); // Listen–Kopf und
    ende = new Elem(""); // Listen–Ende anlegen
    kopf.next = ende.next = ende;
```

```
class Elem {
    String   name;
    Elem     next;

    Elem(String s) {  name = s; }
}
```

Zwei Pseudoknoten kopf und ende

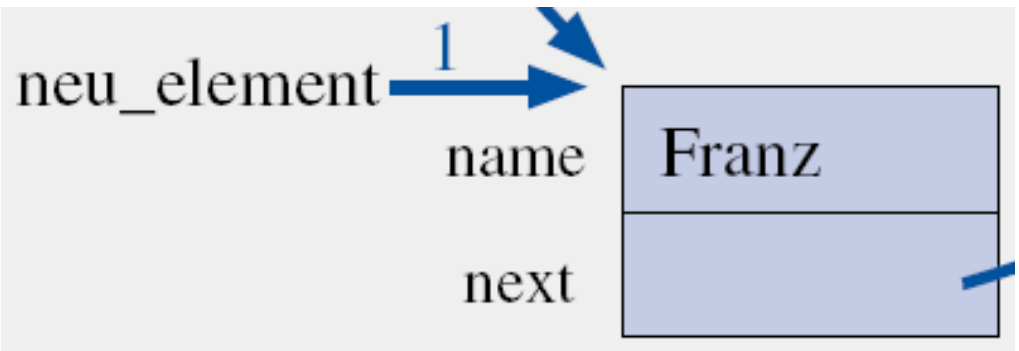


Sortierte Listen und Operationen für einfach verkettete Listen

Methode einfuegen()

1. Einlesen eines Namens, Anlegen neues Listenelement und Kopieren des eingelesenen Namens in neues Listenelement

```
Elem neu_element;
String name = ein.readLine("Gib den einzufuegenden Namen ein: ");
neu_element = new Elem(name);
```



Sortierte Listen und Operationen für einfach verkettete Listen

Methode einfuegen()

2. Suchen des Listenelements, nach dem das neue Listenelement einzufügen ist

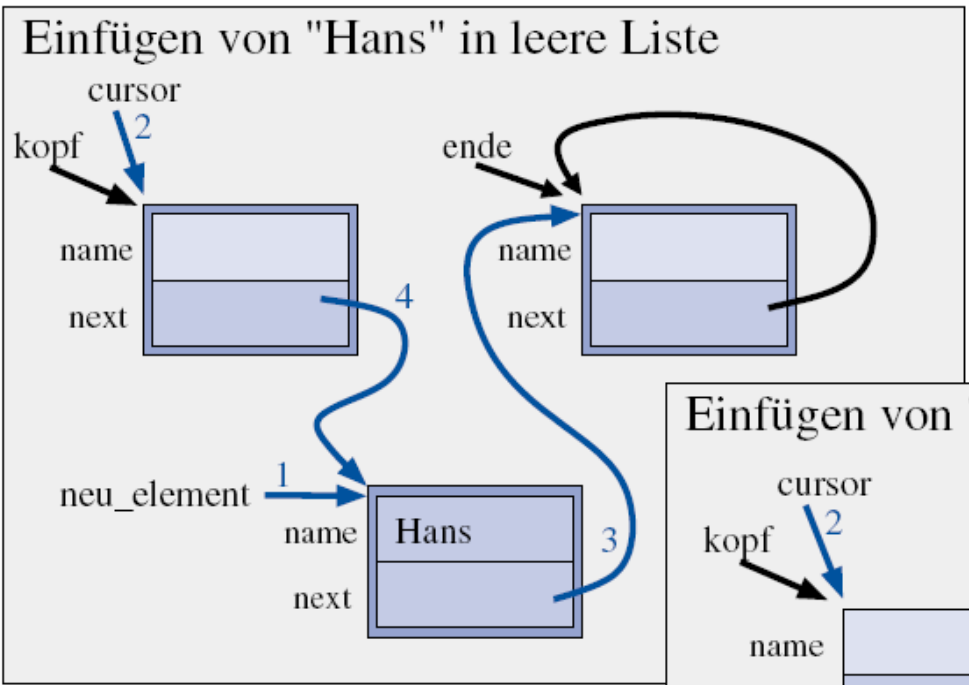
```
cursor = kopf; // Finden der Position, wo neuer Name in Liste einzufuegen ist
while (cursor.next != cursor.next.next) {
    if (name.compareTo(cursor.next.name) <= 0)
        break;
    cursor = cursor.next;
}
```

3. und 4. Einfügen des neuen Listenelements in der Liste

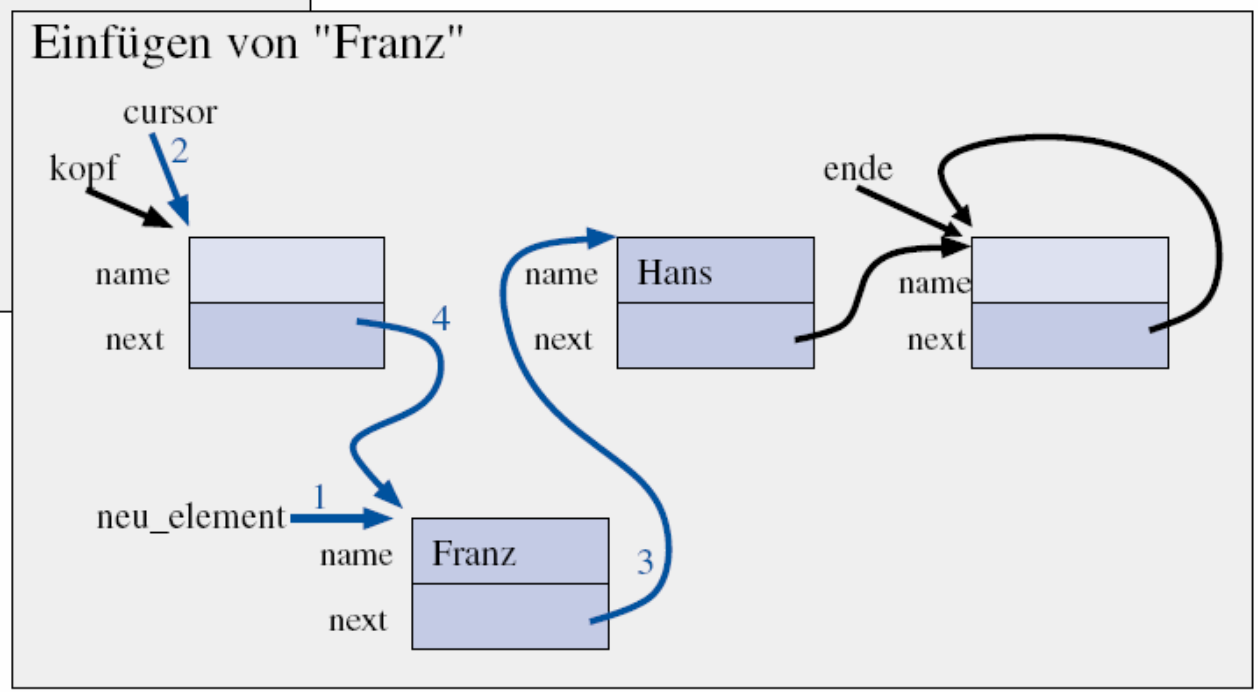
```
neu_element.next = cursor.next; // 3
cursor.next      = neu_element; // 4
```

Sortierte Listen und Operationen für einfach verkettete Listen

Methode einfuegen()



Einfügen von „Hans“ und danach „Franz“ mit einfuegen()



Sortierte Listen und Operationen für einfach verkettete Listen

Methode loeschen()

1. Einlesen zu löschender Namen und Suchen Listenelement, nach dem sich der zu löschende Knoten (Name) befindet

```
String name = ein.readLine("Gib den zu loeschenden Namen ein: ");
cursor = kopf; // Finden der Position des zu loeschenden Namens in Liste
while (cursor != cursor.next) {
    if (name.equals(cursor.next.name))
        break;
    cursor = cursor.next;
}
```

Trifft Suche auf Ende der Liste, ist der vom Benutzer eingegebene Name nicht in der Liste vorhanden:

```
if (cursor == cursor.next)
    System.out.println(" ..... Name '" + name +
                       "' nicht in Liste vorhanden....\n\n");
```

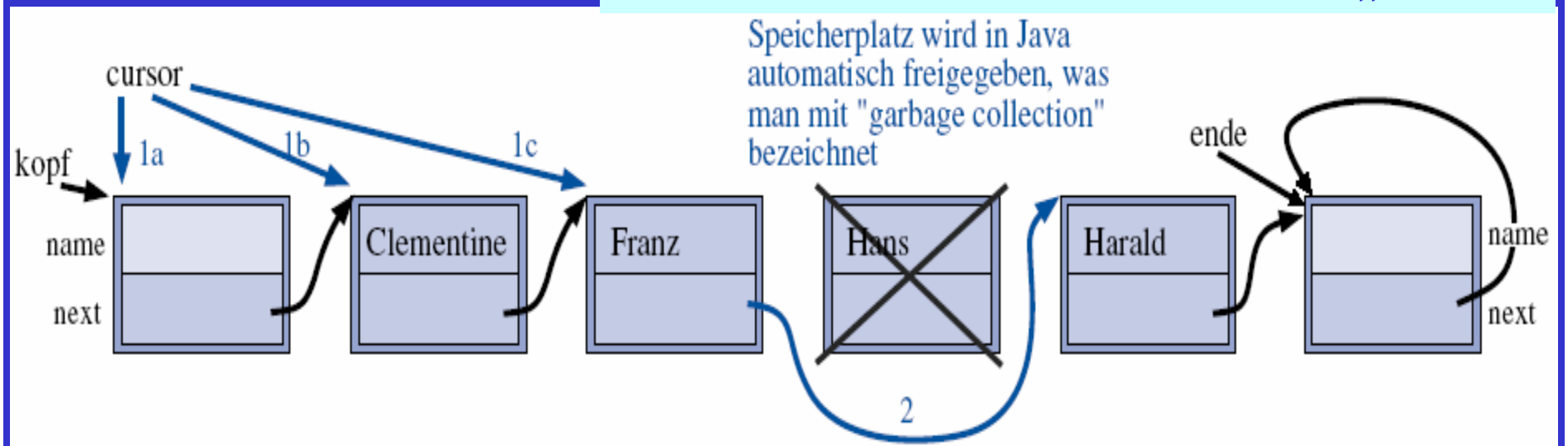
Sortierte Listen und Operationen für einfach verkettete Listen

Methode loeschen()

2. „Aushängen“ zu löschend. Elements (nur wenn gefunden)

```
cursor.next = cursor.next.next;
```

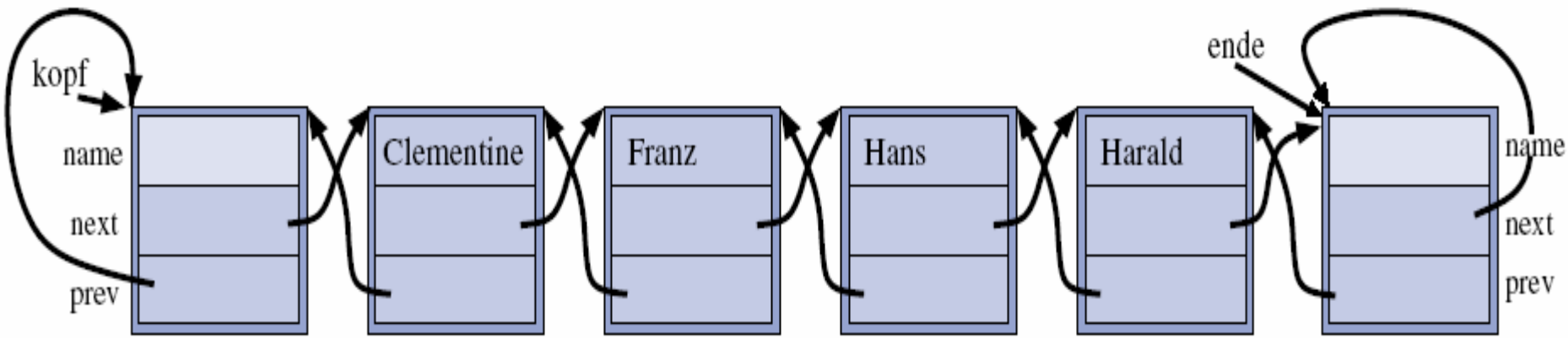
Löschen des Listenelements „Hans“



Doppelt verkettete Listen

```c
/* Doppelt verkettete Liste in C */
struct elem {
    char          name[20];
    struct elem  *next;
    struct elem  *prev; /* Vorgaenger */
};
```

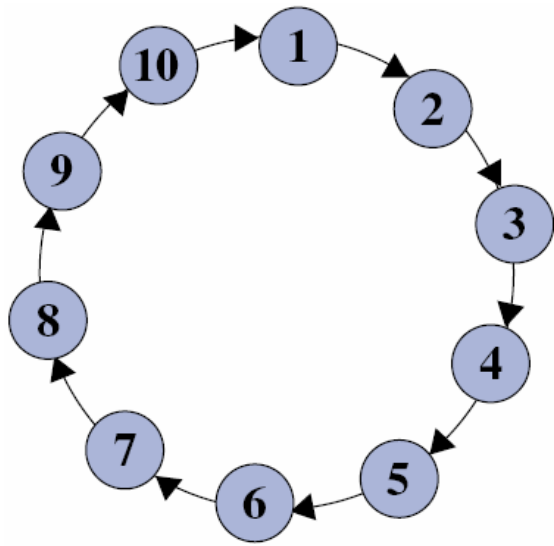
```java
/* Doppelt verkettete Liste in Java */
class Elem {
    String  name;
    Elem    next;
    Elem    prev; // Vorgaenger
     Elem(String s) { name = s; }
}
```



In doppelt verketteter Liste vorwärts und rückwärts bewegen

Ringlisten

```
class Person {
    int      nummer;
    Person  next;
    Person(int nr) { nummer = nr; }
}
```




```
Person pers, cursor, anfang = new Person(1);
cursor = anfang;
for (int i=2; i<=n; i++) {
    pers = new Person(i);
    cursor.next = pers;
    cursor      = pers;
}
cursor.next = anfang;
```

```
do { // Aussondern (Simulieren des Abzaehlvo
    for (int i=1; i<z; i++)
        cursor = cursor.next;
    System.out.print(cursor.next.nummer + ", ");
    cursor.next = cursor.next.next;
} while (cursor != cursor.next);
System.out.println(cursor.nummer);
```

Verkettete Listen

Vorteile von verketteten Listen gegenüber Arrays

+ geringerer Aufwand → Einfügen bzw. Entfernen eines Elements in Liste viel weniger aufwändig als bei Arrays
+ Anzahl der Elemente und dafür benötigter Speicherplatz muss nicht im Voraus angegeben werden
  → spätere Erweiterung → kein aufwändiges Umkopieren (Verschieben) an anderen zusammenhäng. Speicher

Nachteile von verketteten Listen gegenüber Arrays

- kein Direktzugriff auf beliebiges Element
  → Liste ab bestimmtem Punkt durchlaufen, bis man zu gewünschtem Element gelangt → kostet natürlich Zeit
- Zeiger- bzw. Referenzen belegen zusätzl. Speicherplatz