Hochschule **Rosenheim**
University of Applied Sciences

Prozedurale Programmierung

Datenstrukturen

Hochschule Rosenheim - University of Applied Sciences

WS 2018/19

Prof. Dr. F.J. Schmitt

Problem

- Felder sind relativ unflexibel
- wie löscht man z.B. eine Adresse in der Mitte eines Felds?
  - alle nachfolgenden verschieben? → sehr zeitaufwändig
  - ungültig markieren → Speicherverschwendung
- was, wenn man mehr Einträge braucht
  - realloc mit dynamischer Speicherverwaltung → zeitaufwändig
  - evtl. ist gar kein genügend großer Speicherblock mehr verfügbar
- Adressen sortieren
  - man muss die einzelnen Einträge umkopieren

- Hauptproblem mit Feldern:
  - alles liegt am Stück im Speicher
  - dadurch sehr unflexibel
  - einzelne Einträge können nur durch Umkopieren verschoben werden

Überblick

- Grundlagen
  - Unterschied zwischen Datentyp und Datenstruktur
- Verkettete Listen
  - charakteristische Eigenschaften
  - Implementierung
  - typische Operationen

Datentyp

- Was ist ein Datentyp?
  - Klassifizierung von Werten gleicher Art, wie bspw. ganze Zahlen, Gleitpunktzahlen oder Zeichen

- Elementare Datentypen in C:
  - `int`
  - `float`
  - `double`
  - `char`

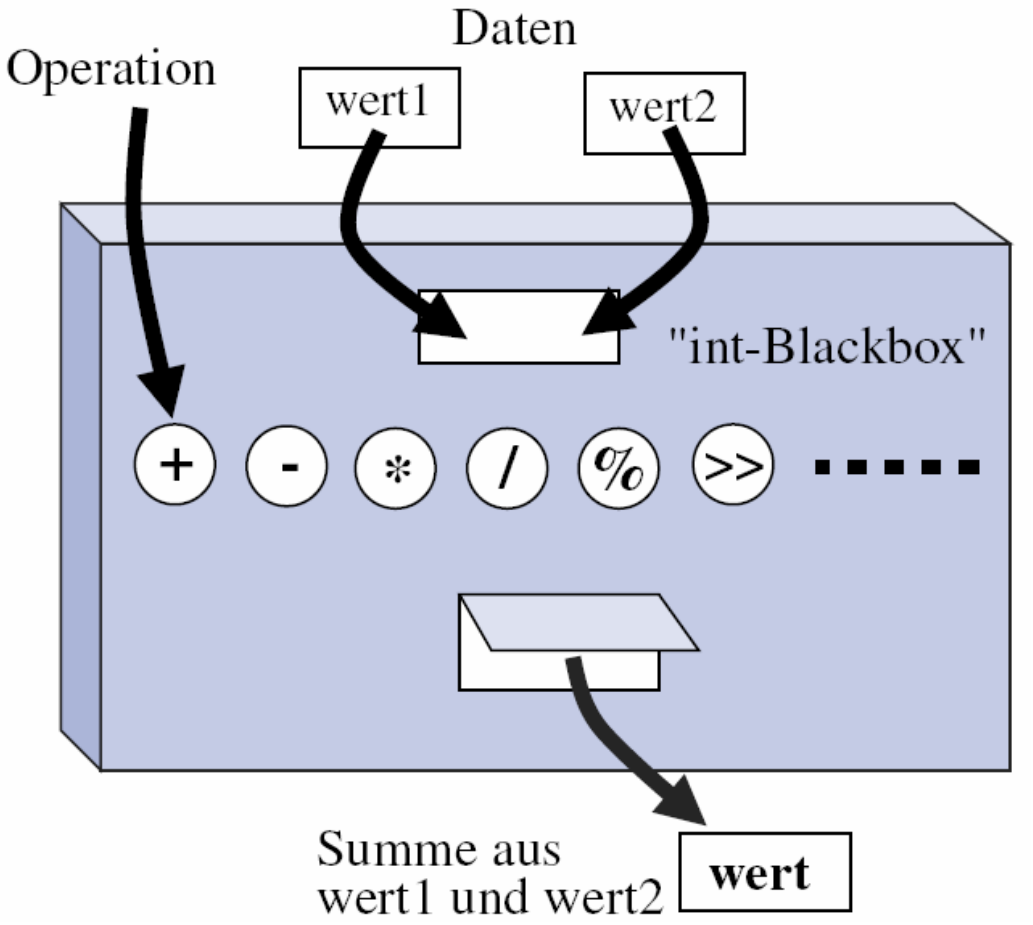
Datenstruktur (1)

- Was versteht man unter einer Datenstruktur?
  - Datentyp zusammen mit einer Menge von Operationen, die auf diesem Datentyp erlaubt sind
  - Beispiel: Auswahl elementarer Datentypen mit vordefinierten Operationen:

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)
...	...	...

Datenstruktur (2)

- Beispiel: Datenstruktur `int`



Datentyp `int` mit seinen vordefinierten Operationen

Konzept:

Abstrakter Datentyp (ADT)

Verkettete Listen (1)

- Typisches Beispiel einer Datenstruktur
- Charakteristische Eigenschaften:
  - Sequentielle Aneinanderreihung von Elementen
  - Folge von Elementen kann während des Programmablaufs dynamisch verlängert bzw. verkürzt werden
  - Die einzelnen Element müssen nicht hintereinander im Speicher liegen (können im Speicher verstreut sein)
    - Jedes einzelne Element der Liste muss eine Verbindung zum nächsten Element oder auch zum vorherigen Element haben
    - Realisierung: jedes Element enthält Speicheradresse des folgenden bzw. vorherigen Elements

Verkettete Listen (2)

- Durch die Verwendung von Zeigern und dynamischer Speicherverwaltung können verkettete Listen aufgebaut werden

- Verschiedene Arten:
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
  - Ringlisten

Einfach verkettete Liste (1)

- Realisierung von Listen in C mittels rekursiver Strukturen
  - Zeiger auf das nächste und/oder vorherige Element sind enthalten

- Beispiel:

```
struct element_s
{
    char name[20];
    struct element_s *next;
};
```

Einfach verkettete Liste Beispiel

- Erstes Beispiel: Eintragen von Namen in eine Liste und dann rückwärts ausgeben

- Erläuterungen C-Programm:
  - Zwei Zeiger:
    - `anfang` zeigt immer auf das erste Element (Programmstart : `anfang` wird auf `NULL` gesetzt, da noch kein Eintrag in Liste vorhanden ist)
    - `cursor` zeigt immer auf den neu reservierten Speicherblock

Einfach verkettete Liste
Element vorne einfügen (1)

Hochschule **Rosenheim**
University of Applied Sciences

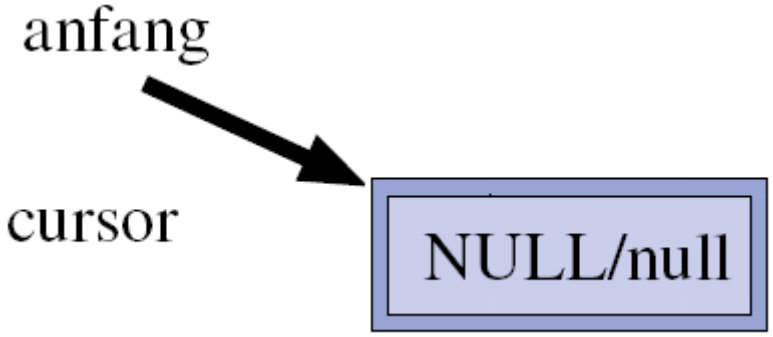
```
int main(void)
{
  struct element_s  *anfang = NULL, *cursor = NULL;
  char name[20];

  printf("Namen eingeben:");
  while (1)
  {
    fgets(name, 20, stdin); // Name + \n
    if (strlen(name) == 1)  // Ende wenn nur \n (Enter)
      break;

    // weiter auf der nächsten Folie
```

Einfach verkettete Liste
Element vorne einfügen (2)

```
    // dynamisch Speicher für element_s anfordern
    cursor = (struct element_s*)
              malloc(sizeof(struct element_s));
    if (cursor == NULL)
    {
       printf("Speicherplatzmangel\n");
       exit(1);
    }

    // eingelesenen Namen im angeforderten Speicher ablegen
    strcpy(cursor->name, name);

    // neues Element vorne in Liste einhängen
    cursor->next = anfang;
    anfang = cursor;
  }
}
```

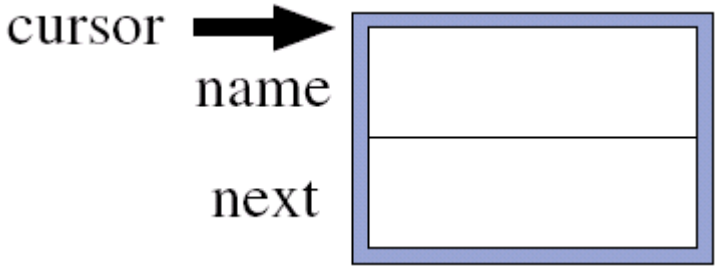
Einfach verkettete Liste
Element vorne einfügen (3)

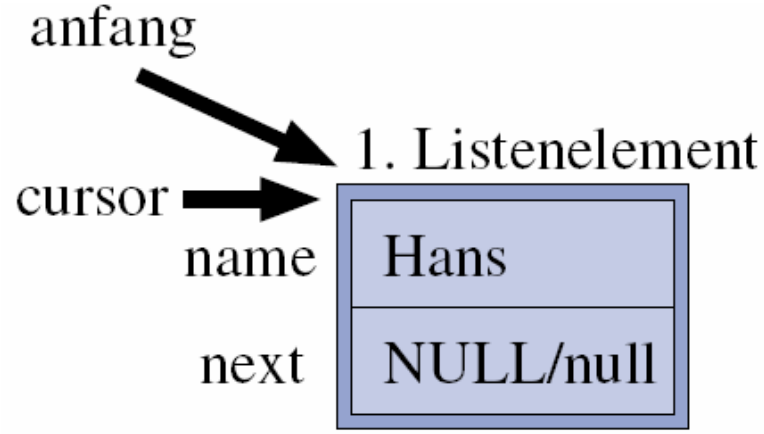
- Ausgangsituation Programmablauf:



Einfach verkettete Liste
Element vorne einfügen (4)

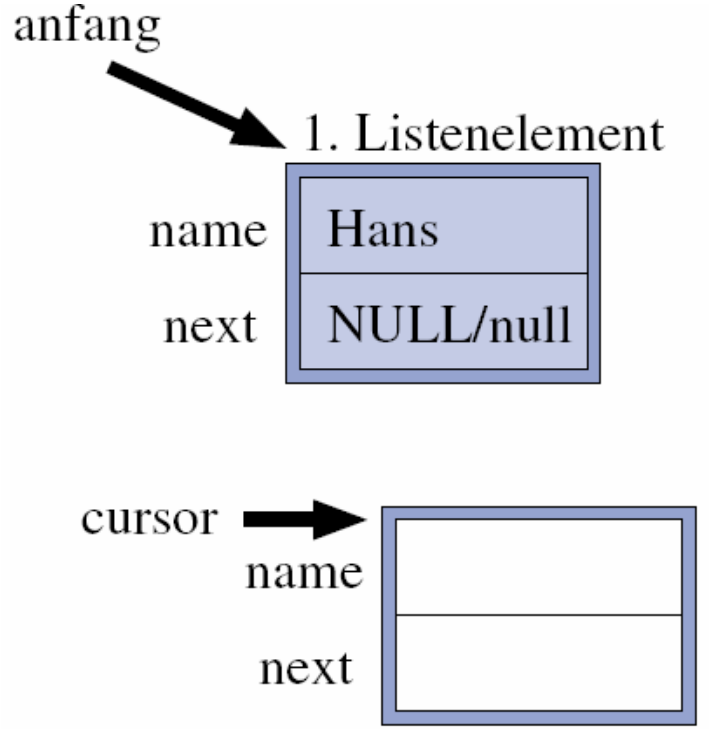
- Mögliches Szenario Programmablauf:

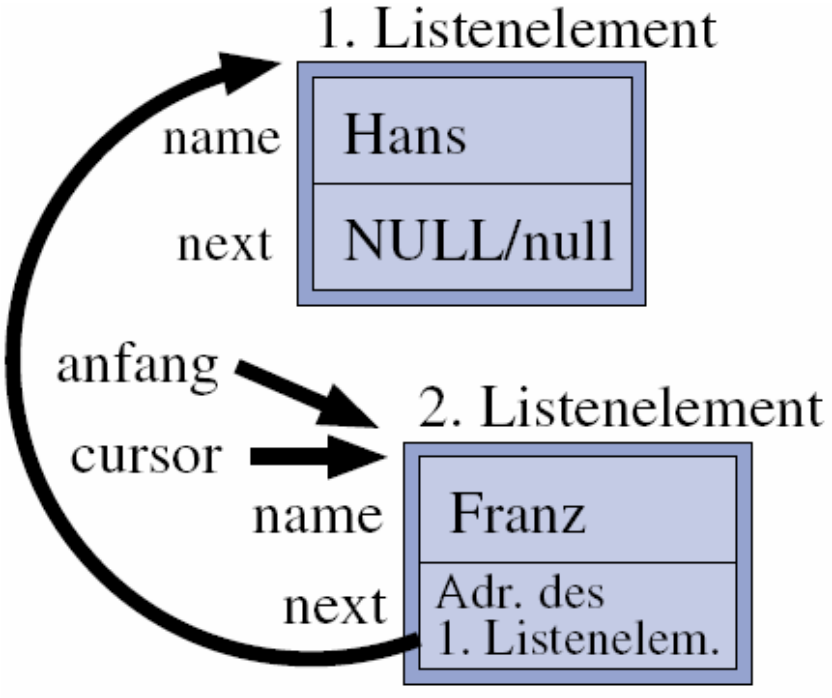




Einfach verkettete Liste
Element vorne einfügen (5)

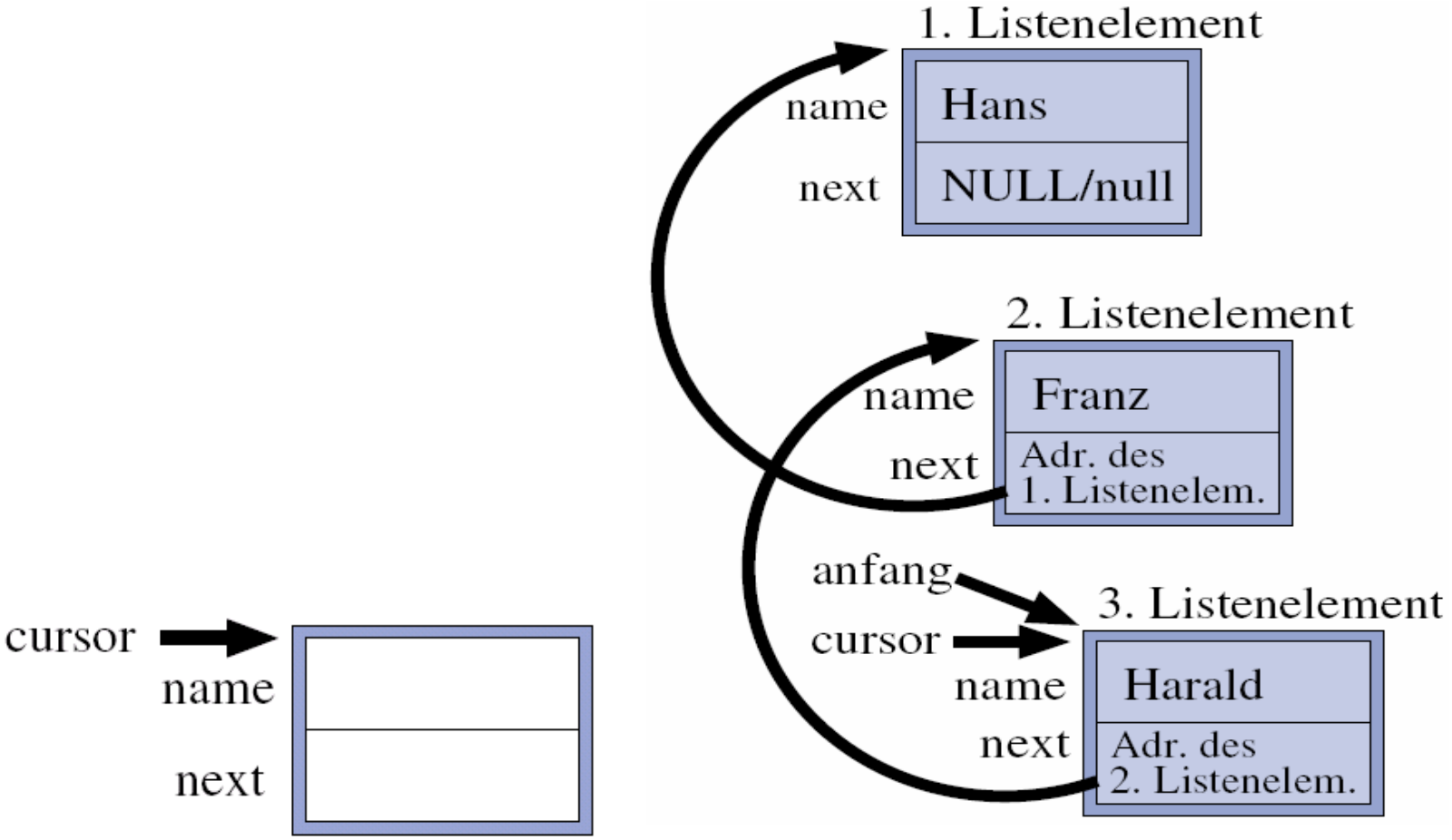
- Mögliches Szenario Programmablauf:





Einfach verkettete Liste
Element vorne einfügen (6)

- Mögliches Szenario Programmablauf:

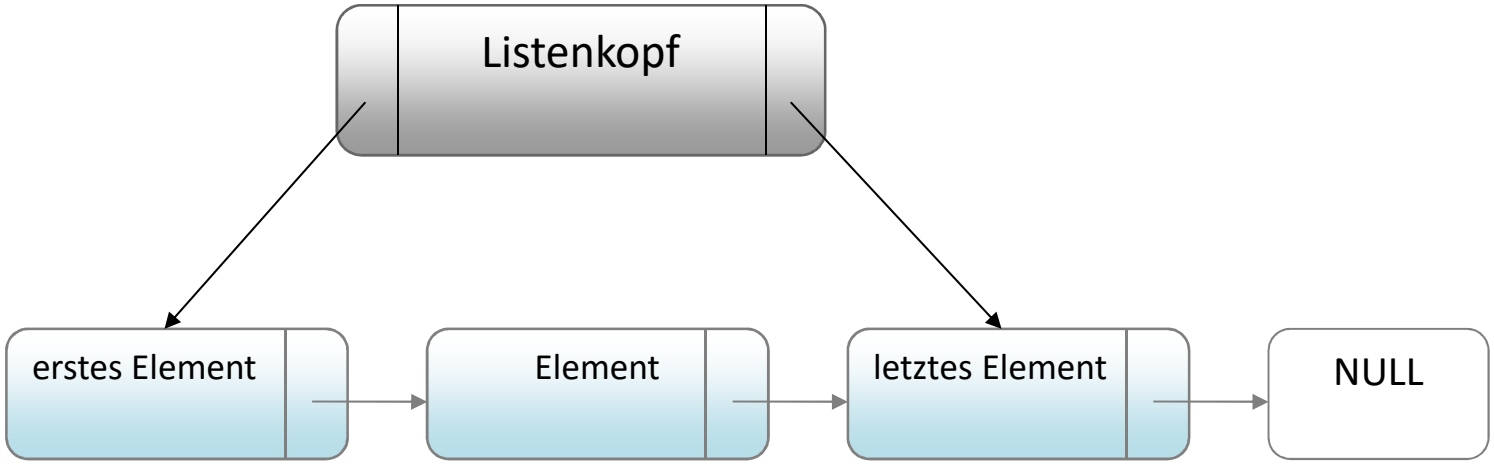


Einfach verkettete Liste
Ausgabe

Liste ausgeben

```
...
// Liste in umgekehrter Reihenfolge ausgeben
printf(".....umgekehrt:\n");

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

Beispiel: ADT Liste (1)



Beispiel: ADT Liste (2)

- Struktur Listenelement

```
#define LAENGE 80

typedef struct s_element
{
  char inhalt[LAENGE];
  struct s_element *next;
} t_element;
```

Beispiel: ADT Liste (3)

- Struktur Listenkopf

```
typedef struct
{
   t_element *erstesElement;
   t_element *letztesElement;
   int anzahlElemente;
} t_Listenkopf;
```

Beispiel: ADT Liste – Init

Operation Liste initialisieren

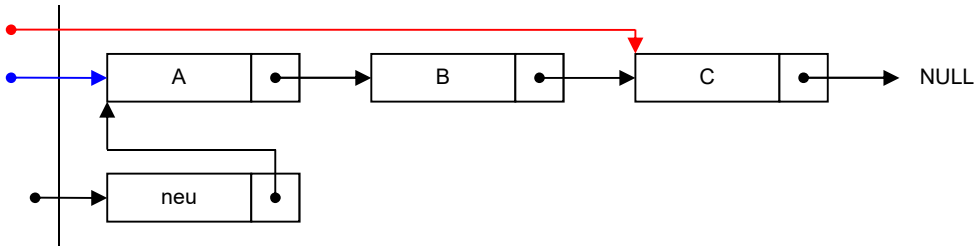
- `void initListe(t_Listenkopf *li)`
  - Zeiger `erstesElement` und `letztesElement` verweisen auf den Nullzeiger
  - Anzahl der Listenelemente ist 0

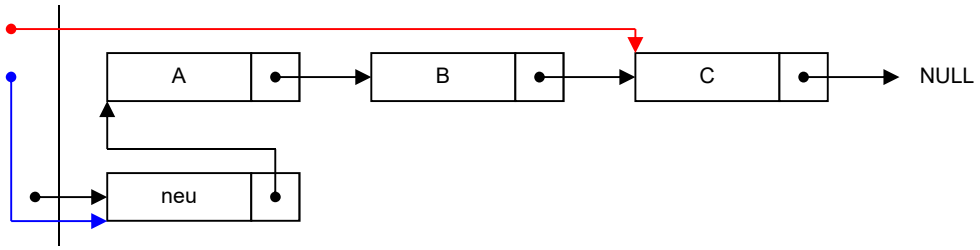
Beispiel: ADT Liste – pushFront

Operation vorne einfügen (pushFront)

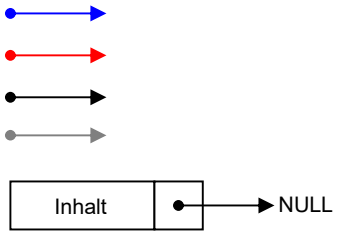
- `void pushFront(t_Listenkopf *li, char *s)`
  - Dynamisch Speicher für das neue Element anfordern
  - Prüfen, ob Speicheranforderung erfolgreich
    - Nein: entsprechende Fehlermeldung
  - Neuen Speicherblock füllen mit
    - `char *s` in erste Strukturkomponente
    - Zeiger `next` zeigt auf `NULL`
  - Element vorne in Liste einfügen
    - Neues Element zeigt auf bisher erstes Element
    - Aktualisierung Listenkopf (`erstesElement`, `letztesElement` und `anzahlElemente`)

Beispiel: ADT Liste – pushFront





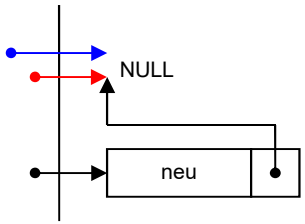
Symbolerläuterung (Skizzen von Sebastian Keller)

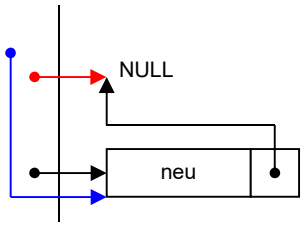


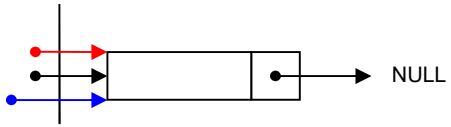
Zeiger auf erstes Element
Zeiger auf letztes Element
Hilfszeiger p
Hilfszeiger q

Ein Element mit Inhalt und dem next-Zeiger auf NULL

wenn Liste bisher leer:



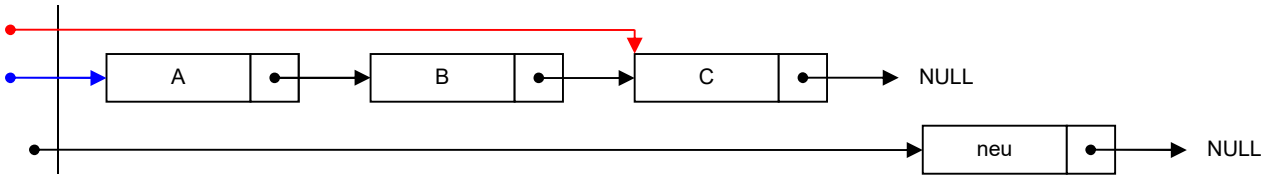


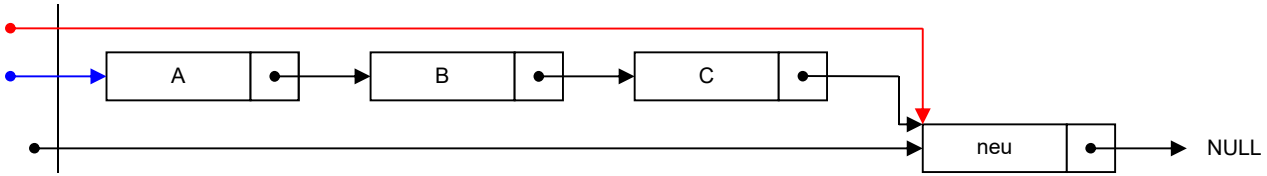


Beispiel: ADT Liste – pushBack

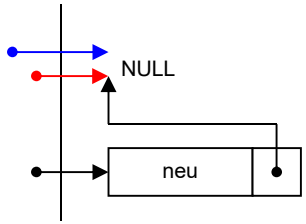
- Operation hinten einfügen (pushBack)
- `void pushBack(t_Listenkopf *li, char *s)`
  - Überprüfen ob Liste leer (=> identisch zu `pushFront`)
  - Dynamisch Speicher für das neue Element anfordern
  - Prüfen, ob Speicheranforderung erfolgreich
    - Nein: entsprechende Fehlermeldung
  - Neuen Speicherblock füllen mit
    - `char *s` in erste Strukturkomponente
    - Zeiger `next` zeigt auf `NULL`
  - Element hinten in Liste einfügen
    - Bisher letztes Element zeigt auf neues Element
    - Aktualisierung Listenkopf (`letztesElement` und `anzahlElemente`)

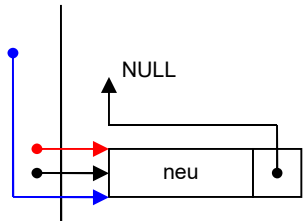
Beispiel: ADT Liste – pushBack



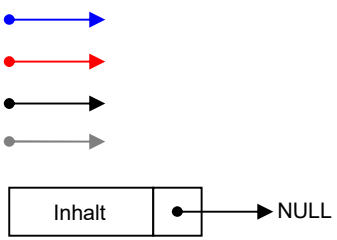


wenn Liste bisher leer:





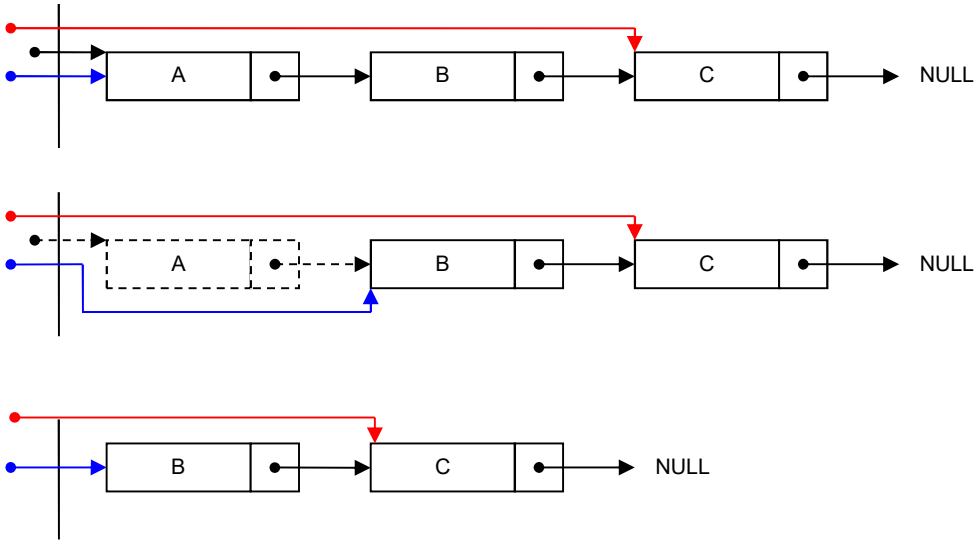
Symbolerläuterung (Skizzen von Sebastian Keller)



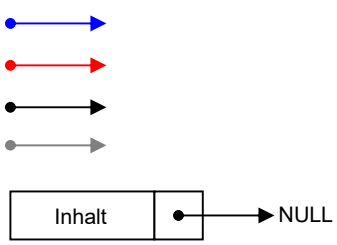
Zeiger auf erstes Element
Zeiger auf letztes Element
Hilfszeiger p
Hilfszeiger q

Ein Element mit Inhalt und dem next-Zeiger auf NULL

Beispiel: ADT Liste – popFront



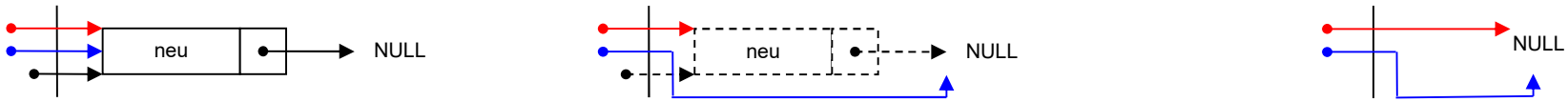
Symbolerläuterung (Skizzen von Sebastian Keller)



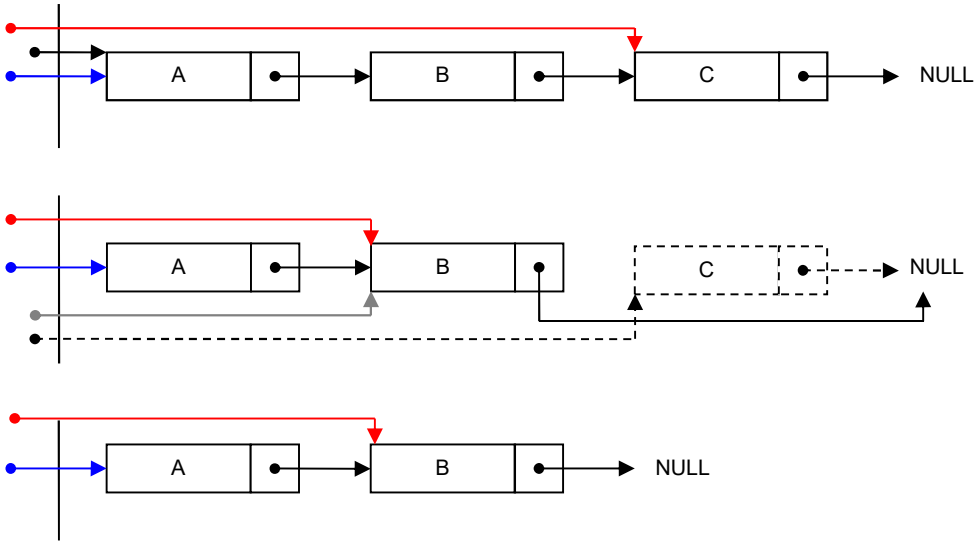
Zeiger auf erstes Element
Zeiger auf letztes Element
Hilfszeiger p
Hilfszeiger q

Ein Element mit Inhalt und dem next-Zeiger auf NULL

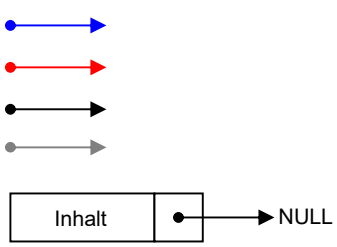
wenn nur ein Element:



Beispiel: ADT Liste – popBack



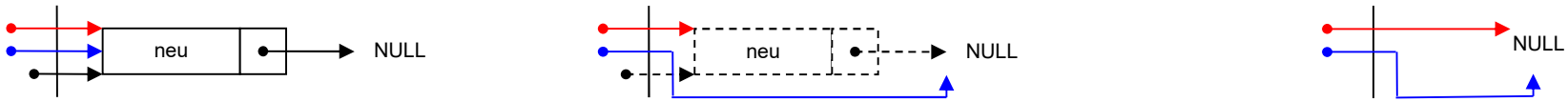
Symbolerläuterung (Skizzen von Sebastian Keller)



Zeiger auf erstes Element
Zeiger auf letztes Element
Hilfszeiger p
Hilfszeiger q

Ein Element mit Inhalt und dem next-Zeiger auf NULL

wenn nur ein Element: wie popFront



Beispiel: ADT Liste – Ausgabe

- Operation Liste ausgeben (printListe)
- `void printListe(t_Listenkopf *li)`
  - Durchhangeln durch die Liste und solange den Inhalt ausgeben bis man auf ein Element stößt, dass auf den Nullzeiger verweist

Stack/Queue

- Stack (Keller)
  - arbeitet nach LIFO-Prinzip (last in first out)
  - Operationen
    - pushFront() / popFront() oder alternativ
    - pushBack() / popBack()

- Queue (Warteschlange)
  - arbeitet nach FIFO-Prinzip (first in first out)
  - Operationen
    - pushBack() / popFront() oder alternativ
    - pushFront() / popBack()

Weitere Operationen

- Teilen
- Mischen / Vereinigen
- Sortieren
- Kopieren
- Umdrehen
- sortiert einfügen (Sortieren beim Einfügen)
- löschen

Aufgabe

Gegeben:

```
typedef struct
{
  // … enthält Daten einer Adresse
} t_Adresse;

typedef struct s_element
{
  t_Adresse daten;
  struct s_element *next;
} t_element;
```

```
typedef struct
{
  t_element *erstesElement;
  t_element *letztesElement;
  int anzahlElemente;
} t_Listenkopf;
```

Schreiben Sie eine Funktion, die zwei Listen zu einer einzigen vereint, indem die beiden Listen aneinandergehängt werden.
Der Prototyp der Funktion sieht wie folgt aus:

```
void listMerge(t_Listenkopf *l1, t_Listenkopf *l2);
```

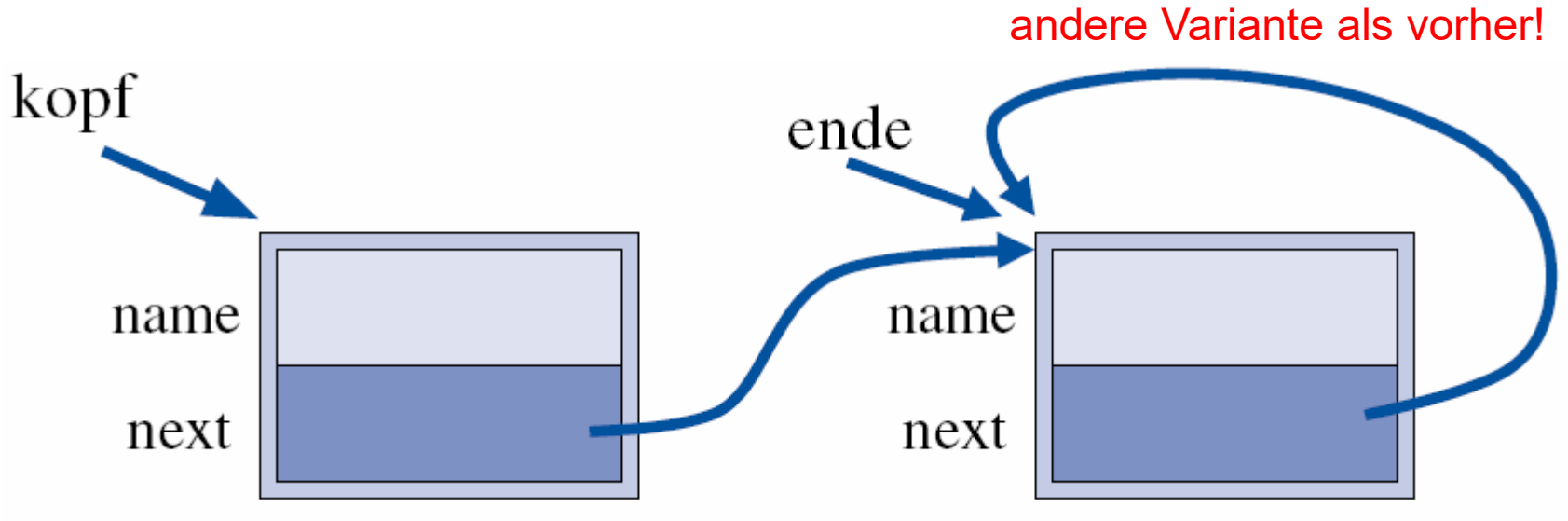
Am Ende steht in `l1` das Ergebnis der Vereinigung, `l2` ist leer.

Einfach verkettete Liste
Sortierte Liste

- Beispiel: Sortierte Liste und Operationen
- Erläuterungen C-Programm:
  1. Zwei Pseudoknoten: `kopf` und `ende`
  2. Vier Funktionen:
     ```
     liste_initialisieren()
     liste_ausgeben()
     element_einfuegen()
     element_loeschen()
     ```

Einfach verkettete Liste
Sortierte Liste – Init (1)

- Initialisieren einer Liste



Einfach verkettete Liste
Sortierte Liste – Init (2)

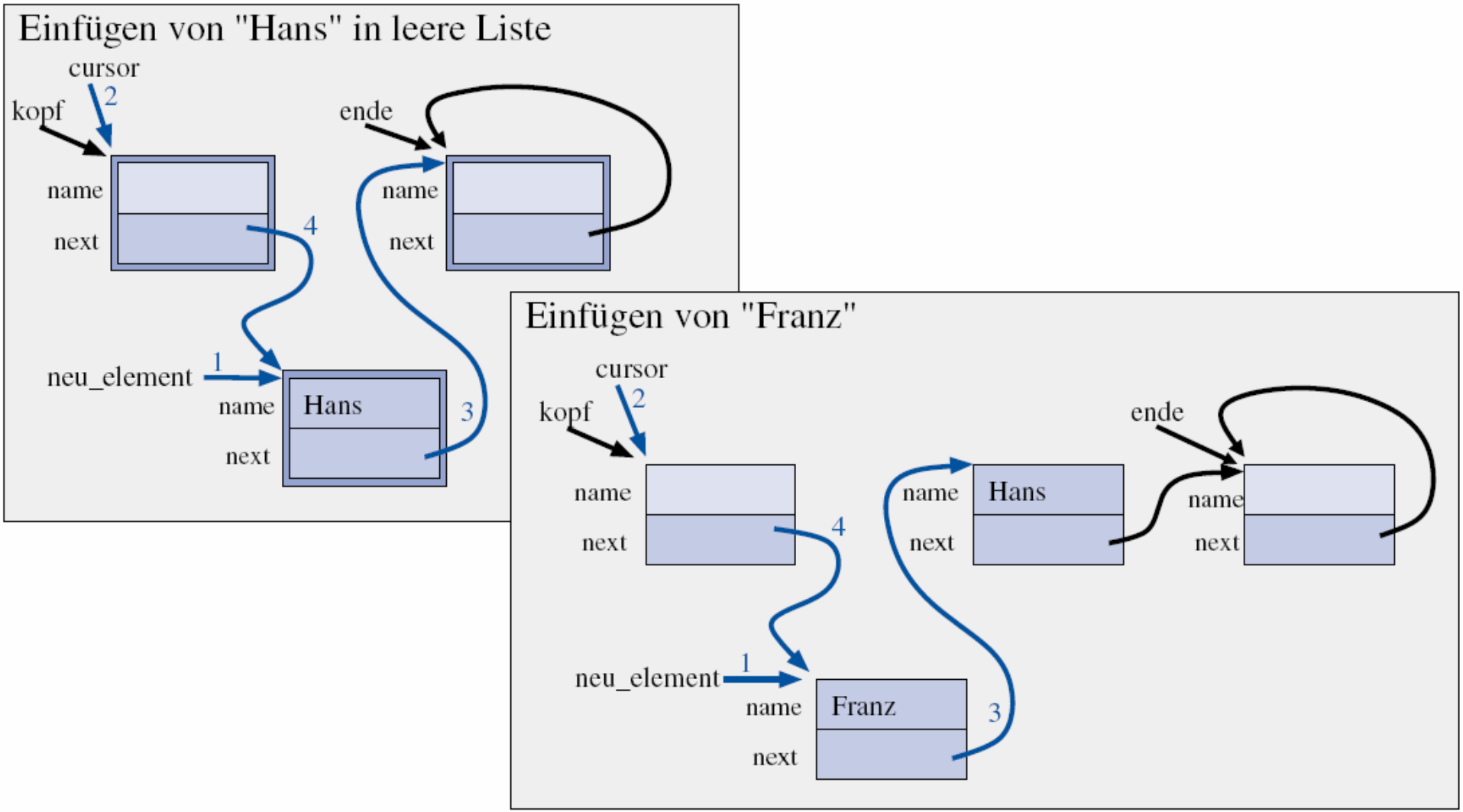
```
void liste_initialisieren(struct element_s **kopf,
                          struct element_s **ende)
{
  *kopf = (struct element_s*)
          malloc(sizeof(struct element_s));
  *ende = (struct element_s*)
          malloc(sizeof(struct element_s));

  if (*kopf == NULL || *ende == NULL)
  {
    printf("...Speicherplatzmangel\n");
    exit(1);
  }

  (*kopf)->next = (*ende)->next = *ende;
}
```

Einfach verkettete Liste
Sortierte Liste – Ausgabe

Hochschule **Rosenheim**
University of Applied Sciences

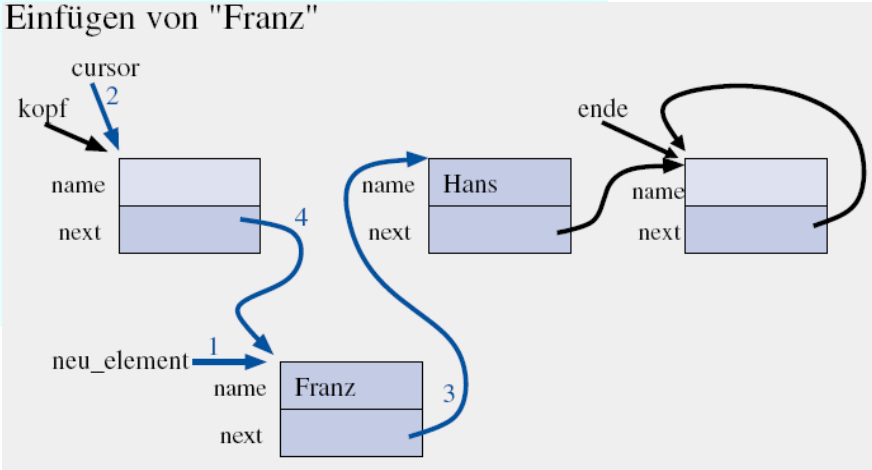
```
void liste_ausgeben(struct element_s *kopf)
{
  struct element_s  *cursor = kopf->next;
  printf("....Liste.....\n");

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

  printf("....Listenende.....\n\n");
}
```

Einfach verkettete Liste
Sortierte Liste – Einfügen (1)



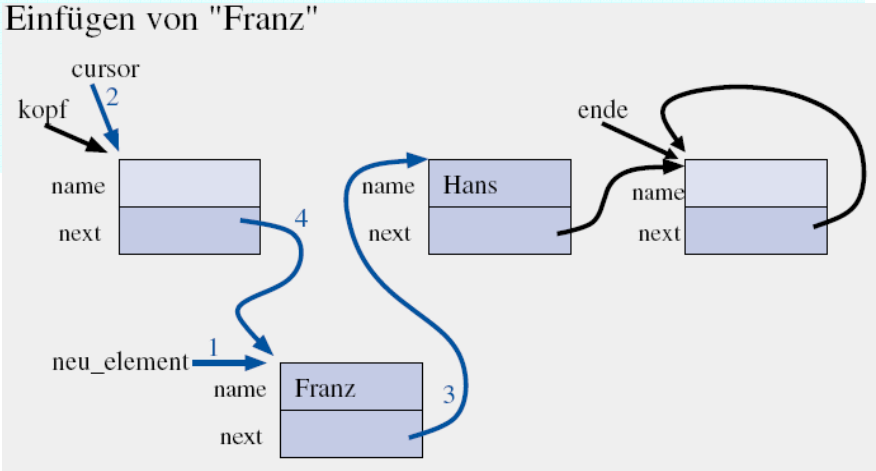
Einfach verkettete Liste
Sortierte Liste – Einfügen (2)


```
void element_einfuegen(struct element_s *kopf, char name[])
{
  struct element_s *cursor = kopf;
  struct element_s *neu_element = NULL;
  Boolean_t ende = FALSE;

  while (!ende)
  {
    if(cursor->next == cursor->next->next)
      ende = TRUE;
    else if(strcmp(name, cursor->next->name) > 0)
      ende = TRUE;
    else
      cursor = cursor->next;
  }

  // weiter auf nächster Folie
```



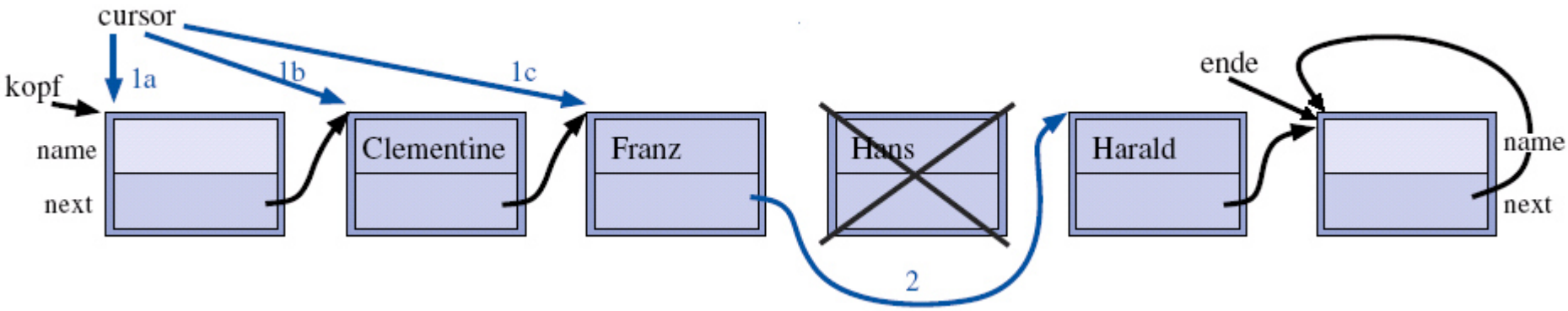
Einfach verkettete Liste
Sortierte Liste – Einfügen (3)


```
  // Dynamisch Speicherplatz für Struktur element_s anfordern
  neu_element = (struct element_s*)malloc(sizeof(struct element_s));

  if (neu_element == NULL)
  {
    printf("Speicherplatzmangel\n");
    exit(1);
  }

  // Namen in angeforderten Speicherbereich ablegen
  strcpy(neu_element->name, name);
  neu_element->next = cursor->next;
  cursor->next = neu_element;
}
```



Einfach verkettete Liste
Sortierte Liste – Löschen (1)

- Löschen des Listenelements „Hans“



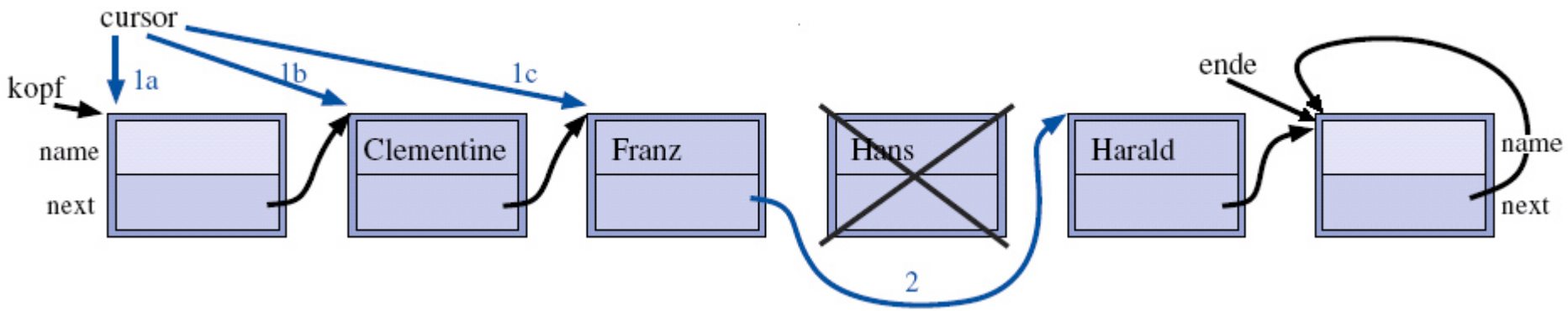
Einfach verkettete Liste
Sortierte Liste – Löschen (2)

Hochschule **Rosenheim**
University of Applied Sciences

```
void element_loeschen(struct element_s *kopf, char name[])
{
  struct element_s  *cursor = kopf;
  struct element_s  *tmp = NULL;
  Boolean_t ende = FALSE;

  while (!ende)
  {
    if(cursor->next == cursor->next->next)
      ende = TRUE;
    else if(strcmp(name, cursor->next->name) == 0)
      ende = TRUE;
    else
      cursor = cursor->next;
  }

 // weiter auf nächster Folie
```

Einfach verkettete Liste
Sortierte Liste – Löschen (3)


```
  if(cursor->next != cursor->next->next)
  {
    // Lösche Element cursor->next
    tmp = cursor->next;
    cursor->next = cursor->next->next;
    free(tmp);
  }
  else
    printf("Element nicht in Liste enthalten!\n");
}
```



Weitere Arten von Listen

- doppelt verkettete Listen
  - enthält zusätzlich Zeiger auf das vorherige Element
  - Vorteile:
    - man kann damit in beiden Richtungen durch die Liste laufen
    - Löschen/Einfügen einfacher
  - Nachteil:
    - höherer Speicherverbrauch

- Ringlisten/Ringbuffer
  - feste Größe (Anzahl Elemente)
  - wenn Buffer voll, werden die ältesten Daten überschrieben

Aufgabe – doppelt verkettete Liste

Gegeben:

```
typedef struct
{
  // … enthält Daten einer Adresse
} t_Adresse;

typedef struct s_element
{
  t_Adresse daten;
  struct s_element *next;
  struct s_element *prev;
} t_element;
```

```
typedef struct
{
  t_element *erstesElement;
  t_element *letztesElement;
  int anzahlElemente;
} t_Listenkopf;
```

Schreiben Sie eine Funktion, die einen Zeiger auf Element n einer Liste von hinten gezählt (letztes Element = 0) zurückliefert, wenn dieses existiert.
Die Listenenden sind durch Nullzeiger gekennzeichnet.
Rückgabewert:

- Zeiger, wenn erfolgreich
- NULL wenn Element nicht existiert

Der Prototyp der Funktion sieht wie folgt aus:

```
t_element* ElementZeiger(t_Listenkopf *l, int n);
```

Zusammenfassung

- Datenstruktur
  - Datentyp + Operationen
- Listen
  - einfach/doppelt verkettet
  - Ringlisten
  - sortierte Listen
- Elemente
  - werden typischerweise dynamisch angefordert
  - liegen nicht am Stück im Speicher
  - enthalten Zeiger auf Nachfolger und evtl. Vorgänger
- Typische Operationen
  - einfügen, löschen, sortieren, vereinigen, kopieren, …
- auf ähnliche Art können weitere Datenstrukturen aufgebaut werden
  - z.B. Bäume