Listen in C

Einführung in die Programmierung

Michael Felderer

Institut für Informatik, Universität Innsbruck

Datenstrukturen

- Neben einfachen Datentypen benötigt man in der Informatik komplexere zusammengesetzte Typen.
  - Ähnlich wie in der Mathematik Mengen, Vektoren, Matrizen etc. verwendet werden.
  - Moderne Programmiersprachen bieten entsprechende Möglichkeiten für die Konstruktion von komplexen Datenstrukturen an.
- Man benutzt in C für die Konstruktion komplexer Datenstrukturen
  - Arrays
  - Strukturen
  - Zeigerstrukturen (Verknüpfung beliebiger Datenelemente mit Hilfe von Zeigern)

Verkettete Listen

- Eine verkettete Liste ist eine Folge von Elementen, die **dynamisch** während des Programmablaufs verlängert bzw. verkürzt werden kann.
  - Eine verkettete Liste kann nach Bedarf **zur Laufzeit** wachsen und schrumpfen.
  - Anders als bei einem Array ist dabei aber **nicht** garantiert, dass die Elemente hintereinander im Speicher liegen.
- Für die einzelnen Elemente einer Liste wird eine Struktur verwendet.
  - In dieser Struktur wird wiederum auf ein Struktur-Element verwiesen.
  - Um eine Verbindung zwischen den Elementen herzustellen, werden Zeiger verwendet.

Verkettete Liste – Variante 1 (einfach)

- Einfach verkettete Liste
  - Jedes Element beinhaltet nur einen Integerwert.
- Grundlegende Deklarationen (global):

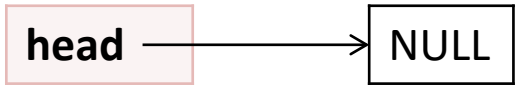
```
struct node {
    int value;
    struct node *next;
};
```

- Mögliche typedefs

```
typedef struct node node_t;
typedef struct node* nodeptr_t;
nodeptr_t head = NULL;
```

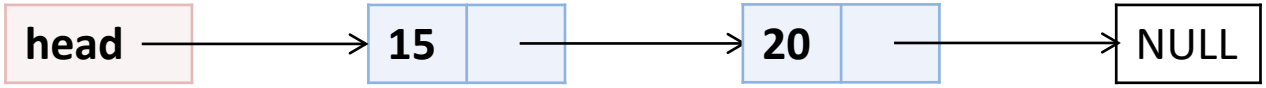
Einfügen – Prinzip

- Ausgangslage (leere Liste)



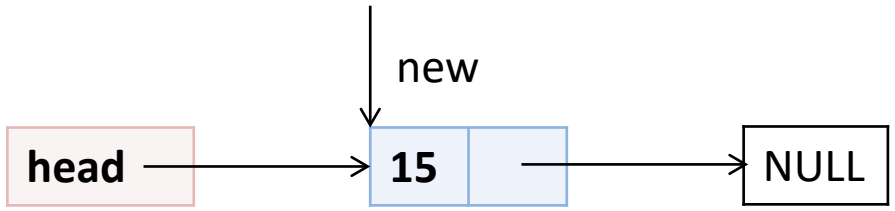
- Einfügen von 15

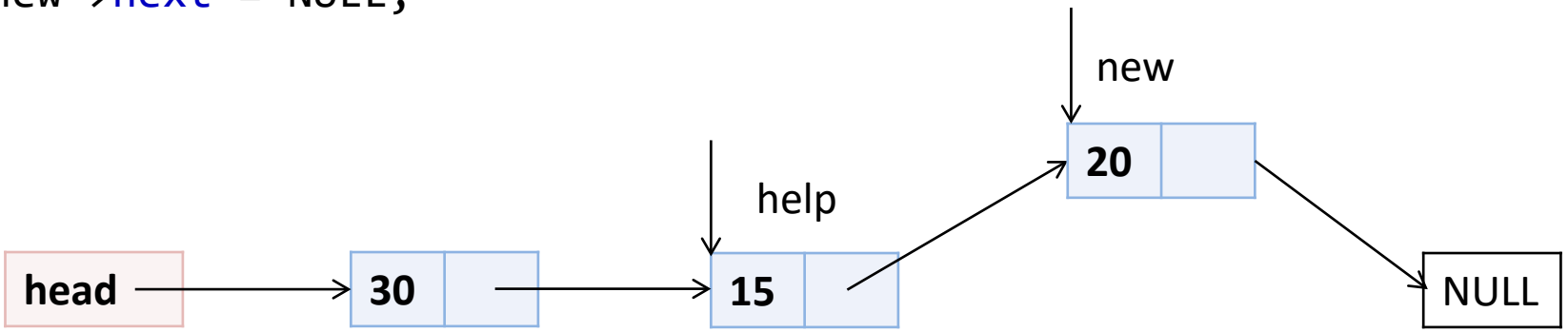
- Einfügen von 20 (am Ende der Liste)



Einfügen – Implementierung

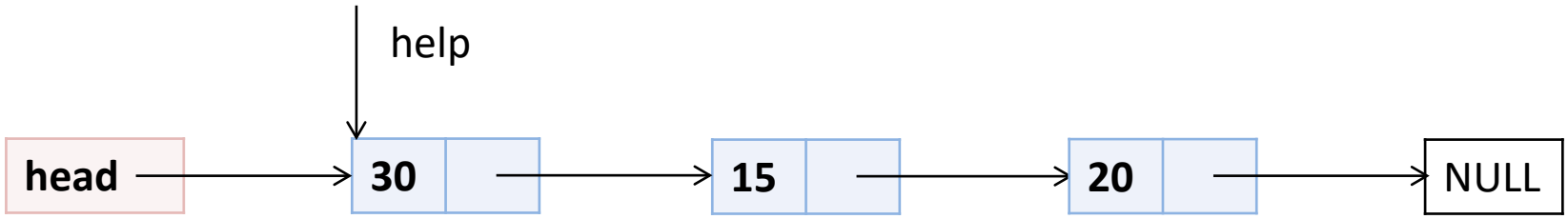
```
void insert_node(nodeptr_t new) {
    if (head == NULL) {
        head = new;
        new->next = NULL;
    } else {
        nodeptr_t help = head;
        while (help->next != NULL) {
            help = help->next;
        }
        help->next = new;
        new->next = NULL;
    }
}
```





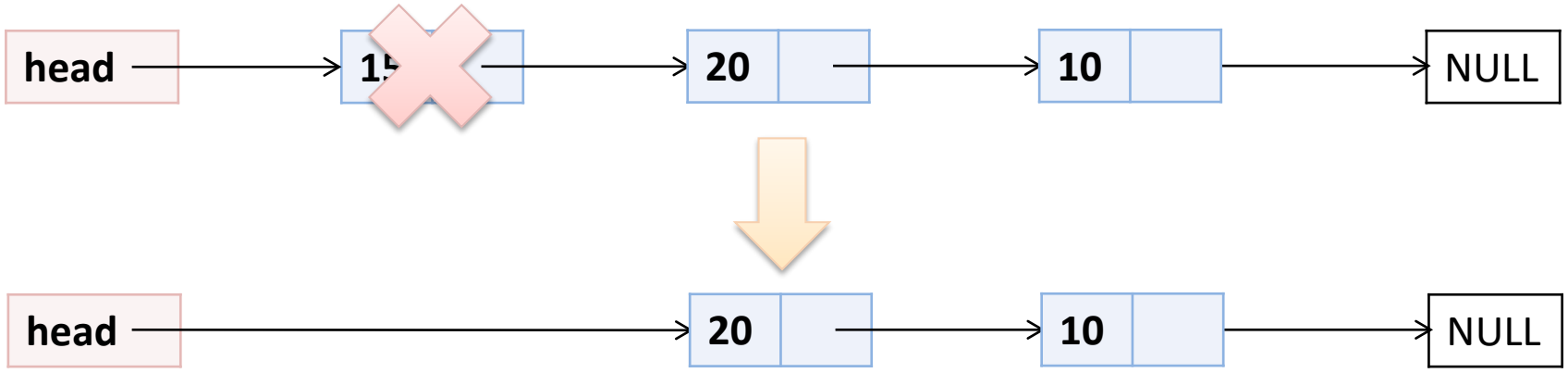
Ausgeben aller Knoten

```
void list_nodes(void) {
    nodeptr_t help = head;
    while (help != NULL) {
        printf("%d\n", help->value);
        help = help->next;
    }
}
```

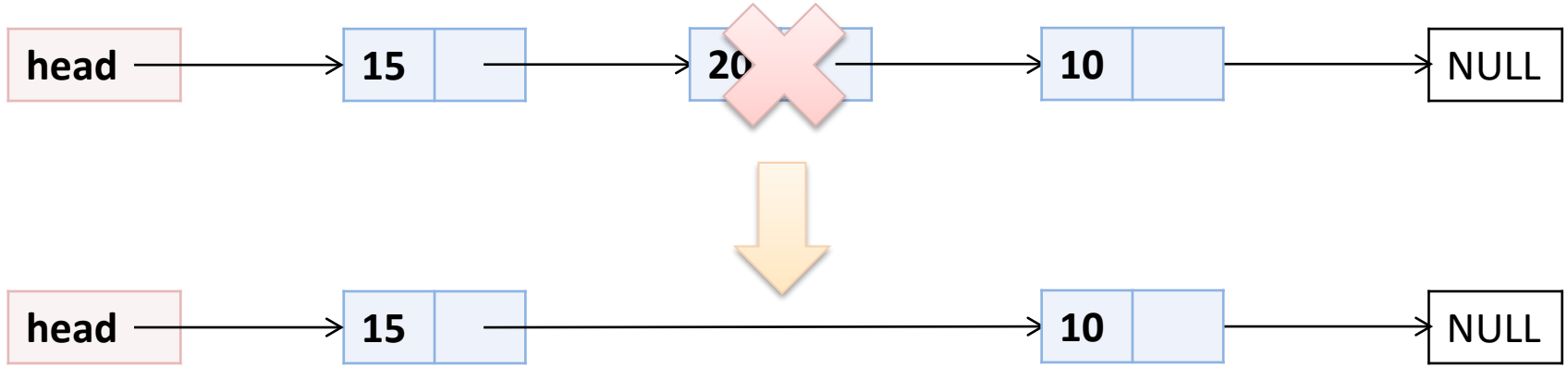


Löschen – Prinzip

- Hier müssen zwei Fälle unterschieden werden
  - 1: Ersten Knoten in der Liste löschen (ein Hilfszeiger notwendig)



  - 2: Beliebigen anderen Knoten in der Liste löschen (zwei Hilfszeiger notwendig)

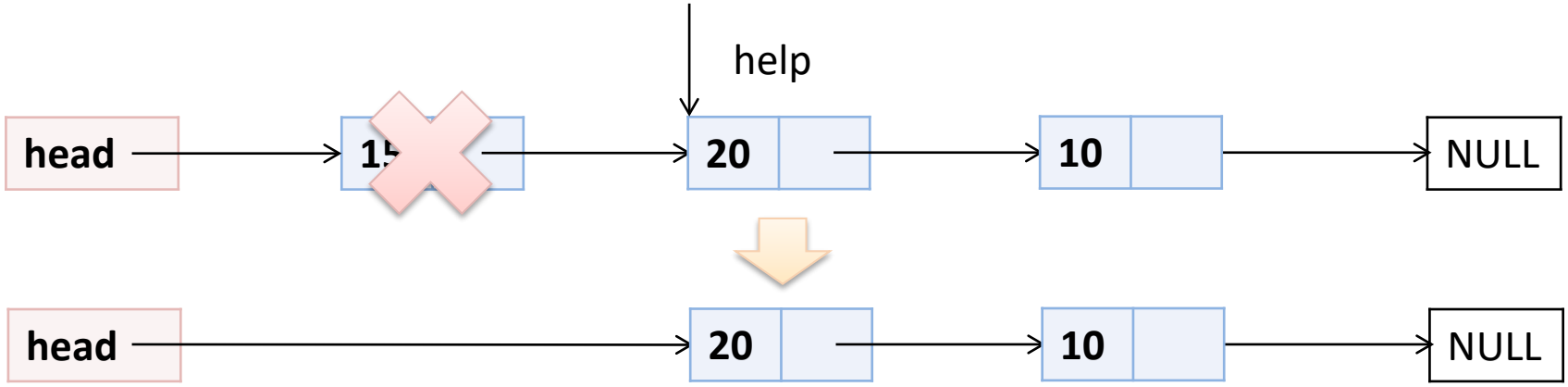


Löschen – Implementierung

```
void delete_node(int val) {
     if (head != NULL) {
          if (head->value == val) {
               nodeptr_t help = head->next;
               free(head);
               head = help;
          } else {
               nodeptr_t help1 = head;
               while (help1->next != NULL) {
                    nodeptr_t help2 = help1->next;
                    if (help2->value == val) {
                         help1->next = help2->next;
                         free(help2);
                         break;
                    }
                    help1 = help2;
               }
          }
     }
}
```

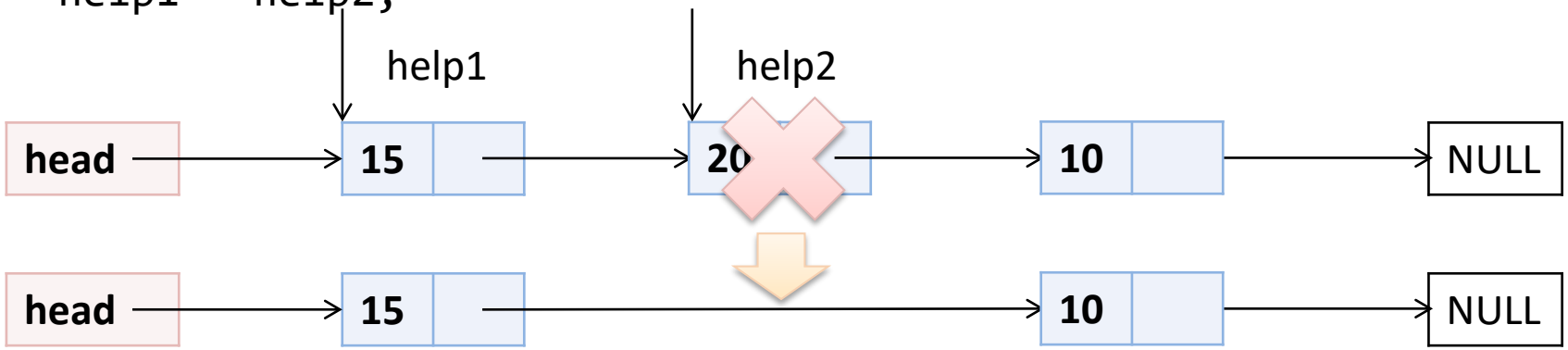
Löschen – 1. Fall

```
…
if (head->value == val) {
    nodeptr_t help = head->next;
    free(head);
    head = help;
} else {
…
```



Löschen – 2. Fall

```
…
} else {
    nodeptr_t help1 = head;
    while (help1->next != NULL) {
        nodeptr_t help2 = help1->next;
        if (help2->value == val) {
            help1->next = help2->next;
            free(help2);
            break;
        }
        help1 = help2;
    }
}
…
```



Vor- und Nachteile verketteter Listen

- Vorteile
  - Einfügen bzw. Entfernen eines Elements in der Liste ist viel weniger speicheraufwändig als bei Arrays.
  - Die Anzahl der Elemente und der dafür benötigte Speicherplatz muss nicht im Voraus angegeben werden.
    - Spätere Erweiterung ist viel einfacher.
    - Es ist kein aufwändiges Umkopieren (Verschieben) in andere zusammenhängende Speicher notwendig.
- Nachteile
  - Kein Direktzugriff auf beliebiges Element!
    - Man muss die Liste ab einem bestimmtem Punkt durchlaufen, bis man zu dem gewünschtem Element gelangt (Zeitaufwand!).
  - Zeiger- bzw. Referenzen belegen zusätzlichen Speicherplatz.

Verkettete Liste – Variante 2 (mit Container)

- Grundlegende Datenstruktur im Beispiel
  - Die gesamte Liste wird über den Listencontainer angesprochen.
  - Vereinfacht einige Überprüfungen.
  - Container kann zusätzliche Informationen (z.B. Länge der Liste) beinhalten.

```c
/* Listenelement */
struct ielem {
        int val;
        struct ielem *next;
};

/* Listencontainer */
struct ilist {
        int count;
        struct ielem *first;
};
```

Verkettete Liste – Variante 2 (Modularisierung)

- Für eine allgemeinere Verwendung wird die Implementierung in ein Modul gegeben.
  - Header-Datei mit Datenstruktur und Funktionsprototypen
  - Implementierungsdatei mit eigentlicher Implementierung
- Implementierung gegenüber Variante 1 leicht verändert!

Header

```c
#ifndef _ILIST_H
#define _ILIST_H

/* List element */
struct ielem {
	int val;
	struct ielem *next;
};

/* List container */
struct ilist {
	int count; // size(first)
	struct ielem *first;
};

/* Initializes the new list */
struct ilist *ilist_init(void);

/* Adds a new element at position pos or at the end of the list */
void insert_node(struct ilist *list, int pos, int val);

/* Adds a variable number of elements at the end of the list */
void add_nodes(struct ilist *list, int num, ...);

/* Removes an element from position pos and does not change anything if the
 * element is not found in the list*/
void delete_node(struct ilist *list, int pos);

/* Frees the memory allocated by the list */
void ilist_free(struct ilist *list);

/* Prints the content of the list */
void list_nodes(struct ilist *list);

#endif
```

Initialisierung der Liste

```
struct ilist *ilist_init(void) {
    struct ilist *new = malloc(sizeof(struct ilist));
    if (new == NULL) {
        fprintf(stderr, "out of memory");
        exit(EXIT_FAILURE);
    }
    new->count = 0;
    new->first = NULL;
    return new;
}
```

Einfügen (mit Position, mit Wert)

```
void insert_node(struct ilist *list, int pos, int val) {
     struct ielem *tmp = list->first, *last = NULL, *new;
     int i = 0;

     new = malloc(sizeof(struct ielem));
     if (new == NULL) {
          fprintf(stderr, "out of memory");
          exit(EXIT_FAILURE);
     }
     new->val = val;
     new->next = NULL;

     while (tmp && i++ < pos) {
          last = tmp;
          tmp = tmp->next;
     }
     new->next = tmp;
     if (last)
          last->next = new;
     else
          list->first = new;
     list->count++;
}
```

Variable Argumentlisten

- In C dürfen Funktionen mit variabel langen Argumentlisten aufgerufen werden.
- Damit man eigene Funktionen mit einer variablen Argumentliste schreiben kann, sind in der Headerdatei `stdarg.h` folgender Datentyp und folgende Makros deklariert:

Typ/Makro	Syntax	Beschreibung
va_list	*va_list argPtr;*	Abstrakter Datentyp (wird auch als Argumentzeiger bezeichnet), mit dem die Liste der Parameter definiert wird und mit dem der Zugriff auf die optionalen Argumente realisiert wird.
va_start	**void va_start(va_list argPtr, lastarg);**	Argumentliste initialisiert den Argumentzeiger `argPtr` mit der Position des ersten optionalen Arguments. An `lastarg` muss der letzte fixe Parameter in der Liste übergeben werden.
va_arg	**type va_arg(va_list argPtr, type);**	Gibt das optionale Argument zurück, auf das `argPtr` im Augenblick verweist, und setzt den Argumentzeiger auf das nächste Argument. Mit `type` gibt man den Typ des zu lesenden Arguments an.
va_end	**void va_end(va_list argPtr);**	Hiermit kann man den Argumentzeiger `argPtr` beenden, wenn man diesen nicht mehr benötigt.

Einfügen einer variablen Anzahl von Knoten

```
void add_nodes(struct ilist *list, int num, ...) {
   va_list zeiger;
   va_start(zeiger,num);
   while (num-- > 0)
      insert_node(list, list->count, va_arg(zeiger,int));
   va_end(zeiger);
}
```

Variante mit Zähler (wird in num übergeben).
2. Variante wäre ein bestimmtes Element (z.B. 0) als Abbruchbedingung zu benutzen.

Löschen (mit Position)

```
void delete_node(struct ilist *list, int pos) {
    struct ielem *tmp = list->first, *last = NULL;
    int i = 0;
    while (tmp && i++ < pos) {
        last = tmp;
        tmp = tmp->next;
    }
    if (!tmp)
        return;
    if (last)
        last->next = tmp->next;
    else
        list->first = list->first->next;
    free(tmp);
    list->count--;
}
```

Ausgeben und Freigeben der Liste

```
void list_nodes(struct ilist *list) {
    struct ielem *tmp = list->first;
    printf("size: %d [ ", list->count);
    while (tmp != NULL) {
        printf("%d ", tmp->val);
        tmp = tmp->next;
    }
    printf("]\n");
}

void ilist_free(struct ilist *list) {
    struct ielem *l = list->first, *tmp;
    while (l) {
        tmp = l;
        l = l->next;
        free(tmp);
    }
    free(list);
}
```

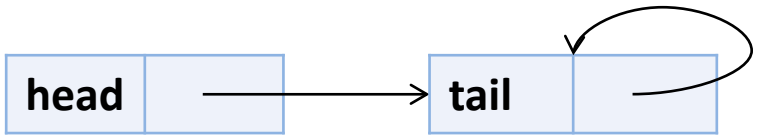
Variante 3 – Erweiterung um einen tail-Zeiger

- Erweiterung der Implementierung
  - Einen Zeiger auf das letzte Element anlegen.
  - Hat Vorteile, aber erzeugt mehr Verwaltungsaufwand.
- Nachfolgend
  - Spezielle tail-Zeiger-Variante
  - Ermöglicht nicht den direkten Zugriff auf das letzte Element!
  - Verkürzt aber einige Funktionen (Überprüfungen)!

tail-Zeiger (Beispiel – Deklaration und Initialisierung)

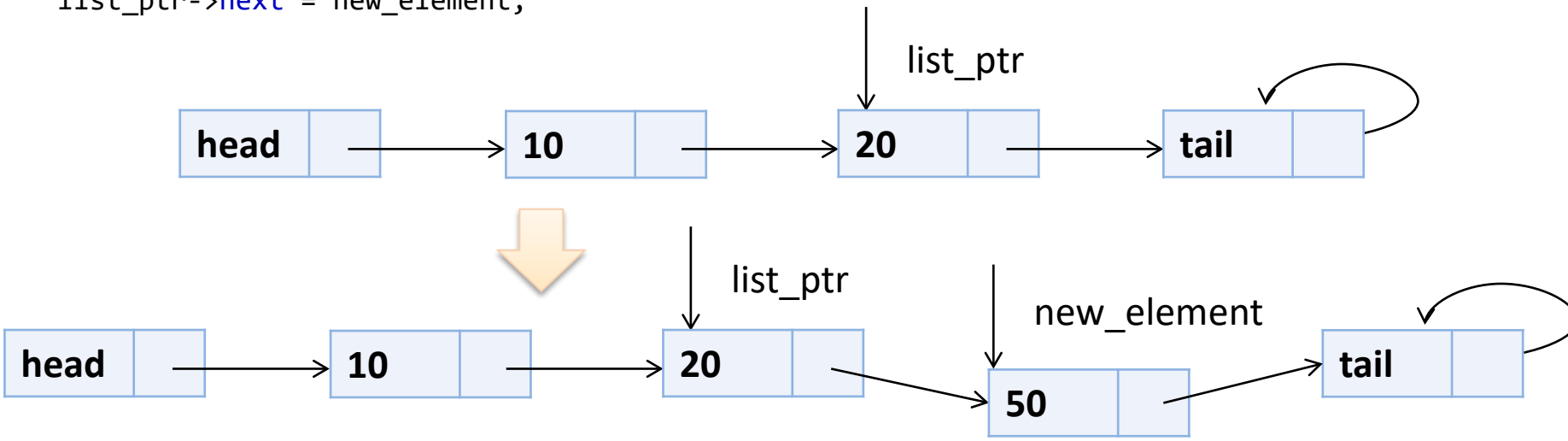
```
struct list_element {
    int val;
    struct list_element *next;
};
struct list_element *head, *tail;


void list_init(void) {
    head = malloc(sizeof(struct list_element));
    tail = malloc(sizeof(struct list_element));
    if (head == NULL || tail == NULL) {
        printf(".......Out of memory\n");
        exit(1);
    }
    head->next = tail->next = tail;
}
```



tail-Zeiger (Beispiel – Einfügen)

```
void insert_node(int elem) {
     struct list_element *new_element = malloc(sizeof(struct list_element));
     if (new_element == NULL) {
          printf(".......Out of memory!\n");
          exit(1);
     }
     new_element->val = elem;
     struct list_element *list_ptr = head;
     while (list_ptr->next != list_ptr->next->next) {
          list_ptr = list_ptr->next;
     }
     new_element->next = list_ptr->next;
     list_ptr->next = new_element;
}
```



Doppelt verkettete Liste

- In dieser Liste kann man vorwärts und rückwärts navigieren.
- Einfache Deklaration (ohne Listencontainer etc.)

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

Ausblick

- Verkettete Listen sind nur eine Beispielsklasse von dynamischen Datenstrukturen.
- Weitere Beispiele
  - Bäume
  - Hashtabellen
- Mehr darüber gibt es im 2. Semester in der LV Algorithmen und Datenstrukturen ☺