Verkettete Strukturen - Listen

- Sie wissen, was Listen sind
- Sie können Einfüge und Lösch Operationen auf der Liste implementieren
- Sie wissen, wie man über eine Liste iteriert
- Sie kennen die allgemeine Liste

Wieso Listen - Rangliste

- Teilnehmerliste für einen Wettkampf
  - Sie müssen ein Programm für die An- und Abmeldung implementieren
  - Teilnehmer
    - *Anzahl unbekannt*
    - *können sich zu beliebigen Zeitpunkt an- und abmelden*

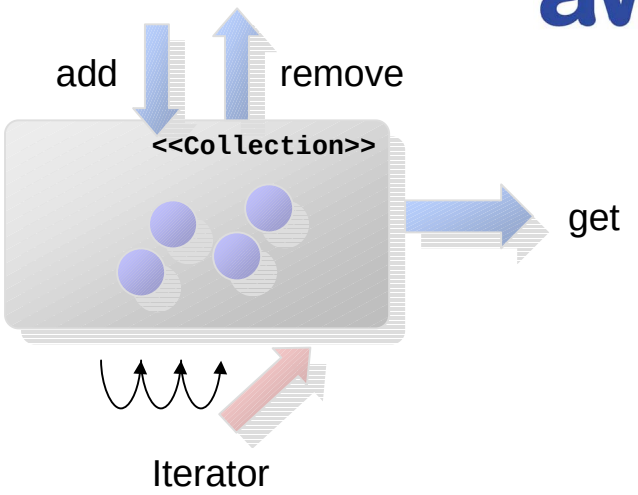
- Wie implementieren?
  - unbekannte Anzahl Teilnehmer anmelden → dynamischer allozierter Array

  aber:
  - Neue (unerwartete) Teilnehmer
    - *Array muss zur Laufzeit vergrössert werden*
  - Abmeldungen/Disqualifikationen hinterlassen "Löcher" im Array

- Verwendung von Listen
  - Bei Listen können beliebig Elemente (als struct) zur Laufzeit hinzugefügt und gelöscht werden

Sammlungen, Arrays vs Listen

- Generelle Sammlungen/Collection
  - Sammlung von Elementen
  - ein gemeinsamer Satz von Funktionen
    - *hinzufügen: add*
    - *löschen: remove*
    - *herausholen: get*
    - *über alle Elemente iterieren*



Sammlungen in C

- vs. Skalar
- Statischer Array
- Dynamischer Array
- List

Beispiel

int a; int* pa = malloc(sizeof(int));

int a[10]

int a[] = malloc(n* sizeof(int)),

lstCreate()

Verkettete Strukturen

- Deklaration besser/lesbarer mittels *typedef*
- für die Pointer innerhalb des Structs muss aber trotzdem die "struct" Version gearbeitet werden (weil Typ noch nicht bekannt)

```
typedef struct Datum {
    int tag;
    int monat;
    int jahr;
    struct Datum  *pNext;
} Datum;
```

- Nachher kann aber das Stuct einfach wiefolgt alloziert werden.

```
Datum* pDatum = (Datum*)malloc(sizeof(Datum));
```

Verkettete Strukturen


```
typedef struct Datum {
    int tag;
    int monat;
    int jahr;
    struct Datum  *pNext;
} Datum;

Datum* mw1= (Datum*)malloc(sizeof(Datum));
Datum* mw2= (Datum*)malloc(sizeof(Datum));
mw1->pNext = mw2;    /* Strukturen verketten */
```

Listen Operationen

- Eine beliebig lange Liste von Structs;
- Beliebige Structs -> T

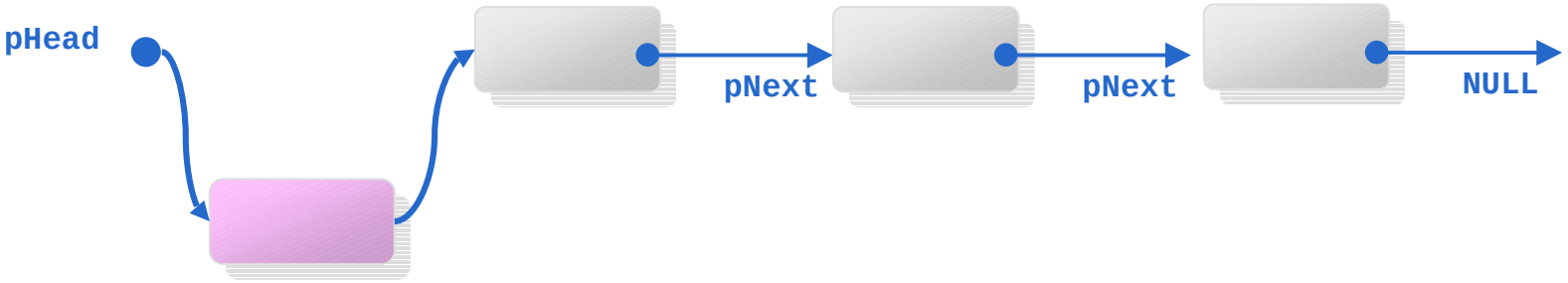
- Liste ist eine der grundlegenden Datenstrukturen in der Informatik.

- Die Liste wird auch oft zur Implementierung anderer Datenstrukturen verwendet (z.B. Queue, Stack).

Minimale Operationen

Funktionskopf	Beschreibung
`void addFirst(T x)`	Füge am Anfang an
`void addLast(T x)`	Hänge am Schluss an
`void add(int pos, T x)`	Fügt x an der pos in die Liste ein
`void addSorted(T x)`	Fügt x geordnet ein
`T get(int pos)`	Gibt Element an pos zurück
`T first()`	Gebe erstes Listenelem
`T next()`	Gebe nächstes Listenelem
`T removeFirst()`	Entferne erstes Element
`T removeLast()`	Entferne letztes Element
`T remove(int pos)`	Entfernt das pos Element und gibt es als Rückgabewert zurück
`void clear()`	Lösche ganze Liste
`int size()`	Gibt Anzahl Element zurück

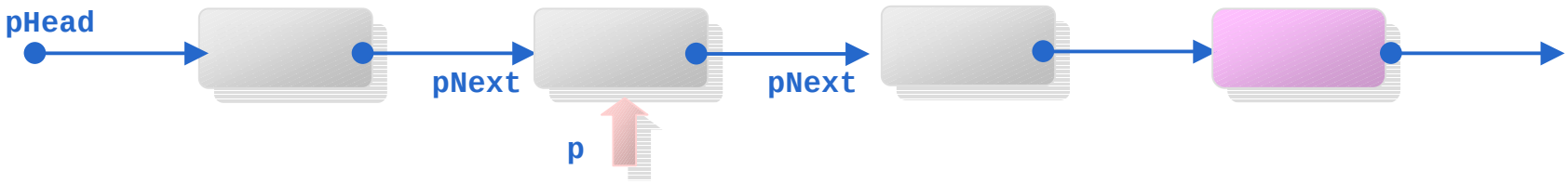
AddFirst - Einfügen am Anfang



- ein neues Element wird am Anfang hinzugefügt

```
Datum* pHead = NULL;
void addFirst(Datum* pNew) {
  // Zeiger von pNext und neues pHead setzen
  pNew->pNext = pHead;
  pHead = pNew;
}
...
Datum* pNew = (Datum*)malloc(sizeof(Datum));
pNew->pNext = NULL;
addFirst(pNew)
```

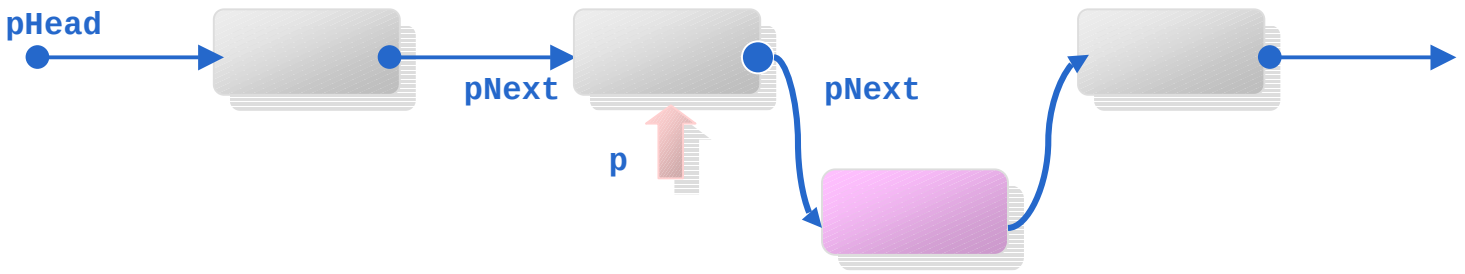
AddLast - Anfügen am Schluss



- ein neues Element wird am Schluss hinzugefügt

```
Datum* pHead = NULL;
void addLast(Datum* pNew) {
  if (pHead == NULL) pHead = pNew;
  else {
    Datum* p = pHead;
    while (p->pNext != NULL) p = p->pNext;
    p->pNext = pNew;
  }
}
...
Datum* pNew = (Datum*)malloc(sizeof(Datum));
pNew->pNext = NULL;
addLast(pNew)
```

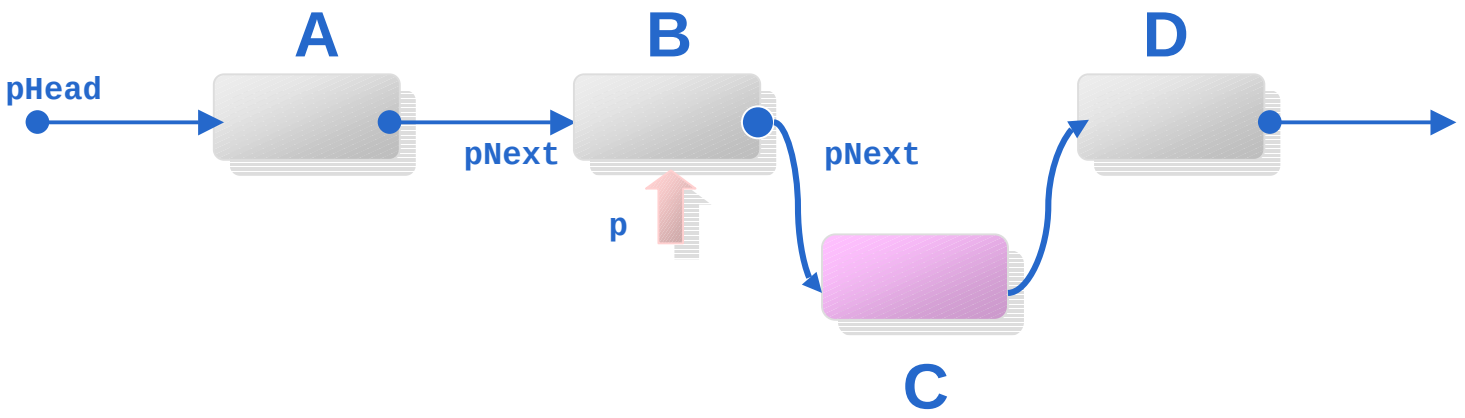
Add - Einfügen an beliebiger Stelle



- ein neues Element wird an bestimmter Position hinzufügen

```
void add(int pos, Datum* pNew) {
  if (pHead == NULL || pos == 0) {
      pNew->pNext = pHead;
      pHead = pNew;
  } else {
    Datum* p = pHead;
    for (;p->pNext != NULL && pos>0;pos--) p = p->pNext;
    pNew->pNext = p->pNext;
    p->pNext = pNew;
  }
}
```

AddSorted - Sortiertes Einfügen



- ein neues Element an **sortiert** richtiger Position hinzufügen

```
void addSorted(Datum* pNew)
   ...
    while (vergleiche(*p, *pNew) < 0) {
         p = p->pNext;
    }
   ...
```

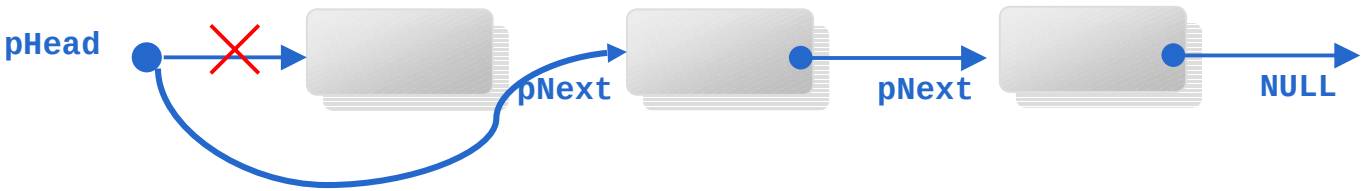
Finde richtige Stelle in Liste

- oder allgemein Datum Vergleich über Comparator Funktion beim Aufruf mitgegeben

```
int datumCompare(Datum* d1, Datum* d2) { .... } // <0 =0 >0
...
addSorted(datum, &vergleiche);
```

Übergabe der Funktion als Parameter

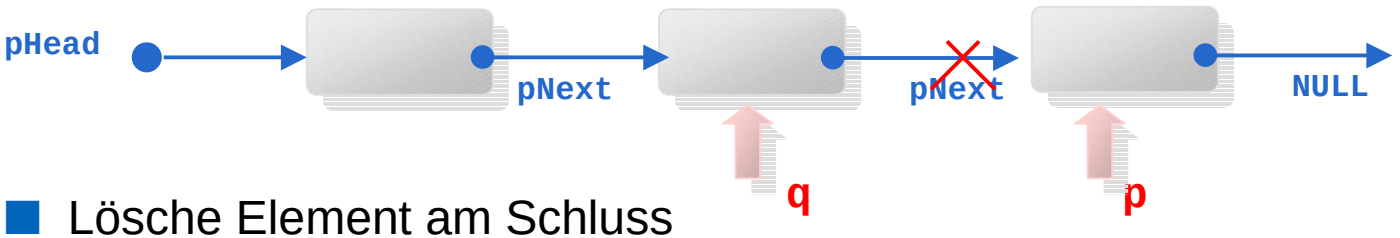
RemoveFirst - Löschen am Anfang



- Lösche Element am Anfang

```
Datum* removeFirst() {
  Datum* p = NULL;
  if (pHead != NULL) {
     p = pHead;
     pHead = pHead->pNext;
  }
  return p;
}
```

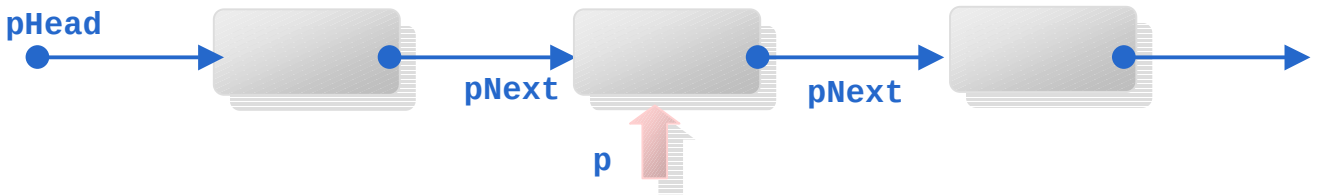
RemoveLast - Löschen am Schluss



- Lösche Element am Schluss

```
Datum* removeLast() {
  Datum* p = NULL;
  if (pHead != NULL) {
     Datum* q = pHead;
     if (pHead->pNext != NULL) {
         while (q->pNext->pNext != NULL) q = q->pNext;
         p = q->pNext;
         q->pNext = NULL;
     } else {
         return removeFirst();
     }
  }
  return p;
}
```

Size - Gebe Anzahl Elemente zurück

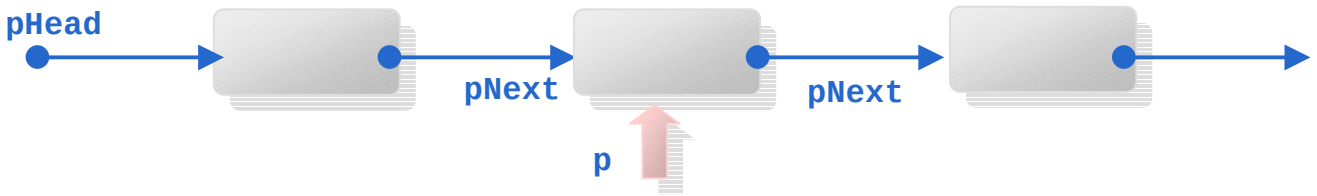


- die Grösse der Liste bestimmen

```
int size() {
  int size = 0;
  Datum* p = pHead;
  while (p != NULL) {
        p = p->pNext;
       size++;
  }
  return size;
}
```

Alternative: size wird bei Einfüge und Löschoperationen nachgeführt

Get - Gebe Element an Position zurück



- das Element an einer bestimmten Postion wird zurückgegeben

```
Datum* get(int pos) {
  Datum* p = pHead;
  for (;pos>0;pos--) p = p->pNext;
  return p;
}
...
Datum* pElement =  get(2);
```

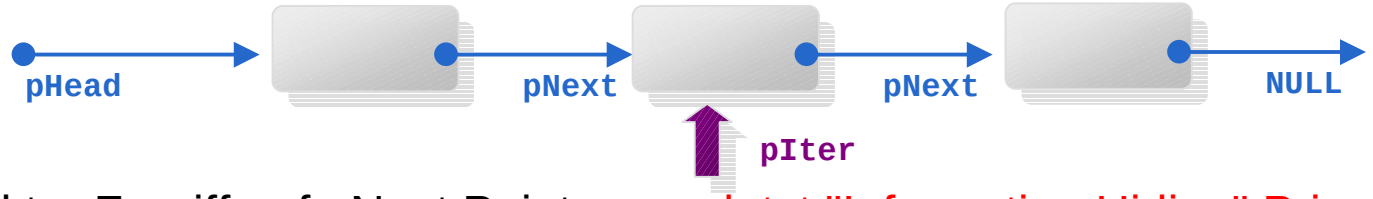
- Über alle Elemente iterieren

```
// durch alle Elemente hindurch iterieren mittels get
for (int i = 0; i < size(); i++) {
   Datum* pElement = get(i);
}
```

get selber muss immer vom Anfang der Liste aus starten -> ineffizient

Iterieren durch alle Elemente

- Diese Listen können einfach traversiert werden



- Direkter Zugriff auf pNext Pointer - verletzt "Information Hiding" Prinzip

```
Datum* pIter = pHead;
while (pIter != NULL) {
   printf("%d:%d:%d\n", pIter->h,pIter->min,pIter->sec);
   p = p -> pNext;
}
```

- besser über (versteckte) iterator-Variable und Funktionen **first, next**

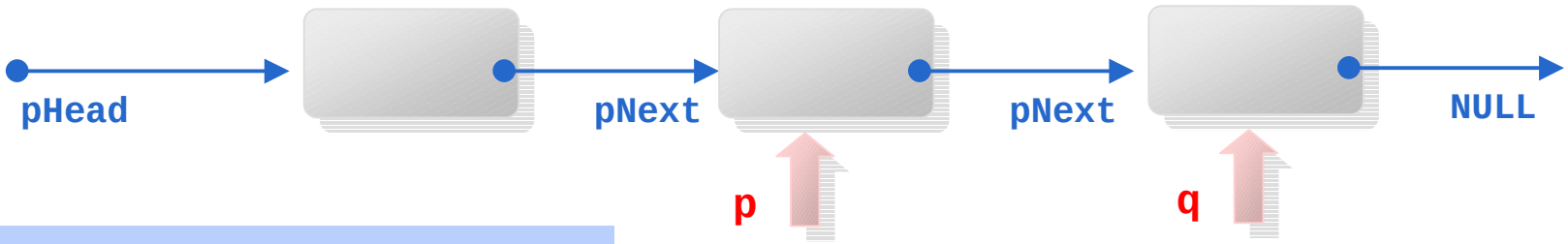
```
Datum *pIter;
Datum* first() {
   pIter = pHead; return pIter;
}
Datum* next() {
   pIter = pIter -> pNext;
   return pIter;
}
```

Verwendung

```
Datum* p = first();
while (p != NULL) {
   printf("%d:%d:%d\n …   )
   p = next();
}
```

Clear - Freigeben der ganzen Liste

- Zum löschen muss ein zweiter Zeiger (z.B. q) eingeführt werden, da sonst auf die schon freigegebene struct zugegriffen würde




```
void clear() {
  Datum *p = pHead, *q;
  while (p != NULL) {
    q = p -> pNext;
    free(p);
    p = q;
  }
}
```

Allgemeine Liste
ADTs

Die allgemeine Liste, Modul

outlook

- Die Liste ist eine fundamentale Datenstruktur ähnlich zu Arrays
  - wird für andere Datenstrukturen in der Implementation verwendet, z.B. Stack, Queue
  - Vorteil gegenüber Array: die Grösse muss nicht festgelegt werden
- In den meisten Programmiersprachen Teil der Bibliothek, z.B. Java, C#, C++
- In C muss sie oft leider selber implementiert werden
- Es ist sinnvoll, eine gute Implementation als Abstrakten Datentyp (ADT) zur Verfügung zu haben
- ADTs sind Vorstufe zu Klassen/Objekte

Implementierungsapekte

outlook

- Liste als ADT implementieren
  - Schnittstelle in .H File
  - Implementierung versteckt in .C File

Vorstufe zu Klassen/Objekte

  - Als sog. ADT https://www.edn.com/electronics-blogs/embedded-basics/4441661/5-simple-steps-to-create-an-abstract-data-type-in-C
  - mit Prefix (z.B. lst) damit es zu keinen Namenskonflikten kommt
    - *oder C++ Namespaces (später)*
  - wiederverwendbar falls entsprechend implementiert
  - Arduino: https://github.com/ivanseidel/LinkedList

vorgestellte Schnittstelle von Arduino und Java "inspiriert"

- Wichtig → Testen
  - Anwendungsfälle müssen überprüft werden
  - arbeiten mit dynamischem Speicher → fehleranfällig
  - etc.

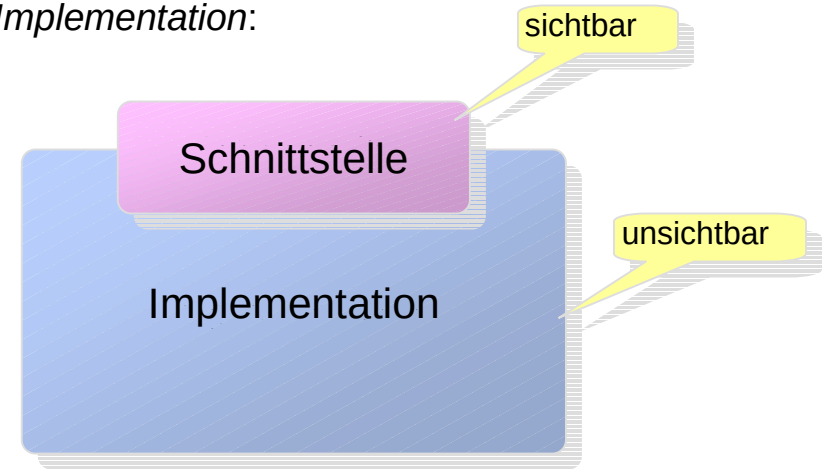
Abstrakte Datentypen (ADT)

outlook

Ein grundlegendes Konzept in der Informatik ist das ***Information Hiding***:

Nur gerade soviel wie für die Verwendung nötig ist, wird auch für andere sichtbar gemacht.

Der ADT besteht aus einer von aussen sichtbaren ***Schnittstelle***, und aus einer ausserhalb des Moduls unsichtbaren *Implementation*:



■ *Schnittstelle*

Ein wesentliches Konzept von ADT's ist

die Definition einer Schnittstelle in Form von Zugriffsmethoden

Nur diese Zugriffsmethoden können die eigentlichen Daten des ADT's lesen oder verändern.

Dadurch ist sichergestellt, dass die innere Logik der Daten erhalten bleibt.

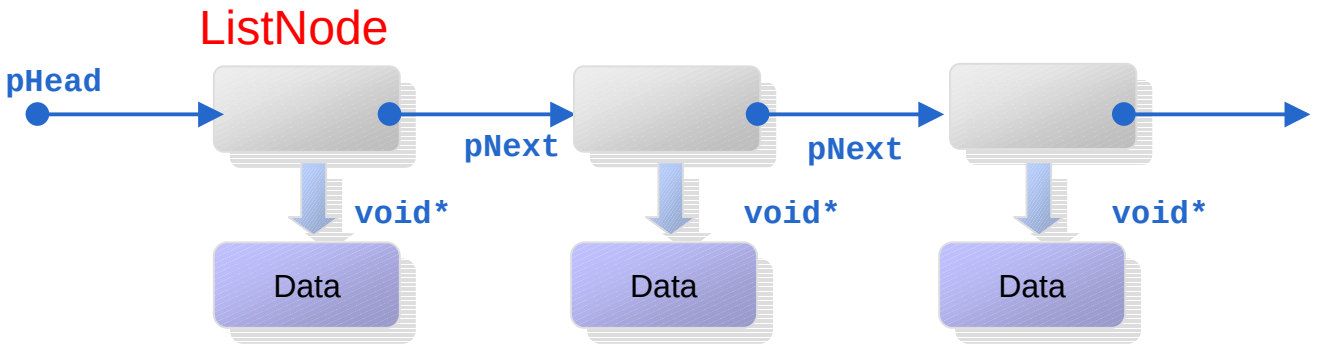
■ *Implementation*

Die Implementation eines ADT's kann verändert werden, ohne dass dies das verwendende Programm merkt.

Man kann sogar verschiedene Implementationen in Erwägung ziehen, die sich zum Beispiel bezüglich Speicherbedarf und Laufzeit unterscheiden.

Allgemeine Liste Datenstruktur

outlook



- Die Verknüpfung wird im (neuen) ListNode Struct vorgenommen
- Die Daten werden über einen void* Pointer verknüpft
  - es braucht keinen *next* Pointer mehr in den Daten selber

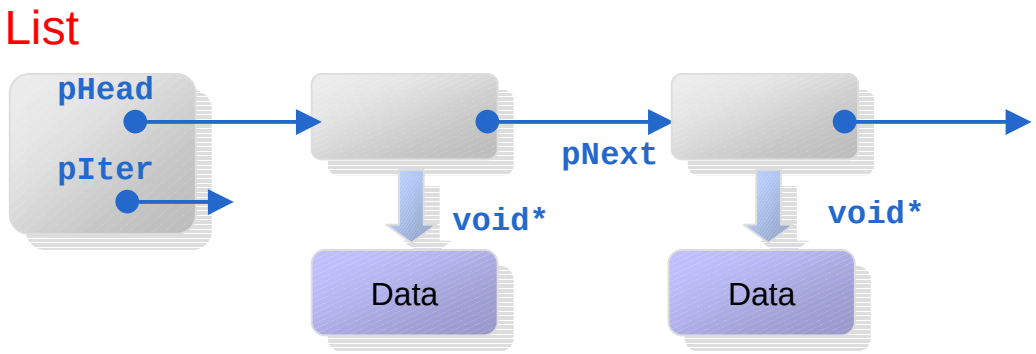
```
typedef struct ListNode {
   struct ListNode *pNext;
   void* pData;
} ListNode;
```

als void* damit beliebige Daten erlaubt

```
typedef struct Datum {
    int tag;
    int monat;
    int jahr;
} Datum;
```

… allgemeine Liste Datenstruktur

outlook



- Damit mehrere Listen in Programm verwendet werden können
- Es wird ein List-Struct eingeführt
- Es wird lediglich ein Pointer auf das "versteckte" Struct im .H File definiert

im H File

```
typedef LinkedList* lstList;
lstList lstCreate();
```

```
struct LinkedList {
    ListNode* pHead;
    ListNode* pIter;
    int size;
};
```

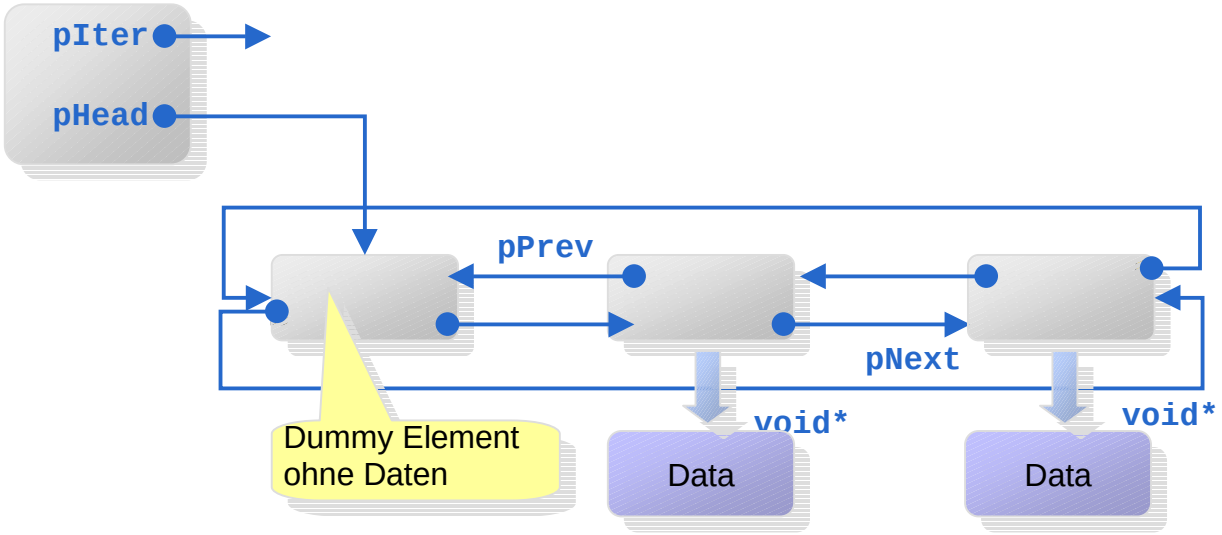
im C File

```
lstList lstCreate() {
   lstList lNew = (lstList)
       malloc(sizeof(struct LinkedList));
   lNew -> pHead = NULL;
   lNew -> pIter = NULL;
   lNew -> size = 0;
   return lNew;
}
```

… allgemeine Liste Datenstruktur - Implement

outlook

Zyklische doppelt verkette Liste



- Es wird in beide Richtungen verknüpft
- Das letzte Element zeigt wieder auf das erste
- Es wird ein leeres Dummy Element eingeführt
- In der Praxis wird diese Implementation vorgezogen:
  - Code ist kürzer, einfacher und effizienter -> doubleLinkedLists.c

… allgemeine Liste Datenstruktur (ADT)

outlook

- Für beliebige Daten verwendbare Liste
- Allen Funktionen die Liste als erstes Argument

im H File

```
lstList lstCreate();
void lstAdd(lstList pList, int pos, void* pData);
void lstAddFirst(lstList pList, void* pData);
void lstAddLast(lstList pList, void* pData);
void* lstGet(lstList pList, int pos);
void* lstFirst(lstList pList);
void* lstNext(lstList pList);
int lstSize(lstList pList);
void* lstRemove(lstList pList, int pos);
void* lstRemoveFirst(lstList pList);
void* lstRemoveLast(lstList pList);
void lstClear(lstList pList);
```

Anwendungen der Liste

Rangliste mit Liste

- Die Rangliste lässt sich einfach mit der allgemeinen Liste implementieren

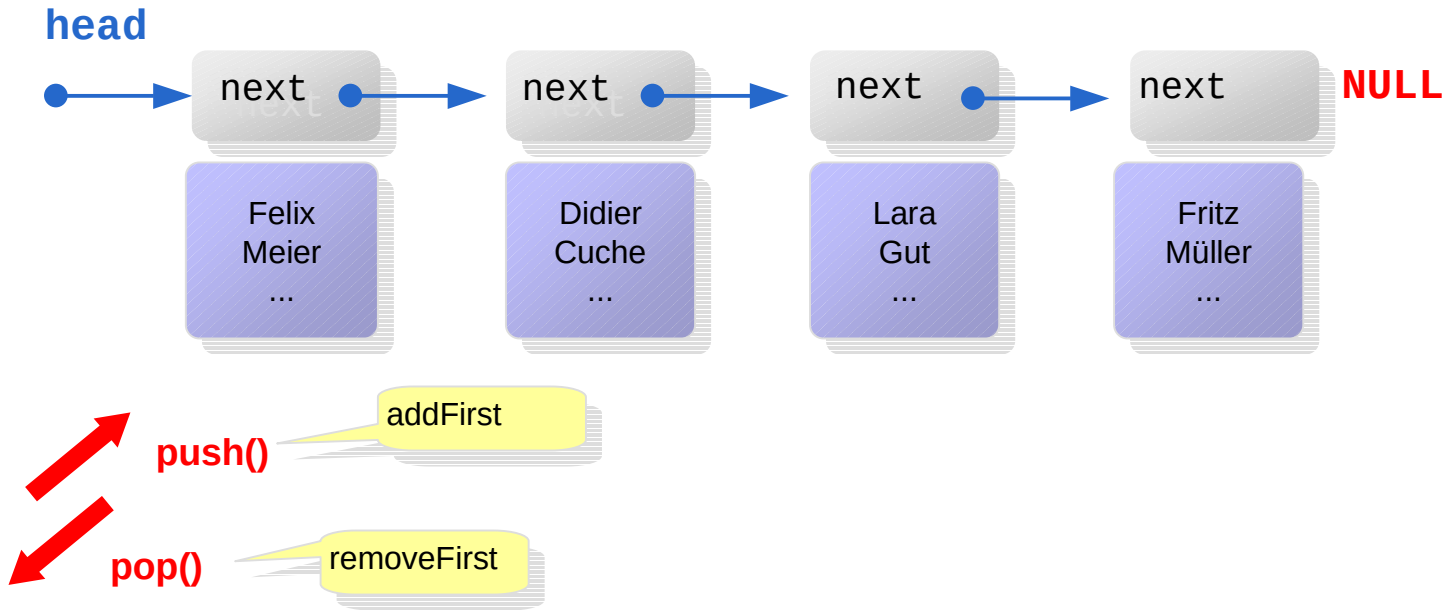
```
typedef struct Teilnehmer {
    char name[20];
    int h, min, sec;
} Teilnehmer;

lstList rangliste;
```

```
...
rangliste = lstCreate();
Teilnehmer *p;
while(true) {
    p = leseTeilnehmer();
    if (p->h == 0 && ...) break;
    lstAddLast(rangliste,p);
}
```

```
void teilnehmerAusgeben(lstList rangliste) {
    Teilnehmer *p;
    p = lstFirst(rangliste);
    while (p != NULL) {
        schreibeTeilnehmer(p);
        p = lstNext(rangliste);
    }
}
```

Anwendungen von Listen: Stack

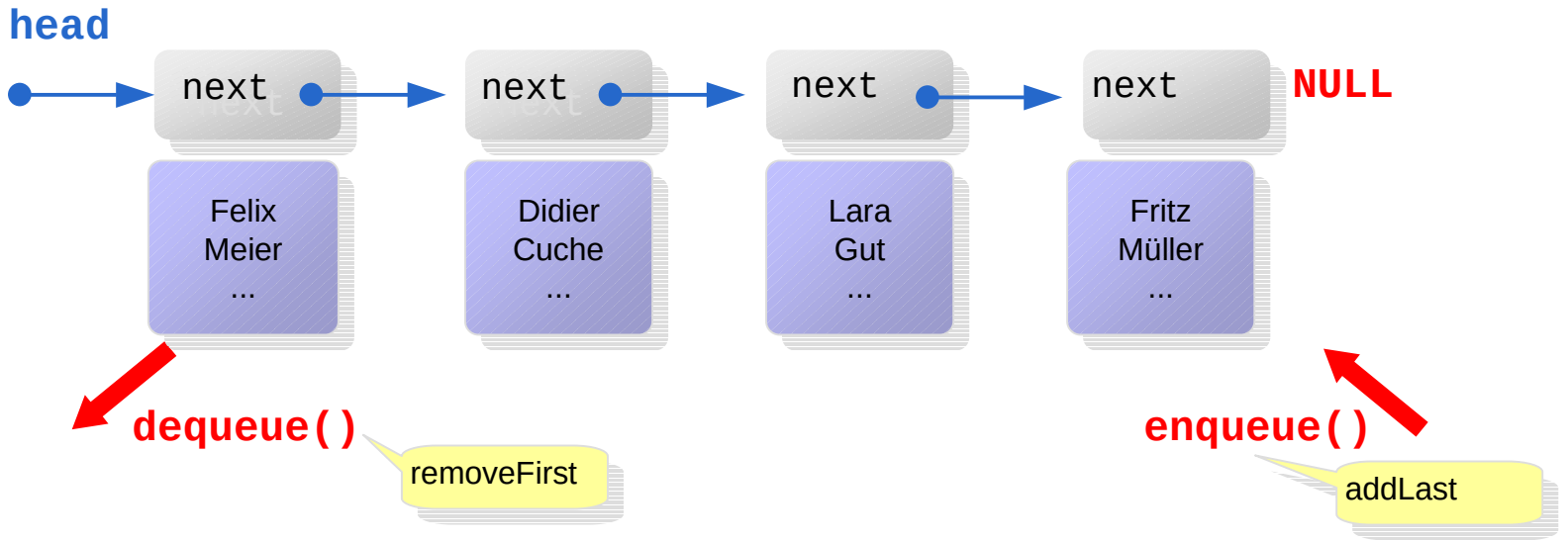
- Stack (LIFO[1])
  - Element am Kopf einfügen: **`push(element)`**
  - Element am Kopf entnehmen: **`element = pop()`**



1) Last In First Out

Anwendungen von Listen: Queue

- Queue (Warteschlange, FIFO[1])
  - Element am Ende anhängen: **`enqueue(elem)`**
  - Element am Kopf entnehmen: **`elem = dequeue()`**



1) First In First Out

Noch Fragen?