

# Kapitel 4

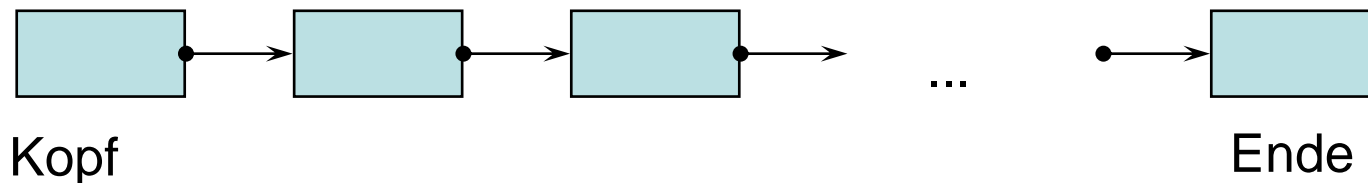
## Listen

Eine (lineare) Liste ist eine Datenstruktur zur Verwaltung einer **beliebig großen Anzahl** von Elementen eines **einheitlichen Typs**.

Der Zugriff auf die einzelnen Elemente einer (simplen) Liste ist nur vom **Kopf (Head)** aus möglich.

Das **Ende** der Liste wird auch als **Tail** bezeichnet.

Die Elemente werden in einer **Sequenz** angeordnet, die sich (meist) aus der **Eintragereihenfolge** ableiten lässt (ungeordnet).



Datenstrukturen stellen eine Abstraktion eines  
Vorstellungsmodells dar

Begriff des **Abstrakten Datentyps (ADT)**

Sagt nichts über die physische Realisierung am Computer aus  
Verschiedene Realisierungen denkbar!

Realisierung oft abhängig von Problemstellung, Programmierumgebung,  
Zielsetzungen, ...

Mögliches Vorstellungsmodell „Liste“  
„Perlenschnur“, Perlen werden an einem Ende aufgefädelt

## Definitionen:

- Eine Liste  $L$  ist eine geordnete Menge von Elementen

$$L = (x_1, x_2, \dots, x_n)$$

- Die Länge einer Liste ist gegeben durch

$$|L| = |(x_1, x_2, \dots, x_n)| = n$$

- Eine leere Liste hat die Länge 0.
- Das  $i$ -te Element einer Liste  $L$  wird mit  $L[i]$  bezeichnet, daher gilt  $1 \leq i \leq |L|$

## Einfügen

*Add*: Element am Kopf einfügen

## Zugriff

*FirstElement*: Kopfelement bestimmen

## Löschen

*RemoveFirst*: Kopfelement entfernen

## Erzeugen

*Constructor*: Liste neu anlegen

## Längenbestimmung

*Length*: Anzahl der Elemente bestimmen

## Inklusionstest

*Member*: Test, ob Element enthalten ist

andere  
Operationen denkbar

## Deklaration C++, Klasse

```
typedef ... ItemType;  
...  
class List {  
public:  
    List(); // Constructor  
    void Add(itemType a);  
    ItemType FirstElement();  
    void RemoveFirst();  
    int Length();  
    int Member(itemType a);  
}
```

zur Verwendung in  
der Klassen Def.,  
besserer Ansatz  
mit C++ Templates

## 4.2 Implementierung von Listen

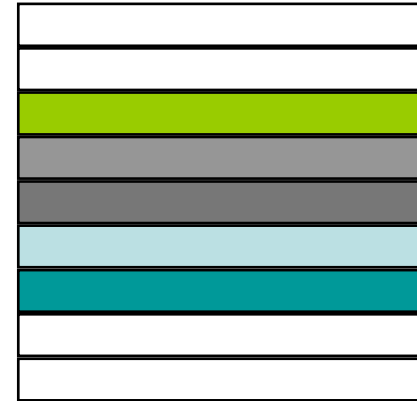


### Speichertypen

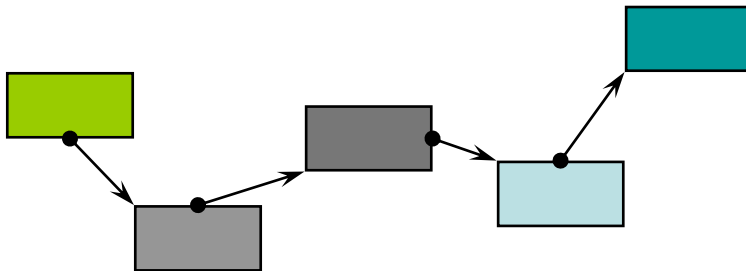
#### Contiguous memory



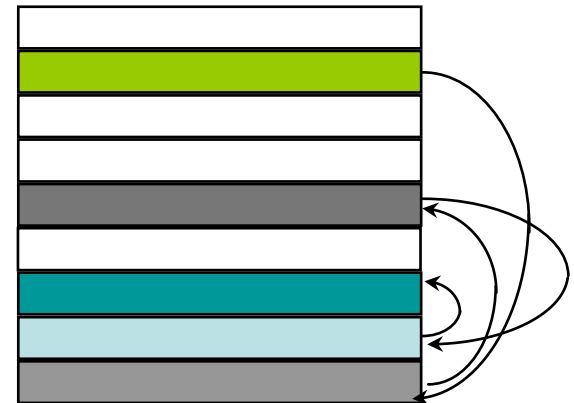
22200  
22208  
22216  
22224  
22232  
22240  
22248  
22256  
22264



#### Scattered (Linked) memory



22200  
22208  
22216  
22224  
22232  
22240  
22248  
22256  
22264



## Contiguous memory

Physisch **zusammenhängender** Speicherplatzbereich, äußerst starr, da beim Anlegen die endgültige Größe fixiert wird.

Verwaltung über das **System**.

Datenstrukturen auf der Basis von contiguous memory können nur eine **begrenzte** Anzahl von Elementen aufnehmen.

## Scattered (Linked) memory

Physisch verteilter Speicherbereich, sehr **flexibel**, da die Ausdehnung **dynamisch** angepasst werden kann.

Verwaltung (meist) über das **Anwendungsprogramm**.

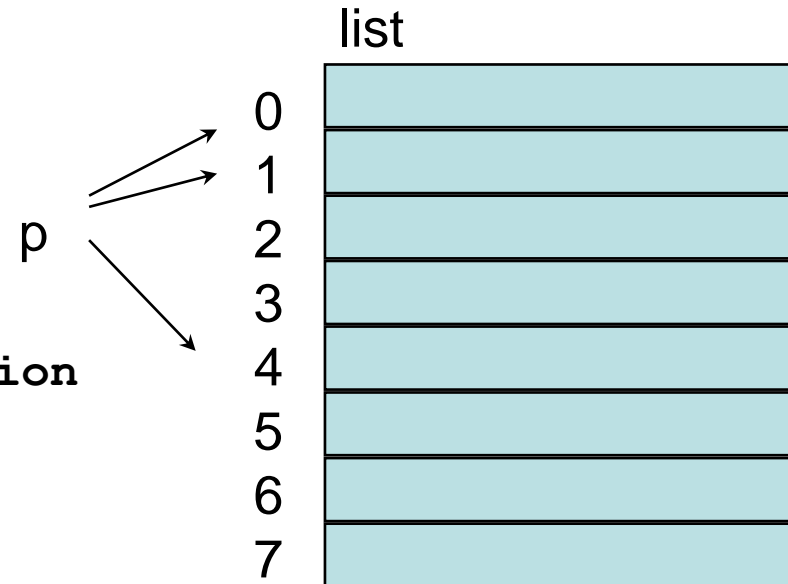
Datenstrukturen können **beliebig groß** werden.



### Statische Implementation - contiguous memory

Speichern der Elemente in einem Feld begrenzter Länge

```
typedef int ItemType;  
class List {  
private:  
    ItemType list[8];  
    // Datenstruktur  
    int p;  
    // nächste freie Position  
    ...  
}
```



Länge = 8

# Liste - statisch - Erzeugen - Zerstören - Einfügen



Erzeugen,

```
List::List() {p = 0;}
```

Zerstören

```
List::~~List() {p = 0;}
```

Einfügen

```
void List::Add(ItemType a) {  
    if(p < 8) {  
        list[p] = a;  
        p++;  
    }  
    else cout << "Error-add\n";  
}
```



## Zugriff

```
ItemType List::FirstElement() {  
    if(p > 0) return list[p-1];  
    else cout << "Error-first\n";  
}
```

## Löschen

```
void List::RemoveFirst() {  
    if(p > 0) p--;  
    else cout << "Error-remove\n";  
}
```

Länge,

```
int List::Length() {  
    return p;  
}
```

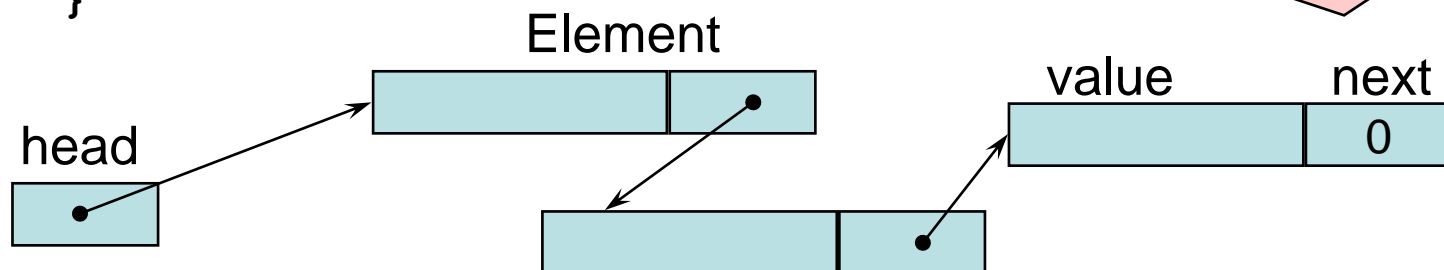
Inklusionstest

```
int List::Member(ItemType a) {  
    int i = 0;  
    while(i < p && list[i] != a) i++;  
    if(i < p) return 1;  
    else return 0;  
}
```

### Dynamische Implementation - linked memory

Dynamisch erweiterbare Liste unbegrenzter Länge

```
typedef int ItemType;  
class List {  
public:  
    class Element {          // Elementklasse  
        ItemType value;  
        Element* next;  
    };  
    Element* head;  // DS Kopf  
    ...  
}
```



Eine Datenstruktur heißt **rekursiv** oder **zirkulär**, wenn sie sich in ihrer Definition selbst referenziert

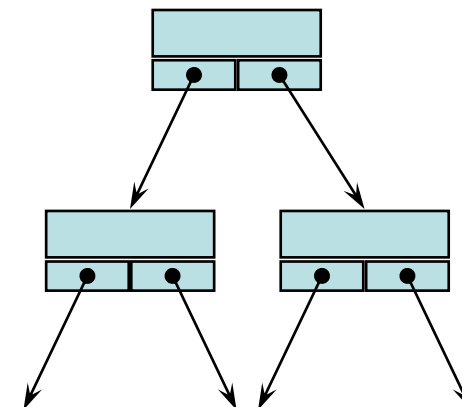
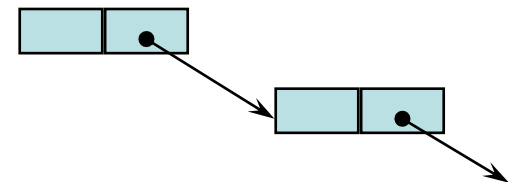
Basismodell für dynamisch erweiterbare Datenstrukturen

Liste

```
class Element{  
    InfoType Info;  
    Element* Next;  
}
```

Baum

```
class Node {  
    KeyType Key;  
    Node* LeftChild;  
    Node* RightChild;  
}
```



## Einfügen

```
void List::Add(ItemType a) {  
    Element* help;  
    help = new Element;  
    help->next = head;  
    help->value = a;  
    head = help;  
}
```

Typischerweise  
am Kopf der Liste



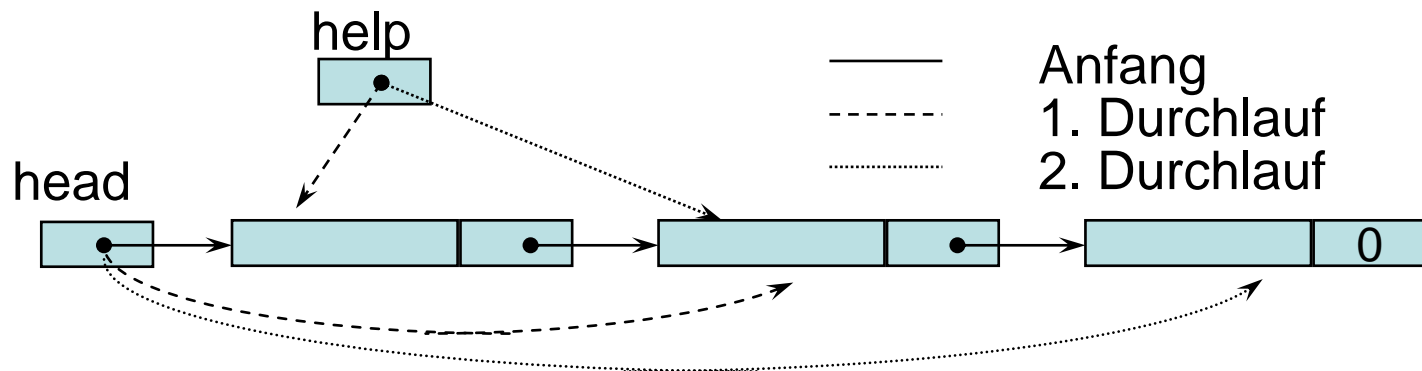
Vor dem Einfügen



## Erzeugen, Löschen

```
List::List() { head = 0; }
```

```
List::~~List() {  
    Element* help;  
    while(head != 0) {  
        help = head;  
        head = head->next;  
        delete help;  
    }  
}
```





## Zugriff

```
ItemType List::FirstElement() {  
    if(head != 0)  
        return head->value;  
    else  
        cout << "Error-first\n";  
}
```

## Löschen

```
void List::RemoveFirst() {  
    if(head != 0) {  
        Element* help;  
        help = head;  
        head = head->next;  
        delete help;  
    } else cout << "Error-remove\n";  
}
```

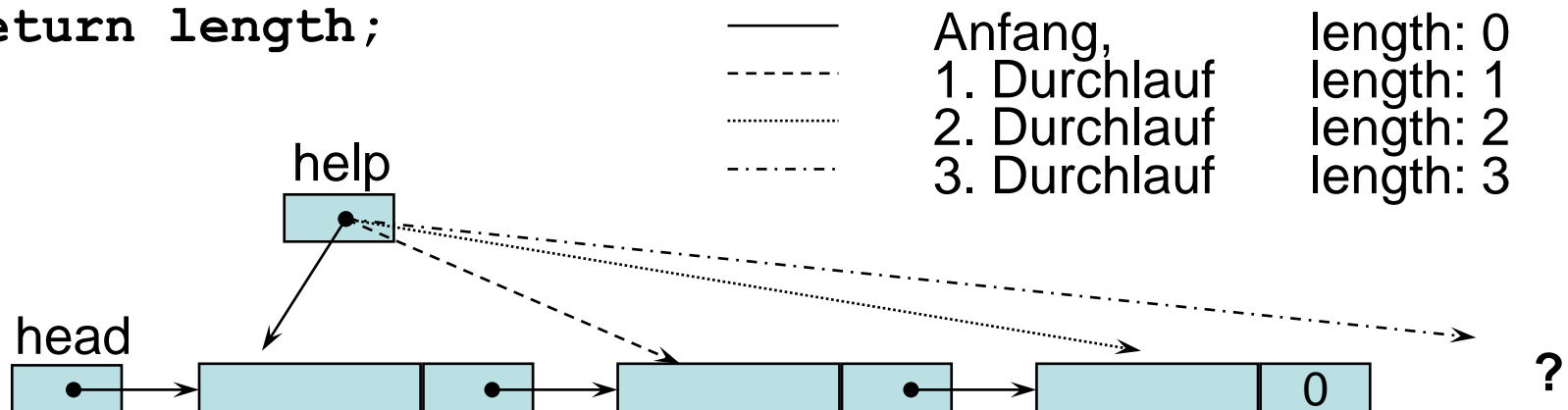


Vor dem Löschen



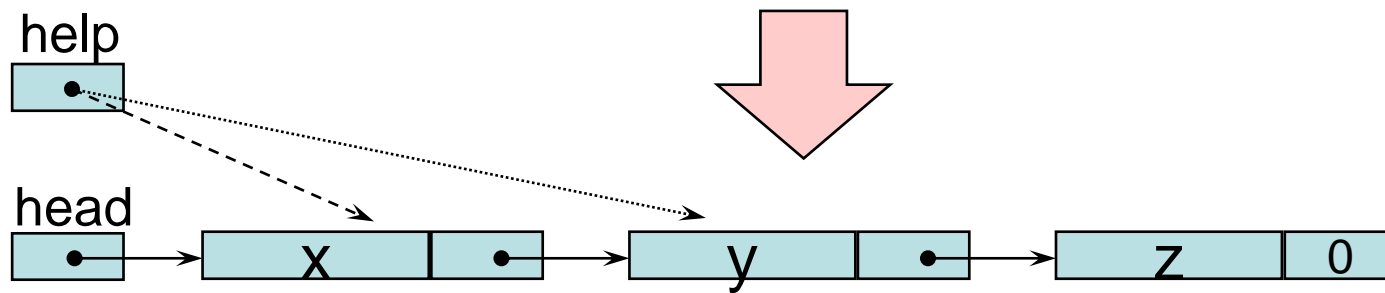
## Länge

```
int List::Length() {  
    Element* help = head;  
    int length = 0;  
    while(help != 0) {  
        length++;  
        help = help->next;  
    }  
    return length;  
}
```



## Inklusionstest

```
int List::Member(ItemType a) {  
    Element* help = head;  
    while(help != 0 && help->value != a)  
        help = help->next;  
    if(help != 0) return 1;  
    else return 0;  
}
```

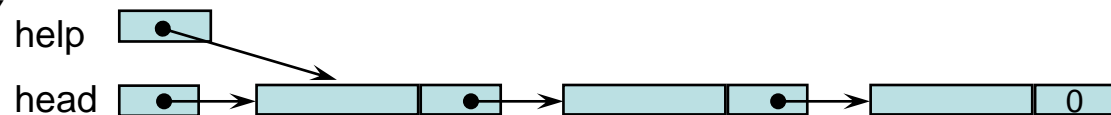


Schema für Sequentielles Abarbeiten einer Liste (iterativ),  
d.h. „Besuchen aller Elemente“



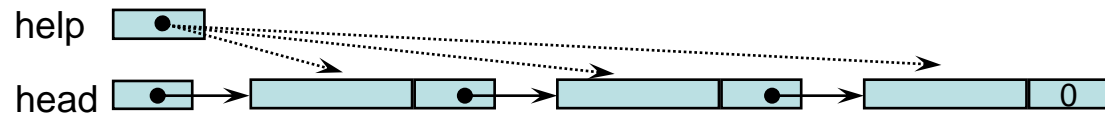
1. Initialisieren des Hilfszeigers

```
help = head;
```



2. Weitersetzen des Hilfszeigers (Position)

```
help = help -> next;
```



3. Abfrage auf Listenende (und Suchkriterium)

```
while ( help != 0 && ... ) { ... }
```



Der **Stack (Kellerspeicher)** ist ein Spezialfall der Liste, die die Elemente nach dem **LIFO** (last-in, first-out) Prinzip verwaltet

Idee des Stacks: Man kann nur auf das oberste, zuletzt daraufgelegte Element zugreifen (vergleiche Buchstapel, Holzstoß, ...)

Anwendungen: Kellerautomaten, Speicherverwaltung, HP-Taschenrechner (UPN), ...



Das Verhalten des Stacks lässt sich über seine (recht einfachen) Operationen beschreiben

push: Element am Stack ablegen

top: Auf oberstes Element des Stacks zugreifen

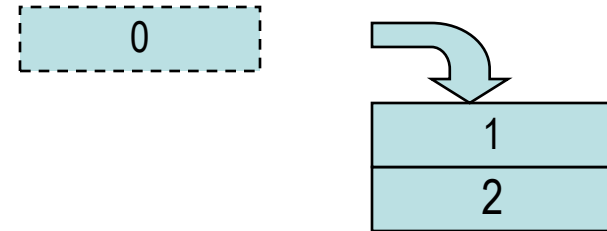
pop: Element vom Stack entfernen

isEmpty: Test auf leeren Stack



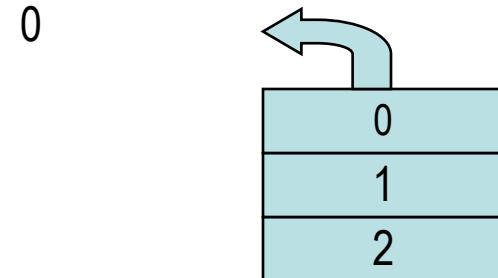
## Methode 'Push'

Element wird auf dem Stack  
abgelegt (an oberster  
Position).



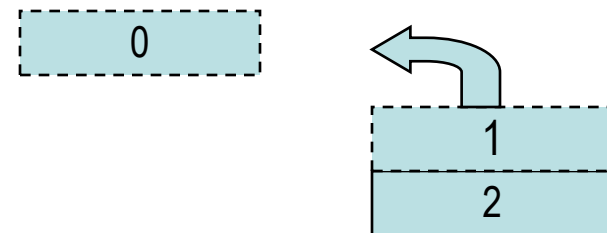
## Methode 'Top'

Liefert den Inhalte des obersten  
Elementes des Stacks.



## Methode 'Pop'

Oberstes Element des Stacks  
wird entfernt.



Stack ist eine spezielle Liste, daher können die Stack-  
operationen durch Listenoperationen ausgedrückt werden.

Push(S,a)  $\Rightarrow$  Add(S,a)

Top(S)  $\Rightarrow$  FirstElement(S)

Pop(S)  $\Rightarrow$  RemoveFirst(S)

IsStackEmpty(S)

$\Rightarrow$  wenn Length(S) = 0 return true  
sonst false



Die Queue (Warteschlange) ist ein Spezialfall der Liste, die die Elemente nach dem **FIFO** (first-in, first-out) Prinzip verwaltet

Idee: Die Elemente werden hintereinander angereiht, wobei nur am Ende der Liste Elemente angefügt und vom Anfang der Liste weggenommen werden können

Anwendungen: Warteschlangen, Pufferverwaltung, Prozessmanagement, Stoffwechsel,...

Einfache Operationen

Enqueue

Element am Ende der Queue ablegen

Front

Erstes Element der Queue zugreifen

Dequeue

Erstes Element aus der Queue entfernen

IsEmpty

Test auf leere Queue



## Methode 'Enqueue'

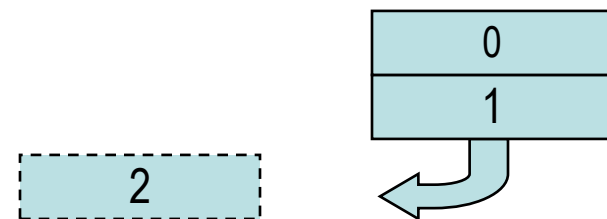
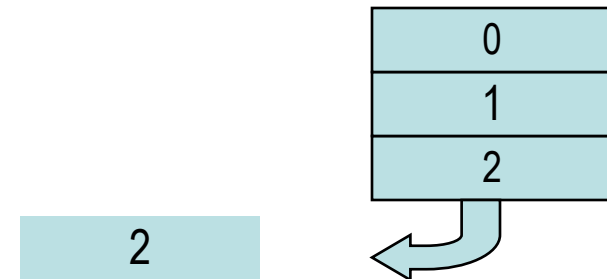
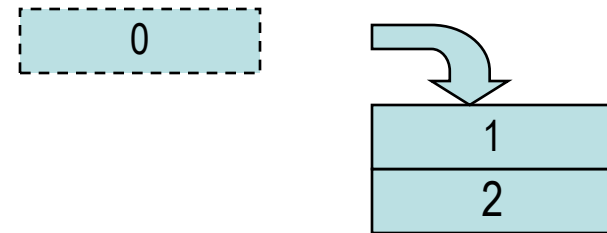
Element wird am Ende der Queue abgelegt (an letzter Position)

## Methode 'Front'

Liefert den Inhalte des ersten Elementes der Queue

## Methode 'Dequeue'

Erstes Element der Queue wird entfernt



Queue ist ebenfalls eine spezielle Liste, daher sollten alle Queueoperationen auch durch Listenoperationen ausgedrückt werden können.

Enqueue(Q,a)  $\Rightarrow$  Add(S,a)

Front(Q)  $\Rightarrow$  ?

Dequeue(Q)  $\Rightarrow$  ?



Nicht ganz trivial !

Möglichkeit (umständlich!)

Zugriff auf das erste Queueelement (letzte in der Liste) durch iteratives Entfernen aller Elemente und gleichzeitigen Aufbau einer 'gestürzten' Hilfsliste. Danach Vorgang umkehren.

## Generelle Unterscheidung zwischen statischer und dynamischer Realisierung

statische R.: contiguous memory, Felder

dynamische R.: dynamic memory, dynamische Objekte

## Datenverwaltung

Einfügen und Löschen wird unterstützt

## Datenmenge

statische R.: beschränkt, abhängig von der Feldgröße

dynamische R.: unbeschränkt

abhängig von der Größe des vorhandenen Speicherplatzes

## eher simple Modelle

# Aufwandsvergleich "unserer" Listen Implementationen



	Liste statisch	Liste dynamisch
Speicherplatz	<b><math>O(n)</math></b>	<b><math>O(n)</math></b>
Konstruktor	$O(1)$	$O(1)$
Destruktor	$O(1)$	<b><math>O(n)</math></b>
Add	$O(1)$	$O(1)$
FirstElement	$O(1)$	$O(1)$
RemoveFirst	$O(1)$	$O(1)$
Length	$O(1)$	<b><math>O(n)</math></b>
Member	<b><math>O(n)</math></b>	<b><math>O(n)</math></b>

Achtung: Eigentlicher Aufwand  $O(n)$  in  
Add und RemoveFirst versteckt

### Doubly Linked List

doppelt verkettet Liste

### Circular List

Zirkulär verkettete Liste

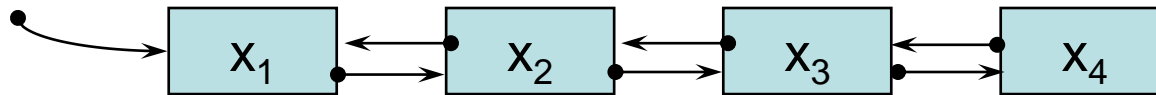
### Ordered List

Geordnete Liste

### Double Ended List

Doppelköpfige Liste

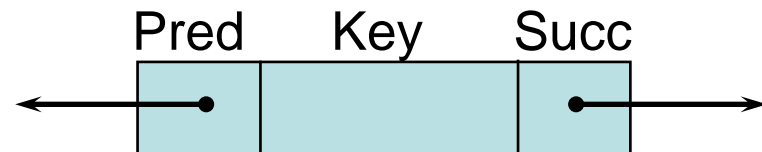
## Doppelt verkettete Liste



jedes Element besitzt 2 Zeiger, wobei der eine auf das vorhergehende  
und der andere auf das nachfolgende Element zeigt

Basis-Operationen einfach

```
class Node {  
    KeyType Key;  
    Node* Pred;  
    Node* Succ;  
}
```

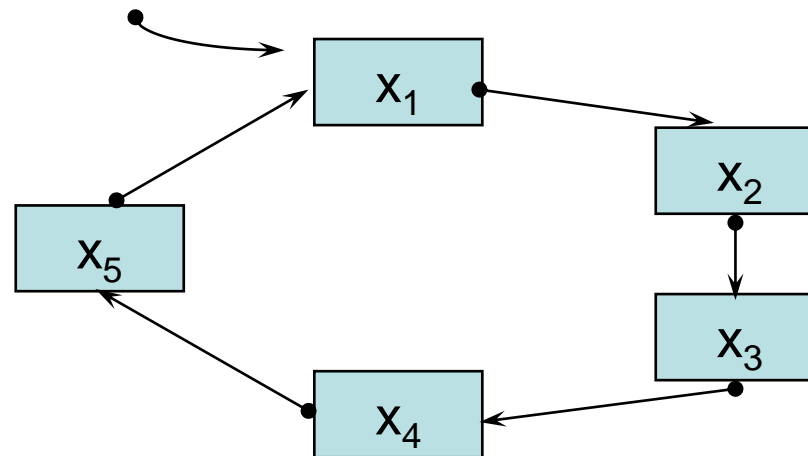


## Zirkulär verkettete Liste

Zeiger des letzten Element verweist wieder auf das erste Element

Ring Buffer

Vorsicht beim Eintragen und Löschen des ersten Elementes!



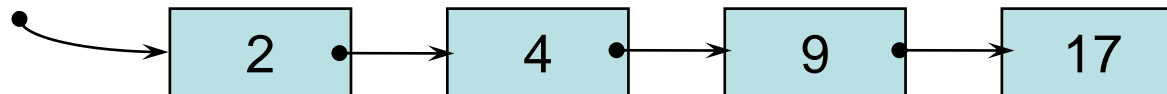


## Geordnete Liste

Elemente werden entsprechend ihres Wertes in die Liste an spezifischer Stelle eingetragen

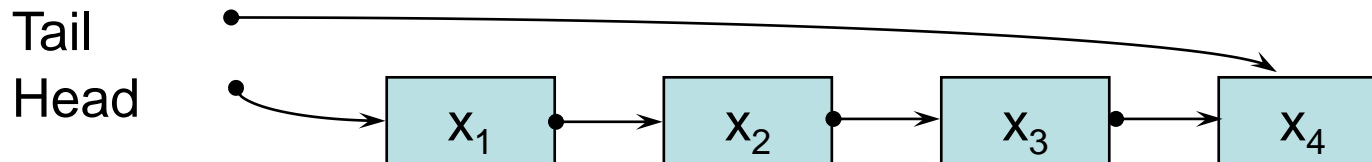
Meist der Größe nach geordnet

Eintragen an spezifischer Stelle, die erst gefunden werden muss → Traversieren

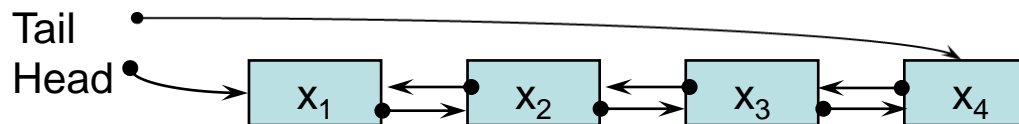


## Liste mit 2 “Köpfen”

Jede Liste besitzt 2 Zeiger, die zum Kopf und zum Ende der Liste zeigen  
Vereinfacht das Einfügen am Kopf und am Ende der Liste



Kann mit anderen Listen Strukturen kombiniert werden, z.B.  
Doubly Linked Double Ended List



## Listen

- Operationen

- Speicherung

- Contiguous - Scattered memory

## Stack – Queue

## Vergleich

## Spezielle Listen

- Doubly Linked List

- Circular List

- Ordered List

- Double Ended List