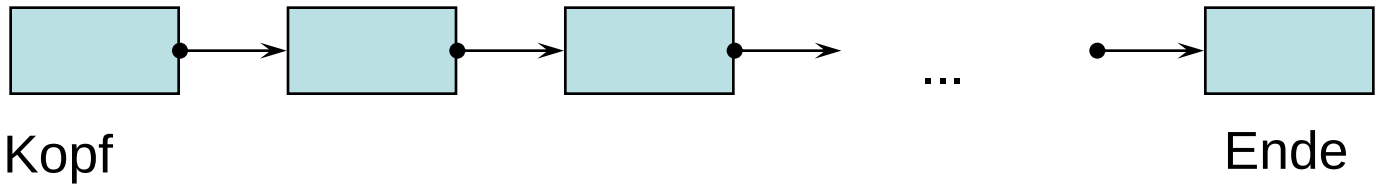
universität wien

Kapitel 4
Listen

4.1 Definition 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).



Abstraktion

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

Liste

Definitionen:

- Eine Liste L ist eine geordnete Menge von Elementen

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

- Die Länge einer Liste ist gegeben durch

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

- Eine leere Liste hat die Länge 0.

- Das i-te Element einer Liste L wird mit L[i] bezeichnet, daher gilt $1 \leq i \leq |L|$

Methoden auf ADT Liste

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 siehe später!

Methode ‘Add’

Einfügen eines Elementes a am Kopf einer Liste L, d.h.

$$L = (x_1, x_2, \ldots, x_n), x_0$$

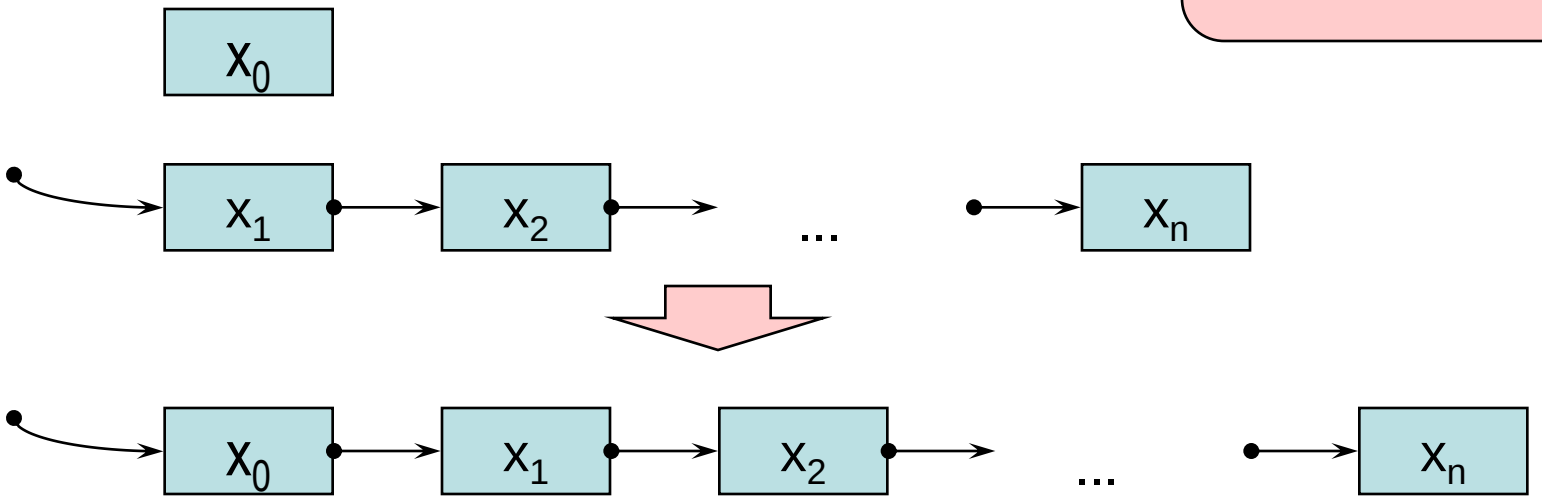
$$\Downarrow$$

$$Add(L, x_0)$$

$$\Downarrow$$

$$L = (x_0, x_1, \ldots, x_n)$$

Vorstellungsmodell „Perlenschnur“: *Perle auffädeln*



Methode ‘FirstElement’

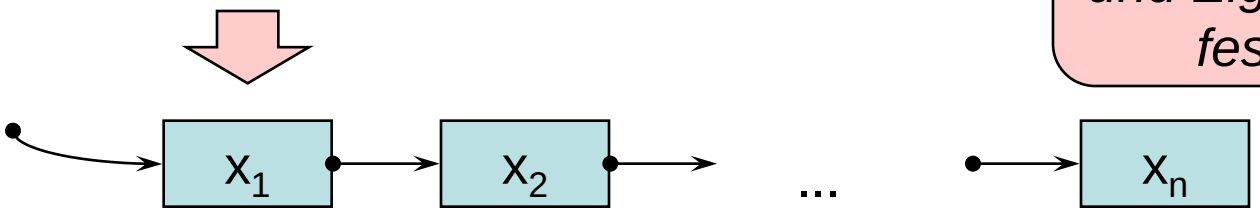
Zugriff über das Kopfelement (x_1, das erste Listen-element) auf die Liste , d.h.

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

$$\text{FirstElement}(L) \rightarrow x_1$$

Fehler falls Liste leer

Vorstellungsmodell „Perlenschnur“: *Perle betrachten und Eigenschaften feststellen*



$$x_1$$

liefert den Wert des Elements, NICHT das Listenelement !

Methode ‘RemoveFirst’

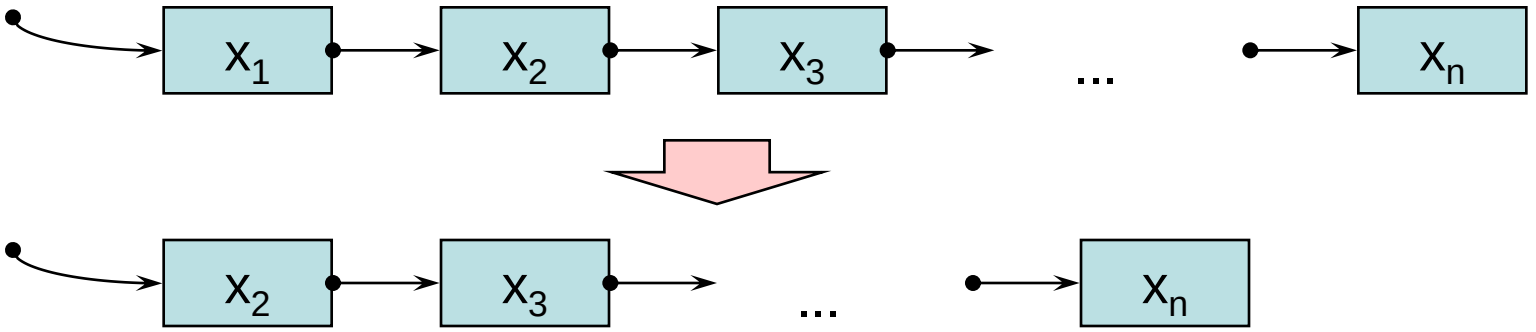
Löscht das Kopfelement (x_1, das erste Listenelement) aus der Liste L, d.h.

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

RemoveFirst(L)

$L = (x_2 \ldots, x_n)$, mit $|L| = n-1$

Vorstellungsmodell „Perlenschnur“: *Perle abziehen*

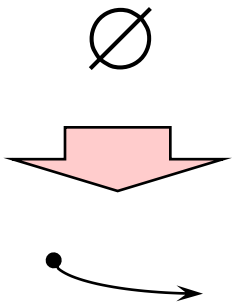


Methode ‘Constructor'

Erzeugt eine neue Liste, die leer ist, d.h. keine Elemente enthält und daher die Länge 0 hat,

$$\text{Constructor}() \rightarrow L, \text{ mit } |L| = 0$$

Vorstellungsmodell „Perlenschnur“: *Perlenschnur vorbereiten*

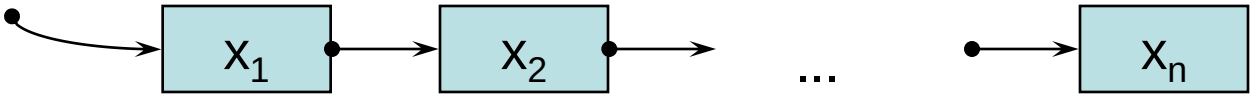


Methode ‘Length’

Bestimmt die Anzahl der Elemente der Liste L, d.h.

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

Length(L) $\rightarrow$ n



n

liefert den ganzzahligen Wert n

Vorstellungsmodell „Perlenschnur“: *Anzahl Perlen bestimmen*

Methode ‘Member’

Überprüft, ob ein gegebenes Element a in der Liste L enthalten ist, d.h.

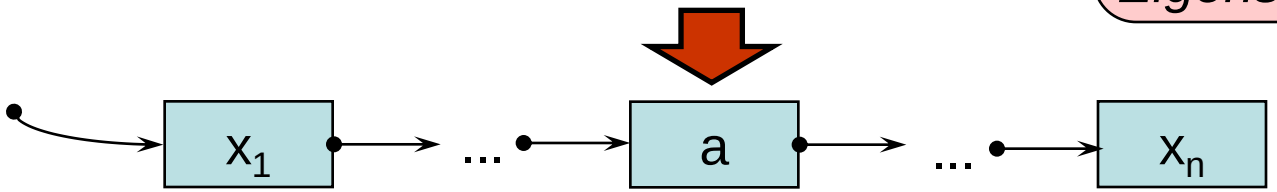
$$L = (x_1, x_2, \ldots, x_n)$$

Member(L, a) $\rightarrow$ [true, false]

true …$\exists$ i | $1 \leq i \leq |L| \ \wedge \ a = x_i$

false … sonst

Vorstellungsmodell „Perlenschnur“: *Perle mit spezifischer Eigenschaft suchen*



true

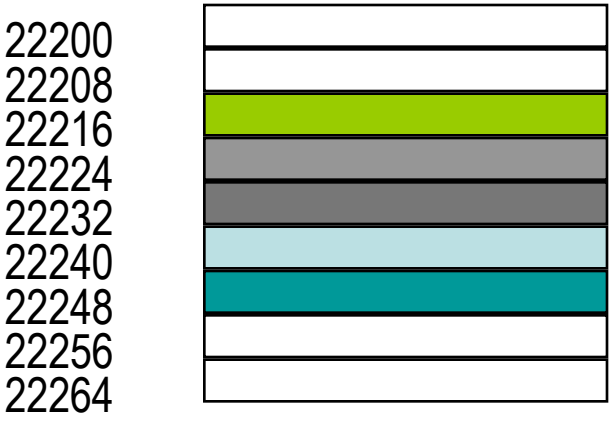
Deklaration C++, Klasse

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

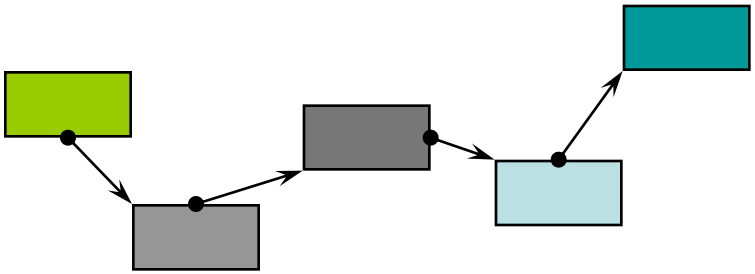
zur Verwendung in der Klassen Def., besserer Ansatz mit C++ Templates

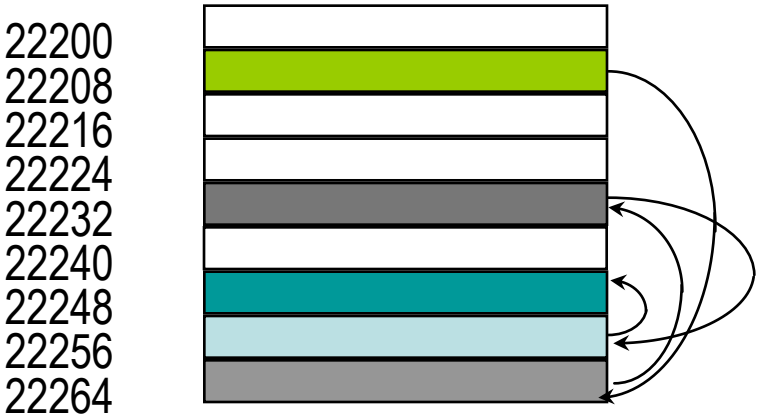
Speichertypen

Contiguous memory



Scattered (Linked) memory





Memory Typen

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.

C++ Konstrukt: Feld

Contiguous memory

Ein Feld bzw. Array ist einer Anreihung einer fixen Anzahl von Elementen des gleichen Typs.

Der Zugriff auf ein einzelnes Feldelement erfolgt über eine Index (die relative Position im Feld). Der Index startet oBdA mit *0* und endet mit *Anzahl-1*.

Vereinbarung

<ETyp> <Feldname> ’[‘ < EZahl> ’]’

z.B.:

```
double x[10];
int a[10000];
char name[25];
```

Zugriff

<Feldname> ‘[‘ < index> ’]’

z.B.:

```
x[0] = 3.1415 * r * r;
a[9999] = 0;
```

C++ Konstrukt: Dynamische Objekte

universität wien

Scattered memory

- Dynamische Objekte
  - Objekte die **zur Laufzeit** des Programmes durch Programmanweisungen **erzeugt** und für die Speicherplatz angelegt wird.
  - Diese Objekte besitzen **keinen Namen** und werden über sog. **Zeiger** (Pointer) verwaltet.
- Zeiger bzw. Pointer
  - Eine Variable, die die Adresse eines Objekts (Adressoperator '&') enthält

z.B.:

```
double x = 3.14159;
double* p = &x;
  // & liefert Adr. von x
cout << *p;
  // * deref. Ptr., d.h.
  // druckt 3.14159
```

x	≡	22192	3.14158
p	≡	22200	22192

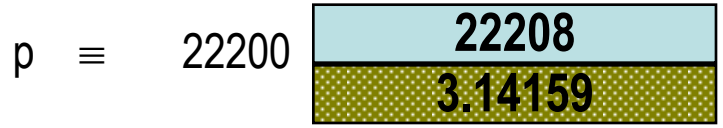
Erzeugung bzw. Zerstörung

new … erzeugt ein neues Objekt

delete … zerstört ein existierendes Objekt

```
double * p = new double;
* p = 3.14159;
```


```
delete p;
```



alter, undefinierter Wert

vom System freigegeben

Achtung:

```
p = 0 // erlaubt,
      // Initialisierung
p = 4711 // verboten
   // Adressmanipulation
```

Operatoren

& ... liefert die Adresse des Objekts

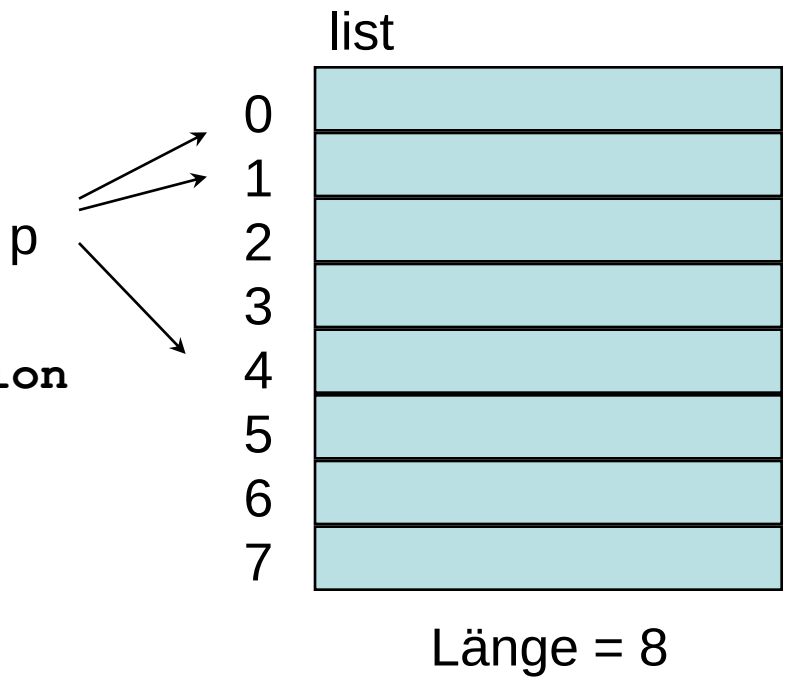
* ... dereferenziert den Zeiger und liefert den Inhalt der ref. Objektes

-> ... Zugriff auf eine Strukturkomponente über einen Zeiger,
d.h. x->y entspricht (*x).y

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
  …
}
```



Liste - statisch - Erzeugen - Zerstören - Einfügen

universität wien

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";
}
```

universität wien

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";
}
```

Liste - statisch - Länge - Inklusionstest

universität wien

Länge,

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

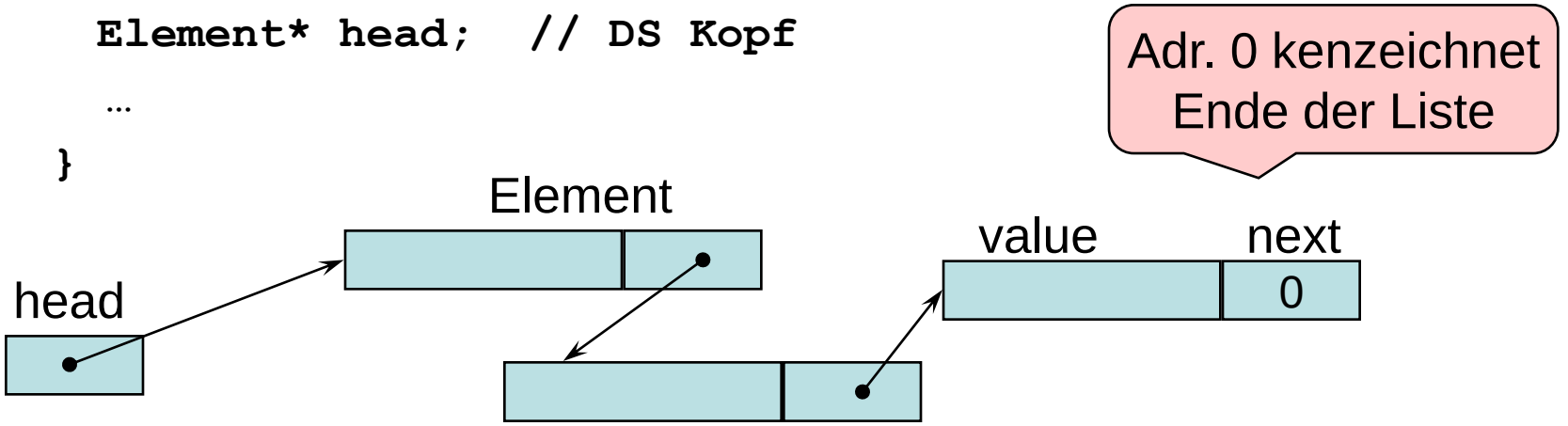
Inklusionstest

```
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
  …
}
```



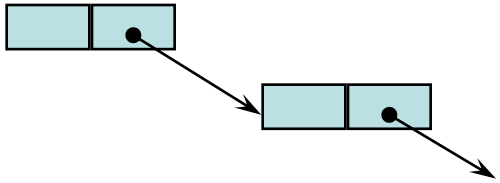
Rekursive Datenstruktur

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

Basismodell für dynamisch erweiterbare Datenstrukturen

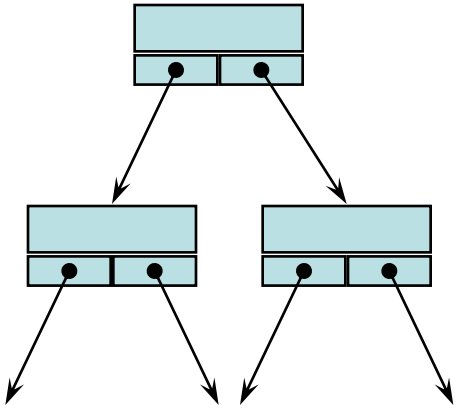
Liste

```
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;
}
```

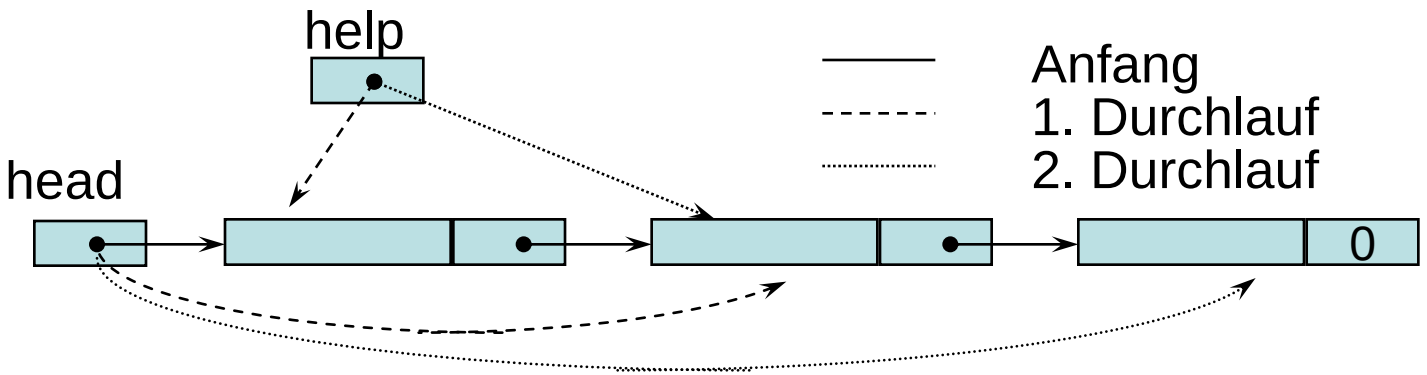
Typischerweise
am Kopf der Liste

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

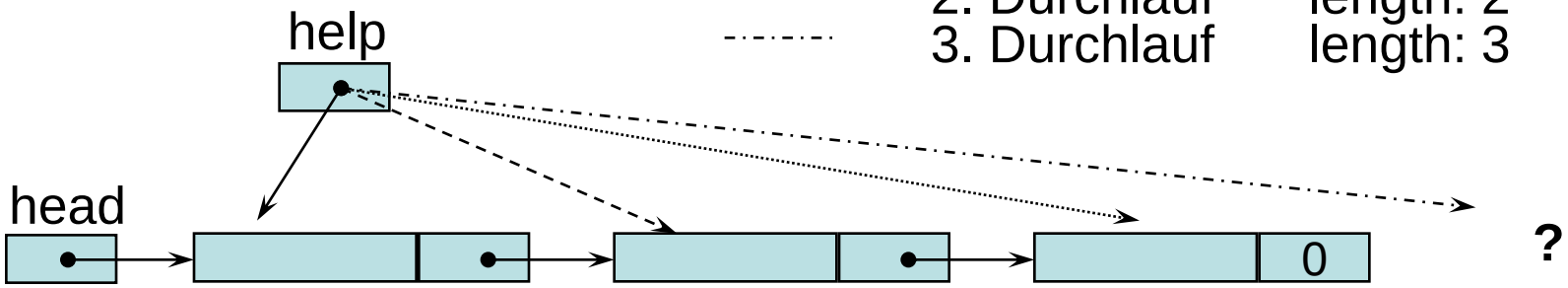
Liste - dynamisch - Länge

Länge

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

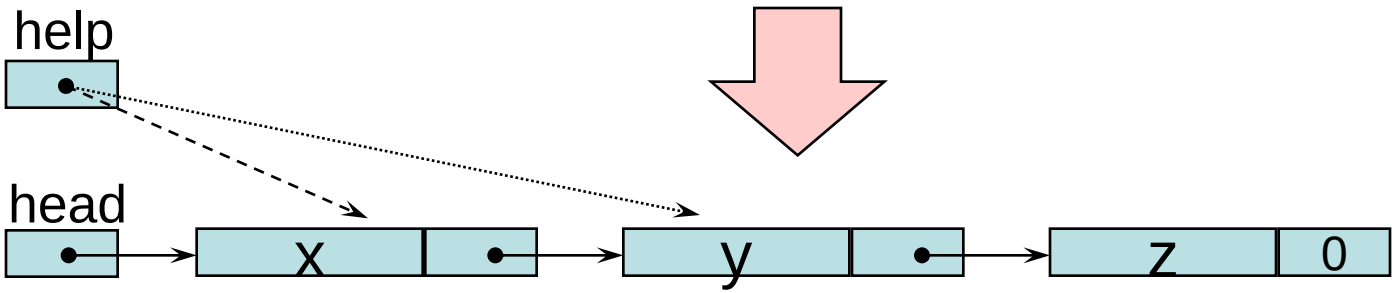
Checkpoint

Anfang, length: 0
1. Durchlauf length: 1
2. Durchlauf length: 2
3. Durchlauf length: 3



Inklusionstest

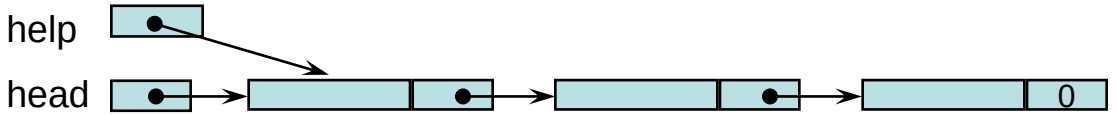
```
int List::Member(ItemType a) {
  Element* help = head;
  while(help != 0 && help->value != a)
    help = help->next;
  if(help != 0) return 1;
  else return 0;
}
```



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

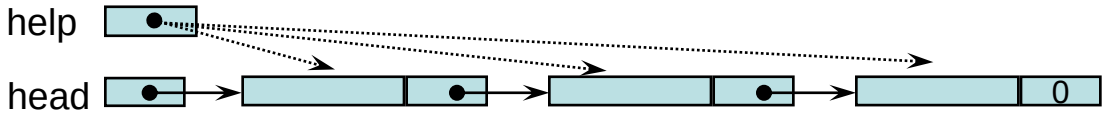
1. Initialisieren des Hilfszeigers

```
help = head;
```



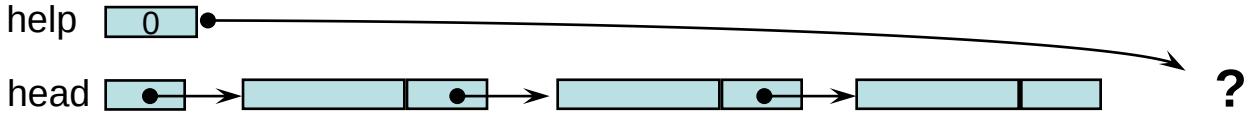
2. Weitersetzen des Hilfszeigers (Position)

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



3. Abfrage auf Listenende (und Suchkriterium)

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

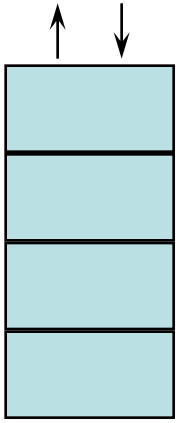


Der **Stack** (**Kellerspeicher**) ist ein Spezialfall der Liste, die die Elemente nach dem **LIFO** (last-in, first-out) Prinzip verwaltet

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



Methoden auf Stacks

Methode ‘Push’

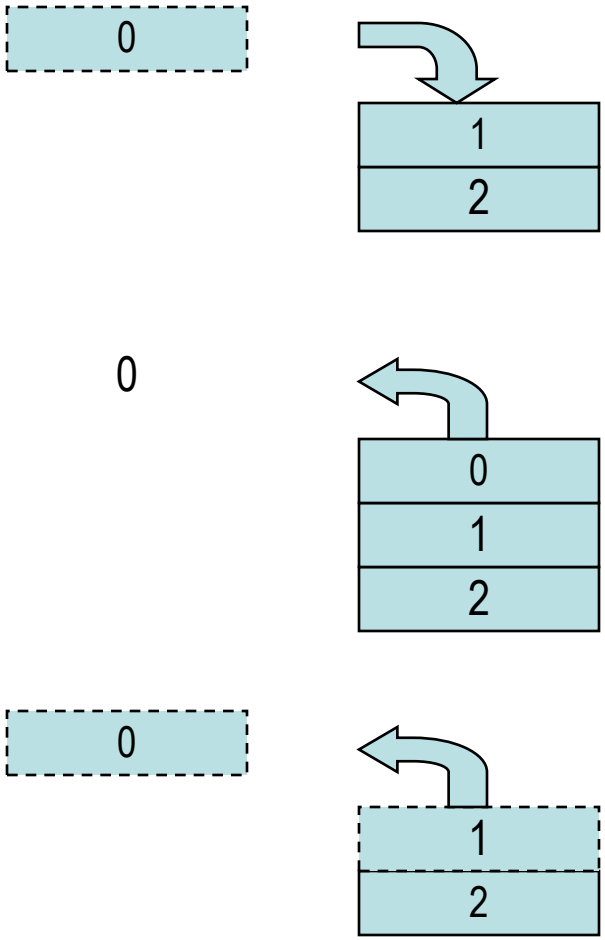
Element wird auf dem Stack abgelegt (an oberster Position).

Methode ‘Top’

Liefert den Inhalte des obersten Elementes des Stacks.

Methode ‘Pop’

Oberstes Element des Stacks wird entfernt.



Stack als Liste

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

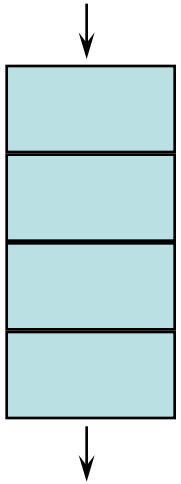
Push(S,a)	$\Rightarrow$	Add(S,a)
Top(S)	$\Rightarrow$	FirstElement(S)
Pop(S)	$\Rightarrow$	RemoveFirst(S)
IsStackEmpty(S)		
	$\Rightarrow$	wenn Length(S) = 0 return true
		sonst false

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, Bufferverwaltung, Prozessmanagement, Stoffwechsel,…

Einfache Operationen

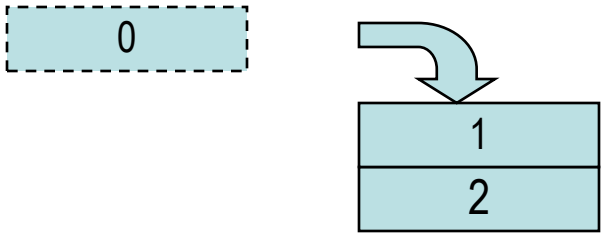
- Enqueue
  - Element am Ende der Queue ablegen
- Front
  - Erstes Element der Queue zugreifen
- Dequeue
  - Erstes Element aus der Queue entfernen
- IsQueueEmpty
  - Test auf leere Queue



Methoden auf Queue

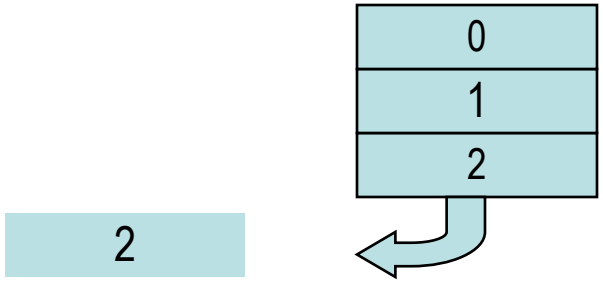
Methode ‘Enqueue’

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



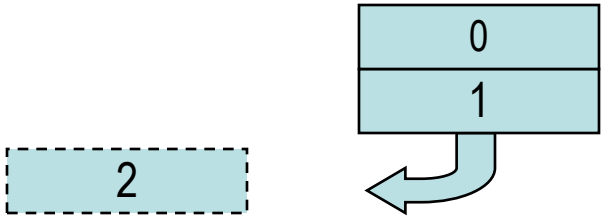
Methode ‘Front’

- Liefert den Inhalte des ersten Elementes der Queue



Methode ‘Dequeue’

- Erstes Element der Queue wird entfernt



Queue als Liste

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)	$\Rightarrow$	?

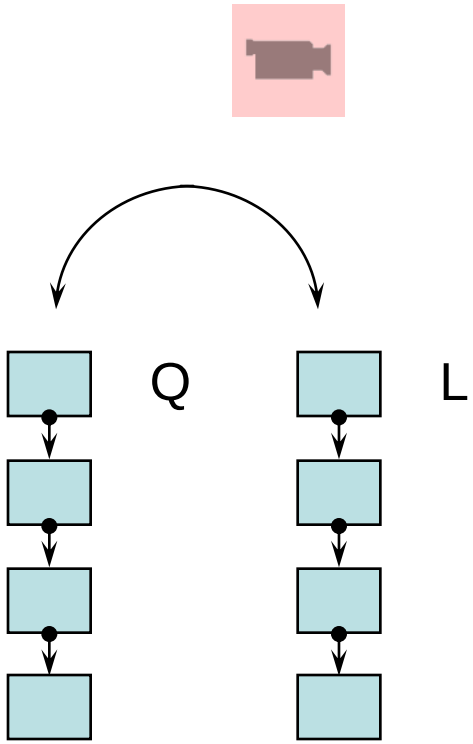
Nicht ganz trivial !

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.

Front(Q)

```
ItemType e;     // Hilfselement
List L;         // Hilfsliste
int n = Q.Length();
for (int i = 1; i <= n - 1; i++) {
  L.Add(Q.FirstElement());
  Q.RemoveFirst();
}
e = Q.FirstElement();
n = L.Length();
for (i = 1; i <= n; i++) {
  Q.Add(L.FirstElement());
  L.RemoveFirst();
}
return e;
```



Dequeue

Dequeue(Q)

```
List L;
int l = Q.Length();
for (int i = 1; i <= n - 1; i++) {
  L.Add(Q.FirstElement());
  Q.RemoveFirst();
}
Q.RemoveFirst();
int n = L.Length();
for (i = 1; i <= n; i++) {
  Q.Add(L.FirstElement());
  L.RemoveFirst();
}
```

Zugriff auf beliebiges Listenelement

Besser: Einführen einer neuen Listenoperation

Methode ‘AccessElement’

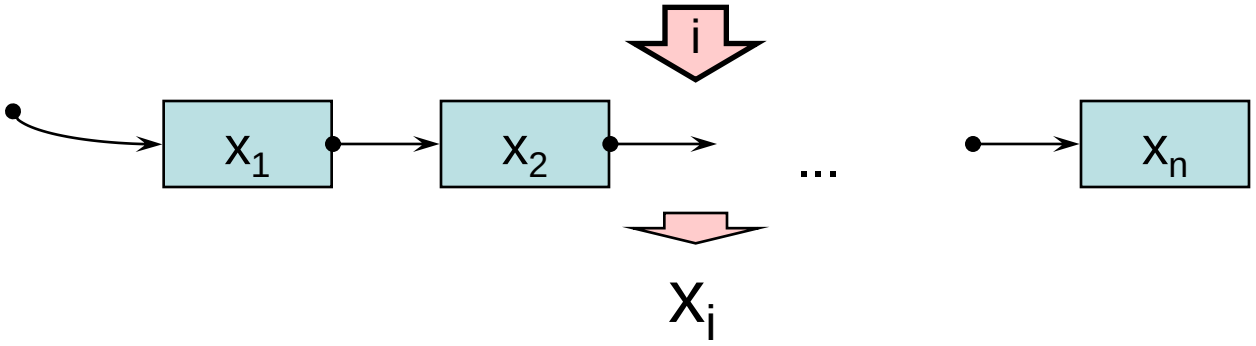
Zugriff auf ein beliebiges Listenelement über die Position in der Liste, d.h.

$L = (x_1, \ldots, x_n), i$

$\Downarrow$

AccessElement(L, i) $\rightarrow x_i$

Fehler falls Position nicht definiert



C++-Methode: **`ItemType AccessElement(position i);`**

Methode AccessElement

```
ItemType List::AccessElement(int pos) {
  Element* act = head;
  int actpos = 1;
  …
  while(actpos < pos) {
   act = act->next;
   actpos++;
  }
  return act->value;
}
```

Queue als Liste (2. Versuch)

Neuerlicher Definitionsansatz mit zusätzlicher Listenoperation etwas einfacher, aber ...

Enqueue(Q,a)	⇒	Add(Q,a)
Front(Q)	⇒	AccessElement(Q,Length(Q))
Dequeue(Q)	⇒	?
		RemoveElement(Q, Length(Q))

Möglichkeit:

Analog zu positionsbezogener Zugriffsfunktion eine positionsbezogene Teilungsfunktion entwerfen, die eine Liste an einer vorgegebener Stelle in 2 Teillisten zerlegt.

Methode ‘RemoveElement’

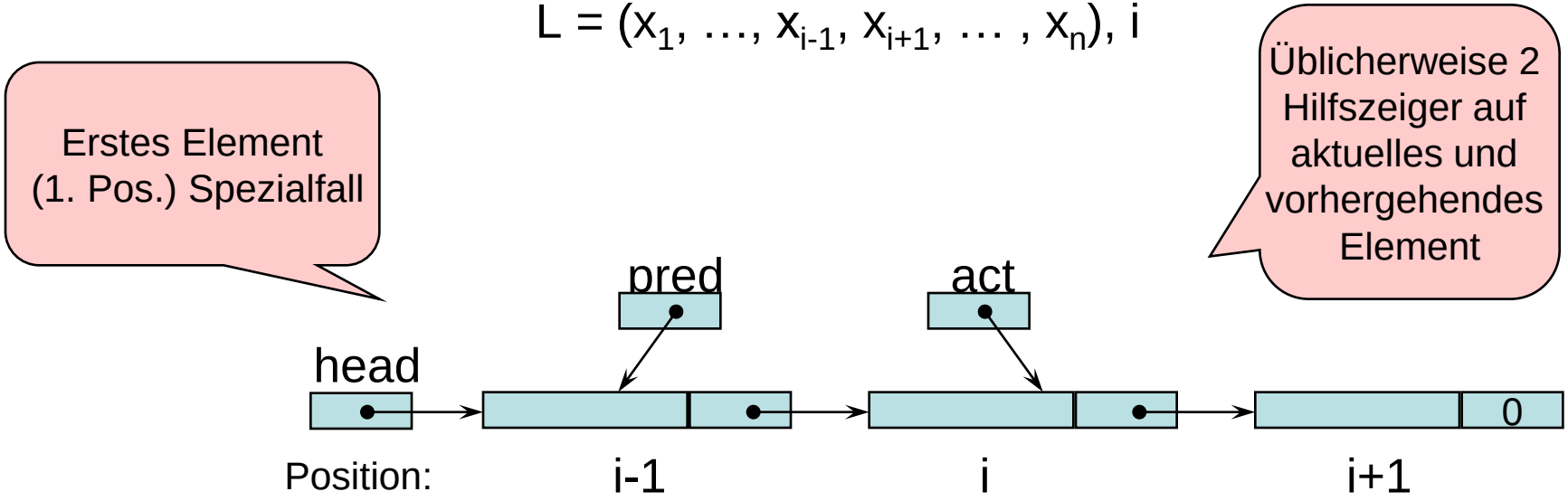
Löschen eines beliebigen Listenelement über die Position in der Liste, d.h.

$$L = (x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n), i$$

$$\text{RemoveElement}(L, i) \rightarrow L$$

Fehler falls Position i nicht definiert

$$L = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n), i$$



Methode RemoveElement

```
void List::RemoveElement(int pos) {
  Element* pred, * act;
  int actpos = 2;
  if(pos == 1) RemoveFirst();
  else {
    pred = head;
    act = head->next;
    while(act != 0 && actpos < pos) {
      pred = act;
      act = act->next;
      actpos++;
    }
    if(act == 0) return;
    pred->next = act->next;
    delete act;
  }
}
```

erstes Element

2 Hilfszeiger

Methode AddElement

Analog auch AddElement mgl.: Einfügen nach beliebiger Stelle

```
void List::AddElement(ItemType a, int pos) {
  Element* pred, * act;
  int actpos = 2;
  if(pos == 1) Add(a);
  else {
   pred = head;
   act = head->next;
   while(act != 0 && actpos < pos) {
          pred = act;
          act = act->next;
          actpos++;
   }
   pred->next = new Element;
   pred->next->value = a;
   pred->next->next = act;
  }
}
```

erstes Element

2 Hilfszeiger

universität wien

C++ Klassen Deklaration (Skizze)

Stack

```
class Stack {
…
public:
        Stack();
        bool Push(ItemType a);
        ItemType Top();
        bool Pop();
        bool IsStackEmpty();
}
```

Queue

```
class Queue {
…
public:
        Queue();
        bool Enqueue(ItemType a);
        ItemType Front();
        bool Dequeue();
        bool IsQueueEmpty();
}
```

Realisierung durch „Komposition“, anderer Ansatz wäre „Vererbung“

Vergleich "unserer" Datenstrukturen

universität wien

- Generelle Unterscheidung zwischen statischer und dynamischer Realisierung
  - statische R.: contigous 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	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	**O(n)**
Add	O(1)	O(1)
FirstElement	O(1)	O(1)
RemoveFirst	O(1)	O(1)
Length	O(1)	**O(n)**
Member	**O(n)**	**O(n)**

	Liste statisch	Liste dynamisch
AccessElement	O(1)	**O(n)**
RemoveElement	**O(n)**	**O(n)**
AddElement	**O(n)**	**O(n)**

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

Aufwandsvergleich "unserer" Stack - Queue Implementationen

universität wien

	Stack statisch	Stack dynamisch
Speicherplatz	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	**O(n)**
Push	O(1)	O(1)
Pop	O(1)	O(1)
Top	O(1)	O(1)
IsStackEmpty	O(1)	O(1)

	Queue statisch	Queue dynamisch
Speicherplatz	**O(n)**	**O(n)**
Konstruktor	O(1)	O(1)
Destruktor	O(1)	**O(n)**
Enqueue	O(1)	O(1)
Dequeue	O(1)	**O(n)**
Front	O(1)	**O(n)**
IsQueueEmpty	O(1)	O(1)

4.5 Spezielle Listen

Doubly Linked List

doppelt verkettet Liste

Circular List

Zirkulär verkettete Liste

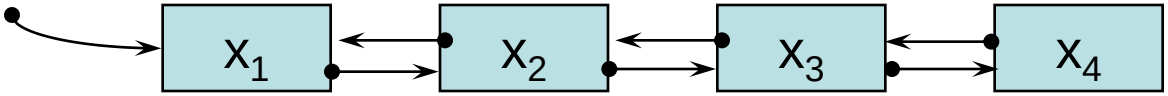
Ordered List

Geordnete Liste

Double Ended List

Doppelköpfige Liste

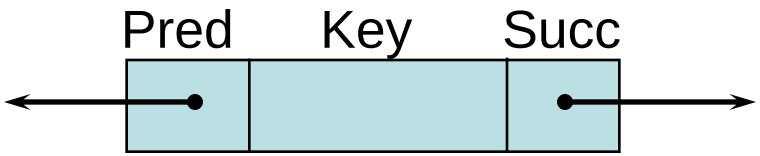
Doppelt verkettete Liste



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

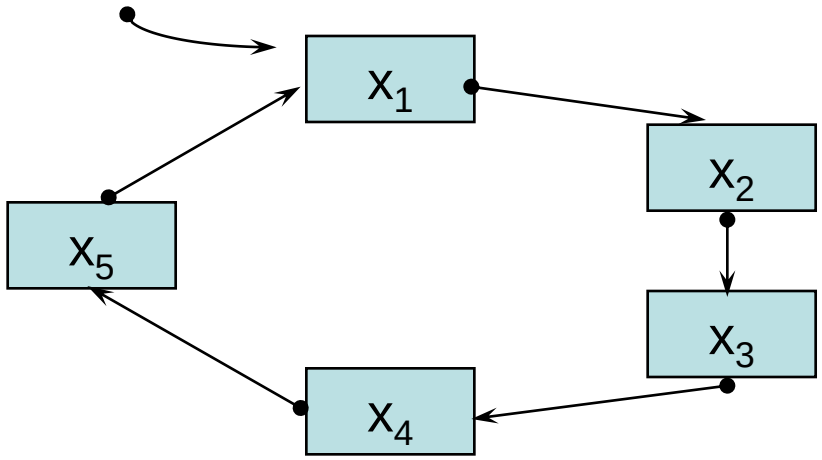
Basis-Operationen einfach

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



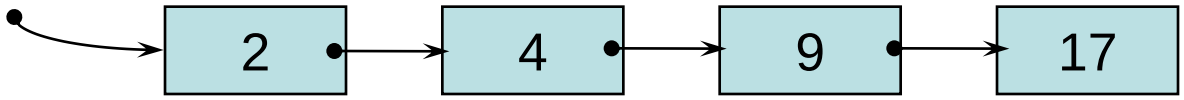
Zirkulär verkettete Liste

- Zeiger des letzten Element verweist wieder auf das erste Element
- Ring Buffer
- Vorsicht beim Eintragen und Löschen des ersten Elementes!



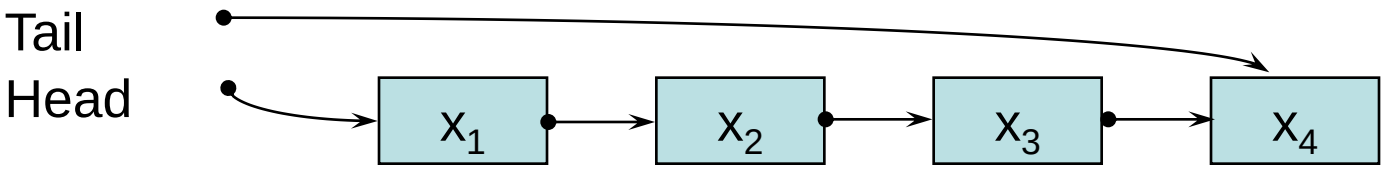
Geordnete Liste

- 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



Was nehmen wir mit?

- Listen
  - Operationen
  - Speicherung
  - Contigous - Scattered memory
- Stack – Queue
- Vergleich
- Spezielle Listen
  - Doubly Linked List
  - Circular List
  - Ordered List
  - Double Ended List