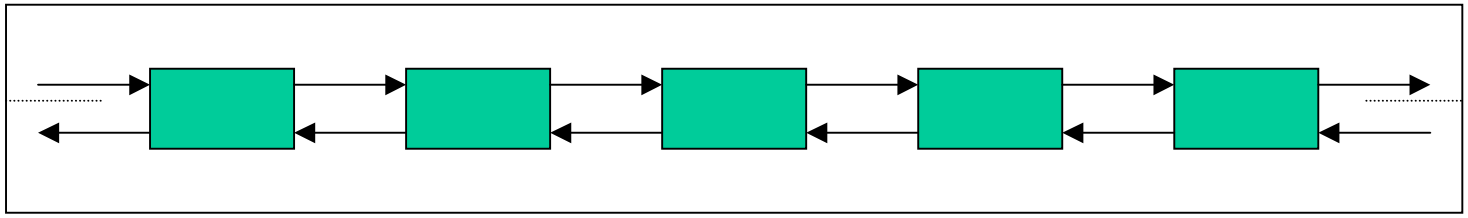
Der Container `list`

Jan Große

Überblick

- Struktur und Eigenschaften
- Listenfunktionen
- Spezielle Listenfunktionen
- Initialisierungsproblem
- Der Container **`slist`**
- Zusammenfassung

Struktur und Eigenschaften



- Doppelt verkettete Liste mit Elementen gleichen Typs
- Jedes Element hat Vorgänger und Nachfolger
- damit ist List ein Reversible Container
- Kein wahlfreier Zugriff
- Einfügen und Entfernen von Elementen in konstanter Zeit

Verwendung

```
#include <list>

namespace std {
    template <class T, class Allocator = allocator<T> >
    class list {
    ...
    };
}

T         - Typ der Elemente
Allocator - Speichermodell
```

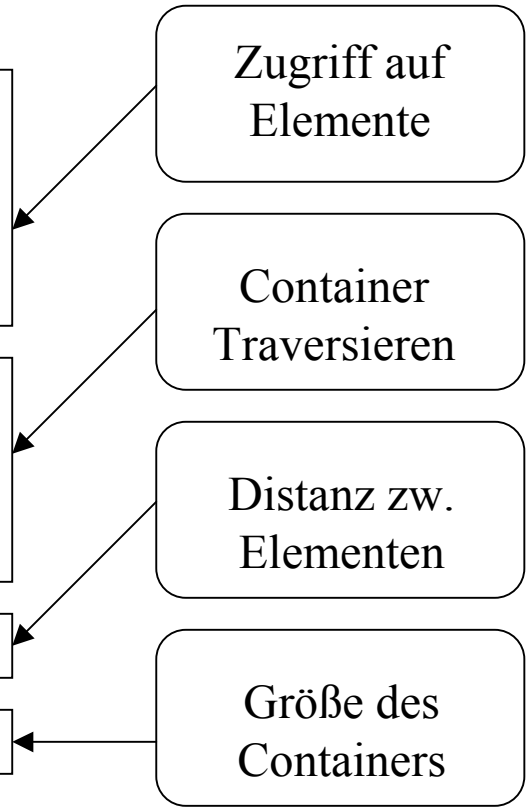
Nested Types

```
template <class T, ...>
class list {
    ...
    typedef ... value_type;       // T
    typedef ... reference;        // T&
    typedef ... const_reference; // const T&
    typedef ... pointer;          // T*
    typedef ... const_pointer;   // const T*

    typedef ... iterator;
    typedef ... const_iterator;
    typedef ... reverse_iterator;
    typedef ... const_reverse_iterator;

    typedef ... difference_type;

    typedef ... size_type;
};
```



Erzeugen von Listen

Ausdruck	Bedeutung
`list<T> m;`	erzeugt eine leere Liste ohne Elemente
`list<T> m1(m2);`	erzeugt Liste **`m1`** als Kopie von Liste **`m2`**
`list<T> m(10);`	erzeugt eine Liste mit 10 Elementen
`list<T> m(10, e);`	erzeugt eine Liste mit 10 Kopien von **`e`**
`list<T> m(anf, end);`	erzeugt eine Liste und initialisiert sie mit Kopien aus **`[anf, end)`**

Erzeugen von Listen (Bsp)

```
#include <list>
#include <vector>

using namespace std;

int main() {

    int n = 1;
    vector<int> v(10);
    vector<int>::iterator i;
    for(i = v.begin(); i != v.end(); ++i, ++n)
        *i = n;

    list<int> l(v.begin()+1, v.end()-5);
}
```

Erzeugt vector v mit 10 Elementen und füllt diese mit Zahlen: 1,2, ..., 10

Erzeugt Liste und initialisiert sie mit Zahlen 2,3,4,5 des Vectors

Vergleichsfunktionen

Ausdruck	Bedeutung
`m1 == m2`	Liefert, ob beide Listen die gleichen Elemente in derselben Reihenfolge enthalten
`m1 != m2`	entspricht `!(m1 == m2)`
`m1 < m2`	? (wenn `m1` kleiner `m2`)
`m1 > m2`	...
`m1 <= m2`	`(m1 < m2) || (m1 == m2)`
`m1 >= m2`	...

Vergleich von Listen (Bsp)

```
list<int> l1, l2, l3;

l1.push_back(1), l2.push_back(1), l3.push_back(1);
l1.push_back(2), l2.push_back(3), l3.push_back(2);
l1.push_back(3), l2.push_back(2), l3.push_back(3);
                                  l3.push_back(4);

l1 != l2; // true
l1 < l2;  // true
l1 < l3;  // true
l2 > l3;  // true
```

Eigenschaften der Vergleiche

- **`m1 < m2`** gdw. (alle Elemente gleich und |**`m1`**| **`<`** |**`m2`**|) oder (für ersten ungleichen Elemente **`a`** aus **`m1`** und **`b`** aus **`m2`** gilt: **`a < b`**)
- Lexikographische Ordnung !
- Äquivalent zu: **`lexicographical_compare()`**
- Garantierte Laufzeit: linear in der Länge der Listen

Abfrage weiterer Listeneigenschaften

- Aktuelle Anzahl von Elementen: **`m.size()`**
- Test ob Liste leer: **`m.empty()`**
- Maximal Anzahl an Elementen: **`m.max_size()`**
- keine Kapazitätsfunktion, da nicht notwendig
- Test auf leere Liste nicht mit **`m.size()==0`**, sondern mit **`m.empty()`**, da hier garantierte Laufzeit O(1)

Zuweisungen

Ausdruck	Bedeutung
`m1 = m2`	weist **m1**alle Elemente von **m2** zu
~~`m.assign(10)`~~	~~weist **m** genau 10 Elemente zu, die per Def-Konstruktor erzeugt~~
`m.assign(10, e)`	weist **m** genau 10 Kopien von **e** zu
`m.assign(anf, end)`	weist **m** Kopien der Elemente aus **[anf, end)** zu
`m1.swap(m2)`	vertauscht die Elemente von **m1** mit denen von **m2** in O(1)
`swap(m1, m2)`	als globale Funktion in O(N)

Josuttis !

Elementzugriff

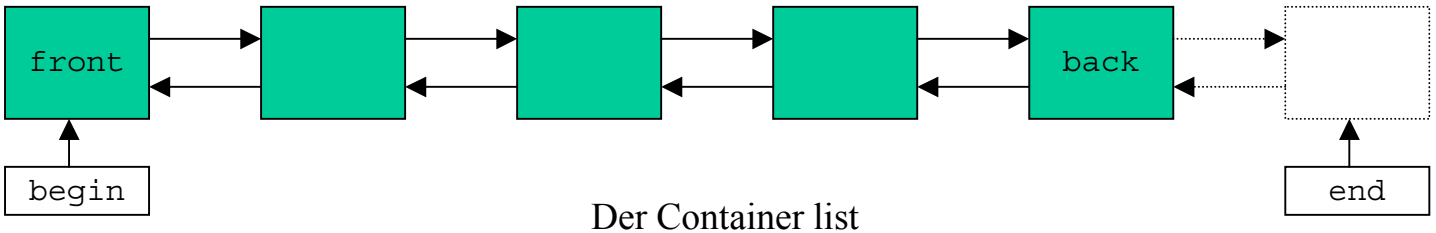
```
reference           front();
reference           back();
const_reference     front() const;
const_reference     back() const;
```

Zugriff über erstes und letztes Element

```
*(l.begin()) == l.front()
*(--(l.end()) == l.back()
```

```
iterator            begin();
iterator            end();
const_iterator      begin() const;
const_iterator      end() const;
```

Zugriff über Iteratoren



Manipulierende Funktionen

Manipulieren am Anfang und Ende der Liste:

```
void push_back(const T& x);
void push_front(const T& x);

void pop_back(void);
void pop_front(void);
```

Einfügen eines Elementes x:
- am Ende der Liste
- am Anfang der Liste

Entfernen:
- des letzten Elementes
- des ersten Elementes

Zeitkomplexität O(1)

Manipulierende Funktionen

Manipulieren an beliebiger Stelle der Liste:

```
1) iterator insert(iterator pos, const T& x);
2) void insert(iterator pos, size_type n, const T& x);
3) void insert(iterator pos, InputIter anf, InputIter end);
```

Eingefügt wird:

1. Kopie von **`x`** vor Position **`pos`** O(1)
2. n Kopien von **`x`** vor Position **`pos`** O(n)
3. Kopien der Elemente in **`[anf, end)`** vor **`pos`** O(n)

Manipulierende Funktionen (Bsp)

```
...
int main()
{
   list<int>   L;
   vector<int> V;

   V.push_back(20), V.push_back(21), V.push_back(22);
   L.push_back(2), L.insert(L.begin(), 1), L.insert(L.end(),3);
   list<int>::iterator old_begin = L.begin();
   L.insert(L.begin(), V.begin(), V.end());

   copy(L.begin(), L.end(), ostream_iterator<int>(cout, ", "));
   // Ausgabe: 20, 21, 22, 1, 2, 3,
   cout << *old_begin() << endl;
   // Ausgabe: ???
}
```

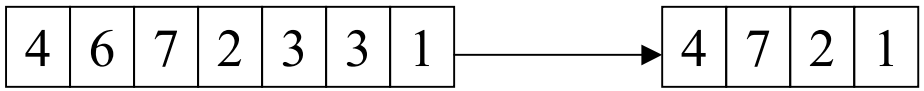
Manipulierende Funktionen

- Beim Einfügen und Löschen von Elementen werden Verweise auf andere Elemente nicht ungültig !
- **`old_begin`** zeigt nach Einfügen auf alten Beginn der Liste, die Ausgabe ist daher **`1`**
- Löschoperationen:
  - **`erase(pos)`** in O(1)
  - **`erase(anf, end)`** in O(n)
  - **`clear()`** in O(n)
- Größe ändern:
  - **`resize(n, T)`**

Spezielle Listenfunktionen

Nochmal Löschen von Listenelementen:

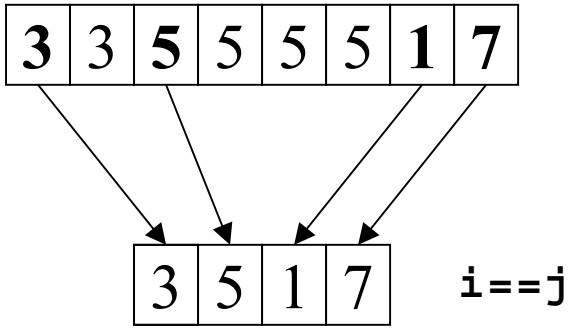
- **`void remove(const T& val);`**
  - entfernt alle Elemente aus **`*this`** die äquivalent sind zu **`val`**
  - Benötigt genau **`size()`** viele Vergleichsoperationen
- **`void remove_if(Predicate Pred)`**
  - entfernt alle Elemente **`e`** aus **`*this`** mit **`Pred(e)==true`**
  - Laufzeit ebenfalls linear
- Beispiel: Entfernen aller durch 3 teilbaren Zahlen
  - **`bool div_by_3(int i) { return (i % 3 == 0);}`**
  - **`l.remove_if(div_by_3);`**



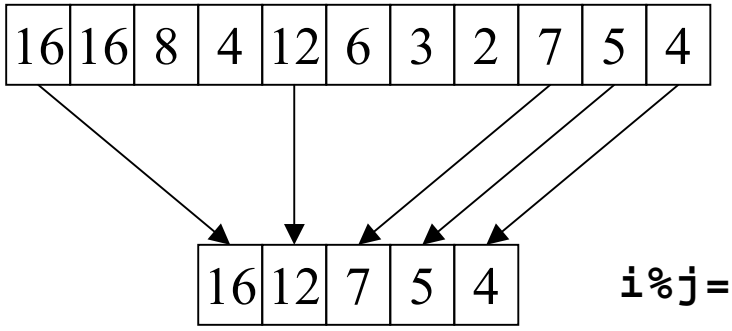
Spezielle Listenfunktionen

Spezielles Löschen von Elementen:

- **`void unique(void)`** Entfernt alle Elemente, die gleich ihrem Vorgänger sind
- **`void unique(BinaryPredicate p)`** Entfernt alle Elemente (außer dem ersten) in einer konsekutiven Teilliste von äquivalenten Elemente (**`*i`** und **`*j`** sind äquivalent gdw. **`p(*i,*j)`** wahr ist.)



`i==j`

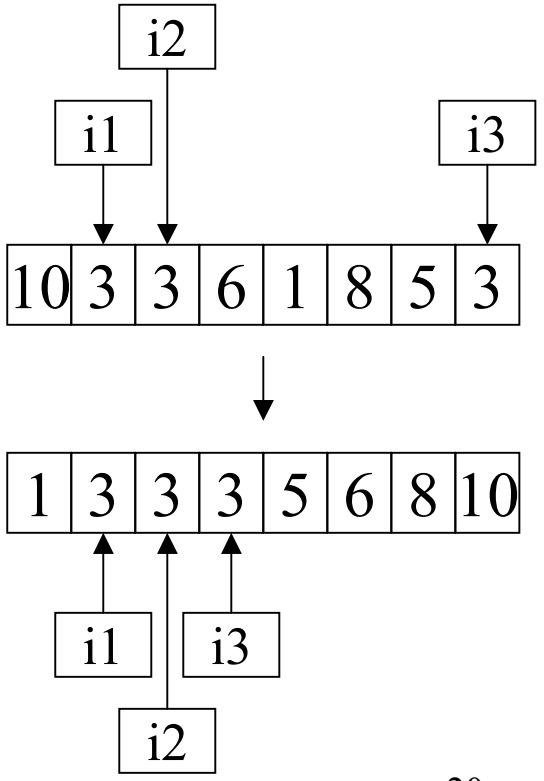


`i%j==0`

Spezielle Listenfunktionen

Sortieren von Listen:

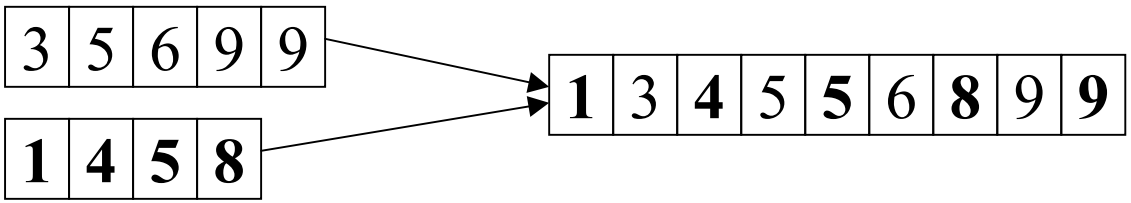
- **`void sort(void)`**
  - sortiert bezüglich des **`<`** Operators
  - relative Ordnung äquivalenter Elemente bleibt gleich
  - Iteratoren zeigen danach auf selben Elemente
- **`void sort(StrictWeakOrdering Comp)`**
  - sortiert bezüglich **`Comp`**
  - sonst wie **`sort(void)`**
- Sortierung erfolgt in O(N logN)



Spezielle Listenfunktionen

Verschmelzen von vorsortierten Listen:

- **`void merge(list<T>& x);`**
  - entfernt alle Elemente aus **`x`** und fügt sie in **`*this`** bezüglich **`<`** Operator ein
  - **`x`** und **`*this`** müssen unterschiedlich sein
  - ist stabil: ein Element aus **`*this`** äquivalent zu einem aus **`x`** geht diesem in verschmolzener Liste voraus
  - lineare Laufzeit (höchstens **`size()+x.size()-1`** Vergleiche)
- **`void merge(list<T>& x, BinaryPredicate Pred)`**
  - entfernt alle Elemente aus **`x`** und fügt sie in **`*this`** bezüglich **`Pred`** ein
  - sonst wie **`merge(list<T>& x)`**



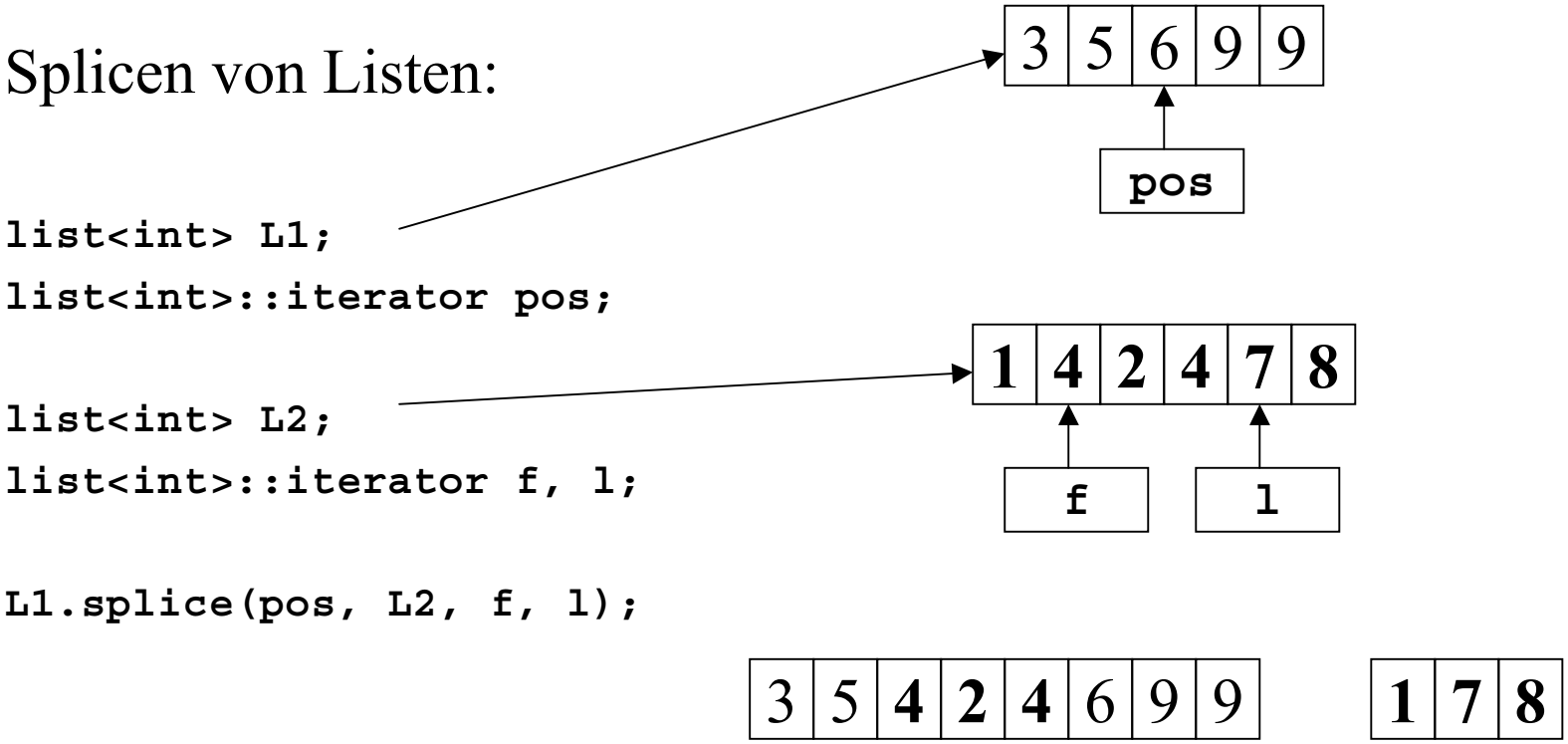
Spezielle Listenfunktionen

Umketten von Listen:

- **`void splice(iterator pos, list<T>& x);`**
  - alle Elemente aus **`x`** werden entfernt und vor **`pos`** in **`*this`** eingefügt
  - **`*this`** und **`x`** müssen verschieden sein
- **`void splice(iterator pos, list<T>& x, iterator i);`**
  - entfernt Element **`*i`** aus **`x`** und hängt es vor **`pos`** in **`*this`** ein
  - **`*this`** und **`x`** können identisch sein
- **`void splice(iter pos, list<T>& x, iter f, iter l);`**
  - entfernt alle Elemente in **`[f,l)`** aus **`x`** und fügt diese vor **`pos`** ein
  - **`*this`** und **`x`** können identisch sein
  - falls identisch dann darf **`pos`** nicht in **`[f,l)`** sein (sonst wird Bereich nur entfernt)
- Iteratoren bleiben gültig
- Lauzeit O(1)

Spezielle Listenfunktionen

Splicen von Listen:



Initialisierungsproblem

Wollen Messdaten in Liste einlesen:

```
ifstream dataFile("ints.dat");

list<int> data(istream_iterator<int>(dataFile),
               istream_iterator<int>());
```

- kompiliert
- liest aber keine Zahlen ein
- erzeugt gar keine Liste

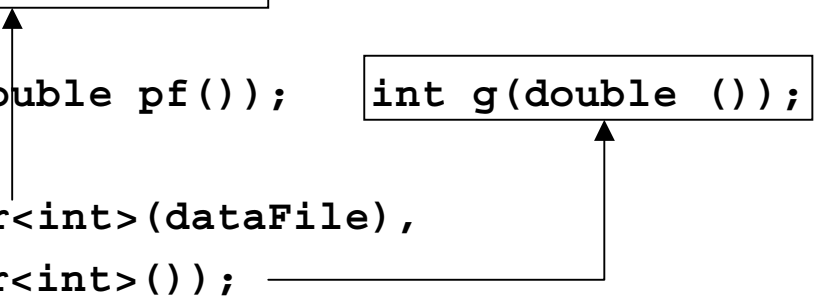
Initialisierungsproblem

Grund:

```
int f(double d);        int f(double (d));     int f(double);

int g(double (*pf)()); int g(double pf());     int g(double ());

list<int> data(istream_iterator<int>(dataFile),
               istream_iterator<int>());
```



- Damit haben wir Funktion **`data`** deklariert mit Rückgabetyp **`list<int>`** die zwei Parameter hat
- Erster Parameter hat Namen **`dataFile`** ist vom Typ **`istream_iterator<int>`**
- Zweiter (namenloser) Parameter ist Zeiger auf Funktion mit Rückgabetyp **`istream_iterator<int>`**

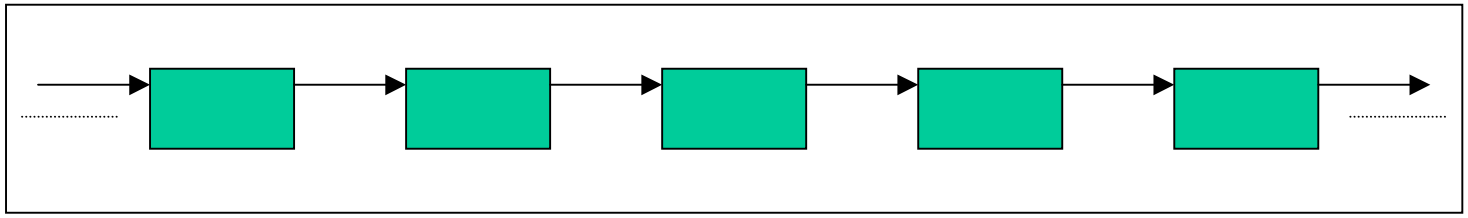
Initialisierungsproblem

Lösung: Vermeidung von anonymen Objekten in Deklaration

```
istream_iterator<int> dataBegin(dataFile);
istream_iterator<int> dataEnd;

list<int> data(dataBegin, dataEnd);
```

Der Container `slist`



- Ist einfach verkettete Liste (jedes Element hat Nachfolger)
- Forward Container
- Einfügen und Löschen von Elementen in O(1)
- Nicht Teil des C++ Standard
- Nicht in HP-Implementation (BCC 5.5) enthalten
- Aber in SGI-Implementation (G++ 2.95.3) enthalten in Header `slist` definiert

Spezielle Funktionen

- **`iterator previous(iterator pos);`**
  - gibt Vorgänger von **`pos`** zurück
  - Laufzeit: linear in der Anzahl der Vorgänger
- **`... insert_after(iterator pos, ...);`**
  - Einfüge-Operationen analog zu **`list`**
  - es wird aber nach **`pos`** eingefügt
- **`iterator erase_after(...)`**
- **`void splice_after(iterator pos, iterator prev)`**
  - entfernt das Element, welches **`prev`** folgt und hängt es nach **`pos`** ein O(1)

Zusammenfassung

- Listen geeignet für Anwendungen mit vielen Einfüge- und Löschoperationen
- Iteratoren bleiben nach Operationen gültig und verweisen auf dieselben Elemente wie davor
- Spezielle Listenfunktionen (`splice, sort,...`)
- Einfach verkettete Liste `slist`