5 Die Standard Template Library *(STL)*

5.1 Container
5.2 Iteratoren (*)
5.3 Funktionsobjekte
5.4 Algorithmen

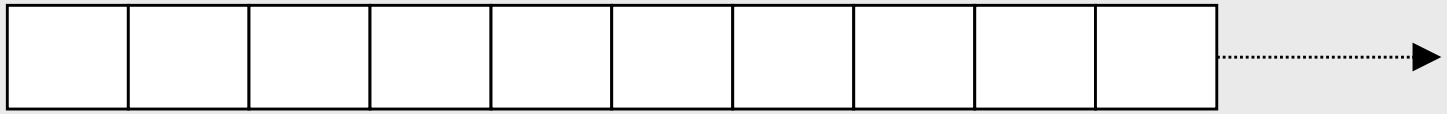
(*) Auch in dieser Datei

Lehrbuch: 6.4
Kompendium, 3. Auflage: 8.12, 11.5
Kompendium, 4. Auflage: 11.5

Container der STL - Sequentielle Container

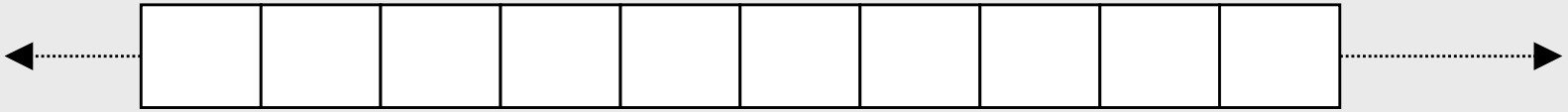
Sequentielle Container sind geordnete Container, in denen jedes Element eine bestimmte Position besitzt, der durch den Zeitpunkt und den Ort des Einfügens bestimmt ist (Josuttis [96, S.45ff]).

Vektor:

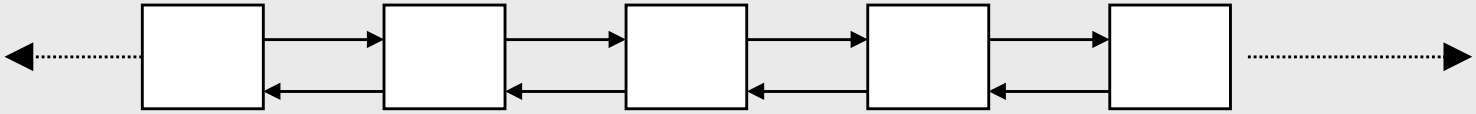


Array, C-Array:

Deque:



Liste:



Der Container *Vektor (vector)*

```
#include <iostream>
#include <vector>
using namespace std;

vector<int>  cont;

// Elemente 1 bis 6 jeweils
// hinten anfuegen  */
for (int i=1; i<7; ++i) {
   cont. push_back(i);
}

for (int j=0;j<cont.size();++j) {
   cout << cont [j] << ' ';
}
cout << endl;
```

Die Elemente werden in einem dynamischen Feld verwaltet, d.h. wahlfreier Zugriff ist über einen Index-Operator ([]) möglich.

Das Einfügen und Löschen von Elementen am Ende geht optimal schnell.

Einfügen und Löschen am Anfang und in der Mitte ist dagegen langsam.

Der Container Vektor (2)

```
vector<int> cont;          // Es wird ein leerer Container von Typ vector für
                           // int-Elemente angelegt.

cont. push_back(i);        // push_back fügt das Argument am Ende des
                           // Containers ein.

int anzahl = cont. size(); // Die Methode size() gibt die Anzahl der
                           // Elemente im Container zurück.

cont [i];                  // Der Index-Operator [] wird zum wahlfreien
                           // Zugriff (lesend und schreibend) auf
                           // das Element an der Position i verwendet.

cont. begin();             // liefert Verweis auf das erste Element
                           // Mit cont.begin()-> … kann man damit arbeiten
```

Der Container Deque (sprich Deck)

```
#include <deque>
using namespace std;

deque<int>  cont;  //  Container-Deque für int

// Elemente 1 bis 6 jeweils vorne und  hinten anfuegen
for (int i=; i<7; ++i) {
   cont. push_front(i);
   cont. push_back(i);
}

cout <<  "Alle Elemente 6,5 ... 1,1, 2 ... 5, 6 ausgeben" << endl;
for (int j=0; j<cont.size(); ++j) {
   cout << cont[j] << ' ';
}
```

Die Methode cont. push_front(i) steht beim Typ vector<> nicht zur Verfügung

Der Container Liste

```
#include <list>
using namespace std;

list<float>  cont;  //  Container list für float

// Elemente jeweils vorne und hinten anfügen
for (int i=1; i<4; ++i) {
   cont. push_front(i*2.2);
   cont. push_back(i*1.1);
}

// Jedes Element ausgeben und danach entfernen
while (! cont. empty()) {
   cout << cont.front() << ' ';
   cont. pop_front();
}
```

Der Container Liste (2)

Der Container Liste ist als doppelt verkettete Liste implementiert, d.h. jedes Element verweist auf den Vorgänger und den Nachfolger. Der direkte Zugriff auf ein beliebiges Element ist damit nicht mehr in konstanter Zeit möglich, jedoch ist das Einfügen an beliebiger Position in etwa gleich aufwändig, vorausgesetzt sie muss nicht mehr gesucht werden.

```
cont.empty()          // ergibt true, wenn die Liste leer ist.

cont.front()          // liefert das erste Element der Menge, ohne es
                      // zu entfernen.
pop_front()           // entfernt das erste Element liefert aber void.
// Entsprechend gibt es:
cont.back(), cont.pop_back(), push_front(), push_back() ...

cont. begin();        // liefert Verweis auf das erste Element
                      // Mit cont.begin()-> … kann man damit arbeiten
```

Der Container array

```
#include <stdexcept>
#include <iostream>
#include <array>
using namespace std;

int main() {
array<int, 7>  a;
try {
    a.at(8) = 44;
    }
catch (const out_of_range& e)    {
      cerr << "out of range: " << e.what() << endl;
   }
//a[8] = 22;  // Speicherschutzverletztung
}
```

unique

```
void apply2()
{ // use func to eliminate duplicates in a range
    array<int, 7> data = { 1, 1, 2, 3, 3, 3, 5 };
    auto end = unique(data.begin(), data.end());
    copy(data.begin(), end,
         ostream_iterator<int>(cout, " "));
    cout << '\n';
}
void main(){ apply2();}
```

Ausgabe:
1 2 3 5

Removes all duplicates from every consecutive group of equivalent elements in the range [first,last)

unique

```
void demo2_2(){
    vector<int> cont = { 1, 3, 3, 3, 5, 3, 1 };
    auto newEnd = unique(cont.begin(), cont.end());
    copy(cont.begin(), newEnd,
            ostream_iterator<int>(cout, ";"));
    cout << endl;
}
```

Ausgabe:
1; 3; 5; 3; 1;

Aufgabe 1 zum Code auf Homepage

In der Datei Aufgabe1.cxx finden Sie:

```
void PrintBeforeUniqueAfterVector(vector<int>& cont)
{
   PrintTo(cout, cont.begin(), cont.end(), "Mit vector: before: ");
   auto newEnd = cont.end();
   // auto newEnd = Aufruf von unique um Dubletten aus cont zu entfernen
   cout << endl;
   PrintTo(cout, cont.begin(), newEnd, "after:  ");
   cout << endl;
}
```

Implementieren Sie die grüne Zeile, sodass der Container cont keine Dubletten mehr enthält. „auto newEnd …. vorher entfernen“

Entsprechend für array und deque.

Verwenden Sie cont.end() anstatt newEnd im zweiten Aufruf von PrintTo. Versuchen Sie das Ergebnis zu erklären!

Aufgabe 2 zum Code auf Homepage

In der Datei Aufgabe1.cxx haben Sie nun drei fast identische Funktionen implementiert

```
void PrintBeforeUniqueAfterVector(vector<int>& cont) { ... }

void PrintBeforeUniqueAfterDeque(deque<int>& cont) { ... }

void PrintBeforeUniqueAfterArray(array<int, 7>& cont) { ... }
```

Ersetzen Sie diese drei Funktionen durch **eine einzige** Template-Funktion.

void PrintBeforeUniqueAfterTemp(/* */)

Probieren Sie auch list<int> und set<int> aus.

Aufgabe 2 zum Code auf Homepage (2)

```
/* entfernen Sie den Kommentar vor der folgenden Zeile und
   bringen Sie das Programm zum Laufen. An den Stellen
   mit ... ist auf jeden Fall Ihr Input erforderlich.
   Sie koennen von Aufgabe 1 die vector, array, deque
   Loesung uebernehmen
   */
//#define AUFGABE2
#ifdef AUFGABE2
/* ...  */
void PrintBeforeUniqueAfterTemp(/*... */)
void aufgabe2()
{
    deque<int> dequeInt = { 1, 3, 3, 3, 5, 5, 7 };
    PrintBeforeUniqueAfterTemp(dequeInt);
    array<int, 7> arr = { 1, 3, 3, 3, 5, 5, 7 };
    PrintBeforeUniqueAfterTemp(arr);
    vector<int> vec = { 1, 3, 3, 3, 5, 5, 7 };
    PrintBeforeUniqueAfterTemp(vec);
}
```

Aufgabe (3) Schablonen-Algorithmus myUnique

Implementieren Sie einen eigenen Schablonen-Algorithmus myUnique der aufeinanderfolgende doppelten Elemente aus einem Container entfernt.

```
// andere Implemenierung für unique
/* Im Bereich zwischen anf und end werden alle doppelten
   Elemente ueberschrieben. Doppelte bedeutet,
   dass die Funktion f auf zwei Bereichselemente
   angewandt true liefert.
   Rueckgabewert ist ein Iterator, der ein Element
   hinter das neue Ende verweist.*/
template <typename Iter, typename Func>
Iter myUnique(Iter anf, Iter end, Func f){
```

Erweitern Sie Ihre Implementierung, sodass nun wirklich ALLE Dubletten (nicht nur aufeinanderfolgende) entfernt werden.

unique als Algorithmus und nicht als Elementfunktion

```
vector<int> cont = { 1, 3, 3, 3, 5, 5, 7 };
// output is 1, 3, 3, 3, 5, 5, 7
PrintTo(cout, cont.begin(), cont.end(), "\nbefore: ");
auto newEnd = unique(cont.begin(), cont.end());
// output is 1, 3, 5, 7, 5, 5, 7
PrintTo(cout, cont.begin(), cont.end(), "\nafter1: ");
// output is 1, 3, 5, 7
PrintTo(cout, cont.begin(), newEnd, "\nafter2: ");
cout << "\n";
```

Siehe als Erklärung: Algorithmus remove versus Methode erase im Foliensatz STL-BeispielLieferantKomponente-Teil2

Aufgabe Schablonen-Algorithmus myUnique (2)

Zum Testen können Sie verwenden:

```
void applyTest() {
    array<int, 7> test = { 1, 1, 2, 3, 3, 3, 5 };
    // Alle Doubletten entfernen
    auto end = myUnique(test.begin(), test.end(),
                        justEqual);
    copy(test.begin(), end,
         ostream_iterator<int>(cout, ";"));
    cout << '\n';
    // Alle Doubletten entfernen, aber nur  wenn < 3
    array<int, 7> test2 = { 1, 1, 2, 3, 3, 3, 5 };
    auto end2 = myUnique(test2.begin(), test2.end(),
                         equalAndLessThan3);
    copy(test2.begin(), end2,
         ostream_iterator<int>(cout, ";"));
    cout << '\n';
}
```

```
bool equalAndLessThan3(int al, int ar) {
    return al == ar && al < 3;
}
bool justEqual(int al, int ar) {
    return al == ar;
}
```

```
1;2;3;5;
1;2;3;3;3;5;
```

Etwas allgemeiner

```
#include <array>
#include <iostream>
#include <functional>
using namespace std;

template <class Func>
void apply(Func func)
{ // use func to eliminate duplicates in a range
    array<int, 7> data = { 1, 1, 2, 3, 3, 3, 5 };
    auto end = unique(data.begin(), data.end(), func);
    copy(data.begin(), end, ostream_iterator<int>(cout, ",,"));
    cout << '\n';
}
void binaryNegate() {
    // use predicate equal_to to eliminate duplicates in a range
    apply(binary_negate<not_equal_to<int> >(not_equal_to<int>()));
    apply(equal_to<int>());  // einfacher ist natürlich
}
void main(){ binaryNegate();}
```

Ausgabe:
1,,2,,3,,5,,
1,,2,,3,,5,,

Der Container array (3)

```
typedef int elt;
typedef array<elt, 7> arr;
typedef arr::iterator arr_it;

template <class Func>
void apply(Func func)
  { // use func to eliminate duplicates in a range
  arr data = { 1, 1, 2, 3, 3, 3, 5 };
  arr_it end = unique(data.begin(), data.end(), func);
  copy(data.begin(), end, ostream_iterator<elt>(cout, " "));
  cout << '\n';
  }
void binaryNegate()   {
            // use predicate equal_to to eliminate duplicates in a range
  apply(binary_negate<not_equal_to<int> >(not_equal_to<int>()) );
  apply(equal_to<int>()) ;  // einfacher ist natürlich
  }
```

Ausgabe:
1 2 3 5
1, 2, 3, 5

Etwas allgemeiner

```
template <class Func>
void apply(Func func)
{ // use func to eliminate duplicates in a range
   array<int, 7> data = { 1, 1, 2, 3, 3, 3, 5 };
   auto end = unique(data.begin(), data.end(), func);
   copy(data.begin(), end, ostream_iterator<int>(cout, ",,"));
   cout << '\n';
}
```

Ausgabe:
1,,2,,3,,3,,3,,5,,

```
template<class T>
struct MyEqualTo
   : public binary_function<T, T, bool>
{  // functor for operator==
   bool operator()(const T& al, const T& ar) const
   {  // apply operator== to operands
      return (al == ar && al<3);
   }
};
void main(){  apply(MyEqualTo<int>()); }
```

Assoziative Container

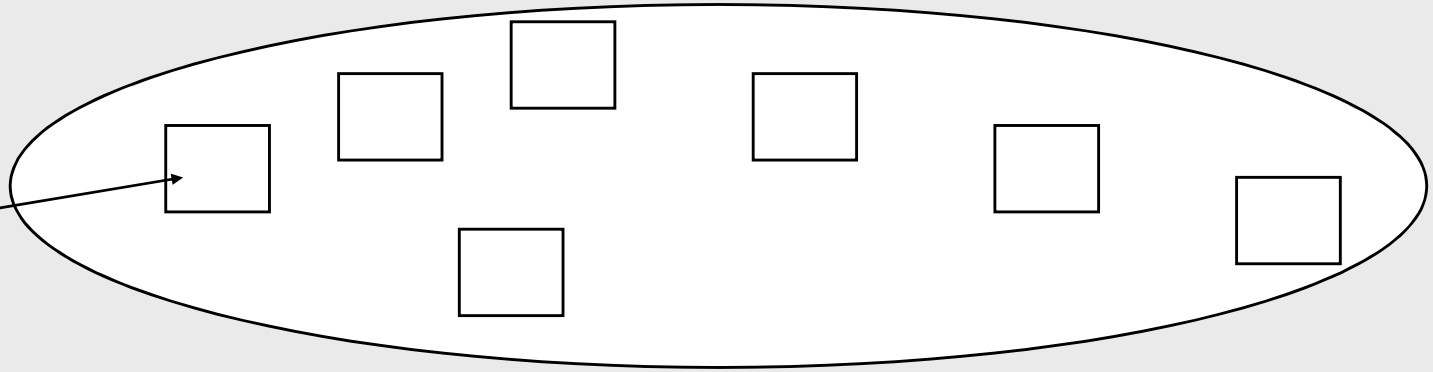
Set / Multiset
Map / Multimap
unordered_set (Hash-Tabelle)

Container der STL - Assoziative Container

Assoziative Container sind sortierte Container, bei denen die Position der Elemente durch ein Sortierkriterium bestimmt wird. Sie sind nicht speziell zum Sortieren von Elementen gedacht, sondern dienen vor allem zum schnellen Auffinden und Suchen von Elementen.

Set / Multiset:

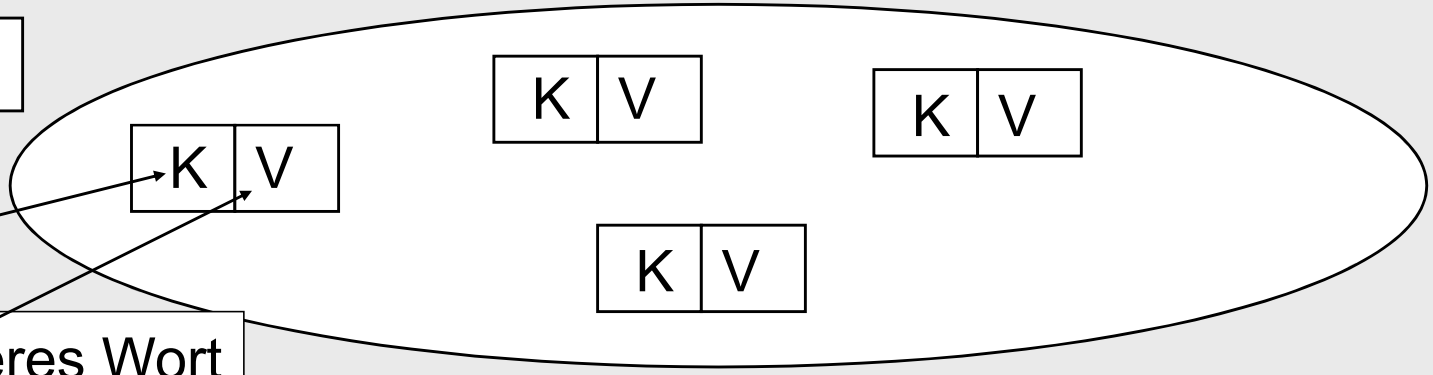
z.B. Zahlen, Personen, **...**



Map / Multimap:

Key, z.B. Wort

Value, z.B. anderes Wort



Container Set / Multiset

```
set<char> cont;
// alle Zeichen von HartmutHelmke einfügen
cont.insert('H'); cont.insert('a'); cont.insert('r'); ...
cont.insert('m'); cont.insert('k'); cont.insert('e');
// Ausgabe in der Sortierreihenfolge
set<char>::const_iterator iter;
for (iter = cont.begin(); iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
// Ausgabe ist  „ H a e k l m r t u “
```

Container Set / Multiset (2)

Bei der Definition bzw. Deklaration eines Sets kann auch eine Sortierfunktion übergeben wird. Wird keine übergeben, wird als Default ***Typ::operator<*** angenommen.

Um die umgekehrte Sortierreihenfolge zu verwenden, kann ***greater<T>*** verwendet werden, z.B

```
set<int, greater<int> >intSet;
```

Das vordefinierte Funktionsobjekt *greater* wird noch genauer besprochen.

Die Methode ***insert*** dient zum Einfügen eines Elementes in die Menge. Es wird dann an die richtige Stelle einsortiert.

Container Set / Multiset (3)

```
multiset<char> cont;
// alle Zeichen von HartmutHelmke einfügen
cont.insert('H'); cont.insert('a'); cont.insert('r'); ...
. . .
cont.insert('m'); cont.insert('k'); cont.insert(,H');
// Ausgabe in der Sortierreihenfolge:
multiset<char>::const_iterator iter;
for (iter = cont.begin(); iter != cont.end(); ++iter) {
   cout << *iter << ' ';
}
// Ausgabe ist  „ H H a e e k l m m r t t u “
```

Set / Mulitsets find

Die Elementfunktion find ermittelt, ob das Argument in der Menge enthalten ist.

cont.find('c‘) ermittelt, ob das Zeichen (bzw. was immer sonst in der Menge abgelegt ist) in der Menge vorhanden ist.
Falls es gefunden wird, verweist der Rückgabewert (ein Iterator) auf dieses Zeichen, sonst auf end():

```
if (cont.find(‘c‘) != cont.end() ) {
   cout << “Das Zeichen ist in der Menge vorhanden.“;
}
```

```
find(cont.begin(), cont.end(),‘c‘);
```

Map / Multimap

```
#include <string>
#include <map>
using namespace std;

int main() {

  // Datentyp des Containers:
 typedef
 map<string, float, greater<string> >
                                StringFloatMap;
  StringFloatMap cont;
  cont["Meier"] = 8400.0f;
  cont["Schulz"] = 4600.0f;
  cont["Kohl"] = 28800.67f;
  cont["Eser"] = 147.4f;
```

```
  // Ausgabe in der Sortierreihenfolge
  StringFloatMap ::const_iterator iter;
  for (iter = cont.begin();
       iter != cont.end(); ++iter) {
    cout << "Key: " << (*iter).first << ' '
         << "Value: " << (*iter).second
         << endl;
  }
}
```

Ausgabe des Programmes:

```
Key: Schulz Value: 4600
Key: Meier Value: 8400
Key: Kohl Value: 28800.7
Key: Eser Value: 147.4
```

Map (2)

Bei Maps ist ein Element ein Schlüssel/Werte-Paar, es wird zusammen mit dem optionalen Sortierkriterium in der Deklaration festgelegt:

```
typedef  map<string, float, greater<string> > T_StrFlMap;
```

Zum Einfügen / Zugriff kann der Index-Operator verwendet werden:

```
cont["Eser"] = 147.4f;
cout << cont["Eser"];
cout << cont[“Stadler"];
```

Wenn zum Index kein Element vorhanden ist, wird eines (mit dem Default-Konstruktor) von Value eingetragen. Es gibt somit keinen falschen Schlüssel.

Map / Pair

Der Operator * liefert eine Struktur pair.

```
struct pair {
   typedef T1 first_type;
   typedef T2 second_type;

   T1 first;
   T2 second;
   pair() : first(T1()), second(T2()) {}
   pair(const _1& a, const T2& b) : first(a), second(b) {}
};
```

Mit (*iter).first wird also auf den Schlüssel zugegriffen.
Mit (*iter).second erhält man den Wert.

```
void f() {
   map<string, double> d;
   d["Auto"] = 16.4;
   auto iter = d.find("Auto");
   if (iter != d.end()) {
      iter->second = 56.7;
      //d["Auto"] = 56.7; // This is not efficient to
                          // search again, what is already found
   }
}
```

Multimap

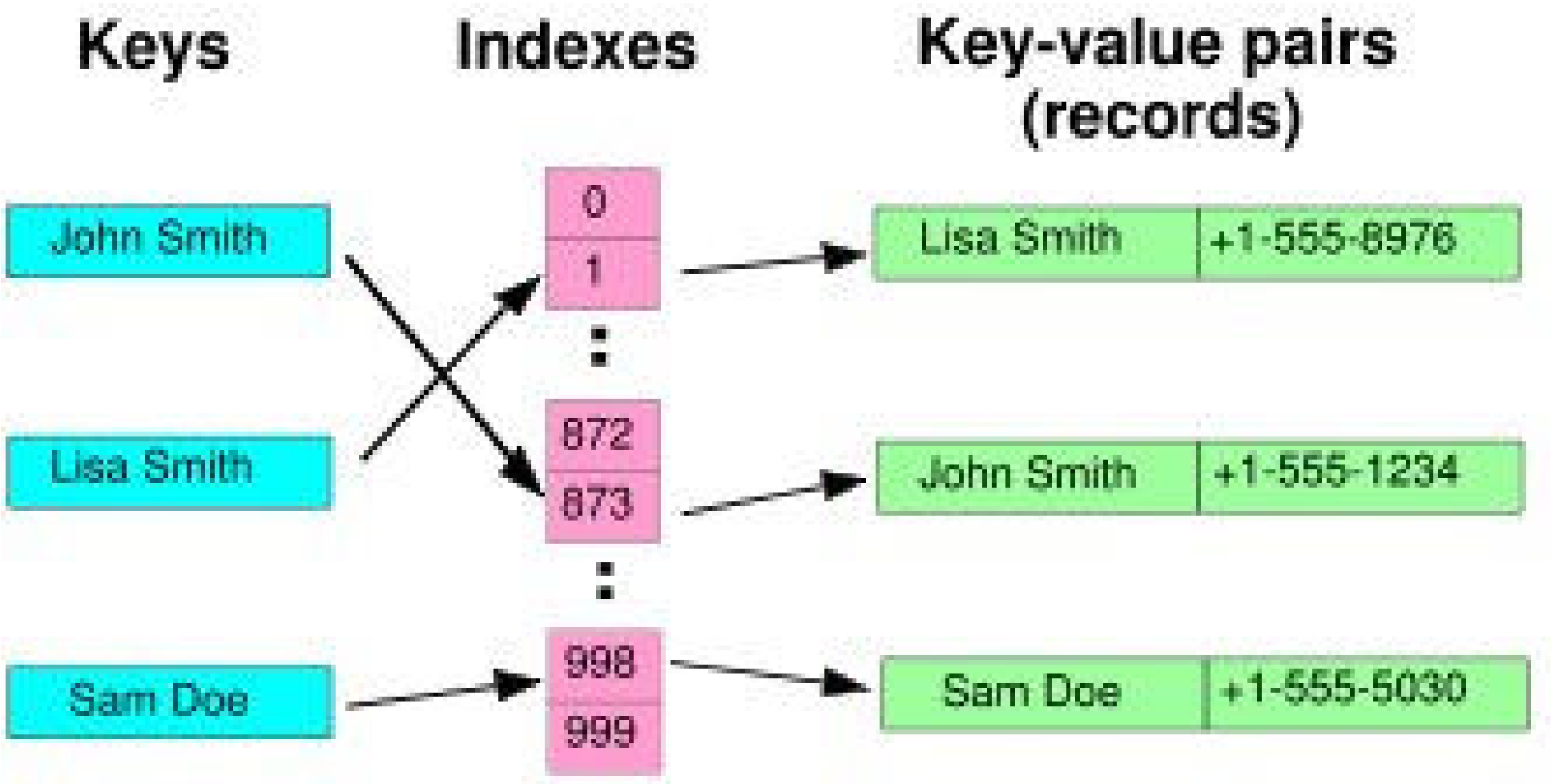
Entsprechend dem Unterschied vom set zu multiset ist der Unterschied von map und multimap.

Iterieren durch eine Multimap // equal_range

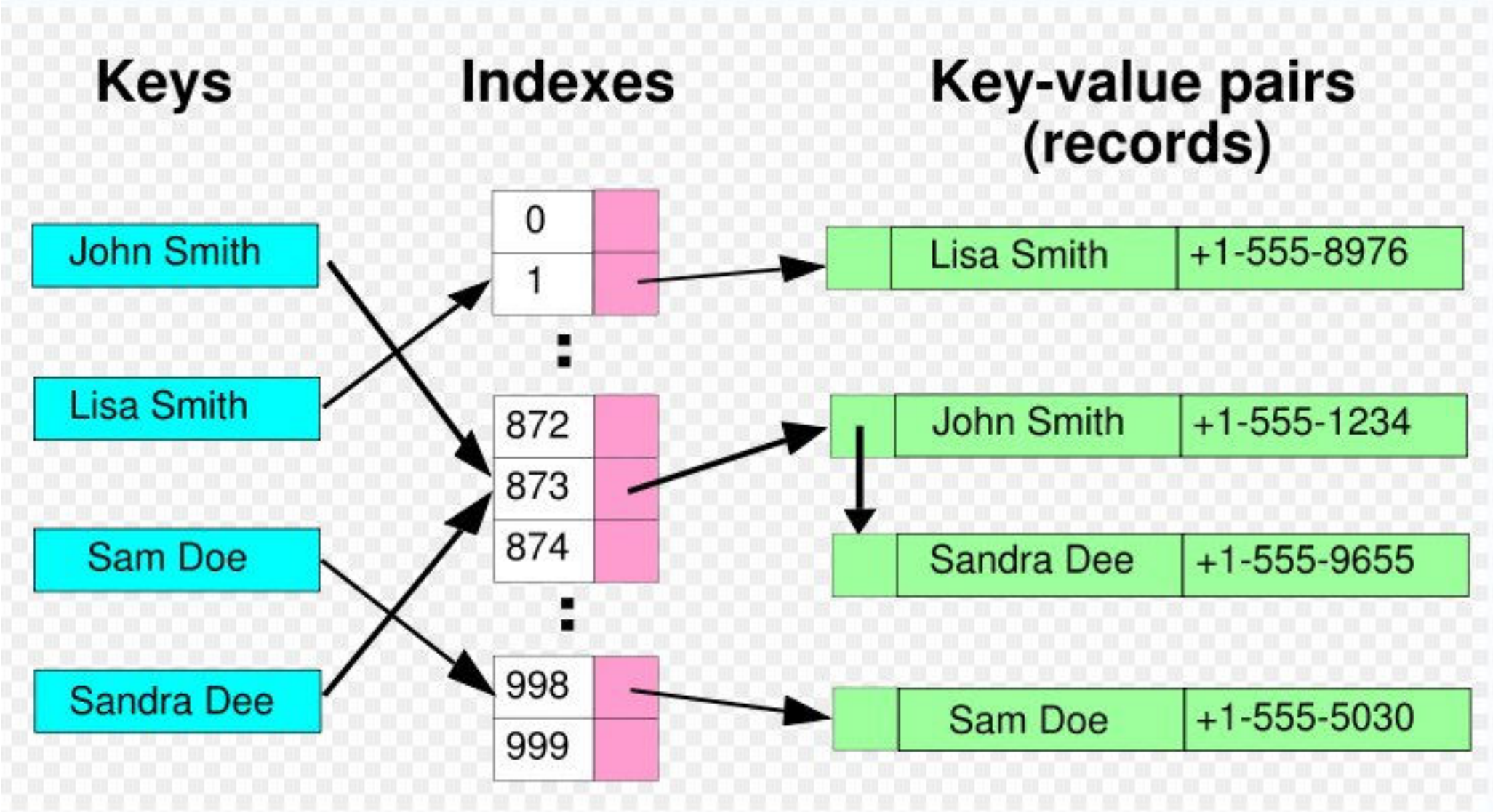
```
void f1() {
   multimap<string, double> d;
   d.insert(pair<string, double>("Auto", 56.3));
   d.insert(pair<string, double>("Auto", 41.3));

 auto iterN = d.find("Auto"); // nur ein Iterator
 auto iterM = d.equal_range("Auto");
      // liefert ein pair von Iteratoren (das erste
      // „Auto“ und ein Element hinter dem letzten „Auto“
   while (iterM.first != iterM.second) {
      iterM.first->second = 56.9;
//      iterM.first->first = "Meier"; // would destroy
                    // the binary_tree in the background
                    // Therefore, syntax error
      ++iterM.first;
   }
}
```

Hash-Tabellen

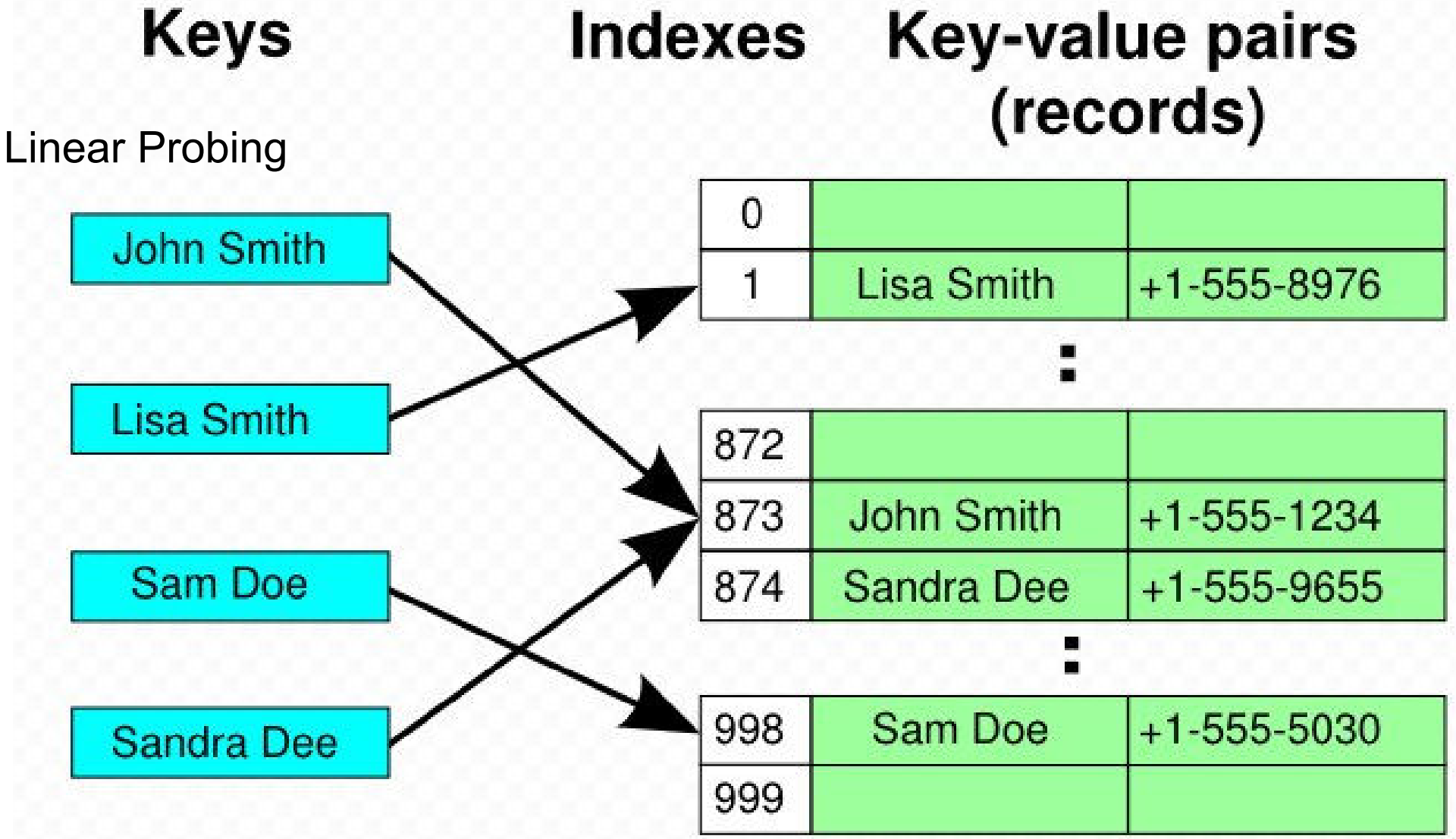
Telefonbuch als Hash-Tabelle



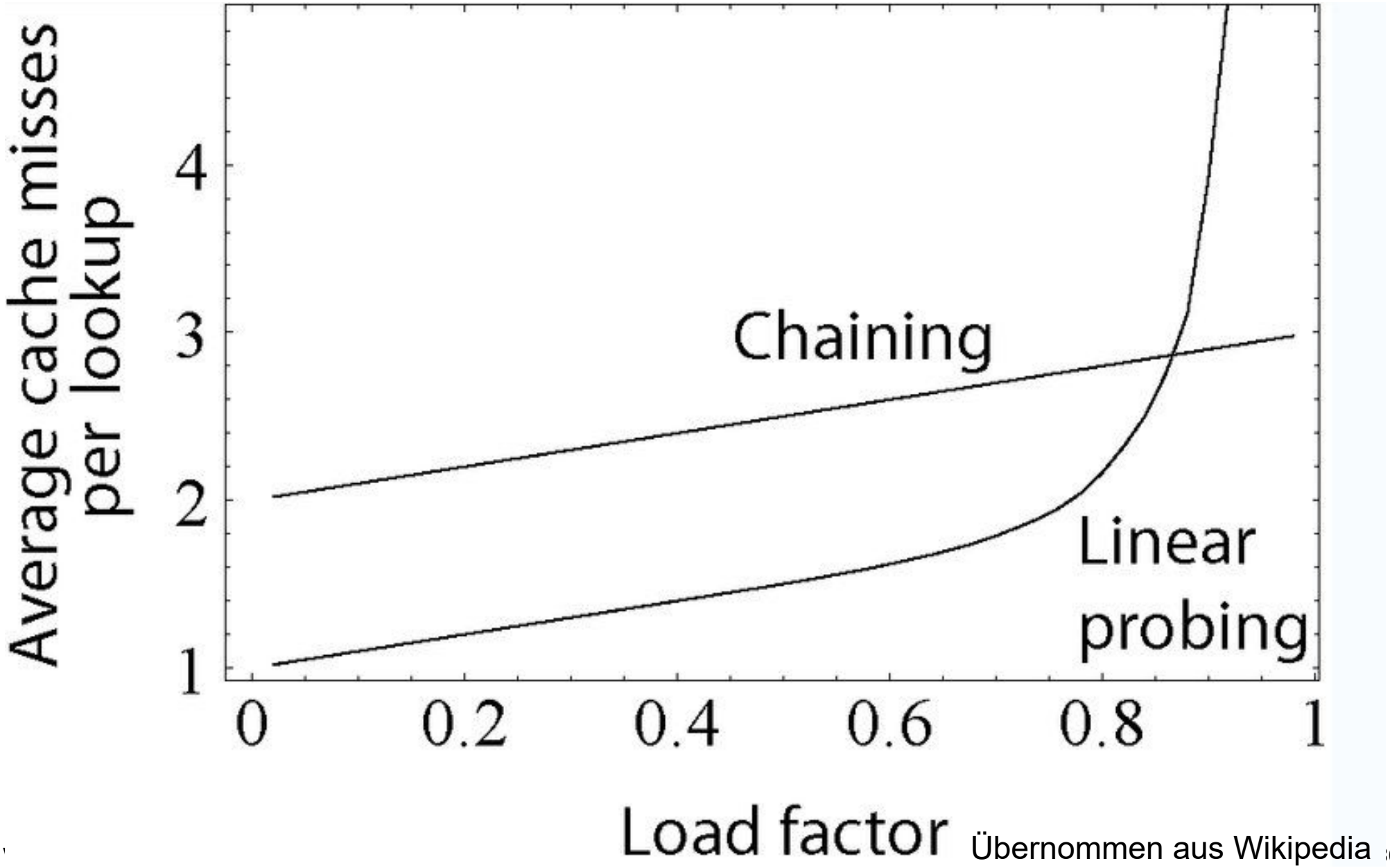
Kollisionsauflösung durch einen speziellen Kollisionsauflösungsbereich



Kollisionsauflösung durch lineare Suche in der gleichen Tabelle



Zeitverhalten



Übernommen aus Wikipedia

Container unordered_set

```
void demoSimpleHash1() {
  typedef unordered_map<int, char>
          HashTable;
  typedef HashTable::iterator Iter;
  typedef HashTable::value_type elt;

  HashTable cont;
  // alle Zeichen von HartmutHelmke einfügen
  cont.insert(elt(1, 'H'));
  cont.insert(elt(2, 'e'));
  cont.insert(elt(11, 'l'));
  cont.insert(elt(1, 'm'));
  cont.insert(elt(11, 'k'));
  cont.insert(elt(3, 'e'));
  cont.insert(elt(4, 'H'));
  cont.insert(elt(4, 'a'));
  cont.insert(elt(9, 'r'));
  cont.insert(elt(6, 't'));
  cont.insert(elt(7, 'm'));
  cont.insert(elt(8, 'u'));
  cont.insert(elt(8, 't'));
```

```
Iter iter;
  for (iter = cont.begin(); iter !=
          cont.end(); ++iter) {
    cout << "Key: "
         << (*iter).first;
    cout << " Value: "
         << (*iter).second << '\n';
```

```
Key: 9 Value: r
Key: 1 Value: H
Key: 2 Value: e
Key: 3 Value: e
Key: 11 Value: l
Key: 4 Value: H
Key: 6 Value: t
Key: 7 Value: m
Key: 8 Value: u
```

Insert ersetzt hier und in normaler Map keinen Wert.
Das erfolgt nur bei [].
‘u’ und nicht ‘t’ wird bei 8 Ausgegeben, d.h. das zuerst eingefügte Element.

Container unordered_set mit [], statt insert.

```
void demoSimpleHash1() {
  typedef unordered_map<int, char>
          HashTable;
  typedef HashTable::iterator Iter;
  typedef HashTable::value_type elt;
  HashTable cont;
 // alle Zeichen von HartmutHelmke einfügen
  cont[1] = 'H';
  cont[2] = 'e';
  cont[11] = 'l';
  cont[1] = 'm';
  cont[11] = 'k';
  cont[3] = 'e';
  cont[4] = 'H';
  cont[4] = 'a';
  cont[9] = 'r';
  cont[6] = 't';
  cont[7] = 'm';
  cont[8] = 'u';
  cont[8] = 't';
```

```
Iter iter;
  for (iter = cont.begin(); iter !=
          cont.end(); ++iter) {
    cout << "Key: "
         << (*iter).first;
    cout << " Value: "
         << (*iter).second << '\n';
```

Ausgabe bei insert:

Key: 9 Value: r
Key: 1 Value: H
Key: 2 Value: e
Key: 3 Value: e
Key: 11 Value: l
Key: 4 Value: H
Key: 6 Value: t
Key: 7 Value: m
Key: 8 Value: u

Ausgabe bei []:

Key: 9 Value: r
Key: 1 Value: m
Key: 2 Value: e
Key: 3 Value: e
Key: 11 Value: k
Key: 4 Value: a
Key: 6 Value: t
Key: 7 Value: m
Key: 8 Value: t

Container unordered_multiset

```
void demoSimpleHash1() {
  typedef unordered_multimap<int, char>
          HashTable;
  typedef HashTable::iterator Iter;
  typedef HashTable::value_type elt;

  HashTable cont;
  // alle Zeichen von HartmutHelmke einfügen
  cont.insert(elt(1, 'H'));
  cont.insert(elt(2, 'e'));
  cont.insert(elt(11, 'l'));
  cont.insert(elt(1, 'm'));
  cont.insert(elt(11, 'k'));
  cont.insert(elt(3, 'e'));
  cont.insert(elt(4, 'H'));
  cont.insert(elt(4, 'a'));
  cont.insert(elt(9, 'r'));
  cont.insert(elt(6, 't'));
  cont.insert(elt(7, 'm'));
  cont.insert(elt(8, 'u'));
  cont.insert(elt(8, 't'));
```

```
Key: 9 Value: r
Key: 1 Value: H
Key: 1 Value: m
Key: 2 Value: e
Key: 3 Value: e
Key: 11 Value: l
Key: 11 Value: k
```

```
Iter iter;
  for (iter = cont.begin(); iter !=
          cont.end(); ++iter) {
    cout << "Key: "
         << (*iter).first;
    cout << " Value: "
         << (*iter).second << '\n';
}
```

```
(fortgesetzte Ausgabe)
Key: 4 Value: H
Key: 4 Value: a
Key: 6 Value: t
Key: 7 Value: m
Key: 8 Value: u
Key: 8 Value: t
```

Das STL-Konzept: Überblick

unordered_set etc. fehlt

Elementfunktionen fur alle Container:

begin, end, rbegin, rend, // Iteratorpositionen
size, empty, clear,
=, ==, !=, <, <=, >, >=, // Zuweisung und Vergleiche
. . . . . .

Container:

vector	deque	list	stack	queue
[i] front back push_back	[i] front back push_back push_front	front back push_back insert push_front	top push pop	front back push pop . . . . .

priority_queue	set	multiset	map	multimap
top push pop	find insert erase	find insert erase	[key] find insert erase	find insert erase . . . . .

Iteratoren:

x = *p, *p = x, ++p, p++, --p, p--, p == q, p != q,
p[n], p + n, n + p, p - n, p += n, p -= n, p - q, p < q, p <= q, p >= q

Generische Funktionen:

for_each, find, copy, reverse, sort, binary_search, set_union, . . .

Aufgabe 4

```
void addHHeLetterToContainer(multiset<char>& cont)
{
    cont.insert('H');
    cont.insert('a');
    cont.insert('r');
    cont.insert('t');
    cont.insert('m');
}
```

```
void displayHHe()
{
    multiset<char> cont2;
    addHHeLetterToContainer(cont2);
    copy(cont2.begin(), cont2.end(),
        ostream_iterator<char>(cout, ";"));
    cout << "\n";
```

Aufgabe 4

```
/* Verwenden Sie nun ein unordered_set
   und ein unordered_multiset
   Geht auch map etc.?
   Erweitern Sie die Loesung sodass
   addHHeLetterToContainer ein Template ist*/
```

Aufgabe 5

```
#include <string>
#include <map>
using namespace std;

int main() {
  // Datentyp des Containers:
 typedef
 map<string, float, greater<string> >
                                StringFloatMap;
  StringFloatMap cont;
  cont["Meier"] = 8400.0f;
  cont["Schulz"] = 4600.0f;
  cont["Kohl"] = 28800.67f;
  cont["Eser"] = 147.4f;
  // Ausgabe in der Sortierreihenfolge
  StringFloatMap ::const_iterator iter;
  for (iter = cont.begin();
       iter != cont.end(); ++iter) {
    cout << "Key: " << (*iter).first << ' '
         << "Value: " << (*iter).second
         << endl;
  }
}
```

Ausgabe des Programmes:

```
Key: Schulz Value: 4600
Key: Meier Value: 8400
Key: Kohl Value: 28800.7
Key: Eser Value: 147.4
```

Implementieren Sie nun verschiedene assoziative Container zur Verwaltung der Gehälter (z.B. double) verschiedener Personen (z.B. string).
Verwenden Sie map, multimap, unordered_map und unorderded_multimap.

Aufgabe 5 konkret

```
void FillElems(map<string, double >& ar_cont) {
   typedef map<string, double >::value_type
          elem;
   ar_cont.insert(elem("Meier", 8400.14));
   ar_cont.insert(elem("Schulz", 4600.26));
   ar_cont.insert(elem("Kohl", 8800.67));
   ar_cont.insert(elem("Eser", 147.73));
// es wird nicht ueberschrieben, wenn vorhanden
   ar_cont.insert(elem("Meier", 2200.11));
   ar_cont.insert(elem("Kohl", 7880.67));
}

double CalcDurch(
   const map<string, double >& ar_cont){
   double durch = 0.0;
   for (const auto& iter : ar_cont) {
      durch += iter.second;
   }
   return ar_cont.empty() ? 0.0 : durch /
          ar_cont.size();
}
```

```
void PrintCont(
   const map<string, double >& ar_cont,
   double ad_durch
) {
   for (const auto& iter : ar_cont)   {
      cout << "Key: " << setw(7) <<
         iter.first << ' '
         << "Value: " << setw(8) <<
         iter.second
         << setw(6)
         << (iter.second > ad_durch ? "
         reich" : "arm")
         << endl;
   }
}
void berechneDurchgehalt1() {
   map<string, double >  cont;
   FillElems(cont);
   PrintCont(cont, CalcDurch(cont));
}
```

Aufgabe 5 konkret (2)

Implementieren Sie die gleiche Funktionalität bei Verwendung eines unordered_map Containers

Vereinfachen Sie nun, sodass Sie aus den beiden Implementierungen ein Template erstellen und sorgen Sie dafür, dass es auch mit multimap und unorder_multimap etc. nutzbar sind

```
void berechneDurchgehaltTempMultiOrdered(){
   unordered_multimap<string,double>  cont;
   FillElemsTemp(cont);
   PrintContTemp(cont, CalcDurchTemp(cont));
}
```

```
  Mit ordered_map:
Key:   Meier Value:  8400.14 reich
Key:  Schulz Value:  4600.26   arm
Key:    Kohl Value:  8800.67 reich
Key:    Eser Value:   147.73   arm

   Mit map als Temp:
Key:    Eser Value:   147.73   arm
Key:    Kohl Value:  8800.67 reich
Key:   Meier Value:  8400.14 reich
Key:  Schulz Value:  4600.26   arm

  Mit ordered_map als Temp:
Key:   Meier Value:  8400.14 reich
Key:  Schulz Value:  4600.26   arm
Key:    Kohl Value:  8800.67 reich
Key:    Eser Value:   147.73   arm

   Mit map als Temp:
Key:    Eser Value:   147.73   arm
Key:    Kohl Value:  8800.67 reich
Key:    Kohl Value:  7880.67 reich
Key:   Meier Value:  8400.14 reich
Key:   Meier Value:  2200.11   arm
Key:  Schulz Value:  4600.26   arm

  Mit ordered_map als Temp:
Key:   Meier Value:  8400.14 reich
Key:   Meier Value:  2200.11   arm
Key:  Schulz Value:  4600.26   arm
Key:    Kohl Value:  8800.67 reich
Key:    Kohl Value:  7880.67 reich
Key:    Eser Value:   147.73   arm
```

Arm, wenn weniger als oder gleich Durchschnitt, reich, wenn mehr