12 Container der STL

12.1 Übungsaufgabe

Wir arbeiten mit der Standard Template Library und verwenden verschiedene Container der STL vom Typ wstring zur Programmierung der Klasse Lexikon aus der Aufgabe 10. Verwenden Sie bei Schleifen grundsätzlich nur Iteratoren, testen Sie auch auto getypte Iteratoren.

Verwenden Sie für die `is_element()` Methode in ihrer Klasse Lexikon nur Algorithmen aus der STL, z.B. `find`.

Definieren Sie eine Klasse Lexikon. Im ihrem private Bereich soll ein Vector definiert werden, der Wörter speichern kann.

```
public:
          bool is_element(wstring);
          void insert(wstring);
          void sort_me()
          void print_number_of_words();
          void head(int n);

     private:

          void rm_punct(wstring&);
          vector<wstring> words;
          int number_of_words;
```

Prüfen Sie welche Container der STL-Library verwendet werden können, die es ermöglichen die Klasse Lexikon inklusiv der Routine `is_element()` zu implementieren.

Zur Auswahl stehen:

- array
- vector
- deque
- list
- set

- map
- unordered_set
- unordered_map

12.2 Wiederholung und zusätzliche Informationen

12.2.1 Container Überblick

Container	**Deklarieren**	**Einfügen**
array	array<wstring, maxsize> words	words[numberOfWords]=word
vector	vector<wstring> words	words.push_back(word)
deque	deque<wstring> words	words.push_back(word)
list	list<wstring> words	words.push_back(word)
set	set<wstring> words	words.insert(word)
map	map<wstring,int> words	words.insert(pair<wstring,int> (word,1))
unordered_set	unordered_set<wstring> words	words.insert(word)
unordered_map	unordered_map<wstring,int> words	words.insert(pair<wstring,int> (word,1))

Tabelle 12.1: Container der STL: Teil 1

Container	**Element finden**	**Sortieren**	**Sonstiges**
array	find (algorithm)	sort (algorithm)	C++11
vector	find (algorithm)	sort (algorithm)	
deque	find (algorithm)	sort (algorithm)	
list	find (algorithm)	sort (algorithm)	
set	words.find(word)	ist bereits sortiert	
map	words.find(word)	ist bereits sortiert	Duplikate möglich
unordered_set	words.find(word)	kann nicht sortiert werden (Hash)	C++11
unordered_map	words.find(word)	kann nicht sortiert werden (Hash)	C++11, Duplikate möglich

Tabelle 12.2: Container der STL: Teil 2

12.2.2 Checkliste beim Programmieren

- Alle nötigen `include`-Anweisungen verwendet?
- Alle Anweisungen mit Strichpunkt beendet?
- Alle Strings in Anführungszeichen?
- Alle geöffneten Klammern wieder geschlossen?
- Variablen deklariert und initialisiert?
- Groß-/Kleinschreibung korrekt verwendet?

- korrekte Operatoren verwendent (Zahlen vs. String)?
- Codierung korrekt eingestellt?
- Regex: Metazeichen ausgeschaltet?
- einheitliche Formatierung (Einrücken am Anfang der Zeilen)?
- Codeteile kommentiert?
- Sinnvolle Variablen/ Funktionsnamen verwendet?

12.3 Lösungsvorschlag

12.3.1 Lexikon.cpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * Lexikon.cpp
 */
#include "LexikonArray.hpp"
#include "LexikonVector.hpp"
#include "LexikonList.hpp"
#include "LexikonDeque.hpp"
#include "LexikonSet.hpp"
#include "LexikonMap.hpp"
#include "LexikonUnorderedSet.hpp"
#include "LexikonUnorderedMap.hpp"
#include <locale>
#include <fstream>
#include <sstream>
using namespace std;

int main() {
   //setzt Codierung auf utf8
   setlocale(LC_ALL, "de_DE.UTF-8");

   //Element Lexikon - hier Klasse auswählen
   LexikonArray myLexikon;
   // LexikonVector myLexikon;
   // LexikonList myLexikon;
   // LexikonDeque myLexikon;
   // LexikonSet myLexikon;
   // LexikonMap myLexikon;
   // LexikonUnorderedSet myLexikon;
   // LexikonUnorderedMap myLexikon;

   //es wird ein Vektor gefüllt um die Zeit für jeden Container
   //gleich messen zu können
   vector<wstring> search_words;

   //Füllen des Vectors mit den Worten einer Datei
```

```
    wifstream input("derIdiot.txt");
    wstring line;
    input.imbue(locale("de_DE.UTF-8"));

    while (getline(input, line)) {
       wstringstream myStream;
       myStream << line;
       wstring word;

       while(myStream >> word) {
          search_words.push_back(word);
       }
    }

    //Anzahl der Wörter wird ausgegeben
    myLexikon.print_number_of_words();

    //alle Wörter des Lexikons werden ausgegeben
    wcout << endl << L"Die ersten zehn Wörter des sortierten Lexikons: "
    << endl;
    myLexikon.head(10);
}
```

12.3.2 Lexikonarray.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonArray.hpp
 */
#ifndef LEXIKONARRAY_HPP
#define LEXIKONARRAY_HPP
#include <iostream>
#include <array>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonArray {
   public:
      LexikonArray();
      bool is_element(wstring word);
      bool insert(wstring word);
      void sort_me();
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
      static const int max_size = 10;
      array< wstring , max_size > words;
      int number_of_words;
};
```

```
//Konstruktor, erstellt ein leeres Lexikon
LexikonArray::LexikonArray() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonArray::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
   wstring::iterator first = word.begin();

   while (iswpunct(*first)) {
      word.erase(first);
   }

   //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
   wstring::iterator last = word.end() - 1;

   while (iswpunct(*last)) {
      word.erase(last);
      last = word.end() - 1;
   }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonArray::insert(wstring word) {
   rm_punct(word);

   //wenn word nicht schon im Lexikon ist
   if (!is_element(word)) {

      //trägt das Wort ins Lexikon ein
      words[ number_of_words] = word;
      number_of_words++;
      return true;
   }
   return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonArray::is_element(wstring Wort) {
   rm_punct(Wort);

   //sucht in words nach Wort
   auto pos = find( words .begin(), words.end(), Wort);

   //wenn die Iterator-Position hinter dem letzten Element ist,
   ist es nicht enthalten
```

```
    if (pos == words.end()) {
        return false;
    }

    //sonst ist es enthalten
    else {
        return true;
    }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonArray::print_number_of_words() {
    wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter." <<
    endl;
}
;

//Funktion, die die Daten des Lexikons sortiert
void LexikonArray::sort_me() {
    //Achtung sort sortiert nach ASCII Sortierung
    //Achtung hier kann nicht words.end() geschrieben werden, sonst werden die
    //leeren Elemente an erster Stelle geschrieben
    sort( words .begin(), words .begin()+ number_of_words);
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonArray::head(int n) {
    sort_me();

    //gibt die Elemente aus bis n = 0 ist
    for ( auto myIter = words .begin(); myIter != words.end() &&
    n != 0; ++myIter) {
        wcout << L"Element: " << *myIter << endl;
        n--;
    }
}
;

#endif
```

12.3.3 Lexikondeque.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonDeque.hpp
 */
#ifndef LEXIKONDEQUE_HPP
#define LEXIKONDEQUE_HPP
#include <iostream>
```

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

class LexikonDeque {
   public:
      LexikonDeque();
      bool is_element(wstring word);
      bool insert(wstring word);
      void sort_me();
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
      deque<wstring> words;
      int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonDeque::LexikonDeque() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonDeque::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
   wstring::iterator first = word.begin();

   while (iswpunct(*first)) {
      word.erase(first);
   }

   //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
   wstring::iterator last = word.end() - 1;

   while (iswpunct(*last)) {
      word.erase(last);
      last = word.end() - 1;
   }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonDeque::insert(wstring word) {
   rm_punct(word);

   //wenn word nicht schon im Lexikon ist
   if (!is_element(word)) {
```

```
        //trägt das Wort ins Lexikon ein
        words.push_back(word);
        number_of_words++;
        return true;
    }
    return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonDeque::is_element(wstring Wort) {
    rm_punct(Wort);

    //sucht in words nach Wort
    deque<wstring>::iterator pos = find(words.begin(), words.end(), Wort);

    //wenn die Iterator-Position hinter dem letzten Element ist,
    //ist es nicht enthalten
    if (pos == words.end()) {
        return false;
    }

    //sonst ist es enthalten
    else {
        return true;
    }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonDeque::print_number_of_words() {
    wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
    <<endl;
}
;

//Funktion, die die Daten des Lexikons sortiert
void LexikonDeque::sort_me() {
    //Achtung sort sortiert nach ASCII Sortierung
    sort(words.begin(), words.end());
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonDeque::head(int n) {
    sort_me();

    //gibt die Elemente aus bis n = 0 ist
    deque<wstring>::iterator myIter;
```

```
    for (myIter = words.begin(); myIter != words.end() && n != 0; ++myIter) {
        wcout << L"Element: " << *myIter << endl;
        n--;
    }
}
;

#endif
```

12.3.4 Lexikonlist.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonList.hpp
 */
#ifndef LEXIKONLIST_HPP
#define LEXIKONLIST_HPP
#include <iostream>
#include <list>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonList {
    public:
        LexikonList();
        bool is_element(wstring word);
        bool insert(wstring word);
        void sort_me();
        void print_number_of_words();
        void head(int n);

    private:
        void rm_punct(wstring &word);
        list<wstring> words;
        int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonList::LexikonList() {
    number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonList::rm_punct(wstring &word) {
    //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
    wstring::iterator first = word.begin();

    while (iswpunct(*first)) {
        word.erase(first);
    }
```

```
    //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
    wstring::iterator last = word.end() - 1;

    while (iswpunct(*last)) {
       word.erase(last);
       last = word.end() - 1;
    }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonList::insert(wstring word) {
    rm_punct(word);

    //wenn word nicht schon im Lexikon ist
    if (!is_element(word)) {

       //trägt das Wort ins Lexikon ein
       words.push_back(word);
       number_of_words++;
       return true;
    }
    return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonList::is_element(wstring Wort) {
    rm_punct(Wort);

    //sucht in words nach Wort
    list<wstring>::iterator pos = find(words.begin(), words.end(), Wort);

    //wenn die Iterator-Position hinter dem letzten Element ist,
    ist es nicht enthalten
    if (pos == words.end()) {
       return false;
    }

    //sonst ist es enthalten
    else {
       return true;
    }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonList::print_number_of_words() {
    wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
    <<endl;
```

```
}
;

//Funktion, die die Daten des Lexikons sortiert
void LexikonList::sort_me() {

   //auch diese sort Funktion sortiert nach ASCII
   words.sort();
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonList::head(int n) {
   sort_me();

   //gibt die Elemente aus bis n = 0 ist
   list<wstring>::iterator myIter;

   for (myIter = words.begin(); myIter != words.end() && n != 0; ++myIter) {
      wcout << L"Element: " << *myIter << endl;
      n--;
   }
}
;

#endif
```

12.3.5 Lexikonvector.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonVector.hpp
 */
#ifndef LEXIKONVECTOR_HPP
#define LEXIKONVECTOR_HPP
#include <iostream>
#include <vector>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonVector {
   public:
      LexikonVector();
      bool is_element(wstring word);
      bool insert(wstring word);
      void sort_me();
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
```

```
        vector<wstring> words;
        int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonVector::LexikonVector() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonVector::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
   wstring::iterator first = word.begin();

   while (iswpunct(*first)) {
      word.erase(first);
   }

   //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
   wstring::iterator last = word.end() - 1;

   while (iswpunct(*last)) {
      word.erase(last);
      last = word.end() - 1;
   }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonVector::insert(wstring word) {
   rm_punct(word);

   //wenn word nicht schon im Lexikon ist
   if (!is_element(word)) {

      //trägt das Wort ins Lexikon ein
      words.push_back(word);
      number_of_words++;
      return true;
   }
   return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonVector::is_element(wstring Wort) {
   rm_punct(Wort);

   //sucht in words nach Wort
   vector<wstring>::iterator pos = find(words.begin(), words.end(), Wort);
```

```
    //wenn die Iterator-Position hinter dem letzten Element ist,
    //ist es nicht enthalten
    if (pos == words.end()) {
       return false;
    }

    //sonst ist es enthalten
    else {
       return true;
    }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonVector::print_number_of_words() {
    wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
    <<endl;
}
;

//Funktion, die die Daten des Lexikons sortiert
void LexikonVector::sort_me() {
    //Achtung sort sortiert nach ASCII Sortierung
    sort(words.begin(), words.end());
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonVector::head(int n) {
    sort_me();

    //gibt die Elemente aus bis n = 0 ist
    vector<wstring>::iterator myIter;

    for (myIter = words.begin(); myIter != words.end() && n != 0; ++myIter) {
       wcout << L"Element: " << *myIter << endl;
       n--;
    }
}
;

#endif
```

12.3.6 Lexikonset.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonSet.hpp
 */
#ifndef LEXIKONSET_HPP
#define LEXIKONSET_HPP
```

```
#include <iostream>
#include <set>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonSet {
   public:
      LexikonSet();
      bool is_element(wstring word);
      bool insert(wstring word);
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
      set<wstring> words;
      int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonSet::LexikonSet() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonSet::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
   wstring::iterator first = word.begin();

   while (iswpunct(*first)) {
      word.erase(first);
   }

   //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
   wstring::iterator last = word.end() - 1;

   while (iswpunct(*last)) {
      word.erase(last);
      last = word.end() - 1;
   }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonSet::insert(wstring word) {
   rm_punct(word);

   //wenn word nicht schon im Lexikon ist
   if (!is_element(word)) {
```

```
      //trägt das Wort ins Lexikon ein
      words.insert(word);
      number_of_words++;
      return true;
   }
   return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonSet::is_element(wstring Wort) {
   rm_punct(Wort);

   //sucht in words nach Wort
   set<wstring>::iterator pos = words.find(Wort);

   //wenn die Iterator-Position hinter dem letzten Element ist,
   //ist es nicht enthalten
   if (pos == words.end()) {
      return false;
   }

   //sonst ist es enthalten
   else {
      return true;
   }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonSet::print_number_of_words() {
   wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
   <<endl;
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonSet::head(int n) {
   //gibt die Elemente aus bis n = 0 ist
   set<wstring>::iterator myIter;

   for (myIter = words.begin(); myIter != words.end() && n != 0; ++myIter) {
      wcout << L"Element: " << *myIter << endl;
      n--;
   }
}
;

#endif
```

12.3.7 Lexikonmap.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonMap.hpp
 */
#ifndef LEXIKONMAP_HPP
#define LEXIKONMAP_HPP
#include <iostream>
#include <map>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonMap {
   public:
      LexikonMap();
      bool is_element(wstring word);
      void insert(wstring word);
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
      map<wstring, int> words;
      int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonMap::LexikonMap() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonMap::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
   wstring::iterator first = word.begin();

   while (iswpunct(*first)) {
      word.erase(first);
   }

   //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
   wstring::iterator last = word.end() - 1;

   while (iswpunct(*last)) {
      word.erase(last);
      last = word.end() - 1;
   }
}
```

```
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
void LexikonMap::insert(wstring word) {
   rm_punct(word);

   //wenn word nicht schon im Lexikon ist
   if (!is_element(word)) {

      //trägt das neue Wort ins Lexikon ein
      words.insert(pair<wstring, int> (word, 1));
   }

   else {
      //erhöht die Anzahl des Wortes um 1
      words[word]++;
   }
   number_of_words++;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonMap::is_element(wstring Wort) {
   rm_punct(Wort);

   //sucht in words nach Wort
   map<wstring, int>::iterator pos = words.find(Wort);

   //wenn die Iterator-Position hinter dem letzten Element ist,
   //ist es nicht enthalten
   if (pos == words.end()) {
      return false;
   }

   //sonst ist es enthalten
   else {
      return true;
   }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonMap::print_number_of_words() {
   wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
   <<endl;
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonMap::head(int n) {
   //gibt die Elemente aus bis n = 0 ist
```

```
    map<wstring, int>::iterator myIter;

    for (myIter = words.begin(); myIter != words.end() && n != 0; ++myIter) {
       wcout << L"Element: " << myIter->first << " mit der Anzahl " <<
       myIter->second << endl;

       // Alternative:
       // wcout << L"Element: " << (*myIter).first << " mit der Anzahl " <<
       (*myIter).second << endl;
       n--;
    }
}
;

#endif
```

12.3.8 Lexikonunorderedset.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonUnorderedSet.hpp
 */
#ifndef LEXIKONUNORDEREDSET_HPP
#define LEXIKONUNORDEREDSET_HPP
#include <iostream>
#include <unordered_set>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonUnorderedSet {
   public:
      LexikonUnorderedSet();
      bool is_element(wstring word);
      bool insert(wstring word);
      void print_number_of_words();
      void head(int n);
   private:
      void rm_punct(wstring &word);
      unordered_set< wstring > words;
      int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonUnorderedSet::LexikonUnorderedSet() {
   number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonUnorderedSet::rm_punct(wstring &word) {
   //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
```

```
    wstring::iterator first = word.begin();

    while (iswpunct(*first)) {
       word.erase(first);
    }

    //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
    wstring::iterator last = word.end() - 1;

    while (iswpunct(*last)) {
       word.erase(last);
       last = word.end() - 1;
    }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
bool LexikonUnorderedSet::insert(wstring word) {
    rm_punct(word);

    //wenn word nicht schon im Lexikon ist
    if (!is_element(word)) {

       //trägt das Wort ins Lexikon ein
       words.insert(word);
       number_of_words++;
       return true;
    }
    return false;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonUnorderedSet::is_element(wstring Wort) {
    rm_punct(Wort);

    //sucht in words nach Wort
    unordered_set< wstring >::iterator pos = words.find(Wort);

    //wenn die Iterator-Position hinter dem letzten Element ist,
    //ist es nicht enthalten
    if (pos == words.end()) {
       return false;
    }

    //sonst ist es enthalten
    else {
       return true;
    }
}
;
```

```
//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonUnorderedSet::print_number_of_words() {
   wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
   << endl;
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonUnorderedSet::head(int n) {
   //gibt die Elemente aus bis n = 0 ist
   unordered_set< wstring>::iterator myIter;

   for (myIter = words .begin(); myIter != words.end() && n != 0; ++myIter) {
      wcout << L"Element: " << *myIter << endl;
      n--;
   }
}
;

#endif
```

12.3.9 Lexikonunorderedmap.hpp

```
/*Autor: Nicola Greth
 * Uebung 11
 * LexikonUnorderedMap.hpp
 */
#ifndef LEXIKONUNORDEREDMAP_HPP
#define LEXIKONUNORDEREDMAP_HPP
#include <iostream>
#include <unordered_map>
#include <locale>
#include <algorithm>
using namespace std;

class LexikonUnorderedMap {
   public:
      LexikonUnorderedMap();
      bool is_element(wstring word);
      void insert(wstring word);
      void print_number_of_words();
      void head(int n);

   private:
      void rm_punct(wstring &word);
      unordered_map< wstring , int > words;
      int number_of_words;
};

//Konstruktor, erstellt ein leeres Lexikon
LexikonUnorderedMap::LexikonUnorderedMap() {
```

```
    number_of_words = 0;
}
;

//Funktion, die Punktuationszeichen aus am Anfang und Ende von word entfernt
void LexikonUnorderedMap::rm_punct(wstring &word) {
    //solange am Anfang Punktuationszeichen sind, wird der Anfang gelöscht
    wstring::iterator first = word.begin();

    while (iswpunct(*first)) {
        word.erase(first);
    }

    //solange am Ende Punktuationszeichen sind, wird das Ende gelöscht
    wstring::iterator last = word.end() - 1;

    while (iswpunct(*last)) {
        word.erase(last);
        last = word.end() - 1;
    }
}
;

//Funktion insert, die word unter bestimmten Bedingungen ins Lexikon einträgt
void LexikonUnorderedMap::insert(wstring word) {
    rm_punct(word);

    //wenn word nicht schon im Lexikon ist
    if (!is_element(word)) {

        //trägt das neue Wort ins Lexikon ein
        words .insert( pair< wstring , int> (word, 1));
    }

    else {
        //erhöht die Anzahl des Wortes um 1
        words[word]++;
    }
    number_of_words++;
}
;

//Funktion is_element, die testet ob word schon element im Lexikon ist
bool LexikonUnorderedMap::is_element(wstring Wort) {
    rm_punct(Wort);

    //sucht in words nach Wort
    unordered_map< wstring , int >::iterator pos = words.find(Wort);

    //wenn die Iterator-Position hinter dem letzten Element ist,
    //ist es nicht enthalten
```

```
    if (pos == words.end()) {
        return false;
    }

    //sonst ist es enthalten
    else {
        return true;
    }
}
;

//Funktion, die die Anzahl der Wörter im Lexikon ausdruckt
void LexikonUnorderedMap::print_number_of_words() {
    wcout << L"Ihr Lexikon enthält " << number_of_words << L" Wörter."
    <<endl;
}
;

//Funktion, die die ersten n Elemente des sortieren Lexikons ausgibt
void LexikonUnorderedMap::head(int n) {
    //gibt die Elemente aus bis n = 0 ist
    unordered_map< wstring , int>::iterator myIter;

    for (myIter = words .begin(); myIter != words.end() && n != 0; ++myIter) {
        wcout << L"Element: " << myIter->first << " mit der Anzahl " <<
        myIter->second << endl;

        // Alternative:
        // wcout << L"Element: " << (*myIter).first << " mit der Anzahl " <<
        (*myIter).second << endl;
        n--;
    }
}
;
#endif
```