# Hashtable

## Lösungsidee

### Hashtable

Bei der Realisierung der *Hashtable* sollte die vorgegeben Header Datei verwendet werden. Dabei werden alle grundlegenden Methoden implementiert, welche für eine *Hashtable* nötig sind. Die zu implementierenden Funktion sind in der Angabe beschrieben.Alle Elemente die neu hinzugefügt werden innerhalb der Hashtable in folgende Datenstruktur abgespeichert:

* *vector<list<value\_type>>*

Der *value\_type* ist ein Template Parameter der im beim instanziieren der Klasse gewählt wird. Der Vorteil bei dieser Datenstruktur ist, dass der *vector* die Eigenschaft hat, dass es leicht ist ihn zu vergrößern bzw. zu verkleinern und es die Möglichkeit gibt, auf Elemente innerhalb dieses Vektors mittels Index zuzugreifen. Die Elemente dieses Vektors wurden als *list* implementiert, da diese im Prinzip nur durchlaufen werden und nach dem gewünschten Element durchsucht werden. Um einen möglichst schnellen Zugriff auf die einzelnen Elemente zu ermöglichen, wird mittels einer im Konstruktor übergebenen Hash-Funktion ein Hash generiert. Der Einfachheit halber wird bei dieser Hashtable für die Index des aktuellen Elementes der Wert des Elementes der Hash Funktion übergeben, dieser Hash dann modulo der Größe des Vektors gerechnet, dass ein vorhandener Index gefunden wird. Diese Logik für das Berechnen dieses Hash Wertes sollte in eine eigene Funktion namens *getHashedValue*() ausgelagert werden.Beim Hinzufügen eines Elementes wird dazu vorher überprüft ob das Element bereits in der Hashtable vorhanden ist, wenn nein wird der Hash Wert wie oben beschrieben berechnet und danach mit dem zurückgegebenen Index die Liste im Vektor selektiert welcher das Element angefügt werden sollte. Dieser Liste wird das Element hinten angefügt und anschließend die aktuelle Größe erhöht. Wenn diese Größe einen gewissen Wert übersteigt, in unserem Fall sollte das der *MaxLoadFactor* mal der Kapazität sein, wird die Funktion *rehash* mit der verdoppelten aktuellen Größe aufgerufen. In der Funktion *rehash* sollte die Tabelle vergrößert und anschließend neu befüllt werden. Dazu wird zu Erst eine Kopie der aktuellen Tabelle angelegt, anschließend wird die Kapazität mit der gewünschten Größe überschrieben und die aktuelle Größe auf 0 gesetzt. Die Tabelle wird anschließend auf die neue Kapazität geändert. Dies sollte mit Hilfe der *resize* Funktion geschehen. Diese Tabelle wird anschließend geleert und dann neu befüllt. Aufgrund der Tatsache, dass sich die Größe der Tabelle geändert hat, wird jetzt auch ein neuer Hash Wert für die einzelnen Elemente generiert, daher muss die Tabelle neu befüllt werden.

Die erase Funktion sollte gleich aufgebaut sein wie die insert Funktion. Statt der Überprüfung ob das Element nicht vorhanden ist, sollte überprüft werden, ob das Element vorhanden ist, wenn nicht kann es auch nicht entfernt werden und es muss daher nichts passieren. Falls ein Element mit dem Wert vorhanden ist, wird die Liste mit dem Element mit dem Wert den wir aus der Funktion *getHashedValue* bekommen (wie oben beschrieben) selektiert. Da wir uns sicher sein können, dass der Wert in dieser Liste existiert, brauchen wir anschließend nur noch die *remove* Funktion auf die selektierte Liste aufrufen. Danach wird die aktuelle Größe wieder um eines verringert. Wenn diese Größe unter einen bestimmten Wert sinkt, muss wieder die *rehash* Funktion aufgerufen werden. Der bestimmte Wert ergibt sich dabei aus der Multiplikation von *MinLoadFactor* und der aktuellen Kapazität. Die Funktion *rehash* wird dabei mit der halbierten aktuellen Größe aufgerufen und funktioniert wie oben beschrieben.

Die *contains* Funktion sollte feststellen, ob ein bestimmtes Element in der Tabelle vorhanden ist. Dazu wird über die *getHashedValue* der Hash Wert für das aktuelle Element generiert, über diesen Wert wird schließlich die Liste aus dem Vektor selektiert und diese Liste wird danach durchsucht ob der Wert enthalten ist. Wenn das Ergebnis dieser Suche dem End-Iterator der Liste entspricht existiert es nicht ansonsten gibt es ein Element.

Bei der Überladung des *== Operators* sollte zu Erst überprüft werden ob die übergebene Tabelle die gleiche Adresse hat, wie die aktuelle Klasse. Dann ist es in jedem Fall das gleiche Objekt bzw. hat die gleichen Elemente. Wenn die Größe der übergebenen Tabelle sich von der aktuellen Tabelle unterscheidet kann in jedem Fall gesagt werden, dass die Tabellen unterschiedlich sind. Falls diese beiden Fälle nicht eintreten, sollte die übergebene Liste in einer Schleife durchlaufen werden und für jedes Element dieser Tabelle überprüft werden, ob es im aktuellen Objekt vorhanden ist. Falls nicht kann davon ausgegangen werden, dass es nicht vorhanden ist. Falls die Schleife nicht abgebrochen wurde, sind alle Elemente der übergebenen Tabelle vorhanden und es kann daher davon ausgegangen werden, dass diese Tabellen zumindest die gleichen Elemente besitzen. Für diese Funktion wird es nötig sein einen Iterator zu implementieren. Wie dieser aussehen sollte wird weiter unten beschrieben.

Die Überladung des *<< Operators* geht in einer Schleife alle Elemente der übergebenen Tabelle durch und gibt diese auf dem übergebenen *ostream* aus. Auch diese Funktion benötigt die Implementierung des Iterators.

Die weiteren Funktionen sind trivial und sollten entweder einzelne Werte zurückgeben oder eine Kombination dieser.

### Iterator

Bei der Implementierung des Iterators für die Hashtable sollte darauf geachtet werden, dass alle Elemente nacheinander durchlaufen werden. Das heißt, wir müssen nacheinander alle Listen aus dem Vektor selektieren und die einzelnen darin enthaltenen Elemente selektieren. Beim instanziieren des Iterators sollte hierzu auf das erste gültige Element in der Hashtable verwiesen werden. Das erste gültige Element wird dabei dadurch gefunden, dass man mittels vector.begin() startet und solange die zurückgegebene Liste leer ist den dabei zurückgegeben Iterator erhöht, bis die erste Liste kommt welche mindestens ein Element enthält. Sobald dieses Gefunden wird, werden zwei Membervariablen in der Iterator Klasse gesetzt. Diese Funktionalität sollte in der *begin* Funktion realisiert werden. Falls das Ende der Hashtable gewünscht ist, wird der gleiche Prozess durchgeführt, nur wird vom Ende des Vektors ausgegangen und so lange verringert bis die aktuelle Liste mindestens ein Element enthält. Die Membervariablen werden dann auf diesen Iterator mit der ersten Liste gesetzt und der zweite Iterator wird auf das letzte Element dieser Liste gesetzt. Diese Funktionalität sollte in einer Funktion end() realisiert werden.

Falls auf den Iterator der Hashtable *operator++* augerufen wird, wird zu Erst der Listeniterator erhöht. Wenn dieser am Ende der Liste angekommen ist, dann wird der Vektoriterator erhöht und der Listeniterator danach auf den Anfang der aktuellen Liste gesetzt. Dieser Vorgang wird so lange durchgeführt bis eine Liste gefunden wurde die mindestens ein Element hat.

Wenn *operator--* aufgerufen wird, sollte zu Erst überprüft ob der *listIterator* dem Beginn des Vektoriterators entspricht. Wenn nicht sollte er verringert werden. Falls diese Aussage wahr ist, wird der *vectorIterator* solange verringert, bis er entweder dem Anfang des Vektors entspricht oder die Liste auf die der Vektoriterator verweist eine Größe größer als 0 hat. Falls der Vektoriterator also dem Beginn des Vektors entspricht sollte die oben beschrieben Funktion *begin* aufgerufen werden. Falls nicht, wird der *listIterator* an das Ende des Vektoriterators gesetzt und um eins verringert.

Bei der Überladung des == Operators sollte überprüft werden ob der *vectorIterator* dem übergeben *vectorIterator* entspricht und ob der *listIterator* dem übergebenen entspricht.

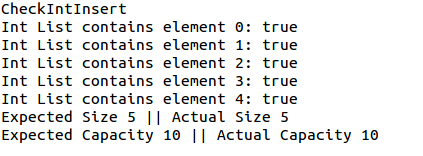
Bei dem != Operator sollte überprüft werden ob der *vectorIterator* ungleich dem übergeben *vectorIterator* ist oder der *listIterator* ungleich dem übergebenen entspricht.

Die Funktionen für die *reference* und den *pointer* sind trivial und benötigen daher keine nähere Beschreibung.

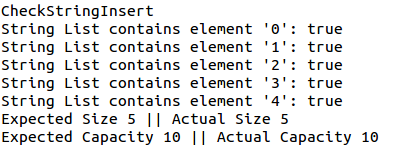
Es sollten auch die *begin*() *end*() bzw. die *cbegin*() und *cend*() Funktionen für die Hashtable implementiert werden. Dabei sollte für die *begin* Funktion einfach eine Instanz des zu implementierenden *const\_iterators* erstellt weden und bei der *end* Funktion ebenfalls. Der Unterschied ist, dass bei der end Funktion der Iterator noch an das Ende gesetzt wird. Die Funktionen *cbegin* und *cend* sowie deren Überladungen rufen im Prinzip nur *begin* und end auf.

## Tests

### Check Int Insert



### Check String Insert



### Check Int Operator==



### Check String Operator==



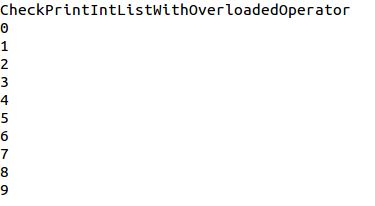
### Check Erase Int



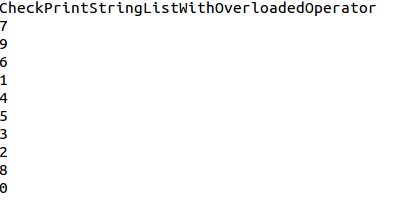
### Check Erase String



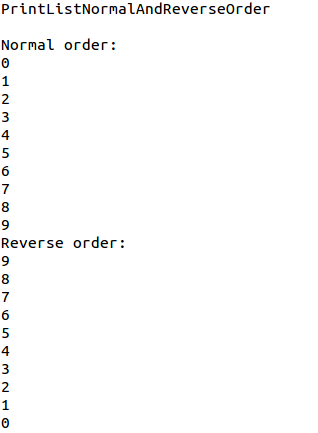
### Check Print Int List With Operator<<



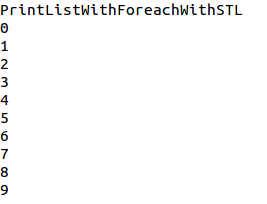
### Check Print String List With Operator<<



### Print List Normal and reversed Order



### Print List With Foreach STL



### Find Element With STL



## Quelltext

### Hashtable.hpp

#ifndef hashtable\_hpp

#define hashtable\_hpp

#include <cassert>

#include <vector>

#include <list>

#include <iostream>

#include <iterator>

#include <iomanip>

using namespace std;

template<typename V, typename H, typename C>

class hashtable;

template<typename V, typename H, typename C>

std::ostream & operator << (std::ostream & os, const hashtable<V, H, C> &ht);

template<typename V, typename H, typename C>

class hashtable {

friend std::ostream & operator << <V, H, C>(std::ostream & os, const hashtable<V, H, C> &ht);

public:

typedef V value\_type;

typedef H hash\_function\_type;

typedef C key\_equal\_function\_type;

typedef unsigned int size\_t;

typedef value\_type const \* const\_pointer;

typedef value\_type const & const\_reference;

typedef ptrdiff\_t difference\_type;

typedef const\_pointer pointer;

typedef const\_reference reference;

typedef size\_t size\_type;

hashtable(size\_t n\_buckets = 10,

hash\_function\_type hasher = hash<V>(),

key\_equal\_function\_type equals = equal\_to<V>(),

double max\_load\_factor = 0.8,

double min\_load\_factor = 0.2)

: \_capacity(n\_buckets), \_hasher(hasher), \_equals(equals), \_maxLoadFactor(max\_load\_factor), \_minLoadFactor(min\_load\_factor)

{

rehash(\_capacity);

}

virtual ~hashtable() {}

void insert(const V &value);

void erase(const V &value);

bool contains(const V &value) const;

void rehash(size\_t new\_n\_buckets);

double load\_factor() const;

size\_t size() const;

size\_t capacity() const;

bool empty() const;

bool operator == (const hashtable &other) const;

typedef std::iterator <bidirectional\_iterator\_tag,

value\_type,

difference\_type,

const\_pointer,

const\_reference> iterator\_base;

class const\_iterator : public iterator\_base {

private:

const hashtable\* \_table;

typename vector<list<value\_type>>::const\_iterator \_vectorIterator;

typename list<value\_type>::const\_iterator \_listIterator;

public:

typedef typename iterator\_base::difference\_type difference\_type;

typedef typename iterator\_base::iterator\_category iterator\_category;

typedef typename iterator\_base::pointer pointer;

typedef typename iterator\_base::reference reference;

const\_iterator(const hashtable \*table);

bool operator == (const\_iterator const & rhs) const;

bool operator != (const\_iterator const & rhs) const;

reference operator \* () const;

pointer operator -> () const;

void begin();

void end();

void next();

void previous();

const\_iterator & operator ++ ();

const\_iterator & operator -- ();

const\_iterator operator ++ (int unused);

const\_iterator operator -- (int unused);

};

typedef const\_iterator iterator;

const\_iterator begin() const;

const\_iterator end() const;

iterator cbegin();

iterator cend();

const\_iterator cbegin() const;

const\_iterator cend() const;

private :

vector<list<V>> \_table;

size\_t \_capacity;

size\_t \_currentSize;

double \_minLoadFactor;

double \_maxLoadFactor;

hash\_function\_type \_hasher;

key\_equal\_function\_type \_equals;

size\_t getHashedValue(const V& value) const;

};

/\*

\*----------------------------------------------------------------------------------------------------------------------------

\* Hashtable Implementation

\*----------------------------------------------------------------------------------------------------------------------------

\*/

template<typename V, typename H, typename C>

ostream & operator << (ostream & os, const hashtable<V, H, C> &ht){

for (auto element : ht){

os << element << endl;

}

return os;

}

template<typename V, typename H, typename C>

bool hashtable<V, H, C>::operator == (const hashtable &other) const{

if (&other == this)

return true;

if (other.size() != this->size())

return false;

for (auto element : other){

if (!this->contains(element)){

return false;

}

}

return true;

}

template <typename V, typename H, typename C>

bool hashtable<V, H, C>::contains(const V &value) const {

const list<value\_type>& whichList = \_table[getHashedValue(value)];

return find(whichList.begin(), whichList.end(), value) != whichList.end();

}

template <typename V, typename H, typename C>

double hashtable<V, H, C>::load\_factor() const {

return \_currentSize / \_capacity;

}

template <typename V, typename H, typename C>

size\_t hashtable<V, H, C>::size() const {

return \_currentSize;

}

template <typename V, typename H, typename C>

size\_t hashtable<V, H, C>::capacity() const {

return \_capacity;

}

template <typename V, typename H, typename C>

bool hashtable<V, H, C>::empty() const {

return \_table.empty();

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::insert(const V &value) {

if (!contains(value)){

list<value\_type>& whichList = \_table[getHashedValue(value)];

whichList.push\_back(value);

if (++\_currentSize > (\_maxLoadFactor \* \_capacity))

rehash(static\_cast<size\_t>(\_currentSize \* 2));

}

else{

// Element is already available in hashtable so there is no need to add it

}

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::erase(const V &value){

if (contains(value)){

list<value\_type>& whichList = \_table[getHashedValue(value)];

whichList.remove(value);

if (--\_currentSize > \_minLoadFactor \* \_capacity){

rehash(static\_cast<size\_t>(\_currentSize \* 0.5));

}

}

else{

// Element is not existent in hashtable so there is no need to remove it

}

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::rehash(size\_t new\_n\_buckets){

vector<list<V>> old = \_table;

\_capacity = new\_n\_buckets;

\_currentSize = 0;

\_table.resize(\_capacity);

for (size\_t i = 0; i < \_table.size(); i++) {

\_table[i].clear();

}

for (auto elements : old) {

for (auto element : elements) {

insert(element);

}

}

}

template <typename V, typename H, typename C>

size\_t hashtable<V, H, C>::getHashedValue(const V& value) const{

size\_t hashValue = \_hasher(value) % \_capacity;

if (hashValue < 0){

hashValue += \_table.size();

}

return hashValue;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::begin() const{

return const\_iterator(this);

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::end() const {

auto it = const\_iterator(this);

it.end();

return it;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::iterator hashtable<V, H, C>::cbegin() {

return begin();

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::iterator hashtable<V, H, C>::cend(){

return end();

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::cbegin() const {

return begin();

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::cend() const{

return end();

}

/\*

\*----------------------------------------------------------------------------------------------------------------------------

\* Iterator Implementation

\*----------------------------------------------------------------------------------------------------------------------------

\*/

template <typename V, typename H, typename C>

hashtable<V, H, C>::const\_iterator::const\_iterator(const hashtable \*table) : \_table(table){

begin();

}

template <typename V, typename H, typename C>

bool hashtable<V, H, C>::const\_iterator::operator == (const\_iterator const & rhs) const{

return \_vectorIterator == rhs.\_vectorIterator && \_listIterator == rhs.\_listIterator;

}

template <typename V, typename H, typename C>

bool hashtable<V, H, C>::const\_iterator::operator != (const\_iterator const & rhs) const{

return \_vectorIterator != rhs.\_vectorIterator || \_listIterator != rhs.\_listIterator;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator::reference hashtable<V, H, C>::const\_iterator::operator \* () const{

return \*\_listIterator;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator::pointer hashtable<V, H, C>::const\_iterator::operator -> () const{

return &(\*\_listIterator);

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::const\_iterator::begin(){

\_vectorIterator = \_table->\_table.begin();

while (\_vectorIterator != \_table->\_table.end() && \_vectorIterator->size() == 0)

++\_vectorIterator;

\_listIterator = \_vectorIterator->begin();

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::const\_iterator::end(){

\_vectorIterator = --\_table->\_table.end();

while (\_vectorIterator != \_table->\_table.begin() && \_vectorIterator->size() == 0)

--\_vectorIterator;

\_listIterator = \_vectorIterator->end();

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::const\_iterator::next(){

if (++\_listIterator == \_vectorIterator->end()){

do {

++\_vectorIterator;

} while (\_vectorIterator != \_table->\_table.end() && \_vectorIterator->size() == 0);

if (\_vectorIterator == \_table->\_table.end()){

end();

}

else{

\_listIterator = \_vectorIterator->begin();

}

}

}

template <typename V, typename H, typename C>

void hashtable<V, H, C>::const\_iterator::previous(){

if (\_listIterator != \_vectorIterator->begin()){

--\_listIterator;

}

else{

do {

--\_vectorIterator;

} while (\_vectorIterator != \_table->\_table.begin() && \_vectorIterator->size() == 0);

if (\_vectorIterator == \_table->\_table.begin()){

begin();

}

else{

\_listIterator = \_vectorIterator->end();

--\_listIterator;

}

}

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator & hashtable<V, H, C>::const\_iterator::operator ++ (){

next();

return \*this;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator & hashtable<V, H, C>::const\_iterator::operator -- (){

previous();

return \*this;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::const\_iterator::operator ++ (int unused){

next();

return \*this;

}

template <typename V, typename H, typename C>

typename hashtable<V, H, C>::const\_iterator hashtable<V, H, C>::const\_iterator::operator -- (int unused){

previous();

return \*this;

}

#endif // hashtable\_hpp

### Main.cpp

#include <cassert>

#include <vector>

#include <list>

#include <iostream>

#include <iterator>

#include <iomanip>

#include <algorithm>

#include <sstream>

#include "hashtable.hpp"

using namespace std;

void CheckIntInsert(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 5; i++){

h.insert(i);

cout << "Int List contains element " << i << ": " << (h.contains(i) ? "true" : "false") << endl;

}

cout << "Expected Size 5 || Actual Size " << h.size() << endl;

cout << "Expected Capacity 10 || Actual Capacity " << h.capacity() << endl;

}

void CheckStringInsert(){

hashtable<string, hash<string>, equal\_to<string>> h(5, hash<string>(), equal\_to<string>());

for (int i = 0; i < 5; i++){

h.insert(to\_string(i));

cout << "String List contains element '" << i << "': " << (h.contains(to\_string(i)) ? "true" : "false") << endl;

}

cout << "Expected Size 5 || Actual Size " << h.size() << endl;

cout << "Expected Capacity 10 || Actual Capacity " << h.capacity() << endl;

}

void CheckEraseInt(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

}

cout << "Int List contains Value 5 before erase: " << (h.contains(5) ? "true" : "false") << endl;

h.erase(5);

cout << "Int List contains Value 5 after erase: " << (h.contains(5) ? "true" : "false") << endl;

}

void CheckEraseString(){

hashtable<string, hash<string>, equal\_to<string>> h(5, hash<string>(), equal\_to<string>());

for (int i = 0; i < 10; i++){

h.insert(to\_string(i));

}

cout << "String List contains Value '5' before erase: " << (h.contains(to\_string(5)) ? "true" : "false") << endl;

h.erase(to\_string(5));

cout << "String List contains Value '5' after erase: " << (h.contains(to\_string(5)) ? "true" : "false") << endl;

}

void CheckIntOperatorEquals(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

hashtable<int, hash<int>, equal\_to<int>> h2(100, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

h2.insert(i);

}

cout << "Int Lists are the same: " << (h == h2 ? "true" : "false") << endl;

}

void CheckStringOperatorEquals(){

hashtable<string, hash<string>, equal\_to<string>> h(5, hash<string>(), equal\_to<string>());

hashtable<string, hash<string>, equal\_to<string>> h2(100, hash<string>(), equal\_to<string>());

for (int i = 0; i < 10; i++){

h.insert(to\_string(i));

h2.insert(to\_string(i));

}

cout << "String Lists are the same: " << (h == h2 ? "true" : "false") << endl;

}

void CheckPrintIntListWithOverloadedOperator(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

}

cout << h << endl;

}

void CheckPrintStringListWithOverloadedOperator(){

hashtable<string, hash<string>, equal\_to<string>> h(5, hash<string>(), equal\_to<string>());

for (int i = 0; i < 10; i++){

h.insert(to\_string(i));

}

cout << h << endl;

}

void PrintListNormalAndReverseOrder(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

}

auto it = h.begin();

cout << "Normal order: " << endl;

while (it != h.end()){

cout << (\*it) << endl;

it++;

}

cout << "Reverse order: " << endl;

auto reverseIt = h.end();

do{

--reverseIt;

cout << (\*reverseIt) << endl;

} while (reverseIt != h.begin());

}

void PrintListWithForeachWithSTL(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

}

for\_each(h.begin(), h.end(), [](const int& element){

cout << element << endl;

});

}

void FindElementWithSTL(){

hashtable<int, hash<int>, equal\_to<int>> h(5, hash<int>(), equal\_to<int>());

for (int i = 0; i < 10; i++){

h.insert(i);

}

auto foundIt = find(h.begin(), h.end(), 5);

cout << "Should find element '5': Found " << (foundIt != h.end() ? "true" : "false") << endl;

auto foundItNot = find(h.begin(), h.end(), 15);

cout << "Should not find element '15': Found " << (foundItNot != h.end() ? "true" : "false") << endl;

}

int main(int argc, char\*\* argv) {

cout << "CheckIntInsert" << endl;

CheckIntInsert();

cout << endl;

cout << "CheckStringInsert" << endl;

CheckStringInsert();

cout << endl;

cout << "CheckIntOperatorEquals" << endl;

CheckIntOperatorEquals();

cout << endl;

cout << "CheckStringOperatorEquals" << endl;

CheckStringOperatorEquals();

cout << endl;

cout << "CheckEraseInt" << endl;

CheckEraseInt();

cout << endl;

cout << "CheckEraseString" << endl;

CheckEraseString();

cout << endl;

cout << "CheckPrintIntListWithOverloadedOperator" << endl;

CheckPrintIntListWithOverloadedOperator();

cout << endl;

cout << "CheckPrintStringListWithOverloadedOperator" << endl;

CheckPrintStringListWithOverloadedOperator();

cout << "PrintListNormalAndReverseOrder" << endl;

cout << endl;

PrintListNormalAndReverseOrder();

cout << endl;

cout << " PrintListWithForeachWithSTL " << endl;

PrintListWithForeachWithSTL ();

cout << endl;

cout << " FindElementWithSTL " << endl;

FindElementWithSTL();

cout << endl;

return 0;

}