

# Übung2:

## 1. Lösungs-Idee:

Es wird eine Template Klassen entwickelt welche eine HashTable realisiert. Dazu bekamen wir schon ein vorgefertigtes Template, welche Methoden zu implementieren sind.

Ich habe mir nach einiger Recherche überlegt, dass ich als Daten Komponente für meine HashTable einen `Vector<List<V>>` nehmen werde. Der Grund liegt daran, dass bei einem Vector der Zugriff direkt stattfinden kann. Die Liste dient dazu, um mögliche Kollisionen von Hashwerten vorzubeugen.

Der Vector wird im Konstruktor mit der übergebenen Größe `n_buckets` initialisiert. Fügt man nun ein Element ein, sieht dies folgendermaßen aus. Es wird der Hashwert berechnet mittels, der als Template Parameter übergebenen Hashfunktion. Danach wird Modulo `n_buckets` gerechnet. Das Ergebnis ist diejenige Position, an der ich beim Vector einfüge. Bevor überhaupt ein Wert eingefügt wird, stelle ich mit der Methode „contains“ sicher, dass der Wert nicht schon mal eingefügt wurde. Meine HashTable soll keine doppelten Werte speichern können.

Einfügen kann ganz einfach erfolgen indem man sich die Liste holt, also `data[myGeneratedKey].push_back`. fügt das Element einfach hinzu. Dadurch, dass der Vector am Anfang bis `n_buckets` initialisiert wurde, kann da auch kein Problem auftreten.

Das Lösungsverfahren ergibt sich danach aus dem einfüge verfahren, und braucht glaube ich nicht näher beleuchtet werden.

Wird ein Wert hinzugefügt, oder gelöscht, muss danach die Kapazität geprüft werden. HashTables würden ansonsten entarten, da man zu viele Kollisionen zusammenbekommt. Je weniger Kollisionen, desto schneller ist der Zugriff. Bei einem rehash merke ich mir die Daten setze danach die `n_buckets` korrigiert, je nach dem nach oben oder unten, danach wird die eigentliche hashtable geleert und alle Werte der Hilfs Tabelle werden wieder eingefügt.

Den Iterator kann ich für meine Datenstruktur nun relativ einfach schreiben. Ich merke mir lediglich intern die hashtable welche ich über den Konstruktor bekomme, sowie einen Iterator vom Vector und einen von der Liste.

Wird nun ein `Iterator++` aufgerufen versuche ich als erstes den `ListIterator` zu erhöhen, ist dieser am Ende, wird der `VectorIterator` erhöht. Dieser wird so lange erhöht, bis er entweder am Ende ist, oder an einer Position wo wieder eine Liste dranhängt. Sind beide am Ende wird Ende zurückgegeben.

Die Funktion `Iterator--` funktioniert eigentlich genau gleich, nur in die entgegen gesetzte Richtung, solange zuerst `listIterator` dann `VectorIterator` usw. bis beide auf begin sind.

## Code:

HashTable.hpp:

```

#ifndef HASHTABLE_H
#define HASHTABLE_H

#include<algorithm>
#include <map>
#include <list>
#include <iostream>
#include <iterator>
#include <iomanip>

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);

private:
    /***** TYPE DEFS *****/
    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 std::ptrdiff_t difference_type;
    typedef const_pointer pointer;
    typedef const_reference reference;
    typedef std::size_t size_type;
    /***** MEMBERS *****/
    std::vector<std::list<V>> data;
    size_t currentSize;
    size_t buckets;
    hash_function_type hasher;
    key_equal_function_type equals;
    double minLoadFactor;
    double maxLoadFactor;
    /***** METHODS *****/
    size_t calculateKey(const V& value) const{
        auto result = (hasher(value) % this->buckets);
        if(result < 0){
            result = result + this->buckets;
        }
        return result;
    }

public:
    /***** CONSTRUCTOR *****/
    HashTable(size_t n_buckets = 50,
               hash_function_type hasher = std::hash<V>(),
               key_equal_function_type equals = std::equal_to<V>(),
               double max_load_factor = 0.8,

```

```

        double min_load_factor = 0.2):
        buckets(n_buckets), hasher(hasher), equals(equals),
        minLoadFactor(min_load_factor),
maxLoadFactor(max_load_factor),

currentSize(0), data(std::vector<std::list<V>>(n_buckets)){

}
/***** DESTRUCTOR *****/
virtual ~HashTable(){};
/***** GETTER / SETTER *****/

/***** METHODS *****/
/**
 * Insert new value into HashTable
 * If Collision happens the value is attached on position Collision
 * It is not possible to Insert equal values multiple times
 */
void insert(const V &value){
    // calculate position first
    if(this->contains(value)){
        std::cout << "Element allready in HashTable" << std::endl;
        return;
    }
    size_t key = this->calculateKey(value);
    // insert value on correct position
    // with using array index and pushback the element get
    // attached if there is a collision with our hashvalue
    (data[key]).push_back(value);
    currentSize++;
    if(load_factor() > maxLoadFactor){
        rehash((int)(buckets * 2));
    }
}

/**
 * Erase the given Element from HashTable
 */
void erase(const V &value){
    if(!this->contains(value)){
        std::cout << "Element is not in HashTable" << std::endl;
        return;
    }
    size_t key = calculateKey(value);
    auto it = data[key];
    data[key].remove_if([&](const V &itemVal) {
        return equals(itemVal, value);
    });
    currentSize--;
    if(load_factor() < minLoadFactor){
        rehash((int)(buckets * 0.5));
    }
}

/**
 * Checks if an Element is in HashTable
 */
bool contains(const V &value) const{
    bool found = false;
    size_t key = calculateKey(value);
    std::list<V> it = data[key];
    if(it.empty()){

```

```

        return false;
    }
    for_each((it).begin(), (it).end(), [&](V v){
        if(equals(v,value)){
            found = true;
        }
    });
    return found;
}
/**
 * Rehash the table with new size of buckets
 */
void rehash(size_t new_n_buckets){
    this->buckets = new_n_buckets;
    std::vector<std::list<V>> help = this->data;
    this->data = std::vector<std::list<V>>(new_n_buckets);
    this->currentSize = 0;

    for_each(help.begin(), help.end(), [&](std::list<V> l){
        for_each(l.begin(), l.end(), [&](V value){
            this->insert(value);
        });
    });
}
/**
 * returns the load_factor of table
 */
double load_factor()const{
    if(this->buckets > 0){
        return this->currentSize / this->buckets;
    }
    return 0.0;
}
/**
 * return size of hashTable
 */
size_t size()const{
    return this->currentSize;
}
/**
 * returns capacity of hashTable
 */
size_t capacity()const{
    return this->buckets;
}
/**
 * check if HashTable is empty
 */
bool empty() const{
    return this->capacity == 0;
}
/**
 * check equality of two hashTables
 */
bool operator == (const HashTable &other) const{
    if(*this == other) return true;
    if(size() != other.size()) return false;
    for(auto element: other){
        if(!contains(element)){
            return false;
        }
    }
}
}

```

```

/**** ITERATOR *****/
typedef std::iterator <std::bidirectional_iterator_tag,
                    value_type,
                    difference_type,
                    const_pointer,
                    const_reference> iterator_base;

class const_iterator : public iterator_base {
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;
    typedef typename iterator_base::value_type         value_type;
private:
    HashTable &table;
    typename std::vector<std::list<value_type>>::iterator
vectorIterator;
    typename std::list<value_type>::iterator listIterator;

    /**
     * calculates the next valid position
     */
    void calculateNextNotEmptyPos(bool begin){
        if(!begin) vectorIterator++;
        while(vectorIterator != table.data.end() &&
(*vectorIterator).size() == 0){
            vectorIterator ++;
        }
        if(vectorIterator != table.data.end()){
            listIterator = (*vectorIterator).begin();
        }else{
            setToEnd();
        }
    }

    /**
     * calculates the previous valid position
     */
    void calculatePrevNotEmptyPos(bool end){
        if(!end) vectorIterator--;
        while(vectorIterator != table.data.begin() &&
(*vectorIterator).size() == 0){
            vectorIterator--;
        }
        if((*vectorIterator).size != 0){
            listIterator = --(*vectorIterator).end();
            return;
        }
        setToBegin();
    }

    /**
     * Set Iterator to first possible value
     */
    void setToBegin(){
        vectorIterator = table.data.begin();
        calculateNextNotEmptyPos(true);
    }

    /**
     * Set Iterator to last possible value
     */
    void setToEnd(){

```

```

        vectorIterator = --table.data.end();
        listIterator = (*vectorIterator).end();

    }
public:
    /**
     * constructor
     */
    const_iterator (HashTable &table, bool begin) : table(table){
        if(begin){
            setToBegin();
        }else{
            setToEnd();
        }
    }
    bool operator == (const_iterator const & rhs) const{
        return (rhs.vectorIterator == vectorIterator &&
rhs.listIterator == listIterator);
    }
    bool operator != (const_iterator const & rhs) const{
        return !(operator==(rhs));
    }

    reference operator * () const{
        return *listIterator;
    }
    pointer operator -> () const{
        return &(*listIterator);
    }

    const_iterator & operator ++ (){
        listIterator++;
        if(this->listIterator != (*vectorIterator).end()){
            return *this;
        }
        vectorIterator++;
        calculateNextNotEmptyPos(false);
        return *this;
    }
    const_iterator & operator -- (){
        if(this->listIterator != (*vectorIterator).begin()){
            listIterator--;
            return *this;
        }
        vectorIterator--;
        calculatePrevNotEmptyPos(false);
        return *this;
    }

    const_iterator operator ++ (int){
        const_iterator help = *this;
        help.operator++();
        return help;
    }
    const_iterator operator -- (int){
        const_iterator help = *this;
        help.operator--();
        return help;
    }

};

typedef const_iterator iterator;

```

```
const_iterator begin() const{
    return const_iterator(const_cast<HashTable&>(*this), true);
}
const_iterator end() const{
    return const_iterator(const_cast<HashTable&>(*this), false);
}

iterator cbegin(){
    return begin();
}
iterator cend(){
    return end();
}
const_iterator cbegin() const{
    return begin();
}
const_iterator cend() const{
    return end();
}
};
/**
 * friend method for output
 */
template<typename V, typename H, typename C>
std::ostream & operator << (std::ostream & os, const HashTable<V, H, C>
&ht) {
    for_each(ht.data.begin(), ht.data.end(), [&](std::list<V> l){
        for_each(l.begin(), l.end(), [&](V value){
            os << " [ " << value << " ] " << std::endl;
        });
    });
    return os;
}
#endif
```

Main.cpp:

```
#include <iostream>
#include "../include/HashTable.h"
using namespace std;

int main()
{
    HashTable<int, std::hash<int>, std::equal_to<int>> table;
    cout << "Empty Table: " << endl << table << endl;
    cout << "Insert values 1,2,3,4" << endl;
    table.insert(1);
    table.insert(2);
    table.insert(3);
    table.insert(4);

    cout << "Table: " << endl << table << endl;

    table.erase(1);

    cout << "Removed 1" << endl << table << endl;
    cout << "Contains: 2 ? " << table.contains(2) << endl;
    cout << "Contains with not containing value 8 ?" << table.contains(8)
<< endl;

    cout << "Rehash Table to try producing kollisions" << endl << endl;
    table.rehash(2);

    cout << "Table rehashed: " << endl << table << endl;

    table.insert(1);
    table.insert(2);
    cout << "Added value 1 and already containing value 2 " << endl <<
table << endl;

    table.insert(10);
    table.erase(2);
    cout << "Added 10 Removed 2: " << endl << table << endl;

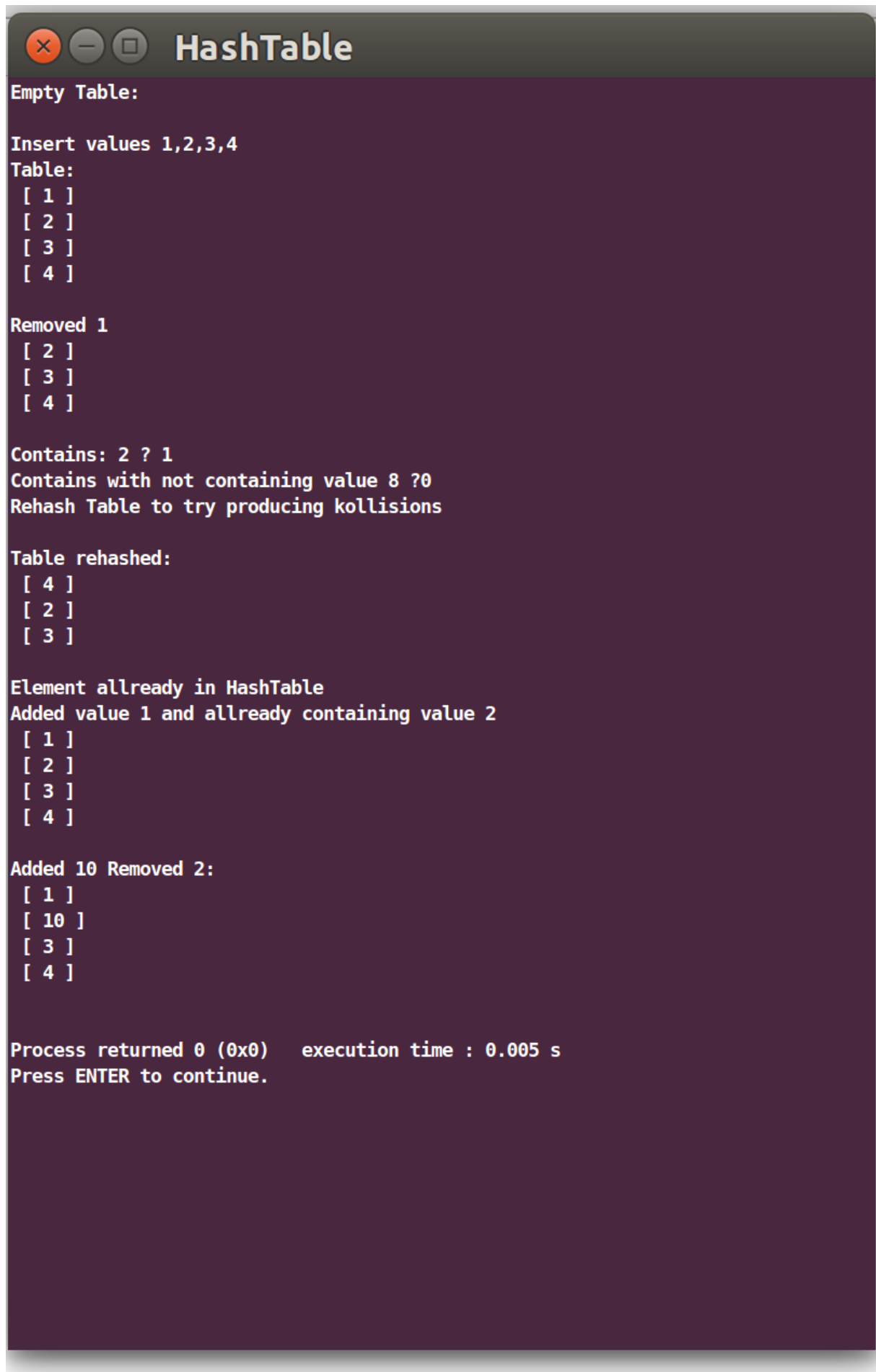
    return 0;
}
```



## 2. Test-Case:

Es wurden alle Methoden der Klasse während der Entwicklung getestet.

Als Abschlusstest wurde danach noch das oben ersichtliche Hauptprogramm entwickelt, und ausgeführt.



```
HashTable
Empty Table:

Insert values 1,2,3,4
Table:
[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]

Removed 1
[ 2 ]
[ 3 ]
[ 4 ]

Contains: 2 ? 1
Contains with not containing value 8 ?0
Rehash Table to try producing kollisions

Table rehashed:
[ 4 ]
[ 2 ]
[ 3 ]

Element already in HashTable
Added value 1 and already containing value 2
[ 1 ]
[ 2 ]
[ 3 ]
[ 4 ]

Added 10 Removed 2:
[ 1 ]
[ 10 ]
[ 3 ]
[ 4 ]

Process returned 0 (0x0)   execution time : 0.005 s
Press ENTER to continue.
```

