Übung2:

# 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 folgender maßen 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 Löschverfahren 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 setzte 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 hahstable 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 elment 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 capacitiy**()**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 previouse 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 allready containing value 2 " **<<** endl **<<** table **<<** endl**;**

table**.**insert**(**10**);**

table**.**erase**(**2**);**

cout **<<** "Added 10 Removed 2: " **<<** endl **<<** table **<<** endl**;**

**return** 0**;**

**}**

# Test-Case:

Es wurden alle Methoden der Klasse während der Entwicklung getestet.   
Als Abschlusstest wurde danach noch das oben ersichtliche Hauptprogramm entwickelt, und ausgeführt.

