SWE 4x Übung zu Softwareentwicklung mit modernen Plattformen 4 — SS 2015, Übung 1

Abgabetermin: Ostersamstag in der KW 14

☐ Gr. 1, E. Pitzer
☐ Gr. 2, F. Gruber-Leitner

Name _____ **Aufwand in h** _____

Punkte _____ **Kurzzeichen Tutor / Übungsleiter** _____ / _____

STL Hashtable (12 + 12 Punkte)

Da wir nun bereits viele Containerklassen der STL kennengelernt haben wird es an der Zeit zu versuchen selbst einen generischen Container zu implementieren. Dazu bietet sich eine einfache Hashtabelle an, die einigermaßen flexibel sein soll. Wir wollen, der Einfachheit halber, lineare Verkettung zur Auflösung von Kollisionen verwenden und nur die wichtigsten Operationen anbieten, sowie die Ausgabe auf einen Stream ermöglichen.

Außerdem wollen wir die eigentliche Hashfunktion sowie die Gleichheit von Elementen konfigurierbar machen. Dazu können wir im ersten Schritt gleich die Funktoren `std::hash()` und `std::equal_to()` der STL verwenden. Da es sich bei diesen Funktionen sowohl um einfache C Funktionszeiger als auch um C++ Funktoren handeln kann, müssen die Typen dieser Funktionen als Templateparameter deklariert werden. Es kommen also zusätzlich zum Templateparameter `V` für den Wertetype (value) noch die Templateparameter `H` für die Hashfunktion sowie `C` für die Vergleichsfunktion (compare) dazu.

Als krönenden Abschluss brauchen wir noch die Implementierung eines einfachen Iterators um auch die bereits bestehenden Algorithmen der STL auf unsere Hashtabelle anwenden zu können.

1. Implementieren Sie dir Grundfunktionalität einer STL-kompatiblen Hashtabelle, die mindestens die folgende Schnittstelle erfüllen muss und testen Sie Ihre Implementierung ausführlich.

```
template<typename V, typename H, typename C>
class hashtable {
public:
  void insert(const V &value);
  void erase(const V &value);
  bool contains(const V &value);
  void rehash(size_t new_n_buckets);

  double load_factor() const;
  size_t size() const;
  size_t capacity() const;
  bool empty() const;
};

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

Überlegen Sie sich dazu eine geeignete Komposition aus STL Containern um diese Funktionalität in Form einer Hashtabelle mit linearer Verkettung zu erhalten.

2. Erweitern Sie Ihre Implementierung nun auch um einen einfachen Iterator den Sie z.B. von `std::iterator` ableiten können.

```
typedef std::iterator <std::bidirectional_iterator_tag,
                       value_type,
                       difference_type,
                       const_pointer,
                       const_reference> iterator_base;

class const_iterator : public iterator_base {
public:
  bool operator == (const_iterator const & rhs) const;
  bool operator != (const_iterator const & rhs) const;
  reference operator *  () const;
  pointer   operator -> () const;

  const_iterator & operator ++ ();
  const_iterator & operator -- ();

  const_iterator operator ++ (int);
  const_iterator operator -- (int);
};
```

Abschließend können Sie diesen Iterator dann auch noch verwenden um den Vergleichsoperator für Hashtabellen so zu überschreiben, dass zwei Hashtabellen dann gleich sind, wenn sie die gleichen Elemente enthalten.

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

Hinweis: Um unnötige Stolperfallen im Umgang mit der STL zu vermeiden, verwenden Sie am besten die Vorlage `hashtable_template.hpp` die Sie nur mehr erweitern müssen.

1 HashTable

Folgend ist die Dokumentation der Aufgabenstellung HashTable angeführt.
Als Vorlage soll das zur Verfügung gestellte Template verwendet werden.
Die Implementierung soll innerhalb des Templates (*.hpp) erfolgen und nicht in eine *.cpp Datei ausgelagert werden.
Als Container soll ein *vector* verwendet werden, der wiederum als Werte eine Liste enthält die letztendlich die Werte vom Typ V enthält.
Es soll ein *vector* verwendet werden da dieser es erlaubt mittels *operator[]* auf die Elemente des Containers zuzugreifen. Somit ist es möglich mittels Indexierung über den ermittelten Hash Index auf die *vector* Elemente zuzugreifen.

Falls der *load factor* den maximal erlaubten *load factor* überschreitet soll der *vector* um den Faktor 1.5 vergrößert werden.
Falls der *load factor* den minimal erlaubten *load factor* unterschreitet soll der *vector* um den Faktor 0.5 verkleinert werden, jedoch nur wenn die definierte minimale Kapazität nicht bei der Verkleinerung unterschritten wird. (Sollte 10 sein, steht aber frei)

Für die Behandlung der Kollision soll der Chaining Mechanismus verwendet werden, wobei die Elemente mit demselben Hash Index in der Liste miteinander verkettet werden. Da der *vector* bereits *list* Elemente enthält kann hierbei einfach über *table[idx].push_front(..)* ein Element hinzugefügt werden. Bevor ein Element hinzugefügt wird muss ermittelt werden ob sich dieses Element nicht bereits in der Chain befindet, da ansonsten Duplikate auftreten würden.

Für die Implementierung von operator== sei folgendes angemerkt.
Zwei Hashtables sind logisch gleich wenn einer der beiden folgenden Bedingungen erfüllt ist:

1. Es handelt sich um dieselbe Instanz
2. Beide Hashtables haben dieselbe Größe und beinhalten dieselben Elemente.

Die beiden Methoden *insert* und *erease* sollen so implementiert werden, sodass keine Aktion ausgeführt wird wenn ein Element entweder bereits vorhanden ist (*insert*) oder nicht vorhanden ist (*erease*). Es sollen keine Fehler gemeldet werden.

Hilfsmethoden sowie die verwendeten Container sollen innerhalb des *private* Bereichs gehalten werden. Zugriffe sollen ausschließlich über, die zur Verfügung gestellten Schnittstellen erfolgen.

1.1 Implementierung

1.1.1 hashtable.hpp

Folgend ist die Implementierung der Aufgabenstellung angeführt.

```
#ifndef hashtable_hpp
#define hashtable_hpp

#include <cassert>
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <iomanip>
#include <climits>
#include <math.h>
#include <bitset>
#include <algorithm>
#include <sstream>

using namespace std;

template<typename V, typename H, typename C>
class hashtable;

/**
 * Overloaded output operator
 */
template<typename V, typename H, typename C>
std::ostream & operator <<(std::ostream & os, const hashtable<V, H, C> &ht) {
    for_each(ht.table.begin(), ht.table.end(),
            ([&ht, &os](const list<V>& l) {
                for_each(l.begin(), l.end(), ([&ht, &os](const V& value) {
                                os << "hash-key: " << ht.calculateHashIdx(value) << " - " << value << endl << flush;
                            }));
            }));
    return os;
} /* operator << */

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 std::ptrdiff_t difference_type;
    typedef const_pointer pointer;
    typedef const_reference reference;
    typedef std::size_t size_type;

private:
    size_t currentCapacity;
    hash_function_type hasher;
    key_equal_function_type equals;
    double maxLoadFactor;
    double minLoadFactor;
    const int MIN_CAPACITY = 10;
    int elementCount = 0;
    vector<list<value_type>> table;

    /**
     * This method calculates the hash index for the bakced table depending on the current set capacity.
     * @param
     *      const V& value: the value to calculates hash index for
     * @return
     *      the calculated hash index
     */
    int calculateHashIdx(const V& value) const {
        return hasher(value) % currentCapacity;
    } /* calculateHashIdx */

    /**
     * Validates if an rehash of the backed values is necessary.
     */
    void validateForReHash() {
        double currentLoad = load_factor();
        double factor = 0;
        if ((currentLoad < minLoadFactor)
                && ((currentCapacity * 0.5) > MIN_CAPACITY)) {
            factor = 0.5;
        } else if (currentLoad > maxLoadFactor) {
            factor = 1.5;
        } /* if */

        if (factor != 0) {
            rehash(currentCapacity * factor);
        } /* if */
    } /* validateForReHash */

public:
```

```
/**
 * Default constructor which initializes all members but the buckets.
 */
hashtable(size_t n_buckets = 10, 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) :
        currentCapacity(n_buckets), hasher(hasher), equals(equals), maxLoadFactor(
                max_load_factor), minLoadFactor(min_load_factor) {
    clear();
} /* hashtable */

/**
 * No dynamic members therefore nothing to do.
 */
virtual ~hashtable() {
    // nothing to do
} /* ~hashtable */

void clear() {
    elementCount = 0;
    table.clear();
    if (currentCapacity < MIN_CAPACITY) {
        currentCapacity = MIN_CAPACITY;
    } /* if */
    table.resize(currentCapacity);
} /* clear */

/**
 * Inserts the value in the given list if this value is not managed by this list. Does nothing otherwise.
 * @param:
 *      const V& value: the value to inserted
 */
void insert(const V &value) {
    validateForReHash();
    size_t idx = calculateHashIdx(value);
    if (find(table[idx].begin(), table[idx].end(), value)
            == table[idx].end()) {
        table[idx].push_front(value);
        elementCount++;
    } /* if */
} /* insert */

/**
 * Deletes the value from the hashtable if this value is managed by this hashtable. Does nothing otherwise.
 * @param:
 *      const V& value: the value to be removed from the hastable
 */
void erase(const V &value) {
    size_t idx = calculateHashIdx(value);
    list<string>& valueList = table[idx];
    auto it = find(valueList.begin(), valueList.end(), value);
    if (it != valueList.end()) {
        valueList.remove(value);
        elementCount--;
    } /* if */
    validateForReHash();
} /* erase */

/**
 * Answers tthe question if this hashtable contains the given value
 * @param
 *      const V& value: the values to be searched in the table
 * @return
 *      true if this value is managed by this hastable, false otherwise
 */
bool contains(const V &value) const {
    size_t idx = calculateHashIdx(value);
    list<string> valueList = table[idx];
    auto it = find(valueList.begin(), valueList.end(), value);
    return (it != valueList.end());
} /* contains */

/**
 * Rehashes the hashtable entries and rebuilds the map with the new has indexes
 * @param:
 *      the new buckets
 */
void rehash(size_t newCapacity) {
    currentCapacity = newCapacity;
    vector<list<value_type>> oldData(table);
    table.clear();
    table.resize(currentCapacity);
    for_each(oldData.begin(), oldData.end(),
            ([this](const list<value_type> values) {
                for_each(values.begin(), values.end(), ([this](const string value) {
                                size_t idx = calculateHashIdx(value);
                                table[idx].push_front(value);
                            }));
            }));
} /* rehash */

/**
 * Gets the current load factor
 * @return
 *      the current load factor
 */
double load_factor() const {
    size_t curSize = size();
    return (curSize > 0) ? ((double) (size()) / (currentCapacity)) : 0.0;
} /* load_factor */
```

```
/**
 * Gets the current size.
 * @return
 *      the current size
 */
size_t size() const {
    return elementCount;
} /* size */

/**
 * Returns the current capacity
 * @return
 *      the current capacity
 */
size_t capacity() {
    return currentCapacity;
} /* capacity */

/**
 * Answers the question if this hashtable is empty
 * @return
 *      true if this instance has no entries, false otherwise
 */
bool empty() const {
    return size() == 0;
} /* empty */

/**
 * hashtables are logically equal if the fit one of the following requirements.
 * 1. Same instance
 * 2. Other instance
 *      a. Same size
 *      b. Contain same elements
 *
 * @param
 *      const hashtable& other: the other instance to compare with
 * @return
 *      true if the hastables are logically equal, false otherwise
 */
bool operator ==(const hashtable &other) const {
    bool result = false;
    // same instance
    if (this == (&other)) {
        result = true;
    }
    // Other instance has same size
    else if (size() == other.size()) {
        int i = 0;
        result = true;
        // check if other contains same elements
        while ((result) && (i < other.table.size())) {
            auto it = other.table[i].begin();
            while ((result) && (it != other.table[i].end())) {
                result = contains((*it));
                it++;
            } /* while */
            i++;
        } /* while */
    } /* if */

    return result;
} /* operator== */

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

/**
 * The hashtable iterator class
 * NOT WORKING AT ALL !!!!!!!!!!!!!!!!!!!!!!!!
 */
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:
    int idx = 0;
    const vector<list<value_type>>& table;
    list<string> values;
    const bool isEnd;
    const bool reverse;
    typename list<value_type>::const_iterator listIt;

    void prepareIterator() {
        if (isEnd) {
            idx = (table.size() - 1);
            values = table.at(idx);
            listIt = values.end();
            if (reverse) {
                setNextIterator();
            }
        } else {
            idx = 0;
            values = table.at(idx);
            listIt = values.begin();
            if (!reverse) {
                setNextIterator();
            }
        }
    }
```

```
        void setNextIterator() {
            if (isEnd) {
                while ((idx != 0) && (listIt == values.begin())) {
                    idx--;
                    values = table.at(idx);
                    listIt = values.end();
                }
            } else {
                while ((idx != (table.size() - 1)) && (listIt == values.end())) {
                    idx++;
                    values = table.at(idx);
                    listIt = values.begin();
                }
            }
        }

    public:
        const_iterator(const vector<list<value_type>>& table,
                bool isEnd = false, bool reverse = false) :
                table(table), isEnd(isEnd), reverse(reverse) {
            prepareIterator();
        }

        bool operator ==(const_iterator const & rhs) const {
            return (idx == rhs.idx) && (listIt == rhs.listIt);
        }
        bool operator !=(const_iterator const & rhs) const {
            return (idx != rhs.idx) && (listIt != rhs.listIt);
        }

        reference operator *() const {
            return *listIt;
        }

        pointer operator ->() const {
            return listIt;
        }

        const_iterator & operator ++() {
            // List still has elements
            if (listIt != values.end()) {
                listIt++;
            }
            // End reached therefore start form new
            if ((listIt != values.end()) && (idx < table.size())) {
                idx++;
                values = table[idx];
                listIt = values.begin();
                setNextIterator();
            }
            return (*this);
        }

        const_iterator & operator --() {
            // List still has elements
            if (listIt != values.begin()) {
                listIt--;
            }
            // list end reached
            if ((listIt != values.begin()) && (idx != 0)) {
                idx--;
                values = table[idx];
                listIt = values.end();
                setNextIterator();
            }
            return (*this);
        }

        const_iterator operator ++(int) {
            const_iterator res(*this);
            ++(*this);
            return res;
        }
        const_iterator operator --(int) {
            const_iterator res(*this);
            --(*this);
            return res;
        }
    };

    typedef const_iterator iterator;

    const_iterator begin() const {
        return const_iterator(table);
    } /*begin */

    const_iterator end() const {
        return const_iterator(table, true);
    } /* end */

    const_iterator rbegin() const {
        return const_iterator(table, false, true);
    } /* rbegin */

    const_iterator rend() const {
        return const_iterator(table, true, true);
    } /* rend */
};

#endif // hashtable_hpp
```

1.1.2 main.cpp

Folgend ist die Main Source angeführt, die die Tests Funktionen ausführt.

```
/*
 * main.cpp
 * This is the main which invokes the tests.
 *
 *  Created on: Mar 27, 2015
 *      Author: Thomas Herzog
 */
#include "test/hashtableTests.hpp"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"

using namespace std;
using namespace cute;

static void runSuite(int argc, char const *argv[]) {
    suite setTestSuite, bagTestSuite;
    xml_file_opener xmlfile(argc, argv);
    xml_listener<ide_listener<> > lis(xmlfile.out);

    setTestSuite = createTestSuite();
    makeRunner(lis, argc, argv)(setTestSuite, "AllTests");
} /* runSuite */

int main(int argc, char const *argv[]) {
    runSuite(argc, argv);
} /* main */
```

1.1.3 hashtableTests.hpp

Folgend ist die Spezifikation der Tests angeführt.

```
/*
 * hashtableTests.hpp
 * This header file specifies the test for the hashtable implementation
 *
 *  Created on: Apr 3, 2015
 *      Author: Thomas Herzog
 */

#ifndef HASHTABLETESTS_HPP_
#define HASHTABLETESTS_HPP_

#include <cute.h>
#include "../hashtable.hpp"

cute::suite createTestSuite();

//#####################################################
// hashtable tests
//#####################################################
void test_insert_new();
void test_insert_duplicate();


void test_erease_existing();

void test_erease_not_existing();

void test_contains_true();

void test_contains_false();

void test_clear_empty();

void test_clear_full();

void test_increase_load_factor();

void test_decrease_load_factor();

void test_equal_operator_same_instance();

void test_equal_operator_different_instance_true();

void test_equal_operator_different_instance_false();

#endif /* HASHTABLETESTS_HPP_ */
```

1.1.4 hashTableTests.cpp

Folgend ist die Implementierung der hashtableTests.hpp angeführt.

```
/*
 * hashtableTests.cpp
 * This is the implementation of the specification hashtableTests.hpp
 *
 *  Created on: Apr 3, 2015
 *      Author: Thomas Herzog
 */
#include "hashtableTests.hpp"

using namespace std;
using namespace cute;

suite createTestSuite() {
    suite testSuite;
    testSuite.push_back(CUTE(test_insert_new));
    testSuite.push_back(CUTE(test_insert_duplicate));
    testSuite.push_back(CUTE(test_erease_existing));
    testSuite.push_back(CUTE(test_erease_not_existing));
    testSuite.push_back(CUTE(test_contains_true));
    testSuite.push_back(CUTE(test_contains_false));
    testSuite.push_back(CUTE(test_clear_empty));
    testSuite.push_back(CUTE(test_clear_full));
    testSuite.push_back(CUTE(test_increase_load_factor));
    testSuite.push_back(CUTE(test_decrease_load_factor));
    testSuite.push_back(CUTE(test_equal_operator_same_instance));
    testSuite.push_back(CUTE(test_equal_operator_different_instance_true));
    testSuite.push_back(CUTE(test_equal_operator_different_instance_false));
    return testSuite;
} /* createTestSuite */

static void printTable(
        hashtable<string, std::hash<string>, std::equal_to<string>> ht,
        string desc) {
    cout << "Hashtable content: " << desc << endl << "-------------------------"
            << endl << ht << endl << "-------------------------" << endl;
} /* printTable */

//#####################################################
// hashtable tests
//#####################################################
void test_insert_new() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    printTable(ht, "before insert");

    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(string("C++"));

    printTable(ht, "after insert");

    ASSERT_EQUAL(3, ht.size());
} /* test_insert_new */

void test_insert_duplicate() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    printTable(ht, "before insert");

    ht.insert(string("Hello"));
    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(string("you"));
    ht.insert(string("C++"));
    ht.insert(string("C++"));

    printTable(ht, "after insert");

    ASSERT_EQUAL(3, ht.size());
} /* test_insert_duplicate */

void test_erease_existing() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    string s("C++");
    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(s);

    printTable(ht, "before erease");

    ht.erase(s);

    printTable(ht, "after erease");

    ASSERT_EQUAL(2, ht.size());
} /* test_erease_existing */

void test_erease_not_existing() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    string s("not-existing");
    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(string("C++"));

    printTable(ht, "before erease");
```

```
    ht.erase(s);

    printTable(ht, "after erease");

    ASSERT_EQUAL(3, ht.size());

} /* test_erease_not_existing */

void test_contains_true() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    string s("C++");
    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(s);

    printTable(ht, "");

    ASSERT_EQUAL(true, ht.contains(s));
} /* test_contains_true */

void test_contains_false() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    string s("not-existing");
    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(string("C++"));

    printTable(ht, "");

    ASSERT_EQUAL(false, ht.contains(s));
} /* test_contains_false */

void test_clear_empty() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    printTable(ht, "before clear");

    ht.clear();

    printTable(ht, "after clear");

    ASSERT_EQUAL(0, ht.size());
} /* test_clear_empty */

void test_clear_full() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    ht.insert(string("Hello"));
    ht.insert(string("you"));
    ht.insert(string("C++"));

    printTable(ht, "before clear");

    ht.clear();

    printTable(ht, "after clear");

    ASSERT_EQUAL(0, ht.size());
} /* test_clear_full */

void test_increase_load_factor() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    for (int i = 0; i < 20; i++) {
        stringstream ss;
        ss << "Element_" << i;
        ht.insert(ss.str());
    }

    ASSERT_EQUAL(33, ht.capacity());
} /* test_increase_load_factor */

void test_decrease_load_factor() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(30);

    for (int i = 0; i < 20; i++) {
        stringstream ss;
        ss << "Element_" << i;
        ht.insert(ss.str());
    }

    for (int i = 0; i < 20; ++i) {
        stringstream ss;
        ss << "Element_" << i;
        ht.erase(ss.str());
    }

    ASSERT_EQUAL(16, ht.capacity());

} /* test_decrease_load_factor */

void test_equal_operator_same_instance() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht(10);

    ht.insert(string("Hello"));

    ASSERT_EQUAL(true, ht == ht);
} /* test_equal_operator_same_instance */
```

```
void test_equal_operator_different_instance_true() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht1(10);
    hashtable<string, std::hash<string>, std::equal_to<string>> ht2(10);

    ht1.insert(string("Hello"));
    ht2.insert(string("Hello"));

    ASSERT_EQUAL(true, ht1 == ht2);
} /* test_equal_operator_different_instance_true */

void test_equal_operator_different_instance_false() {
    hashtable<string, std::hash<string>, std::equal_to<string>> ht1(10);
    hashtable<string, std::hash<string>, std::equal_to<string>> ht2(10);

    ht1.insert(string("Hello"));
    ht2.insert(string("You too"));

    ASSERT_EQUAL(false, ht1 == ht2);

} /* test_equal_operator_different_instance_false*/
```

1.2 Tests

Folgend sind die Tests angeführt.

```
#beginning AllTests 13

#starting test_insert_new
Hashtable content: before insert
------------------------

------------------------
Hashtable content: after insert
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------

#success test_insert_new OK

#starting test_insert_duplicate
Hashtable content: before insert
------------------------

------------------------
Hashtable content: after insert
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------

#success test_insert_duplicate OK

#starting test_erease_existing
Hashtable content: before erease
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------
Hashtable content: after erease
------------------------
hash-key: 3 - Hello
hash-key: 4 - you

------------------------

#success test_erease_existing OK

#starting test_erease_not_existing
Hashtable content: before erease
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------
Hashtable content: after erease
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------

#success test_erease_not_existing OK
```

```
#starting test_contains_true
Hashtable content:
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------

#success test_contains_true OK

#starting test_contains_false
Hashtable content:
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------

#success test_contains_false OK

#starting test_clear_empty
Hashtable content: before clear
------------------------

------------------------
Hashtable content: after clear
------------------------

------------------------

#success test_clear_empty OK

#starting test_clear_full
Hashtable content: before clear
------------------------
hash-key: 3 - Hello
hash-key: 4 - you
hash-key: 6 - C++

------------------------
Hashtable content: after clear
------------------------

------------------------

#success test_clear_full OK


#starting test_increase_load_factor

#success test_increase_load_factor OK

#starting test_decrease_load_factor

#success test_decrease_load_factor OK

#starting test_equal_operator_same_instance

#success test_equal_operator_same_instance OK

#starting test_equal_operator_different_instance_true

#success test_equal_operator_different_instance_true OK

#starting test_equal_operator_different_instance_false

#success test_equal_operator_different_instance_false OK

#ending AllTests
```