VE281 — Data Structures and Algorithms

Programming Assignment 2

Instructor: Hongyi Xin

— UM-SJTU-JI (Fall 2021)

Notes

- Due Date: 10/31/2021
- Submission: on JOJ (`joj.sjtu.edu.cn`)

1 Introduction

In this project, you are asked to implement a STL-like hash table data structure.

In the Standard Template Library, hash table is implemented as `std::unordered_map`[1], which is introduced in the C++11 standard. It is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity. It is named "unordered" because there already exists an ordered data structure of key-value pairs based on RB-tree, `std::map`.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. This allows fast access to individual elements, since once the hash is computed, it refers to the exact bucket the element is placed into.

The C++ standard does not mandates whether `unordered_map` use *separate chaining* or *open addressing* for collision resolution. Actually, different compilers use different policies: `g++` uses *separate chaining* and `LLVM/Clang` uses *open addressing*. In this project, you're going to use *separate chaining* with `std::forward_list`[2], which is easier to implement and maintain.

You will also have a basic knowledge on iterators in C++: what are iterators and how they work. You can further try to use range-based for loops[3] (a feature introduced in C++11) to make use of iterators.

2 Programming Assignment

2.1 The HashTable Template

2.1.1 Overview

In the project starter code, we define a template class for you:

```cpp
template<
        typename Key, typename Value,
        typename Hash = std::hash<Key>,
        typename KeyEqual = std::equal_to<Key>
>
class HashTable;
```

Here `Key` and `Value` are the types of the key-value pair stored in the hash table. `Hash` is the type of a function object, whose instance returns the hash value of a key. The standard library defines a class template `std::hash<T>`, which can be used as the default. Similarly, `KeyEqual` is another type of a function object, whose instance returns whether two

Key objects are equal. It is used to determine which key-value pair should be returned when there is a hash collision, and it also help ensure the keys are unique in the hash table.

If you want to define your own hash function and key-equal function, you can refer to the following definitions:

```
template<typename Key>
struct hash<Key> {
        size_t operator()(const Key &key) const;
}

template<typename Key>
struct equal_to<Key> {
        bool operator()(const Key &lhs, const Key &rhs) const;
}
```

2.1.2 Number of Buckets

Basically, you will use the "Hashing by Modulo" scheme. Suppose n is the number of buckets in the hash table, you should put a key into the i-th bucket where $i = hash(key) \mod n$. As we've discussed in the lecture, you should choose the hash table size n as a large prime number in ideal.

The number of buckets in your hash table should be dynamic (like a `std::vector`, but not exactly the same). A vector in C++ doubles its size when it is full (in most implementations), but this strategy may not be applied to a hash table because you should choose a prime number as the new size. Here we define the following strategy to choose a new hash table size before a rehash.

- First we define a list of prime numbers, each doubles its predecessor approximately. The list is taken from the g++ source code, and is provided with you as the file `hash_prime.hpp`.
- When a hash table is considered full (its load factor exceeds a preset maximum value), use the next prime as the new number of buckets and perform a rehash.

Furthermore, the maximum load factor value can also be changed in runtime; you can set a minimum number of buckets manually (i.e., to prevent rehashes if you've already know the data size) despite the load factor is small. **One important thing is that whenever the number of buckets is changed, you need to do a rehash.** You will fulfill all these requirements in the function `findMinimumBucketSize`. Please read its description carefully before implementing it. Though not mandatory, you are encouraged to use binary search to find the nearest prime. You can use the standard function `std::lower_bound` to perform the binary search so that you don't need to implement it by yourself.

2.1.3 Internal Data Structures

We've already defined the internal data structures for you in this project. The `HashNode` is a key-value pair. The `HashNodeList` is a single directional linked list in each bucket, here we use `std::forward_list` to implement it. The `HashTableData` is a `std::vector` of these lists, representing the buckets in the hash table.

```
class HashTable {
public:
    typedef std::pair<const Key, Value> HashNode;
    typedef std::forward_list<HashNode> HashNodeList;
```

```
    typedef std::vector<HashNodeList> HashTableData;
protected:
    HashTableData buckets;
    size_t tableSize;
    double maxLoadFactor;
    Hash hash;
    KeyEqual keyEqual;
}
```

`tableSize` means the number of key-value pairs in the hash table. You can also add your own `private` / `protected` variables (or attributes) after these if necessary, but mostly the defined ones are enough. Here we use the `protected` keyword so that your `HashTable` class can be further inherited and overwritten for testing.

2.1.4 Iterators

We're introducing iterators in this section. First of all, why you need to use iterators in this project? Supposing you want to find a key-value pair in the hash table, and then you may erase it from the hash table if it satisfies certain conditions. There are three implementations to achieved this:

(1) Use the *find* method to find a key-value pair, use the *erase* method to erase it.

(2) Use the *find* method to find a pointer to a key-value pair, use the *erase* method which accepts a pointer to erase it.

(3) Use the *find* method to find an iterator to a key-value pair, use the *erase* method which accepts an iterator to erase it.

In the first implementation, two lookups of the key is performed.

In the second implementation, only one lookup of the key is performed. However, the user can access the internal data structure of the hash table with the pointer, which is dangerous and violates the rule of packaging in Object-Oriented Programming.

In the third implementation, only one lookup of the key is performed as well, and the iterator only provides a restricted access to the key-value pair, so it's a better solution.

In this project, we've already provided you with a completed iterator. You are going to use the iterator in the implementation of other methods in the hash table.

```
class Iterator {
private:
    typedef typename HashTableData::iterator VectorIterator;
    typedef typename HashNodeList::iterator ListIterator;

    const HashTable &hashTable;
    VectorIterator bucketIt;
    ListIterator listItBefore;
    bool endFlag = false;
}
```

Here `bucketIt` means which bucket the iterator is in, `listItBefore` means the iterator pointer that points "before" the key-value pair in the linked list.

We'll give a simple explanation of the "before" iterator. If you want to erase a node in a linked list, you need to link the previous node and the next node. However, the list is single directional, so that you can't get the previous node in $\mathcal{O}(1)$ time unless you've saved it. If you always use a "before" iterator, the deletion of node will be possible, and you can access the current node by "next" of the "before" iterator. This is a built-in functionality of the `std::forward_list`[2] class. The class also provides a `before_begin` method, which is different from other STL containers. You can check the documentation for more information.

Since the list is single directional (in order to save memory), the iterator only supports single directional iteration. The operator `++` is already overloaded for you. The iterator doesn't have a const version, you can implement it if you're interested in iterators, but it's not mandatory. You can see the detail implementation of the iterator in the starter code.

We also defines a variable

```
typename HashTableData::iterator firstBucketIt;
```

in the hash table. It provides an $\mathcal{O}(1)$ access to the the first key-value pair in the hash table. You need to update its value whenever an insert, erase or rehash operation is applied to the hash table.

What's more, the hash table supports basic iteration and range-for loops[3] with the help of iterators and two methods: `begin` and `end`. More specifically, with the C++98 standard you can iterate the hash table by:

```
for (auto it = hashTable.begin(); it != hashTable.end(); ++it) {
    std::cout << it->first << " " << it->second << std::endl;
}
```

With C++11, you can use the following as an alternative:

```
for (auto &item : hashTable) {
    std::cout << item.first << " " << item.second << std::endl;
}
```

With C++17, you can even do it in a more graceful way (similar to `python` and some other modern languages):

```
for (auto &[key, value] : hashTable) {
    std::cout << key << " " << value << std::endl;
}
```

The order of the output of the above code is arbitrary and is dependent on implementation. We'll not test the internal order of the key-value pairs in your hash table (returned by iteration), since they're meant to be "unordered".

2.2 Comparison of Hash Table and Linked List

We also recommend you to study the performances of hash table and linked list. Basically, you will compare these classes:

- `HashTable`
- `std::unordered_map`
- `std::list` or `std::forward_list` (their performances should be similar)

You can design your own comparison metric. For example, you can insert n key-value pairs into all these classes, then run k find, update, insert or erase operations. In a linked list, you can use linear search, it should be very simple to implement these operations with `std::list` or `std::forward_list`.

No report is required, but you are recommended to study their differences for your better understanding in terms of hash table.

2.2.1 Hints

- The performance of programs on Windows is usually not stable, so you should do the experiments on a Unix based system.
- You may want to write another program to do this study.
- You can use the C++11 pseudo-random number generation library to generate more randomized numbers (instead of using `std::rand`).
- You can use the C++11 chrono library to get more accurate runtime of functions than `std::clock`.
- (Optional) You can use GNU Perf (only available on Linux) to find the bottleneck of your implementation.
- Although the major factor that affects the runtime is the size of the input array, however, the runtime for an algorithm may also weakly depend on the detailed input array. Thus, for each size, you should generate a number of arrays of that size and obtain the mean runtime on all these arrays. Also, for fair comparison, the same set of arrays should be applied to all the data structures.
- You should try at least 5 representative sizes.

3 Implementation Requirements and Restrictions

3.1 Requirements

- You must make sure that your code compiles successfully on a Linux operating system with `g++` and the options `-std=c++1z -Wconversion -Wall -Werror -Wextra -pedantic`.
- You should not change the definitions of the functions and variables, as well as the `public` or `protected` keywords for them in `hashtable.hpp`.
- You can define helper functions and new variables, don't forget to mark them as `private` or `protected`.
- You should only hand in one file `hashtable.hpp`.
- You can use any header file defined in the `C++17` standard. You can use cppreference as a reference.

You only need to implement the methods (functions) marked with "TODO" in the file `hashtable.hpp`. Here is a list of the methods (functions):

- Copy Constructor and Assignment Constructor
- `findMinimumBucketSize`
- `find`
- `insert`
- `erase`
- `operator[]`
- `rehash`

Please refer to the descriptions of these functions in the starter code.

3.2 Memory Leak

Hint: You're not going to use any dynamic memory allocation functions (`new`, `malloc`, etc.) directly in this project, thus it's not possible to have memory leak in your program. This section is only for your reference.

You may not leak memory in any way. To help you see if you are leaking memory, you may wish to call valgrind, which can tell whether you have any memory leaks. (You need to install valgrind first if your system does not have this program.) The command to check memory leak is:

```
valgrind --leak-check=full <COMMAND>
```

You should replace `<COMMAND>` with the actual command you use to issue the program under testing. For example, if you want to check whether running program

```
./main < input.txt
```

causes memory leak, then `<COMMAND>` should be "`./main < input.txt`". Thus, the command will be

```
valgrind --leak-check=full ./main < input.txt
```

4 Grading

Your program will be graded along five criteria:

4.1 Functional Correctness

Functional Correctness is determined by running a variety of test cases against your program, checking your solution using our automatic testing program.

4.2 Implementation Constraints

We will grade Implementation Constraints to see if you have met all of the implementation requirements and restrictions. In this project, we will also check whether your program has memory leak. For those programs that behave correctly but have memory leaks, we will deduct some points.

4.3 General Style

General Style refers to the ease with which TAs can read and understand your program, and the cleanliness and elegance of your code. Part of your grade will also be determined by the performance of your algorithm.

4.4 Performance

We will test your program with some large test cases. If your program is not able to finish within a reasonable amount of time, or it uses too much memory, you will lose the performance score for those test cases.

5 Acknowledgement

The programming assignment is co-authored by Yihao Liu, an alumni of JI and the chief architect of JOJ.

References

[1] std::unordered_map - cppreference : `https://en.cppreference.com/w/cpp/container/unordered_map`

[2] std::forward_list - cppreference : `https://en.cppreference.com/w/cpp/container/forward_list`

[3] Range-based for loop - cppreference: `https://en.cppreference.com/w/cpp/language/range-for`

Appendix

hash_prime.hpp

```
// adopted from /usr/include/c++/10.2.0/ext/pb_ds/detail/resize_policy/hash_prime_size_policy_imp.hpp

#include <utility>

namespace HashPrime {
    enum {
        num_distinct_sizes_32_bit = 30,
        num_distinct_sizes_64_bit = 62,
        num_distinct_sizes = sizeof(std::size_t) != 8 ?
                                     num_distinct_sizes_32_bit : num_distinct_sizes_64_bit,
    };

    // Originally taken from the SGI implementation; acknowledged in the docs.
    // Further modified (for 64 bits) from tr1's hashtable.
    static constexpr std::size_t g_a_sizes[num_distinct_sizes] = {
            /* 0     */              5ul,
            /* 1     */              11ul,
            /* 2     */              23ul,
            /* 3     */              47ul,
            /* 4     */              97ul,
            /* 5     */              199ul,
            /* 6     */              409ul,
            /* 7     */              823ul,
            /* 8     */              1741ul,
            /* 9     */              3469ul,
            /* 10    */              6949ul,
            /* 11    */              14033ul,
            /* 12    */              28411ul,
            /* 13    */              57557ul,
            /* 14    */              116731ul,
            /* 15    */              236897ul,
            /* 16    */              480881ul,
            /* 17    */              976369ul,
            /* 18    */              1982627ul,
            /* 19    */              4026031ul,
            /* 20    */              8175383ul,
            /* 21    */              16601593ul,
            /* 22    */              33712729ul,
            /* 23    */              68460391ul,
            /* 24    */              139022417ul,
            /* 25    */              282312799ul,
            /* 26    */              573292817ul,
            /* 27    */              1164186217ul,
```

```
            /* 28    */                2364114217ul,
            /* 29    */                4294967291ul,
            /* 30    */ (std::size_t) 8589934583ull,
            /* 31    */ (std::size_t) 17179869143ull,
            /* 32    */ (std::size_t) 34359738337ull,
            /* 33    */ (std::size_t) 68719476731ull,
            /* 34    */ (std::size_t) 137438953447ull,
            /* 35    */ (std::size_t) 274877906899ull,
            /* 36    */ (std::size_t) 549755813881ull,
            /* 37    */ (std::size_t) 1099511627689ull,
            /* 38    */ (std::size_t) 2199023255531ull,
            /* 39    */ (std::size_t) 4398046511093ull,
            /* 40    */ (std::size_t) 8796093022151ull,
            /* 41    */ (std::size_t) 17592186044399ull,
            /* 42    */ (std::size_t) 35184372088777ull,
            /* 43    */ (std::size_t) 70368744177643ull,
            /* 44    */ (std::size_t) 140737488355213ull,
            /* 45    */ (std::size_t) 281474976710597ull,
            /* 46    */ (std::size_t) 562949953421231ull,
            /* 47    */ (std::size_t) 1125899906842597ull,
            /* 48    */ (std::size_t) 2251799813685119ull,
            /* 49    */ (std::size_t) 4503599627370449ull,
            /* 50    */ (std::size_t) 9007199254740881ull,
            /* 51    */ (std::size_t) 18014398509481951ull,
            /* 52    */ (std::size_t) 36028797018963913ull,
            /* 53    */ (std::size_t) 72057594037927931ull,
            /* 54    */ (std::size_t) 144115188075855859ull,
            /* 55    */ (std::size_t) 288230376151711717ull,
            /* 56    */ (std::size_t) 576460752303423433ull,
            /* 57    */ (std::size_t) 1152921504606846883ull,
            /* 58    */ (std::size_t) 2305843009213693951ull,
            /* 59    */ (std::size_t) 4611686018427387847ull,
            /* 60    */ (std::size_t) 9223372036854775783ull,
            /* 61    */ (std::size_t) 18446744073709551557ull,
    };

}
```

hashtable.hpp

```cpp
#include "hash_prime.hpp"

#include <exception>
#include <functional>
#include <vector>
#include <forward_list>

/**
 * The Hashtable class
 * The time complexity of functions are based on n and k
 * n is the size of the hashtable
 * k is the length of Key
 * @tparam Key          key type
 * @tparam Value        data type
 * @tparam Hash         function object, return the hash value of a key
 * @tparam KeyEqual     function object, return whether two keys are the same
 */
template<
        typename Key, typename Value,
        typename Hash = std::hash<Key>,
        typename KeyEqual = std::equal_to<Key>
>
class HashTable {
public:
    typedef std::pair<const Key, Value> HashNode;
    typedef std::forward_list<HashNode> HashNodeList;
    typedef std::vector<HashNodeList> HashTableData;

    /**
     * A single directional iterator for the hashtable
     * ! DO NOT NEED TO MODIFY THIS !
     */
    class Iterator {
    private:
        typedef typename HashTableData::iterator VectorIterator;
        typedef typename HashNodeList::iterator ListIterator;

        const HashTable *hashTable;
        VectorIterator bucketIt;    // an iterator of the buckets
        ListIterator listItBefore;  // a before iterator of the list, here we use "before" for quick erase and insert
        bool endFlag = false;       // whether it is an end iterator

        /**
         * Increment the iterator
         * Time complexity: Amortized O(1)
```

```
         */
        void increment() {
            if (bucketIt == hashTable->buckets.end()) {
                endFlag = true;
                return;
            }
            auto newListItBefore = listItBefore;
            ++newListItBefore;
            if (newListItBefore != bucketIt->end()) {
                if (++newListItBefore != bucketIt->end()) {
                    // use the next element in the current forward_list
                    ++listItBefore;
                    return;
                }
            }
            while (++bucketIt != hashTable->buckets.end()) {
                if (!bucketIt->empty()) {
                    // use the first element in a new forward_list
                    listItBefore = bucketIt->before_begin();
                    return;
                }
            }
            endFlag = true;
        }

        explicit Iterator(HashTable *hashTable) : hashTable(hashTable) {
            bucketIt = hashTable->buckets.begin();
            listItBefore = bucketIt->before_begin();
            endFlag = bucketIt == hashTable->buckets.end();
        }

        Iterator(HashTable *hashTable, VectorIterator vectorIt, ListIterator
         ↪  listItBefore) :
                hashTable(hashTable), bucketIt(vectorIt), listItBefore(listItBefore) {
            endFlag = bucketIt == hashTable->buckets.end();
        }

    public:
        friend class HashTable;

        Iterator() = delete;

        Iterator(const Iterator &) = default;

        Iterator &operator=(const Iterator &) = default;

        Iterator &operator++() {
```

```
            increment();
            return *this;
        }

        Iterator operator++(int) {
            Iterator temp = *this;
            increment();
            return temp;
        }

        bool operator==(const Iterator &that) const {
            if (endFlag && that.endFlag) return true;
            if (bucketIt != that.bucketIt) return false;
            return listItBefore == that.listItBefore;
        }

        bool operator!=(const Iterator &that) const {
            if (endFlag && that.endFlag) return false;
            if (bucketIt != that.bucketIt) return true;
            return listItBefore != that.listItBefore;
        }

        HashNode *operator->() {
            auto listIt = listItBefore;
            ++listIt;
            return &(*listIt);
        }

        HashNode &operator*() {
            auto listIt = listItBefore;
            ++listIt;
            return *listIt;
        }
    };

protected:                                                     // DO NOT USE private HERE!
    static constexpr double DEFAULT_LOAD_FACTOR = 0.5;         // default maximum load factor is 0.5
    static constexpr size_t DEFAULT_BUCKET_SIZE = HashPrime::g_a_sizes[0];  // default number of buckets is 5

    HashTableData buckets;                                     // buckets, of singly linked lists
    typename HashTableData::iterator firstBucketIt;            // help get begin iterator in O(1) time

```

```cpp
    size_t tableSize;                                   // number of elements
    double maxLoadFactor;                               // maximum load factor
    Hash hash;                                          // hash function instance
    KeyEqual keyEqual;                                  // key equal function instance

    /**
     * Time Complexity: O(k)
     * @param key
     * @param bucketSize
     * @return the hash value of key with a new bucket size
     */
    inline size_t hashKey(const Key &key, size_t bucketSize) const {
        return hash(key) % bucketSize;
    }

    /**
     * Time Complexity: O(k)
     * @param key
     * @return the hash value of key with current bucket size
     */
    inline size_t hashKey(const Key &key) const {
        return hash(key) % buckets.size();
    }

    /**
     * Find the minimum bucket size for the hashtable
     * The minimum bucket size must satisfy all of the following requirements:
     * - It is not less than (i.e. greater or equal to) the parameter bucketSize
     * - It is greater than floor(tableSize / maxLoadFactor)
     * - It is a (prime) number defined in HashPrime (hash_prime.hpp)
     * - It is minimum if satisfying all other requirements
     * Time Complexity: O(1)
     * @throw std::range_error if no such bucket size can be found
     * @param bucketSize lower bound of the new number of buckets
     */
    size_t findMinimumBucketSize(size_t bucketSize) const {
        // TODO: implement this function
    }

    // TODO: define your helper functions here if necessary


public:
```

```cpp
    HashTable() :
            buckets(DEFAULT_BUCKET_SIZE), tableSize(0), maxLoadFactor(DEFAULT_LOAD_FACTOR),
            hash(Hash()), keyEqual(KeyEqual()) {
        firstBucketIt = buckets.end();
    }

    explicit HashTable(size_t bucketSize) :
            tableSize(0), maxLoadFactor(DEFAULT_LOAD_FACTOR),
            hash(Hash()), keyEqual(KeyEqual()) {
        bucketSize = findMinimumBucketSize(bucketSize);
        buckets.resize(bucketSize);
        firstBucketIt = buckets.end();
    }

    HashTable(const HashTable &that) {
        // TODO: implement this function
    }

    HashTable &operator=(const HashTable &that) {
        // TODO: implement this function
    };

    ~HashTable() = default;

    Iterator begin() {
        if (firstBucketIt != buckets.end()) {
            return Iterator(this, firstBucketIt, firstBucketIt->before_begin());
        }
        return end();
    }

    Iterator end() {
        return Iterator(this, buckets.end(), buckets.begin()->before_begin());
    }

    /**
     * Find whether the key exists in the hashtable
     * Time Complexity: Amortized O(k)
     * @param key
     * @return whether the key exists in the hashtable
     */
    bool contains(const Key &key) {
        return find(key) != end();
    }

    /**
```

```
     * Find the value in hashtable by key
     * If the key exists, iterator points to the corresponding value, and it.endFlag = false
     * Otherwise, iterator points to the place that the key were to be inserted, and it.endFlag = true
     * Time Complexity: Amortized O(k)
     * @param key
     * @return the iterator of the value
     */
    Iterator find(const Key &key) {
        // TODO: implement this function
    }

    /**
     * Insert value into the hashtable according to an iterator returned by find
     * the function can be only be called if no other write actions are done to the hashtable after the find
     * If the key already exists, overwrite its value
     * firstBucketIt should be updated
     * If load factor exceeds maximum value, rehash the hashtable
     * Time Complexity: O(k)
     * @param it an iterator returned by find
     * @param key
     * @param value
     * @return whether insertion took place (return false if the key already exists)
     */
    bool insert(const Iterator &it, const Key &key, const Value &value) {
        // TODO: implement this function
    }

    /**
     * Insert <key, value> into the hashtable
     * If the key already exists, overwrite its value
     * firstBucketIt should be updated
     * If load factor exceeds maximum value, rehash the hashtable
     * Time Complexity: Amortized O(k)
     * @param key
     * @param value
     * @return whether insertion took place (return false if the key already exists)
     */
    bool insert(const Key &key, const Value &value) {
        // TODO: implement this function
    }

    /**
     * Erase the key if it exists in the hashtable, otherwise, do nothing
     * DO NOT rehash in this function
```

```cpp
     * firstBucketIt should be updated
     * Time Complexity: Amortized O(k)
     * @param key
     * @return whether the key exists
     */
    bool erase(const Key &key) {
        // TODO: implement this function
    }

    /**
     * Erase the key at the input iterator
     * If the input iterator is the end iterator, do nothing and return the input
    ↪ iterator directly
     * firstBucketIt should be updated
     * Time Complexity: O(1)
     * @param it
     * @return the iterator after the input iterator before the erase
     */
    Iterator erase(const Iterator &it) {
        // TODO: implement this function
    }

    /**
     * Get the reference of value by key in the hashtable
     * If the key doesn't exist, create it first (use default constructor of Value)
     * firstBucketIt should be updated
     * If load factor exceeds maximum value, rehash the hashtable
     * Time Complexity: Amortized O(k)
     * @param key
     * @return reference of value
     */
    Value &operator[](const Key &key) {
        // TODO: implement this function
    }

    /**
     * Rehash the hashtable according to the (hinted) number of buckets
     * The bucket size after rehash need not be same as the parameter bucketSize
     * Instead, findMinimumBucketSize is called to get the correct number
     * firstBucketIt should be updated
     * Do nothing if the bucketSize doesn't change
     * Time Complexity: O(nk)
     * @param bucketSize lower bound of the new number of buckets
     */
    void rehash(size_t bucketSize) {
        bucketSize = findMinimumBucketSize(bucketSize);
        if (bucketSize == buckets.size()) return;
```

```
        // TODO: implement this function
    }

    /**
     * @return the number of elements in the hashtable
     */
    size_t size() const { return tableSize; }

    /**
     * @return the number of buckets in the hashtable
     */
    size_t bucketSize() const { return buckets.size(); }

    /**
     * @return the current load factor of the hashtable
     */
    double loadFactor() const { return (double) tableSize / (double) buckets.size(); }

    /**
     * @return the maximum load factor of the hashtable
     */
    double getMaxLoadFactor() const { return maxLoadFactor; }

    /**
     * Set the max load factor
     * @throw std::range_error if the load factor is too small
     * @param loadFactor
     */
    void setMaxLoadFactor(double loadFactor) {
        if (loadFactor <= 1e-9) {
            throw std::range_error("invalid load factor!");
        }
        maxLoadFactor = loadFactor;
        rehash(buckets.size());
    }

};
```