VE281 — Data Structures and Algorithms

Programming Assignment 2

Instructor: Hongyi Xin
— UM-SJTU-JI (Fall 2020)

Notes

- Due Date: 11/13/2020
- 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:

```
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 the "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 ask 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`.

Write about **one page** on the report about your findings, plots and explanations.

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.

2.3 Study of Combining Hash Functions

In this part, you are going to do some literature study on how to combine two hash functions. More specifically, if the type of the key of the hash table is `std::pair<T1, T2>` and the hash functions are already defined for `T1` and `T2`, how to combine the results of these two hash functions and write a new function `hash<std::pair<T1, T2>>`.

The definition and a trivial implementation (which is not good) of the combined hash function (object) is provided:

```
struct PairHash {
    template<typename T1, typename T2>
    size_t operator()(const pair<T1, T2> &p) const {
        // TODO: this implementation is not good, improve it!
        return std::hash<T1>()(p.first) ^ std::hash<T2>()(p.second);
    }
};
```

Improve the implementation of the `PairHash` above in your report. Make sure it can compile and work with `HashTable` and `std::unordered_map`. You should also give a brief explanation of **one paragraph** about why you choose this implementation. You can refer to any source (i.e., paper, open-source code), but you need to reference it in your report.

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, you will lose the performance score for those test cases.

4.5 Report on the performance study

Finally, we will also read your report and grade it based on the quality of your performance study.

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

```cpp
// 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_64_bit] = {
            /* 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++() {
```

```
92              increment();
93              return *this;
94          }
95
96          Iterator operator++(int) {
97              Iterator temp = *this;
98              increment();
99              return temp;
100         }
101
102         bool operator==(const Iterator &that) const {
103             if (endFlag && that.endFlag) return true;
104             if (bucketIt != that.bucketIt) return false;
105             return listItBefore == that.listItBefore;
106         }
107
108         bool operator!=(const Iterator &that) const {
109             if (endFlag && that.endFlag) return false;
110             if (bucketIt != that.bucketIt) return true;
111             return listItBefore != that.listItBefore;
112         }
113
114         HashNode *operator->() {
115             auto listIt = listItBefore;
116             ++listIt;
117             return &(*listIt);
118         }
119
120         HashNode &operator*() {
121             auto listIt = listItBefore;
122             ++listIt;
123             return *listIt;
124         }
125     };
126
127 protected:                                                  // DO NOT USE
    ↪ private HERE!
128     static constexpr double DEFAULT_LOAD_FACTOR = 0.5;      // default
      ↪ maximum load factor is 0.5
129     static constexpr size_t DEFAULT_BUCKET_SIZE = HashPrime::g_a_sizes[0];  // default
      ↪ number of buckets is 5
130
131     HashTableData buckets;                                  // buckets,
      ↪ of singly linked lists
132     typename HashTableData::iterator firstBucketIt;         // help get
      ↪ begin iterator in O(1) time
133
```

```
134     size_t tableSize;                                                   // number of
        ↪ elements
135     double maxLoadFactor;                                               // maximum
        ↪ load factor
136     Hash hash;                                                          // hash
        ↪ function instance
137     KeyEqual keyEqual;                                                  // key equal
        ↪ function instance
138 
139     /**
140      * Time Complexity: O(k)
141      * @param key
142      * @param bucketSize
143      * @return the hash value of key with a new bucket size
144      */
145     inline size_t hashKey(const Key &key, size_t bucketSize) const {
146         return hash(key) % bucketSize;
147     }
148 
149     /**
150      * Time Complexity: O(k)
151      * @param key
152      * @return the hash value of key with current bucket size
153      */
154     inline size_t hashKey(const Key &key) const {
155         return hash(key) % buckets.size();
156     }
157 
158     /**
159      * Find the minimum bucket size for the hashtable
160      * The minimum bucket size must satisfy all of the following requirements:
161      * - It is not less than (i.e. greater or equal to) the parameter bucketSize
162      * - It is greater than floor(tableSize / maxLoadFactor)
163      * - It is a (prime) number defined in HashPrime (hash_prime.hpp)
164      * - It is minimum if satisfying all other requirements
165      * Time Complexity: O(1)
166      * @throw std::range_error if no such bucket size can be found
167      * @param bucketSize lower bound of the new number of buckets
168      */
169     size_t findMinimumBucketSize(size_t bucketSize) const {
170         // TODO: implement this function
171     }
172 
173     // TODO: define your helper functions here if necessary
174 
175 
176 public:
```

```
177     HashTable() :
178             buckets(DEFAULT_BUCKET_SIZE), tableSize(0),
                ↪  maxLoadFactor(DEFAULT_LOAD_FACTOR),
179             hash(Hash()), keyEqual(KeyEqual()) {
180         firstBucketIt = buckets.end();
181     }
182 
183     explicit HashTable(size_t bucketSize) :
184             tableSize(0), maxLoadFactor(DEFAULT_LOAD_FACTOR),
185             hash(Hash()), keyEqual(KeyEqual()) {
186         bucketSize = findMinimumBucketSize(bucketSize);
187         buckets.resize(bucketSize);
188         firstBucketIt = buckets.end();
189     }
190 
191     HashTable(const HashTable &that) {
192         // TODO: implement this function
193     }
194 
195     HashTable &operator=(const HashTable &that) {
196         // TODO: implement this function
197     };
198 
199     ~HashTable() = default;
200 
201     Iterator begin() {
202         if (firstBucketIt != buckets.end()) {
203             return Iterator(this, firstBucketIt, firstBucketIt->before_begin());
204         }
205         return end();
206     }
207 
208     Iterator end() {
209         return Iterator(this, buckets.end(), buckets.begin()->before_begin());
210     }
211 
212     /**
213      * Find whether the key exists in the hashtable
214      * Time Complexity: Amortized O(k)
215      * @param key
216      * @return whether the key exists in the hashtable
217      */
218     bool contains(const Key &key) {
219         return find(key) != end();
220     }
221 
222     /**
```

```
223         * Find the value in hashtable by key
224         * If the key exists, iterator points to the corresponding value, and it.endFlag =
    ↪ false
225         * Otherwise, iterator points to the place that the key were to be inserted, and
    ↪ it.endFlag = true
226         * Time Complexity: Amortized O(k)
227         * @param key
228         * @return a pair (success, iterator of the value)
229         */
230        Iterator find(const Key &key) {
231            // TODO: implement this function
232        }
233
234        /**
235         * Insert value into the hashtable according to an iterator returned by find
236         * the function can be only be called if no other write actions are done to the
    ↪ hashtable after the find
237         * If the key already exists, overwrite its value
238         * firstBucketIt should be updated
239         * If load factor exceeds maximum value, rehash the hashtable
240         * Time Complexity: O(k)
241         * @param it an iterator returned by find
242         * @param key
243         * @param value
244         * @return whether insertion took place (return false if the key already exists)
245         */
246        bool insert(const Iterator &it, const Key &key, const Value &value) {
247            // TODO: implement this function
248        }
249
250        /**
251         * Insert <key, value> into the hashtable
252         * If the key already exists, overwrite its value
253         * firstBucketIt should be updated
254         * If load factor exceeds maximum value, rehash the hashtable
255         * Time Complexity: Amortized O(k)
256         * @param key
257         * @param value
258         * @return whether insertion took place (return false if the key already exists)
259         */
260        bool insert(const Key &key, const Value &value) {
261            // TODO: implement this function
262        }
263
264        /**
265         * Erase the key if it exists in the hashtable, otherwise, do nothing
266         * DO NOT rehash in this function
```

```
     * 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;
```

```
313             // TODO: implement this function
314         }
315
316         /**
317          * @return the number of elements in the hashtable
318          */
319         size_t size() const { return tableSize; }
320
321         /**
322          * @return the number of buckets in the hashtable
323          */
324         size_t bucketSize() const { return buckets.size(); }
325
326         /**
327          * @return the current load factor of the hashtable
328          */
329         double loadFactor() const { return (double) tableSize / (double) buckets.size(); }
330
331         /**
332          * @return the maximum load factor of the hashtable
333          */
334         double getMaxLoadFactor() const { return maxLoadFactor; }
335
336         /**
337          * Set the max load factor
338          * @throw std::range_error if the load factor is too small
339          * @param loadFactor
340          */
341         void setMaxLoadFactor(double loadFactor) {
342             if (loadFactor <= 1e-9) {
343                 throw std::range_error("invalid load factor!");
344             }
345             maxLoadFactor = loadFactor;
346             rehash(buckets.size());
347         }
348
349     };
```