VE281 — Data Structures and Algorithms

Programming Assignment 2

Instructor: Yutong Ban
— UM-SJTU-JI (Summer 2024)

Notes

- Due Date: 5/7/2024
- Submission: on JOJ (https://joj.sjtu.edu.cn/d/ece281_24su/)

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".

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.

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