Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем

Расчетно-графическая работа
по дисциплине «Современные технологии программирования»
на тему «Реализация шаблонного типа данных HashMap»

Выполнил:
ст. гр. ИС-042 Рябов К. А.

Проверил:
ст. пр. Пименов Е. С.

Новосибирск 2023

Содержание

1. Постановка задачи

Спроектировать шаблонный тип данных HashMap. Использовать стандарт языка C++20. Реализовать итераторы, совместимые с алгоритмами стандартной библиотеки. Покрыть модульными тестами. При реализации не пользоваться контейнерами стандартной библиотеки, реализовать управление ресурсами в идиоме RAII.

В качестве системы сборки использовать CMake. Структурировать проект в соответствии с соглашениями The Pitchfork Layout (Merged Header, Separate Test). Всю разработку вести в системе контроля версий git. Настроить автоматическое форматирование средствами clang-format.

Проверить код анализаторами Valgrind и clang-tidy.

2. Мотивация

HashMap (хеш-таблица) представляет собой структуру данных для хранения пар вида «ключ» — «значение». Доступ к элементам контейнера предоставляется по ключу. В отличии от структур данных вроде массива, ключи могут быть различными типами данных (строки, числа, указатели и др.). Данная особенность достигается при помощи хеш-функции, которая преобразует ключ в целочисленное значение индекса.

Основной мотивацией применения хеш-таблицы служит константная сложность доступа к элементам контейнера. Другими словами, в лучшем случае для базовых операций вроде добавления, поиска и удаления элементов из структуры данных асимптотическая сложность будет равна $O(1)$. Тем не менее, для некоторых задач выбор HashMap может быть не самым лучшем решением из-за того, что этот контейнер является неупорядоченным.

3. Реализация

3.1 Внутреннее устройство

Структура данных HashMap имеет достаточно известную проблему, связанную с возможным возникновением так называемых коллизий. Широкое распространение получили два основных способа разрешения коллизий: метод цепочек и открытая адресация. В данной реализации хеш-таблицы использовался первый способ.

Метод цепочек заключается в том, что каждая ячейка (bucket) в HashMap содержит указатель на голову связного списка, в который помещаются все ключи имеющие одинаковый хеш-код. Соответственно, в каждом узле хранится ключ, значение и указатель на следующий узел.

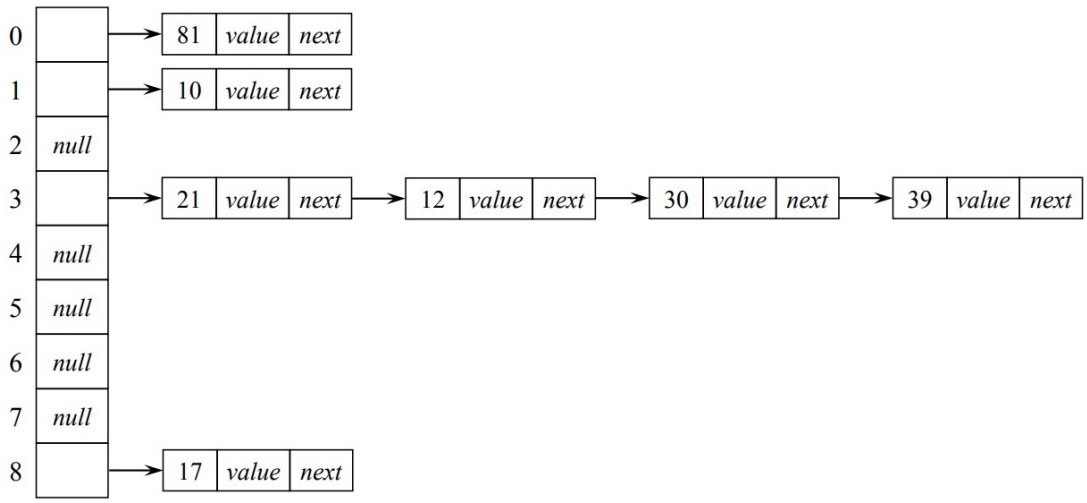


Рисунок 3.1: Структура хеш-таблицы с использованием метода цепочек

3.2 Интерфейс

Как и любой другой ассоциативный массив HashMap имеет базовый набор операций, состоящий из добавления, поиска и удаления элементов. Рассмотрим каждую из этих операций отдельно.

1. Вставка (insert). Перед добавлением необходимо проверить, существует ли уже в контейнере элемент с указанным ключом. Если существует, то возвращаем итератор на этот элемент, иначе вставляем новый элемент.

   Для добавления элемента в хеш-таблицу сначала вычисляется хеш-код для заданного ключа, чтобы получить доступ к ячейке для вставки. Затем создается новый узел и вставляется в начало списка, на который указывает полученная по хеш-коду ячейка. После успешной вставки возвращается итератор на добавленный элемент.

   Вычислительная сложность данной операции могла бы быть $O(1)$, поскольку время вычисления хеш-функции не зависит от количества ключей в контейнере, а добавление в начало связного списка требует порядка $O(1)$ операций. Но поскольку

перед добавлением, нам необходимо убедиться в том, что элемента с указанным ключом еще нет в хеш-таблице (это в свою очередь требует в худшем случае просмотра одного связного списка), то вычислительная сложность операции вставки в худшем случае будет достигать $O(m)$, где m — длина связного списка, в который добавляется новый узел.

```
129     ForwardIterator insert(const Key &key, const T &value) {
130         auto iter = find(key);
131         if (iter != end()) {
132             return iter;
133         }
134         const auto hash_value = calculate_hash_(key) % bucket_count_;
135         auto *node = new Node(key, value);
136         node->next_ = table_[hash_value];
137         table_[hash_value] = node;
138         return ForwardIterator(node, table_, hash_value, bucket_count_);
139     }
```

Рисунок 3.2: Реализация операции вставки элемента в хеш-таблицу

2. Поиск (find). Для заданного ключа вычисляется его хеш-код, чтобы получить доступ к ячейке со связным списком, в котором осуществляется поиск узла по ключу. Если элемент найден, то возвращается итератор на этот элемент. В ином же случае возвращается итератор на конец HashMap.

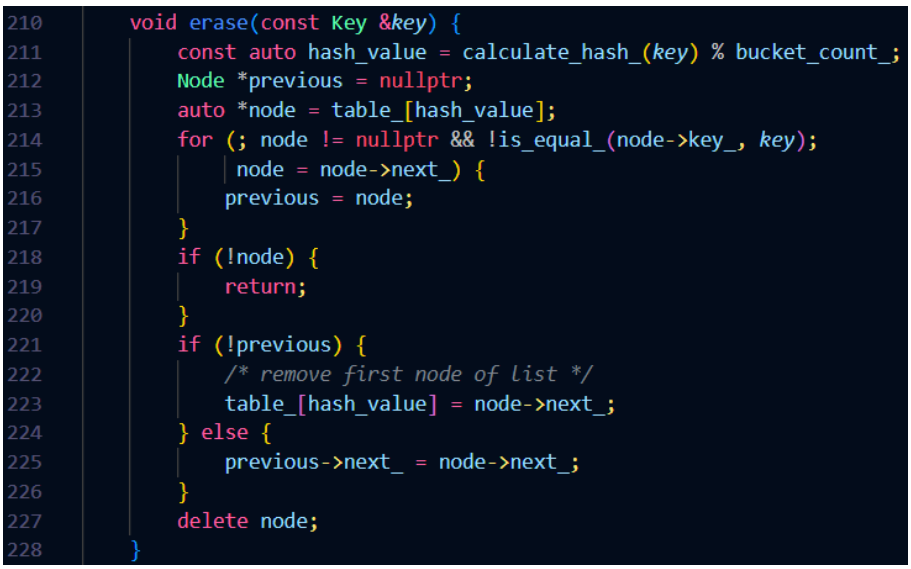
   Вычислительная сложность поиска, как уже было сказано ранее, в худшем случае требует полного прохождения по соответствующему связному списку, то есть равна $O(m)$.

```
166     ForwardIterator find(const Key &key) const {
167         const auto hash_value = calculate_hash_(key) % bucket_count_;
168         for (auto *node = table_[hash_value]; node; node = node->next_) {
169             if (is_equal_(node->key_, key)) {
170                 return ForwardIterator(node, table_, hash_value, bucket_count_);
171             }
172         }
173         return end();
174     }
```

Рисунок 3.3: Реализация операции поиска элемента в хеш-таблице

3. Удаление (erase). Для удаления элемента из HashMap необходимо все также вычислить хеш-код. Затем в соответствующем связном списке происходит поиск узла для удаления. При этом необходимо запомнить элемент, который находится перед удаляемым. Это нужно для того, чтобы сохранить целостность списка, соединив указатели узлов, находящихся между удаленным.

   Вычислительная сложность в худшем случае требует перебора соответствующего связного списка и равна $O(m)$.



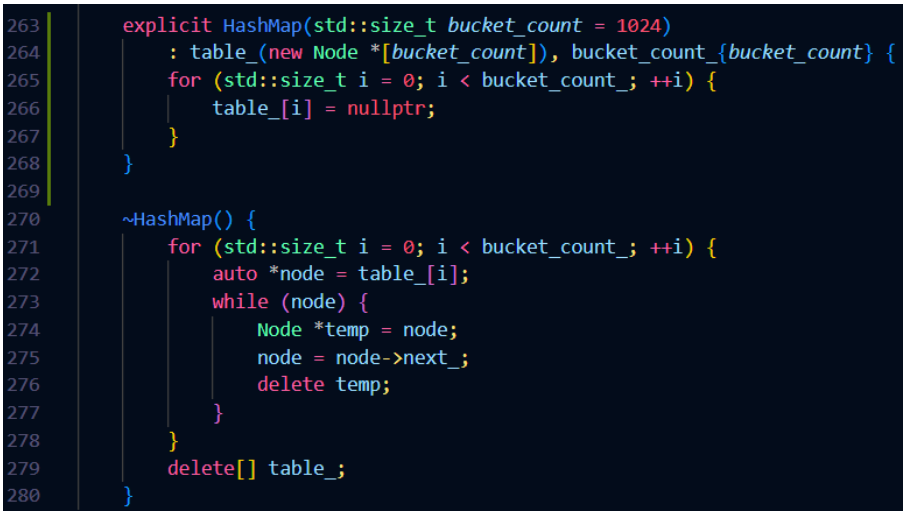
```
210     void erase(const Key &key) {
211         const auto hash_value = calculate_hash_(key) % bucket_count_;
212         Node *previous = nullptr;
213         auto *node = table_[hash_value];
214         for (; node != nullptr && !is_equal_(node->key_, key);
215              node = node->next_) {
216             previous = node;
217         }
218         if (!node) {
219             return;
220         }
221         if (!previous) {
222             /* remove first node of list */
223             table_[hash_value] = node->next_;
224         } else {
225             previous->next_ = node->next_;
226         }
227         delete node;
228     }
```

Рисунок 3.4: Реализация операции удаления элемента из хеш-таблицы

Хеш-таблица требует предварительную инициализацию. При создании объекта класса HashMap можно указать количество ячеек, из которых будет состоять контейнер (по умолчанию значение 1024). Выделяется память под указанное число ячеек, а затем в каждую из них записывается nullptr. Вычислительная сложность $O(n)$, где n — количество ячеек в структуре данных.

Поскольку выделяется память, ее необходимо по завершению работы с HashMap освободить. Для этого написан деструктор, который проходится по всем элементам хеш-таблицы в цикле и освобождает память. Вычислительная сложность $O(n \cdot m)$.

Также из-за того, что был реализован деструктор, в соответствии с Rule of Five необходимо реализовать (или удалить) конструкторы копирования и перемещения, операторы присваивания копированием и перемещением.



```
263     explicit HashMap(std::size_t bucket_count = 1024)
264         : table_(new Node *[bucket_count]), bucket_count_{bucket_count} {
265         for (std::size_t i = 0; i < bucket_count_; ++i) {
266             table_[i] = nullptr;
267         }
268     }
269
270     ~HashMap() {
271         for (std::size_t i = 0; i < bucket_count_; ++i) {
272             auto *node = table_[i];
273             while (node) {
274                 Node *temp = node;
275                 node = node->next_;
276                 delete temp;
277             }
278         }
279         delete[] table_;
280     }
```

Рисунок 3.5: Реализация конструктора и деструктора для хеш-таблицы

Помимо вышеуказанных операций для удобства использования с алгоритмами стандартной библиотеки в HashMap присутствуют методы begin и end. Первый возвращает итератор на начало хеш-таблицы (на первый встречный элемент отличный от nullptr). Второй

метод же возвращает итератор на конец контейнера, а именно на самый последний nullptr. Вычислительная сложность в худшем случае для begin — $O(n)$, а для метода end — $O(m)$.

```
141     ForwardIterator end() const {
142         if (bucket_count_ == 0) {
143             return ForwardIterator();
144         }
145         const auto index = bucket_count_ - 1;
146         auto iter =
147             ForwardIterator(table_[index], table_, index, bucket_count_);
148         if (!table_[index]) {
149             return iter;
150         }
151         while (iter->next_) {
152             ++iter;
153         }
154         return ++iter;
155     }
156
157     ForwardIterator begin() const {
158         for (std::size_t i = 0; i < bucket_count_; ++i) {
159             if (table_[i]) {
160                 return ForwardIterator(table_[i], table_, i, bucket_count_);
161             }
162         }
163         return end();
164     }
```

Рисунок 3.6: Реализация методов begin и end для хеш-таблицы

Бывает такое, что в некоторых задачах требуется узнать, сколько элементов содержит та или иная коллекция. В случае с HashMap может потребоваться узнать количество ячеек или длину связного списка в какой-либо ячейке. Для таких нужд в интерфейсе структуры данных присутствуют методы size, bucket_count и bucket_size. Первый возвращает количество элементов в хеш-таблице. Второй — количество ячеек. Третий — размер связного списка для указанной ячейки. Вычислительные сложности, соответственно, равны в худшем случае $O(n \cdot m)$, $O(1)$ и $O(m)$.

```
176     std::size_t bucket_size(std::size_t bucket_index) const {
177         std::size_t size = 0;
178         if (bucket_index >= bucket_count_) {
179             return size;
180         }
181         for (auto *node = table_[bucket_index]; node; node = node->next_) {
182             ++size;
183         }
184         return size;
185     }
186
187     std::size_t bucket_count() const { return bucket_count_; }
188
189     std::size_t size() const { return std::distance(begin(), end()); }
```

Рисунок 3.7: Реализация методов size, bucket_count и bucket_size для хеш-таблицы

При работе с HashMap достаточно удобно обращаться к элементам через оператор квадратные скобки, поэтому в реализованном интерфейсе присутствует перегрузка данного оператора. Реализация следующая: вызываем метод find, если не находим элемент по указанному ключу, то вставляем новый с этим ключом и возвращаем ссылку на значение. Вычислительная сложность равна $O(m)$ в худшем случае.

Также удобно использовать оператор квадратные скобки в паре с методом at, который сигнализирует о том, найден ли элемент по указанному ключу (в ином случае выбрасывается исключение std::out_of_range), поскольку неаккуратное применение оператора квадратные скобки может привести к добавлению нежелательных узлов в хеш-таблицу. Вычислительная сложность метода at в худшем случае равна $O(m)$.

```
191     T &at(const Key &key) {
192         auto iter = find(key);
193         if (iter == end()) {
194             throw std::out_of_range("oops.. key not found =(");
195         }
196         return iter->value_;
197     }
198 
199     T &operator[](const Key &key) {
200         auto iter = find(key);
201         if (iter == end()) {
202             T value;
203             iter = insert(key, value);
204         }
205         return iter->value_;
206     }
207 
208     T &operator[](Key &&key) { return operator[](key); }
```

Рисунок 3.8: Реализация метода at и перегрузок оператора квадратные скобки

3.3 Итераторы

Структура данных HashMap предоставляет итератор категории LegacyForwardIterator, который можно инкрементировать и применять для обхода несколько раз.

Итератор имеет следующие поля: указатель на текущий узел, ячейки с указателями на связные списки, индекс ячейки с текущим узлом и количество ячеек в хеш-таблице. Такое число полей обусловлено реализацией префиксного и постфиксного инкрементов. Рассмотрим реализацию только первого, поскольку второй легко выражается из него.

Префиксный инкремент реализован так, что если текущий узел не nullptr и он не последний в связном списке, то просто устанавливаем указатель на следующий элемент. Иначе инкрементируем индекс ячейки с текущим узлом и в цикле по оставшимся ячейкам ищем ячейку отличную от nullptr, после чего устанавливаем указатель на нее (то есть на первый элемент в связном списке). Если все оставшиеся ячейки равны nullptr, то устанавливаем указатель тоже в nullptr.

Помимо инкрементов интерфейс итератора реализует перегрузку операторов star и arrow для предоставления доступа к методам класса узла в HashMap (например, метод value, который возвращает значение текущего узла), а также присутствует перегрузка операторов равно и неравно, которые сравнивают указатели на текущие элементы двух итераторов.

```
77          ForwardIterator &operator++() {
78              if (ptr_ && ptr_->next_) {
79                  ptr_ = ptr_->next_;
80                  return *this;
81              }
82              ptr_ = nullptr;
83              for (++index_; index_ < bucket_count_; ++index_) {
84                  if (table_[index_]) {
85                      ptr_ = table_[index_];
86                      break;
87                  }
88              }
89              return *this;
90          }
91
92          ForwardIterator operator++(int) {
93              auto old = *this;
94              ++(*this);
95              return old;
96          }
```

Рисунок 3.9: Перегрузка операторов префиксного и постфиксного инкрементов

4. Список источников

1. Аксенов А. Что мы знаем про хэши [Электронный ресурс]. URL: https://backendconf.ru/2018/abstracts/3445 (дата обращения: 06.04.2023).

2. Алгоритмы и структуры обработки информации. / М. Г. Курносов, Д. М. Берлизов – Новосибирск: Параллель, 2019. – 211 с.

3. Алгоритмы: построение и анализ. 3-е изд. / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, К. Штайн. — М. : Вильямс, 2013

5. Приложение

```
// file libcsc/libcsc/hashmap/hashmap.hpp
#pragma once

#include <functional>
#include <memory>

namespace csc::hashmap {

template <
    typename Key,
    typename T,
    typename Hash = std::hash<Key>,
    typename KeyEqual = std::equal_to<Key>>
class HashMap {
private:
    class ForwardIterator;

    /* HashMap's node implementation */
    class Node {
    private:
        friend HashMap;
        friend ForwardIterator;
        Key key_;
        T value_;
        Node *next_;

    public:
        Node(const Key &key, const T &value)
            : key_{key}, value_{value}, next_{nullptr} {}

        Key key() const { return key_; }

        T value() const { return value_; }

        bool operator==(const Node &other) const {
            if (this == &other) {
                return true;
            }
            return (key_ == other.key_) && (value_ == other.value_);
        }
    };

    /* ForwardIterator implementation */
    class ForwardIterator {
    public:
        using value_type = T;
        using reference = Node &;
        using pointer = Node *;
        using iterator_category = std::forward_iterator_tag;
        using difference_type = std::ptrdiff_t;

    private:
        pointer ptr_;
        pointer *table_;
        std::size_t index_;
        std::size_t bucket_count_;

    public:
```

```
    ForwardIterator()
        : ptr_{nullptr}, table_{nullptr}, index_{0}, bucket_count_{0} {}

    ForwardIterator(
        pointer ptr,
        pointer *table,
        std::size_t index,
        std::size_t bucket_count)
        : ptr_{ptr}, table_{table}, index_{index}, bucket_count_{
                                                        bucket_count} {}

    reference operator*() { return *ptr_; }

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

    pointer operator->() { return ptr_; }

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

    ForwardIterator &operator++() {
        if (ptr_ && ptr_->next_) {
            ptr_ = ptr_->next_;
            return *this;
        }
        ptr_ = nullptr;
        for (++index_; index_ < bucket_count_; ++index_) {
            if (table_[index_]) {
                ptr_ = table_[index_];
                break;
            }
        }
        return *this;
    }

    ForwardIterator operator++(int) {
        auto old = *this;
        ++(*this);
        return old;
    }

    bool operator==(const ForwardIterator &other) const {
        if (this == &other) {
            return true;
        }
        return ptr_ == other.ptr_;
    }

    bool operator!=(const ForwardIterator &other) const {
        return !(*this == other);
    }
};

/* HashMap's fields */
Node **table_;
std::size_t bucket_count_;
Hash calculate_hash_;
KeyEqual is_equal_;

void swap(HashMap &other) noexcept {
    std::swap(bucket_count_, other.bucket_count_);
    std::swap(table_, other.table_);
```

```
    }

public:
    explicit HashMap(std::size_t bucket_count = 1024)
        : table_(new Node *[bucket_count]), bucket_count_{bucket_count} {
        for (std::size_t i = 0; i < bucket_count_; ++i) {
            table_[i] = nullptr;
        }
    }

    ForwardIterator insert(const Key &key, const T &value) {
        auto iter = find(key);
        if (iter != end()) {
            return iter;
        }
        const auto hash_value = calculate_hash_(key) % bucket_count_;
        auto *node = new Node(key, value);
        node->next_ = table_[hash_value];
        table_[hash_value] = node;
        return ForwardIterator(node, table_, hash_value, bucket_count_);
    }

    ForwardIterator end() const {
        if (bucket_count_ == 0) {
            return ForwardIterator();
        }
        const auto index = bucket_count_ - 1;
        auto iter =
            ForwardIterator(table_[index], table_, index, bucket_count_);
        if (!table_[index]) {
            return iter;
        }
        while (iter->next_) {
            ++iter;
        }
        return ++iter;
    }

    ForwardIterator begin() const {
        for (std::size_t i = 0; i < bucket_count_; ++i) {
            if (table_[i]) {
                return ForwardIterator(table_[i], table_, i, bucket_count_);
            }
        }
        return end();
    }

    ForwardIterator find(const Key &key) const {
        const auto hash_value = calculate_hash_(key) % bucket_count_;
        for (auto *node = table_[hash_value]; node; node = node->next_) {
            if (is_equal_(node->key_, key)) {
                return ForwardIterator(node, table_, hash_value, bucket_count_);
            }
        }
        return end();
    }

    std::size_t bucket_size(std::size_t bucket_index) const {
        std::size_t size = 0;
        if (bucket_index >= bucket_count_) {
            return size;
```

```
    }
    for (auto *node = table_[bucket_index]; node; node = node->next_) {
        ++size;
    }
    return size;
}

std::size_t bucket_count() const { return bucket_count_; }

std::size_t size() const { return std::distance(begin(), end()); }

T &at(const Key &key) {
    auto iter = find(key);
    if (iter == end()) {
        throw std::out_of_range("oops.. key not found =(");
    }
    return iter->value_;
}

T &operator[](const Key &key) {
    auto iter = find(key);
    if (iter == end()) {
        T value;
        iter = insert(key, value);
    }
    return iter->value_;
}

T &operator[](Key &&key) { return operator[](key); }

void erase(const Key &key) {
    const auto hash_value = calculate_hash_(key) % bucket_count_;
    Node *previous = nullptr;
    auto *node = table_[hash_value];
    for (; node != nullptr && !is_equal_(node->key_, key);
         node = node->next_) {
        previous = node;
    }
    if (!node) {
        return;
    }
    if (!previous) {
        /* remove first node of list */
        table_[hash_value] = node->next_;
    } else {
        previous->next_ = node->next_;
    }
    delete node;
}

void erase(const ForwardIterator &iterator) { erase(iterator->key_); }

/* Rule of five */
HashMap(const HashMap<Key, T, Hash, KeyEqual> &other)
    : HashMap(other.bucket_count()) {
    for (auto iter = other.begin(); iter != other.end(); ++iter) {
        insert(iter->key_, iter->value_);
    }
}

HashMap(HashMap<Key, T, Hash, KeyEqual> &&other) noexcept
```

```
        : table_{other.table_}, bucket_count_{other.bucket_count_} {
        other.table_ = nullptr;
        other.bucket_count_ = 0;
    }

    HashMap &operator=(const HashMap &rhs) {
        if (this != &rhs) {
            HashMap copy(rhs);
            copy.swap(*this);
        }
        return *this;
    }

    HashMap &operator=(HashMap &&rhs) noexcept {
        if (this != &rhs) {
            table_ = nullptr;
            bucket_count_ = 0;
            rhs.swap(*this);
        }
        return *this;
    }

    ~HashMap() {
        for (std::size_t i = 0; i < bucket_count_; ++i) {
            auto *node = table_[i];
            while (node) {
                Node *temp = node;
                node = node->next_;
                delete temp;
            }
        }
        delete[] table_;
    }
};

} // namespace csc::hashmap

// file libcsc/libcsc/absolutepath.hpp.in
#pragma once

#include <string>

const std::string_view c_absolute_path = "${PROJECT_SOURCE_DIR}";

// file libcsc/libcsc/hashmap.cpp
#include <libcsc/absolutepath.hpp>
#include <libcsc/hashmap/hashmap.hpp>

#include <gtest/gtest.h>

#include <algorithm>
#include <filesystem>
#include <fstream>
#include <unordered_map>

TEST(HashMap, HashMapTest) {
    const std::filesystem::path path(c_absolute_path);
    std::ifstream file(path / "words-5m.txt");
    std::istream_iterator<std::string> start(file), end;
    std::vector<std::string> words(start, end);
    csc::hashmap::HashMap<std::string, std::size_t> hashmap;
```

```
    std::unordered_map<std::string, std::size_t> exp;
    for (const auto &word : words) {
        ++hashmap[word];
        ++exp[word];
    }
    EXPECT_EQ(hashmap.size(), exp.size());
    for (auto &node : hashmap) {
        const auto iter = exp.find(node.key());
        if (iter == exp.end()) {
            FAIL();
        }
        EXPECT_EQ(node.key(), iter->first);
        EXPECT_EQ(node.value(), iter->second);
    }
}

TEST(HashMap, InsertElements) {
    csc::hashmap::HashMap<std::string, std::size_t> hashmap;
    EXPECT_EQ(hashmap.begin(), hashmap.end());
    hashmap["brackets"] = 88;
    EXPECT_EQ(hashmap.find("brackets")->value(), 88);
    const auto val = hashmap["consttttttt"];
    EXPECT_EQ(hashmap.find("consttttttt")->value(), val);
    hashmap.insert("brackets", 0);
    EXPECT_EQ(hashmap.find("brackets")->value(), 88);
}

TEST(HashMap, GetBucketSize) {
    csc::hashmap::HashMap<std::string, std::size_t> hashmap(1);
    EXPECT_EQ(hashmap.begin(), hashmap.end());
    hashmap["bucket"] = 8;
    hashmap["buckets"] = 88;
    hashmap["aaaaaaaaaaaaaaaaaaa"] = 91392;
    EXPECT_EQ(hashmap.bucket_count(), 1);
    EXPECT_EQ(hashmap.bucket_size(0), 3);
    hashmap.insert("aaaaaaaaaaaaaaaaaaa", 0);
    EXPECT_EQ(hashmap.bucket_size(0), 3);
}

TEST(HashMap, EraseElements) {
    csc::hashmap::HashMap<std::string, std::size_t> hashmap;
    EXPECT_EQ(hashmap.begin(), hashmap.end());
    EXPECT_EQ(hashmap.begin() == hashmap.begin(), true);
    EXPECT_EQ(hashmap.begin() == hashmap.end(), true);
    EXPECT_EQ(hashmap.size(), 0);
    hashmap["remove"] = 132213;
    EXPECT_EQ(hashmap.begin() == hashmap.begin(), true);
    EXPECT_EQ(hashmap.begin() == hashmap.end(), false);
    hashmap["delete"] = 93193121;
    hashmap["erase"] = 13213;
    hashmap["clear"] = 939123;
    hashmap["etc."] = 8192329321;
    EXPECT_EQ(hashmap.size(), 5);
    hashmap.erase("etc.");
    EXPECT_EQ(hashmap.size(), 4);
    /* trying to erase non-existent element */
    hashmap.erase("hahahahahahahaha");
    EXPECT_EQ(hashmap.size(), 4);
    hashmap.erase(hashmap.begin());
    EXPECT_EQ(hashmap.size(), 3);
    std::vector<std::string> keys;
```

```
    for (auto &node : hashmap) {
        keys.push_back(node.key());
    }
    EXPECT_EQ(keys.size(), 3);
    for (const auto &key : keys) {
        hashmap.erase(key);
    }
    EXPECT_EQ(hashmap.size(), 0);
}

TEST(HashMap, RuleOfFiveTest) {
    csc::hashmap::HashMap<std::string, std::size_t> hashmap;
    EXPECT_EQ(hashmap.begin(), hashmap.end());
    EXPECT_EQ(hashmap.size(), 0);
    hashmap["ummmmm"] = 132213;
    hashmap["hahahaahaha"] = 93193121;
    hashmap["mmmmmmmmmmmm"] = 13213;
    hashmap["testtesttest"] = 939123;
    hashmap["yosh!!"] = 8192329321;
    EXPECT_EQ(hashmap.size(), 5);
    auto hashmap2(hashmap);
    EXPECT_EQ(hashmap.size(), hashmap2.size());
    EXPECT_EQ(hashmap.bucket_count(), hashmap2.bucket_count());
    for (auto &node : hashmap2) {
        auto iter = hashmap.find(node.key());
        if (iter == hashmap.end()) {
            FAIL();
        }
        EXPECT_EQ(node.key(), iter->key());
        EXPECT_EQ(node.value(), iter->value());
    }
    auto hashmap3(std::move(hashmap2));
    EXPECT_EQ(hashmap2.size(), 0);
    EXPECT_EQ(hashmap3.size(), 5);
    for (auto &node : hashmap3) {
        auto iter = hashmap.find(node.key());
        if (iter == hashmap.end()) {
            FAIL();
        }
        EXPECT_EQ(node.key(), iter->key());
        EXPECT_EQ(node.value(), iter->value());
    }
    auto hashmap4 = hashmap;
    EXPECT_EQ(hashmap4.size(), 5);
    for (auto &node : hashmap4) {
        auto iter = hashmap.find(node.key());
        if (iter == hashmap.end()) {
            FAIL();
        }
        EXPECT_EQ(node.key(), iter->key());
        EXPECT_EQ(node.value(), iter->value());
    }
    hashmap = hashmap;
    EXPECT_EQ(hashmap.size(), 5);
    hashmap2 = std::move(hashmap4);
    EXPECT_EQ(hashmap4.size(), 0);
    EXPECT_EQ(hashmap.size(), hashmap2.size());
    EXPECT_EQ(hashmap.bucket_count(), hashmap2.bucket_count());
    for (auto &node : hashmap2) {
        auto iter = hashmap.find(node.key());
        if (iter == hashmap.end()) {
```

```
            FAIL();
        }
        EXPECT_EQ(node.key(), iter->key());
        EXPECT_EQ(node.value(), iter->value());
    }
    hashmap2 = std::move(hashmap2);
}

TEST(HashMap, StlAlgorithms) {
    csc::hashmap::HashMap<std::string, std::string> hashmap;
    hashmap["fox"] = "bop";
    hashmap["strong"] = "zero gravity";
    hashmap["don't"] = "wanna talk";
    hashmap["haunted"] = "castle";
    hashmap["masquerade"] = "memememememe";
    auto exp = std::max_element(
        hashmap.begin(), hashmap.end(), [](auto &lhs, auto &rhs) {
            return lhs.key() < rhs.key();
        });
    EXPECT_EQ(exp->key(), "strong");
    exp = std::find_if(hashmap.begin(), hashmap.end(), [](auto &iter) {
        return iter.value() == "castle";
    });
    EXPECT_EQ(exp->key(), "haunted");
    auto node = *hashmap.find("masquerade");
    exp = std::find(hashmap.begin(), hashmap.end(), node);
    EXPECT_EQ(exp->key(), node.key());
    EXPECT_EQ(exp->value(), node.value());
}
```