МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра математического обеспечения и применения ЭВМ

ОТЧЁТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Хеш-таблицы

Студент гр. 8381 _____ Почаев Н.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Изучить основные характеристики и реализовать структуру данных хеш-таблица. В данной реализации для разрешения коллизий использовать метод цепочек. Исследовать сложность основных операций, таких как: вставка, удаление и поиск по ключу.

Постановка задачи.

Реализовать хеш-таблицу с цепочками. Обязательная функциональность:

1. По заданному файлу F (типа file of Elem), все элементы которого различны, построить структуру данных определённого типа – БДП или хеш-таблицу;
2. Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если не входит, то добавить элемент e в структуру данных. Предусмотреть возможность повторного выполнения с другим элементом.

Основные теоретические положения.

Хеш-таблица (англ. *hash-table*) — структура данных, реализующая интерфейс ассоциативного массива. В отличие от деревьев поиска, реализующих тот же интерфейс, обеспечивают меньшее время отклика в среднем. Представляет собой эффективную структуру данных для реализации словарей, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Хеширование (англ. *hashing*) — класс методов поиска, идея которого состоит в вычислении хеш-кода, однозначно определяемого элементом с помощью хеш-функции, и использовании его, как основы для поиска (индексирование в памяти по хеш-коду выполняется за $O(1)$). В общем случае, однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что

количество значений хеш-функций меньше, чем вариантов исходных данных, поэтому существуют элементы, имеющие одинаковые хеш-коды — так называемые коллизии, но если два элемента имеют разный хеш-код, то они гарантированно различаются.

Разрешение коллизий с помощью цепочек. Каждая ячейка i массива H содержит указатель на начало списка всех элементов, хеш-код которых равен i, либо указывает на их отсутствие. Коллизии приводят к тому, что появляются списки размером больше одного элемента.

В зависимости от того нужна ли нам уникальность значений операции вставки у нас будет работать за разное время. Если не важна, то мы используем список, время вставки в который будет в худшем случае равна $O(1)$. Иначе мы проверяем есть ли в списке данный элемент, а потом в случае его отсутствия мы его добавляем. В таком случае вставка элемента в худшем случае будет выполнена за $O(n)$.

Время работы поиска в наихудшем случае пропорционально длине списка, а если все n ключей захешировались в одну и ту же ячейку (создав список длиной n) время поиска будет равно $\Theta(n)$ плюс время вычисления хеш-функции, что ничуть не лучше, чем использование связного списка для хранения всех n элементов.

Удаления элемента может быть выполнено за $O(1)$, как и вставка, при использовании двусвязного списка.

Визуализация примера работы метода цепочек для разрешения коллизий представлена на рис. 1.

Перехеширование. При добавлении в хеш-таблицу большого количества элементов могут возникнуть ухудшения в ее работе. Обработка любого вызова будет занимать больше времени из-за увеличения размеров цепочек при хешировании на списках или кластеризации при хешировании с открытой адресацией, также, при хешировании с открытой адресацией может произойти пере-

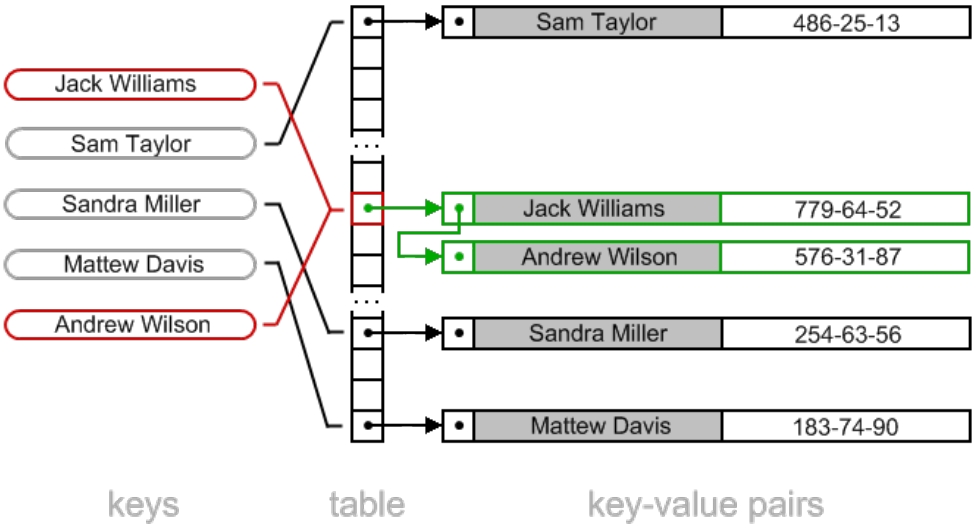

Рисунок 1 - Метод цепочек

полнение таблицы. Для избежания таких ситуаций используется выбор новой хеш-функции и (или) хеш-таблица большего размера. Этот процесс называется перехеширование (англ. *rehashing*).

При использовании хеширования цепочками , элементы с одинаковым результатом хеш-функции помещают в список. Так как операции добавления, поиска и удаления работают за $O(l)$, где l - длина списка, то с некоторого момента выгодно увеличить размер хеш-таблицы, чтобы поддерживать амортизационную стоимость операции $O(1)$.

Рассмотрим следующий алгоритм перехеширования: когда в хеш-таблицу добавлено $\frac{4n}{3}$ элементов, где n - размер хеш-таблицы, создадим новую хеш-таблицу размера $2n$, и последовательно переместим в нее все элементы первой таблицы. При этом, сменим хеш-функцию так, чтобы она выдавала значения $[0..2n-1]$.

Найдем амортизационную стоимость добавления, после которого было сделано перехеширование, используя метод предоплаты. С момента последнего перехеширования было произведено не менее $\frac{2n}{3}$ операций добавления, так как изначально в массиве находится $\frac{2n}{3}$ элементов (или 0 в начале работы), а перехеширование происходит при наличии $\frac{4n}{3}$ элементов.

Для проведения перехеширования необходимо произвести $\frac{4n}{3}$ операций добавления, средняя стоимость которых составляет $O(1)$, потратить $\frac{4n}{3}$ операций на проход хеш-таблицы, и столько же на удаление предыдущей таблицы. В итоге, если мы увеличим стоимость каждой операции добавления на 6, то есть на $O(1)$, операция перехеширования будет полностью предоплачена. Значит, амортизационная стоимость перехеширования при открытом типе хеш-таблицы равна $O(1)$.

Визуализация примеры работы перехеширования представлена на рис. 2.

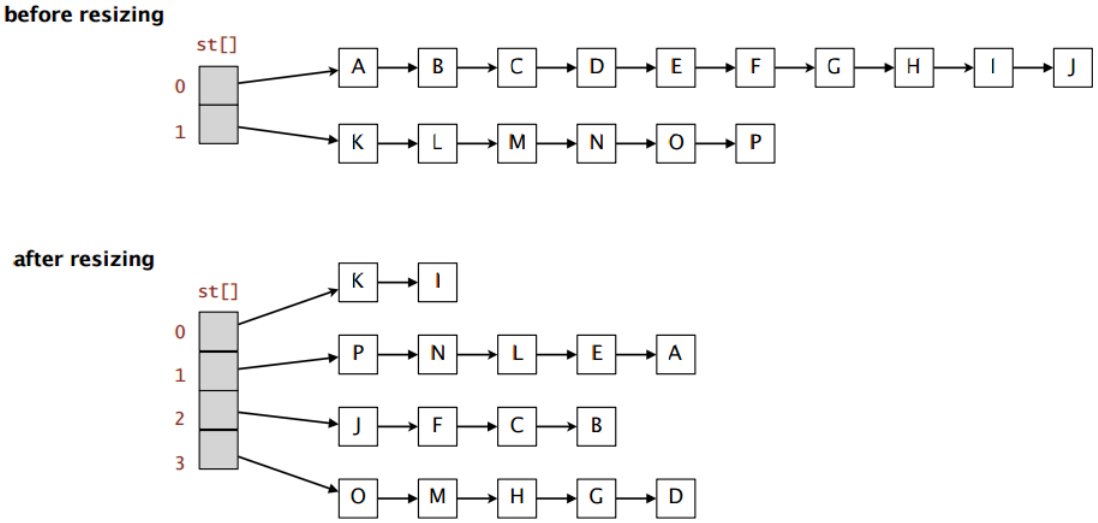

Рисунок 2 - Перехеширование

Выполнение работы.

Написание работы производилось на базе операционной системы Linux Manjaro в среде разработки Qt Creator с использование фреймворка Qt.

Для реализации графического интерфейса в стиле Material Design была использована сторонняя библиотека `laserpants/qt-material-widgets` по открытой лицензии с GitHub.

Хеш-таблица была реализована при помощи контейнерного класса `HashTable`, содержащий в себе класс `Iterator` для произведения обхода и поиска по рабочей хеш-таблице. Исходный код представлен в Приложении А.

Тестирование

Для тестирования созданного класса хеш-таблицы была выбрана структура данных `std::string`. Результаты приведены в таблице 1.

Таблица 1 – Результаты тестирования программы

Кол-во элементов	Размер хеш-функции	Коэффициент заполнения	Количество перехеширований
10	7	1.43	0
25	7	3.57	0
50	14	3.57	0
100	28	3.57	1
500	56	8.93	3
700	112	6.25	4
1000	112	8.93	4
10000	1792	5.58	7

Исходя из данной таблицы можно сделать вывод о корректности алгоритма перехеширования, позволяющего сохранять в допустимых пределах коэффициент заполнения таблицы, а, следовательно, и скорость работы реализованной структуры данных.

Сравнения работы созданной `HashMap` и `std::hash_set` приведено на рис. 3.

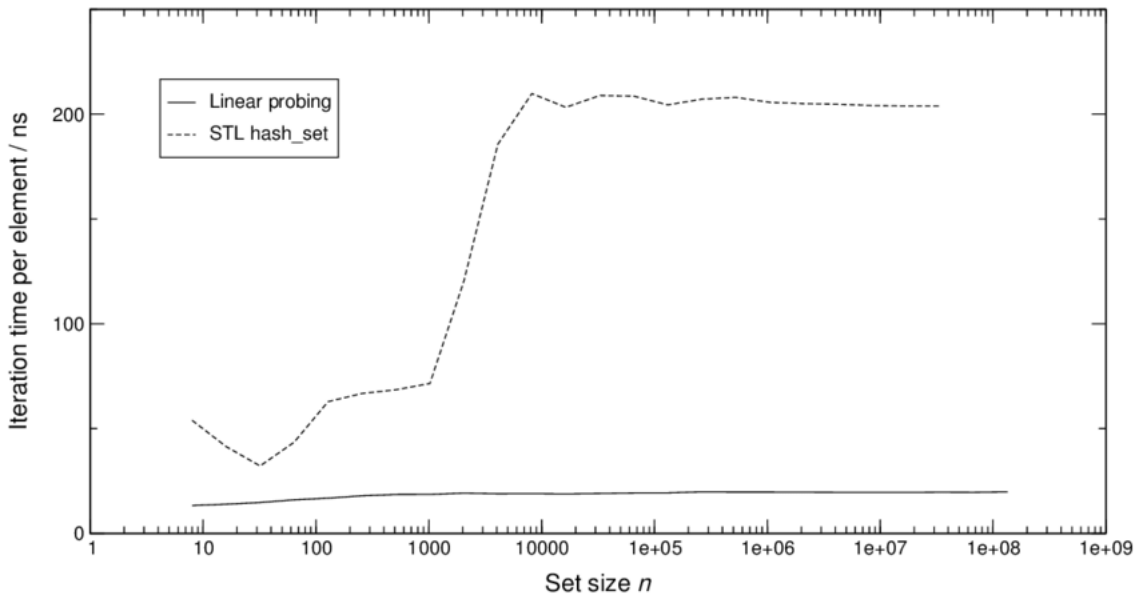

Рисунок 3 - Эффективность работы хеш-таблицы с цепочками

Выводы.

В ходе выполнения лабораторной работы был создан собственный контейнерный класс хэш-функции, использующий для разрешения коллизий метод цепочек. В ходе тестирования были доказаны теоретические положения о сложности работы и эффективности данного алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: HashTable.h

```
#ifndef HASHTEBLE_H
#define HASHTEBLE_H

#include ”allheaders.h”
#include ”customvector.h”

#define DEBUG 0
#define FILE_LOG 1

namespace lrstruct {
    template <typename Key, size_t N = 7/* implementation-defined */>
    class HashTable {

    public:
        /* ----- CONTAINER FIELDS ----- */
        class Iterator;

        // using is to call variable from class namespace without ::
        using value_type = Key;

        //  key_type - the first template parameter (Key)
        //  In other words, it’s just alias name
        using key_type = Key;

        // declares a named variable as a reference,
        // that is, an alias to an already-existing key
        using reference = key_type&;

        // is the same as const Ty&
        using const_reference = const key_type&;
        using size_type = size_t;
        using difference_type = std::ptrdiff_t;
        using iterator = Iterator;
        using const_iterator = Iterator;
        // works for anything that supports a less-than comparison
        using key_compare = std::less<key_type>;   // B+-Tree
        // HASHING
        using key_equal = std::equal_to<key_type>;
```

```
    /**
     * Returns a value of type std::size_t that represents
     * the hash value of the parameter.
     */
    using hasher = std::hash<key_type>;

private:
    /**
     * @brief The Node struct
     * for B+-Tree.
     */
    struct Node {
        Key data;
        Node *next = nullptr;
        Node *head = nullptr;
        ~Node() {
            delete head;
            delete next;
        }
    };
    Node *table = nullptr;
    size_type maxSize{N};
    size_type cSize{0};            // current Hash Table size

    /**
     * @brief insert_unchecked
     * Insert an ellement without checking for collision
     * (for copy constructor mostly)
     * @param k
     */
    void insert_unchecked(const_reference k) {
        Node *help = new Node;
        auto position = hash_idx(k);

        help->next = table[position].head;
        help->data = k;

        table[position].head = help;
        ++cSize;

        if (maxSize * 10 < cSize) rehash();
    }
```

```
/**
 * @brief hash_idx
 * @param k
 * @return
 * load factor of current hash table
 */
size_type hash_idx(const key_type& k) const {
    return hasher{}(k) % maxSize;
}

/**
 * @brief rehash
 * The size of the array is increased (doubled) and all the values
 * are hashed again and stored in the new double sized array
 * to maintain a low load factor and low complexity
 */
void rehash() {
    lrstruct::Vector<Key> dataBuff;
    dataBuff.reserve(cSize);

    for (const auto &k : *this)
        dataBuff.push_back(k);

    delete[] table;
    maxSize *= 2;
    cSize = 0;

    table = new Node[maxSize];
    /*
     * FIXME: for test with more then 71 element
     * it falls. Why??
    for (const auto &fromBuff : dataBuff)
        insert_unchecked(fromBuff);
    */
    for (int i = 0; i < static_cast<int>(dataBuff.size()); i++) {
        insert_unchecked(dataBuff[i]);
    }
    if(DEBUG) qDebug() << ”table rehashed!” << endl;
    if(FILE_LOG) {
        std::fstream fs;
        fs.open (LOG_FILE_WAY, std::fstream::in | \
```

```
            std::fstream::out | std::fstream::app);
            fs << "\nTable rehashed!\n";
            fs << "New max size of hash is: " << maxSize << \
            ", current size is: " << cSize << "\n";
            fs.close();
        }
    }

public:

    /* ----- CONSTRUCTORS ----- */

    HashTable() : table(new Node[N]) {
        if(DEBUG) qDebug() << "DEFAULT_CONSTRUCTOR" << endl;
        if(FILE_LOG) {
            std::fstream fs;
            fs.open (LOG_FILE_WAY, std::fstream::in | \
            std::fstream::out | std::fstream::app);
            fs << "DEFAULT_CONSTRUCTOR\n";
            fs.close();
        }
    }

    HashTable(std::initializer_list<key_type> ilist) : HashTable() {
        insert(ilist);
    }

    /**
     * Recursion call as foreach work
     */
    template<typename InputIt> HashTable(InputIt first, InputIt last) : \
    table(new Node[N]){
        insert(first, last);
        if(DEBUG) qDebug() << "RANGE_CONSTRUCTOR" << endl;
        if(FILE_LOG) {
            std::fstream fs;
            fs.open (LOG_FILE_WAY, std::fstream::in | \
            std::fstream::out | std::fstream::app);
            fs << "RANGE_CONSTRUCTOR\n";
            fs.close();
        }
    }
```

```
HashTable(const HashTable &other) : table(new Node[N]) {
    for (const auto &key : other)
        insert_unchecked(key);
    if(DEBUG) qDebug() << ”COPY_CONSTRUCTOR” << endl;
    if(FILE_LOG) {
        std::fstream fs;
        fs.open (LOG_FILE_WAY, std::fstream::in | \
        std::fstream::out | std::fstream::app);
        fs << ”COPY_CONSTRUCTOR\n”;
        fs.close();
    }
}

~HashTable() {
    delete[] table;
}

/* ----- METHODS ----- */

size_type size() const {
    return cSize;
}

bool empty() const {
    if (cSize == 0)
        return true;
    return false;
}

size_type count(const key_type &key) const {
    const auto row = hash_idx(key);
    for (Node *node = table[row].head; node != nullptr; node = node->next)
        if (key_equal{}(node->data, key)) {
            return 1;
        }
    return 0;
}

iterator find(const key_type& key) const {
    const auto row = hash_idx(key);
    for (Node* n = table[row].head; n != nullptr; n = n->next)
```

```
        if (key_equal{}(n->data, key)) {
            return const_iterator{ this, n, row, maxSize };
        }
    return end();
}

bool findOrInsert(const key_type& key) const {
    const auto row = hash_idx(key);
    for (Node* n = table[row].head; n != nullptr; n = n->next)
        if (key_equal{}(n->data, key)) {
            return true;
        }
    return false;
}

void swap(HashTable& other){
    std::swap(maxSize, other.maxSize);
    std::swap(table, other.table);
    std::swap(cSize, other.cSize);
}

void insert(std::initializer_list<key_type> ilist) {
    for (auto const &element : ilist)
        if (!count(element))
            insert_unchecked(element);
}

std::pair<iterator,bool> insert(const_reference key) {
    if (!count(key)) {
        insert_unchecked(key);
        return std::make_pair(find(key), true);
    }
    return std::make_pair(find(key), false);
}

template<typename InputIt> void insert(InputIt first, InputIt last) {
    for (auto it = first; it != last; ++it)
        insert(*it);
}

void clear() {
    delete[] table;
```

```
    cSize = 0;
    maxSize = N;
    table = new Node[maxSize];
}

size_type erase (const key_type& key) {
    if (count(key)) {
        auto idx = hash_idx(key);

        Node *current = table[idx].head;
        Node *previous = nullptr;

        for ( ; current != nullptr; current = current->next) {
            if (key_equal{}(current->data, key)) {
            if (current == table[idx].head) {
                    table[idx].head = current->next;
                    current->next = nullptr;
                    delete current;

                    --cSize;
                    return 1;
                }
                if (previous) {
                    previous->next = current->next;
                    current->next = nullptr;
                    delete current;

                    previous = nullptr;
                    delete previous;

                    --cSize;
                    return 1;
                }
            }
            previous = current;
        }
    }
    return 0;
}

/* -------- ITERATORS --------*/
```

```
const_iterator begin() const {
    for (size_t i = 0; i < maxSize; i++)
        if (table[i].head) {
            return const_iterator(this, table[i].head, i, maxSize);
        }
    return end();
}

const_iterator end() const {
    return const_iterator(nullptr);
}

void dump(std::ostream& o = std::cerr) const {
    for (size_type i{0}; i < maxSize; ++i) {
        if (table[i].head == nullptr) {
            o << "[" << i << "]" << ": nullptr" << '\n';
            continue;
        }
        o << "[" << i << "]" << ": ";
        for (Node* a = table[i].head; a != nullptr; a = a->next) {
            o << a->data;
            if (a->next != nullptr)
            o << " -> ";
        }
        o << '\n';
    }
}

HashTable& operator = (const HashTable& other) {
    clear();

    for (const auto &key : other)
    insert(key);

    return *this;
}

HashTable& operator = (std::initializer_list<key_type> ilist) {
    clear();
    insert(ilist);

    return *this;
```

```
    }

    friend bool operator == (const HashTable& lhs, const HashTable& rhs) {
        if (lhs.cSize != rhs.cSize)
            return false;

        for (const auto &key : lhs) {
            if (!rhs.count(key)) {
                return false;
            }
        }

        return true;
    }


    friend bool operator != (const HashTable& lhs, const HashTable& rhs) {
        return !(lhs == rhs);
    }
};

/* -------- HASH TABLE ITERATORS --------*/

template <typename Key, size_t N>
class HashTable<Key,N>::Iterator {
    private:
    const HashTable<Key, N> *ptr;
        Node *to;
        size_type itPos;
        size_type tblSz;

    public:
        using value_type = Key;
        using difference_type = std::ptrdiff_t;
        using reference = const value_type&;
        using pointer = const value_type*;
        using iterator_category = std::forward_iterator_tag;

    explicit Iterator(const HashTable *ads = nullptr, Node *tbl = nullptr,
                        size_type idx = 0, size_type sz = 0)
                            : ptr(ads), to(tbl), itPos(idx), tblSz(sz){
        if(DEBUG) qDebug() << "ITERATOR_CONSTRUCTOR" << endl;
```

```
    }

    reference operator*() const {
        return to->data;
    }

    Iterator& operator++() {
        while (itPos < tblSz) {
            if (to->next) {
                to = to->next; return *this;
            }
            else ++itPos;

            if (itPos == tblSz) {
                to = nullptr; return *this;
            }

            auto transit = ptr->table[itPos].head;
            if (transit) {
                to = transit;
                return *this;
            }
        }
        return *this;
    }

    Iterator operator ++ (int) {
        Iterator tmp(*this);
        operator++();

        return tmp;
    }

    friend bool operator == (const Iterator& lhs, const Iterator& rhs) {
        return lhs.to == rhs.to;
    }

    friend bool operator != (const Iterator& lhs, const Iterator& rhs) {
        return !(lhs.to == rhs.to);
    }
};
```

```
    template <typename Key, size_t N> void swap(HashTable<Key,N>& lhs, \
    HashTable<Key,N>& rhs) {
        lhs.swap(rhs);
    }
}

#endif // HASHTEBLE_H
```