



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики

Кафедра высшей математики

КУРСОВАЯ РАБОТА
по дисциплине
«Программирование»

Тема курсовой работы
«Моделирование хеш-таблицы»

Студент группы КТСО-04-20

Литков А.А.

Руководитель курсовой работы
к.ф.-м.н., доцент

Петрусевич Д.А.

Работа представлена к
защите

« 8 » июня 20 21 г. (подпись студента)

«Допущен к защите»

« 25 » мая 20 21 г.  (подпись руководителя)



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт кибернетики

Кафедра высшей математики

Утверждаю
Заведующий
кафедрой _____ Ю. И. Худак
« 15 » _____ марта 2021 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Программирование»

Студент *Литков А.А.* Группа *КТСО-04-20*

1. Тема: «Моделирование хеш-таблицы»

2. Исходные данные:

Смоделировать хеш-таблицу в виде класса с необходимыми методами: добавление, удаление элементов, поиск, хеширование.

При совпадении хеша элементы хранятся в связном списке.

Реализовать произвольный алгоритм хеширования, например, md5.

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:

Продемонстрировать работу хеш-таблицы. Оценить скорость работы по сравнению с обычным массивом при разном количестве элементов.

Продемонстрировать применение элементов методики безопасного программирования.

4. Срок представления к защите курсовой работы: до «29» мая 2021 г.

Задание на курсовую
работу выдал

«15» марта 2021 г.

Задание на курсовую
работу получил

«15» марта 2021 г.

ОТЧЕТ
о курсовой работе
студента 1 курса учебной группы КТСО-04-20
института кибернетики
Российского технологического университета (МИРЭА)
Литкова А.А.

1. Задание на курсовую работу выполнил

В полном объеме

(указать: в полном объеме или частично)

1.1. Не выполнены следующие задания:

(указать также причины невыполнения)

2. Подробное содержание выполненных задач в ходе курсовой работы и достигнутые результаты:

- Проведен обзор литературы по теме построения хэш-таблиц
- Построено решение задачи в виде системы классов
- Приведены примеры работы кода
- Реализованы меры по обеспечению безопасности кода

Студент заслуживает оценки «отлично»

Руководитель практики

доцент кафедры Высшей математики
(должность)


(подпись)

Петрушевич Д.А.
(фамилия и инициалы)

«08»июня 2021 г.

Оглавление

Глава 1. Хэш-таблица, как структура данных	3
1.1 Разрешение коллизий	4
Глава 2. Практическая реализация хэш-таблицы	8
2.1 Описание используемых в реализации классов и функций	8
2.2. Результаты запуска программы	11
2.3. Обработка результатов измерений	12
Глава 3. Методы создания безопасного кода.....	14
Заключение	15
Список литературы	16
Приложение	17

Глава 1. Хэш-таблица, как структура данных

Хэш-таблицы - неотъемлемая часть современной жизни, с ними мы сталкиваемся ежедневно: при входе или регистрации на каких-либо ресурсах, при работе с текстовыми редакторами и переводчиками.

Хэш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, индексация в котором производится путем преобразования некоторого ключа с помощью хэш-функции в индекс.

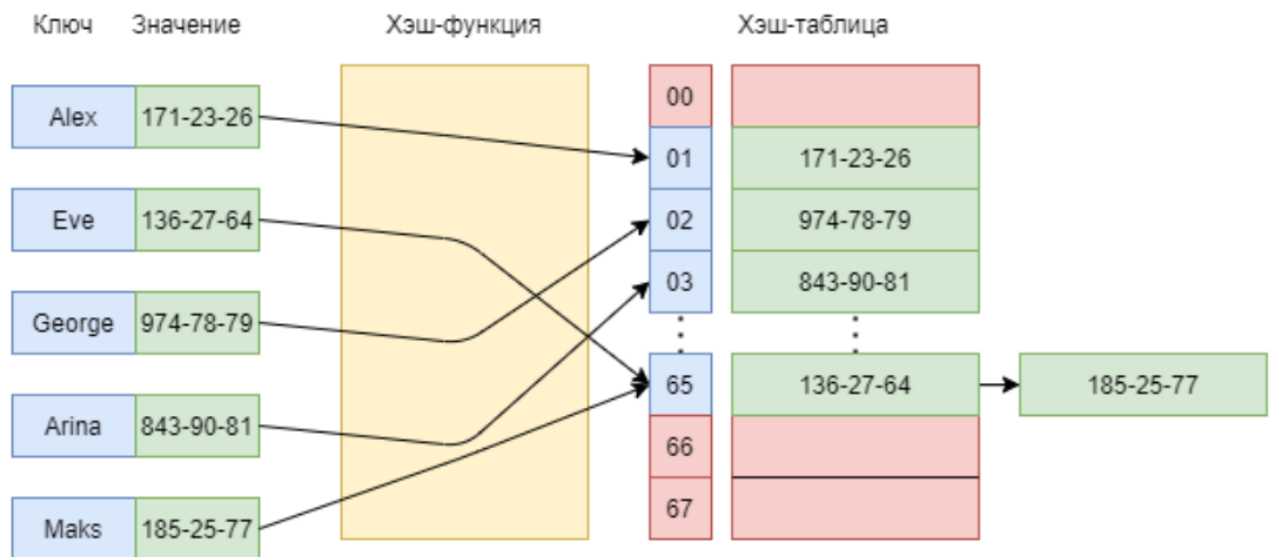


Рисунок 1. Визуализация работы хэш-таблицы, основанной на методе цепочке

Хэш-таблицы являются очень удобным способом хранения данных, а также более выгодным по скорости исполнения наиболее часто используемых методов со стандартными контейнерами данных.

Таблица 1. Сравнение выполнения классических методов в разных контейнерах данных.

Контейнер:	Insert	Remove	Find
Array	$O(N)$	$O(N)$	$O(N)$
Sorted array	$O(N)$	$O(N)$	$O(\log N)$
List	$O(1)$	$O(1)$	$O(\log N)$
Binary search tree	$O(\log N)$	$O(\log N)$	$O(\log N)$
Hash-Table	$O(1)$	$O(1)$	$O(1)$

Из Таблицы 1 видно, что использование хэш-таблиц выгодно по скорости выполнения классических методов, но в свою очередь они занимают очень много памяти.

Хеш-функция – это функция вида hash-function (key), которая берет введенный пользователем ключ и возвращает индекс, по которому в дальнейшем значение будет либо записано в таблицу, либо наоборот будет получено из неё. В идеальной ситуации хэш-функция будет назначать каждому ключу уникальное значение. Хорошая же хеш-функция должна удовлетворять двум требованиям: ее вычисление должно выполняться очень быстро; она должна минимизировать число коллизий. Чаще всего используют один из двух типов хэш-функций: один основан на умножении, а другой на делении.

Теоретически невозможно определить хеш-функцию так, чтобы она создавала случайные данные из реальных неслучайных файлов. Однако на практике реально создать достаточно хорошую имитацию с помощью простых арифметических действий. Более того, зачастую можно использовать особенности данных для создания хеш-функций с минимальным числом коллизий (меньшим, чем при истинно случайных данных)

Метод деления:

Данный метод чаще всего основан на использовании остатка деления на допустимом количестве значений в таблице:

$$h(key) = key \bmod M, \quad (1)$$

где M – допустимое количество различных значений в таблице.

В таком случае, при четном значении M результат $h(key)$ будет четным, при четном K и нечетном – при нечетном, что может привести к значительному сдвигу данных.

Метод умножения (мультипликативный):

Для данного хеширования используется следующая формула:

$$h(K) = [M * ((C * K) \bmod 1)], \quad (2)$$

где C – константа, лежащая в интервале $[0;1]$. После умножения на константу берется дробная часть полученного результата и умножается на M .

1.1 Разрешение коллизий

Чаще всего в конструкции хэш-таблиц используется неидеальная хэш-функция, которая может привести к появлению коллизиям хэш-функции, когда функция возвращает одинаковое значение для разных ключей. Такие случаи должны быть учтены разработчиком каким-либо способом.

Открытая адресация

Один из вариантов решения проблемы коллизий строится на отказе от ссылок и просмотре различных записей таблицы по порядку, пока не будет найден ключ key или пустая ячейка. Данная идея основывается на составлении правила, согласно которому полученному ключу создаётся “пробная последовательность”, а именно последовательность ключей, которая должна быть рассмотрена при вставке или поиске ключа. В таком случае, если будет встречена пустая ячейка, то делается вывод, что ключ в таблице отсутствует, так как алгоритм проходил ту же самую цепочку.

Линейная открытая адресация

Данная схема открытой адресации использует циклическую последовательность проверок и описывается следующим образом ([1], стр. 559):

$$h(key), h(key) - 1, \dots, 0, M - 1; M - 2, \dots, h(key) + 1 \quad (3)$$

Этот алгоритм выполняет поиск данного ключа key в хэш-таблице с M узлами. Если key отсутствует в таблице и таблица не полна, ключ будет вставлен в первую пустую ячейку.

Алгоритм:

1. [Хеширование.] Установить $i = h(key)$. Ячейки таблицы обозначаются как $TABLE[i]$, где $0 \leq i < M$ и могут быть или пустыми, или занятыми.
2. [Сравнение.] Если узел $TABLE[i]$ пуст, то перейти к шагу 4.
3. [Переход к следующему.] Установить $i = i - 1$, если $i < 0$, то $i = i + M$. Вернуться к шагу 2.
4. [Вставка.] Вспомогательная переменная N используется для отслеживания количества занятых узлов. Она увеличивается на 1 при каждой вставке. Если $N = M - 1$, алгоритм завершается в связи с переполнением. В другом случае: увеличить N , пометить ячейку $TABLE[i]$ как занятую и установить в нее значение ключа key .

Данный алгоритм хорошо работает в начале заполнения таблицы, однако по мере заполнения процесс замедляется, а длинные серии проб становятся все более частыми, так как данный алгоритм раз за разом просматривает всю таблицу.

Открытая адресация с двойным хешированием

Данный алгоритм в большей степени похож на предыдущий, но проверяет таблицу иначе, используя две хэш-функции: $h_1(key)$ и $h_2(key)$. Как и в предыдущем алгоритме, значения $h_1(key)$ находятся в диапазоне от 0 до $M - 1$

включительно, а функция $h_2(\text{key})$ должна возвращать значения 1 до $M - 1$, взаимно простые с M ([1], стр. 562).

Алгоритм:

1. [Первое хеширование.] Установить $i = h_1(K)$.
2. [Первая проба] Если узел $\text{TABLE}[i]$ пуст, то перейти к шагу 6.
3. [Второе хеширование.] Установить $s = h_2(K)$.
4. [Переход к следующему.] Установить $i = i - s$, если после этого $i < 0$, установить $i = i + M$.
5. [Сравнение.] Если узел $\text{TABLE}[i]$ пуст, то перейти к шагу 6.
6. [Вставка.] Если $N = M - 1$, алгоритм завершается в связи с переполнением. В другом случае: увеличить N , пометить ячейку $\text{TABLE}[i]$ как занятую и установить в нее значение ключа key .

Этот вариант будет давать значительно более хорошее распределение и независимые друг от друга цепочки. Однако, он медленнее из-за введения дополнительной функции.

Квадратичная и произвольная адресация

Вместо постоянного изменения на единицу, как в случае с линейной адресацией, можно воспользоваться следующей формулой

$$h = h + a^2, \quad (4)$$

где a – это номер попытки. Этот вид адресации достаточно быстр. Чем больше коллизий в таблице, тем дольше этот путь. С одной стороны, этот метод дает хорошее распределение по таблице, а с другой занимает больше времени для подсчета. Произвольная адресация использует заранее сгенерированный список случайных чисел для получения последовательности. Это дает выигрыш в скорости, но несколько усложняет задачу.

Метод цепочек

Данный алгоритм используется в случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, т.к. при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.

Узлы таблицы обозначаются через $TABLE[i]$, $0 \leq i < M$, и могут быть двух различных типов – пустыми и занятыми. В занятом узле $KEY[i]$, после ссылки $LINK[i]$ ([1], стр. 555).

Алгоритм:

1. [Первое хеширование.] Установить $i = h_1(K)$.
2. [Поиск списка] Если $TABLE[i]$ пуст, перейти к шагу 6. В случае, если $TABLE[i]$ занят, то приступаем к поиску в списке, начинающемся в данном узле.
3. [Сравнение.] Если $key = KEY[i]$, алгоритм успешно завершается.
4. [Переход к следующему.] Если $LINK[i] \neq 0$, установить $i = LINK[i]$ и вернуться к шагу 3.
5. [Неудачный поиск.] При неудачном поиске значение R уменьшается один или более раз, пока не будет найдено такое значение, при котором узел $TABLE[R]$ будет пуст. Если $R = 0$, алгоритм завершается в связи с переполнением таблицы.
6. [Вставка нового ключа.] В случае наличия пустого узла, данный узел $TABLE[i]$ помечается как занятый и в него вносится значение ключа.

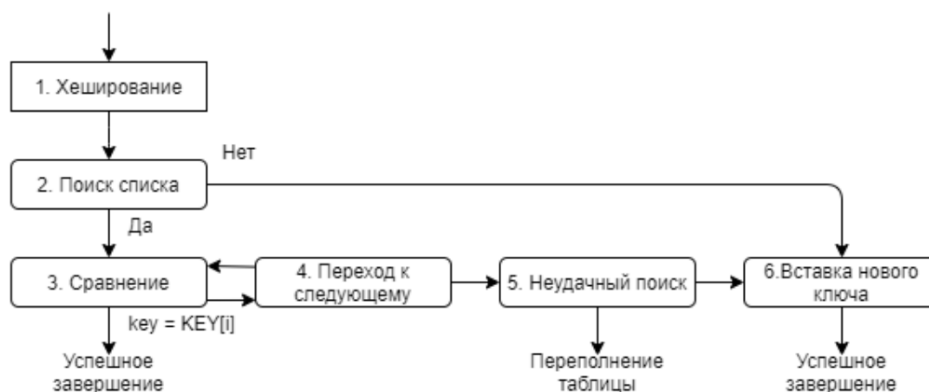


Рисунок 2. Поиск и вставка в хэш-таблице с цепочками.

Данный способ решений коллизий использован в исходном коде на C++.

Глава 2. Практическая реализация хэш-таблицы

Как было ранее упомянуто в Главе 1, для моделирования хэш-таблицы мною был выбран метод цепочек. Он позволяет с легкостью избежать проблему коллизий, а также облегчает подбор хэш-функции.

2.1 Описание используемых в реализации классов и функций

Для хранения Ключа и Значения я создаю класс `Element`, в котором инициализирую защищенные поля класса. Для хранения этих значений использую тип данных `String`, так как он подходит для хранения необходимых данных.

```
class Element
{
private:
    string key;
    string value;

public:
    //Конструктор, принимающий ключ и значение
    Element(string Key, string v)
        :key(Key), value(v) {}

    //Перегруженный оператор логического равенства, необходимый для сравнения двух
    //объектов типа Elementa
    bool operator==(const Element& e12) const;

    //Геттер для получения ключа из Класса
    string get_key() const { return key; }

    //Геттер для получения значения из Класса
    string get_value() const { return value; }

    //Перегруженный оператор побитового сдвига влево, необходимый для вывода объекта
    Element
    friend ostream& operator<<(ostream& strm, const Element& entry) {
        return strm << entry.key << " => " << entry.value;
    }
};
```

Листинг 1. Класс `Element`

Оператор сравнений, а именно равенства, являющийся методом класса `Element` служит нам для сравнения двух объектов типа `Element`, так как сравнение по умолчанию нас не устраивает. Данный оператор используется в методах класса `Hashtable`, который с классом элемент находится в отношениях агрегации.

Для реализации самой хэш-таблицы был создан класс `HashTable`, который содержит в себе поле `max_load`, которое отвечает за максимальную вместимость нашей хэш-таблицы. Данная переменная сразу инициализирована по той причине, что написанная хэш-функция возвращает значения в интервале от 0 до 99, следовательно максимальная загруженность таблицы: 100 ячеек.

Второе поле отвечает за саму структуру хэш-таблицы, храня в себе всю совокупность элементов, где индекс определяется по значению, полученному из хэш-функции при обработке ключа. Представляет из себя вектор списков. Вектор выбран за счёт встроенного механизма контроля выделенной памяти и расширяемости, а список - для решения проблемы коллизий, а также быстрого перебора элементов, добавления и удаления.

```
class HashTable
{
private:
    int max_load = 100;
    vector<list<Element>> hashtable;

public:
    //Описание: Конструктор по умолчанию, который создаёт хэш-таблицу на 100 ячеек.
    HashTable();

    //Описание: Функция, которая определяет заполненность хэш-таблицы
    bool empty() { return hashtable.empty(); }

    //Описание: Хэш-функция, которая преобразует полученные ключи в индексы хэш-таблицы
    static int hashfunction(string key);

    //Описание: Функция вставки нового значения в хэш-таблицу
    void insert(string key, string value);

    //Описание: Функция удаления значения в хэш-таблицу
    void erase(string key, string value);

    //Описание: Функция, которая производит поиск значения по введенному ключу
    string get_value(string key);

    //Описание: Функция которая выводит хэш-таблицу в консоль
    void print_table();

    //Описание: Функция, которая заполняет хэш-таблицу псевдорандомными значениями
    void FillHashTable(const int N = 100);
};
```

Листинг 2. Класс HashTable

Конструктор по умолчанию, используемый в данном классе, с помощью встроенного метода класса `Vector`: *resize*, автоматически изменяет размер задействованного вектора, в моем случае на сто ячеек.

Функция *empty* может быть использована для определения наличия каких-либо данных. Как и в предыдущем конструкторе в ней используется встроенный метод класса `Vector`: *empty*, который возвращает `true` в случае, если контейнер и `false` в противном.

Функция *hashfunction* преобразует ключ, в индекс реализуемой хэш-таблицы. Она возвращает остаток от деления на сто полученного хэша. Значение, полученное из данной функции, располагается на интервале от 0 до

99. Это позволяет, сразу определить максимальную вместимость хэш-таблицы.

Функция *insert* производит вставку полученного значения в таблицу. Она получает на вход ключ (key) и значение (value). Внутри себя она вызывает *hashfunction* и преобразовывает полученный ключ в индекс контейнера и создаёт объект класса *Element* в котором хранятся ключ и значение. Данный объект добавляется в список, который хранится в ячейке вектора по индексу, полученному из ключа. В своей реализации я использую контейнер *List*, который представляет собой двусвязный список. Данной структуры данных имеется встроенный метод: *push_front*, добавляющий наш новый элемент в начало списка. Также я использую встроенную функцию: *unique*, которая удаляет все дубликаты из списка.

Функция *erase* производит удаление значения из хэш-таблицы. Она получает ключ и значение, которое необходимо удалить. С помощью функции *hashfunction* получает значение индекса, по которому располагается список, в котором располагается объект, который необходимо удалить. Далее в цикле просматривается список от его начала до конца, это производится с помощью встроенных функций контейнера *List*: *cbegin* и *cend*. Они возвращают константные итераторы, адресуящие первый элемент в списке и последний соответственно. Далее элемент контейнера, на который указывает итератор сравнивается с удаляемым, и в случае их соответствия удаляется, в противоположной ситуации проверка продолжается.

Функция *get_value* используется для получения значения по переданному ей ключу. Как и в предыдущей функции полученный ключ преобразуется в индекс и с помощью него находится ячейка вектора, где располагается список с необходимым нам значением. Далее в цикле просматривается список от его начала до конца. С помощью геттера: *get_key*, введённого в классе *Element*, происходит сравнение искомого ключа с полученным. В случае, если они совпадают, с помощью другого геттера из класса *Element*: *get_value*, возвращается искомое значение, в противоположном случае выводится ошибка: "Nothing was found".

Функция *print_table* необходима для вывода полученной таблицы в консоль. В ней в цикле просматривается список в каждой ячейке вектора и в случае, если там не пусто, находящиеся в ней объекты выводятся в консоль.

Функция *FillHashTable* используется для заполнения нашей таблицы псевдослучайными ключами и значениями. Она необходима для демонстрации работы хэш-таблицы с большим количеством данных.

2.2. Результаты запуска программы

Основной задачей выполнения данной работы является реализация хэш-таблицы и реализация ее базовых методов: добавление новых значение в хэш-таблицу, удаление значений из таблицы, поиск элементов по ключу. Данные функции подробно описаны в главе 2.1, здесь же я привожу результаты работы программы. Код, вводимый для вызова данных процедур, можно найти в приложении [1].

Демонстрация базовых методов:

1) Создаётся хэш-таблица, заполненная четырьмя псевдослучайными значениями. Ключи для кодирования кодируются таким же образом.

2) В функцию *Insert* я передаю значение, которое я хочу расположить в контейнере, а также ключ. В ходе работы данной функции, работает метод *hashfunction*, который преобразует ключ в хэш.

3) С помощью метода *print_table* я вывожу измененную таблицу в консоль. Как видно на Рисунке 3, данное значение находится в таблице с хэшем равным 41.

4) Используя *get_value*, я могу найти записанное значение, передав в данную функцию ключ: Alex. Как видно на Рисунке 3, значение будет равно 171-23-26.

5) Используя функцию *erase* и передав в неё значение и ключ, которые необходимо удалить, я очищаю ячейку и вновь вывожу таблицу в консоль.

```
IZ => 5153 with hash 13
7W => 4176 with hash 27
jn => 546 with hash 66
28 => 6045 with hash 86

Insert value: 171-23-26 with key: Alex

Jj =>  with hash 10
IZ => 5153 with hash 13
7W => 4176 with hash 27
Alex => 171-23-26 with hash 41
jn => 546 with hash 66
28 => 6045 with hash 86

Found value 171-23-26

Erase value: 171-23-26 with key: Alex

Jj =>  with hash 10
IZ => 5153 with hash 13
7W => 4176 with hash 27
jn => 546 with hash 66
28 => 6045 with hash 86
```

Рисунок 3. Демонстрация базовых методов.

2.3. Обработка результатов измерений

Одной из целей данной работы является сравнение скорости работы хэш-таблицы с обычным одномерным массивом. Для чистоты эксперимента я сравниваю скорость выполнения при разной заполненности контейнеров: 1) при 10 элементах, 2) при 10.000 элементов, 3) при 10.000.000 элементов. Все измерения проводились с помощью библиотеки *<chrono>*, а также все измерения получены в наносекундах. Для каждого измерения, я вызывал базовые методы трижды для получения усреднённого значения.

Измерение №1.

Основываясь на медианных значениях из приложениях Приложения 2 и Приложения 3, я построил график, изображенный на графике в Приложении 4, для сравнения скорости работы базовых методов в хэш-таблице и в массиве, при 10 элементах. Из него видно, что при малой загрузке контейнеров вставка нового элемента и удаление элемента, работает намного быстрее в массиве, чем в хэш-таблицы, но поиск в свою очередь работает быстрее в хэш-таблицы.

Измерение №2.

Основываясь на медианных значениях из приложениях Приложения 5 и Приложения 6, я построил график, изображенный на графике в Приложении 7, для сравнения скорости работы базовых методов в хэш-таблице и в массиве, при 10.000 элементах. Как видно из полученной диаграммы, при подобной заполненности скорость работы функций в данных контейнера соизмерима.

Измерение №3.

Основываясь на медианных значениях из приложениях Приложения 8 и Приложения 9, я построил график, изображенный на графике в Приложении 10, для сравнения скорости работы базовых методов в хэш-таблице и в массиве, при 10.000.000 элементах. При подобной заполненности можно заметить, что выполнение базовых функций в массиве занимает больше времени примерно в 15 раз.

Вывод:

Основываясь на проведенных экспериментах, можно с уверенностью сказать, что использование хэш-таблицы при малом количестве элементов является неликвидным. Но если дело касается массива данных, котором больше 10.000 элементов, то хэш-таблица является выгодным по скорости работы решением.

Как видно из диаграмм, изображенный в приложениях 4, 7 и 10: скорость выполнения базовых методов в хэш-таблицы не зависит от количества

элементов в таблице и является константным. В свою очередь скорость выполнения этих задач напрямую зависит от количества элементов в нём.

Глава 3. Методы создания безопасного кода

Безопасность кода является очень важной частью разработки кода, ведь если код не безопасен, то он может работать некорректно, или вовсе не выполняться.

Выделение памяти:

В своей реализации хэш-таблицы я использую класс *vector*. Когда переменная-вектор выходит из области видимости, то она автоматически освобождает память, которую занимала. Это не только удобно (так как не нужно это делать вручную), но также помогает предотвратить утечки памяти. Именно из-за этого использование `std::vector` является более безопасным, чем динамическое выделение памяти через оператор `new`.

В данной реализации хэш-таблицы для решения проблемы коллизий используется метод цепочек, который позволяет задать фиксированный размер таблицы. Таким образом, выход за пределы области видимости невозможен.

В методе *FillHashTable(int N)*, который заполняет хэш-таблицу псевдослучайными значениями используется динамическое выделение памяти с помощью оператора `new[]`.

Обращение к освобожденной памяти:

В данной работе используется класс *List*, который имеет встроенные метод удаления элементов списка: *erase()*. Он освобождает память, выделенную под объект, и перераспределяет указатели, таким образом возможность обращения к освобожденной памяти отсекается.

Методы используемые для обеспечения безопасности в коде:

Для того, что бы избежать ситуации, когда в таблицу внесены одинаковые значения с одинаковыми ключами, используется метод класса *List*: *unique()*. При добавлении нового элемента весь список просматривается с начала до конца, и в случае, если найдены дубликаты, они удаляются. Данный хоть и позволяет избавиться от дубликатов, но в свою очередь на его выполнения тратится очень много времени.

Заключение

Хэш-таблица — это структура данных, реализующая интерфейс ассоциативного массива. Она является тяжеловесной структурой, но в свою очередь скорость выполнения её базовых методов: поиск, удаление, добавление нового элемента — не зависит от количества элементов в таблице. Как видно из проведенных измерений, использование хэш-таблиц выгодно лишь тогда, когда речь идёт о больших массивах данных, в противном случае, обычный массив будет работать быстрее.

Также я хотел бы сравнить хэш-таблицу со сбалансированным по высоте двоичным деревом поиска и красно-чёрным деревом. В отличие от хэш-таблицы, у которой операции поиска, вставки и удаления выполняются за $O(N)$, у данных структур данных они выполняются за $O(\log N)$. Из этого следует, что хэш-таблица — не самая выгодная структура, но в свою очередь она хорошо подходит для формирования словарей, кодирования данных и ещё для множества других задач.

Список литературы

1. Кнут Д. Искусство программирования, т.3. М.: Вильямс, 2000. - 800с.
2. Шилдт Г. Искусство программирования на C++. СПб.: БХВ-Петербург, 2004.-496 с.
3. Грэхем И. Объектно-ориентированные методы. Принципы и практика/ 3-е изд. М.: Вильямс, 2004. – 880с.

Приложение

```
int main()
{
    HashTable t;
    t.FillHashTable(5);
    t.print_table();

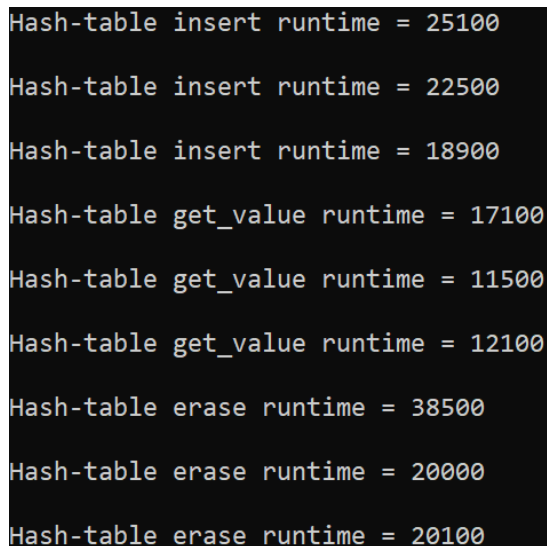
    cout << "Insert value: 171-23-26 with key: Alex" << endl<< endl;
    t.insert("Alex", "171-23-26");
    t.print_table();

    cout << "Found value " << t.get_value("Alex")<< endl<< endl;

    t.erase("Alex", "171-23-26");
    cout << "Erase value: 171-23-26 with key: Alex" << endl << endl;
    t.print_table();

    return 0;
}
```

Приложение 1. Вызов базовых методов.



```
Hash-table insert runtime = 25100
Hash-table insert runtime = 22500
Hash-table insert runtime = 18900
Hash-table get_value runtime = 17100
Hash-table get_value runtime = 11500
Hash-table get_value runtime = 12100
Hash-table erase runtime = 38500
Hash-table erase runtime = 20000
Hash-table erase runtime = 20100
```

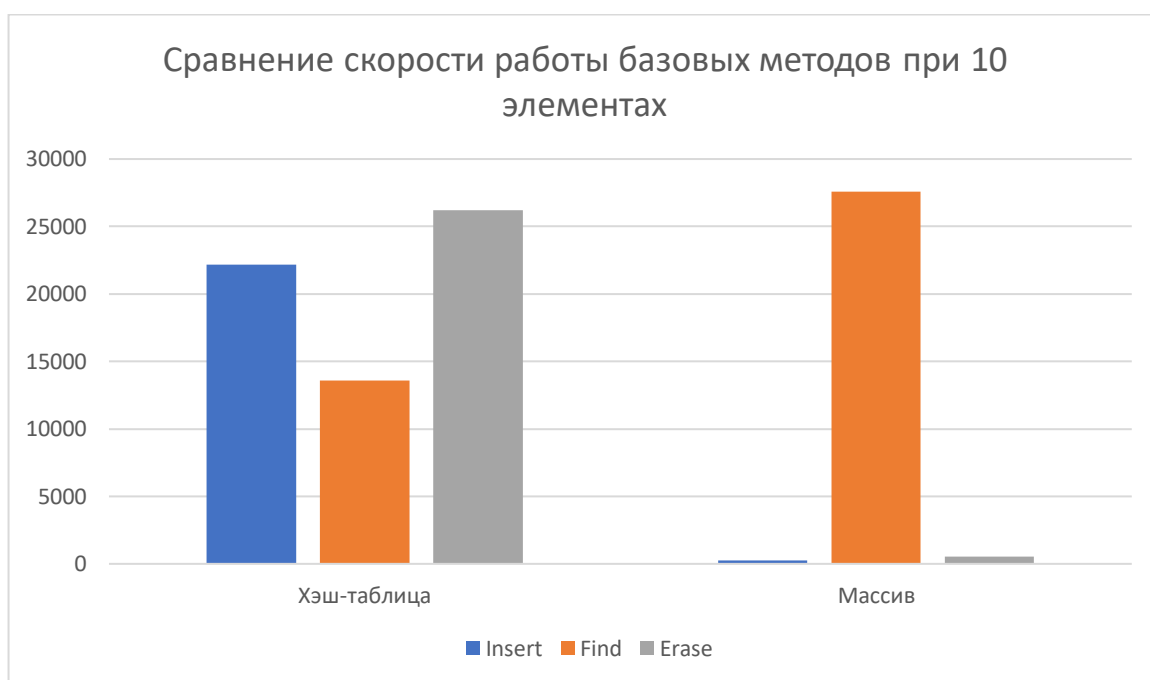
Приложение 2. Скорость выполнения базовых методов класса HashTable в наносекундах при 10 элементах.

```

Array insert runtime = 300
Array insert runtime = 300
Array insert runtime = 200
index of element 2
Array Find runtime = 67100
index of element 8
Array Find runtime = 59900
not foundindex of element -1
Array Find runtime = 70080
Array Remove runtime = 600
Array Remove runtime = 500
Array Remove runtime = 500

```

Приложение 3. Скорость выполнения базовых методов класса ArrayParent в наносекундах при 10 элементах.



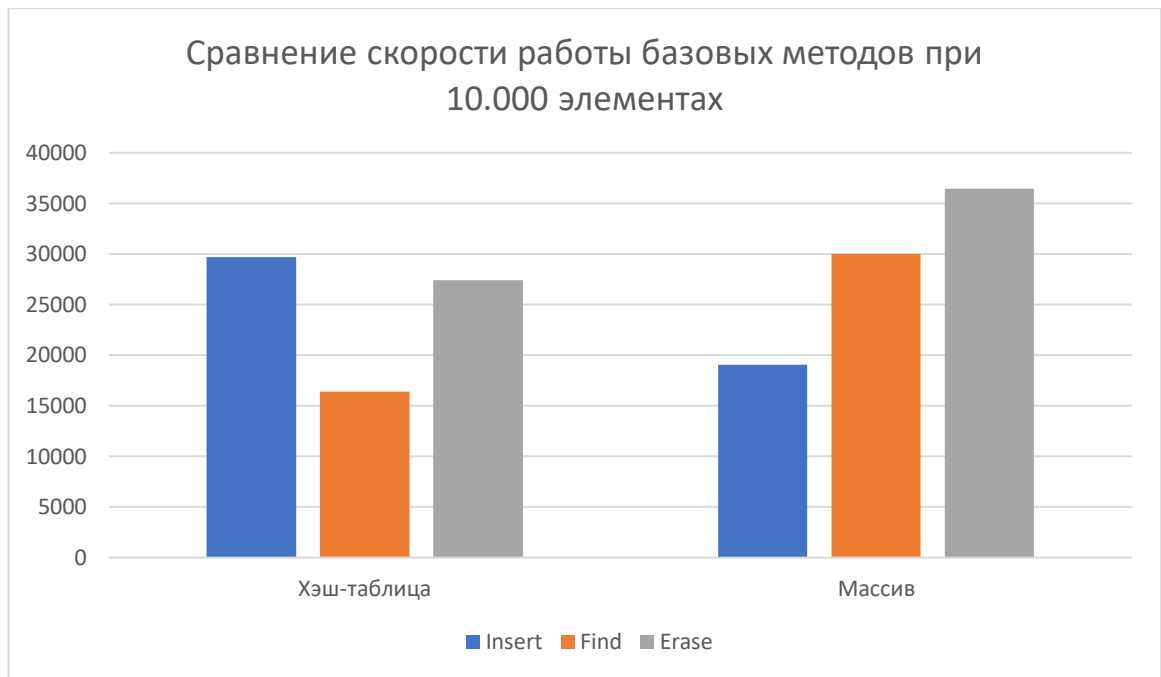
Приложение 4. Диаграмма, изображающая сравнение скорости работы хэш-таблицы и обычного массива в наносекундах, которая основана на приложении [2] и [3].

```
Hash-table insert runtime = 43500
Hash-table insert runtime = 26300
Hash-table insert runtime = 19200
Hash-table get_value runtime = 17600
Hash-table get_value runtime = 11300
Hash-table get_value runtime = 20300
Hash-table erase runtime = 41400
Hash-table erase runtime = 20500
Hash-table erase runtime = 20400
```

Приложение 5. Скорость выполнения базовых методов класса HashTable в наносекундах при 10000 элементах.

```
Array insert runtime = 500
Array insert runtime = 26200
Array insert runtime = 30400
index of element 0
Array Find runtime = 81900
index of element 5
Array Find runtime = 76300
index of element 2
Array Find runtime = 74110
Array Remove runtime = 18200
Array Remove runtime = 39800
Array Remove runtime = 51400
```

Приложение 6. Скорость выполнения базовых методов класса ArrayParent в наносекундах при 10000 элементах.



Приложение 7. Диаграмма, изображающая сравнение скорости работы хэш-таблицы и обычного массива в наносекундах, которая основана на приложении [5] и [6].

```

Hash-table insert runtime = 33900
Hash-table insert runtime = 31800
Hash-table insert runtime = 23800
Hash-table get_value runtime = 16000
Hash-table get_value runtime = 11500
Hash-table get_value runtime = 11700
Hash-table erase runtime = 38500
Hash-table erase runtime = 20100
Hash-table erase runtime = 20000
  
```

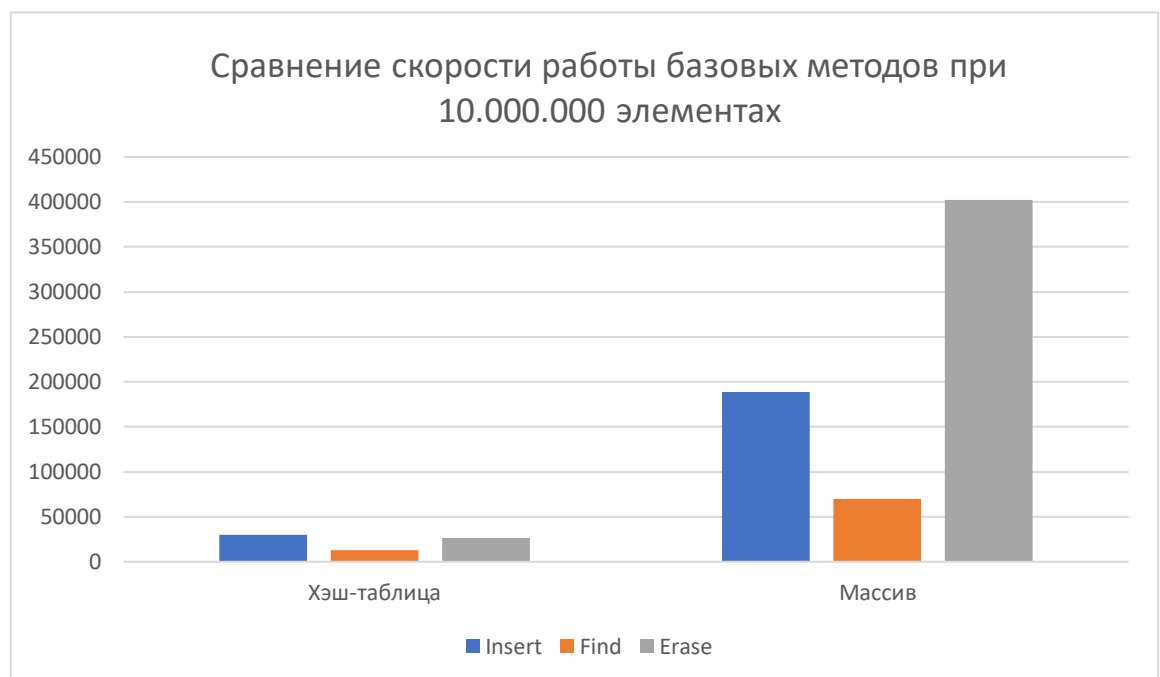
Приложение 8. Скорость выполнения базовых методов класса HashTable в наносекундах при 10000000 элементах.

```

Array insert runtime = 23500
Array insert runtime = 260800
Array insert runtime = 302730
index of element 1
Array Find runtime = 92700
index of element 5
Array Find runtime = 80300
index of element 0
Array Find runtime = 378290
Array Remove runtime = 349843
Array Remove runtime = 411780
Array Remove runtime = 445862

```

Приложение 9. Скорость выполнения базовых методов класса ArrayParent в наносекундах при 10000000 элементах.



Приложение 10. Диаграмма, изображающая сравнение скорости работы хэш-таблицы и обычного массива в наносекундах, которая основана на приложении [8] и [9].

Код курсовой работы:

```
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <algorithm>
#include <cfloat>
#include "Element.h"
using namespace std;

class Element;

class HashTable
{
private:
    int max_load = 100;
    vector<list<Element>> hashtable;

public:
    //Описание: Конструктор по умолчанию, который создаёт хэш-таблицу на 100 ячеек.
    HashTable();

    //Описание: Функция, которая определяет заполненность хэш-таблицы
    bool empty() { return hashtable.empty(); }

    //Описание: Хэш-функция, которая преобразует полученные ключи в индексы хэш-таблицы
    static int hashfunction(string key);

    //Описание: Функция вставки нового значения в хэш-таблицу
    void insert(string key, string value);

    //Описание: Функция удаления значения в хэш-таблицу
    void erase(string key, string value);

    //Описание: Функция, которая производит поиск значения по введенному ключу
    string get_value(string key);

    //Описание: Функция которая выводит хэш-таблицу в консоль
    void print_table();

    //Описание: Функция, которая заполняет хэш-таблицу псевдорандомными значениями
    void FillHashTable(const int N = 100);

};

#endif
```

Приложение 11. Заголовочный файл HashTable.h

```
#include "HashTable.h"
#include <vector>
#include <string>
#include <list>

using namespace std;

HashTable::HashTable()
```



```

{
    hashtable.resize(1000);
}

void HashTable::print_table()
{
    Element someel("0", "0");

    for (auto it1 = hashtable.cbegin(); it1 != hashtable.cend(); ++it1)
    {
        if ((*it1).empty()) continue;
        else
            for (auto it2 = (*it1).cbegin(); it2 != (*it1).cend(); ++it2)
                cout << (*it2) << " with hash " <<
hashfunction((*it2).get_key())<<endl;
    }
    cout << endl;
}

int HashTable::hashfunction(string key)
{
    int hash = 0;
    for (size_t i = 0; i < key.size(); ++i)
    {
        int tmp = key[i];
        tmp = tmp << (i * 8 % 24);
        hash = hash ^ tmp;
    }
    return hash % 100;
}

void HashTable::insert(string key, string value)
{
    int index = hashfunction(key);

    Element someElement(key, value);
    hashtable[index].push_front(someElement);
    //hashtable[index].unique();
}

void HashTable::erase(string key, string value)
{
    int index = hashfunction(key);

    Element someelement(key, value);

    for (auto it = hashtable[index].cbegin(); it != hashtable[index].cend(); ++it)
        if (*it == someelement) {
            hashtable[index].erase(it);
            break;
        }
}

string HashTable::get_value(string key)
{
    int index = hashfunction(key);

    for (auto it = hashtable[index].cbegin(); it != hashtable[index].cend(); ++it)
        if ((*it).get_key() == key)
        {
            return (*it).get_value();
        }

    cerr << "Nothing was found";
}

```

```

        return 0;
    }

    void HashTable::FillHashTable(int N)
    {
        srand(unsigned(time(NULL)));
        string* arr = new string [N];
        string* ptr = new string [N];
        string key;
        string value1;

        char sym = '/';
        for (int j = 0; j < N; ++j) {
            key = "";
            for (int I = 0; I < 2; ++i) {
                sym = '/';

                while ((sym < '0') || (sym > '9' && sym < 'A') || (sym > 'Z' && sym <
'a') || (sym > 'z'))
                    sym = static_cast<char> (rand() % 123 + 48);

                key += sym;
            }
            arr[j] = key;

            int value = 0;
            for (int I = 0; I < N-1; i++)
            {
                value = rand() % 10000;
                ptr[i] = to_string(value);
            }

            for (int I = 0; I < N; i++)
            {
                key = "";
                value1 = "";
                key = arr[i];
                value1 = ptr[i];
                insert(key, value1);
            }
        }
    }
}

```

Приложение 12: Файл с определениями методов класса HashTable

```

#ifndef ELEMENT_H
#define ELEMENT_H

#include <iostream>
#include <vector>
#include <string>
#include <list>
#include <algorithm>
#include <cmath>

using namespace std;

```

```

class Element
{
private:
    string key;
    string value;

public:
    //Конструктор, принимающий ключ и значение
    Element(string Key, string v)
        :key(Key), value(v) {}

    //Перегруженный оператор логического равенства, необходимый для сравнения двух
    объектов типа Elementa
    bool operator==(const Element& el2) const;

    //Геттер для получения ключа из Класа
    string get_key() const { return key; }

    //Геттер для получения значения из Класа
    string get_value() const { return value; }

    //Перегруженный оператор побитового сдвига влево, необходимый для вывода объекта
    Element
    friend ostream& operator<<(ostream& strm, const Element& entry) {
        return strm << entry.key << " => " << entry.value;
    }
};

#endif

```

Приложение 13. Заголовочный файл Element.h

```

#include "Element.h"

bool Element::operator==(const Element& el2) const
{
    if ((this->key == el2.key) && (this->value == el2.value))
        return true;
    return false;
}

```

Приложение 14. Файл с определениями методов класса Element

```

#ifndef ARRAYPARENT_H
#define ARRAYPARENT_H

#include <iostream>
#include <ctime>

using namespace std;
class ArrayParent
{
protected:
    //сколько памяти выделено
    int capacity;
    //количество элементов
    int count;
    //массив

```

```

    int* ptr;
public:
    //конструкторы и деструктор
    ArrayParent(int Dimension);

    //конструктор принимает существующий массив
    ArrayParent(int SonCount, int* SonPtr);

    //конструктор копий
    ArrayParent(const ArrayParent& V);

    ~ArrayParent();

    int operator[] (int index);
    ArrayParent& operator=(const ArrayParent& V);
    void print();
    void push_back(int value);
    void Fill();
    int Find(double value);
    void InsertAt(int value, int index = -1);
    void RemoveAt(int index);
};

#endif

```

Приложение 15. Заголовочный файл ArrayParent.h

```

#include "ArrayParent.h"
#include <iostream>
#include <ctime>

using namespace std;

ArrayParent::ArrayParent(int Dimension)
{
    ptr = new int[Dimension];
    capacity = Dimension;
    count = 0;
}

ArrayParent::ArrayParent(int SonCount, int* SonPtr)
{
    count = SonCount;
    capacity = SonCount;

    ptr = new int[capacity];
    for (int i = 0; i < SonCount; i++)
        ptr[i] = SonPtr[i];
}

ArrayParent::~ArrayParent()
{
    if (ptr != NULL)
    {
        delete[] ptr;
        ptr = NULL;
    }
}

ArrayParent::ArrayParent(const ArrayParent& V)
{
    count = V.count;
    capacity = V.capacity;
}

```

```

        ptr = new int[V.capacity];

        for (int i = 0; i < V.count; i++)
            ptr[i] = V.ptr[i];
    }

    // Оператор [], для возвращения индекса элемента
    int ArrayParent::operator [] (int index)
    {
        return ptr[index];
    }

    //Оператор присваивания
    ArrayParent& ArrayParent::operator=(const ArrayParent& V)
    {
        count = V.count;
        capacity = V.capacity;

        ptr = new int[V.capacity];
        for (int i = 0; i < V.count; i++)
            ptr[i] = V.ptr[i];
        return *this;
    }

    //Функция вывода
    void ArrayParent::print()
    {
        cout << "\nMyArrParent, size: " << capacity << ", values: {";
        int i = 0;
        for (i = 0; i < capacity; i++)
        {
            cout << ptr[i]<<" ";
            //if (i != capacity - 1)
            //cout << ", ";
        }
        cout << "}";
    }

    //заполнение массива случайными числами
    void ArrayParent::Fill()
    {
        srand((unsigned)time(NULL));

        double value = 0;
        for (int i = 0; i < capacity; i++)
        {
            value = rand() % 10000;
            ptr[i] = value;
        }
    }

    //Функция поиска
    int ArrayParent::Find(double value)
    {
        for (int i = 0; i < capacity; i++)
        {
            if (ptr[i] == value)
                return i;
        }
        cout << "not found";
        return -1;
    }

```

```

//добавление в конец нового значения
void ArrayParent::push_back(int value)
{
    if (count < capacity)
    {
        ptr[count] = value;
        count++;
    }
}

//Функция добавления
void ArrayParent::InsertAt(int value, int index)
{
    for (int i = 0; i < index; i++)
    {
        ptr[i] = ptr[i+1];
    }
    ptr[index] = value;
}

//удаление элемента
void ArrayParent::RemoveAt(int index)
{
    for (int i = index; i < capacity - 1; i++)
        ptr[i] = ptr[i + 1];

    capacity--;
}

```

Приложение 15. Файл с определениями методов класса ArrayParent

```

#include <iostream>
#include <fstream>
#include "Element.h"
#include "HashTable.h"
#include "ArrayParent.h"
#include <chrono>

using namespace std;
using namespace chrono;

int main()
{
    setlocale(LC_ALL, "Russian");
    int choice = 0;
    HashTable t, t1, t2, t3;
    system_clock::time_point start;
    system_clock::time_point end;
    cout << "Сравнить скорость работы базовы методов:"
        <<endl<<"при 10 элементах [1],"
        <<endl<<"при 10.000 элементах [2],"
        <<endl<<"при 1.000.000 элемтах [3],"
        <<endl<<endl<<"Использовать базовые методы [4].";
    cin >> choice;
    if ((choice != 1) && (choice != 2) && (choice != 3)&&(choice !=4))
    {
        cout << "Введено некоректное значение. Работа программы остановлена";
        return 0;
    }
    switch (choice)
    {
    case 1:
    {

```

```

t.FillHashTable(10);
cout << "Hash-table"<<endl;
system_clock::time_point start = system_clock::now();
t.insert("Alex", "171-23-26");
system_clock::time_point end = system_clock::now();
cout << "Time of insert in hash-table with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
t.get_value("Alex");
end = system_clock::now();
cout << "Time of search in hash-table with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
t.erase("Alex", "171-23-26");
end = system_clock::now();
cout << "Time of delete in hash-table with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;
cout << endl;

cout << "Array"<<endl;
ArrayParent Array(10);
start = system_clock::now();
Array.InsertAt(23, 5);
end = system_clock::now();
cout << "Time of insert in Array with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
Array.Find(23);
end = system_clock::now();
cout << "Time of find in Array with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
Array.RemoveAt(5);
end = system_clock::now();
cout << "Time of delete in Array with 10 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;
break;
}
case 2:
{
t1.FillHashTable(10000);

cout << "Hash-table" << endl;
start = system_clock::now();
t1.insert("Alex", "171-23-26");
end = system_clock::now();
cout << "Time of insert in hash-table with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
t1.get_value("Alex");
end = system_clock::now();
cout << "Time of search in hash-table with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

start = system_clock::now();
t1.erase("Alex", "171-23-26");
end = system_clock::now();
cout << "Time of delete in hash-table with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

```

```

        cout << endl;

        cout << "Array" << endl;
        ArrayParent Array1(10000);
        start = system_clock::now();
        Array1.InsertAt(23, 6666);
        end = system_clock::now();
        cout << "Time of insert in Array with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        Array1.Find(2789);
        end = system_clock::now();
        cout << "Time of find in Array with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        Array1.RemoveAt(9098);
        end = system_clock::now();
        cout << "Time of delete in Array with 10.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;
        break;
    }

    case 3:
    {
        t2.FillHashTable(1000000);

        cout << "Hash-table" << endl;
        start = system_clock::now();
        t2.insert("Alex", "171-23-26");
        end = system_clock::now();
        cout << "Time of insert in hash-table with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        t2.get_value("Alex");
        end = system_clock::now();
        cout << "Time of search in hash-table with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        t2.erase("Alex", "171-23-26");
        end = system_clock::now();
        cout << "Time of delete in hash-table with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;
        cout << endl;

        cout << "Array" << endl;
        ArrayParent Array2(1000000);
        start = system_clock::now();
        Array2.InsertAt(2356, 25);
        end = system_clock::now();
        cout << "Time of insert in Array with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        Array2.Find(9356);
        end = system_clock::now();
        cout << "Time of find in Array with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        start = system_clock::now();
        Array2.RemoveAt(500000);

```



```

        end = system_clock::now();
        cout << "Time of delete in Array with 1.000.000 elements: " <<
duration_cast<nanoseconds>(end - start).count() << " nanoseconds" << endl;

        break;
    }
    case 4:
    {
        t3.FillHashTable(10);

        t3.print_table();

        string key, value;
        cout << "Вставка элемента" << endl;
        cout << "Введите ключ" << endl;
        cin >> key;
        cout << "Введите значение" << endl;
        cin >> value;
        t3.insert(key, value);

        t3.print_table();

        cout << "Удаление элемента" << endl;
        cout << "Введите ключ" << endl;
        cin >> key;
        cout << "Введите значение" << endl;
        cin >> value;
        t3.erase(key,value);

        t3.print_table();
    }

}
}
}

```