



МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации информационных
технологий

Отчет по практической работе №3

по дисциплине «Структуры и алгоритмы обработки данных»
по теме «Хеш-таблицы»

Выполнил:

Студент группы ИКБО-13-22

Лещенко Вячеслав Романович

Проверил:

ассистент Муравьёва Е.А.

МОСКВА 2023 г.

Практическая работа № 3

Цель работы

Получить навыки по разработке хеш-таблиц и их применении при поиске данных в других структурах данных (файлах).

Ход работы

Вариант 20

20	Открытый адрес(смещение на 1)	Продажи товаров: код товара, название, цена, дата продажи.
----	-------------------------------	--

Задание 1

Формулировка задачи:

Ответьте на вопросы:

- 1) Расскажите о назначении хеш-функции.
- 2) Что такое коллизия?
- 3) Что такое «открытый адрес» по отношению к хеш-таблице?
- 4) Как в хеш-таблице с открытым адресом реализуется коллизия?
- 5) Какая проблема, может возникнуть после удаления элемента из хештаблицы с открытым адресом и как ее устранить?
- 6) Что определяет коэффициент нагрузки в хеш-таблице?
- 7) Что такое «первичный кластер» в таблице с открытым адресом?
- 8) Как реализуется двойное хеширование?

Решение:

- 1) Расскажи о назначении хеш-функции

Ответ:

Хеш-функция — это математическая функция, которая преобразует входные данные (обычно переменной длины) в фиксированную строку битов определенной длины. Этот результат, называемый хеш-значением или просто хешем, обычно представляет собой уникальное значение, которое определяется данными входа. Назначение хеш-функций разнообразно и включает в себя следующие аспекты:

- **Сохранение данных:** Хеш-функции используются для хранения данных в структурах данных, таких как хеш-таблицы, которые обеспечивают быстрый доступ к данным. Хеш-функция позволяет быстро определить местоположение данных в таблице на основе их ключей.

- **Цифровая подпись:** Хеш-функции используются в криптографии для создания и проверки цифровых подписей. Подпись, созданная с использованием хеш-функции, позволяет убедиться в целостности и аутентичности данных.

- **Проверка целостности данных:** Хеш-функции используются для проверки целостности данных, например, при скачивании файлов из интернета. Полученное хеш-значение файла сравнивается с заранее известным хешем, чтобы убедиться, что файл не был поврежден в процессе передачи.

- **Шифрование паролей:** При хранении паролей в базах данных часто не хранят сами пароли, а только их хеши. Это повышает безопасность, так как даже если база данных подвергается атаке, злоумышленники не могут легко получить исходные пароли.

- **Уникальные идентификаторы:** Хеш-функции используются для создания уникальных идентификаторов для объектов или данных. Например, в многих базах данных для каждой записи создается уникальный хеш-идентификатор.

- **Поиск дубликатов:** Хеш-функции могут использоваться для поиска дубликатов данных, таких как дубликаты файлов на диске.

- **Многие другие приложения:** Хеш-функции также используются в различных алгоритмах и приложениях, включая поиск, криптографию, компьютерную безопасность, компьютерную графику и т. д.

Важно выбирать подходящую хеш-функцию для конкретной задачи, учитывая требования к быстродействию, уникальности хешей и устойчивости к коллизиям (ситуация, когда двум разным входам соответствует один и тот же хеш). Также следует помнить, что хеш-функции

не обратимы, то есть невозможно восстановить исходные данные из хеш-значения.

2) Что такое коллизия?

Ответ:

Коллизия — это ситуация, при которой два разных входных набора данных (например, сообщения, файлы или значения) приводят к одинаковым хеш-значениям при использовании определенной хеш-функции. То есть, двум разным входам соответствует один и тот же хеш-код. Коллизии могут быть нежелательными в большинстве приложений, использующих хеш-функции.

Например:

- **Хеш-таблицы:** В хеш-таблицах используется хеш-функция для быстрого поиска данных по их ключам. Если возникают коллизии, то двум разным ключам может быть назначено одно и то же местоположение в хеш-таблице, что может замедлить доступ к данным.

- **Криптография:** В криптографических приложениях, таких как цифровые подписи или хеширование паролей, коллизии могут повредить безопасность системы, так как злоумышленник может создать данные, которые генерируют тот же хеш, что и оригинальные данные.

- **Проверка целостности данных:** Когда используются хеши для проверки целостности данных (например, при скачивании файлов из интернета), коллизии могут привести к ложным положительным или ложным отрицательным результатам, что может сделать механизм проверки целостности ненадежным.

Для многих приложений важно выбирать хеш-функции с высокой степенью устойчивости к коллизиям, чтобы минимизировать возможность их возникновения. Криптографические хеш-функции, такие как SHA-256 и SHA3, разработаны с учетом этой требовательности и обеспечивают высокий уровень безопасности относительно коллизий. Однако, при разработке и использовании хеш-функций, всегда необходимо учитывать возможность

коллизий и принимать соответствующие меры для их управления или предотвращения.

3) Что такое «открытый адрес» по отношению к хеш-таблице?

Ответ:

"Открытый адрес" — это одна из стратегий разрешения коллизий в хеш-таблицах. Хеш-таблицы используются для эффективного хранения данных и быстрого поиска по ключу. Однако иногда возникают ситуации, когда нескольким разным ключам соответствует одно и то же местоположение в хештаблице (коллизия). "Открытый адрес" представляет собой метод обработки коллизий, при котором новые элементы вставляются в другое доступное место, если первоначальное место, вычисленное с использованием хеш-функции, уже занято.

Суть "открытого адреса" заключается в следующем:

1) При вставке элемента в хеш-таблицу сначала вычисляется хеш ключа.

2) Если место, соответствующее хешу, уже занято другим элементом, то происходит поиск следующего доступного места в таблице, чаще всего по некоторому заданному правилу.

3) Найденное свободное место занимает новый элемент.

4) При поиске элемента в хеш-таблице выполняется та же последовательность операций: сначала вычисляется хеш ключа, а затем проверяются места в соответствии с выбранным правилом поиска.

Существует несколько способов выполнения "открытого адреса," такие как **линейное пробирование** (последовательное сканирование ячеек таблицы), **квадратичное пробирование** (попытка вставить элементы в квадратичных интервалах) и **двойное хеширование** (использование второй хеш-функции для определения новой позиции в случае коллизии).

Преимуществом "открытого адреса" является отсутствие дополнительных структур данных, таких как списки или цепочки, для разрешения коллизий. Однако важно правильно выбирать и настраивать

методы поиска при использовании этой стратегии, чтобы обеспечить эффективное выполнение операций вставки и поиска, особенно при высокой загрузке хеш-таблицы.

4) Как в хеш-таблице с открытым адресом реализуется коллизия?

Ответ:

В хеш-таблице с открытым адресом коллизии решаются путем поиска и замены новых ключей, которые имеют коллизию с уже занятыми местами (так называемое "пробирование"). Вот основные шаги того, как это происходит:

Хеширование ключа: Сначала хеш-функция применяется к ключу, чтобы вычислить хеш-код для вставки или поиска элемента.

Попытка вставки (или поиска): Начиная с места, соответствующего вычисленному хеш-коду, производится попытка вставки элемента или поиска. Если место свободно (т.е., нет коллизии), операция успешно завершается.

Обработка коллизии: Если выбранное место уже занято другим элементом (коллизия), то используется метод "пробирования" для поиска следующего доступного места в таблице. Существует несколько методов пробирования:

- **Линейное пробирование:** При этом методе производится последовательный перебор мест в таблице. Например, если начальное место занято, то вы просто двигаетесь к следующему месту и проверяете его. Этот процесс продолжается до тех пор, пока не будет найдено свободное место.

- **Квадратичное пробирование:** Здесь вы ищете новое место с использованием квадратичной функции вместо линейной. Это может помочь распределить элементы более равномерно по таблице.

- **Двойное хеширование:** Вы используете вторую хеш-функцию для определения следующего места, если первое занято. Это может обеспечить лучшее распределение элементов.

Вставка (или поиск) в новом месте: Как только найдено новое свободное место, элемент вставляется в это место, или, в случае поиска, осуществляется поиск элемента в этом новом месте.

Повторение при необходимости: Если и новое место оказалось занято или в процессе поиска не был найден нужный элемент, то операция пробирется дальше до тех пор, пока не будет найдено подходящее место или не будет достигнут лимит попыток.

Важно настроить метод пробирования и выбрать хорошую хешфункцию для минимизации коллизий и обеспечения эффективного выполнения операций вставки и поиска.

5) Какая проблема, может возникнуть после удаления элемента из хештаблицы с открытым адресом и как ее устранить?

Ответ:

После удаления элемента из хеш-таблицы с открытым адресом может возникнуть проблема с производительностью и правильностью операций поиска. Это связано с тем, что удаление элемента приводит к освобождению места, которое теперь может быть использовано для вставки новых элементов или для последующих операций поиска. Однако удаление элемента может оставить "пробелы" в таблице, и, если они не учитываются, это может привести к неправильным результатам при поиске и, возможно, к ухудшению производительности.

Основные проблемы, которые могут возникнуть после удаления элемента из хеш-таблицы с открытым адресом, включают:

Проблема поиска: После удаления элемента, другие элементы могут быть перемещены, и они могут быть найдены в неправильных местах, если не учтены "пробелы" (освобожденные ячейки).

Производительность: Если "пробелы" остаются в таблице, производительность операций поиска и вставки может ухудшиться, так как элементы будут разбросаны по таблице и поиск потребует больше времени.

Для устранения этих проблем после удаления элемента из хеш-таблицы с открытым адресом можно использовать следующие методы:

Метод удаления флага: Вместо непосредственного удаления элемента из таблицы, вы можете установить специальный флаг (например, "удален") в элементе, чтобы пометить его как удаленный. При поиске элементов вы учитываете этот флаг и пропускаете удаленные элементы. Позднее, при необходимости, можно выполнить очистку удаленных элементов в фоновом режиме.

Метод мягкого удаления: Вместо немедленного удаления элемента, заменяйте его специальным значением (например, "пусто" или "удалено"). Это оставит место в таблице, но обозначит его как пустое. При вставке нового элемента учитывайте такие "пустые" места и, при достижении определенной загрузки таблицы, проводите операцию перехеширования для перераспределения элементов и устранения "пустых" мест.

Агрессивное перехеширование: Регулярно проводите операцию перехеширования для очистки "пустых" мест и перераспределения элементов. Это может быть ресурсоемкой операцией, но позволяет поддерживать высокую производительность таблицы.

Какой метод выбрать, зависит от конкретных требований вашего приложения и хеш-таблицы. Важно также учесть, что частые операции удаления могут сделать хеш-таблицу с открытым адресом менее эффективной, и в некоторых случаях стоит рассмотреть альтернативные структуры данных или методы обработки коллизий.

6) Что определяет коэффициент нагрузки в хеш-таблице?

Ответ:

Коэффициент нагрузки (load factor) в хеш-таблице определяет, насколько заполнена таблица данными в процентах. Он рассчитывается как отношение числа элементов, хранящихся в таблице, к общему числу доступных ячеек в таблице. Формула для вычисления коэффициента нагрузки выглядит следующим образом:

Коэффициент нагрузки = (Количество элементов в таблице) / (Общее количество доступных ячеек в таблице)

Коэффициент нагрузки показывает, насколько "плотно" заполнена хеш-таблица. Он важен, потому что влияет на производительность и эффективность операций вставки, поиска и удаления элементов в таблице.

Важные аспекты, связанные с коэффициентом нагрузки в хеш-таблицах:

Производительность: Если коэффициент нагрузки становится слишком большим (близким к 1), то таблица будет переполнена элементами, что может привести к увеличению времени выполнения операций. В худшем случае, это может вызвать коллизии и ухудшение производительности.

Плотность данных: Коэффициент нагрузки также связан с эффективностью использования памяти. Слишком низкий коэффициент нагрузки означает, что много памяти тратится на пустые ячейки, а слишком высокий коэффициент нагрузки может привести к увеличенному расходу памяти на управление коллизиями.

Управление коллизиями: Коэффициент нагрузки также влияет на вероятность возникновения коллизий, когда нескольким ключам соответствует одно и то же место в таблице. В зависимости от метода разрешения коллизий, высокий коэффициент нагрузки может потребовать дополнительных мер для обработки коллизий.

Обычно коэффициент нагрузки поддерживается на относительно низком уровне (обычно меньше 0.7 или даже меньше), чтобы обеспечить хорошую производительность и уменьшить вероятность коллизий. Если коэффициент нагрузки становится слишком высоким, рекомендуется увеличить размер таблицы (рехешировать) или использовать другие методы управления коллизиями, чтобы поддерживать надежное и эффективное функционирование хеш-таблицы.

7) Что такое «первичный кластер» в таблице с открытым адресом?

Ответ:

В хеш-таблицах с открытым адресом, термин "первичный кластер" относится к ситуации, когда несколько элементов имеют одинаковый хеш-код и находятся рядом друг с другом в таблице. Первичный кластер может возникнуть в результате коллизии, когда два или более элемента пытаются встаться в одну и ту же ячейку, и один из них уже занимает это место. Как только первичный кластер возникает, он может привести к дополнительным коллизиям и ухудшению производительности.

Проблемы, связанные с первичными кластерами в хеш-таблицах с открытым адресом, включают:

Повторные коллизии: Если внутри первичного кластера другие элементы имеют те же хеш-коды, они будут искать свободное место внутри кластера. Это может привести к новым коллизиям внутри кластера и дополнительным попыткам пробирования.

Увеличение времени доступа: Чем больше первичный кластер, тем больше времени потребуется для поиска свободной ячейки и вставки элемента. Это ухудшает производительность операций вставки.

Увеличение вероятности коллизий: Поскольку элементы в первичном кластере имеют одинаковые хеш-коды, вероятность возникновения дополнительных коллизий в будущем также увеличивается.

Для уменьшения негативного воздействия первичных кластеров в хеш-таблицах с открытым адресом можно использовать методы разрешения коллизий, такие как двойное хеширование, квадратичное пробирование или линейное пробирование, которые помогут более равномерно распределить элементы и уменьшить вероятность образования первичных кластеров. Также важно правильно выбирать размер таблицы и коэффициент нагрузки, чтобы минимизировать вероятность возникновения первичных кластеров и обеспечить эффективную работу хеш-таблицы.

8) Как реализуется двойное хеширование?

Ответ:

Двойное хеширование (Double Hashing) — это метод разрешения коллизий в хеш-таблицах с открытым адресом. Этот метод позволяет находить новые места для элементов, которые имеют коллизии, путем применения двух хеш-функций вместо одной. Вот как это работает:

1) **Исходная хеш-функция:** Сначала применяется исходная хеш-функция к ключу элемента для определения начальной позиции (индекса) в хештаблице.

2) **Первая хеш-функция:** Если начальная позиция уже занята другим элементом (коллизия), то вместо простого перехода к следующей ячейке, используется первая хеш-функция для вычисления смещения от начальной позиции. Это смещение может быть, например, равно $h1(key)$.

3) **Попытка вставки или поиска:** Элемент вставляется в новое место (начальная позиция + смещение), и, если это место также занято, применяется вторая хеш-функция для вычисления дополнительного смещения. Вторая хеш-функция может быть, например, $h2(key)$.

4) **Повторение при необходимости:** Если и новое место также занято, повторяются операции смещения, используя обе хеш-функции, пока не будет найдено свободное место или пока не будет достигнут предел попыток.

Преимущество двойного хеширования заключается в том, что он может равномерно распределять элементы по таблице и уменьшать вероятность образования "первичных кластеров" (групп элементов с одинаковым хешкодом). Это помогает улучшить производительность хеш-таблицы и уменьшить вероятность коллизий.

Важно правильно выбрать и настроить хеш-функции для двойного хеширования, чтобы обеспечить хорошую равномерность распределения элементов. Выбор хороших хеш-функций и правильная настройка параметров (например, размера таблицы и коэффициента нагрузки) важны для эффективного использования метода двойного хеширования.

Задание 2 (Вариант 20)

Формулировка задачи:

Разработать приложение, которое использует хеш-таблицу для организации прямого доступа к записям двоичного файла. Метод разрешения коллизии – **открытый код (сдвиг на 1)**.

Для обеспечения прямого доступа к записи в файле элемент хештаблицы должен включать обязательные поля: ключ записи в файле, номер записи с этим ключом в файле. Элемент может содержать другие поля, требующиеся методу (указанному в вашем варианте), разрешающему коллизию.

Управление хеш-таблицей.

– Определить структуру элемента хеш-таблицы и структуру хеш-таблицы в соответствии с методом разрешения коллизии, указанным в варианте.

– Разработать хеш-функцию (метод определить самостоятельно), выполнить ее тестирование, убедиться, что хеш (индекс элемента таблицы) формируется верно.

– Разработать операции: вставить ключ в таблицу, удалить ключ из таблицы, найти ключ в таблице, рехешировать таблицу. Каждую операцию тестируйте по мере ее реализации.

– Подготовить тесты (последовательность значений ключей), обеспечивающие:

- вставку ключа без коллизии
- вставку ключа и разрешение коллизии
- вставку ключа с последующим рехешированием
- удаление ключа из таблицы
- поиск ключа в таблице

– Выполнить тестирование операций управления хеш-таблицей. При тестировании операции вставки ключа в таблицу предусмотрите вывод списка индексов, которые формируются при вставке элементов в таблицу.

```

uint64_t hash(std::string val)
{
    uint64_t sum = (uint64_t) 'S';
    for (size_t i = 0; i < val.size(); i++)
        sum += (int)val[i];

    uint64_t val1 = (uint64_t) 'C';
    for (size_t i = 0; i < val.size(); i++)
        val1 += (uint64_t)val[i] << ((i * 5) % 64);

    uint64_t val2 = ~(val1 << 32) + (val1 >> 32);
    val2 %= ~(val1) >> 32;
    return ((val1 / val2) << 16) + ((val2 / val1) << 32) + (val2 % (val1 << 16))
+ sum;
}

```

```

template<typename T>
class map
{
private:

    class cell
    {
    public:
        T val;
        std::string key;

        cell()
        {
            this->key = "\0";
        }
    };

    size_t _capacity;
    size_t _occupied;

    std::vector<cell> _table;

    uint16_t _get_percentage()
    {
        return this->_occupied * 100 / this->_capacity;
    }

    void _extend(size_t size)
    {
        this->_capacity = size;
        this->_occupied = 0;
        std::vector<cell> old_table = this->_table;
        this->_table.clear();
        this->_table = std::vector<cell>(this->_capacity);

        for (cell _cell : old_table)
            if (_cell.key != "\0" and _cell.key != "\1")
                this->insert(_cell.key, _cell.val);
    }

    void _check_capacity()
    {
        if (_get_percentage() > 75)
            this->_extend(this->_capacity * 2);
    }
}

```

```

size_t _get_index(std::string key)
{
    return hash(key) % this->_capacity;
}

public:
map()
{
    this->_capacity = 16;
    this->_occupied = 0;
    this->_table = std::vector<cell>(this->_capacity);
}

void remove(std::string key)
{
    size_t offset = 0;
    size_t index = this->_get_index(key);
    cell *tmp = &this->_table[index];
    for (; tmp->key != key and tmp->key != "\0" and index + offset + 1 <
this->_capacity; tmp = &this->_table[index + ++offset]) {}
    if (tmp->key == "\0" or index + offset >= this->_capacity)
        return;
    tmp->val = T();
    tmp->key = "\1";
    this->_occupied--;
}

void insert(std::string key, T val)
{
    size_t offset = 0;
    size_t index = this->_get_index(key);
    cell* tmp = &this->_table[index];
    for (; tmp->key != "\0" and tmp->key != "\1" and index + offset + 1 <
this->_capacity; tmp = &this->_table[index + ++offset]) {}

    if (index + offset + 1 >= this->_capacity)
    {
        this->_extend(this->_capacity * 2);
        this->insert(key, val);
        return;
    }

    tmp->val = val;
    tmp->key = key;

    this->_occupied++;
    this->_check_capacity();
}

T& operator[](std::string key)
{
    size_t offset = 0;
    size_t index = this->_get_index(key);
    cell* tmp = &this->_table[index];
    for (; tmp->key != key and tmp->key != "\0" and index + offset + 1 <
this->_capacity; tmp = &this->_table[index + ++offset]) {}
    if (tmp->key == "\0" or index + offset >= this->_capacity)
        throw std::invalid_argument("Key not found.");
    return tmp->val;
}
};

```

```

struct goodSales
{
    char id[10] = { 0 };
    char name[15] = { 0 };
    size_t price;
    char sale_date[11] = { 0 };
};

int hash_table_one()
{
    time_t s = time(0);
    srand(s);

    map<goodSales> data;
    std::set<std::string> ids;

    std::vector<goodSales> entries;

    size_t size = io.input<int>("Enter amount: ");

    goodSales entry;

    for (size_t i = 0; i < size; i++)
    {
        strcpy_s(entry.name, ("Good_" + std::to_string(i)).c_str());
        strcpy_s(entry.sale_date, "11.09.2001");
        entry.price = rand();
        std::string id;

        do
        {
            id = "";
            for (size_t j = 0; j < 6; j++)
            {
                int num = rand() % 36;
                id += (char)(num < 10 ? num + '0' : num - 10 + 'A');
            }
        } while (ids.count(id) > 0);

        strcpy_s(entry.id, id.c_str());

        entries.push_back(entry);

        ids.insert(id);
    }

    measure(
        for (auto& val : entries)
            data.insert(val.id, val);
        , "Table is ready.\nElapsed time: ");

    measure(
        for (auto key : ids)
        {
            io.output(data[key].name);
            data.remove(key);
            break;
        }, "Element is found.\nElapsed time: ");

    return 0;
}

```

Листинг 1 – Код задачи 2

```
Enter amount: 1000
Elapsed time: 33982 microseconds
```

Рисунок 3 – результат тестирования записи для N=1000

```
Enter amount: 10000
Elapsed time: 151269 microseconds
```

Рисунок 4 – результат тестирования записи для N=10000

```
Enter amount: 100000
Elapsed time: 24569918 microseconds
```

Рисунок 5 – результат тестирования записи для N=100000

Таблица тестирований (таблица 1)

N	Время (мкс)
1000	33 982
10 000	151 269
100 000	24 569 918


```
Enter amount: 1000
Good_755
Elapsed time: 354 microseconds
```

Рисунок 6 – результат тестирования поиска для N=1000

```
Enter amount: 10000
Good_2343
Elapsed time: 732 microseconds
```

Рисунок 7 – результат тестирования поиска для N=10000

```
Enter amount: 100000
Good_39377
Elapsed time: 680 microseconds
```

Рисунок 8 – результат тестирования поиска для N=100000

Таблица тестирований (таблица 2)

N	Время (мкс)
1000	354
10 000	732
100 000	680

Задание 3 (Вариант 20)

Формулировка задачи:

Управление бинарным файлом посредством хеш-таблицы.

В заголовочный файл подключить заголовочные файлы: управления хештаблицей, управления двоичным файлом. Реализовать поочередно все перечисленные ниже операции в этом заголовочном файле, выполняя их тестирование из функции main приложения. После разработки всех операций выполнить их комплексное тестирование (программы (все базовые операции, изменение размера и рехеширование), тест-примеры определите самостоятельно. Результаты тестирования включите в отчет по выполненной работе).

Разработать и реализовать операции.

1) Прочитать запись из файла и вставить элемент в таблицу (элемент включает: ключ и номер записи с этим ключом в файле, и для метода с открытой адресацией возможны дополнительные поля).

2) Удалить запись из таблицы при заданном значении ключа и соответственно из файла.

3) Найти запись в файле по значению ключа (найти ключ в хеш-таблице, получить номер записи с этим ключом в файле, выполнить прямой доступ к записи по ее номеру).

4) Подготовить тесты для тестирования приложения:

– *Заполните файл небольшим количеством записей.*

- Включите в файл записи как не приводящие к коллизиям, так и приводящие.

- Обеспечьте включение в файл такого количества записей, чтобы потребовалось рехеширование.

– *Заполните файл большим количеством записей (до 1 000 000).*

Определите время чтения записи с заданным ключом: для первой записи файла, для последней и где-то в середине. Убедитесь (или нет), что время доступа для всех записей одинаково.

```

struct goodSales
{
    char id[10] = { 0 };
    char name[15] = { 0 };
    size_t price;
    char sale_date[11] = { 0 };
};

struct fileKey
{
    char id[10] = { 0 };
    size_t offset;
};

int hash_table_two()
{
    FIO bin_fio("", "hash_data.bin", "bin");
    time_t s = time(0);
    srand(s);

    map<fileKey> data;
    std::set<std::string> ids;
    goodSales entry;

    size_t size = io.input<int>("Enter amount: ");

    for (size_t i = 0; i < size; i++)
    {
        strcpy_s(entry.name, ("Good_" + std::to_string(i)).c_str());
        strcpy_s(entry.sale_date, "11.09.2001");
        entry.price = rand();
        std::string id;

        do
        {
            id = "";
            for (size_t j = 0; j < 6; j++)
            {
                int num = rand() % 36;
                id += (char)(num < 10 ? num + '0' : num - 10 + 'A');
            }
        } while (ids.count(id) > 0);

        strcpy_s(entry.id, id.c_str());

        bin_fio.write(reinterpret_cast<const char*>(&entry),
sizeof(entry));

        ids.insert(id);
    }

    io.output("Bin file is ready.");

    bin_fio.close_outf();
    bin_fio.set_out("");
    bin_fio.set_in("hash_data.bin");
    bin_fio.open_inf();

    size_t offset = 0;
    fileKey entry_pair;

    while (bin_fio.read(reinterpret_cast<char*>(&entry), sizeof(entry)))
    {

```

```

        entry_pair.offset = offset++;
        strcpy_s(entry_pair.id, entry.id);
        data.insert(entry_pair.id, entry_pair);
    }

    io.output("Table is ready.");

    bin_fio.close_inf();
    bin_fio.open_inf();

    measure(
        for (auto key : ids)
        {
            bin_fio.move(data[key].offset * sizeof(entry));
            bin_fio.read(reinterpret_cast<char*>(&entry),
sizeof(entry));

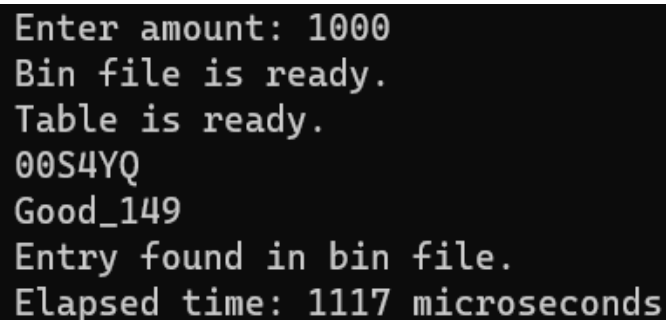
            io.output(entry.id);
            io.output(entry.name);
            break;
        }, "Entry found in bin file.\nElapsed time: "
        );

    bin_fio.close();

    return 0;
}

```

Листинг 5 – код задачи 3

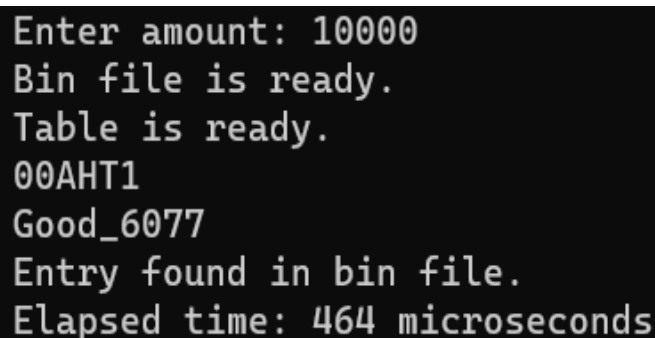


```

Enter amount: 1000
Bin file is ready.
Table is ready.
00S4YQ
Good_149
Entry found in bin file.
Elapsed time: 1117 microseconds

```

Рисунок 6 – результат тестирования для N=100



```

Enter amount: 10000
Bin file is ready.
Table is ready.
00AHT1
Good_6077
Entry found in bin file.
Elapsed time: 464 microseconds

```

Рисунок 7 – результат тестирования для N=1000

```
Enter amount: 100000
Bin file is ready.
Table is ready.
000UF6
Good_46614
Entry found in bin file.
Elapsed time: 860 microseconds
```

Рисунок 8 – результат тестирования для N=10 000

Таблица тестирований (таблица 2)

N	Время (мкс)
1000	1117
10 000	464
100 000	860

Вывод

Были получены навыки по разработке хеш-таблиц и их применении при поиске данных в других структурах данных (файлах).