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 «Применение хеш-таблицы для поиска данных в двоичном файле с записями фиксированной длины»

**Выполнил:**
Студент группы ИКБО-13-22 Руденко А.Д.

**Проверил:** ассистент Муравьёва Е.А.

МОСКВА 2023 г.

# 1. ВВЕДЕНИЕ

**Цель**: получить навыки по разработке хеш-таблиц и их применении при поиске данных в других структурах данных (файлах).

## 1.1. Задание 1

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 хеш-таблицы 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?

## 1.2. Задание 2 (Вариант 26)

Разработать приложение main, которое использует хеш-таблицу для организации прямого доступа к записям двоичного файла. Метод разрешения коллизии – **цепное хеширование**.

Для обеспечения прямого доступа к записи в файле элемент хеш-таблицы 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, указанном 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.

### 1.3. Задание 3

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.

# 2. ХОД РАБОТЫ

## 2.1. Задание 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.

**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 SHA-3, 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.

**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 хеш-таблице (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 занимается новым элементом.
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.

**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 пробируется дальше до тех пор, пока не будет найдено подходящее место или не будет достигнут лимит попыток.

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

**Какая проблема, может возникнуть после удаления элемента из хеш-таблицы 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.

**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.

**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 вставиться 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.

**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 хеш-таблице.
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 хеш-кодом). 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.

## 2.2. Задание 2

### 2.2.1. Описание алгоритмов

*Листинг 1. Вставка ключа в таблицу*

```
void Insert(unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long int
key, int tableSize)
{
    HashTableEntry entry;
    entry.key = key;
    entry.hash = HashFun(key, tableSize);
    hashTable[entry.hash].push_back(entry);
}
```

Алгоритм выполняет следующие действия:

Создается новый экземпляр структуры, который содержит ключ и хеш-значение. На основе вычисленного хеш-значения, элемент добавляется в соответствующий список. Это обеспечивает разрешение коллизий методом цепочек, где элементы с одинаковыми хеш-значениями добавляются в один список.

Этот алгоритм предназначен для вставки элемента в хеш-таблицу с учетом коллизий, что делает его более эффективным и надежным для хранения данных с возможными коллизиями хеш-значений.

*Листинг 2. Удаление ключа из таблицы*

```
void Remove(unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long int
key, int tableSize)
{
    int hash = HashFun(key, tableSize);
    auto& entries = hashTable[hash];//----> Получение ссылки на список (цепь
записей)
    for (auto it = entries.begin(); it != entries.end(); ++it)
    {
        if (it->key == key)
        {
            entries.erase(it);
            break;//----> Выход из цикла после удаления ключа
        }
    }
}
```

Алгоритм выполняет следующие действия:

– Вычисляет хеш-значение для ключа с использованием функции. Это значение определяет, в какой "ячейке" хеш-таблицы находится элемент.

– Получает ссылку на список (цепь записей), который находится по индексу в хэш-таблице. Этот список содержит все элементы, которые имеют одинаковое хеш-значение.
– Проходится по списку, используя итератор.
– Для каждого элемента, проверяет, совпадает ли ключ элемента с ключом, который нужно удалить.
– Если ключи совпадают, то удаляет элемент из списка с помощью ***erase()*** и завершает цикл. Это обеспечивает удаление только одного элемента с заданным ключом (первого найденного), чтобы не удалять дополнительные элементы с тем же ключом, если они есть в цепочке записей.

В результате выполнения этого алгоритма, элемент с заданным ключом будет удален из хеш-таблицы, и это позволяет обрабатывать коллизии методом цепочек, сохраняя все элементы с одинаковыми хеш-значениями в списке.

*Листинг 3. Поиск ключа в таблице*

```
bool Find(const unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long
int key, int tableSize)
{
    int hash = HashFun(key, tableSize);
    auto& entries = hashTable.at(hash);//----> Получение ссылки на список (цепь
записей)
    for (const HashTableEntry& entry : entries)
    {
        if (entry.key == key)
        {
            return true;//----> Ключ найден
        }
    }
    return false;//----> Ключ не найден
}
```

Алгоритм выполняет следующие действия:

– Вычисляет хеш-значение для ключа с использованием функции. Это значение определяет, в какой "ячейке" хеш-таблицы находится элемент.
– Получает ссылку на список (цепь записей), который находится по индексу в хеш-таблице. Этот список содержит все элементы, которые имеют одинаковое хеш-значение.

– Проходится по списку, используя цикл ***for***.
– Для каждого элемента в списке, проверяет, совпадает ли ключ элемента с ключом, который нужно найти.
– Если ключи совпадают, возвращает, что означает, что ключ был найден.
– Если цикл завершается, и ключ не был найден, возвращается, что означает, что ключ не найден в хеш-таблице.

Этот алгоритм предназначен для поиска элемента в хеш-таблице, используя метод цепочек для разрешения коллизий. Он проверяет все элементы в цепочке записей, чтобы найти элемент с заданным ключом и возвращает ***true***, если такой элемент найден, и ***false***, если не найден.

*Листинг 4. Рехеширование таблицы*

```
void Rehash(unordered_map<int, list<HashTableEntry>>& hashTable, int newTableSize)
{
    unordered_map<int, list<HashTableEntry>> newHashTable(newTableSize);
    for (const auto& entry : hashTable)//----> Проход по всем элементам хеш-
таблицы
    {
        for (const HashTableEntry& hashEntry : entry.second)//----> Проход по всем
записям хеша
        {
            int newHash = HashFun(hashEntry.key, newTableSize);
            newHashTable[newHash].push_back(hashEntry);
        }
    }
    hashTable = newHashTable;
}
```

Алгоритм выполняет следующие действия:

– Создает новый пустой контейнер с новым именем и размером. Этот новый контейнер будет использоваться для хранения элементов после рехеширования.
– Проходится по всем элементам исходной хеш-таблицы, используя цикл ***for***.
– Во внутреннем цикле ***for***, проходится по всем записям в цепи записей (списке) элемента в хеш-таблице.
– Для каждой записи вычисляется новое хеш-значение для ключа, используя функцию и новый размер.

– Затем запись добавляется в новую хеш-таблицу в соответствии с новым хеш-значением.
– После завершения всех итераций, исходная хеш-таблица переприсваивается значению новой хеш-таблицы. Теперь хеш-таблица будет представлять хеш-таблицу с новым размером и перераспределенными элементами.

Этот алгоритм выполняет рехеширование с сохранением всех существующих элементов и изменением размера хеш-таблицы с учетом нового размера. Это может быть полезно для управления производительностью и распределением элементов в таблице.

### 2.2.2. Код программы

*Листинг 5.*

```
#include <unordered_map>// Непорядковое отображение (коллекция пар: ключ-значение)
#include <iostream>
#include <fstream>
#include <format>
#include <string>
#include <vector>
#include <list>
#include <set>

using namespace std;


//Структура записи о пациенте: номер полиса, фамилия, имя, отчество, код
заболевания, дата установки диагноза, код врача.
struct InsuranceNote {
    unsigned long int number = 0;
    string surname;
    string first_name;
    string mid_name;
    unsigned long int diseaseCode = 0;
    string date;
    unsigned long int doctorCode = 0;
};

//Структура записи в хеш-таблице
struct HashTableEntry {
    unsigned long int key;
    int hash;
};

//Номер полиса, состоящий из 8 чисел
unsigned long int Randomizer() {
    unsigned long int min = 10000000;
    unsigned long int max = 99999999;
    return min + rand() % (max - min + 1);
}
```

```
//Хеш-функция методом деления
int HashFun(unsigned long int key, int tableSize) {
    return key % tableSize;
}

//Вставка ключа в таблицу
void Insert(unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long int
key, int tableSize) {
    HashTableEntry entry;
    entry.key = key;
    entry.hash = HashFun(key, tableSize);
    hashTable[entry.hash].push_back(entry);
}

//Удаление ключа из таблицы
void Remove(unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long int
key, int tableSize) {
    int hash = HashFun(key, tableSize);
    auto& entries = hashTable[hash];// Получение ссылки на список (цепь записей)
    for (auto it = entries.begin(); it != entries.end(); ++it) {
        if (it->key == key) {
            entries.erase(it);
            break;// Выход из цикла после удаления ключа
        }
    }
}

//Поиск ключа в таблице
bool Find(const unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long
int key, int tableSize) {
    int hash = HashFun(key, tableSize);
    auto& entries = hashTable.at(hash); // Получение ссылки на список (цепь
записей)
    for (const HashTableEntry& entry : entries) {
        if (entry.key == key) {
            return true;// Ключ найден
        }
    }
    return false;// Ключ не найден
}

//Рехеширование таблицы
void Rehash(unordered_map<int, list<HashTableEntry>>& hashTable, int newTableSize)
{
    unordered_map<int, list<HashTableEntry>> newHashTable(newTableSize);
    for (const auto& entry : hashTable) { // Проход по всем элементам хеш-таблицы
        for (const HashTableEntry& hashEntry : entry.second) { // Проход по всем
записям хеша
            int newHash = HashFun(hashEntry.key, newTableSize);
            newHashTable[newHash].push_back(hashEntry);
        }
    }
    hashTable = newHashTable;
}

//Определение простого числа
bool IsPrime(int n) {
    if (n <= 1) {
        return false;
    }
    if (n <= 3) {
        return true;
    }
    if (n % 2 == 0 || n % 3 == 0) {
```

```
        return false;
    }
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}

int CalculateSize(int currentSize) {
    //Умножение текущего размера на некоторый множитель
    int newSize = currentSize * 2;

    //Проверка на простое число, чтобы уменьшить вероятность коллизий
    while (!IsPrime(newSize)) {
        newSize++;
    }
    return newSize;
}

void Output(unordered_map<int, list<HashTableEntry>> hashTable) {
    //Вывод списка хеш-кода и ключа
    cout << endl << "После изменения" << endl;
    for (const auto& entry : hashTable) {
        int hash = entry.first;// Получение хеш-кода
        const list<HashTableEntry>& entries = entry.second;// Получение списка
записей
        for (const HashTableEntry& hashEntry : entries) {
            unsigned long int key = hashEntry.key;// Получение ключа
            printf("   Ключ: %d   Хеш: %d\n", key, hash);
        }
    }
}

int main() {
    // ЗАДАНИЕ 3.2

    setlocale(LC_ALL, "ru");
    srand(static_cast<unsigned int>(time(nullptr)));// Инициализация генератора

    ofstream textFile("textFile.txt");// Текстовый файл
    if (!textFile) {
        cout << "Текстовый файл не удалось открыть для записи" << endl;
        return 1;
    }
    cout << "Текстовый файл успешно открыт для записи" << endl << endl;

    // Открываем файл для записи в бинарном режиме (флаг - ios::binary)
    ofstream binaryFileOut("binaryFile.bin", ios::binary);// Бинарный файл
    if (!binaryFileOut) {
        cerr << "Бинарный файл не удалось открыть для записи" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для записи" << endl << endl;

    int quant = 7;// Количество записей
    set<int> unique;

    InsuranceNote insNote;
    for (int i = 0; i < quant; ++i) {
        do {
            insNote.number = Randomizer();
        } while (unique.count(insNote.number) > 0);
```

```
        unique.insert(insNote.number);
        insNote.surname = "Фамилия_" + to_string(i);
        insNote.first_name = "Имя_" + to_string(i);
        insNote.mid_name = "Отчество_" + to_string(i);
        insNote.date = (to_string(Randomizer() % 32 + 1) + "." +
to_string(Randomizer() % 13 + 1) + "." + to_string(Randomizer() % 23 + 2000));
        insNote.diseaseCode = Randomizer() % 10 + 100;
        insNote.doctorCode = Randomizer() % 101000 + 100000;

        textFile << insNote.number << ' ' << insNote.surname << ' ' <<
insNote.first_name << ' ' << insNote.mid_name << ' ' << insNote.date << ' ' <<
insNote.diseaseCode << ' ' << insNote.doctorCode << '\n';

        binaryFileOut.write(reinterpret_cast<const char*>(&insNote),
sizeof(insNote));
    }
    cout << "Генерация текстового и бинарного файла завершена" << endl;

    size_t bytes = sizeof(InsuranceNote);
    cout << "Размер записи в байтах: " << bytes << endl;
    cout << "Количество записей: " << quant << endl << endl;

    textFile.close();
    binaryFileOut.close();

    const int tableSize = 7;// Размер хеш-таблицы
    unordered_map<int, list<HashTableEntry>> hashTable;// Создание хеш-таблицы с
цепочками

    // Добавляем элементы в хеш-таблицу
    ifstream binaryFileIn("binaryFile.bin", ios::binary);
    if (!binaryFileIn) {
        cout << "Бинарный файл не удалось открыть для чтения" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для чтения" << endl << endl;

    cout << "   Проверка хеш-индексов" << endl << endl;
    while (binaryFileIn.read(reinterpret_cast<char*>(&insNote), sizeof(insNote)))
{
        // Создание записи для хеш-таблицы
        HashTableEntry entry;
        entry.key = insNote.number;
        entry.hash = HashFun(entry.key, tableSize);// Вычисление хеш-кода для
ключа

        hashTable[entry.hash].push_back(entry);// Добавление записи в цепочку хеш-
таблицы

        printf("   Ключ: %d   Хеш: %d\n", entry.key, entry.hash);
    }

    binaryFileIn.close();

    // Фактор заполнения
    double loadFactor = static_cast<double>(hashTable.size()) / tableSize;

    bool flag = true;
    while (flag) {
        int choice;
        cout << endl << " Выберите действие со структурой:\n [1] - вставка ключа\n
[2] - поиск ключа\n [3] - удаление ключа\n [4] - выход из тестирования\n";
        cin >> choice;
        switch (choice) {
        case 1:
```

```
            unsigned long int insert;
            cout << "Введите номер полиса, который вы желаете добавить (8 цифр):
";
            cin >> insert;
            Insert(hashTable, insert, tableSize);
            Output(hashTable);
            if (loadFactor > 0.8)
            {
                int newTableSize = CalculateSize(tableSize); // Определение нового
размера таблицы
                Rehash(hashTable, newTableSize);
            }
            break;
        case 2:
            unsigned long int found;
            cout << "Введите номер полиса, который вы желаете найти (8 цифр): ";
            cin >> found;
            if (Find(hashTable, found, tableSize))
                cout << "Страховой полис найден!" << endl;
            else
                cout << "Страховой полис не найден!" << endl;
            break;
        case 3:
            unsigned long int remove;
            cout << "Введите номер полиса, который вы желаете удалить (8 цифр): ";
            cin >> remove;
            Remove(hashTable, remove, tableSize);
            Output(hashTable);
            break;
        case 4:
            flag = false;
            break;
        default:
            break;
        }
    }
    return 0;
}
```

### 2.2.3. Тестирование

```
Текстовый файл успешно открыт для записи

Бинарный файл успешно открыт для записи

Генерация текстового и бинарного файла завершена
Размер записи в байтах: 184
Количество записей: 7

Бинарный файл успешно открыт для чтения

   Проверка хеш-индексов

   Ключ: 10001269   Хеш: 5
   Ключ: 10002583   Хеш: 3
   Ключ: 10014491   Хеш: 4
   Ключ: 10016347   Хеш: 5
   Ключ: 10002022   Хеш: 2
   Ключ: 10030054   Хеш: 6
   Ключ: 10000350   Хеш: 3

 Выберите действие со структурой:
 [1] - вставка ключа
 [2] - поиск ключа
 [3] - удаление ключа
 [4] - выход из тестирования
2
Введите номер полиса, который вы желаете найти (8 цифр): 10000350
Страховой полис найден!

 Выберите действие со структурой:
 [1] - вставка ключа
 [2] - поиск ключа
 [3] - удаление ключа
 [4] - выход из тестирования
2
Введите номер полиса, который вы желаете найти (8 цифр): 1
```

```
Количество записей: 7

Бинарный файл успешно открыт для чтения

   Проверка хеш-индексов

   Ключ: 10001632   Хеш: 4
   Ключ: 10020385   Хеш: 4
   Ключ: 10014306   Хеш: 1
   Ключ: 10022948   Хеш: 5
   Ключ: 10017320   Хеш: 5
   Ключ: 10002710   Хеш: 4
   Ключ: 10031930   Хеш: 6

 Выберите действие со структурой:
 [1] - вставка ключа
 [2] - поиск ключа
 [3] - удаление ключа
 [4] - выход из тестирования
1
Введите номер полиса, который вы желаете добавить (8 цифр): 11111111

После изменения
   Ключ: 10001632   Хеш: 4
   Ключ: 10020385   Хеш: 4
   Ключ: 10002710   Хеш: 4
   Ключ: 11111111   Хеш: 4
   Ключ: 10014306   Хеш: 1
   Ключ: 10022948   Хеш: 5
   Ключ: 10017320   Хеш: 5
   Ключ: 10031930   Хеш: 6

 Выберите действие со структурой:
 [1] - вставка ключа
 [2] - поиск ключа
 [3] - удаление ключа
 [4] - выход из тестирования
3
Введите номер полиса, который вы желаете удалить (8 цифр): 11111111

После изменения
   Ключ: 10001632   Хеш: 4
   Ключ: 10020385   Хеш: 4
   Ключ: 10002710   Хеш: 4
   Ключ: 10014306   Хеш: 1
   Ключ: 10022948   Хеш: 5
   Ключ: 10017320   Хеш: 5
   Ключ: 10031930   Хеш: 6

 Выберите действие со структурой:
 [1] - вставка ключа
 [2] - поиск ключа
 [3] - удаление ключа
 [4] - выход из тестирования
```

## 2.3. Задание 3

### 2.3.1. Описание алгоритмов

*Листинг 6. Удаление ключа из таблицы и файла*

```
void Remove(unordered_map<int, list<HashTableEntry>>& hashTable, InsurancePolicy&
policy, const string binary, unsigned long int key, int tableSize)
{
    int hash = HashFun(key, tableSize);
    bool corr = false;

    auto& entries = hashTable[hash];// Получение ссылки на список (цепь записей)

    list<HashTableEntry> tempEntries;// Временная цепочка

    for (auto it = entries.begin(); it != entries.end(); ++it)
    {
        if (it->key != key)
        {
            tempEntries.push_back(*it);// Удаление из хеш-таблицы
        }
        else if (it->key == key)
        {
            // Удаление из файла
            corr = true;

            cout << endl << "Начался процесс удаления записи из файла" << endl;
            ofstream file(binary, ios::binary | ios::in | ios::out);
            if (file)
            {
                file.seekp(it->index * sizeof(InsurancePolicy), ios::beg);

                // Пустая запись
                InsurancePolicy empty;
                empty.number = 0;
                empty.company = "";
                empty.surname = "";
                file.write(reinterpret_cast<char*>(&empty),
sizeof(InsurancePolicy));
                cout << "   Полученная запись удалена из файла" << endl;
            }
            file.close();
        }
    }
    entries = tempEntries;
    if (corr == false)
        cout << endl << "Полис не удалось удалить из файла " << endl;
}
```

Алгоритм выполняет следующие действия:

– Вычисляется хэш ключа с использованием функции метода деления. Результат хэша используется для определения в каком списке (цепи записей) хранятся элементы с данным хэшем.

- Создается флаг, инициализированный значением ***false***. Этот флаг будет использоваться для определения, был ли найден и удален элемент с заданным ключом из файла.
- Получается ссылка на список (цепь записей), связанный с найденным хэшем.
- Создается временный список, который будет использоваться для временного хранения элементов, которые не должны быть удалены из хэш-таблицы.
- Запускается цикл ***for***, который перебирает элементы внутри списка.
- Внутри цикла проверяется, если ключ текущего элемента не совпадает с заданным ключом. Если это условие выполняется, то элемент добавляется во временный список, что фактически приводит к его удалению из хэш-таблицы.
- Если ключ текущего элемента совпадает с заданным ключом, то устанавливается флаг в ***true***. Затем выполняется процесс удаления элемента из файла, используя информацию о расположении элемента в файле. Этот процесс включает в себя открытие файла, перемещение указателя в позицию элемента и запись пустой записи (***InsurancePolicy***) в файл для замещения существующей записи.
- После завершения цикла, список перезаписывается содержимым временного списка, что обновляет хэш-таблицу без удаленного элемента.
- Если флаг остается ***false***, это означает, что элемент не был найден в хэш-таблице и, следовательно, не мог быть удален из файла. В этом случае выводится сообщение, указывающее, что полис не удалось удалить из файла.

Общая цель этой функции - удалить элемент с заданным ключом из хэш-таблицы и соответствующего файла, если элемент с таким ключом существует.

*Листинг 7. Поиск ключа в таблице и файле*

```
void Find(const unordered_map<int, list<HashTableEntry>>& hashTable,
InsurancePolicy& policy, const string& binary, unsigned long int key, int
tableSize)
{
    int hash = HashFun(key, tableSize);
    bool corr = false;

    // Проверка существования ключа в таблице
    auto it = hashTable.find(hash);
    if (it != hashTable.end())
    {
        // Получение ссылки на цепь записей
        auto& entries = it->second;

        for (const HashTableEntry& entry : entries)
        {
            if (entry.key == key)
            {
                cout << endl << "Страховой полис найден в таблице" << endl;
                ifstream file(binary, ios::binary);
                if (file)
                {
                    cout << "   Получаем запись из файла" << endl;
                    corr = true;

                    file.seekg(entry.index * sizeof(InsurancePolicy), ios::beg);
                    if (file.read(reinterpret_cast<char*>(&policy),
sizeof(InsurancePolicy)))
                    {
                        cout << "   Номер полиса ---> " << policy.number << endl;
                        cout << "   Компания ---> " << policy.company << endl;
                        cout << "   Фамилия владельца ---> " << policy.surname <<
endl;
                    }
                }
                file.close();
                return;
            }
            else
                corr = false;
        }
    }
    if (corr == false)
        cout << endl << "Страховой полис не найден в файле" << endl;
}
```

Алгоритм выполняет следующие действия:

– Вычисляется хэш ключа с использованием функции метода деления. Результат хэша используется для определения в каком списке (цепи записей) хранятся элементы с данным хэшем.

– Создается флаг, инициализированный значением ***false***. Этот флаг будет использоваться для определения, был ли найден и извлечен элемент с заданным ключом из файла.

– Производится проверка существования хэша с использованием метода ***find()***. Результат поиска сохраняется в итераторе. Если хэш существует в хэш-таблице, то выполняется следующее:
– Получается ссылка на список (цепь записей), связанный с найденным хэшем.
– Запускается цикл ***for***, который перебирает элементы внутри списка entries.
– Внутри цикла проверяется, если ключ текущего элемента совпадает с заданным ключом. Если это условие выполняется, то страховой полис найден в таблице, и выполняется следующее:
– Открывается бинарный файл с именем ***binary*** в режиме чтения. Если файл успешно открыт, выполняется чтение записи, начиная с найденной позиции, из файла и сохраняется в объект ***policy***.
– Если чтение прошло успешно, выводится информация о найденном страховом полисе, включая его номер, компанию и фамилию владельца.
– Файл закрывается, и функция завершает выполнение.
– Если ключ текущего элемента не совпадает с заданным ключом, флаг устанавливается в ***false***, что означает, что полис не был найден в текущем списке.
– Если в конечном итоге флаг остается ***false***, это означает, что элемент не был найден в хэш-таблице, и выводится сообщение, указывающее на то, что страховой полис не найден в файле.

Общая цель этой функции - поиск страхового полиса в хэш-таблице, и, если он найден, извлечение его информации из соответствующего бинарного файла.

### 2.3.2. Код программы

*Листинг 8. Общий код программы задания 3.3*

```
#include <unordered_map>// Непорядковое отображение (коллекция пар: ключ-значение)
#include <iostream>
#include <fstream>
#include <format>
#include <string>
#include <vector>
```

```
#include <list>
#include <set>
#include <chrono>


using namespace std;


//Структура записи о пациенте: номер полиса, фамилия, имя, отчество, код
заболевания, дата установки диагноза, код врача.
struct InsuranceNote {
    unsigned long int number = 0;
    string surname;
    string first_name;
    string mid_name;
    unsigned long int diseaseCode = 0;
    string date;
    unsigned long int doctorCode = 0;
};
// Структура записи в хеш-таблице
struct HashTableEntry
{
    unsigned long int key;
    int index;
};


// Номер полиса, состоящий из 8 чисел
unsigned long int Randomizer()
{
    unsigned long int min = 10000000;
    unsigned long int max = 99999999;
    return min + rand() % (max - min + 1);
}


// Хеш-функция методом деления
int HashFun(unsigned long int key, int tableSize)
{
    return key % tableSize;
}


// Вставка ключа в таблицу
void Insert(unordered_map<int, list<HashTableEntry>>& hashTable, unsigned long int
key, int tableSize, int index)
{
    HashTableEntry entry;
    entry.key = key;
    entry.index = index;
    int hash = HashFun(key, tableSize);
    hashTable[hash].push_back(entry);
}


// Удаление ключа из таблицы и файла
void Remove(unordered_map<int, list<HashTableEntry>>& hashTable, InsuranceNote&
insNote, const string binary, unsigned long int key, int tableSize)
{
    int hash = HashFun(key, tableSize);
    bool corr = false;

    auto& entries = hashTable[hash];// Получение ссылки на список (цепь записей)

    list<HashTableEntry> tempEntries;// Временная цепочка
```

```
    for (auto it = entries.begin(); it != entries.end(); ++it)
    {
        if (it->key != key)
        {
            tempEntries.push_back(*it);// Удаление из хеш-таблицы
        }
        else if (it->key == key)
        {
            // Удаление из файла
            corr = true;

            cout << endl << "Начался процесс удаления записи из файла" << endl;
            ofstream file(binary, ios::binary | ios::in | ios::out);
            if (file)
            {
                file.seekp(it->index * sizeof(InsuranceNote), ios::beg);

                // Пустая запись
                InsuranceNote empty;
                empty.number = 0;
                empty.surname = "";
                empty.first_name = "";
                empty.mid_name = "";
                empty.diseaseCode = 0;
                empty.date;
                empty.doctorCode = 0;
                file.write(reinterpret_cast<char*>(&empty),
sizeof(InsuranceNote));
                cout << "   Полученная запись удалена из файла" << endl;
            }
            file.close();
        }
    }
    entries = tempEntries;
    if (corr == false)
        cout << endl << "Полис не удалось удалить из файла" << endl;
}


// Поиск ключа в таблице и файле
void Find(const unordered_map<int, list<HashTableEntry>>& hashTable,
InsuranceNote& insNote, const string& binary, unsigned long int key, int
tableSize)
{
    int hash = HashFun(key, tableSize);
    bool corr = false;

    // Проверка существования ключа в таблице
    auto it = hashTable.find(hash);
    if (it != hashTable.end())
    {
        // Получение ссылки на цепь записей
        auto& entries = it->second;

        for (const HashTableEntry& entry : entries)
        {
            if (entry.key == key)
            {
                cout << endl << "Страховой полис найден в таблице" << endl;
                ifstream file(binary, ios::binary);
                if (file)
                {
                    cout << "   Получаем запись из файла" << endl;
                    corr = true;
```

```
                    file.seekg(entry.index * sizeof(InsuranceNote), ios::beg);
                    if (file.read(reinterpret_cast<char*>(&insNote),
sizeof(InsuranceNote)))
                    {
                        cout << "   Номер полиса ---> " << insNote.number << endl;
                        cout << "   Код доктора ---> " << insNote.doctorCode <<
endl;
                        cout << "   Фамилия владельца ---> " << insNote.surname <<
endl;
                        cout << "   Имя владельца ---> " << insNote.first_name <<
endl;
                        cout << "   Отчество владельца ---> " << insNote.mid_name
<< endl;
                        cout << "   Дата диагноза ---> " << insNote.date << endl;
                        cout << "   Код болезни ---> " << insNote.diseaseCode <<
endl;
                    }
                }
                file.close();
                return;
            }
            else
                corr = false;
        }
    }
    if (corr == false)
        cout << endl << "Страховой полис не найден в файле" << endl;
}


// Рехеширование таблицы
void Rehash(unordered_map<int, list<HashTableEntry>>& hashTable, int newTableSize)
{
    unordered_map<int, list<HashTableEntry>> newHashTable(newTableSize);
    for (const auto& entry : hashTable)// Проход по всем элементам хеш-таблицы
    {
        for (const HashTableEntry& hashEntry : entry.second)// Проход по всем
записям хеша
        {
            int newHash = HashFun(hashEntry.key, newTableSize);
            newHashTable[newHash].push_back(hashEntry);
        }
    }
    hashTable = newHashTable;
}


// Определение простого числа
bool IsPrime(int n)
{
    if (n <= 1)
    {
        return false;
    }
    if (n <= 3)
    {
        return true;
    }
    if (n % 2 == 0 || n % 3 == 0)
    {
        return false;
    }
    for (int i = 5; i * i <= n; i += 6)
    {
```

```
        if (n % i == 0 || n % (i + 2) == 0)
        {
            return false;
        }
    }
    return true;
}


int CalculateSize(int currentSize)
{
    // Умножение текущего размера на некоторый множитель

    int newSize = currentSize * 2;
    // Проверка на простое число, чтобы уменьшить вероятность коллизий
    while (!IsPrime(newSize))
    {
        newSize++;
    }
    return newSize;
}


void Output(unordered_map<int, list<HashTableEntry>>& hashTable)
{
    // Вывод списка хеш-кода и ключа
    cout << endl << "После изменения" << endl;
    for (const auto& entry : hashTable)
    {
        int hash = entry.first;// Получение хеш-кода
        const list<HashTableEntry>& entries = entry.second;// Получение списка
записей
        for (const HashTableEntry& hashEntry : entries)
        {
            unsigned long int key = hashEntry.key;// Получение ключа
            printf("   Индекс записи:  %3d   Ключ:  %8d   Хеш:  %2d\n",
hashEntry.index, key, hash);
        }
    }
}

int main()
{
    // ЗАДАНИЕ 3.3

    setlocale(LC_ALL, "ru");
    srand(static_cast<unsigned int>(time(nullptr)));// Инициализация генератора
    const string text = "textFile.txt";
    const string binary = "binaryFile.bin";

    ofstream textFile(text);// Текстовый файл
    if (!textFile)
    {
        cout << "Текстовый файл не удалось открыть для записи" << endl;
        return 1;
    }
    cout << "Текстовый файл успешно открыт для записи" << endl << endl;

    // Открываем файл для записи в бинарном режиме (флаг - ios::binary)
    ofstream binaryFileOut(binary, ios::binary);// Бинарный файл
    if (!binaryFileOut)
    {
        cerr << "Бинарный файл не удалось открыть для записи" << endl;
        return 1;
    }
```

```
    cout << "Бинарный файл успешно открыт для записи" << endl << endl;

    int quant = 5;// Количество записей
    set<int> unique;

    InsuranceNote insNote;
    for (int i = 0; i < quant; ++i)
    {
        do {
            insNote.number = Randomizer();
        } while (unique.count(insNote.number) > 0);
        unique.insert(insNote.number);
        insNote.surname = "Фамилия_" + to_string(i);
        insNote.first_name = "Имя_" + to_string(i);
        insNote.mid_name = "Отчество_" + to_string(i);
        insNote.date = (to_string(Randomizer() % 32 + 1) + "." +
to_string(Randomizer() % 13 + 1) + "." + to_string(Randomizer() % 23 + 2000));
        insNote.diseaseCode = Randomizer() % 10 + 100;
        insNote.doctorCode = Randomizer() % 101000 + 100000;

        textFile << insNote.number << ' ' << insNote.surname << ' ' <<
insNote.first_name << ' ' << insNote.mid_name << ' ' << insNote.date << ' ' <<
insNote.diseaseCode << ' ' << insNote.doctorCode << '\n';

        binaryFileOut.write(reinterpret_cast<const char*>(&insNote),
sizeof(insNote));
    }
    cout << "Генерация текстового и бинарного файла завершена" << endl;


    size_t bytes = sizeof(InsuranceNote);
    cout << "Размер записи в байтах: " << bytes << endl;
    cout << "Количество записей: " << quant << endl << endl;

    textFile.close();
    binaryFileOut.close();

    int tableSize = 3;// Размер хеш-таблицы
    unordered_map<int, list<HashTableEntry>> hashTable;// Создание хеш-таблицы с
цепочками

    // Добавляем элементы в хеш-таблицу
    ifstream binaryFileIn(binary, ios::binary);
    if (!binaryFileIn)
    {
        cout << "Бинарный файл не удалось открыть для чтения" << endl;
        return 1;
    }
    cout << "Бинарный файл успешно открыт для чтения" << endl << endl;

    cout << "   Проверка хеш-индексов" << endl << endl;
    int index = 0;
    while (binaryFileIn.read(reinterpret_cast<char*>(&insNote), sizeof(insNote)))
    {
        // Создание записи для хеш-таблицы
        HashTableEntry entry;
        entry.key = insNote.number;
        entry.index = index;
        index++;
        int hash = HashFun(entry.key, tableSize);// Вычисление хеш-кода для ключа
        hashTable[hash].push_back(entry);// Добавление записи в цепочку хеш-
таблицы

        printf("   Индекс записи:  %3d   Ключ:  %2d   Хеш:  %2d\n", entry.index,
entry.key, hash);
```

```
    }
    binaryFileIn.close();

    // Фактор заполнения (проверка при создании файла)
    double loadFactor = static_cast<double>(hashTable.size()) / tableSize;
    if (loadFactor > 0.8)
    {
        int newTableSize = CalculateSize(tableSize);// Определение нового размера
таблицы
        tableSize = newTableSize;
        Rehash(hashTable, tableSize);
    }

    bool flag = true;
    auto start_time = chrono::high_resolution_clock::now();
    auto end_time = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end_time -
start_time);
    while (flag)
    {
        int choice;
        cout << endl << "Выберите действие со структурой: " << endl << "   1 -
вставка ключа в таблицу" << endl << "   2 - поиск ключа в файле" << endl << "   3
- удаление ключа из файла и таблицы" << endl << "   4 - выход из тестирования" <<
endl;
        cin >> choice;
        switch (choice)
        {
        case 1:
            unsigned long int insert;
            cout << "Введите номер полиса, который вы желаете добавить (8 чисел) -
--> ";
            cin >> insert;
            Insert(hashTable, insert, tableSize, index);
            loadFactor = static_cast<double>(hashTable.size()) / tableSize;
            if (loadFactor > 0.8)
            {
                int newTableSize = CalculateSize(tableSize);// Определение нового
размера таблицы
                tableSize = newTableSize;
                Rehash(hashTable, newTableSize);
            }
            Output(hashTable);
            break;
        case 2:
            unsigned long int found;
            cout << "Введите номер полиса, который вы желаете найти (8 чисел) --->
";
            cin >> found;
            start_time = chrono::high_resolution_clock::now();

            Find(hashTable, insNote, binary, found, tableSize);

            end_time = chrono::high_resolution_clock::now();
            duration = chrono::duration_cast<chrono::microseconds>(end_time -
start_time);
            cout << endl << " Время выполнения: " << duration.count() << " [мкс]"
<< endl;
            break;
        case 3:
            unsigned long int remove;
            cout << "Введите номер полиса, который вы желаете удалить (8 чисел) --
-> ";
            cin >> remove;
            Remove(hashTable, insNote, binary, remove, tableSize);
```

```
            Output(hashTable);
            break;
        case 4:
            flag = false;
            break;
        default:
            break;
        }
    }
    return 0;
}
```

### 2.3.3. Тестирование

```
Размер записи в байтах: 184
Количество записей: 5

Бинарный файл успешно открыт для чтения

   Проверка хеш-индексов

   Индекс записи:    0   Ключ:  10007680   Хеш:   1
   Индекс записи:    1   Ключ:  10017477   Хеш:   0
   Индекс записи:    2   Ключ:  10022736   Хеш:   0
   Индекс записи:    3   Ключ:  10005569   Хеш:   2
   Индекс записи:    4   Ключ:  10028428   Хеш:   1

Выберите действие со структурой:
   1 - вставка ключа в таблицу
   2 - поиск ключа в файле
   3 - удаление ключа из файла и таблицы
   4 - выход из тестирования
1
Введите номер полиса, который вы желаете добавить (8 чисел) ---> 12345678

После изменения
   Индекс записи:    0   Ключ:  10007680   Хеш:   4
   Индекс записи:    4   Ключ:  10028428   Хеш:   4
   Индекс записи:    1   Ключ:  10017477   Хеш:   1
   Индекс записи:    2   Ключ:  10022736   Хеш:   3
   Индекс записи:    3   Ключ:  10005569   Хеш:   0
   Индекс записи:    5   Ключ:  12345678   Хеш:   2

Выберите действие со структурой:
   1 - вставка ключа в таблицу
   2 - поиск ключа в файле
   3 - удаление ключа из файла и таблицы
   4 - выход из тестирования
3
Введите номер полиса, который вы желаете удалить (8 чисел) ---> 12345678

Начался процесс удаления записи из файла
   Полученная запись удалена из файла

После изменения
   Индекс записи:    0   Ключ:  10007680   Хеш:   4
   Индекс записи:    4   Ключ:  10028428   Хеш:   4
   Индекс записи:    1   Ключ:  10017477   Хеш:   1
   Индекс записи:    2   Ключ:  10022736   Хеш:   3
   Индекс записи:    3   Ключ:  10005569   Хеш:   0
```

```
Текстовый файл успешно открыт для записи

Бинарный файл успешно открыт для записи

Генерация текстового и бинарного файла завершена
Размер записи в байтах: 184
Количество записей: 5

Бинарный файл успешно открыт для чтения

   Проверка хеш-индексов

   Индекс записи:    0   Ключ:  10007830   Хеш:   1
   Индекс записи:    1   Ключ:  10029577   Хеш:   1
   Индекс записи:    2   Ключ:  10025316   Хеш:   0
   Индекс записи:    3   Ключ:  10024246   Хеш:   1
   Индекс записи:    4   Ключ:  10012037   Хеш:   2

Выберите действие со структурой:
   1 - вставка ключа в таблицу
   2 - поиск ключа в файле
   3 - удаление ключа из файла и таблицы
   4 - выход из тестирования
1
Введите номер полиса, который вы желаете добавить (8 чисел) ---> 12345678

После изменения
   Индекс записи:    0   Ключ:  10007830   Хеш:   0
   Индекс записи:    2   Ключ:  10025316   Хеш:   0
   Индекс записи:    4   Ключ:  10012037   Хеш:   0
   Индекс записи:    1   Ключ:  10029577   Хеш:   5
   Индекс записи:    3   Ключ:  10024246   Хеш:   1
   Индекс записи:    5   Ключ:  12345678   Хеш:   2

Выберите действие со структурой:
   1 - вставка ключа в таблицу
   2 - поиск ключа в файле
   3 - удаление ключа из файла и таблицы
   4 - выход из тестирования
2
Введите номер полиса, который вы желаете найти (8 чисел) ---> 12345678

Страховой полис найден в таблице
   Получаем запись из файла

 Время выполнения: 9060 [мкс]
```

# 3. ВЫВОД

Изучение навыков разработки и применения хеш-таблицы при поиске данных в других структурах данных, таких как файлы, предоставляет ценные знания и навыки в информатике и программировании. Вот несколько ключевых выводов:

***Понимание хеш-таблиц:*** Изучение хеш-таблиц позволяет понять их внутреннее устройство и принципы работы. Хеш-таблицы представляют собой эффективные структуры данных для быстрого доступа к данным с использованием хеш-функций.

***Разрешение коллизий***: Навык разрешения коллизий — это важная часть работы с хеш-таблицами. Методы разрешения коллизий, такие как метод цепочек или открытая адресация, позволяют обработать ситуации, когда двум или более ключам соответствует одно и то же хеш-значение.

***Хеш-функции:*** Понимание проектирования эффективных хеш-функций имеет большое значение. Хорошо выбранная хеш-функция способствует равномерному распределению данных в хеш-таблице.

***Использование в поиске***: Навык применения хеш-таблицы при поиске данных в файлах или других структурах данных позволяет создавать эффективные алгоритмы поиска. Это особенно полезно при работе с большими объемами данных, так как уменьшает временную сложность операций поиска.

***Оптимизация производительности***: Правильное использование хеш-таблиц может значительно повысить производительность при поиске данных. Это особенно важно в приложениях, где быстрый доступ к данным играет ключевую роль.

***Комплексное тестирование***: Необходимо уметь разрабатывать и проводить тестирование для уверенности в правильности работы алгоритмов и структур данных, включая хеш-таблицы.

***Распространенное применение:*** Хеш-таблицы широко применяются в реальных приложениях, таких как базы данных, поисковые системы, кэширование данных и другие. Понимание их работы и применение в практических задачах имеют высокую ценность.

Итак, изучение хеш-таблиц и их применение при поиске данных в файлах представляет собой важный аспект в области программирования и информатики, который может значительно улучшить навыки разработки и решения реальных задач.