МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания № 6.1

Тема:

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

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнила студент группы ИКБО-42-23 Туляшева А.Т.

Принял ассистент Муравьева Е.А.

Москва 2024

Содержание

ЦЕЛЬ РАБОТЫ 3

ЗАДАНИЕ 1 3

ЗАДАНИЕ 2 6

ЗАДАНИЕ 3 13

ВЫВОД 18

ЦЕЛЬ РАБОТЫ

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

ЗАДАНИЕ 1

Необходимо ответить на вопросы:

1. Что такое хеширование? В каких областях оно применяется?

Хеширование — это процесс преобразования входных данных (например, строки, файла или любого другого типа данных) в фиксированный размер, который называется хеш-значением или хеш-кодом. Процесс осуществляется с помощью хеш-функции. Хеширование используется в различных областях, включая базы данных (для быстрого поиска и доступа к данным), криптографию (для обеспечения целостности данных), кэширование (для быстрого доступа), а также в системах контроля версий и распределенных системах.

2. Расскажите о назначении хеш-фунции.

Назначение хеш-функции заключается в том, чтобы взять произвольные данные и преобразовать их в фиксированное значение (хеш), которое будет уникально представлять эти данные. Хеш-функции должны иметь некоторые свойства, такие как быстрый расчет, предсказуемость (одни и те же входные данные всегда дают один и тот же хеш), а также устойчивость к коллизиям (разные входные данные должны давать разные хеши).

3. Что такое коллизия? Назовите приёмы устранения (разрешения) коллизий.

Коллизия — это ситуация, когда два разных входных значения генерируют одно и то же хеш-значение. Устранение коллизий может быть достигнуто с помощью различных методов, таких как:

- Открытая адресация (поиск другого свободного места в хеш-таблице).
- Цепное хеширование (хранение нескольких значений в одном

пространстве хеш-значения в виде списка).

- Двойное хеширование (использование второй хеш-функции для вычисления смещения).

4. Что такое «открытый адрес» по отношению к хеш-таблице? Открытый адрес в хеш-таблице — это метод разрешения коллизий, при котором все элементы хранятся непосредственно в самой хеш-таблице. Когда возникает коллизия, поиск следующего свободного места происходит по заранее определённой схеме (например, линейное пробирование, квадратичное пробирование или двойное хеширование).

5. Как в хеш-таблице с открытым адресом реализуется коллизия?

В хеш-таблице с открытым адресом коллизия реализуется путем поиска следующей свободной ячейки в таблице по определенному алгоритму. Например, если вычисленный индекс уже занят, можно перейти к следующему индексу и проверить его, повторяя процесс, пока не будет найдено свободное место.

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

Проблема, возникающая после удаления элемента из хеш-таблицы с открытым адресом, заключается в том, что последующие поиски могут не найти элементы, которые были добавлены после удалённого, так как у них могла быть зависимость от удалённого элемента. Эту проблему можно устранить, заменив удалённый элемент специальным маркером (например, "удалён"), который будет отмечать, что в этой ячейке раньше хранился элемент, но он был удалён.

7. Что определяет коэффициент нагрузки в хеш-таблице?

Коэффициент нагрузки в хеш-таблице определяет степень заполненности таблицы и выражается как отношение количества элементов, хранящихся в таблице, к общему количеству ячеек в таблице.

Он позволяет оценить эффективность хеширования и определяет вероятность возникновения коллизий.

8. Что такое «первичный кластер» в таблице с открытым адресом?

Первичный кластер — это явление, возникающее в таблицах с открытой адресацией, когда элементы, вставленные в таблицу, образуют непрерывный блок (кластер) занятых ячеек. Это может привести к ухудшению производительности при вставке и поиске, особенно если много ячеек занято рядом друг с другом.

9. Как реализуется двойное хеширование?

Двойное хеширование — это метод разрешения коллизий, при котором при возникновении коллизии используется вторая хеш-функция для вычисления смещения, которое будет применяться для поиска следующего пробного индекса. Это уменьшает вероятность кластеризации и распределяет элементы более равномерно по таблице.

10. Что такое цепное хеширование? С чем связана основная проблема этого метода?

Цепное хеширование — это подход к разрешению коллизий, при котором в каждой ячейке хеш-таблицы хранится ссылка на список, содержащий все элементы, соответствующие этому хеш-значению. Основная проблема этого метода связана с возможным разрастанием цепочек (списков) в ячейках, что может привести к ухудшению производительности, особенно если хеш-функция недостаточно равномерно распределяет элементы.

11. Что такое рехеширование? Назовите критерий необходимости рехеширования.

Рехеширование — это процесс, при котором хеш-таблица увеличивается в размере, и все элементы перехешируются и размещаются в новой таблице. Критерий необходимости рехеширования — это превышение определенного порога коэффициента нагрузки.

12. В чём заключается идея хеширования с открытой адресацией?

Идея хеширования с открытой адресацией заключается в том, что все данные хранятся внутри самой хеш-таблицы, и в случае коллизий осуществляется поиск свободной ячейки по заранее определенному алгоритму. Таким образом, каждый элемент хранится в столбце таблицы, что позволяет быстро находить и извлекать данные.

ЗАДАНИЕ 2

Формулировка задания:

Разработайте приложение, которое использует хеш-таблицу (пары «ключ–хеш») для организации прямого доступа к элементам динамического множества полезных данных (записи в файле). Множество реализуйте на массиве, структура элементов (перечень полей) которого приведена в индивидуальном варианте в таблице 1. Метод разрешения коллизии также представлен в индивидуальном варианте в таблице 1.

Для обеспечения прямого доступа к элементам динамического множества 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.

1. Управление хеш-таблицей.

1) Определить структуру элемента хеш-таблицы и структуру хеш-таблицы в соответствии с методом разрешения коллизии, указанном в варианте.

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

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

4) 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

Примечание. Для метода с открытым адресом подготовить тест для поиска ключа, который размещен в таблице после удаленного ключа, с одним значением хеша для этих ключей.

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

Задание в соответствии с индивидуальным вариантом (31 вариант):

Тип хеш-таблицы (метод разрешения коллизии): Открытый адрес (смещение на 1).

Структура записи двоичного файла: Киноафиша города. Структура записи о сеансе: название кинотеатра, название фильма, дата, время начала, стоимость билета.

Код программы и описание алгоритмов операций

Определили три структуры для работы: MovieSession – структура записи по индивидуальному варианту, HashEntry – структура элемента хеш-таблицы и HashTable – структура хеш-таблицы. Также ввели константой размер хеш-таблицы. Для структуры HashEntry ввели конструктор по умолчанию, а для структуры HashTable ввели и конструктор, и деструктор (рис. 1).

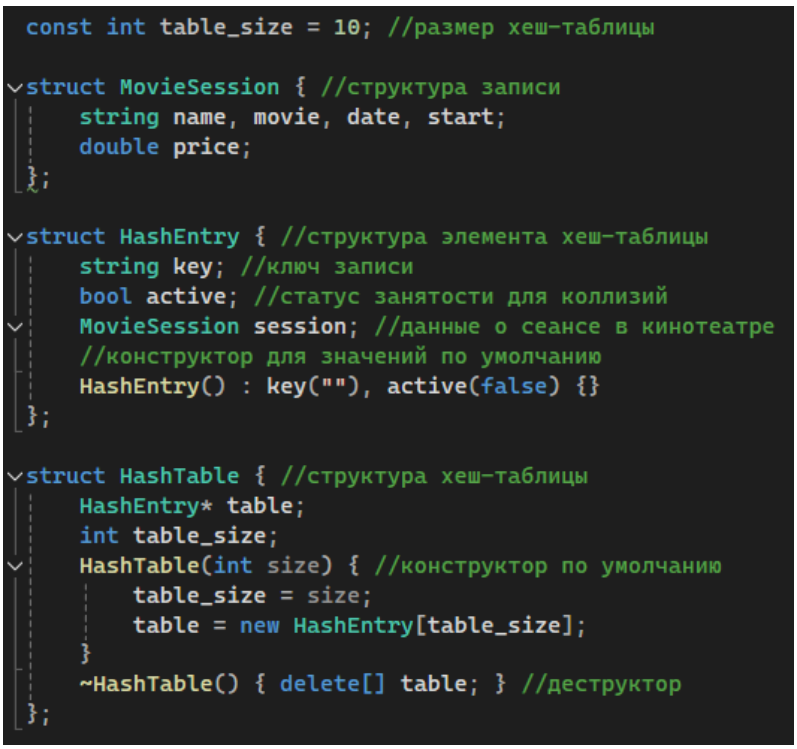


Рисунок 1 – Код задания 2

На рисунке 2 представлена хеш-функция.

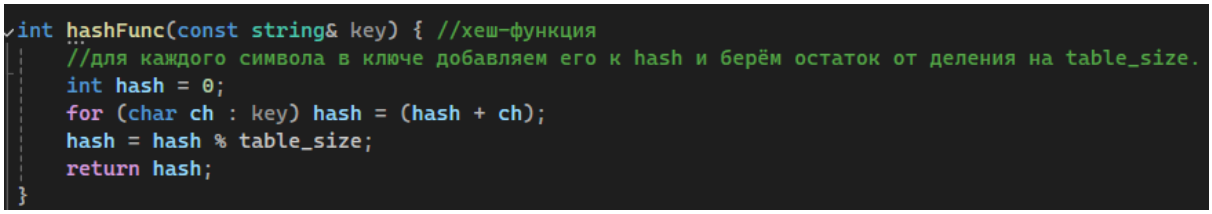


Рисунок 2 – Код задания 2

Была разработана функция для вставки ключа в таблицу. Сначала вычисляет индекс для вставки, используя хеш-функцию. Потом проверяет на наличие свободного места: используется открытое адресование для поиска свободного места. Если нашли свободное место: сохраняем новый ключ, индекс записи и помечаем запись как активную.

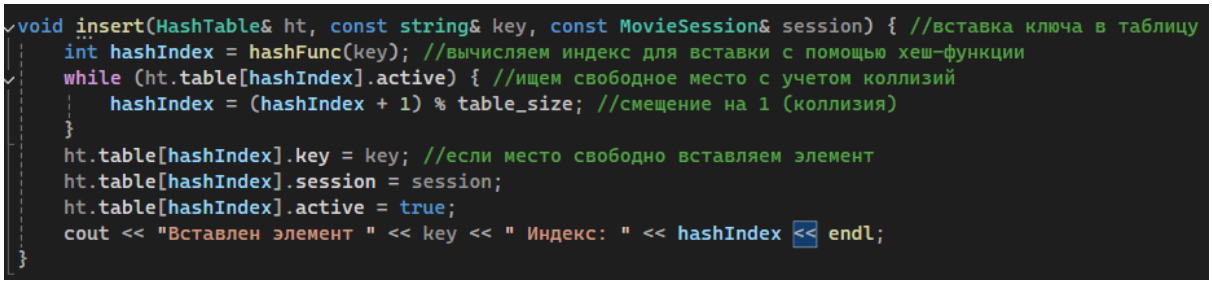


Рисунок 3 – Код задания 2

Булевая функция для поиска ключа в таблице. Она ищет ключ, и если элемент найден, выводит его индекс.

```
bool search(const HashTable& ht, const string& key) { //поиск ключа в таблице
    int hashIndex = hashFunc(key);
    while (ht.table[hashIndex].active) { //если элемент занят
        if (ht.table[hashIndex].key == key) { //если ключ совпадает
            cout << "Найден элемент " << key << " Индекс: " << hashIndex << endl;
            return true;
        }
        hashIndex = (hashIndex + 1) % table_size; //смещение на 1
    }
    cout << "Элемент " << key << " не найден" << endl;
    return false;
}
```

Рисунок 4 – Код задания 2

Функция удаляет элемент из хеш-таблицы, сначала вычисляя индекс. Поиск ключа для удаления: аналогично вставке, проходим по таблице. Если нашли ключ: помечаем запись как неактивную и выводим сообщение об удалении (рис. 5)

```
void remove(HashTable& ht, const string& key) { //удаление ключа из таблицы
    int hashIndex = hashFunc(key);
    while (ht.table[hashIndex].active) {
        if (ht.table[hashIndex].key == key) {
            ht.table[hashIndex].active = false; //удаляем запись
            cout << "Удален элемент " << key << " Индекс: " << hashIndex << endl;
            return;
        }
        hashIndex = (hashIndex + 1) % table_size; //смещение на 1
    }
    cout << "Элемент " << key << " не был найден для удаления" << endl;
}
```

Рисунок 5 – Код задания 2

Функция для рехеширования создает новую таблицу и копирует туда всё содержимое текущей таблицы, при этом размер таблицы увеличен вдвое. Так как выделяется память для новой таблицы, нужно освободить старую память.

```
void rehash(HashTable& ht) { //рехеширование хеш-таблицы
    int new_size = ht.table_size * 2; //увеличиваем размер
    HashTable newTable(new_size); //создаем новую таблицу
    for (int i = 0; i < ht.table_size; i++) {
        if (ht.table[i].active) {
            //вставляем все существующие элементы в новую таблицу
            insert(newTable, ht.table[i].key, ht.table[i].session);
        }
    }
    delete[] ht.table; //освобождаем старую память
    ht.table = newTable.table; //смена хеш-таблицы
    ht.table_size = new_size;
    newTable.table = nullptr; //предотвращаем двойное освобождение памяти
    //(если ~HashTable пытается освободить область памяти, которая была уже освобождена)
    cout << "\nРехеширование было выполнено";
}
```

Рисунок 6 – Код задания 2

Была написана функция для вывода хеш-таблицы: если запись активна, выводим ключ и индекс; иначе - сообщение о пустом месте.

```
void printTable(const HashTable& ht, int s) { //вывод для наглядности
    int ss = s;
    cout << "Хеш-таблица:\n";
    for (int i = 0; i < ss; ++i) {
        if (ht.table[i].active) {
            cout << i << ") "
                << "Кинотеатр: " << ht.table[i].key << ", "
                << "Фильм: " << ht.table[i].session.movie << ", "
                << "Дата: " << ht.table[i].session.date << ", "
                << "Время: " << ht.table[i].session.start << ", "
                << "Цена: " << ht.table[i].session.price << "\n";
        }
        else {
            cout << i << ") (пусто)\n";
        }
    }
    cout << endl;
}
```

Рисунок 7 – Код задания 2

Результаты тестирования

Проведем тестирование. Сначала выведем пустую хеш-таблицу. Затем с помощью функции вставки добавим в таблицу 2 записи о кинотеатре (рис. 8).

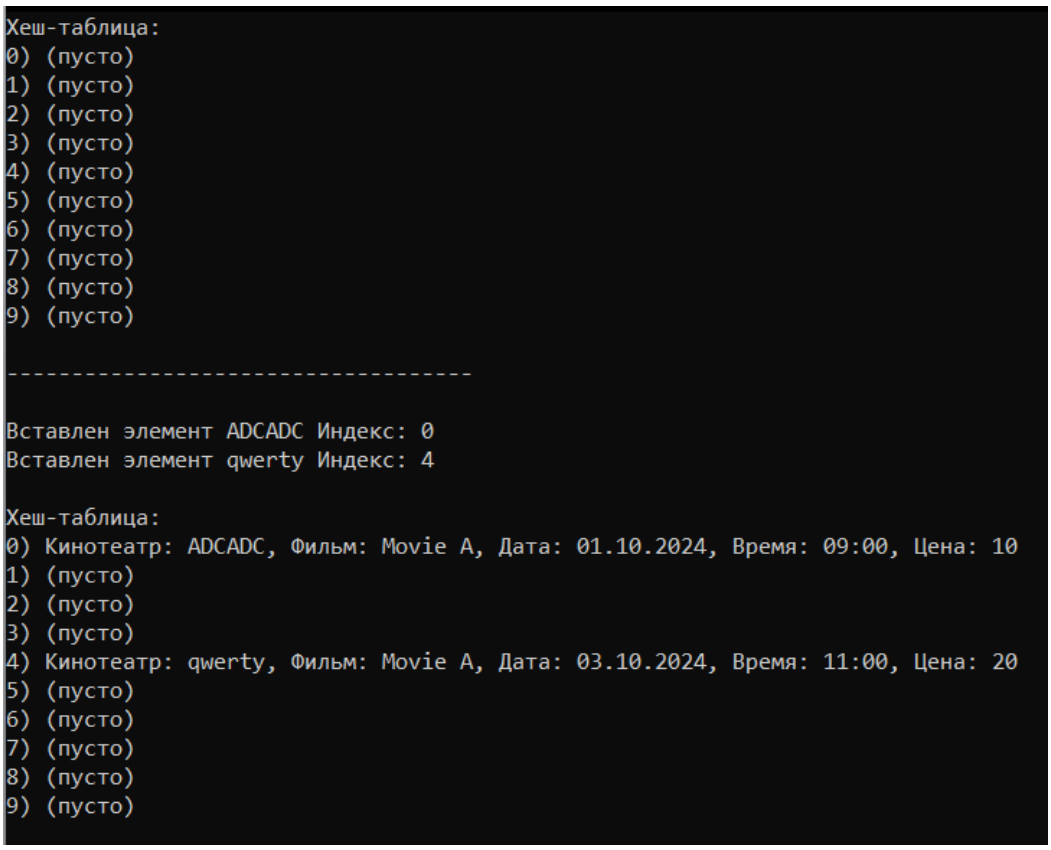


Рисунок 8 – Тестирование функции вставки

Протестируем функцию поиска. Первый элемент был найден и вывелся его индекс. Второго элемента не существует в хеш-таблице, поэтому было выведено соответствующее сообщение (рис. 9).

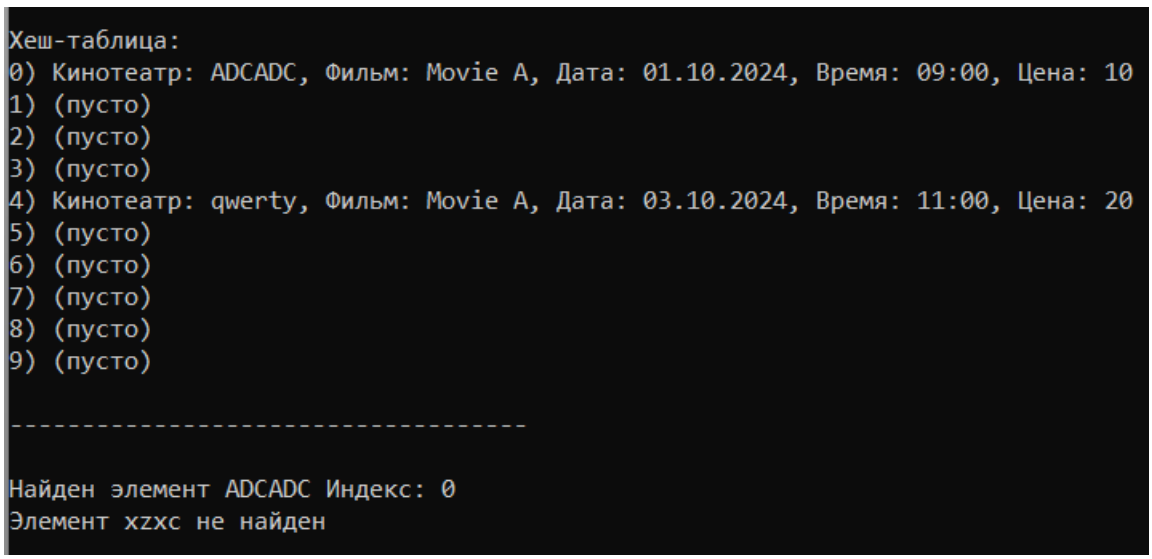


Рисунок 9 – Тестирование функции поиска

С помощью функции удаления по ключу удалили из хеш-таблицы запись под индексом 4 (рис. 10).

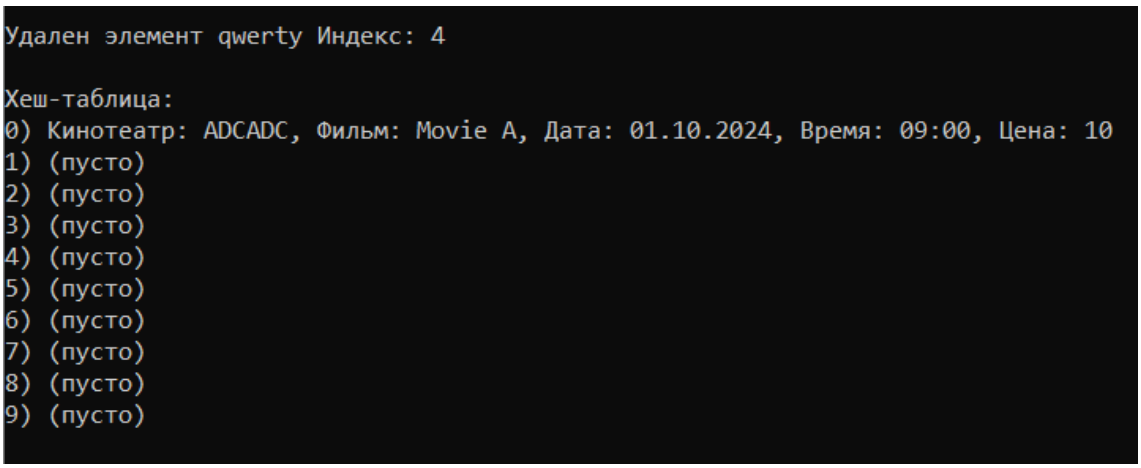


Рисунок 10 – Тестирование функции удаления

Рассмотрим пример, когда с помощью хеш-функции для разных ключей получаются одинаковые хеш-значения. Это пример коллизии. В соответствии с вариантом метод разрешения коллизии - открытый адрес (смещение на 1). Запись под ключом xzxc была вставлена в хеш-таблицу под индексом 2, так как это первый свободный индекс после записи под ключом zxc с индексом 1.

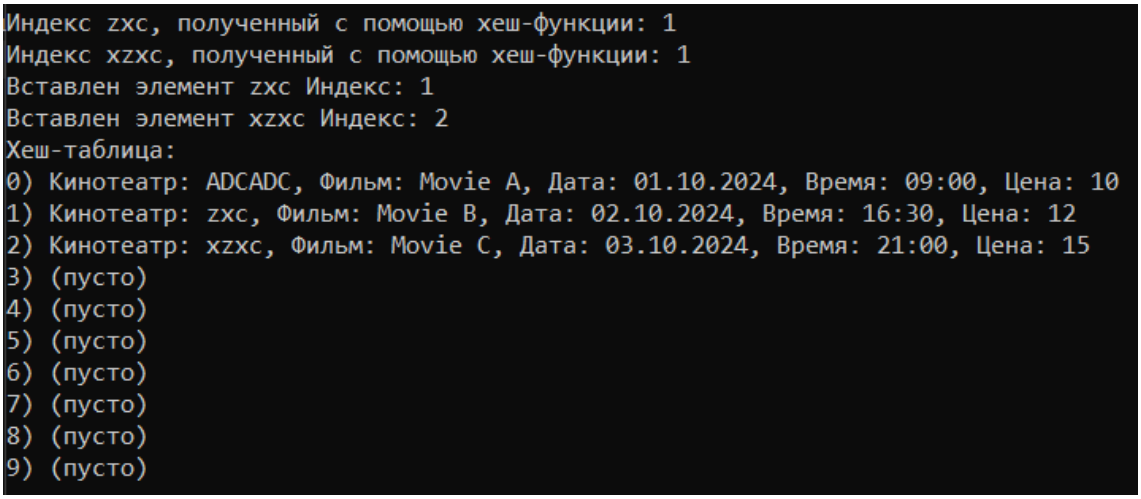


Рисунок 11 – Ситуация коллизии

На рисунке 12 представлено рехеширование. Размер таблицы увеличился вдвое, а записи из старой таблицы были переписаны в новую под теми же индексами (хеш-значениями).

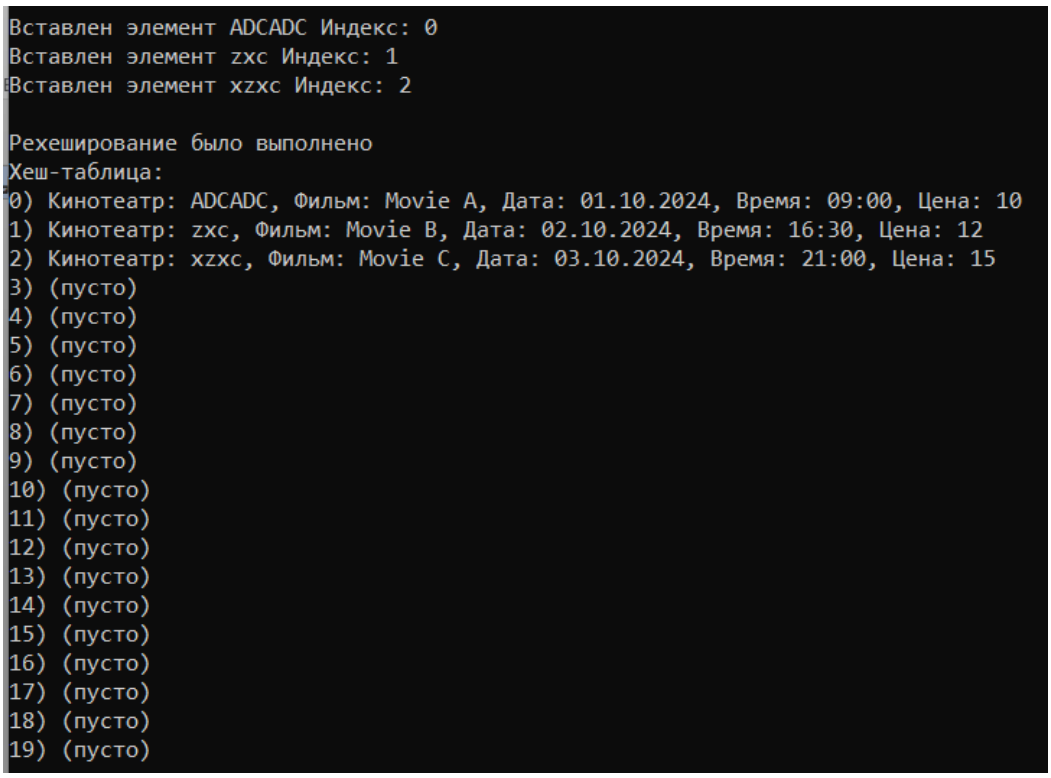


Рисунок 12 – Ситуация коллизии

Тестирование показало, что код работает верно.

ЗАДАНИЕ 3

Формулировка задания:

Управление бинарным файлом посредством хеш-таблицы.

В заголовочный файл подключить заголовочные файлы: управления хеш-таблицей, 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.

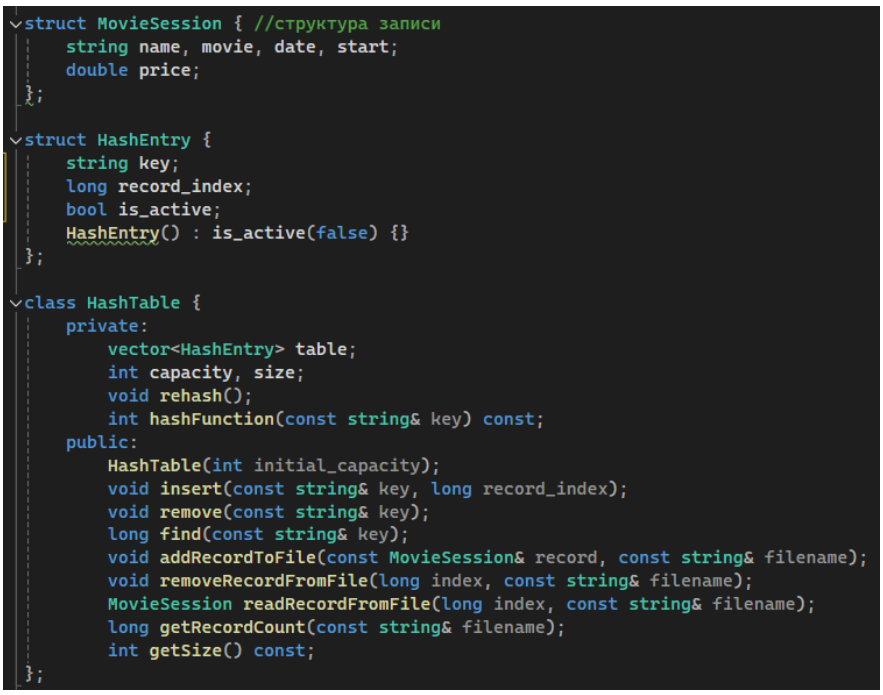
- Включите в файл записи как не приводящие к коллизиям, так и приводящие.

- 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.

Код программы и описание алгоритмов операций

Были по аналогии со вторым заданием разработаны структуры MovieSession и HashEntry. Теперь структура HashTable стала классом, в котором дополнитель интегрированы функции работы с файлами. На рисунке 13 представлен заголовочный файл hash_table.h.



```
struct MovieSession { //структура записи
    string name, movie, date, start;
    double price;
};

struct HashEntry {
    string key;
    long record_index;
    bool is_active;
    HashEntry() : is_active(false) {}
};

class HashTable {
    private:
        vector<HashEntry> table;
        int capacity, size;
        void rehash();
        int hashFunction(const string& key) const;
    public:
        HashTable(int initial_capacity);
        void insert(const string& key, long record_index);
        void remove(const string& key);
        long find(const string& key);
        void addRecordToFile(const MovieSession& record, const string& filename);
        void removeRecordFromFile(long index, const string& filename);
        MovieSession readRecordFromFile(long index, const string& filename);
        long getRecordCount(const string& filename);
        int getSize() const;
};
```

Рисунок 12 – hash_table.h

Основная идея функций вставки, удаления, рехеширования, поиска, хеш-функции осталась такой же, как во втором задании. Также был добавлен конструктор.

```
HashTable::HashTable(int initial_capacity) : capacity(initial_capacity), size(0) {
    table.resize(capacity); //конструктор
}

int HashTable::hashFunc(const string& key) const { //хеш-функция
    int hash = 0;
    for (char ch : key) hash = (hash + static_cast<int>(ch)) % capacity;
    return hash;
}

void HashTable::rehash() { //рехеширование
    vector<HashEntry> old_table = table;
    capacity *= 2; //увеличим объем таблицы
    table.clear();
    table.resize(capacity);
    size = 0;
    for (const HashEntry& entry : old_table) {
        if (entry.active) {
            insert(entry.key, entry.record_index);
        }
    }
}

void HashTable::insert(const string& key, long record_index) { //вставка
    if (size >= capacity / 2) rehash();
    int index = hashFunc(key);
    while (table[index].active) {
        index = (index + 1) % capacity; //открытая адресация
    }
    table[index].key = key;
    table[index].record_index = record_index;
    table[index].active = true;
    size++;
}
```

Рисунок 13 – hash_table.cpp

```cpp
void HashTable::remove(const string& key) { //удаление
    int index = hashFunc(key);
    while (table[index].active) {
        if (table[index].key == key) {
            table[index].active = false;
            size--;
            return;
        }
        index = (index + 1) % capacity;
    }
}

long HashTable::search(const string& key) { //поиск
    int index = hashFunc(key);
    while (table[index].active) {
        if (table[index].key == key) {
            return table[index].record_index;
        }
        index = (index + 1) % capacity;
    }
    throw runtime_error("Key not found");
}
```

Рисунок 14 – hash_table.cpp

Были добавлены в класс HashTable функции для работы с файлами: чтение из файла, запись в файл, удаление записи из файла и получение количества записей.

```cpp
void HashTable::addRecordToFile(const MovieSession& record, const string& filename) { //добавление записи в файл
    ofstream ofs(filename, ios::binary | ios::app);
    if (!ofs) {
        cerr << "Ошибка при открытии файла для добавления записи." << endl;
        return;
    }
    ofs.write(reinterpret_cast<const char*>(&record), sizeof(MovieSession));
    ofs.close();
}

void HashTable::removeRecordFromFile(long index, const string& filename) { //удаление записи из файла
    ifstream ifs(filename, ios::binary); //простой способ: переписываем файл без этой записи
    ofstream ofs("temp.bin", ios::binary);
    MovieSession record;
    long count = 0;
    while (ifs.read(reinterpret_cast<char*>(&record), sizeof(MovieSession))) {
        if (count != index) {
            ofs.write(reinterpret_cast<const char*>(&record), sizeof(MovieSession));
        }
        count++;
    }
    ifs.close();
    ofs.close();
    remove(filename.c_str());
    rename("temp.bin", filename.c_str());
}

MovieSession HashTable::readRecordFromFile(long index, const string& filename) { //чтение записи из файла
    ifstream ifs(filename, ios::binary);
    MovieSession record;
    ifs.seekg(index * sizeof(MovieSession));
    ifs.read(reinterpret_cast<char*>(&record), sizeof(MovieSession));
    ifs.close();
    return record;
}

long HashTable::getRecordCount(const string& filename) { //получение количества записей
    ifstream ifs(filename, ios::binary);
    if (!ifs) return 0;
    ifs.seekg(0, ios::end);
    long count = ifs.tellg() / sizeof(MovieSession);
    ifs.close();
    return count;
}
```

Рисунок 15 – hash_table.cpp

Тестирование

Было проведено тестирование. На рисунке 16 изображен заполненный bin-файл.

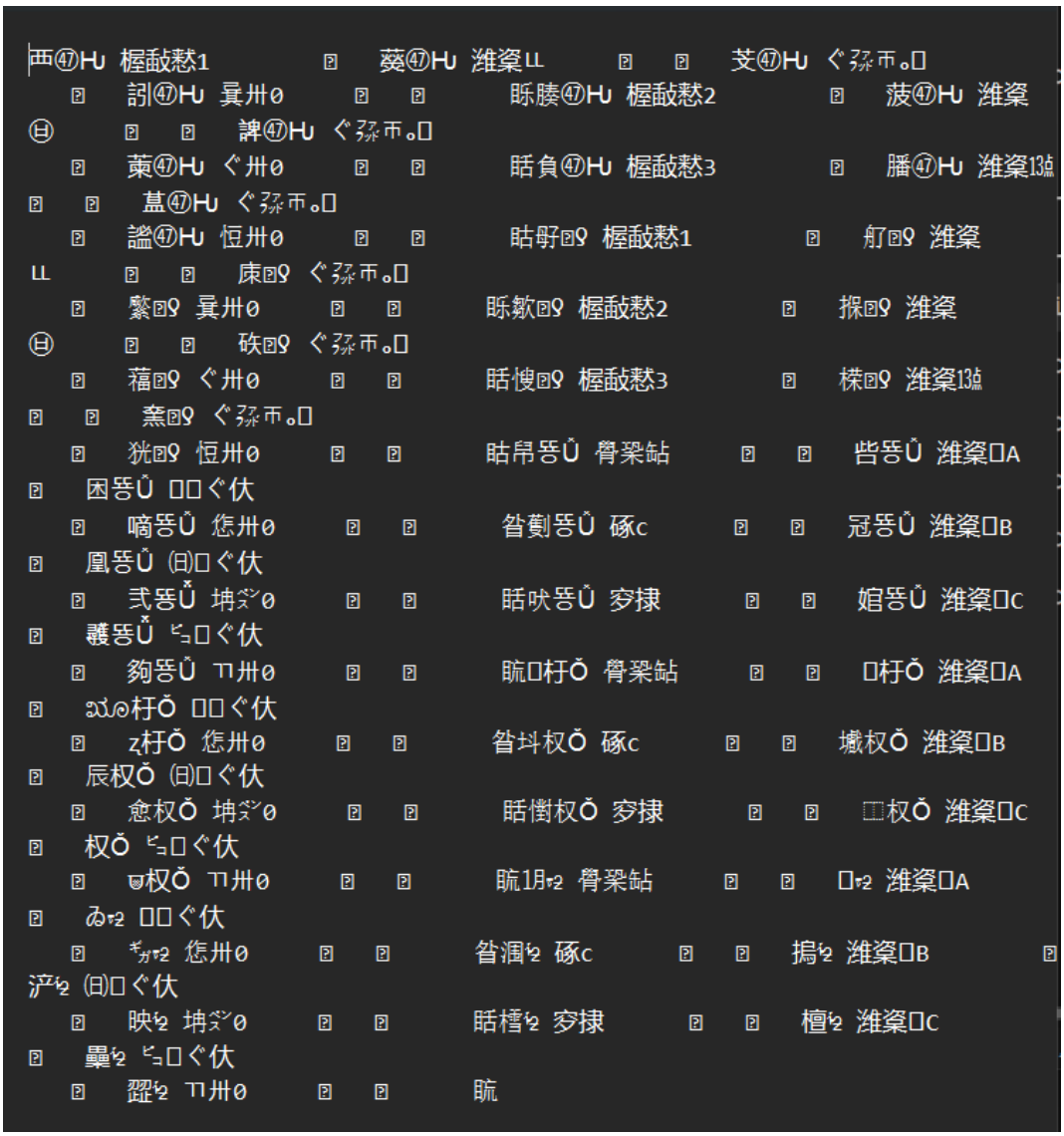


Рисунок 16 – Заполненный bin-файл

Была проверена функция search для поиска записи с ключом «xzxc». Результат представлен на рисунке 17.

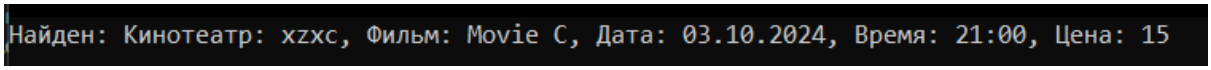


Рисунок 17– Тестирование

Еще была проверена функция remove для удаления записи под ключом ADCADC.

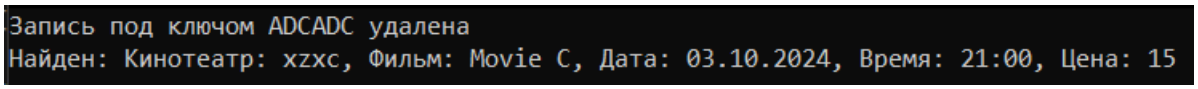


Рисунок 18 – Тестирование

ВЫВОД

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