|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 6.1** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Применение хеш-таблицы для поиска данных в двоичном файле с записями фиксированной длины»** | |
|  | |
| Выполнил студент группы ИНБО-12-23 | Албахтин И.В. |
| Принял ассистент | Муравьёва Е.А. |

Москва 2024

**Оглавление**

[1. Цель работы 3](#_Toc177462904)

[2. Задание №1 3](#_Toc177462905)

[2.1 Постановка задачи 3](#_Toc177462906)

[2.2 Код программы 3](#_Toc177462907)

[2.3 Тестирование 5](#_Toc177462908)

[3. Задание №2 5](#_Toc177462909)

[3.1 Постановка задачи 5](#_Toc177462910)

[3.2 Код программы 5](#_Toc177462912)

[3.3 Тестирование 7](#_Toc177462913)

[3.4 Таблица результатов 7](#_Toc177462914)

[4. Задание №3 7](#_Toc177462915)

[4.1 Постановка задачи 7](#_Toc177462916)

[4.2 Код программы 8](#_Toc177462917)

[4.3 Тестирование 11](#_Toc177462918)

[4.4 Таблица результатов 11](#_Toc177462919)

[5. Вывод 11](#_Toc177462920)

# **Цель работы**

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

# **Задание №1**

## **2.1 Постановка задачи**

Ответить на вопросы:

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

- Это математическое преобразование (отображение) входного ключа – исходных данных (сообщения, массива) в выходную битовую строку определённой длины, производимое алгоритмом хеш-функции. Хеширование как преобразование исходных данных в выходную битовую строку находит применение в таких сферах, как контроль целостности при передаче данных (контрольные суммы), информационная безопасность (защита паролей, ЭЦП) и некоторые другие.

2. Расскажите о назначении **хеш-фунции**.

- Хеширование (хеш-функция) создаёт отображение (соответствие) множества ключей в множество индексов массива (множество целых чисел). Алгоритм хеш-функции преобразует уникальный ключ записи в её индекс в таблице. Структура, позволяющая хранить пары «ключ» –0020«хеш-код» (индекс), называется хеш-таблица. Основное назначение хеш-функции — равномерно распределять элементы в хеш-таблице для минимизации коллизий и ускорения поиска.

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

- Это ситуация, когда для разных ключей хеш-функция создаёт одинаковые значения (индексы). Цепное хеширование (формирование списков из элементов, хешированных с одним индексом). Открытая адресация (хеш-таблица большого объёма).

4. Что такое **«открытый адрес»** по отношению к хеш-таблице?

- **Открытый адрес** – это свободная ячейка хеш-таблицы, ищется следующий свободный элемент; закрытый адрес – это занятая ячейка.

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

- Если в массиве в строке с определённым индексом записи нет, то адрес открыт и в соответствующую строку можно поместить новый элемент. Иначе – адрес закрыт (коллизия) и необходимо по некоему алгоритму осуществить последовательность проб – сместиться относительно закрытого адреса в поисках открытого.

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

- После удаления элемента в хеш-таблице с **открытым адресом** может возникнуть проблема, связанная с нарушением последовательности пробирования. Если после удаления элемента не учесть этот фактор, поиск других элементов, которые были добавлены через пробирование, может завершиться неудачей. **Решение**: Вместо полного удаления элемента используется специальный маркер («удалён»), который обозначает, что ячейка пуста для вставки, но продолжает участвовать в процессе поиска.

7. Что определяет **коэффициент нагрузки** в хеш-таблице?

- Критерием необходимости перестройки массива является соотношение n/m – **коэффициент нагрузки**, где n – это количество уже имеющихся записей, m – длина массива. При достижении значения этого коэффициента 0,75+, следует увеличить длину массива вдвое. Это гарантирует, что длины списков будут относительно небольшими.

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

- **Первичный кластер** — это последовательная группа занятых ячеек в хеш-таблице, которая образуется при использовании линейного пробирования. Кластеры увеличиваются, так как элементы с коллизиями продолжают записываться в соседние ячейки, что снижает эффективность поиска и увеличивает вероятность новых коллизий. В схеме двойного хеширования смещение относительно закрытого адреса кратно величине второй хеш-функции, схожей, но не эквивалентной основной: адрес=h(x)+ih2(x)

9. Как реализуется **двойное хеширование**?

- **Двойное хеширование** — это метод разрешения коллизий, при котором используются две хеш-функции. Первая хеш-функция определяет основное место для вставки элемента, а если возникает коллизия, то вторая хеш-функция вычисляет смещение для поиска следующей свободной ячейки.

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

- **Цепное хеширование** (метод цепочек) — это метод разрешения коллизий, при котором каждая ячейка хеш-таблицы содержит указатель на связный список (цепочку) элементов с одинаковыми хеш-значениями. Когда возникает коллизия, элемент добавляется в конец соответствующей цепочки.

Проблема **однородного хеширования**:

- Желательно, чтобы **цепочки** переполнения были **примерно одной небольшой длины**, иначе сложность поиска увеличивается от константной к **линейной**

Проблема **размера хеш-таблицы**:

- Если сразу создать хеш-таблицу **большого размера**, то многие её элементы могут не использоваться.

- Если создать **небольшую** таблицу, то длина цепочек будет расти (с увеличением времени поиска).

11. Что такое рехеширование? Назовите критерий необходимости рехеширования.

- **Рехеширование** — это процесс изменения размера хеш-таблицы и пересчёта хешей для всех её элементов. Рехеширование необходимо, когда коэффициент нагрузки становится слишком высоким, что может привести к большому количеству коллизий и, как следствие, снижению производительности.

**Критерий**: Обычно рехеширование выполняется, когда коэффициент нагрузки превышает пороговое значение (например, 0,75)

12. В чём заключается идея хеширования с открытой адресацией?

- **Хеширование с открытой адресацией** заключается в хранении всех элементов непосредственно в самой хеш-таблице (в отличие от цепного хеширования). Когда возникает коллизия, элемент не помещается в связанный список, а ищется другая свободная ячейка в таблице с помощью методов пробирования (линейное, квадратичное пробирование или двойное хеширование).

# **Задание №2**

## **3.1 Постановка задачи**

## 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.

|  |  |  |
| --- | --- | --- |
| № | Тип хеш-таблицы (метод разрешения коллизии) | Структура записи двоичного файла |
| 3 | С открытой адресацией (двойное хеширование) | Владелец телефона: номер телефона – последовательность символов, адрес, ФИО. |

## 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.

## **3.2 Описание алгоритмов операции**

1. Вставка ключа в хеш-таблицу (функция insert)

- Вычисляется хеш ключа с использованием хеш-функции, которая возвращает индекс ячейки для размещения элемента. В данном примере хеш-функция использует стандартную хеш-функцию для строк hash<string>{} и вычисляет индекс как остаток от деления хеша на размер таблицы.

- Если ячейка по этому индексу уже занята, используется метод линейного пробирования для поиска следующей свободной ячейки.

- В случае, если коэффициент заполнения таблицы (loadFactor) превышает порог, выполняется рехеширование: таблица увеличивается в два раза, а все существующие элементы заново распределяются по новой таблице.

- Новый элемент добавляется в первую свободную или удалённую ячейку

|  |
| --- |
| void insert(const string &key, int fileIndex) {          if (numElements / (double)tableSize >= loadFactor) {              rehash(); // Выполняем рехеширование при превышении коэффициента нагрузки          }          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty() && !table[index].isDeleted) {              // Обнаружена коллизия, используем линейное пробирование (смещение на 1)              index = (originalIndex + ++step) % tableSize;          }          table[index] = HashTableEntry(key, fileIndex); // Вставляем новый элемент          numElements++;          cout << "Inserted key '" << key << "' with index " << fileIndex << " into the hash table (position: " << index << ")\n";      } |

2. Поиск записи по ключу в таблице (функция find)

Алгоритм поиска ключа в таблице:

1. Вычисляется хеш ключа для получения начального индекса.
2. Если элемент с таким ключом находится в ячейке и он не был удалён, возвращается его индекс в словаре.
3. Если ячейка занята другим элементом или помечена как удалённая, выполняется пробирование (линейное смещение по таблице), пока не будет найден нужный элемент или не обойдётся вся таблица.
4. Если элемент не найден, возвращается -1.

|  |
| --- |
| int find(const string &key) {          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  return table[index].fileIndex; // Возвращаем индекс записи в словаре              }              index = (originalIndex + ++step) % tableSize; // Пробируем дальше              if (index == originalIndex) { // Если обошли всю таблицу                  break;              }          }          return -1; // Ключ не найден      } |

3. Удаление элемента из хеш-таблицы (функция remove)

-Алгоритм удаления:

1. Вычисляется хеш ключа, и выполняется поиск элемента по тому же алгоритму, что и при поиске.
2. Если элемент найден, он помечается как удалённый (с помощью флага isDeleted), но не удаляется физически. Это позволяет избежать проблем с пробированием, когда после удаления элемента поиск мог бы "прерываться" на пустой ячейке.
3. Если элемент не найден, выводится сообщение, что удаление невозможно.

|  |
| --- |
| void remove(const string &key) {          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  table[index].isDeleted = true; // Помечаем элемент как удалённый                  numElements--;                  cout << "Removed key '" << key << "' from the hash table (position: " << index << ")\n";                  return;              }              index = (originalIndex + ++step) % tableSize; // Пробируем дальше              if (index == originalIndex) {                  break;              }          }          cout << "Key '" << key << "' not found for removal.\n";      } |

4. Рехеширование (функция rehash)

- Алгоритм рехеширования увеличивает размер таблицы вдвое, после чего все элементы, не помеченные как удалённые, заново распределяются по новой таблице на основе их хешей. Рехеширование вызывается автоматически при достижении порогового значения коэффициента загрузки.

|  |
| --- |
| void rehash() {          cout << "Rehashing the table...\n";          vector<HashTableEntry> oldTable = table; // Сохраняем старую таблицу          tableSize \*= 2;                          // Увеличиваем размер в 2 раза          table = vector<HashTableEntry>(tableSize); // Создаём новую таблицу большего размера          numElements = 0;          for (const auto &entry : oldTable) {              if (!entry.key.empty() && !entry.isDeleted) {                  insert(entry.key, entry.fileIndex); // Переносим элементы в новую таблицу              }          }      } |

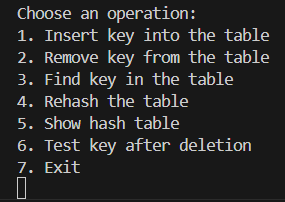
## **3.3 Код программы**

*Листинг 1 – Исходный код программы для Задания 2*

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <functional>  using namespace std;  struct DictionaryEntry { // Структура записи словаря      string englishWord;      string russianWord;      DictionaryEntry() : englishWord(""), russianWord("") {}      DictionaryEntry(string eng, string rus) : englishWord(eng), russianWord(rus) {}  };  struct HashTableEntry { // Структура элемента хеш-таблицы      string key;         // Английское слово (ключ)      int fileIndex;      // Индекс записи в словаре      bool isDeleted;     // Флаг, указывающий на удаление записи      HashTableEntry() : key(""), fileIndex(-1), isDeleted(false) {}      HashTableEntry(string k, int idx) : key(k), fileIndex(idx), isDeleted(false) {}  };  class HashTable { // Класс хеш-таблицы с открытой адресацией  private:      vector<HashTableEntry> table; // Массив для хранения хеш-таблицы      int tableSize;                // Размер таблицы      int numElements;              // Количество элементов в таблице      const double loadFactor = 0.75; // Коэффициент нагрузки      // Хеш-функция для строк      int hashFunction(const string &key) {          return hash<string>{}(key) % tableSize; // Используем стандартную hash-функцию C++      }      // Рехеширование таблицы при переполнении      void rehash() {          cout << "Rehashing the table...\n";          vector<HashTableEntry> oldTable = table; // Сохраняем старую таблицу          tableSize \*= 2;                          // Увеличиваем размер в 2 раза          table = vector<HashTableEntry>(tableSize); // Создаём новую таблицу большего размера          numElements = 0;          for (const auto &entry : oldTable) {              if (!entry.key.empty() && !entry.isDeleted) {                  insert(entry.key, entry.fileIndex); // Переносим элементы в новую таблицу              }          }      }  public:      // Конструктор хеш-таблицы      HashTable(int size = 4) : tableSize(size), numElements(0) {          table.resize(tableSize);      }      // Вставка ключа в хеш-таблицу      void insert(const string &key, int fileIndex) {          if (numElements / (double)tableSize >= loadFactor) {              rehash(); // Выполняем рехеширование при превышении коэффициента нагрузки          }          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty() && !table[index].isDeleted) {              // Обнаружена коллизия, используем линейное пробирование (смещение на 1)              index = (originalIndex + ++step) % tableSize;          }          table[index] = HashTableEntry(key, fileIndex); // Вставляем новый элемент          numElements++;          cout << "Inserted key '" << key << "' with index " << fileIndex << " into the hash table (position: " << index << ")\n";      }      // Поиск ключа в хеш-таблице      int find(const string &key) {          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  return table[index].fileIndex; // Возвращаем индекс записи в словаре              }              index = (originalIndex + ++step) % tableSize; // Пробируем дальше              if (index == originalIndex) { // Если обошли всю таблицу                  break;              }          }          return -1; // Ключ не найден      }      // Удаление ключа из хеш-таблицы      void remove(const string &key) {          int index = hashFunction(key);   // Вычисляем хеш ключа          int originalIndex = index;       // Сохраняем начальный индекс для проверки          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  table[index].isDeleted = true; // Помечаем элемент как удалённый                  numElements--;                  cout << "Removed key '" << key << "' from the hash table (position: " << index << ")\n";                  return;              }              index = (originalIndex + ++step) % tableSize; // Пробируем дальше              if (index == originalIndex) {                  break;              }          }          cout << "Key '" << key << "' not found for removal.\n";      }      // Вывод содержимого хеш-таблицы      void printTable() {          cout << "\nCurrent hash table:\n";          for (int i = 0; i < tableSize; i++) {              if (table[i].key.empty()) {                  cout << "Cell " << i << ": empty\n";              } else if (table[i].isDeleted) {                  cout << "Cell " << i << ": deleted\n";              } else {                  cout << "Cell " << i << ": key = '" << table[i].key << "', index in dictionary = " << table[i].fileIndex << "\n";              }          }      }      // Метод для рехеширования, вызываемый из меню      void manualRehash() {          rehash();          cout << "Manual rehash completed.\n";      }  };  // Функция для меню  void showMenu() {      cout << "\nChoose an operation:\n";      cout << "1. Insert key into the table\n";      cout << "2. Remove key from the table\n";      cout << "3. Find key in the table\n";      cout << "4. Rehash the table\n";      cout << "5. Show hash table\n";      cout << "6. Test key after deletion\n"; // Новый пункт меню для теста      cout << "7. Exit\n";  }  int main() {      vector<DictionaryEntry> dictionary;  // Создаем вектор для хранения словаря      HashTable hashTable;      int choice;      string key;      bool exit = false;      while (!exit) {          showMenu();          cin >> choice;          switch (choice) {              case 1: {                  cout << "Enter the English word to insert: ";                  cin >> key;                  //используем индекс как порядковый номер записи в словаре                  int index = dictionary.size();                  string russianTranslation;                  cout << "Enter the Russian translation: ";                  cin >> russianTranslation;                  dictionary.push\_back(DictionaryEntry(key, russianTranslation));                  hashTable.insert(key, index);                  break;              }              case 2: {                  cout << "Enter the English word to remove: ";                  cin >> key;                  hashTable.remove(key);                  break;              }              case 3: {                  cout << "Enter the English word to find: ";                  cin >> key;                  int fileIndex = hashTable.find(key);                  if (fileIndex != -1) {                      cout << "Found word '" << key << "': translation = " << dictionary[fileIndex].russianWord << "\n";                  } else {                      cout << "Word '" << key << "' not found\n";                  }                  break;              }              case 4: {                  hashTable.manualRehash();                  break;              }              case 5: {                  hashTable.printTable();                  break;              }              case 6: {              // Тест: вставка двух ключей с одинаковым хешем и удаление одного из них              string key1, key2;                cout << "Enter the first English word (key1) to insert: ";              cin >> key1;              cout << "Enter the Russian translation for " << key1 << ": ";              string translation1;              cin >> translation1;              dictionary.push\_back(DictionaryEntry(key1, translation1));              hashTable.insert(key1, dictionary.size() - 1);              cout << "Enter the second English word (key2) to insert (with potential collision): ";              cin >> key2;              cout << "Enter the Russian translation for " << key2 << ": ";              string translation2;              cin >> translation2;              dictionary.push\_back(DictionaryEntry(key2, translation2));              hashTable.insert(key2, dictionary.size() - 1);              // Удаляем первый ключ              cout << "Removing '" << key1 << "'...\n";              hashTable.remove(key1);              // Проверяем, что второй ключ всё ещё можно найти              cout << "Trying to find '" << key2 << "'...\n";              int fileIndex = hashTable.find(key2);              if (fileIndex != -1) {                  cout << "Found word '" << key2 << "': translation = " << dictionary[fileIndex].russianWord << "\n";              } else {                  cout << "Word '" << key2 << "' not found\n";              }              break;  }              case 7: {                  exit = true;                  break;              }              default:                  cout << "Invalid choice. Please try again.\n";          }      }      return 0;  } |

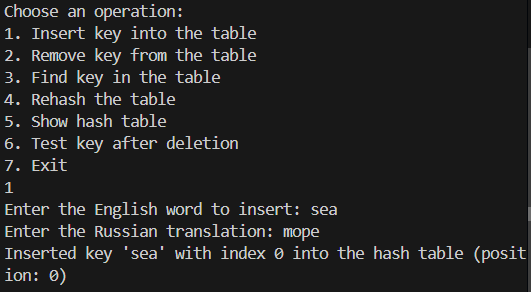
## 

## **3.4 Тестирование программы**



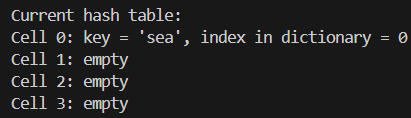
**Рисунок 1. Меню программы**

Вставка ключа в таблицу



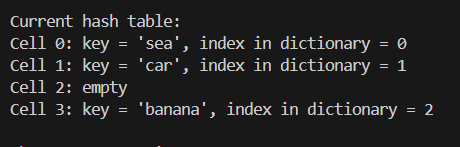
**Рисунок 2. Вставка ключа в таблицу**

Таблица



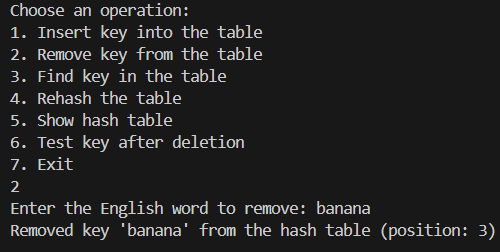
**Рисунок 3. Таблица**

Добавление еще 2 ключей



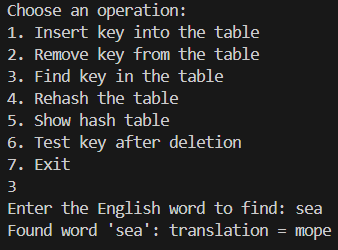
**Рисунок 4. Таблица c 3 элементами**

Удаление ключа



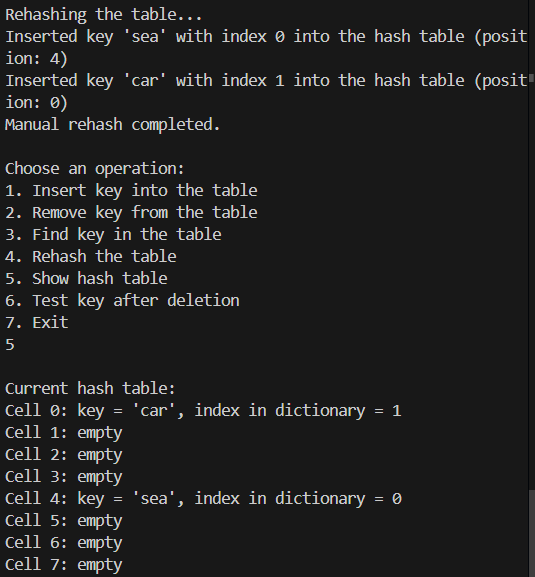
**Рисунок 5. Ключа**

Поиск ключа



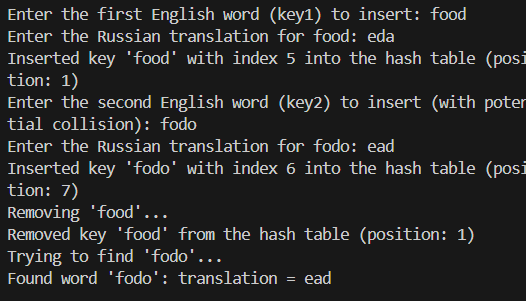
**Рисунок 5. Поиск ключа**

Рехеширование



**Рисунок 6. Рехеширование**

Тестирование после удаления элемента



**Рисунок 7. Тестирование после удаления**

# **Задание №3**

## **4.1 Постановка задачи**

## 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.

## **4.2 Код программы**

*Листинг 3 – Исходный код программы для Задания 3*

|  |
| --- |
| #include <iostream>  #include <vector>  #include <string>  #include <functional>  #include <fstream>  #include <chrono>  using namespace std;  using namespace std::chrono;  // Структура записи словаря  struct DictionaryEntry {      string englishWord;      string russianWord;      DictionaryEntry() : englishWord(""), russianWord("") {}      DictionaryEntry(string eng, string rus) : englishWord(eng), russianWord(rus) {}  };  // Структура элемента хеш-таблицы  struct HashTableEntry {      string key; // Английское слово (ключ)      int fileIndex; // Индекс записи в словаре      bool isDeleted; // Флаг, указывающий на удаление записи      HashTableEntry() : key(""), fileIndex(-1), isDeleted(false) {}      HashTableEntry(string k, int idx) : key(k), fileIndex(idx), isDeleted(false) {}  };  // Класс хеш-таблицы с открытой адресацией  class HashTable {  private:      vector<HashTableEntry> table;      int tableSize;      int numElements;      const double loadFactor = 0.75;      int hashFunction(const string &key) {          return hash<string>{}(key) % tableSize;      }      void rehash() {          cout << "Rehashing the table...\n";          vector<HashTableEntry> oldTable = table;          tableSize \*= 2;          table = vector<HashTableEntry>(tableSize);          numElements = 0;          for (const auto &entry : oldTable) {              if (!entry.key.empty() && !entry.isDeleted) {                  insert(entry.key, entry.fileIndex);              }          }      }  public:      HashTable(int size = 4) : tableSize(size), numElements(0) {          table.resize(tableSize);      }      void insert(const string &key, int fileIndex) {          if (numElements / (double)tableSize >= loadFactor) {              rehash();          }          int index = hashFunction(key);          int originalIndex = index;          int step = 0;          while (!table[index].key.empty() && !table[index].isDeleted) {              index = (originalIndex + ++step) % tableSize;          }          table[index] = HashTableEntry(key, fileIndex);          numElements++;          cout << "Inserted key '" << key << "' with index " << fileIndex << " into the hash table (position: " << index << ")\n";      }      int find(const string &key) {          int index = hashFunction(key);          int originalIndex = index;          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  return table[index].fileIndex;              }              index = (originalIndex + ++step) % tableSize;              if (index == originalIndex) {                  break;              }          }          return -1;      }      void remove(const string &key) {          int index = hashFunction(key);          int originalIndex = index;          int step = 0;          while (!table[index].key.empty()) {              if (table[index].key == key && !table[index].isDeleted) {                  table[index].isDeleted = true;                  numElements--;                  cout << "Removed key '" << key << "' from the hash table (position: " << index << ")\n";                  return;              }              index = (originalIndex + ++step) % tableSize;              if (index == originalIndex) {                  break;              }          }          cout << "Key '" << key << "' not found for removal.\n";      }      void printTable() {          cout << "\nCurrent hash table:\n";          for (int i = 0; i < tableSize; i++) {              if (table[i].key.empty()) {                  cout << "Cell " << i << ": empty\n";              } else if (table[i].isDeleted) {                  cout << "Cell " << i << ": deleted\n";              } else {                  cout << "Cell " << i << ": key = '" << table[i].key << "', index in dictionary = " << table[i].fileIndex << "\n";              }          }      }      void manualRehash() {          rehash();          cout << "Manual rehash completed.\n";      }  };  // Функция для записи тестовых данных в файл  void writeTestDictionaryToFile(const string& filename) {      ofstream file(filename);      file << "apple:яблоко\n";      file << "banana:банан\n";      file << "orange:апельсин\n";      file << "grape:виноград\n";      file << "lemon:лимон\n";      file << "lemonade:лимонад\n";      file << "kiwi:киви\n";      file.close();      cout << "Test dictionary written to file: " << filename << endl;  }  void loadDictionaryFromFile(const string& filename, HashTable& hashTable, vector<DictionaryEntry>& dictionary) {      ifstream file(filename);      string line;      int index = 0;      while (getline(file, line)) {          size\_t pos = line.find(':');          if (pos != string::npos) {              string englishWord = line.substr(0, pos);              string russianWord = line.substr(pos + 1);              dictionary.push\_back(DictionaryEntry(englishWord, russianWord));              hashTable.insert(englishWord, index++);          }      }      file.close();  }  // Функция для записи большого тестового словаря  void writeLargeDictionaryToFile(const string& filename, int numEntries = 10000) {      ofstream file(filename);      for (int i = 0; i < numEntries; ++i) {          file << "word" << i << ":слово" << i << "\n";      }      file.close();      cout << "Large dictionary with " << numEntries << " entries written to file: " << filename << endl;  }  // Функция для загрузки словаря в хеш-таблицу  void loadLargeDictionaryFromFile(const string& filename, HashTable& hashTable, vector<DictionaryEntry>& dictionary) {      ifstream file(filename);      string line;      int index = 0;      while (getline(file, line)) {          size\_t pos = line.find(':');          if (pos != string::npos) {              string englishWord = line.substr(0, pos);              string russianWord = line.substr(pos + 1);              dictionary.push\_back(DictionaryEntry(englishWord, russianWord));              hashTable.insert(englishWord, index++);          }      }      file.close();  }  // Функция для тестирования времени доступа  void testAccessTime(HashTable& hashTable, vector<DictionaryEntry>& dictionary) {      string firstKey = "word0"; // первая запись      string middleKey = "word5000"; // средняя запись      string lastKey = "word9999"; // последняя запись      auto start = high\_resolution\_clock::now();      int index = hashTable.find(firstKey);      auto end = high\_resolution\_clock::now();      auto duration = duration\_cast<microseconds>(end - start).count();      if (index != -1) {          cout << "Access time for the first entry (" << firstKey << "): " << duration << " microseconds\n";      } else {          cout << "First entry not found\n";      }      start = high\_resolution\_clock::now();      index = hashTable.find(middleKey);      end = high\_resolution\_clock::now();      duration = duration\_cast<microseconds>(end - start).count();      if (index != -1) {          cout << "Access time for the middle entry (" << middleKey << "): " << duration << " microseconds\n";      } else {          cout << "Middle entry not found\n";      }      start = high\_resolution\_clock::now();      index = hashTable.find(lastKey);      end = high\_resolution\_clock::now();      duration = duration\_cast<microseconds>(end - start).count();      if (index != -1) {          cout << "Access time for the last entry (" << lastKey << "): " << duration << " microseconds\n";      } else {          cout << "Last entry not found\n";      }  }  // Меню программы  void showMenu() {      cout << "\nChoose an operation:\n";      cout << "1. Insert key into the table\n";      cout << "2. Remove key from the table\n";      cout << "3. Find key in the table\n";      cout << "4. Rehash the table\n";      cout << "5. Show hash table\n";      cout << "6. Save dictionary to binary file\n";      cout << "7. Load dictionary from binary file\n";      cout << "8. Test small dictionary\n";      cout << "9. Exit\n";      cout << "10. Test access time\n"; // Новый пункт меню  }  int main() {      vector<DictionaryEntry> dictionary;      HashTable hashTable;      int choice;      string key, filename = "dictionary.bin";      bool exit = false;      while (!exit) {          showMenu();          cin >> choice;          switch (choice) {              case 1: {                  cout << "Enter the English word to insert: ";                  cin >> key;                  int index = dictionary.size();                  string russianTranslation;                  cout << "Enter the Russian translation: ";                  cin >> russianTranslation;                  dictionary.push\_back(DictionaryEntry(key, russianTranslation));                  hashTable.insert(key, index);                  break;              }              case 2: {                  cout << "Enter the English word to remove: ";                  cin >> key;                  hashTable.remove(key);                  break;              }              case 3: {                  cout << "Enter the English word to find: ";                  cin >> key;                  int fileIndex = hashTable.find(key);                  if (fileIndex != -1) {                      cout << "Found word '" << key << "': translation = " << dictionary[fileIndex].russianWord << "\n";                  } else {                      cout << "Word '" << key << "' not found\n";                  }                  break;              }              case 4: {                  hashTable.manualRehash();                  break;              }              case 5: {                  hashTable.printTable();                  break;              }              case 6: {                  writeTestDictionaryToFile(filename);                  break;              }              case 7: {                  dictionary.clear();                  loadDictionaryFromFile(filename, hashTable, dictionary);                  cout << "Loaded dictionary from file.\n";                  break;              }              case 8: {                  writeTestDictionaryToFile("test\_dictionary.txt");                  loadDictionaryFromFile("test\_dictionary.txt", hashTable, dictionary);                  break;              }              case 9: {                  exit = true;                  break;              }              case 10: { // Новый кейс для тестирования времени доступа                  const string largeFilename = "large\_dictionary.txt";                  writeLargeDictionaryToFile(largeFilename);                  loadLargeDictionaryFromFile(largeFilename, hashTable, dictionary);                  testAccessTime(hashTable, dictionary);                  break;              }              default:                  cout << "Invalid choice. Please try again.\n";          }      }  } |

## **4.3 Тестирование программы**

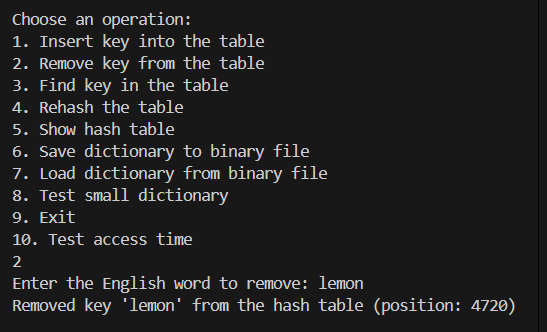
## Тестирование программы на 10 000 записях

## 

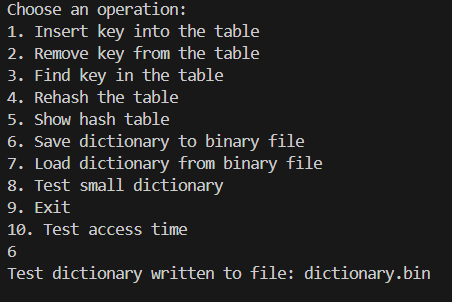
**Рисунок 8. Тестирование на 10 000 записях**

## 

**Рисунок 9. Тестирование функциональности**



**Рисунок 10. Тестирование после удаления**



**Рисунок 11. Тестирование записи двоичного файла**

# **5. Вывод**

В ходе выполнения задания я разработал приложение, использующее хеш-таблицу для организации прямого доступа к записям в файле. Я создал структуру элементов таблицы и протестировал хеш-функцию, чтобы убедиться в её корректности. Реализованные операции, такие как вставка, удаление и поиск ключей, были тщательно протестированы на различных сценариях, включая случаи с коллизиями. Тестирование показало стабильное время доступа к данным, что подтвердило эффективность выбранного подхода в управлении динамическим множеством записей.