МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания №6

Тема: АЛГОРИТМЫ ПОИСКА

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент: Павлов Никита
группа ИКБО-50-23

Москва 2024

СОДЕРЖАНИЕ

1. БЫСТРЫЙ ДОСТУП К ДАННЫМ С ПОМОЩЬЮ ХЕШ-ТАБЛИЦ

1.1 ЦЕЛЬ РАБОТЫ

Освоить приёмы хеширования и эффективного поиска элементов множества.

Персональный вариант: 24

Тип хеширования: Цепное хеширование

Структура записи файла: Товар: код – шестиразрядное число, название, цена

1.2 ЗАДАНИЕ

1.2.1 Формулировка задачи

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

Приложение должно содержать класс с базовыми операциями: вставки, удаления, поиска по ключу, вывода. Включите в класс массив полезных данных и хеш-таблицу. Хеш-функцию подберите самостоятельно, используя правила выбора функции.

Реализуйте расширение размера таблицы и рехеширование, когда это требуется, в соответствии с типом разрешения коллизий.

Предусмотрите автоматическое заполнение таблицы 5-7 записями.

Реализуйте текстовый командный интерфейс пользователя для возможности вызова методов в любой произвольной последовательности, сопроводите вывод достаточными для понимания происходящего сторонним пользователем подсказками.

1.2.2 Математическая модель решения (описание алгоритма)

1. hash(long long key):

- Вход: Ключ (ключ) типа `long long`.

- Операция: Вычислить хеш-индекс, выполняя деление ключа на емкость хеш-таблицы и беря остаток.

- Выход: Возвращает индекс (целое число), по которому будет храниться элемент для данного ключа.

2. resize():

- Вход: Нет.

- Операция:

  - Увеличить емкость хеш-таблицы в два раза.

- Создать новую таблицу (новый вектор), в которой будет храниться элемент типа `Product` с новой емкостью.

- Перебрать все списки в текущей таблице:

- Для каждого продукта в списке вычислить новый индекс с помощью метода `hash`.

- Добавить продукт в новый список по вычисленному индексу.

- Замена старой таблицы на новую.

- Выход: Нет.

3. HashTable(int initialCapacity):

- Вход: Начальная емкость хеш-таблицы (целое число).

- Операция:

- Инициализировать емкость хеш-таблицы значением `initialCapacity`.

- Инициализировать размер (количество элементов) как ноль.

- Создать таблицу (вектор списков) соответствующей емкости.

- Выход: Новый объект `HashTable`.

4. insert(const Product& product):

- Вход: Объект `Product`, который необходимо вставить.

- Операция:

- Вычислить индекс в таблице для данного продукта с помощью метода `hash`.

- Добавить продукт в связный список по этому индексу.

- Увеличить размер хеш-таблицы на единицу.

- Если текущий размер превышает половину емкости, вызвать метод `resize`.

- Выход: Нет.

5. remove(long long key):

- Вход: Ключ (ключ) типа `long long` для удаления.

- Операция:

- Вычислить индекс в таблице для данного ключа с помощью метода `hash`.

- Инициализировать итератор для списка по этому индексу.

- Перебрать все элементы в списке:

- Если текущий продукт имеет код, равный ключу, удалить его из списка и уменьшить размер на единицу, затем завершить выполнение.

- Перейти к следующему продукту.

- Выход: Нет.

6. find(long long key):

- Вход: Ключ (ключ) типа `long long` для поиска.

- Операция:

- Вычислить индекс в таблице для данного ключа с помощью метода `hash`.

- Перебрать все продукты в списке по этому индексу:

- Если продукт имеет код, равный ключу, вернуть указатель на этот продукт.

- Если продукт не найден, вернуть `nullptr`.

- Выход: Указатель на `Product` или `nullptr`, если продукт не найден.

7. display():

- Вход: Нет.

- Операция:

- Вывести заголовок "Хеш-таблица".

- Перебрать каждый индекс от 0 до емкости:

- Вывести индекс.

- Если связный список по этому индексу пуст, вывести "Пусто".

- Иначе, перебрать все продукты в списке и вывести их код, имя и цену.

- Выход: Нет.

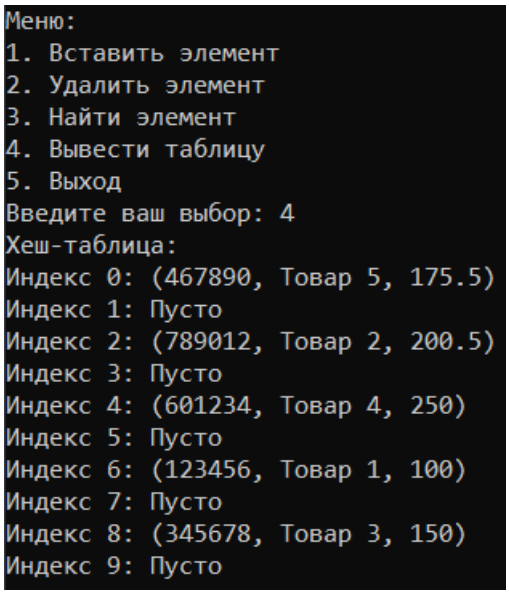
1.2.3 Код программы

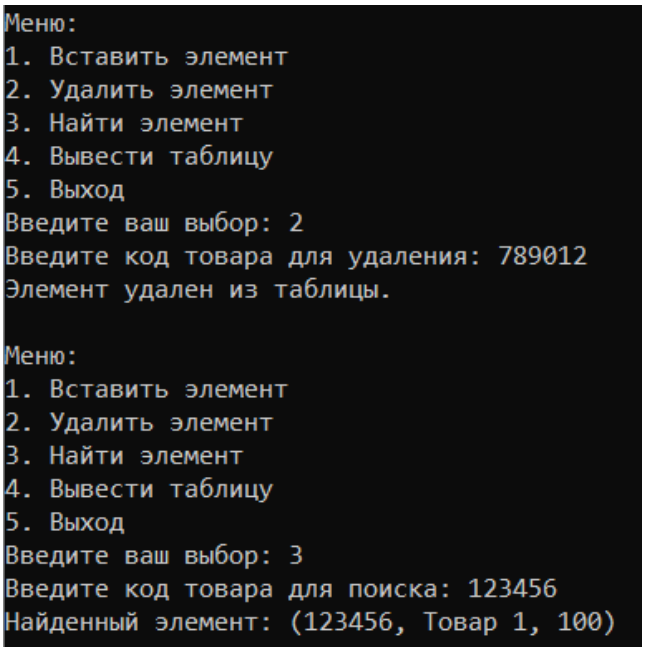
```cpp
int HashTable::hash(long long key) {
    return key % capacity;
}

void HashTable::resize() {
    capacity *= 2;
    vector<list<Product>> newTable(capacity);
    for (auto& list : table) {
        for (auto& product : list) {
            int index = hash(product.code);
            newTable[index].push_back(product);
        }
    }
    table = newTable;
}

HashTable::HashTable(int initialCapacity) : capacity(initialCapacity), size(0) {
    table.resize(capacity);
}

void HashTable::insert(const Product& product) {
    int index = hash(product.code);
    table[index].push_back(product);
    size++;
    /*if (size > capacity / 2) {
        resize();
    }*/
}
```

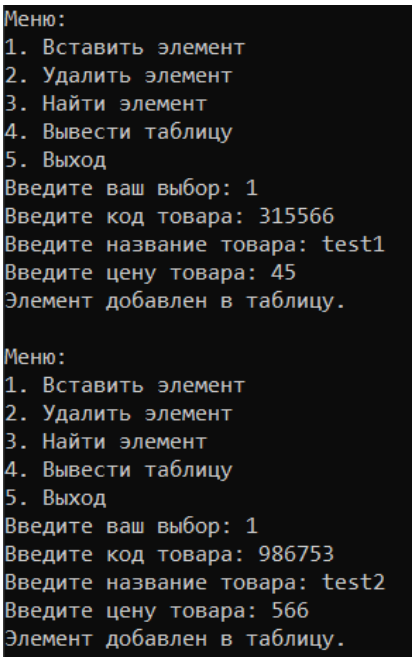
```cpp
void HashTable::remove(long long key) {
    int index = hash(key);
    auto it = table[index].begin();
    while (it != table[index].end()) {
        if (it->code == key) {
            table[index].erase(it);
            size--;
            return;
        }
        it++;
    }
}

Product* HashTable::find(long long key) {
    int index = hash(key);
    for (auto& product : table[index]) {
        if (product.code == key) {
            return &product;
        }
    }
    return nullptr;
}

void HashTable::display() {
    cout << "Хеш-таблица:\n";
    for (int i = 0; i < capacity; i++) {
        cout << "Индекс " << i << ": ";
        if (table[i].empty()) {
            cout << "Пусто\n";
        }
        else {
            for (auto& product : table[i]) {
                cout << "(" << product.code << ", " << product.name << ", " << product.price << ") ";
            }
            cout << "\n";
        }
    }
}
```

Рисунок 1-2 – Код программы

1.2.4 Результаты тестирования







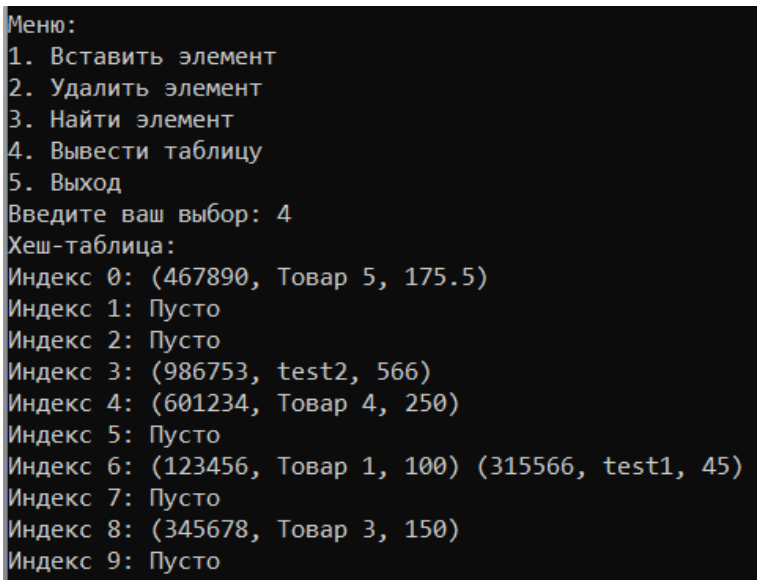


Рисунок 3-6 – Результаты тестирования

1.3 ВЫВОДЫ

Цель работы и сопутствующие задачи были успешно выполнены. Освоены приёмы хеширования и эффективного поиска элементов множества, реализован эффективный алгоритм цепного хеширования.

2. ПОИСК ОБРАЗЦА В ТЕКСТЕ

2.1 ЦЕЛЬ РАБОТЫ

Освоить приёмы реализации алгоритмов поиска образца в тексте

Персональный вариант: 24

Задание 1: Дано предложение, слова в котором разделены пробелами и запятыми. Распечатать те пары слов, расстояние между которыми наименьшее. Расстояние – это количество позиций, в которых слова различаются. Например, МАМА и ПАПА, МЫШКА и КОШКА расстояние этих пар равно двум.

Задание 2: Найти все вхождения подстроки в строку, используя алгоритм Бойера-Мура-Хорспула.

2.2 ЗАДАНИЕ 1

2.2.1 Постановка задачи

Разработайте приложение в соответствии с заданием в индивидуальном варианте.

2.2.2 Математическая модель решения

Алгоритм функции `distance`

1. Инициализация переменной расстояния:

- Создайте целочисленную переменную `dist` и присвойте ей значение 0. Эта переменная будет использоваться для подсчета расстояния между двумя словами.

2. Сравнение букв слов:

- Пройдите по каждому символу обоих слов до тех пор, пока не достигнете конца меньшего из них.

- Для каждого символа сравните символы, находящиеся на соответствующих позициях в обоих словах:

- Если символы различаются, увеличьте `dist` на 1.

3. Учет длины слов:

- После сравнения символов, добавьте к `dist` разницу в длине между двумя словами. Это позволит учесть символы, которые есть только в одном из слов.

4. Возврат результата:

- Верните значение переменной `dist`, которое теперь содержит общее расстояние между двумя словами.

Алгоритм функции `find_closest_pairs`

1. Разделение предложения на слова:

- Создайте пустой вектор `words` для хранения слов.

- Создайте строку `current_word`, чтобы накапливать текущие символы слова.

- Пройдите по каждому символу в строке `sentence`:

- Если символ является пробелом или запятой, проверьте, пустая ли `current_word`. Если нет, добавьте `current_word` в `words` и очистите `current_word`.

- Если символ не является пробелом или запятой, добавьте его к `current_word`.

- После завершения цикла проверьте, остались ли символы в `current_word`. Если да, добавьте их в `words`.

2. Проверка количества слов:

- Если в векторе `words` меньше двух слов, выведите сообщение о недостаточном количестве слов и завершите выполнение функции.

3. Поиск пар слов с минимальным расстоянием:

- Инициализируйте `min_dist` значением `INT_MAX`, чтобы отслеживать минимальное расстояние между парами слов.

- Создайте пустой вектор `closest_pairs` для хранения пар слов с минимальным расстоянием.

- Пройдите по всем парам слов в векторе `words` с помощью вложенных циклов:

- Для каждой пары слов вызовите функцию `distance`, чтобы получить расстояние между ними.

- Если полученное расстояние меньше текущего `min_dist`:

- Обновите `min_dist` до нового значения и очистите `closest_pairs`, добавив в него текущую пару слов.

- Если расстояние равно текущему `min_dist`, добавьте пару слов в `closest_pairs`.

4. Вывод результата:

- Если в `closest_pairs` есть пары, выведите их с минимальным расстоянием.

- Если нет, выведите сообщение о том, что пар слов с минимальным расстоянием нет.

2.2.3 Код программы

```
int distance(const string& word1, const string& word2) {
    int dist = 0;
    for (int i = 0; i < min(word1.size(), word2.size()); i++) {
        if (word1[i] != word2[i]) {
            dist++;
        }
    }
    dist += abs(static_cast<int>(word1.size()) - static_cast<int>(word2.size()));
    return dist;
}

void find_closest_pairs(const string& sentence) {
    vector<string> words;
    string current_word;
    for (char c : sentence) {
        if (c == ' ' || c == ',') {
            if (!current_word.empty()) {
                words.push_back(current_word);
                current_word.clear();
            }
        }
        else {
            current_word += c;
        }
    }
    if (!current_word.empty()) {
        words.push_back(current_word);
    }

    if (words.size() < 2) {
        cout << "В предложении недостаточно слов.\n";
        return;
    }
    int min_dist = INT_MAX;
    vector<pair<string, string>> closest_pairs;
    for (int i = 0; i < words.size() - 1; i++) {
        for (int j = i + 1; j < words.size(); j++) {
            int dist = distance(words[i], words[j]);
            if (dist < min_dist) {
                min_dist = dist;
                closest_pairs.clear();
                closest_pairs.push_back({ words[i], words[j] });
            }
            else if (dist == min_dist) {
                closest_pairs.push_back({ words[i], words[j] });
            }
        }
    }

    if (!closest_pairs.empty()) {
        cout << "Пары слов с минимальным расстоянием:\n";
        for (auto& pair : closest_pairs) {
            cout << pair.first << " " << pair.second << " (Расстояние: " << min_dist << ")\n";
        }
    }
    else {
        cout << "В предложении нет пар слов с минимальным расстоянием.\n";
    }
}
```

Рисунок 7-8 – Код программы

2.2.4 Результаты тестирования

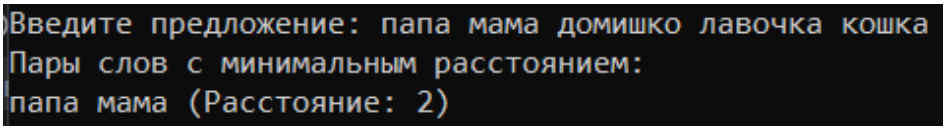


Рисунок 9 – Результаты тестирования

2.3 ЗАДАНИЕ 2

2.3.1 Постановка задачи

Разработайте приложение в соответствии с заданием в индивидуальном варианте.

2.3.2 Математическая модель решения

Алгоритм Бойера-Мура-Хорспула — это эффективный метод поиска подстроки (шаблона) в строке (тексте). Он основан на использовании таблицы сдвигов, чтобы минимизировать количество сравнений при поиске.

1. Подготовка:

• Создается таблица сдвигов, которая показывает, на сколько символов можно сдвинуть шаблон, если происходит несоответствие. Эта таблица заполняется для каждого символа, присутствующего в шаблоне.

2. Поиск:

• Шаблон выравнивается по началу текста.

• Сравнение начинается с последнего символа шаблона и идет к первому.

• Если символы совпадают, продолжается сравнение.

• Если происходит несоответствие, используется таблица сдвигов для определения, насколько можно сдвинуть шаблон вправо.

3. Сдвиг:

• Если символ не найден в таблице, шаблон сдвигается на длину шаблона.

• Если символ найден, шаблон сдвигается на значение, указанное в таблице для этого символа.

4. Повторение:

• Процесс повторяется до тех пор, пока не будет достигнут конец текста.

Пример

• Текст: ABABDABACDABABCABAB

• Шаблон: ABABCAB

1. Создание таблицы сдвигов:

• Длина шаблона: 7.

• Таблица сдвигов (для символов):

• A → 1 (сдвиг на 1, если A не совпадает)

• B → 3

• C → 2

• D → 7 (не встречается в шаблоне)

• Остальные символы → 7

2. Начальный выравнивание:

• Шаблон ABABCAB выровнен по тексту ABABDABACDABABCABAB.

3. Сравнение:

• Сравниваем символы с конца:

• B (шаблон) с B (текст) — совпадает.

• A с A — совпадает.

• C с D — несоответствие.

4. Сдвиг:

• Используем таблицу: для D сдвигаем на 7 (длину шаблона).

• Новый выравненный текст: ABABDABACDABABCABAB.

5. Повторение:

• Повторяем процесс:

• Сравниваем снова, и, когда находим совпадение, записываем индекс начала совпадения.

2.3.3 Код программы

```
vector<int> make_stoptable(string text, string pattern) {
    vector<int> stoptable(256, pattern.size());
    for (int i = 0; i < pattern.size() - 1; i++) {
        stoptable[pattern[i]] = pattern.size() - 1 - i;
    }
    return stoptable;
}
```

```
vector<int> boyer_moore_horspool(string text, string pattern) {
    vector<int> matches;
    vector<int> stoptable = make_stoptable(text, pattern);
    int pos = pattern.size() - 1;
    while (pos <= (text.size() - 1)) {
        bool f = true;
        int dif = 0;
        for (int i = 0; i < pattern.size(); i++) {
            if (pattern[pattern.size() - 1 - i] != text[pos -  i]) {
                f = false;
                dif = i;
                break;
            }
        }
        if (f) {
            matches.push_back(pos - pattern.size() + 1);
            pos++;
        }
        else if (dif == 0) {
            pos += stoptable[text[pos]];
        }
        else {
            pos += stoptable[pattern[pattern.size() - 1 - dif]];
        }
    }
    return matches;
}
```

Рисунок 10-11 – Код программы

2.3.4 Результаты тестирования

```
Введите строку: qwertyqwteryteqwytyrtreyqwrtyqwertytyrty
Введите подстроку: rty
Вхождения подстроки в строку:
3 26 32 37
```

Рисунок 12 – Результаты тестирования

2.4 ВЫВОДЫ

Получен практический опыт по применению алгоритмов поиска образца в тексте различными методами. Проведено тестирование алгоритмов на различных размерах исходных данных и зафиксировано время работы каждого.

СПИСОК ИНФОРМАЦТОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных. Часть 1. Сложность алгоритмов. Сортировки. Линейные структуры данных. Поиск в таблице. : учеб. пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова ; МИРЭА – Российский технологический университет, 2022. – 109 с. – ISBN 978-5-7339-1612-5.

2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01-01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с.