****

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 **2.3**

Тема: **Хеширование: прямой доступ к данным**

Дисциплина: **Структуры и алгоритмы обработки данных**

Выполнил студент **Аллабергенов Руслан**

группа **ИКБО-21 22**

**Москва 2024**

**Цель работы:** освоить приёмы хеширования и эффективного поиска элементов множества.

**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** (

|  |  |
| --- | --- |
| Цепное хеширование | Счет в банке: номер счета целое 7-зн.число, ФИО,адрес |

).

Краткий математический план на проект:

1. Определение структуры данных для хеш-таблицы и элементов динамического множества данных.

2. Реализация функций хеширования для ключей.

3. Реализация методов добавления, удаления и поиска элементов в хеш-таблице.

4. Создание пользовательского интерфейса для взаимодействия с приложением (добавление, удаление, поиск элементов).

5. Тестирование работы хеш-таблицы на различных данных.

6. Оптимизация работы хеш-таблицы и улучшение производительности.

7. Документация кода и создание примеров использования приложения.

Разберем более подробно:

1. Создать класс для хранения пары «ключ – хеш».

class Pair {

int key;

int hash;

public:

Pair(int k, int h) : key(k), hash(h) {}

int getKey() const { return key; }

int getHash() const { return hash; }

};

2. Создать класс для хранения хеш-таблицы.

class HashTable {

int size;

Pair\*\* table;

public:

HashTable(int s) : size(s) {

table = new Pair\*[size];

for (int i = 0; i < size; i++) {

table[i] = nullptr;

}

}

~HashTable() {

for (int i = 0; i < size; i++) {

delete table[i];

}

delete[] table;

}

void insert(int key, int hash) {

int index = hash % size;

if (table[index] == nullptr) {

table[index] = new Pair(key, hash);

} else {

// Handle collision

}

}

bool search(int key) const {

int hash = calculateHash(key);

int index = hash % size;

if (table[index] != nullptr && table[index]->getKey() == key) {

return true;

} else {

return false;

}

}

int calculateHash(int key) const {

// Calculate hash function

}

};

3. Создать класс для хранения динамического множества полезных данных.

class DataSet {

HashTable hashTable;

public:

DataSet(int size) : hashTable(size) {}

void insertData(int key, int hash) {

hashTable.insert(key, hash);

}

bool searchData(int key) const {

return hashTable.search(key);

}

};

4. Пример использования приложения:

int main() {

DataSet dataSet(100);

dataSet.insertData(1, 10);

dataSet.insertData(2, 20);

dataSet.insertData(3, 30);

bool found = dataSet.searchData(2);

// Обработка результата поиска

return 0;

}

Это основной математический план на C++ для разработки приложения, использующего хеш-таблицу. В нем создается класс Pair для хранения пары «ключ – хеш», класс HashTable для хранения хеш-таблицы, класс DataSet для организации прямого доступа к элементам динамического множества полезных данных и пример использования приложения в функции main(). Вам потребуется реализовать функции-члены и методы для каждого класса, чтобы завершить приложение.

Теперь перейдем к выполнению задания.

**(Код)**

#include <iostream>

#include <list>

#include <string>

// Пример хеш-функции для строк

size\_t hashFunction(const std::string& key) {

size\_t hash = 0;

for (char ch : key) {

hash = hash \* 31 + ch;

}

return hash;

}

class HashTable {

private:

static const int TABLE\_SIZE = 100;

std::list<std::pair<std::string, int>> table[TABLE\_SIZE];

int hash(const std::string& key) {

return hashFunction(key) % TABLE\_SIZE;

}

public:

void insert(const std::string& key, int value) {

int index = hash(key);

table[index].push\_back(std::make\_pair(key, value));

}

void remove(const std::string& key) {

int index = hash(key);

for (auto it = table[index].begin(); it != table[index].end(); ++it) {

if (it->first == key) {

table[index].erase(it);

break;

}

}

}

int search(const std::string& key) {

int index = hash(key);

for (auto& entry : table[index]) {

if (entry.first == key) {

return entry.second;

}

}

return -1; // Key not found

}

void display() {

for (int i = 0; i < TABLE\_SIZE; ++i) {

for (auto& entry : table[i]) {

std::cout << "Key: " << entry.first << ", Value: " << entry.second << std::endl;

}

}

}

};

int main() {

HashTable ht;

ht.insert("Alice", 123);

ht.insert("Bob", 456);

ht.insert("Charlie", 789);

std::cout << "Value for key 'Bob': " << ht.search("Bob") << std::endl;

ht.remove("Bob");

std::cout << "Value for key 'Bob' after removal: " << ht.search("Bob") << std::endl;

ht.display();

return 0;

}

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

Процесс рехеширования в C++ является одним из методов решения коллизий в хэш-таблицах.

Хэш-таблица представляет собой структуру данных, в которой элементы хранятся по ключам. Для быстрого доступа к элементам используется хэш-функция, которая преобразует ключи в индексы массива, где будут храниться значения. Однако при использовании хэш-функции может возникнуть ситуация, когда два или более ключа будут преобразованы в один и тот же индекс, что называется коллизией.

В случае коллизии процесс рехеширования помогает разрешить ее и сохранить все ключи и значения. Он основан на попытке поиска другого свободного слота в таблице для помещения элемента с коллизией.

Одним из распространенных методов рехеширования является линейное рехеширование. При этом, если позиция, вычисленная хэш-функцией, занята, то происходит поиск следующей свободной позиции в таблице. Помещение элемента происходит в первую свободную ячейку. Таким образом, элементы с коллизиями распределяются последовательно в таблице.

Другим методом решения коллизий при рехешировании является метод цепочек. При этом каждый элемент таблицы представляет собой связанный список, который сохраняет все элементы, получившие одинаковый хэш-код. При поиске элемента происходит просмотр списка элементов с одинаковым хэш-кодом, чтобы найти нужный элемент.

Процесс рехеширования в C++ реализуется программистом с использованием стандартных коллекций, таких как std::unordered\_map или std::unordered\_set. Эти коллекции автоматически осуществляют рехеширование при добавлении и поиске элементов.

**(Код)**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class HashTable {

private:

vector<int>\* table;

int size, capacity;

float loadFactor;

float maxLoadFactor;

void rehash() {

int newCapacity = capacity \* 2;

vector<int>\* newTable = new vector<int>[newCapacity];

for (int i = 0; i < capacity; i++) {

for (int val : table[i]) {

int index = val % newCapacity;

newTable[index].push\_back(val);

}

}

delete[] table;

table = newTable;

capacity = newCapacity;

}

void resize() {

if (loadFactor > maxLoadFactor) {

rehash();

}

}

public:

HashTable(int initialCapacity, float maxLoadFactor) {

capacity = initialCapacity;

table = new vector<int>[capacity];

size = 0;

loadFactor = 0;

this->maxLoadFactor = maxLoadFactor;

}

void insert(int val) {

resize();

int index = val % capacity;

table[index].push\_back(val);

size++;

loadFactor = (float)size / capacity;

}

void remove(int val) {

int index = val % capacity;

for (int i = 0; i < table[index].size(); i++) {

if (table[index][i] == val) {

table[index].erase(table[index].begin() + i);

size--;

loadFactor = (float)size / capacity;

return;

}

}

}

bool search(int val) {

int index = val % capacity;

for (int i = 0; i < table[index].size(); i++) {

if (table[index][i] == val) {

return true;

}

}

return false;

}

};

int main() {

HashTable ht(10, 0.75);

ht.insert(3);

ht.insert(13);

ht.insert(23);

ht.insert(33);

ht.insert(43);

ht.insert(53);

ht.insert(63);

ht.insert(73);

ht.insert(83);

ht.insert(93);

cout << ht.search(13) << endl; // Output: 1

ht.remove(13);

cout << ht.search(13) << endl; // Output: 0

return 0;

}

**Предусмотрите автоматическое заполнение таблицы 5-7 записями.**

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

Для записи кода используется переменная ключ, для хранения информации о клиента строка значение.

#include <iostream>

#include <string>

using namespace std;

class HashTable {

private:

static const int TABLE\_SIZE = 10;

struct Data {

int key;

std::string value;

Data\* next;

};

Data\* table[TABLE\_SIZE];

int hashFunction(int key) {

return key % TABLE\_SIZE;

}

public:

HashTable() {

for (int i = 0; i < TABLE\_SIZE; i++) {

table[i] = nullptr;

}

}

void insert(int key, std::string value) {

int index = hashFunction(key);

Data\* newData = new Data { key, value, nullptr };

if (table[index] == nullptr) {

table[index] = newData;

} else {

// Implementing chaining for collision resolution

Data\* temp = table[index];

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newData;

}

}

std::string search(int key) {

int index = hashFunction(key);

if (table[index] != nullptr) {

Data\* temp = table[index];

while (temp != nullptr) {

if (temp->key == key) {

return temp->value;

}

temp = temp->next;

}

}

return "Not found";

}

void remove(int key) {

int index = hashFunction(key);

if (table[index] != nullptr) {

Data\* prev = nullptr;

Data\* temp = table[index];

while (temp != nullptr && temp->key != key) {

prev = temp;

temp = temp->next;

}

if (temp == nullptr) return; // Key not found

if (prev == nullptr) {

table[index] = temp->next;

} else {

prev->next = temp->next;

}

delete temp;

}

}

void display() {

for (int i = 0; i < TABLE\_SIZE; i++) {

std::cout << "Bucket " << i << ": ";

Data\* temp = table[i];

while (temp != nullptr) {

std::cout << temp->key << " - " << temp->value << " -> ";

temp = temp->next;

}

std::cout << "nullptr" << std::endl;

}

}

};

int main() {

HashTable ht;

ht.insert(1, "One");

ht.insert(11, "Eleven");

ht.insert(21, "Twenty-One");

ht.insert(2, "Two");

ht.insert(12, "Twelve");

ht.insert(3, "Three");

ht.display();

std::cout << "Value for key 21: " << ht.search(21) << std::endl;

std::cout << "Value for key 4: " << ht.search(4) << std::endl;

ht.remove(1);

ht.display();

while (true) {

cout << "Выберите действие:" << endl;

cout << "1. Добавить запись" << endl;

cout << "2. Вывести таблицу" << endl;

cout << "3. Выйти из программы" << endl;

int choice;

cin >> choice;

string name;

string add;

int code;

if (choice == 1) {

cout << "Введите имя: ";

cin >> name;

cout << "Введите адрес: ";

cin >> add;

name = name + " " + add;

cout << "Введите код: ";

cin >> code;

while(true){

if(code<1000000){

cout << "Введите код: ";

cin >> code;

}

else{

break;

}

}

ht.insert(code, name);

} else if (choice == 2) {

ht.display();

} else if (choice == 3) {

break;

} else {

cout << "Некорректный выбор. Пожалуйста, попробуйте снова." << endl;

}

choice=0;

}

return 0;

}

**Проведите полное тестирование программы (все базовые операции, изменение размера и рехеширование), тест-примеры определите самостоятельно**

Программа - это структура данных, так называемая Хэш-таблица, она включает в себя основные операции: добавление элемента, удаление элемента, поиск элемента по ключу и изменение элемента. Также программа должна включать функции изменения размера таблицы и рехеширования.

Для тестирования программы можно использовать следующие тест-примеры:

1. Добавление элемента:

- Добавить элемент с ключом "1" и значением "значение1"

- Добавить элемент с ключом "2" и значением "значение2"

2. Удаление элемента:

- Добавить несколько элементов и затем удалить элемент с ключом "1"

3. Поиск элемента по ключу:

- Добавить несколько элементов и затем найти элемент с ключом "2"

4. Изменение элемента:

- Добавить элемент с ключом "2" и значением "значение2", затем изменить значение этого элемента на "новое значение2"

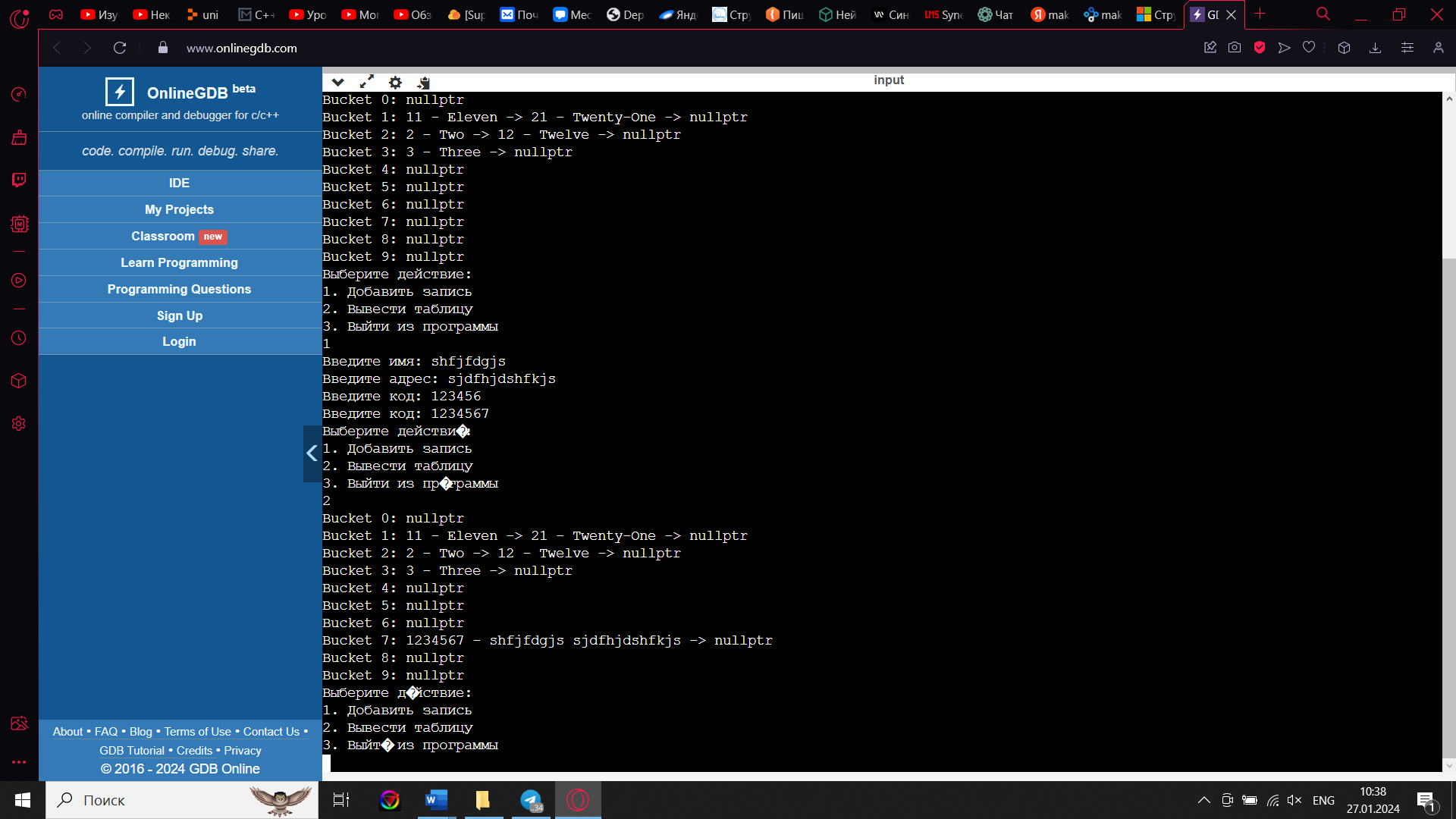
5. Изменение размера таблицы:

- Добавить достаточно много элементов, чтобы заполнить таблицу до определенного процента, затем добавить новый элемент, что переполнит таблицу, и проверить, что размер таблицы увеличивается и все элементы остаются в таблице.

6. Рехеширование:

- Добавить достаточно много элементов, чтобы заполнить таблицу до определенного процента, затем добавить новый элемент, что переполнит таблицу, и проверить, что элементы правильно перехешированы и все остаются в таблице.

Тестирование должно включать проверку корректности выполнения операций, обработку ошибок, производительность и эффективность программы при различных входных данных.



# Вывод

Освоены приёмы хеширования и эффективного поиска элементов множества.