Министерство образования и науки РФ

Новосибирский государственный технический университет

Кафедра систем сбора и обработки данных

**Лабораторная работа №3**

Вариант 15.

Хеш-функции и хеш-таблицы. Применение хеширования.

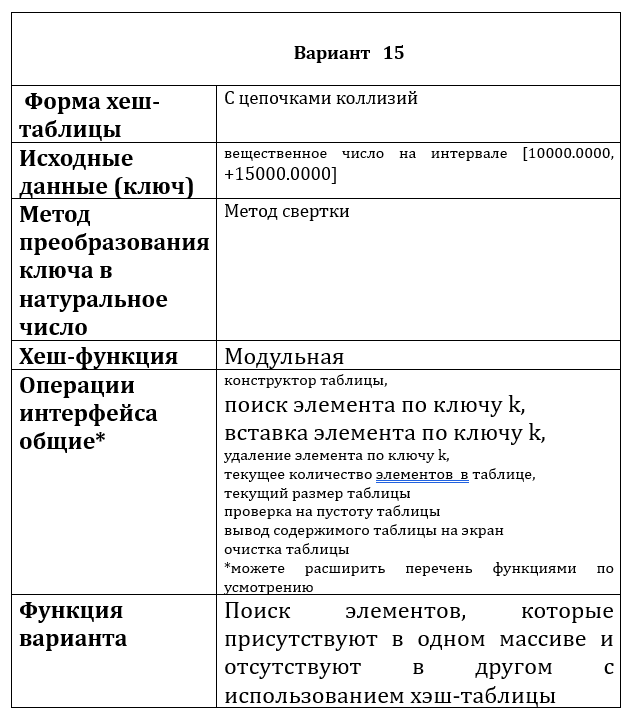
Факультет: АВТФ Преподаватель: Квашнина Е. А.

Группа: АТ-33

Студент: Мельничук В.

Новосибирск, 2025

1. Формулировка задания
2. Спроектируйте и реализуйте абстрактный тип данных «хеш-таблица» для данных в соответствии с вариантом задания.
3. Проведите оценку средней трудоёмкости операций вставки, удаления и поиска элементов в хеш-таблице в зависимости от коэффициента заполнения: 0,5; 0,7; 0,9.



1. Описание Хеш-таблицы

Хеш-таблица реализована с использованием метода цепочек, т.е. в каждой ячейке массива хранится связанный список, куда попадают элементы с одинаковым хешем.

Класс HashTable содержит:

* Массив из 32 указателей на связные списки (Node\* table[SIZE]);
* Поле count — количество элементов в таблице;
* Поле operations — счётчик операций для замеров;
* Поле isEmpty — флаг пустоты таблицы;
* Метод hash() — хеш-функция, преобразующая значение типа double в индекс.

Хеш-функция основана на методе свертки:

int k1 = std::stoi(key.substr(1, 2)) \* 2037;

int k2 = std::stoi(key.substr(3, 2)) \* 1249;

int k3 = std::stoi(key.substr(6, 2)) \* 2174;

int k4 = std::stoi(key.substr(8)) \* 1393;

Сумма k1 + k2 + k3 + k4 затем берётся по модулю SIZE (32). Такая функция стремится равномерно распределить числа в диапазоне.

Основные методы:

* insert(double) — вставка нового узла в начало списка соответствующей цепочки;
* search(double) — линейный поиск по цепочке;
* remove(double) — удаление элемента из цепочки;
* clear() — освобождение всех узлов;
* print() — печать таблицы с отображением всех цепочек.

1. Оценка трудоемкости

**Теоретическая оценка:**

Средний случай (равномерное распределение хеш-функции):

* Вставка: O(1),
* Поиск, удаление: O(1 + α), где α = n / m — коэффициент загрузки (load factor).

Худший случай (все элементы в одной цепочке):

* Вставка: O(1),
* Поиск и удаление: O(n).

**Экспериментальная оценка:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Load Factor** | **Insert (усл. оп./элемент)** | **Search (усл. оп./запрос)** | **Delete (усл. оп./запрос)** |
| 0.5 | 1 | 79 | 40 |
| 0.7 | 1 | 129 | 78 |
| 0.9 | 1 | 154 | 128 |

1. Листинг программы

**Hash.h**

#include <iostream>

#include <string>

#include <iomanip>

#include <cmath>

class HashTable {

private:

    struct Node {

        double value;

        Node\* next;

        Node(double val = -1) : value(val), next(nullptr) {}

    };

    static const int SIZE = 32;

    Node\* table[SIZE];

    int count;

    int operations;

    bool isEmpty;

    int hash(double value) {

        std::string key = std::to\_string(value);

        if (key.find('.') == std::string::npos) {

            key += ".0000";

        } else {

            while (key.length() < 10) key += '0';

        }

        int k1 = std::stoi(key.substr(1, 2)) \* 2037;

        int k2 = std::stoi(key.substr(3, 2)) \* 1249;

        int k3 = std::stoi(key.substr(6, 2)) \* 2174;

        int k4 = std::stoi(key.substr(8)) \* 1393;

        return (k1 + k2 + k3 + k4) % SIZE;

    }

public:

    HashTable() : count(0), isEmpty(true) {

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

            table[i] = new Node();

        }

    }

    ~HashTable() {

        clear();

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

            delete table[i];

        }

    }

    void insert(double key) {

        operations++;

        int index = hash(key);

        Node\* current = table[index];

        Node\* newNode = new Node(key);

        newNode->next = current->next;

        current->next = newNode;

        count++;

        isEmpty = false;

    }

    bool search(double key, Node\*\* prevOut = nullptr) {

        operations++;

        int index = hash(key);

        Node\* temp = table[index];

        Node\* current = temp->next;

        while (current) {

            if (current->value == key) {

                if (prevOut) \*prevOut = temp;

                return true;

            }

            temp = current;

            current = current->next;

            operations++;

        }

        return false;

    }

    void remove(double key) {

        operations++;

        if (isEmpty) return;

        Node\* prev = nullptr;

        if (!search(key, &prev)) return;

        Node\* toDelete = prev->next;

        prev->next = toDelete->next;

        delete toDelete;

        count--;

        isEmpty = (count == 0);

    }

    int Count() const { return count; }

    bool IsEmpty() const { return isEmpty; }

    int Operations() const { return operations; }

    void print() const {

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

            std::cout << i << ": ";

            Node\* current = table[i];

            while (current) {

                std::cout << current->value << " -> ";

                current = current->next;

            }

            std::cout << "nullptr\n";

        }

    }

    void clear() {

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

            Node\* current = table[i]->next;

            while (current) {

                Node\* next = current->next;

                delete current;

                current = next;

            }

            table[i]->next = nullptr;

        }

        count = 0;

        isEmpty = true;

    }

};

**main.cpp**

#include <iostream>

#include <random>

#include <chrono>

#include <iomanip>

#include <cassert>

#include "Hash.h"

void benchmark(double loadFactor, int trials = 10000) {

    int tableSize = 32;

    int elementCount = static\_cast<int>(tableSize \* loadFactor);

    std::vector<double> values;

    std::mt19937 rng(42);

    std::uniform\_real\_distribution<> dist(10000.0000, 15000.0000);

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

        values.push\_back(dist(rng));

    }

    HashTable table;

    for (auto v : values) table.insert(v);

    auto insert\_time = table.Operations();

    for (int i = 0; i < trials; ++i) assert(table.search(values[i % elementCount]));

    auto search\_time = table.Operations();

    for (int i = 0; i < trials; ++i) table.remove(values[i % elementCount]);

    auto delete\_time = table.Operations();

    std::cout << std::fixed << std::setprecision(2);

    std::cout << "Load factor: " << loadFactor

         << " | Insert: " << insert\_time / values.size()

         << " | Search: " << (search\_time - insert\_time) / trials

         << " | Delete: " << (delete\_time - search\_time) / trials << std::endl;

}

int main() {

    HashTable table;

    int choice;

    double value;

    while (true) {

        std::cout << "\n1. Insert\n2. Search\n3. Delete\n4. Print\n5. Clear\n6. Count\n7. Is Empty\n8. Benchmark\n0. Exit\n";

        std::cout << "Enter your choice: ";

        std::cin >> choice;

        switch (choice) {

            case 1:

                std::cout << "Enter value to insert: ";

                std::cin >> value;

                table.insert(value);

                break;

            case 2:

                std::cout << "Enter value to search: ";

                std::cin >> value;

                std::cout << (table.search(value) ? "Found" : "Not found") << std::endl;

                break;

            case 3:

                std::cout << "Enter value to delete: ";

                std::cin >> value;

                table.remove(value);

                break;

            case 4:

                table.print();

                break;

            case 5:

                table.clear();

                std::cout << "Table cleared.\n";

                break;

            case 6:

                std::cout << "Count: " << table.Count() << std::endl;

                break;

            case 7:

                std::cout << (table.IsEmpty() ? "Table is empty" : "Table is not empty") << std::endl;

                break;

            case 8:

                benchmark(0.5);

                benchmark(0.7);

                benchmark(0.9);

                break;

            case 0:

                return 0;

            default:

                std::cout << "Invalid choice.\n";

        }

    }

}