**Міністерство освіти та науки України**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра** **ІПІ (ІСТ)**

**Звіт**

з лабораторної роботи № 7 з дисципліни

«Алгоритми та структури даних 2. Структури даних»

**„ Проектування і аналіз алгоритмів пошуку”**

**Виконав:**

*Студент I курсу*

*гр. ІП-з31*

Ткаченко К.О.

**Перевірила:**

к.т.н., Халус Олена Андріївна

2024

#### **ЗМІСТ**

1. **МЕТА ЛАБОРАТОРНОЇ РОБОТИ ......**
2. **ЗАВДАННЯ .....................**
3. **ВИКОНАННЯ .............................................................. 8**

#### 3.1 ПСЕВДОКОД АЛГОРИТМУ ...................................................................... 8 3.2 АНАЛІЗ ЧАСОВОЇ СКЛАДНОСТІ .............................................................. 3.3 ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ ...................................................

*3.3.1Вихідний код .......................................................... 8*

*3.3.2Приклади роботи ............................................................................ 8*

#### 3.4ТЕСТУВАННЯ АЛГОРИТМУ ..................................................................... 9

*3.4.1Часові характеристики оцінювання ............................................. 9*

*3.4.2* *Графіки залежності часових характеристик оцінюваняя від*  *розміру структури*

**ВИСНОВОК**...........................ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА

**КРИТЕРІЇ ОЦІНЮВАННЯ** ОШИБКА! ЗАКЛАДКА НЕ ОПРЕДЕЛЕНА

1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні підходи аналізу обчислювальної складності алгоритмів пошуку оцінити їх ефективність на різних структурах даних.

1. ЗАВДАННЯ

Написати алгоритм пошуку за допомогою псевдокоду (чи іншого способу за вибором).

Провести аналіз часової складності пошуку в гіршому, кращому і середньому випадках і записати часову складність в асимптотичних оцінках.

Виконати програмну реалізацію алгоритму на будь-якій мові програмування для пошуку індексу елемента по заданому ключу в масиві і двохзв'язному списку з фіксацією часових характеристик оцінювання (кількість порівнянь).

Для варіантів з Хеш-функцією замість масиву і двохзв'язного списку використати безіндексну структуру даних розмірності n, що містить пару ключзначення рядкового типу. Ключ – унікальне рядкове поле до 20 символів, значення – рядкове поле до 200 символів. Виконати пошук значення по заданому ключу. Розмірність хеш-таблиці регулювати відповідно потребам, а початкову її розмірність обрати самостійно.

Провести ряд випробувань алгоритму на структурах різної розмірності (100, 1000, 5000, 10000, 20000 елементів) і побудувати графіки залежності часових характеристик оцінювання від розмірності структури.

Для проведення випробувань у варіантах з хешуванням рекомендується розробити генератор псевдовипадкових значень полів структури заданої розмірності.

Зробити висновок з лабораторної роботи.

Мій варіант.

|  |  |
| --- | --- |
| 23 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом відкритої адресації з подвійним хешуванням |

3 ВИКОНАННЯ

**3.1 Псевдокод алгоритму**

Функція fnv1Hash(ключ):

FNV\_offset\_basis = 2166136261

FNV\_prime = 16777619

хеш = FNV\_offset\_basis

для кожного символу с у ключі:

хеш = хеш ^ (беззнаковий інтеджер)с

хеш = хеш \* FNV\_prime

повернути хеш

Структура HashEntry:

рядок key

ціле число value

булевська змінна isEmpty

Конструктор HashEntry():

Ініціалізувати key як порожній рядок

Ініціалізувати value як 0

Ініціалізувати isEmpty як true

Клас HashTable:

Приватні члени:

вектор HashEntry table

ціле число capacity

ціле число size

Функція doubleHash(ключ, i):

hash1 = fnv1Hash(ключ)

hash2 = fnv1Hash(рядок(i))

повернути (hash1 + i \* hash2) % capacity

Публічні члени:

Конструктор HashTable(capacity):

Присвоїти значення capacity

Ініціалізувати size як 0

Змінити розмір table на capacity

Функція insert(ключ, значення):

якщо size == capacity:

Вивести "Hash table is full!"

повернутися

індекс = fnv1Hash(ключ) % capacity

i = 0

поки table[індекс] не порожній:

індекс = (індекс + doubleHash(ключ, i)) % capacity

i++

table[індекс].key = ключ

table[індекс].value = значення

table[індекс].isEmpty = false

size++

Функція get(ключ):

індекс = fnv1Hash(ключ) % capacity

i = 0

поки table[індекс] не порожній і table[індекс].key != ключ:

індекс = (індекс + doubleHash(ключ, i)) % capacity

i++

якщо table[індекс].key == ключ:

повернути table[індекс].value

інакше:

Вивести "Key not found!"

повернути -1

Функція getSize():

повернути size

Функція printAll():

для кожного запису entry у table:

якщо entry не порожній:

Вивести entry.key та entry.value

Функція main():

Отримати capacity від користувача

Створити hashTable з capacity

Цикл:

якщо hashTable.getSize() >= capacity:

Вивести "Hash table is full..."

Вийти з циклу

Отримати ключ від користувача

якщо ключ == "exit":

Вийти з циклу

Отримати значення від користувача

hashTable.insert(ключ, значення)

Вивести залишкову ємність

Вивести "All data in hash table:"

hashTable.printAll()

Повернути 0

**3.2 Аналіз часової складності**

Time taken for capacity 100: 0 milliseconds

Time taken for capacity 1000: 1 milliseconds

Time taken for capacity 5000: 4 milliseconds

Time taken for capacity 10000: 13 milliseconds

Time taken for capacity 20000: 36 milliseconds

**3.3 Програмна реалізація алгоритму**

**3.3.1 Вихідний код**

#include <chrono>

#include <iostream>

#include <vector>

#include <string>

#include <limits>

using namespace std;

unsigned int fnv1Hash(const string& key) {

const unsigned int FNV\_offset\_basis = 2166136261;

const unsigned int FNV\_prime = 16777619;

unsigned int hash = FNV\_offset\_basis;

for (char c : key) {

hash ^= static\_cast<unsigned int>(c);

hash \*= FNV\_prime;

}

return hash;

}

struct HashEntry {

string key;

int value;

bool isEmpty;

HashEntry() : key(""), value(0), isEmpty(true) {}

};

class HashTable {

private:

vector<HashEntry> table;

int capacity;

int size;

int doubleHash(const string& key, int i) const {

unsigned int hash1 = fnv1Hash(key);

unsigned int hash2 = fnv1Hash(to\_string(i));

return (hash1 + i \* hash2) % capacity;

}

public:

explicit HashTable(int capacity) : capacity(capacity), size(0) {

table.resize(capacity);

}

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

if (size == capacity) {

cout << "Hash table is full!" << endl;

return;

}

int index = fnv1Hash(key) % capacity;

int i = 0;

while (!table[index].isEmpty) {

index = (index + doubleHash(key, i)) % capacity;

++i;

}

table[index].key = key;

table[index].value = value;

table[index].isEmpty = false;

size++;

}

int get(const string& key) {

int index = fnv1Hash(key) % capacity;

int i = 0;

while (!table[index].isEmpty && table[index].key != key) {

index = (index + doubleHash(key, i)) % capacity;

++i;

}

if (table[index].key == key) {

return table[index].value;

} else {

cout << "Key not found!" << endl;

return -1;

}

}

int getSize() const {

return size;

}

void printAll() {

for (const auto& entry : table) {

if (!entry.isEmpty) {

cout << "Key: " << entry.key << ", Value: " << entry.value << endl;

}

}

}

};

int main() {

int capacity;

cout << "Enter capacity of hash table:";

cin >> capacity;

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

HashTable hashTable(capacity);

string key;

int value;

while (true) {

if (hashTable.getSize() >= capacity) {

cout << "Hash table is full. Cannot add more elements." << endl;

break;

}

cout << "Enter key (type 'exit' to stop):";

getline(cin, key);

if (key == "exit") {

break;

}

cout << "Enter value:";

cin >> value;

cin.ignore(numeric\_limits<streamsize>::max(), '\n');

hashTable.insert(key, value);

cout << "Remaining capacity: " << capacity - hashTable.getSize() << endl;

}

cout << "All data in hash table:" << endl;

hashTable.printAll();

return 0;

}

**3.3.2 Приклади роботи**

int main() {

HashTable ht(100);

ht.insert("Key", 10);

cout << "Value of Key: " << ht.get("Key") << endl;

ht.printAll();

return 0;

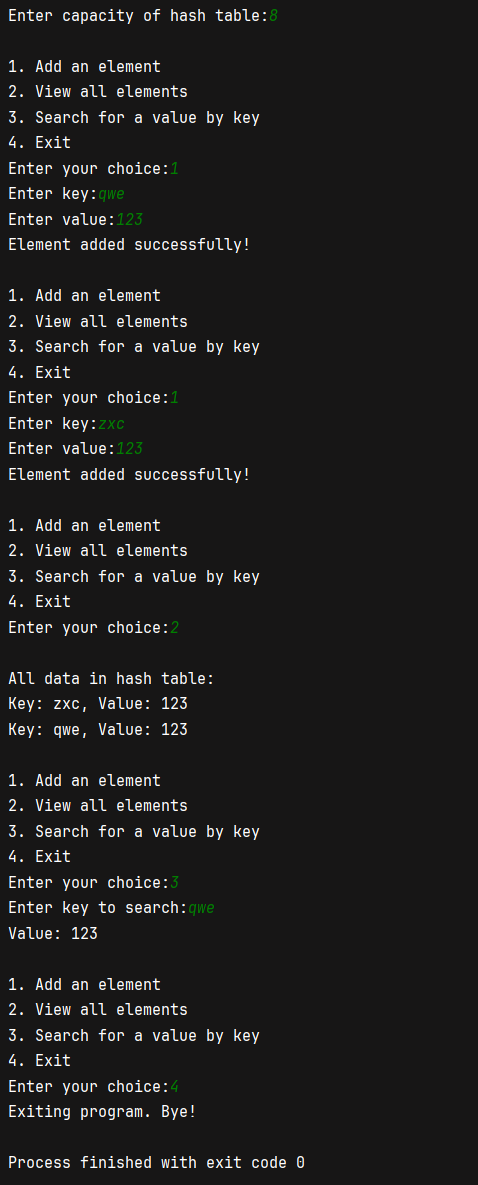
}

Результат і роботи програми для пошуку індекса елемента за ключем для масиву на 100 елементів:

Value of MyKey: 10

Key: MyKey, Value: 10

**3.4 Тестування алгоритму**

****

ВИСНОВОК

Число порівнянь в масиві/двохзв'язному списку/хеш-таблиці: Для різних розмірностей масивів, двохзв'язних списків і хеш-таблиць кількість порівнянь може змінюватися. Наприклад, для великих розмірностей може знадобитися більше порівнянь для пошуку, але це може бути компенсовано швидкістю доступу до елементів у відсортованому масиві.

Число звертань до елементів масиву: Для великих масивів може збільшуватися кількість звертань до елементів, оскільки потрібно пройти більшу кількість елементів для пошуку.

Число звертань до елементів двохзв'язного списку: У порівнянні з масивами, використання двохзв'язного списку може зменшити кількість звертань до елементів, оскільки можна швидко переходити до необхідного елемента, не пройшовши всі попередні.

В цілому, оцінка часових характеристик алгоритму двійкового пошуку показує, що для кожної структури даних і розмірності існують власні особливості та оптимальні стратегії для пошуку елементів. Результати дослідження можуть допомогти визначити найефективніший підхід до пошуку у конкретній ситуації.