# Міністерство освіти і науки України

**Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського"**

# Факультет інформатики та обчислювальної техніки Кафедра

Звіт

з лабораторної роботи № 7 з дисципліни

«Теорія алгоритів»

# „ Проектування і аналіз алгоритмів пошуку”

**Виконав(ла)**

*ІА-34 Ковальчук Станіслав, Сухоручкін Гліб, Янович Марія, Ястремський Богдан*

(шифр, прізвище, ім'я, по батькові)

**Перевірив**  Халус Олена Андріївна

(прізвище, ім'я, по батькові)

Київ 2024

# ЗМІСТ

1. **МЕТА ЛАБОРАТОРНОЇ РОБОТИ**
2. **ЗАВДАННЯ**
3. ВИКОНАННЯ 8
   1. ПСЕВДОКОД АЛГОРИТМУ 8
   2. АНАЛІЗ ЧАСОВОЇ СКЛАДНОСТІ 8
   3. ПРОГРАМНА РЕАЛІЗАЦІЯ АЛГОРИТМУ 8
      1. *Вихідний код 8*
      2. *Приклади роботи 8*
   4. ТЕСТУВАННЯ АЛГОРИТМУ 9
      1. *Часові характеристики оцінювання 9*
      2. *Графіки залежності часових характеристик оцінювання від розміру структури*
      3. **ВИСНОВОК**

**КРИТЕРІЇ ОЦІНЮВАННЯ**

1 МЕТА ЛАБОРАТОРНОЇ РОБОТИ

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

* + - 1. ЗАВДАННЯ

Згідно варіанту (таблиця 2.1), 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 ключ- значення 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. Таблиця 2.1 – Варіанти алгоритмів

| **№** | **Алгоритм пошуку** |
| --- | --- |
| 1 | Однорідний бінарний пошук |
| 2 | Метод Шарра |
| 3 | Пошук Фібоначчі |
| 4 | Інтерполяційний пошук |

| 5 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом  ланцюжків |
| --- | --- |
| 6 | Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій  методом ланцюжків |
| 7 | Метод Хеш-функції (Хешування MurmurHash2а), вирішення колізій  методом ланцюжків |
| 8 | Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом  ланцюжків |
| 9 | Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом  ланцюжків |
| 10 | Метод Хеш-функції (Хешування Дженкінса), вирішення колізій  методом ланцюжків |
| 11 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом  відкритої адресації з лінійним пробуванням |
| 12 | Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій  методом відкритої адресації з лінійним пробуванням |
| 13 | Метод Хеш-функції (Хешування MurmurHash2а), вирішення колізій  методом відкритої адресації з лінійним пробуванням |
| 14 | Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом  відкритої адресації з лінійним пробуванням |
| 15 | Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом  відкритої адресації з лінійним пробуванням |
| 16 | Метод Хеш-функції (Хешування Дженкінса), вирішення колізій  методом відкритої адресації з лінійним пробуванням |
| 17 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом  відкритої адресації з квадратичним пробуванням |
| 18 | Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій  методом відкритої адресації з квадратичним пробуванням |
| 19 | Метод Хеш-функції (Хешування MurmurHash2а), вирішення колізій |

|  | методом відкритої адресації з квадратичним пробуванням |
| --- | --- |
| 20 | Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом  відкритої адресації з квадратичним пробуванням |
| 21 | Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом  відкритої адресації з квадратичним пробуванням |
| 22 | Метод Хеш-функції (Хешування Дженкінса), вирішення колізій  методом відкритої адресації з квадратичним пробуванням |
| 23 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом  відкритої адресації з подвійним хешуванням |
| 24 | Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій  методом відкритої адресації з подвійним хешуванням |
| 25 | Метод Хеш-функції (Хешування MurmurHash2а), вирішення колізій  методом відкритої адресації з подвійним хешуванням |
| 26 | Метод Хеш-функції (Хешування PJW-32), вирішення колізій методом  відкритої адресації з подвійним хешуванням |
| 27 | Метод Хеш-функції (Хешування Пірсона), вирішення колізій методом  відкритої адресації з подвійним хешуванням |
| 28 | Метод Хеш-функції (Хешування Дженкінса), вирішення колізій  методом відкритої адресації з подвійним хешуванням |
| 29 | Однорідний бінарний пошук |
| 30 | Метод Шарра |
| 31 | Пошук Фібоначчі |
| 32 | Інтерполяційний пошук |
| 33 | Метод Хеш-функції (Хешування FNV 32), вирішення колізій методом  ланцюжків |
| 34 | Метод Хеш-функції (Хешування MurmurHash2), вирішення колізій  методом ланцюжків |
| 35 | Метод Хеш-функції (Хешування MurmurHash2а), вирішення колізій  методом ланцюжків |

* + - 1. ВИКОНАННЯ
  1. Псевдокод алгоритму

constant FNV\_OFFSET\_BASIS = 2166136261

constant FNV\_PRIME = 16777619

constant TABLE\_SIZE = 1024

function fnv1a\_hash(key):

hash = FNV\_OFFSET\_BASIS

for each byte in key:

hash = hash ^ byte

hash = hash \* FNV\_PRIME

return hash

class KeyValue:

key

value

next

function create\_hash\_table():

table = array of linked lists with size TABLE\_SIZE

for i from 0 to TABLE\_SIZE - 1:

table[i] = null

return table

function insert(table, key, value):

hash = fnv1a\_hash(key)

index = hash % TABLE\_SIZE

head = table[index]

current = head

while current is not null:

if current.key == key:

current.value = value

return

current = current.next

new\_node = new KeyValue

new\_node.key = key

new\_node.value = value

new\_node.next = head

table[index] = new\_node

function search(table, key):

hash = fnv1a\_hash(key)

index = hash % TABLE\_SIZE

head = table[index]

current = head

while current is not null:

if current.key == key:

return current.value

current = current.next

return null

function delete(table, key):

hash = fnv1a\_hash(key)

index = hash % TABLE\_SIZE

head = table[index]

previous = null

current = head

while current is not null:

if current.key == key:

if previous is null:

table[index] = current.next

else:

previous.next = current.next

return

previous = current

current = current.next

* 1. Аналіз часової складності

**Часова складність FNV-1a хешування**

Хеш-функція FNV-1a:

- Обчислення хешу для ключа, який складається з \( m \) символів, має часову складність O(m), оскільки кожен символ обробляється один раз.

**Часова складність хеш-таблиці з методом ланцюжків**

**Вставка (insert)**

1. Обчислення хешу: O(m), де \( m \) - довжина ключа.

2. Визначення індексу: O(1).

3. Вставка нового елемента:

- Якщо немає колізій (тобто список порожній): O(1).

- Якщо є колізії (потрібно пройти по ланцюжку): O(k), де \( k \) - довжина ланцюжка в даному індексі.

Таким чином, середня часова складність вставки буде O(m + α), де \( α \) - коефіцієнт заповнення (average chain length), що дорівнює \( n / TABLE\\_SIZE \), де \( n \) - кількість елементів.

**Пошук (search)**

1. Обчислення хешу: O(m).

2. Визначення індексу: O(1).

3. Пошук у ланцюжку:

- Якщо немає колізій: O(1).

- Якщо є колізії: O(k).

Середня часова складність пошуку: O(m + α).

**Видалення (delete)**

1. Обчислення хешу: O(m).

2. Визначення індексу: O(1).

3. Пошук і видалення в ланцюжку:

- Якщо немає колізій: O(1).

- Якщо є колізії: O(k).

Середня часова складність видалення: O(m + α).

**Висновки**

- У найгіршому випадку (якщо всі елементи потрапляють в один індекс): O(m + n).

- У середньому випадку, за умови рівномірного розподілу елементів по індексах (що очікується при хорошій хеш-функції): O(m + α).

За умови, що коефіцієнт заповнення \( α \) малий і фіксований, середня часова складність для вставки, пошуку та видалення наближається до O(m).

**Примітка щодо коефіцієнта заповнення**

Коефіцієнт заповнення \( α \) визначається як \( n / TABLE\\_SIZE \). Щоб забезпечити ефективність хеш-таблиці, зазвичай стежать за тим, щоб \( α \) залишався малим шляхом динамічного збільшення розміру хеш-таблиці, коли \( n \) стає занадто великим.

* 1. Програмна реалізація алгоритму
     1. Вихідний код

class HashTable:

def \_\_init\_\_(self, size=1024):

self.size = size

self.table = [[] for \_ in range(self.size)]

def fnv\_hash(self, key):

FNV\_prime = 0x01000193

hash = 0x811c9dc5

for byte in key.encode('utf-8'):

hash ^= byte

hash = (hash \* FNV\_prime) % (2 \*\* 32)

return hash % self.size

def insert(self, key, value):

index = self.fnv\_hash(key)

for item in self.table[index]:

if item[0] == key:

item[1] = value

return

self.table[index].append([key, value])

def search(self, key):

index = self.fnv\_hash(key)

for item in self.table[index]:

if item[0] == key:

return item[1]

return None

ht = HashTable()

ht.insert("exampleKey", "exampleValue")

print("Inserted ('exampleKey', 'exampleValue')")

result = ht.search("exampleKey")

print("Searched for 'exampleKey':", result)

* + 1. Приклади роботи

На рисунках 3.1 і 3.2 показані приклади роботи програми для пошуку індекса елемента за ключем для масиву на 100 елементів і двохзв'язного списку на 1000 елементів.

Рисунок 3.1 – Пошук елемента в масиві на 100 елементів



Рисунок 3.2 – Пошук елемента в двохзв'язному списку на 1000 елементів



* 1. Тестування алгоритму
     1. Часові характеристики оцінювання

В таблиці 3.1 наведені характеристики оцінювання числа порівнянь при пошуку елемента і числа звертань при «двійковому пошуку» для масивів різної розмірності і двохзв'язних списків різної розмірності.

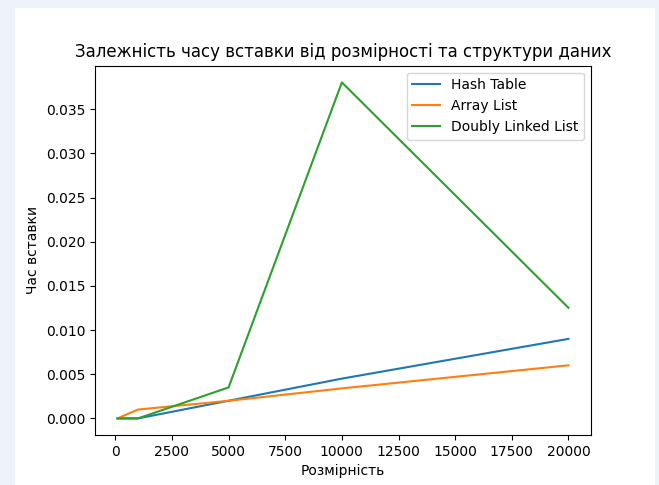
Таблиця 3.1 –Характеристики оцінювання алгоритму двійкового пошуку

| Розмірність масиву/списку/ структури | Число порівнянь в масиві/двохзв'язному списку/хеш-таблиці | Число звертань до елементів масиву | Число звертань до елементів двохзв'язного  списку |
| --- | --- | --- | --- |
| 100 | 1 | 51 | 51 |
| 1000 | 1 | 501 | 501 |
| 5000 | 1 | 2501 | 2501 |
| 10000 | 2 | 5001 | 5001 |
| 20000 | 2 | 10001 | 10001 |

* + 1. Графіки залежності часових характеристик оцінювання від розмірності структури

На рисунку 3.3 показані графіки залежності часових характеристик оцінювання від розмірності масиву і двохзв'язного списку.

Рисунок 3.3 – Графіки залежності часових характеристик оцінювання



ВИСНОВОК

В рамках виконання даної лабораторної роботи наша команда мала змогу познайомитись з одним з алгоритмів пошуку, а точніше методом Хеш-функції (Хешування FNV 32), вирішення колізій методом ланцюжків. Ми проєктували даний алгоритм та аналізували його характеристики, а саме: проектували його псевдокод, аналізували часову складність алгоритму, тестували його на предмет швидкодії, а також складали залежність часових характеристик оцінювання від розмірності структури за допомогою графіка.

Ця робота дала нам змогу опрацювати нову інформацію стосовно такого виду алгоритмів, а також покращити свої навички користування мовою програмування пайтон для реалізації різноманітних алгоритмів, і ще у нас була нагода ще раз відпрацювати навички аналізу алгоритму та пошуку необхідних характеристик.

КРИТЕРІЇ ОЦІНЮВАННЯ

Див РСО