Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

Алгоритми та складність

Варіант №9

“ Ідеальне хешування для комплексних чисел”

Виконав студент 2-го курсу

Групи ІПС-21

Тесленко Назар Олександрович

Київ – 2025

Завдання:

Реалізувати ідеальне хешування для комплексних чисел

Теорія:

Ідеальне хешування (Perfect Hashing) - це спеціальна техніка хешування, яка гарантує відсутність колізій при доступі до даних. На відміну від звичайних хеш-таблиць, де колізії вирішуються методами ланцюжків або відкритої адресації, ідеальне хешування забезпечує доступ до елементів за час O(1) в гіршому випадку, а не тільки в середньому.

Для реалізації ідеального хешування використовується дворівнева структура:

1. **Перший рівень**: звичайна хеш-функція розподіляє елементи по кошиках (buckets)
2. **Другий рівень**: для кожного кошика з колізіями створюється окрема хеш-таблиця з власною хеш-функцією

Хеш-функція (Hash Function) - функція, яка перетворює дані довільного розміру у значення фіксованого розміру (хеш). У контексті даної реалізації, параметризується значеннями a, b, p та m, де

- **a** - випадковий коефіцієнт, не рівний нулю ($1 \leq a < p$)
- **b** - випадковий зсув ($0 \leq b < p$)
- **p** - велике просте число (більше, ніж максимальне значення хешованого ключа)
- **m** - розмір хеш-таблиці

Колізія (Collision) - ситуація, коли для двох різних вхідних даних хеш-функція повертає однакове значення.

Кошик (Bucket) - комірка хеш-таблиці першого рівня, що може містити декілька елементів або посилання на хеш-таблицю другого рівня.

Дворівнева хеш-таблиця (Two-level Hash Table) - структура даних, де на першому рівні елементи розподіляються по кошиках, а на другому рівні для кожного кошика з колізіями створюється окрема хеш-таблиця з власною хеш-функцією.

Первинна хеш-таблиця (Primary Hash Table) - хеш-таблиця першого рівня, що розподіляє елементи по кошиках.

Вторинна хеш-таблиця (Secondary Hash Table) - хеш-таблиця другого рівня, що створюється для кожного кошика з колізіями.

Алгоритм

Підготовка структури даних:

- Спочатку визначається розмір первинної хеш-таблиці.
- Створюється первинна хеш-функція для розподілу елементів на кошики.
- Ініціалізуються допоміжні структури для вторинних хеш-таблиць та їхніх функцій.

Створення первинної хеш-функції:

- Обирається випадкове просте число p, більше за максимальне можливе значення елемента.
- Генеруються випадкові параметри a (в діапазоні [1, p-1]) та b (в діапазоні [0, p-1]) для формули h(x) = ((a * x + b) mod p) mod m, де m - розмір таблиці.
- Ці параметри забезпечують універсальність хеш-функції, мінімізуючи ймовірність колізій між різними елементами.

Розподіл елементів по кошиках:

- Для кожного елемента з вхідного набору обчислюється хеш за допомогою первинної хеш-функції.
- Елемент (або його копія) додається до відповідного кошика в тимчасовій структурі.
- При цьому враховується тип елемента (наприклад, ComplexNumber або ComplexVector) для створення правильної копії.

Обробка кошиків:

- Для кожного кошика визначається кількість елементів.
- Якщо кошик порожній, він пропускається.
- Якщо у кошику лише один елемент, створюється вторинна таблиця розміром 1 і елемент розміщується в ній.
- Якщо у кошику декілька елементів, обробка продовжується додатковими кроками.

Створення вторинних хеш-таблиць:

- Для кошиків з декількома елементами визначається розмір вторинної таблиці як квадрат кількості елементів у кошику.

- Створюється нова хеш-функція для вторинної таблиці з новими випадковими параметрами.
- Ініціалізується вторинна хеш-таблиця відповідного розміру.

Розміщення елементів у вторинних таблицях:

- Для кожного елемента в кошику обчислюється вторинний хеш за допомогою відповідної вторинної хеш-функції.
- Якщо позиція у вторинній таблиці вже зайнята (виникла колізія), перегенеруються параметри вторинної хеш-функції і всі елементи хешуються знову.
- Процес повторюється до тих пір, поки не буде знайдена хеш-функція, що не створює колізій, або не буде вичерпано максимальну кількість спроб.

Пошук елементів:

- Для пошуку елемента спочатку обчислюється його первинний хеш.
- Визначається відповідний кошик у первинній таблиці.
- Якщо кошик порожній або не ініціалізований, елемент відсутній.
- Якщо в кошику один елемент, здійснюється пряме порівняння.
- Якщо в кошику декілька елементів, обчислюється вторинний хеш і здійснюється пошук у вторинній таблиці.

Результат:

- Ідеальне хешування забезпечує доступ до елементів за сталий час O(1) у найгіршому випадку.
- Структура використовує O(n) пам'яті, де n - кількість елементів.
- Структура ефективна для статичних наборів даних, коли всі елементи відомі заздалегідь.

Алгоритм можна реалізувати наступним псевдокодом:

1.Ініціалізація головного хешу:

primaryHashFunction = створити нову хеш-функцію з розміром primarySize
створити primaryTable розміром primarySize (порожній)
створити масиви secondaryHashFuncs, secondaryTableSizes, secondaryTableInitialized розміром primarySize

2. Розподіл елементів по первинних комірках:

створити тимчасові списки tempBuckets розміром primarySize
для кожного елемента у списку elements:
Пропускаємо нульові елементи якщо element == null

primaryIndex = primaryHashFunction.hash(element)

Створення глибокої копії елемента залежно від його типу
якщо element є типу ComplexNumber:
elementCopy = нова копія ComplexNumber(element)
інакше якщо element є типу ComplexVector:
elementCopy = нова копія ComplexVector(element)

Додаємо копію елемента до відповідного кошика
якщо elementCopy != null:
додати elementCopy у tempBuckets[primaryIndex]

3. Ініціалізація вторинних хеш-таблиць:

для кожного i від 0 до primarySize:
Очищення попередніх елементів кошика (якщо є) для кожного element у hashTable[i]
elementsInBucket = розмір tempBuckets[i]
якщо elementsInBucket == 0:
пропустити
якщо elementsInBucket == 1:
secondaryTable[i] = створити таблицю розміром 1
secondaryTable[i][0] = tempBuckets[i][0]
secondaryTableSizes[i] = 1
позначити secondaryTable[i] як ініціалізовану
якщо elementsInBucket > 1:
secondarySize = elementsInBucket * elementsInBucket
secondaryTableSizes[i] = secondarySize
secondaryHashFuncs[i].setTableSize(secondarySize)
повторювати поки не знайдено ідеальну хеш-функцію (або поки не перевищено MAX_ATTEMPTS):
secondaryHashFuncs[i] = нова HashFunction(secondarySize) очистити
hashTable[i] hashTable[i].resize(secondarySize, null)
встановити noCollision = true

для кожного елемента у tempBuckets[i]:
secondaryIndex = secondaryHashFunction[i].hash(element)
якщо secondaryTable[i][secondaryIndex] вже містить інший елемент:
noCollision = false
завершити внутрішній цикл
вставити element у secondaryTable[i][secondaryIndex]

якщо noCollision == true:
позначити secondaryTable[i] як ініціалізовану

4.Пошук елемента у хеш-таблиці:

```
    primaryIndex = primaryHashFunction.hash(елемент)
    якщо primaryIndex поза межами primarySize:
        повернути false
    якщо secondaryTable[i] не ініціалізована або її розмір 0:
        повернути false
    якщо secondaryTableSizes[i] == 1:
        повернути hashTable[primaryIndex][0] != null і
*hashTable[primaryIndex][0] == елемент

    secondaryIndex = secondaryHashFunction[i].hash(елемент)
    якщо secondaryIndex поза межами таблиці:
        повернути false
    повернути (secondaryTable[i][secondaryIndex] == елемент)
```

Мова реалізації: C++

Модулі програми:

Головний клас:

- class PerfectHashing

Реалізує алгоритм Ідеального хешування за допомогою методів вставки та пошуку хеш значення у хеш-таблиці

Методи класу PerfectHashing:

- **PerfectHashing(int size)**
  Конструктор, що ініціалізує первинну хеш-функцію, основну хеш-таблицю та допоміжні структури для вторинного рівня хешування.
- **void insert(const vector<Hashable*>& elements)**
  Виконує вставку множини комплексних чисел у хеш-таблицю, використовуючи дворівневе хешування для уникнення колізій.
- **bool search(const Hashable& element) const**
  Перевіряє, чи знаходиться заданий елемент у хеш-таблиці, виконуючи пошук через первинну та вторинну хеш-функції.
- **void print() const**
  Виводить у консоль структуру побудованої хеш-таблиці, включаючи інформацію про кожен бакет вторинного рівня.

Складність алгоритму PerfectHashing та залжених класів

class Hashable (interface)

class ComplexNumber : Hashable

- operator== і operator!=: ***O(1)*** – порівняння двох чисел виконується за сталий час.
- toInt(): ***O(1)*** – проста арифметична операція.
- printComplex(): ***O(1)*** – виведення на екран.

class ComplexVector : Hashable

- operator== і operator!=: ***O(1)*** – порівняння двох чисел виконується за сталий час.
- toInt(): ***O(1)*** – проста арифметична операція.
- printComplex(): ***O(1)*** – виведення на екран.

class HashFunction

- HashFunction(int tableSize): ***O(1)*** – ініціалізація константних значень та виклик generateHashFunction().
- isPrime(int num): ***O(√n)*** – перебір всіх дільників до √n.
- getRandomPrime(int min, int max): ***O(K√M)***, де K – кількість спроб (обмежена MAX_ATTEMPTS), M – верхня межа діапазону. У гіршому випадку виконується 1000 перевірок простоти.
- generateHashFunction(): ***O(1)*** – генерація випадкових чисел.
- hash(ComplexNumber z): ***O(1)*** – проста функція хешування.
- setTableSize(int tableSize): ***O(1)*** – переприсвоєння змінної.

class PerfectHashing

- **Конструктор PerfectHashing(int size)**: ***O(n)***
  Ініціалізація масивів, створення primaryHashFunc, створення secondaryHashFuncs (хоч вони й створюються з size=1, все одно O(n)).
- **insert(const vector<ComplexNumber>& elements)**
  1. **Первинне хешування**: Перебір всіх elements і розподіл у tempBuckets – ***O(n)***.

2. **Формування вторинних хеш-таблиць**:
    - Якщо elementsInBucket == 1 : ***O(1)***
    - Якщо elementsInBucket > 1:
        - secondarySize = elementsInBucket^2 :***O(1)***
        - Генерація нової хеш-функції: ***O(1)***
        - Перехешування (до MAX_ATTEMPTS разів у найгіршому випадку) → ***O(secondarySize)***
3. **Середня складність –** ***Θ(n)***, якщо хешування рівномірне. У найгіршому випадку (коли всі елементи в одному кошику): ***O(n²)***

- **search(const ComplexNumber& element)**:
    1. Первинне хешування: ***O(1)***
    2. Перевірка наявності вторинної таблиці: ***O(1)***
    3. Вторинне хешування та пошук: ***O(1)***

- **print() const**:
    1. Перебір всіх primarySize кошиків: O(n)
    2. Перебір всіх hashTable[i] в кожному кошику (сумарно O(n))

Загальний підсумок складності:

- Вставка (insert) – **Θ (n), O(n²)**
- Пошук (search) – **O(1)**
- Вивід (print) – **O(n)**
- Конструктор (PerfectHashing) – ***O*(n)**

Тестові приклади:

Тест 1 (вектор комплексних чисел)

F={3+4i, 1+2i, 5-2i, -2+7i, 0+0i}

Find:

- 1+2i
- -2-7i

```cpp
vector<Hashable*> elements;

//Test1
// Додаємо комплексні числа
elements.push_back(new ComplexNumber(3, 4));
elements.push_back(new ComplexNumber(1, 2));
elements.push_back(new ComplexNumber(5, -2));
elements.push_back(new ComplexNumber(-2, 7));
elements.push_back(new ComplexNumber(0, 0));

hashTable.insert(elements);
std::cout << "\n\n";
hashTable.print();

std::cout << "\nDemonstration of search:\n";

ComplexNumber z1(1, 2);
ComplexNumber z2(-2, -7);
std::cout << "Searching for ";
z1.printComplex();
std::cout << ": " << (hashTable.search(z1) ? "Found" : "Not found") << "\n\n";

std::cout << "Searching for ";
z2.printComplex();
std::cout << ": " << (hashTable.search(z2) ? "Found" : "Not found") << "\n\n";
```

```
=== Perfect Hash Table Structure ===
Primary table size: 4

Bucket 0: Empty

Bucket 1 (size: 1):
  Single element: 0+0i

Bucket 2 (size: 1):
  Single element: 3+4i

Bucket 3 (size: 9):
  [0]: Empty
  [1]: 5-2i
  [2]: 1+2i
  [3]: Empty
  [4]: Empty
  [5]: Empty
  [6]: -2+7i
  [7]: Empty
  [8]: Empty

================================
```

```
==================================

Demonstration of search:
Searching for 1+2i: Found

Searching for -2-7i: Not found
```

Тест 2 *(двовимірний вектор)*

F= {{1+1i, 2+2i}, {3+3i,4+4i,5+5i}, {-1-1i, -2-2i}, {-2, -0}}

Find:

- (1+1i, 2+2i)
- (4-4i, -5+5i)
- (3+3i, 4+4i,5+5i)
- (4+4i, -5+5i)

Реалізуємо ідеальне хешування для таблиці різних розмірів (вхідні дані однакові):

Hash table size: 4

```
=== Perfect Hash Table Structure ===
Primary table size: 4

Bucket 0 (size: 1):
  Single element: (-2+0i)

Bucket 1 (size: 1):
  Single element: (3+3i,4+4i,5+5i)

Bucket 2 (size: 1):
  Single element: (-1-1i,-2-2i)

Bucket 3 (size: 1):
  Single element: (1+1i,2+2i)
```

```
==================================

Demonstration of search:
Searching for vector: (1+1i,2+2i): Found

Searching for vector: (4-4i,-5+5i): Not found

Searching for vector: (3+3i,4+4i,5+5i): Found

Searching for vector: (4+4i,-5+5i): Not found
```

Hash table size:1

```
=== Perfect Hash Table Structure ===
Primary table size: 1

Bucket 0 (size: 16):
  [0]: Empty
  [1]: Empty
  [2]: Empty
  [3]: Empty
  [4]: Empty
  [5]: (-1-1i,-2-2i)
  [6]: Empty
  [7]: (1+1i,2+2i)
  [8]: Empty
  [9]: Empty
  [10]: Empty
  [11]: (3+3i,4+4i,5+5i)
  [12]: Empty
  [13]: (-2+0i)
  [14]: Empty
  [15]: Empty
```

```
==================================

Demonstration of search:
Searching for vector: (1+1i,2+2i): Found

Searching for vector: (4-4i,-5+5i): Not found

Searching for vector: (3+3i,4+4i,5+5i): Found

Searching for vector: (4+4i,-5+5i): Not found
```

Алгоритм вірно знайшов вектори у хеш таблиці.

Test 3 *(двовимірні вектори і комплексні числа)*

F= {3+4i, 1+2i, 5-2i, -2+7i, 0+0i, {1+1i, 2+2i}, {3+3i,4+4i,5+5i},{-1-1i, -2-2i},{-2,-0}}

Find:

- 3+4i
- 0+0i
- (1+1i, 2+2i)
- (-2+0i)
- 6+6i
- (9+9i, 10+10i)

```
=== Perfect Hash Table Structure ===
Primary table size: 6

Bucket 0 (size: 1):
  Single element: (-1-1i,-2-2i)

Bucket 1 (size: 1):
  Single element: (3+3i,4+4i,5+5i)

Bucket 2 (size: 9):
  [0]: 1+2i
  [1]: 5-2i
  [2]: 0+0i

Bucket 3: Empty

Bucket 4 (size: 4):
  [0]: (-2+0i)
  [1]: 3+4i
  [2]: Empty
  [3]: Empty

Bucket 5 (size: 4):
  [0]: Empty
  [1]: (1+1i,2+2i)
  [2]: -2+7i
  [3]: Empty
```

```
==================================

Demonstration of search:
Searching for 3+4i: Found

Searching for 0+0i: Found

Searching for vector: (1+1i,2+2i): Found

Searching for vector: (-2+0i): Found

Searching for 6+6i: Not found

Searching for vector: (9+9i,10+10i): Not found
```

Висновок:

Було реалізовано дворівневу хеш-таблицю, що складається з:

- **Первинної хеш-функції**, яка розподіляє елементи по кошиках.
- **Вторинних хеш-таблиць**, які усувають можливі колізії за допомогою окремих хеш-функцій.

Було протестовано різні набори комплексних чисел, перевірено правильність розподілу по хеш-таблицях, а також коректність пошуку елементів. Ідеальне хешування було успішно реалізовано. Всі основні операції виконуються за O(1). Перевірка показала відсутність колізій після завершення побудови структури.

Використані джерела:

- Вікіпедія
- Лекція 2-3 АіС
- StackOverflow