Київський національний університет імені Тараса Шевченка
Факультет комп’ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем
Алгоритми та складність

Лабораторна робота №1
“ Реалізація ідеального хешування ”
Виконала студентка 2-го курсу
Групи ІПС-21
Сенечко Дана Володимирівна

Київ - 2025

Завдання

Реалізація ідеального хешування. Тип даних за варіантом Т7 - дійсні числа.

Теорія

Ідеальна хеш-функція – хеш функція, яка перетворює завчасно відому статичну множину ключів в діапазоні цілих чисел [0,n-1] без колізій, тобто один ключ відповідає лише одному унікальному значенню.

Алгоритм

- Отримуємо набір вхідних даних - ключів (дійсні числа).
- Перевіряємо дані на повтори.
- Створюємо хеш-таблицю, розмір якої дорівнює кількості елементів вхідного набору даних.
- Для кожного елемента рахуємо його індекс у хеш-таблиці за допомогою функції $h(i) = ((a * i + b) \bmod p) \bmod m$ та розподіляємо у відповідну комірку.
- Обробляємо комірки: якщо комірка порожня залишаємо її без змін; якщо комірка містить один елемент він просто зберігається у відповідній комірці; якщо у комірці декілька елементів (колізія) → переходимо до наступного кроку.
- Якщо виникає колізія, тобто до однієї комірки потрапляє кілька елементів створюємо під-хеш-таблицю у цій комірці, розмір якої дорівнює квадрату кількості елементів, що потрапили в одну й ту саму комірку.
- Хешуємо кожний елемент підтаблиці попередньо обравши нові змінні a, b. Повторюємо цей пункт доки не буде нових колізій.

Складність алгоритму

Алгоритм створення таблиці працює за константний час O(n) в найгіршому випадку, алгоритм пошуку – за час O(1) в найгіршому випадку.

Мова реалізації алгоритму

C++

Модулі програми

- **`class PerfectHashTable;`**

Реалізує ідеальну хеш-таблицю з дворівневим хешуванням.

  - **`PerfectHashTable(const vector<double>& keys)`**

  Ініціалізує таблицю та заповнює buckets. Розподіляє елементи по першому рівню хешування. Якщо є колізії, створює другий рівень хешування.

  - **`bool lookup(double key) const`**

  Виконує пошук значення у таблиці. Спочатку використовує перший рівень хешування, а якщо потрібно — другий.

- **`struct HashFunction`**

Реалізує хеш-функцію, що використовується для першого та другого рівнів хешування.

  - **`HashFunction(int size)`**

  Конструктор, що ініціалізує випадкові значення a і b.

  - **`int operator()(double key) const`**

  Оператор виклику, що обчислює хеш для заданого ключа.

- **`struct Bucket`**

Представляє відро (комірку) у хеш-таблиці, що може містити:

- окремий елемент;
- групу елементів, що вимагають другого рівня хешування.

- **`int main()`**

Головна функція програми, у якій виконуються такі дії:

- Створюється вектор keys із заданими тестовими значеннями.
- Виводиться список ключів.
- Ініціалізується PerfectHashTable table(keys).
- Викликається table.printStats() для відображення статистики таблиці.
- Виконується пошук тестових значень у таблиці та виводиться результат.

Інтерфейс користувача

Вхідні дані вводяться програмно (в функції int main()). Результат виводиться в консоль.

Тестові приклади

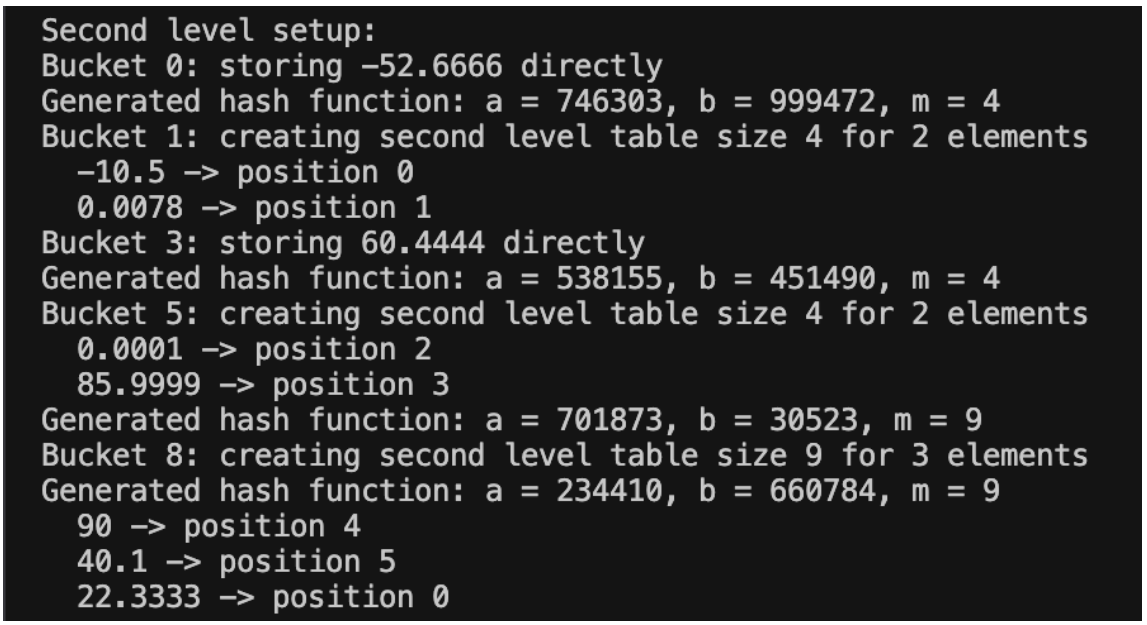
1) Вхідні дані:

Creating perfect hash table with keys:

-10.5 22.3333 0.0078 40.1 -52.6666 60.4444 0.0001 85.9999 90 22.3333

Generated hash function: a = 320235, b = 264825, m = 10

```
First level distribution:
90 -> bucket 8
0.0001 -> bucket 5
60.4444 -> bucket 3
-10.5 -> bucket 1
-52.6666 -> bucket 0
40.1 -> bucket 8
0.0078 -> bucket 1
85.9999 -> bucket 5
22.3333 -> bucket 8
```



Розглянемо роботу програми на прикладі такого набору ключів:

{-10.5, 22.3333, 0.0078, 40.1, -52.6666, 60.4444, 0.0001, 85.9999, 90, 22.3333}

Ключ 22.3333 повторюється двічі, враховуємо його один раз, щоб не було повторів.

Маємо константи:

a = 320235, b = 264825, m = 10, p = 1000000007

Переводимо наші дійсні числа в цілі для подальших обчислень (домножаємо кожен ключ на 10^6 та беремо їх по модулю):

{10500000, 22333300, 7800, 40100000, 52666600, 60444400, 100, 85999900, 90000000}

Далі обчислюємо перший рівень хешування за формулою:

$$h(i) = ((a * i + b) \bmod p) \bmod m$$

key	keyInt	Розрахунок h(key)	Bucket
-10.5	10500000	((320235 · 10500000 + 264825) mod 1000000007) mod 10	1
22.3333	22333300	((320235 · 22333300 + 264825) mod 1000000007) mod 10	8
0.0078	7800	((320235 · 7800 + 264825) mod 1000000007) mod 10	1
40.1	40100000	((320235 · 40100000 + 264825) mod 1000000007) mod 10	8
-52.6666	52666600	((320235 · 52666600 + 264825) mod 1000000007) mod 10	0
60.4444	60444400	((320235 · 60444400 + 264825) mod 1000000007) mod 10	3
0.0001	100	((320235 · 100 + 264825) mod 1000000007) mod 10	5
85.9999	85999900	((320235 · 85999900 + 264825) mod 1000000007) mod 10	5
90	90000000	((320235 · 90000000 + 264825) mod 1000000007) mod 10	8

Обчислюємо другий рівень хешування для комірок, що містять більше ніж один елемент:

Bucket 1: {-10.5, 0.0078}

- Генеруємо нові числа a = 746303, b = 999472. При цьому m = 4, оскільки маємо 2 елементи у комірці, а значення p залишається тим самим.
- Розраховуємо за формулою:
  $h'(key) = ((746303 \cdot keyInt + 999472) \bmod 1000000007) \bmod 4$
- Отримуємо: $-10.5 \rightarrow 0$ та $0.0078 \rightarrow 1$

Bucket 5: {0.0001, 85.9999}

- Генеруємо нові числа a = 538155, b = 451490. При цьому m = 4, оскільки маємо 2 елементи у комірці, а значення p залишається тим самим.
- Розраховуємо за формулою:
  $h'(key) = ((538155 \cdot keyInt + 451490) \bmod 1000000007) \bmod 4$
- Отримуємо: $0.0001 \rightarrow 2$ та $85.9999 \rightarrow 3$

Bucket 8: {22.3333, 40.1, 90}

- Генеруємо нові числа a = 701873, b = 30523. При цьому m = 9, оскільки маємо 3 елементи у комірці, а значення p залишається тим самим.
- Згенеровані числа a і b призводили до появи колізій, тому генеруємо нові значення: a = 234410, b = 660784.
- Розраховуємо за формулою:
  $h'(key) = ((234410 \cdot keyInt + 660784) \bmod 1000000007) \bmod 9$
- Отримуємо: $22.3333 \rightarrow 0$, $40.1 \rightarrow 5$ та $90 \rightarrow 4$

Результат:

Bucket	Keys
0	-52.6666
1	Second hash: $-10.5 \rightarrow 0$, $0.0078 \rightarrow 1$
3	60.4444
5	Second hash: $0.0001 \rightarrow 2$, $85.9999 \rightarrow 3$
8	Second hash: $22.3333 \rightarrow 0$, $40.1 \rightarrow 5$, $90 \rightarrow 4$

Отже, маємо хеш-таблицю без колізій.

2) Ще один приклад:

Ключі залишимо ті ж самі:

```
vector<double> keys;
keys.push_back(-10.5);
keys.push_back(22.3333);
keys.push_back(0.0078);
keys.push_back(40.1);
keys.push_back(-52.6666);
keys.push_back(60.4444);
keys.push_back(0.0001);
keys.push_back(85.9999);
keys.push_back(90.0);
keys.push_back(22.3333);
```

Поміняємо тестові значення для перевірки:

```
vector<double> tests;
tests.push_back(-52.6666);
tests.push_back(60.4444);
tests.push_back(0.0001);
tests.push_back(85.9998);  // майже як існуючий ключ
tests.push_back(100.0);    // відсутній у початковому списку
tests.push_back(-0.0078);  // знак змінили
tests.push_back(40.1001);  // дуже близько до існуючого ключа
tests.push_back(-99999.9); // далеке значення
tests.push_back(1e-6);     // дуже мале позитивне число
tests.push_back(-1e-6);    // дуже мале негативне число
```

Вивід програми:

```
Testing lookups:
-52.6666 -> Found
60.4444 -> Found
0.0001 -> Found
85.9998 -> Not found
100 -> Not found
-0.0078 -> Not found
40.1001 -> Not found
-99999.9 -> Not found
1e-06 -> Not found
-1e-06 -> Not found
```

Отже, програма працює правильно.

Висновки

Ідеальне хешування - це ефективний спосіб зберігання та пошуку даних, який застосовують в обчислювальній техніці та інформаційних системах. При реалізації даного алгоритму варто мати на увазі, що для різних типів даних, необхідно реалізувати різні хеш-функції, щоб досягти оптимальної продуктивності та точності пошуку.

Використані літературні джерела

- Хеш-функція
- Ідеальна хеш-функція
- Лекція №1 з предмету «Алгоритми та складність»