Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних інформаційних систем

Алгоритми та складність

Лабораторна робота №1

“Реалізація ідеального хешування”

Виконав студент 2-го курсу

Групи ІПС-22

Гончаренко Ілля Сергійович

2024

Завдання:

Реалізувати ідеальне хешування для раціональних чисел за типом даних int.

Предметна область: алгоритми хешування та оптимізація даних.

Примітка: Це включає в себе вивчення методів створення хеш-функцій, їх оптимізацію та застосування для різних типів даних. У даному випадку, ми маємо справу з хешуванням раціональних чисел за типом даних int, що вимагає розробки ефективного алгоритму для конвертації цих чисел у внутрішнє представлення та обчислення хеш-значення..

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.
- *Хеш-таблиця* - це структура даних, в якій всі елементи зберігаються у вигляді пари ключ-значення, де:
  - *ключ* – унікальне число, яке використовується для індексації значень;
  - *значення* - дані, які з цим ключем пов'язані.
- Коли хеш-функція генерує один індекс для кількох ключів, виникає конфлікт: невідомо яке значення потрібно зберегти в цьому індексі. Це називається *колізією хеш-таблиці.*

Алгоритм

Алгоритм складається з таких кроків:

1) Отримуємо набір вхідних даних чисельник типу int та знаменник типу int.
2) Перевіряємо дані на повтори.
3) Створюємо хеш-таблицю, розмір якої дорівнює кількості елементів вхідного набору даних.

4) Для кожного елемента рахуємо його індекс у хеш-таблиці за допомогою функції *на* *h(x)=(a*abs(numerator(x)+b*denominator(x))+b)mod(n),* де x - сума всіх чисел з вектору, num(x)-чисельник x, den(x) - знаменник x. a, b - випадково обрані на початку роботи алгоритму натуральні числа, n - найближче просте число, що є меншим за кількість векторів чисел.
5) 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.
6) Хешуємо кожний елемент підтаблиці. Повторюємо цей пункт доки не буде нових колізій.
7) Виводимо ключі в порядку зростання

Цей код виконує вище описані дії:

```
class Drop:
  int numerator
  int denominator
  Drop(numerator, denominator):
    this.numerator = numerator
    this.denominator = denominator
class H_entry:
  Drop key
  int hash
  H_entry next
  H_entry(key, hash):
    this.key = key
    this.hash = hash
    this.next = null
class HashTable:
  List table
  List keys
  int size
  HashTable(size):
    this.size = size
    this.table = new List of size elements, initialized to null
```

```
    this.keys = new List

 hash(key, i):
    a = 7
    b = 11
    n = findPrime(size)
    return ((a * abs(key.numerator) + b * abs(key.denominator)) + i * b) mod n

 findPrime(n):
    for i from n - 1 down to 2:
        isPrime = true
        for j from 2 up to square root of i:
            if i mod j equals 0:
                isPrime = false
                break
        if isPrime:
            return i
    return 2

 insert(key):
    if getHash(key) is not -1:
        print "Key", key.numerator, "/", key.denominator, "already exists in the hash table."
        return
    i = 0
    do:
        h = hash(key, i)
        if table[h] is null:
            table[h] = new H_entry(key, h)
            keys.append(key)
            return
        i = i + 1
    while i is less than size
    print "Hash table is full, failed to insert key", key.numerator, "/", key.denominator
```

getHash(key):

```
i = 0
do:
    h = hash(key, i)
    entry = table[h]
    while entry is not null:
        if entry.key.numerator equals key.numerator and entry.key.denominator equals key.denominator:
            return entry.hash
        entry = entry.next
    i = i + 1
while table[h] is not null and i is less than size
return -1
```

printHash():

```
sort(keys) using custom comparator function:
    lambda a, b: (double)a.numerator / a.denominator < (double)b.numerator / b.denominator
for key in keys:
    print "Hash for", key.numerator, "/", key.denominator, ":", getHash(key)
```

Складність

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

Мова програмування

C++

Модулі програми

- **int hash(Drop key, int i)**

Ця функція визначає хеш для ключа key у хеш-таблиці.

Пояснення:

1) int a = 7; та int b = 11 - Ці змінні a та b представляють випадково обрані натуральні числа, які використовуються для побудови хешу. Вони впливають на розподіл ключів у таблиці.

2) int n = findPrime(size) Знаходить найближче просте число до розміру таблиці. Це значення n використовується для обмеження результуючого хешу, щоб він не виходив за межі масиву таблиці.

3) ((a * abs(key.numerator) + b * abs(key.denominator)) + i * b) % n - Це основна формула для обчислення хешу. Спочатку обчислюється сума добутків чисельника та знаменника ключа на відповідні коефіцієнти a та b. Потім до отриманої суми додається добуток індексу i на b. Нарешті, результат береться по модулю n, щоб забезпечити, що хеш буде в межах від 0 до n-1, що відповідає індексам таблиці.

Отже, функція hash використовує випадкові коефіцієнти a та b для обчислення хешу ключа, додавання індексу для уникнення колізій та обмеження результату за рахунок найближчого простого числа n.

- **int findPrime(int n)**

Ця функція приймає ціле число n в якості параметра і повертає найбільше просте число, яке менше або рівне n.

Пояснення:

1) for (int i = n - 1; i >= 2; --i) - Цикл ітерується від n - 1 до 2 (включно), шукаючи найбільше просте число менше або рівне n.

2) bool isPrime = true - Змінна isPrime ініціалізується як true, що означає, що ми спочатку припускаємо, що поточне значення i є простим числом.

3) for (int j = 2; j <= sqrt(i); ++j) - Вкладений цикл перевіряє, чи i є простим числом, перебираючи всі числа від 2 до квадратного кореня з i.

4) if (i % j == 0) - Умова перевіряє, чи і ділиться на j без остачі. Якщо умова виконується, це означає, що і не є простим числом, і isPrime встановлюється в false.

5) break - Якщо і не є простим числом, ми завершуємо цикл for і переходимо до наступного значення і.

6) if (isPrime) return i - Після завершення внутрішнього циклу, якщо isPrime залишилося true, це означає, що і є простим числом. Тоді і повертається як результат функції.

7) return 2 - Якщо жодне просте число не було знайдено в діапазоні, функція повертає 2, оскільки це найменше просте число.

Отже, ця функція шукає найбільше просте число, менше або рівне заданому числу n, і повертає його.

- **void insert(Drop key)**

Ця функція призначена для вставки нового ключа key у хеш-таблицю.

Пояснення:

1) if (getHash(key) != -1) - Спочатку перевіряється, чи ключ вже існує у хеш-таблиці, за допомогою функції getHash. Якщо getHash повертає -1, це означає, що ключа ще немає в таблиці, і вставка може продовжуватися. В іншому випадку виводиться повідомлення про те, що ключ вже існує, і функція завершується.

2) int i = 0 - Ініціалізуємо змінну i, яка буде використовуватися для вирішення колізій методом лінійного пошуку.

3) do - Починається цикл do-while, в якому буде виконуватися вставка ключа у таблицю.

4) h = hash(key, i) - Обчислюється хеш для ключа key з використанням функції hash. Змінна i використовується для вирішення колізій.

5) if (table[h] == nullptr) - Перевіряється, чи елемент у таблиці з індексом h є порожнім. Якщо так, це означає, що ця позиція в таблиці є вільною для вставки нового ключа.

6) table[h] = new H_entry(key, h) - Якщо позиція в таблиці вільна, створюється новий об'єкт H_entry, який містить ключ key та відповідний хеш h, і цей об'єкт зберігається у відповідній позиції таблиці.

7) keys.push_back(key) - Ключ також додається до вектора keys, що дозволяє зручно виводити хешування в порядку зростання ключів.

8) return - Після вставки ключа функція завершується.

9) i++ - Якщо позиція в таблиці вже зайнята, змінна i збільшується, і процес повторюється для наступної позиції, використовуючи метод лінійного пошуку.

10) while (i < size) - Цей процес повторюється, поки не буде перевірено всі можливі позиції у таблиці або доки вставка не буде успішною.

11) cerr << "Hash table is full, failed to insert key " << key.numerator << "/" << key.denominator << endl - Якщо усі позиції у таблиці зайняті і вставка не вдалася, виводиться повідомлення про те, що таблиця заповнена, і вставка не вдалася.

- **int getHash(Drop key)**

Ця функція призначена для отримання хешу для певного ключа key в хеш-таблиці.

Пояснення:

1) int i = 0 - Ініціалізуємо змінну i, яка буде використовуватися для вирішення колізій методом лінійного пошуку.

2) do - Починається цикл do-while, в якому буде проводитися пошук ключа у хеш-таблиці.

3)h = hash(key, i) - Обчислюється хеш для ключа key з використанням функції hash. Змінна i використовується для вирішення колізій.

4) H_entry* entry = table[h] - Отримуємо вказівник на перший елемент списку з колізіями для певного хешу h.

5) while (entry != nullptr) - Виконується цикл while, який перебирає всі елементи списку з колізіями.

6) if (entry->key.numerator == key.numerator && entry->key.denominator == key.denominator) - Перевіряється, чи ключ поточного вузла співпадає з шуканим ключем. Якщо так, повертається хеш цього ключа.

7) entry = entry->next - Переходимо до наступного елементу списку з колізіями.

8) i++ -Якщо ключ не знайдено на поточній позиції, збільшуємо значення i, щоб перевірити наступну позицію.

9) while (table[h] != nullptr && i < size) - Цей процес повторюється, поки не буде перевірено всі можливі позиції у таблиці або доки не буде перевірено всі позиції в таблиці або поки не буде знайдено ключ.

10) return -1; // Ключ не знайдено: Якщо ключ не знайдено, функція повертає значення -1, що вказує на те, що ключ відсутній у таблиці.

- **void printHash()**

  Ця функція призначена для виведення хешування ключів у порядку зростання.

  Пояснення:

1) sort(keys.begin(), keys.end(), [](const Drop& a, const Drop& b) { ... }) - Спочатку всі ключі у векторі keys сортуються за допомогою стандартної функції std::sort. У цьому виклику використовується лямбда-функція для порівняння ключів. Ключі порівнюються за їхніми числами, вирахованими як ділення чисельника на знаменник у десятковому форматі, щоб вони були відсортовані у порядку зростання.

2) for (Drop key : keys) - Потім розпочинається цикл, який перебирає всі ключі, що знаходяться у відсортованому векторі keys.

3) cout << "Hash for " << key.numerator << "/" << key.denominator << ": " << getHash(key) << endl - Для кожного ключа виводиться повідомлення, яке містить чисельник і знаменник ключа, а також його хеш, отриманий за допомогою функції getHash.

Отже, ця функція виводить хеш для кожного ключа у хеш-таблиці у порядку зростання значень ключів.

Інтерфейс користувача

Консольний інтерфейс. Всі дані вводяться у main()

Тестові приклади

1. Тест на додавання унікальних ключів:

Ключі додаються у хеш-таблицю, і виводяться в порядку зростання.
Розмір вказаного діапазону 6, найближче просте число до цього числа 5, тому n=5.

Для ключа 3/7: Хеш = ((7 * 3) + (11 * 7) + 0 * 11) % 5 = (21 + 77) % 5 = 98 % 5 = 3
Для ключа 1/5: Хеш = ((7 * 1) + (11 * 5) + 0 * 11) % 5 = (7 + 55) % 5 = 62 % 5 = 2
Для ключа 2/6: Хеш = ((7 * 2) + (11 * 6) + 0 * 11) % 5 = (14 + 66) % 5 = 80 % 5 = 0
Для ключа 3/8: Хеш = ((7 * 3) + (11 * 8) + 0 * 11) % 5 = (21 + 88) % 5 = 109 % 5 = 4

Результат:

Hash for 1/5: 2

Hash for 2/6: 0

Hash for 3/8: 4

Hash for 3/7: 3

Тест додавання унікальних ключів ілюструє можливість вставки унікальних значень у хеш-таблицю та виведення їх хешів у відсортованому порядку. У цьому випадку всі ключі успішно додаються, а їх хеші обчислюються правильно за допомогою функції хешування.

2. Тест на дублювання ключів:

Очікуваний результат: Спроба додати дубльований ключ не приводить до змін у хеш-таблиці. Діапазон таблиці той самий

Для ключа 3/7: Хеш = ((7 * 3) + (11 * 7) + 0 * 11) % 5 = (21 + 77) % 5 = 98 % 5 = 3

Для ключа 1/5: Хеш = ((7 * 1) + (11 * 5) + 0 * 11) % 5 = (7 + 55) % 5 = 62 % 5 = 2

Для ключа 2/6: Хеш = ((7 * 2) + (11 * 6) + 0 * 11) % 5 = (14 + 66) % 5 = 80 % 5 = 0

Для ключа 2/6: Хеш = ((7 * 2) + (11 * 6) + 0 * 11) % 5 = (14 + 66) % 5 = 80 % 5 = 0

Результат:

Key 2/6 already exists in the hash table.

Hash for 1/5: 2

Hash for 2/6: 0

Hash for 3/7: 3

Тест демонструє поведінку при спробі вставити дублюючий ключ. Відповідь включає повідомлення про те, що такий ключ вже присутній у хеш-таблиці, і він не додається повторно. Це підтверджує, що хеш-таблиця коректно виявляє та обробляє дублюючі ключі.

Висновки

Хеш-таблиця: Даний код втілює концепцію хеш-таблиці з відкритим хешуванням. Колізії вирішуються за допомогою методу лінійного зондування, де у випадку конфлікту займається наступна доступна позиція в таблиці. Це стратегічне рішення дозволяє уникнути заторів та забезпечує швидкий доступ до даних.

Хешування: Для перетворення ключа в індекс таблиці використовується функція хешування. У нашому випадку застосовується комбінація двох хеш-функцій, що сприяє зниженню ймовірності колізій.

Керування колізіями: Під час вставки нового ключа у хеш-таблицю перевіряється наявність колізій. У разі виявлення конфлікту використовується метод лінійного зондування для пошуку вільної позиції.

Сортування ключів: Метод printHash() перед виведенням хеш-значень сортує ключі у векторі. Це гарантує вивід хешів у порядку зростання ключів.

Переваги та недоліки: Хеш-таблиці ефективні для пошуку та вставки даних, проте вони можуть стати джерелом проблем у випадку атак на колізії. Невдале розподілення ключів може спричинити значну кількість конфліктів, що знизить швидкодію системи.

Узагальнюючи, даний код реалізує просту, але ефективну хеш-таблицю для зберігання та пошуку даних за ключами.

Література

- https://www.wikidata.uk-ua.nina.az/%D0%93%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D1%96%D1%8F.html
- https://en.wikipedia.org/wiki/Perfect_hash_function
- http://om.univ.kiev.ua/users_upload/15/upload/file/pr_lecture_25.pdf
- https://www.8host.com/blog/xesh-tablica-v-c-c-polnaya-realizaciya/
- https://www.youtube.com/watch?v=__66xMXz7wc&ab_channel=%D0%91%D0%BB%D0%BE%D0%B3%D0%B0%D0%BD