Київський національний університет імені Тараса Шевченка

Факультет комп`ютерних наук та кібернетики

Кафедра інтелектуальних інформаційних систем

Алгоритми та складність

Лабораторна робота №1

“Ідеальне хешування”

Виконав студент 2-го курсу

Групи ІПС-22

Каширець Роман Віталійович

2025

**Завдання**

Реалізувати ідеальне хешування. Тип даних – рядки.

**Теорія**

*Ідеальне хешування:*

**Ідеальне хешування** — це техніка побудови хеш-таблиці, яка **гарантує відсутність колізій** для наперед заданої множини ключів. Вона найкраще підходить для **статичних множин**, які **не змінюються після ініціалізації**.  
1. 🔹 **O(1)** час доступу до елемента навіть у найгіршому випадку.

2. 🔹 Гарантована **відсутність колізій**.

3. 🔹 Використовує дві стадії хешування (двоступеневе ідеальне хешування):

* **Перший рівень**: всі n ключів хешуються в m "відер" (списків) за допомогою випадкової **універсальної хеш-функції**.
* **Другий рівень**: для кожного відра будується окрема вторинна хеш-таблиця з новою хеш-функцією, яка не допускає колізій.

**Алгоритм**

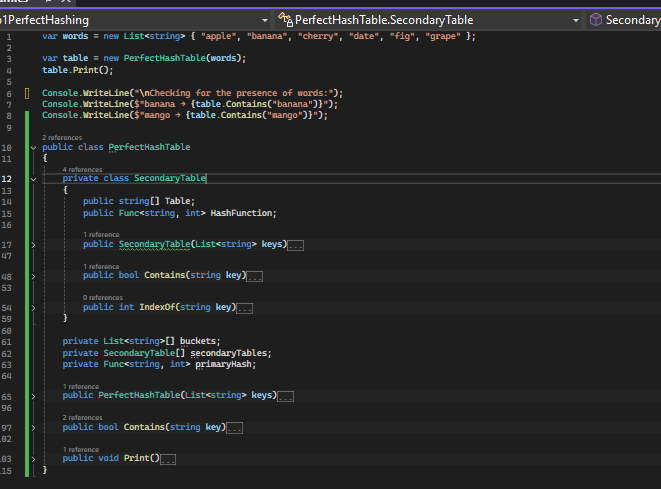
1. **Отримуємо вхідний список ключів (List<string>).**
2. **Формуємо первинну хеш-таблицю (перший рівень):**
   * Вибираємо випадкову хеш-функцію h(s) = ((a \* hash(s) + b) mod p) mod m, де:
     + a і b — випадкові числа,
     + p — велике просте число,
     + m — кількість унікальних ключів.
   * Вставляємо кожен ключ у відповідне "відро" (bucket[i]), тобто в список рядків, який відповідає певному індексу i з m.
3. **Формуємо вторинні таблиці для кожного відра:**
   * Для кожного відра створюємо підтаблицю розміром k², де k — кількість елементів у відрі.
   * Підбираємо нову хеш-функцію так, щоб **не було колізій** у цій таблиці.
   * Якщо при заповненні вторинної таблиці виникає колізія — підбираємо нову хеш-функцію (повторюємо процес).
4. **Пошук:**
   * Знаходимо індекс у первинній таблиці за допомогою первинної хеш-функції.
   * Якщо там є підтаблиця — шукаємо індекс у ній за вторинною хеш-функцією.
   * Якщо ключ знайдено — повертаємо true, інакше — false.

**Складність алгоритму**

**Мова програмування**

C#

**Модулі програми**

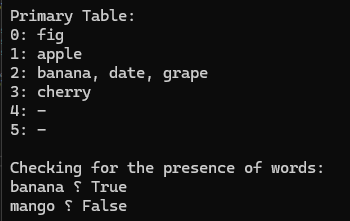
  
**PerfectHashTable(List<string> keys) — конструктор**

1. Створює масив buckets (список списків), де кожен індекс — результат первинного хешування.
2. Для кожного відра створює відповідну SecondaryTable — підтаблицю   
   без колізій.|

### primaryHash — функція первинного хешу

* Розраховується за формулою:  
  ((a \* hash + b) % p) % m
* Використовує GetHashCode() рядка.
* Призначена для розподілу ключів по основному масиву.
* **SecondaryTable(List<string> keys) — конструктор**
* Створює хеш-таблицю розміром n² (n — кількість ключів у відрі).
* Генерує випадкову хеш-функцію.
* Пробує вставити всі ключі.
* Якщо трапляється колізія — повторює з новою функцією, поки не отримає унікальні індекси.
* **🔹 SecondaryTable.Contains(string key)**
* Перевіряє наявність ключа в другорядній таблиці.

**Інтерфейс користувача**



**fig**: 102 + 105 + 103 = 310

**apple**: 97 + 112 + 112 + 108 + 101 = 530

**banana**: 98 + 97 + 110 + 97 + 110 + 97 = 609

**date**: 100 + 97 + 116 + 101 = 414

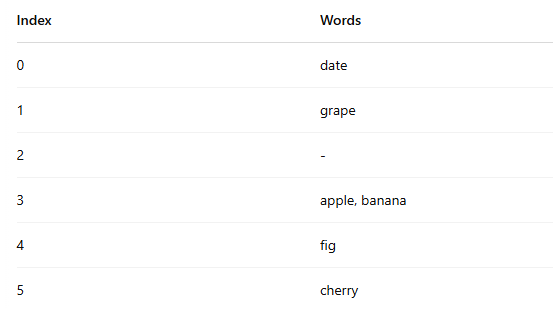
**grape**: 103 + 114 + 97 + 112 + 101 = 527

**cherry**: 99 + 104 + 101 + 114 + 114 + 121 = 653

**Формула:**

((a \* key + b) % p) % m

Рахуємо:

* **fig** → ((3 × 310 + 7) % 101) % 6 = (937 % 101) % 6 = 28 % 6 = 4
* **apple** → ((3 × 530 + 7) % 101) % 6 = (1597 % 101) % 6 = 81 % 6 = 3
* **banana** → ((3 × 609 + 7) % 101) % 6 = (1834 % 101) % 6 = 15 % 6 = 3
* **date** → ((3 × 414 + 7) % 101) % 6 = (1249 % 101) % 6 = 36 % 6 = 0
* **grape** → ((3 × 527 + 7) % 101) % 6 = (1588 % 101) % 6 = 73 % 6 = 1
* **cherry** → ((3 × 653 + 7) % 101) % 6 = (1966 % 101) % 6 = 47 % 6 = 5  
  
* **Для комірки 3 (apple, banana):**
* n = 2 → m′ = n² = 4
* Візьмемо: a′ = 2, b′ = 1, p = 101
* Рахуємо code\_key:
* apple = 530
* banana = 609
* Хеші:
* apple → ((2×530 + 1) % 101) % 4 = (1061 % 101) % 4 = 51 % 4 = 3
* banana → ((2×609 + 1) % 101) % 4 = (1219 % 101) % 4 = 7 % 4 = 3
* Колізія! Міняємо a′ та b′.
* Спробуємо: a′ = 4, b′ = 3
* apple → (4×530 + 3) % 101 = 2123 % 101 = 1 → 1 % 4 = 1
* banana → (4×609 + 3) % 101 = 2439 % 101 = 15 → 15 % 4 = 3

**Висновки**

Ідеальне хешування гарантує відсутність колізій, що забезпечує **константний час доступу** O(1). Його алгоритм не є складним для реалізації, хоча потребує додаткової пам’яті для підтаблиць. Загальна складність алгоритму **лінійна** O(n), що робить його придатним для застосувань, де важливий швидкий пошук і доступ до даних.

**Література**

* Лекція з предмету «Алгоритми та складність #1»
* <https://en.wikipedia.org/wiki/Perfect_hash_function>
* Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms. — 3rd. — MIT Press, 2009.
* https://www.i-programmer.info/babbages-bag/480-hashing-ii.html