## Розділ 10. Алгоритми роботи з хеш-таблицями

**Хеш-таблиця** – це звичайний масив з незвичайною адресацією, що задається хеш-функцією.

**Хеш-функція** – функція, що трансформує ключ у деякий індекс у таблиці.

Ситуація, коли два або більше ключів асоціюються з однією й тією ж коміркою називається *колізією* при хешуванні.

Слід зазначити, що гарною хеш-функцією є така функція, що мінімізує колізії й розподіляє записи рівномірно по всій таблиці.

Хеш-таблиці часто застосовуються в базах даних, і, особливо, у мовних процесорах типу компіляторів і асемблерів, де вони вдало обслуговують таблиці ідентифікаторів. У таких додатках, таблиця – найкраща структура даних.

**Метод відкритого хешування (інша назва: хешування ланцюжками)**

У випадку хешування ланцюжками, елементи з однаковими індексами об'єднуються в зв'язаний список. Наступний рисунок ілюструє це.

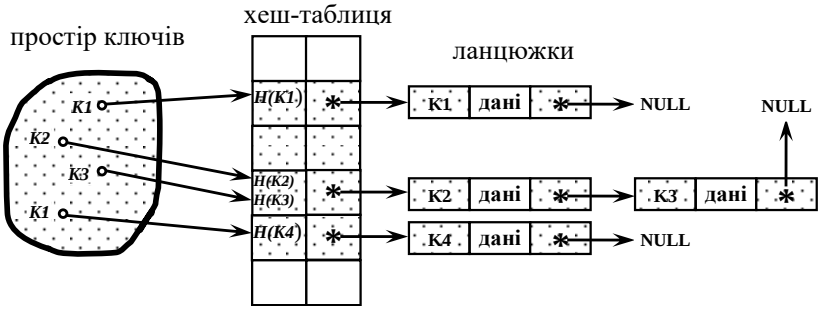


Рисунок 10.1 – Хешування ланцюжками

Тобто, якщо при додаванні в хеш-таблицю в задану комірку ми зустрічаємо посилання на елемент зв'язаного списку, то трапляється колізія. Тоді, ми просто вставляємо наш елемент як вузол у список. При пошуку ми проходимо по ланцюжках, порівнюючи ключі між собою на еквівалентність, поки не доберемося до потрібного. При видаленні ситуація така ж.

Видалення вузла з таблиці, що побудована за методом ланцюжків, полягає просто у виключенні вузла зі зв'язаного списку. Вилучений вузол ніяк не впливає на ефективність алгоритму пошуку. Алгоритм буде працювати так, ніби цей вузол ніколи не вставлявся в таблицю.

Псевдокод алгоритму відкритого хешування:

```
Class node
 begin
    int key
    str st
    link next // посилання на наступний елемент
 end

var link of array mas[0..9]

function int h(int key)
 begin
    h = key mod 10 // mod - залишок від ділення
 end

function link search(int key1, str st1)
begin
    var int i
    link q, link p, link s
    i = h(key1)
    q = nil
    p = mas[i]
    while p != nil do
    begin
        if p.key == key1 then
```

```
            begin
                search = p;
                exit
            end
            q = p
            p = p.link
        end

//Якщо ключ не знайдений, вставляємо новий запис
        new(s)
        s.key = key1
        s.st = st1
        s.next = nil
        if q = nil then
            mas[i] = s
        else
            q.next = s
    search = s
end
```

**Метод закритого хешування (інша назва: відкритої адресації)**

Закриті хеш-таблиці особливо ефективні, коли максимальні розміри вхідного набору даних вже відомі.

У випадку методу закритого хешування всі елементи зберігаються безпосередньо в хеш-таблиці, без використання зв'язаних списків. На відміну від хешування з ланцюжками, при використанні методу відкритої адресації може виникнути ситуація, коли хеш-таблиця виявиться повністю заповненою, так що буде неможливо додавати в неї нові елементи. Так що при виникненні такої ситуації рішенням може бути динамічне збільшення розміру хеш-таблиці, з одночасною її перебудовою.

Для розв’язання колізій застосовуються кілька підходів. Найпростіший з них – це метод *лінійного дослідження*. У цьому випадку при виникненні колізії наступні за поточною комірки перевіряються одна за одною, поки не знайдеться порожня

комірка, куди й записується елемент. А при досягненні останнього індексу таблиці, відбувається перехід на початок, тобто таблиця розглядається як «циклічний» масив. Ілюстрація цього способу представлена на наступному рисунку:

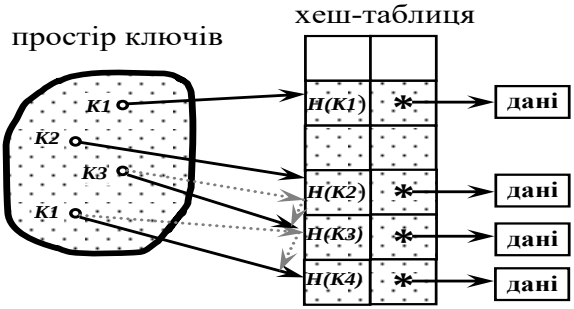


Рисунок 10.2 – Закрите хешування

Лінійне хешування досить просто реалізується, однак з ним зв'язана істотна проблема – *кластеризація*. Це явище створення довгих послідовностей зайнятих комірок, що збільшує середній час пошуку в таблиці. Для зниження ефекту кластеризації використовується інша стратегія вирішення колізій – подвійне хешування. Основна ідея полягає в тому, що для визначення кроку зсуву досліджень при колізії в комірці використовується інша хеш-функція, замість лінійного зсуву на одну позицію.

Розглянемо детальніше реалізацію даного методу. Введемо два масиви:

```
array of T val[0.. n-1]
array of boolean used[0.. n-1]
```

У цих масивах будуть зберігатися елементи множини: вона дорівнює множині всіх val[*i*] для тих і, для яких used[*i*], причому всі ці val[*i*] різні. По можливості будемо зберігати

елемент *t* на місці *h*(*t*), будемо називати це місце «початковим» для даного елементу. Однак може трапитися так, що новий елемент, який ми хочемо додати, претендує на вже зайняте місце (для якого used істинне). У цьому випадку ми відшукаємо найближче праворуч вільне місце й запишемо елемент туди. ("Праворуч" значить "в бік збільшення індексів"; дійшовши до краю, ми перестрибуємо на початок.) За припущенням, число елементів завжди менше n, так що порожні місця гарантовано будуть.

Формально говорячи, у будь-який момент повинна дотримуватися така вимога: для будь-якого елементу множини ділянка праворуч від його «початкового» місця до його фактичного місця повністю заповнена.

Завдяки цьому перевірка приналежності заданого елементу *t* здійснюється легко: вставши на *h*(*t*), рухаємося праворуч, поки не дійдемо до порожнього місця або до елементу *t*. У першому випадку елемент *t* відсутній у множині, у другому є присутнім. Якщо елемент відсутній, то його можна додати на знайдене порожнє місце. Якщо є присутнім, то можна його видалити (привласнивши used = false).

Одним зі складних питань реалізації хешування з відкритою адресацією є операція видалення елементу. Справа в тому, що при видаленні елементу необхідна властивість "відсутності порожнеч" може порушитися. Тому будемо робити так. Створивши пусте місце, будемо рухатися праворуч, поки не натрапимо на ще одне порожнє місце (тоді на цьому можна заспокоїтися) або на елемент, що стоїть не на «початковому» місці. У другому випадку подивимося, чи не потрібно цей елемент поставити на порожнє місце. Якщо ні, то продовжуємо пошук, якщо так, то закриваємо ним порожнє місце, що з’явилося після операції видалення. При цьому утвориться нова діра, з якою робимо все те ж саме.

Псевдокод алгоритму закритого хешування:

```
function int h(int key)
 begin
    var array of int K[0..999]
    h = key mod 1000 // mod - залишок від ділення
 end

function int rh(int i)
 begin
    rh = i + 1 mod 1000 // mod - залишок від ділення
 end

procedure insert(int key)
 begin
    var int i
    i = h(key) //хешуємо ключ
    while ((k(i) != key) and (k(i) != 0)) do
        i = rh(i) //повторне хешування
    if k(i) == 0 then //вставляємо запис у порожню
позицію
        k(i) = key
 end
```

**Вибір хеш-функції**

Звернемося тепер до питання про те, як вибрати гарну хеш-функцію. Ясно, що ця функція повинна створювати якнайменше колізій при хешуванні, тобто вона повинна рівномірно розподіляти ключі на наявні індекси в масиві. Звичайно, не можна визначити, чи буде деяка конкретна хеш-функція розподіляти ключі правильно, якщо ці ключі заздалегідь не відомі. Однак, хоча до вибору хеш-функції рідко відомі самі ключі, деякі властивості цих ключів, які впливають на їхній розподіл, звичайно відомі.

**1) метод ділення.** Деякий цілий ключ ділиться на розмір таблиці й залишок від ділення береться як значення хеш-функції. Ця хеш-функція позначається $h(key) = key \bmod m$.

**2) метод середини квадрата.** Ключ множиться сам на себе і як індекс використовується декілька середніх цифр цього квадрата. Псевдокод:

```
function int h(int key)
 begin
    key = key * key //Піднести до квадрата
    key = key shl 11 //Відкинути 11 молодших бітів
    H = key mod 1024 //Повернути 10 молодших бітів
 end
```

**3) адитивний метод для рядків** (розмір таблиці дорівнює 256). Для рядків цілком розумні результати дає додавання всіх символів і повернення залишку від ділення на 256. Псевдокод:

```
function h(st: string): integer
 begin
    var sum: longint
    i: integer
    for i =0 to length(st) do
        sum = sum + ord(st[i])
    H = sum mod 256;
 end
```

**4) виключаюче АБО для рядків** (розмір таблиці дорівнює 256). Цей метод аналогічний адитивному, але успішно розрізняє схожі слова й анаграми (адитивний метод дасть одне значення для *XY* і *YX*). Метод полягає в тому, що до елементів рядка послідовно застосовується операція "виключаюче або". В алгоритм додається випадковий компонент, щоб ще поліпшити результат. Псевдокод:

```
var
 array of int rand8[0..255]
procedure init
 begin
```

```
    var
     int i
    randomize()
    for i = 0 to 255 do
        rand8[i] = random(255)
 end

function int h(str st)
 begin
    var
     longint sum
     int i
    for i = 0 to length(st) do
        sum = sum + ord(st[i]) xor rand8[i]
    h = sum mod 256 // mod - залишок від ділення
 end
```

**Питання до розділу:**

1. Як хеш-таблиці пов’язані з хеш-функціями?
2. Що таке колізія?
3. Де використовуються хеш-таблиці?
4. Поясніть метод відкритого хешування.
5. Поясніть метод закритого хешування.
6. Як обрати хеш-функцію?