МІНІСТРЕСТВО ОСВІТИ Й НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

“Харківський Політехнічний Інститут”

Кафедра управління проєктами в інформаційних технологіях

Звіт з лабораторної роботи №4

“Однозв’язні та двозв’язні списки й робота з ними”

з дисципліни

“Алгоритми та структури даних”

Варіант №2

Перевірив: ст. викл. каф. УПІТ Мошко Є.О.

Виконав: ст. гр. КН-1223г Ставицький А.А.

Харків – 2024

**Мета:** Вивчити роботу алгоритмів: прямої адресації, хеш-таблиці 1 відкритої адресації. Реалізувати перелічені вище алгоритми і провести їх порівняльний аналіз.

**Порядок виконання роботи:**

1. Написати алгоритми основних операцій (додавання, пошук і видалення) в Псевдокод для прямої адресації (метод ланцюжка), хеш-таблиці і відкритої адресаці.
2. Реалізувати основні операції.
3. Провести аналіз цих алгоритмів 1 зробити висновок по їх застосовності.

**Завдання**

1. Нехай розмір хеш-таблиці дорівнює т = 20, а хеш-функція має вигляд: h(k) = [m(kA mod 1)], где A = (V5 - 1)/2. В які позиції потраплять ключі (див. таблицю, стовпець 3)?
2. 2. Як буде виглядати хеш-таблиця з ланцюжками після того, як в неї послідовно помістили елементи з ключами (див. таблицю, стовпець 2) (в зазначеному порядку)? Число позицій в таблиці дорівнює 9, хеш-функція має вигляд h(k) = k mod 9.
3. Виконайте додавання ключів (в зазначеному порядку, див. таблицю стовпець 2) в хеш-таблицю з відкритою адресацією розміру m = 11. Для обчислення послідовності проб використовується лінійний метод з h'(k) = k mod m. Виконайте те ж завдання, якщо використовується квадратичний метод з тієї ж h', cl = 1, с2 = 3, а також для подвійного хешування з h1 = h' u hz(k) = 1 + (k mod (m - 1)) .

Використовувати готові реалізації структур даних (наприклад, STL)

заборонено, але можна використати реалізацію рядків (наприклад, std::string y C++).

Хеш-таблиця — це структура даних, яка використовується для ефективного зберігання і швидкого доступу до даних на основі ключів. Основний принцип хеш-таблиці полягає у використанні **хеш-функції**, яка обчислює індекс, що вказує на місце зберігання значення у внутрішньому масиві.

**Основні поняття та принципи роботи:**

1. **Ключ і значення**: Кожен елемент в хеш-таблиці представлений парою “ключ-значення”. Ключ унікально ідентифікує значення в таблиці, що дозволяє швидко знайти потрібний елемент.

2. **Хеш-функція**: Це функція, яка обчислює індекс для кожного ключа, дозволяючи розподілити дані у масиві. Хороша хеш-функція рівномірно розподіляє ключі по всій таблиці, що знижує кількість колізій.

3. **Колізії**: Колізія виникає, коли різні ключі мають однаковий індекс після обчислення хеш-функцією. Існують два основних методи обробки колізій:

• **Відкрита адресація**: шукає інше місце у масиві, де можна зберегти значення.

• **Метод ланцюжків**: для кожного індексу таблиці зберігається список значень, що мають однаковий індекс.

4. **Переваги хеш-таблиць**:

•Висока швидкість пошуку, додавання та видалення елементів (у середньому O(1)).

•Зручне зберігання даних, які потрібно швидко отримувати за ключем.

5. **Недоліки**:

•Необхідність правильної хеш-функції, щоб уникнути великої кількості колізій.

•Може знадобитися додатковий простір для зберігання, особливо у разі застосування методу ланцюжків.

**Хеш-таблиця (методом відкритої адресації)**

Метод відкритої адресації є одним зі способів розв’язання колізій у хеш-таблицях. Він передбачає пошук іншого вільного місця у таблиці, якщо індекс, обчислений хеш-функцією, вже зайнятий. Таким чином, усі елементи зберігаються безпосередньо в масиві, а не в окремих списках.

**Основні принципи методу відкритої адресації**

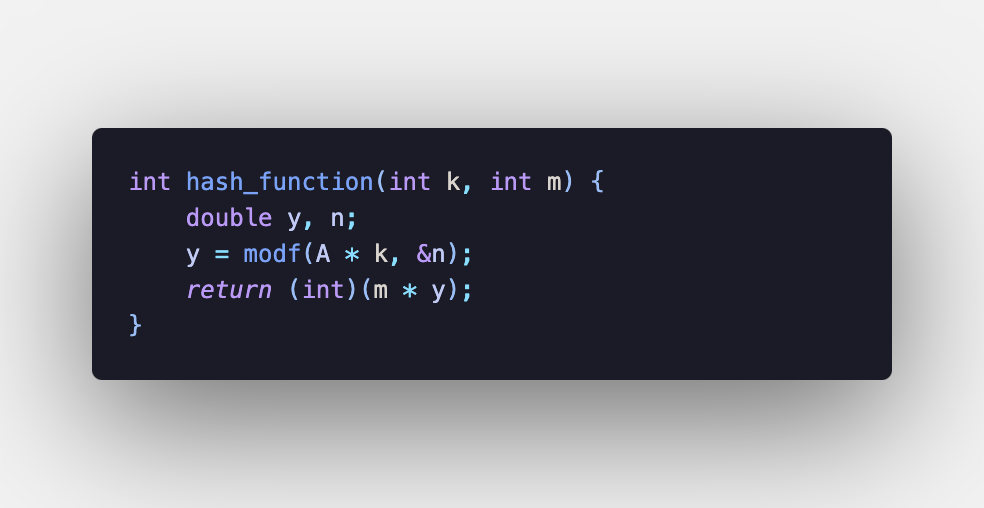
1. **Початкове хешування**: Коли вставляється новий елемент, хеш-функція обчислює індекс, за яким цей елемент має бути збережений. Якщо індекс вільний, елемент займає це місце.

2. **Колізія**: Якщо індекс зайнятий, метод відкритої адресації шукає інший вільний індекс, застосовуючи одну з технік відкритої адресації.

Функція hash\_function обчислює індекс для розміщення значення k у хеш-таблиці розміру m, використовуючи метод множення. Цей метод передбачає застосування певної константи A, яка повинна бути дробовим числом між 0 і 1. Константа A сприяє рівномірному розподілу значень у хеш-таблиці.

Робота функції відбувається так: спершу обчислюється добуток A \* k. Це значення розділяється на цілу та дробову частини за допомогою функції modf із бібліотеки <math.h>. Функція повертає лише дробову частину добутку, позначену як y. Далі значення y множиться на розмір таблиці m і округлюється до цілого типу, щоб отримати індекс у межах від 0 до m - 1.

Метод множення підходить для отримання індексу за рахунок дробової частини добутку A \* k, що забезпечує рівномірніший розподіл індексів для різних значень k(Див. Рис. 1).



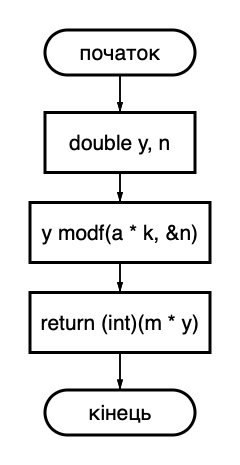


Рис. 1 Хеш-функція hash\_function()

Функція add додає елемент у хеш-таблицю з використанням методу відкритої адресації із лінійним зондуванням для обробки колізій. Розглянемо її покроково(Див. Рис. 2):

1. **Обчислення початкового індексу**: За допомогою hash\_function(key, size) обчислюється початковий індекс h для значення key, враховуючи розмір таблиці size. Це місце, де програма спробує розмістити новий елемент.

2. **Лінійне зондування**: В циклі for (int i = 0; i < size; i++) програма перевіряє size можливих позицій у таблиці, щоб знайти вільне місце. Лінійне зондування означає, що у випадку колізії індекс збільшується на 1 (або іншими словами, індекс обчислюється як (h + 1) % size), що дозволяє послідовно перевіряти наступні позиції.

3. **Додавання елемента**: Якщо знаходиться порожня комірка (hash\_table[h].is\_empty повертає true), в цій комірці зберігається значення key, і прапорець is\_empty змінюється на false, позначаючи комірку як зайняту. Після цього функція завершується.

4. **Повідомлення про заповненість таблиці**: Якщо функція не знаходить вільного місця після size спроб (тобто, всі комірки зайняті), вона виводить повідомлення "Таблиця заповнена!".



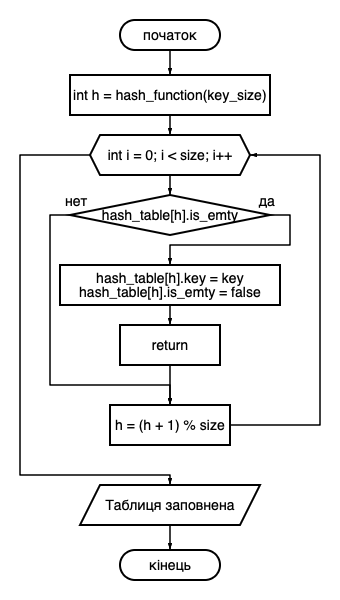


Рис. 2 Функція хеш-таблиці add()

Функція print\_hash\_table виводить вміст хеш-таблиці у зручному для перегляду форматі. Розглянемо її детально(Див. Рис. 3):

1. **Заголовок таблиці**: Спершу виводиться текст "\n

Хеш-таблиця:\n" та горизонтальна лінія "===============================", що створює рамку для таблиці.

2. **Перебір елементів таблиці**: В циклі for (int i = 0; i < size; i++) функція проходить по всіх комірках хеш-таблиці.

3. **Вивід індексу**: Для кожного елемента таблиці виводиться його індекс i і дві табуляції для вирівнювання.

4. **Перевірка на порожню комірку**: Якщо комірка порожня (hash\_table[i].is\_empty дорівнює true), то на екрані відображається слово "Пусто" у кольорі CYAN. Якщо ж комірка заповнена, то виводиться значення ключа (hash\_table[i].key) у кольорі GREEN. В обох випадках використовується RESET для повернення до стандартного кольору після виводу.

5. **Завершення таблиці**: Наприкінці функція виводить ще одну горизонтальну лінію "===============================", щоб завершити рамку.



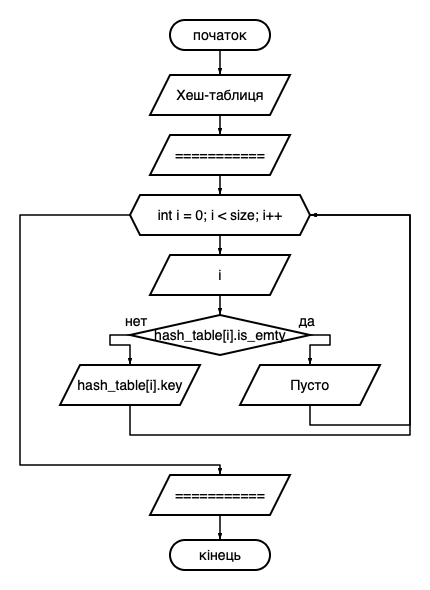


Рис. 3 хеш-функція print\_hash\_table()

Функція delete\_node видаляє вузол із хеш-таблиці за заданим ключем, використовуючи метод лінійного зондування для пошуку(Див. Рис. 4).

1. **Обчислення початкового індексу**: Спочатку функція обчислює початковий індекс для пошуку за допомогою hash\_function(key, size), зберігаючи результат в h.

2. **Лінійне зондування**: За допомогою циклу for функція перебирає всі комірки хеш-таблиці, починаючи з індексу h, і збільшує індекс на 1 у кожній ітерації (h = (h + 1) % size), щоб здійснювати циклічний перехід в межах таблиці.

3. **Перевірка на відповідність ключа**: На кожному кроці перевіряється, чи зберігається в поточній комірці hash\_table[h] шуканий ключ key і чи ця комірка не є порожньою (!hash\_table[h].is\_empty).

• Якщо знайдено комірку з ключем, вона позначається як порожня (hash\_table[h].is\_empty = true;), а функція завершує виконання (return).

4. **Вивід повідомлення про відсутність вузла**: Якщо цикл завершився, але відповідний ключ не було знайдено, функція виводить повідомлення "Вузол не знайдено" червоним кольором (RED), щоб повідомити про невдалий пошук.



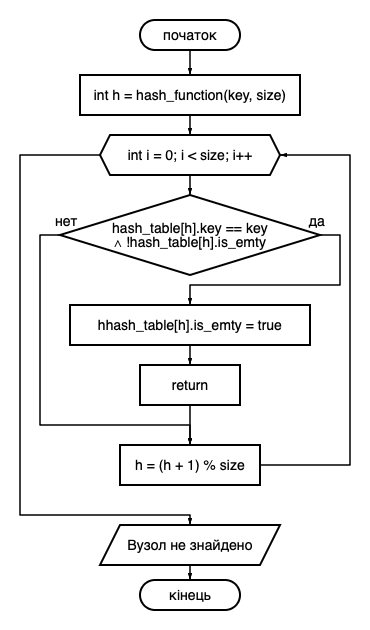


Рис. 4 хеш-функція delete\_node()

Функція search\_node шукає вузол у хеш-таблиці за заданим ключем і виводить повідомлення про його наявність або відсутність, використовуючи метод лінійного зондування для пошуку (Див. рис. 5).

**Пояснення роботи функції**

1. **Обчислення початкового індексу**:

• Використовується хеш-функція hash\_function(key, size), щоб обчислити індекс h, з якого розпочнеться пошук.

2. **Цикл лінійного зондування**:

• За допомогою циклу for, функція перевіряє всі можливі комірки, починаючи з індексу h. На кожному кроці індекс збільшується на 1 (h = (h + 1) % size), що дозволяє обійти таблицю по колу.

3. **Перевірка на наявність ключа**:

• На кожній ітерації перевіряється, чи ключ комірки hash\_table[h] збігається з шуканим значенням key і чи комірка не порожня (!hash\_table[h].is\_empty).

• Якщо ключ знайдено, виводиться повідомлення зеленим кольором (GREEN), і функція завершує виконання.

4. **Завершення пошуку при порожній комірці**:

• Якщо комірка hash\_table[h] порожня (hash\_table[h].is\_empty), пошук припиняється (вихід із циклу for), оскільки в методі відкритої адресації порожня комірка означає, що ключ не існує в таблиці.

5. **Повідомлення про відсутність ключа**:

• Якщо цикл завершується, а ключ не був знайдений, функція виводить червоне повідомлення (RED) про відсутність вузла з таким ключем.



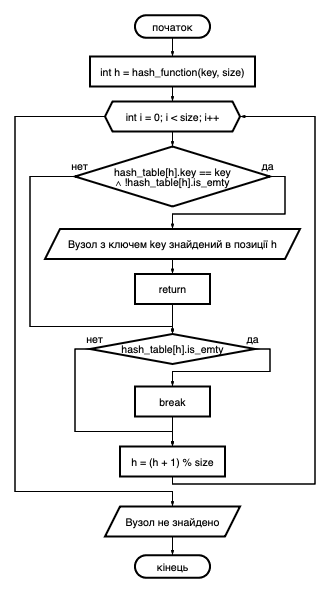


Рис. 5 хеш-функція search\_node()

**Хеш-таблиця (методом ланцюжків)**

**Метод ланцюжків** — це спосіб обробки колізій у хеш-таблиці, коли два або більше ключів мають однаковий хеш-індекс. У цьому методі кожна комірка хеш-таблиці містить не один елемент, а список (або ланцюжок) усіх елементів, що мають однаковий хеш-індекс. Таким чином, колізії обробляються шляхом додавання елементів до цього списку.

**Як працює метод ланцюжків**

1. **Хешування ключа**:

• Хеш-функція обчислює індекс для кожного ключа. Якщо кілька ключів отримують однаковий індекс, ці ключі будуть зберігатися у вигляді списку (або ланцюжка) в одній комірці таблиці.

2. **Структура комірок**:

• Кожна комірка хеш-таблиці містить посилання на список або ланцюжок вузлів. Кожен вузол цього списку містить сам ключ і, можливо, значення, а також посилання на наступний елемент у ланцюжку.

3. **Додавання елементів**:

• При додаванні нового елемента функція хешування обчислює індекс і додає елемент у відповідний ланцюжок. Зазвичай новий елемент додається на початок або в кінець ланцюжка, в залежності від реалізації.

4. **Пошук елементів**:

• Пошук елемента починається з обчислення індексу за допомогою хеш-функції. Потім алгоритм проходить по ланцюжку в цій комірці, порівнюючи ключі. Якщо ключ знайдено, повертається відповідне значення або індекс.

5. **Видалення елементів**:

• Для видалення елемента алгоритм обчислює індекс і проходить по ланцюжку, поки не знайде вузол з відповідним ключем. Потім вузол видаляється зі списку, і зв’язки в ланцюжку оновлюються.

**Переваги та недоліки методу ланцюжків**

**Переваги**:

• Легко реалізується.

• Розмір хеш-таблиці не впливає на кількість елементів, які можуть зберігатися.

• При гарній хеш-функції ланцюжки залишаються короткими, і час пошуку, вставки, та видалення залишається сталим.

**Недоліки**:

• Якщо багато елементів мають однаковий індекс, списки можуть стати довгими, що знижує ефективність.

• Додаткова пам’ять для зберігання вказівників у кожному вузлі списку.

Функція add\_2 додає новий елемент до хеш-таблиці, використовуючи метод ланцюжків для обробки колізій (Див. рис. 6).

1. Спочатку обчислюється хеш-індекс для заданого ключа за допомогою функції хешування hash\_function\_2. Отриманий індекс вказує на позицію в масиві хеш-таблиці.

2. Якщо комірка за цим індексом порожня (тобто в ній ще не зберігаються жодні дані), то новий елемент додається прямо в цю комірку. Для цього ключ зберігається в цій комірці, а поле, що позначає, чи є комірка порожньою, встановлюється в false.

3. Якщо комірка за індексом вже містить елемент (відбувається колізія), то додається новий елемент до кінця ланцюжка. Для цього:

• Спочатку проходимо по всіх елементах ланцюга, починаючи з поточної комірки, поки не дійдемо до останнього елемента.

• Потім створюємо новий вузол для ланцюга, де зберігається новий ключ, і додаємо його в кінець ланцюга.

Цей підхід дозволяє обробляти колізії, не перезаписуючи існуючі елементи, а додаючи нові елементи до ланцюга, що зберігається в кожній комірці хеш-таблиці.



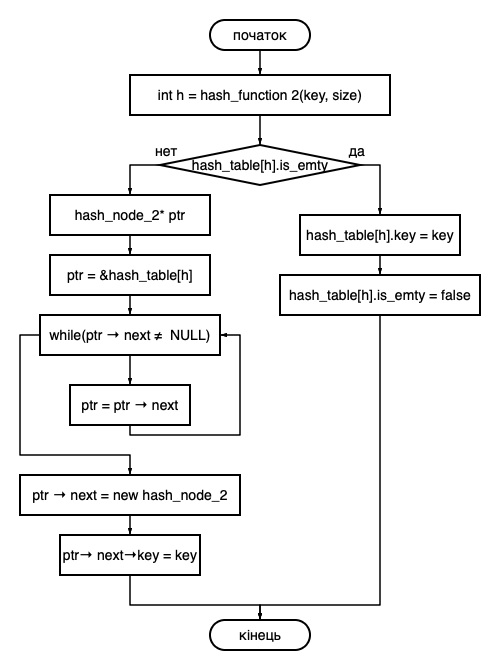


Рис. 6 хеш-функція add\_2()

Функція print\_hash\_table\_2 виводить хеш-таблицю, яка реалізована методом ланцюжків, на екран. Ось покроковий опис її роботи(Див. Рис. 7):

1. Спочатку виводиться заголовок “Хеш-таблиця” і лінія розділення.

2. Далі функція проходить через кожну комірку хеш-таблиці від 0 до size-1:

• Для кожної комірки виводиться її індекс.

• Якщо комірка порожня (тобто в ній немає жодного елемента, поле is\_empty встановлено в true), виводиться повідомлення “Пусто” з кольоровим оформленням.

• Якщо комірка не порожня, то:

• Спочатку виводиться фігурна дужка { для позначення початку ланцюга.

• Далі, поки є елементи в ланцюзі, виводяться їхні значення (ключі), які розділяються стрілкою ->, якщо це не останній елемент.

• Після останнього елемента виводиться закриваюча фігурна дужка }.

1. Після виведення всіх елементів хеш-таблиці виводиться лінія розділення.



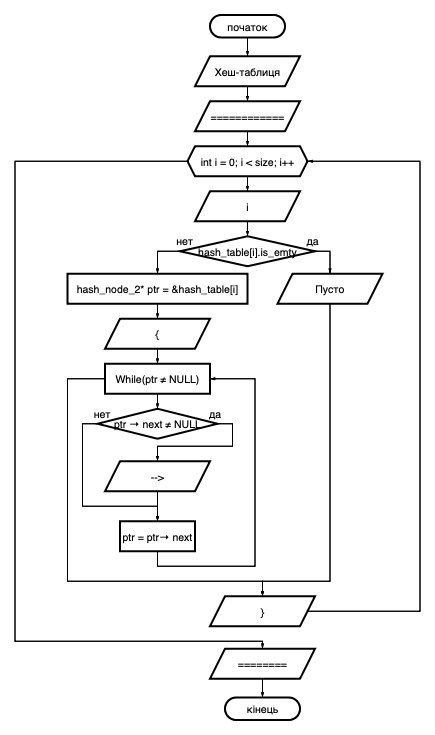


Рис. 7 хеш-функція print\_hash\_table\_2

Функція delete\_node\_2 реалізує видалення вузла з хеш-таблиці, яка використовує метод ланцюжків. Ось покроковий опис її роботи (Див. рис. 8):

1. **Обчислення хеш-індексу**: Спочатку за допомогою хеш-функції (hash\_function\_2) обчислюється індекс для ключа, який потрібно видалити, в масиві хеш-таблиці.

2. **Перевірка на порожню комірку**:

• Якщо виявлено, що ланцюжок на індексі h є порожнім (відсутні елементи), функція виводить повідомлення про те, що вузол не знайдено, і завершується.

3. **Видалення першого елемента в ланцюзі**:

• Якщо ланцюжок містить лише один елемент, то перевіряється, чи є цей елемент тим, який потрібно видалити. Якщо так, вузол позначається як порожній (у полі is\_empty встановлюється true), і функція завершується.

• Якщо це не той елемент, виводиться повідомлення про відсутність вузла в таблиці.

4. **Видалення першого елемента, якщо це той, що шукається**:

• Якщо перший елемент ланцюга — це той вузол, який потрібно видалити, його ключ замінюється на ключ наступного елемента ланцюга. Після цього поточний вузол виводиться з ланцюга, а пам’ять для нього звільняється.

5. **Пошук і видалення вузла в середині ланцюга**:

• Функція проходить по ланцюжку елементів, поки не знайде вузол з потрібним ключем. Якщо такий вузол знайдений, вказівник попереднього елемента (tmp) перенаправляється на наступний елемент ланцюга, тим самим видаляючи поточний вузол. Пам’ять для цього вузла звільняється.

6. **Якщо вузол не знайдено**:

• Якщо по завершенні циклу не вдалося знайти вузол з таким ключем, виводиться повідомлення про те, що вузол не знайдено.



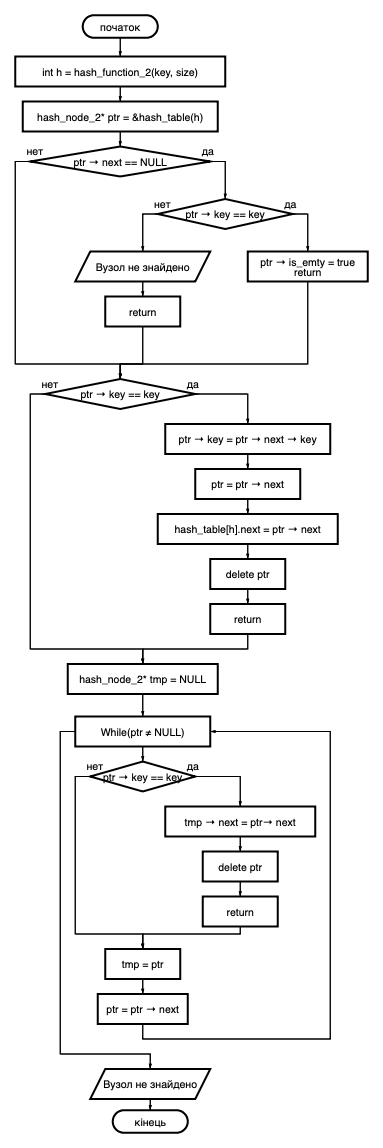


Рис. 8 хеш-функція delete\_node\_2()

**Лінійний алгоритм**

**Лінійний алгоритм** — це алгоритм, складність якого прямо пропорційна розміру вхідних даних, тобто час виконання алгоритму збільшується лінійно при збільшенні кількості елементів. Іншими словами, для кожного елемента вхідних даних виконується певна операція, і загальний час виконання буде пропорційним кількості цих елементів.

У контексті хеш-таблиці з відкритою адресацією (наприклад, лінійне зондування) лінійний алгоритм працює таким чином:

1. **Пошук елемента**: Для пошуку елемента в хеш-таблиці використовується хеш-функція для визначення початкової позиції. Якщо елемент за цією позицією не знайдений (або позиція вже зайнята іншим елементом), перевіряється наступна комірка, і так далі до тих пір, поки не буде знайдений шуканий елемент або не буде знайдена порожня комірка. Кількість операцій у найгіршому випадку пропорційна кількості елементів у хеш-таблиці.

2. **Вставка елемента**: Для вставки елемента в таблицю спочатку виконується пошук відповідної порожньої комірки (якщо на місці, визначеному хеш-функцією, уже є елемент, то перевіряється наступна комірка). Якщо таблиця заповнена, то алгоритм може пройти через усі елементи таблиці, перш ніж знайде порожнє місце для нового елемента.

3. **Видалення елемента**: Як і при пошуку, видалення елемента також вимагає лінійного проходу через комірки таблиці для знаходження елемента з відповідним ключем. Якщо такий елемент знайдений, він видаляється, а для відновлення зв’язків в таблиці інші елементи можуть бути переміщені.

Алгоритм лінійного зондування для вставки елемента в хеш-таблицю. Ось детальний опис функції(Див. рис. 9):

**Опис функції add\_linar:**

1. **Ініціалізація хеш-індексу**:

• Спочатку обчислюється хеш-індекс для елемента за допомогою хеш-функції hash\_function1. Це значення індексу використовується для пошуку відповідної позиції в хеш-таблиці.

2. **Лінійне зондування**:

• Потім запускається цикл, який перевіряє комірки хеш-таблиці. Цикл працює не більше size разів (розмір хеш-таблиці).

• Якщо поточна комірка порожня (hash\_table[h].is\_empty), то елемент вставляється в цю комірку. Відповідний ключ (key) записується в таблицю, і позначається, що комірка більше не є порожньою.

• Якщо комірка не порожня, індекс змінюється на наступний за допомогою h++ (це лінійне збільшення індексу).

• Індекс також нормалізується через операцію h = h % size, щоб у разі досягнення кінця таблиці перейти на початок.

3. **Обробка випадку, коли таблиця заповнена**:

• Якщо в результаті циклу не було знайдено вільного місця для елемента, виводиться повідомлення "Таблиця заповнена".



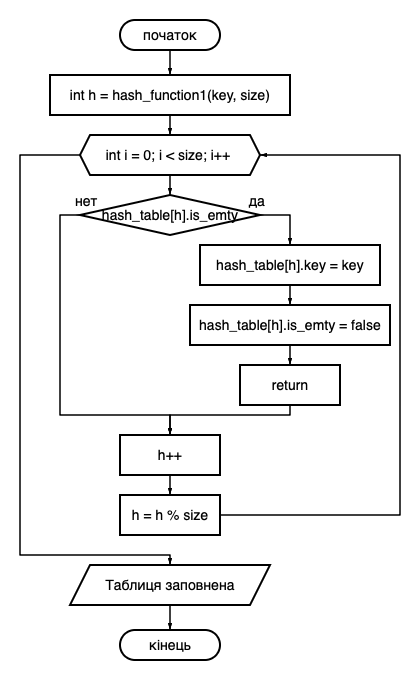


Рис. 9 хеш-функція add\_linar()

Функція print\_hash\_table3 виводить вміст хеш-таблиці, перевіряючи кожен елемент на порожність і виводячи ключі або повідомлення про порожні місця (Див. рис. 10).

**Опис функції:**

1. **Виведення заголовка**:

• Спочатку виводиться заголовок таблиці: "\nХеш-таблиця:\n===============================\n", що форматує виведення для зручності.

2. **Перевірка кожного елемента таблиці**:

• Далі йде цикл, який проходить через всі елементи хеш-таблиці від 0 до size - 1. Для кожного елемента таблиці:

• Якщо елемент порожній (hash\_table[i].is\_empty), то виводиться "Пусто" (за допомогою стилізації кольору через CYAN).

• Якщо елемент не порожній, то виводиться ключ, що зберігається в цьому елементі (hash\_table[i].key), з кольором GREEN для підвищення читабельності.

3. **Виведення роздільника в кінці**:

• Після проходження по всіх елементах таблиці, виводиться ще один роздільник: "===============================\n", щоб завершити виведення.



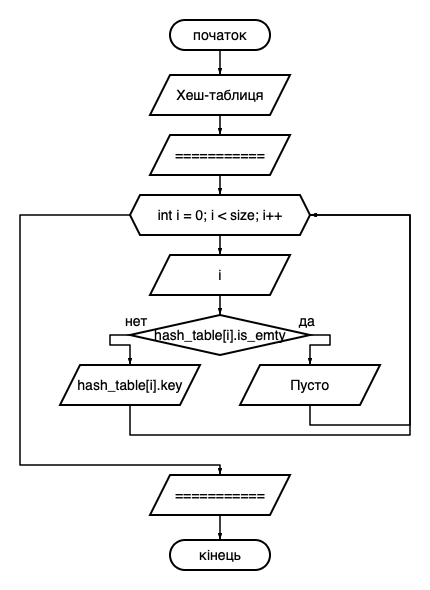


Рис. 10 хеш-функція print\_hash\_table\_3()

**Квадратичний алгоритм**

**Квадратичне зондування** — це метод для розв’язання колізій у хеш-таблицях. Коли два або більше елементів мають однаковий хеш-індекс, квадратичне зондування дозволяє уникнути цієї колізії, змінюючи індекс, за яким шукається вільна позиція, за допомогою квадратичного виразу.

Замість того, щоб просто перевіряти наступний індекс, як у лінійному зондуванні, в квадратичному зондуванні збільшується відстань між індексами пропорційно квадрату числа, що збільшується з кожною новою спробою.

Алгоритм працює наступним чином:

1. Спочатку для ключа обчислюється його хеш-індекс за допомогою хеш-функції.

2. Якщо на цьому індексі вже є елемент, відбувається квадратичне зондування. Наступні перевірки відбуваються за індексами:

• Перший крок: h + 1^2,

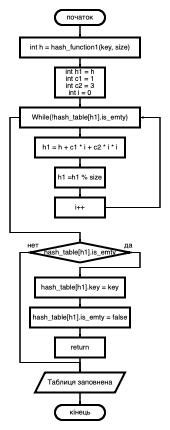
• Другий крок: h + 2^2,

• Третій крок: h + 3^2, і так далі.

3. Якщо на новому індексі теж є елемент, ми перевіряємо наступний індекс за цією ж формулою, поки не знайдемо порожнє місце або не обійдемо всю таблицю.

Таким чином, квадратичне зондування дозволяє зменшити ймовірність формування груп зайнятих місць (кластерів), що характерно для лінійного зондування.





**Висновки:** Під час виконання лабораторної роботи я ознайомився з принципами роботи хеш-таблиць та методами розв’язання колізій, такими як лінійне зондування та метод ланцюжків. Я навчився реалізовувати ці методи у програмному коді для забезпечення ефективного зберігання та пошуку елементів у хеш-таблицях. Особливу увагу я приділив розумінню алгоритмів, що застосовуються для обробки колізій, а також важливості правильно обраної хеш-функції для досягнення оптимальної продуктивності.

У ході роботи я також реалізував функції для додавання, видалення та пошуку елементів у хеш-таблиці, випробував різні методи пошуку та спостерігав за ефективністю кожного з них. Цей досвід допоміг мені краще зрозуміти, як працюють хеш-таблиці в реальних програмах, а також як можна оптимізувати їх роботу залежно від конкретних задач.

**Метод відкритої адресації (лінійне зондування)**

Плюси:

1. Менше пам’яті: Оскільки елементи зберігаються безпосередньо в масиві, не потрібна додаткова пам’ять для зберігання вказівників (як у методі ланцюжків).

2. Простота реалізації: Алгоритм лінійного зондування досить простий для реалізації, оскільки потрібно лише перевіряти наступні позиції в масиві до того моменту, поки не знайдеться вільне місце.

3. Менше накладних витрат: Відсутність додаткових структур даних може зменшити накладні витрати.

Мінуси:

1. Погіршення ефективності при великій кількості колізій: Якщо багато елементів мають однаковий хеш, це може призвести до серйозних проблем з ефективністю, оскільки перевірки можуть займати багато часу (в гіршому випадку – лінійний пошук).

2. Погіршення продуктивності при заповненні таблиці: Якщо таблиця стає заповненою на 70% і більше, час доступу до елементів може значно збільшитися.

3. Не дуже ефективний для великих даних: Якщо кількість елементів значно перевищує кількість місць у таблиці, метод стає менш ефективним через великі витрати часу на пошук вільного місця.

**Метод ланцюжків**

Плюси:

1. Стійкість до колізій: Якщо елементи з однаковим хешем розміщуються у списках, це дозволяє уникнути погіршення ефективності при колізіях.

2. Гнучкість у розмірі таблиці: Таблиця може рости динамічно залежно від кількості елементів, що зберігаються.

3. Простота в обробці великих наборів даних: Завдяки використанню списків для зберігання елементів, метод ланцюжків може бути ефективним навіть для великих таблиць.

Мінуси:

1. Використання додаткової пам’яті: Для зберігання кожного елемента необхідна додаткова пам’ять для вказівників на елементи списку.

2. Може стати повільним при великих списках: Якщо багато елементів мають однаковий хеш, доступ до них може стати повільним, оскільки потрібно пройти через весь список.

3. Не завжди добре для обмежених ресурсів: Залежно від реалізації, метод може використовувати значно більше пам’яті, що важливо в умовах обмежених ресурсів.

В результаті виконання лабораторної роботи я набув навичок реалізації основних операцій хеш-таблиць, а також розуміння їхнього застосування для вирішення задач, що вимагають швидкого доступу до даних.