Open Addressing

Хеш таблицата е структура от данни, която е съставена от два взаимосвързани елемента - ключ и стойност. При хеш таблицата имаме хеш функция, която изчислява индекс, който също се нарича и „хеш код“. По принцип този индекс е уникален и указва позицията, в която се съхранява дадена информация, но повечето хеш таблици употребяват неперфектни функции, което може да доведе до дублиране на индекси (генериране на един и същи индекс два или повече пъти). Този проблем е познат като „колизия“ и добрата новина е, че той е отстраним.
Затова сега ще се запознаем с част от начините, по които можем да отстраним колизиите.

1. Linear probing

Linear Probing е една от техниките за разрешаване на колизии в хеш таблиците, когато два различни ключа се хешират до един и същи индекс. В случай на колизия при линейното пробване, програмата се премества към следващия свободен елемент в масива, докато открие подходящ индекс.

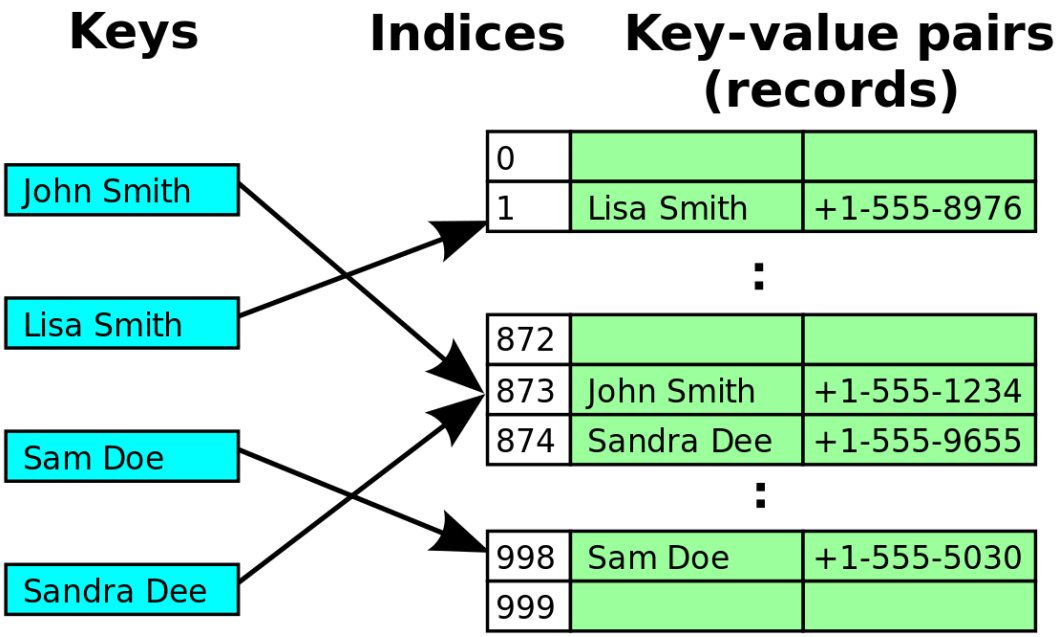
Ето как работи линейното пробване:

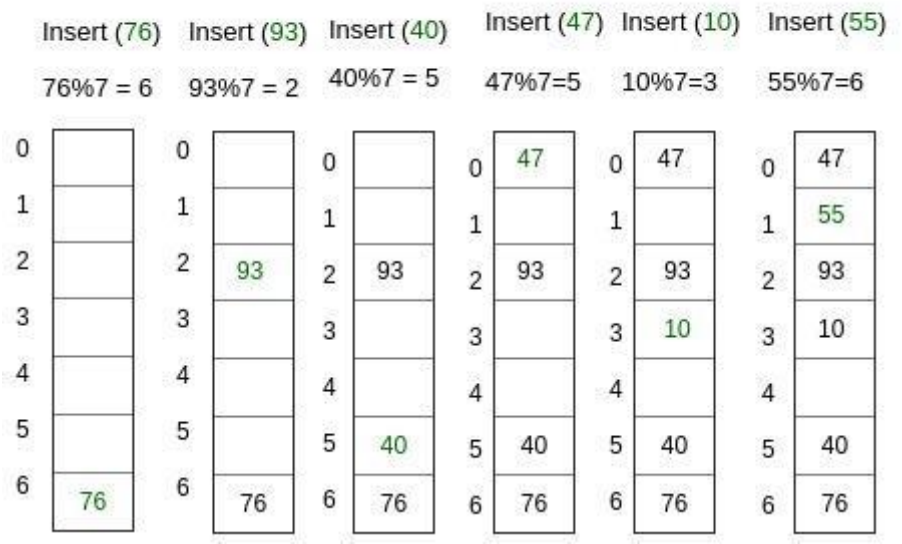
1. **Хеширане на ключа:**
    - Използва се хеш функцията, която превръща ключа в индекс в масива.
2. **Проверка за колизия:**
    - Ако на този индекс вече има стойност, това означава, че се е получила колизия.
3. **Линейно пробване:**
    - Програмата се премества към следващия индекс в масива.
    - Ако и на този нов индекс има стойност, тя отново се премества към следващия индекс.

- Този процес продължава докато се намери свободен индекс.

4. **Вмъкване на стойността:**
    - Когато бъде намерен свободен индекс, стойността се вмъква на този индекс в масива.

„Linear probing“ е проста и лесна за реализация техника, но може да страда от проблема "клъстеризация", при който се образуват групи от последователни заети индекси. Това може да доведе до намаляване на ефективността на търсене, тъй като при увеличаване на големината на клъстера, вероятността за нови колизии се увеличава.





Предимства:

- Лесна за реализация и разбиране.
- Просто линейно търсене в масива.

Недостатъци:

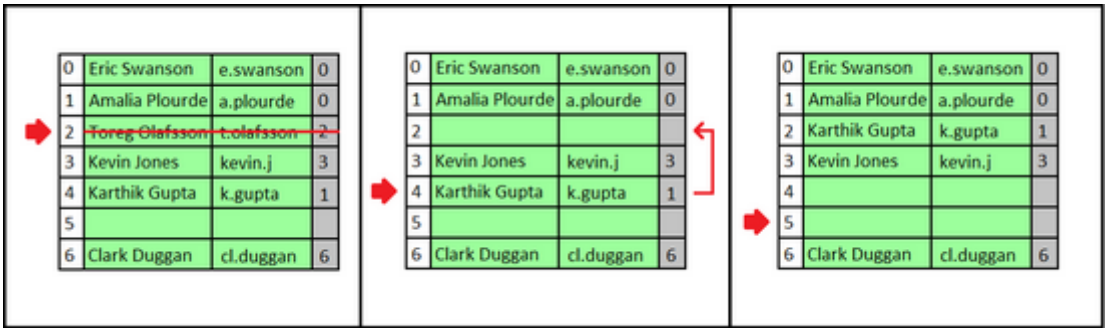
- Начупване (кластеризация) на данните при формиране на групи от последователни заети индекси.
- Възможност за загуба на ефективност при голям брой колизии.

Използване: Линейното пробване се използва в различни приложения и сценарии, където се изисква бърз и прост механизъм за разрешаване на колизии в хеш таблиците. При внимателно избиране на хеш функцията и правилна настройка, линейното пробване може да бъде ефективен метод за управление на колизии в програмите.

Time complexity

- Insertion: O(1) – O(n)
- Search: O(1) – O(n)
- Deletion: O(1)-O(n)

Обновяване при изтриване:



Приложения на линейното пробване:

- Символни таблици: Линейното пробване се използва често в таблиците със символи, които се използват в компилатори и интерпретатори за съхранение на променливи и техните асоциирани стойности. Тъй като символните таблици могат да растат динамично, линейното пробване може да се използва за управление на колизии и гарантиране на ефективно съхранение на променливите.

  (In computer science, a **symbol table** is a data structure used by a language translator such as a compiler or interpreter, where each identifier (or symbol), constant, procedure and function in a program's source code is associated with information relating to its declaration or appearance in the source. In other words, the entries of a symbol table store the information related to the entry's corresponding symbol.)

Operation	Function
allocate	to allocate a new empty symbol table
free	to remove all entries and free storage of symbol table
lookup	to search for a name and return pointer to its entry
insert	to insert a name in a symbol table and return a pointer to its entry
set_attribute	to associate an attribute with a given entry
get_attribute	to get an attribute associated with a given entry

- Кеширане: Линейното пробване може да се използва в системи за кеширане за съхранение на често достъпвани данни в паметта. При настъпване на липса в кеша, данните могат да се зареждат в кеша чрез линейно пробване, а при колизия следващият свободен слот в кеша може да се използва за съхранение на данните.

- Бази данни: Линейното пробване може да се използва в бази данни за съхранение на записи и техните асоциирани ключове. При настъпване на колизия, линейното пробване може да се използва за намиране на следващия свободен слот за съхранение на записа.

- Проектиране на компилатори: Линейното пробване може да се използва в проектирането на компилатори за имплементация на таблиците със символи, механизми за възстановяване от грешки и синтактичен анализ.

- Проверка на правопис: Линейното пробване може да се използва в софтуер за проверка на правописа за съхранение на речника от думи и техните асоциирани честотни броене. При настъпване на колизия, линейното пробване може да се използва за съхранение на думата в следващия свободен слот.

2. Quadratic probing

Quadratic probing (квадратично сондиране) е метод за справяне с колизията. При използване на linear probing ние трябва да преминем през всички слотове, за да намерим първия свободен. Това може да доведе до натрупване (clustering). По тази причина ползваме quadratic probing - за да не се налага да минаваме през всички слотове. Нарича се още the mid-square method. При този процес взимаме индекса на слота, който е на позиция i^2 по време на i-тата итерация. Започваме от началото на таблицата и само ако слота е зает, търсим следващия свободен по следната формула:

Ако hash(x) % S е зает проверяваме (hash(x) + 1*1) % S

Ако (hash(x) + 1*1) % S е зает проверяваме (hash(x) + 2*2) % S

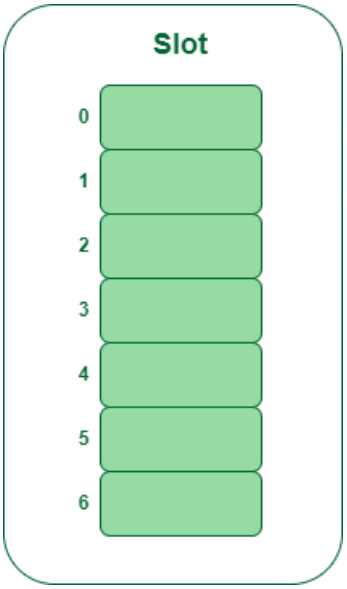
Ако (hash(x) + 2*2) % S е зает проверяваме (hash(x) + 3*3) %

И продължаваме по същия начин до края

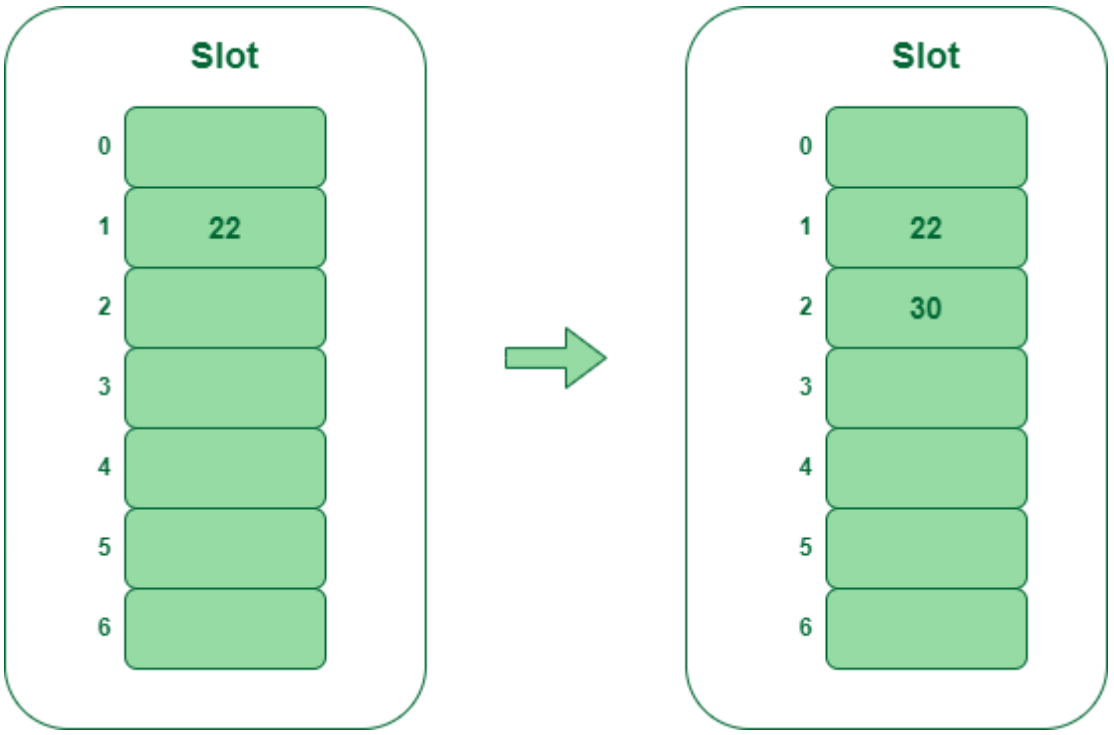
S - размерът на таблицата, hash(x) - първоначалния индекс на слота, изчислен чрез хеш функция

Пример:

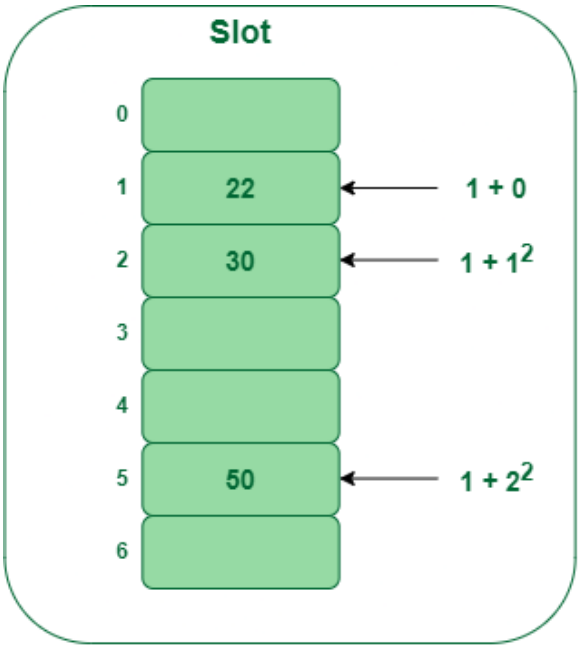
Ако имаме таблица с размер 7. Индексираме я с номерата от 0 до 6. Числата, които искаме да добавим са 22, 30 и 50.



Започваме като делим първото число на размера на таблицата - 22 % 7 = 1. Следователно поставяме 22 на индекс едно, ако е свободен. По същия начин си добавяме 30 на индекс 2.



Когато обаче искаме да добавим 50 индекс 1 вече е зает, затова проверяваме 1 + 1^2, тоест 2. Понеже и той е зает проверяваме 1 + 2^2 = 5. Клетката с индекс 5 е свободна и можем да добавим 50 там.



Методът е средно добър в сравнение с linear probing и double hashing. Проверява по-малко слотове, но все пак може отново да се образува

натрупване (secondary clustering) и да останат празни слотове, които не можем да обходим.

Линк към примерна имплементация:
https://www.geeksforgeeks.org/quadratic-probing-in-hashing/

3.Double Hashing

Това е техника за решаване на колизии, при която се използва втора хеш функция. При колизия се изчислява нов индекс чрез комбиниране на резултата от първата хеш функция и стъпката, която се изчислява чрез втората хеш функция. Този процес се повтаря с определен интервал(често с променлива стъпка), докато се намери свободна позиция. Това помага да се избегнат „клъстъри“ от колизии, където множество ключове се събират около един индекс, като по този начин се осигурява равномерно разпределение на елементите в таблицата и подобрява производителността на хеш таблицата.

Предимства на двойното хеширане:

Равномерно разпределение: двойното хеширане може да поддържа равномерно разпределение на елементите в хеш таблицата.
Използването на втора хеш функция допълнително помага в предотвратяването на формирането на клъстери от колизии.
Ефективност при натоварване: При правилно избрани хеш функции, двойното хеширане може да бъде ефективно дори при високо натоварване на хеш таблицата.
Проста реализация: Въпреки че двойното хеширане включва използването на втора хеш функция, реализацията му обикновено е по-проста в сравнение с някои други методи за решаване на колизии.

Недостатъци на двойното хеширане:

Използват се две хеш функции: Първо, това изисква използването на две хеш функции, което може да увеличи изчислителната сложност на операциите за вмъкване и търсене.

Изисква добър избор на хеш функции: За да бъде ефективно, двойното хеширане изисква добър избор на хеш функции. Ако те не са добре проектирани, честотата на сблъсъци все още може да е висока, което води до намалена ефективност при търсене и вмъкване.

Операции, където можем да използваме двойно хеширане:

Вмъкване(Insertion):

1. Хеширане с първата хеш функция – изчисляваме индекс в таблицата чрез първата хеш функция.
2. Проверка за колизии – ако съответният индекс вече е зает, това е колизия и трябва да се изпозлва втората хеш функция за генериране на нов индекс.
3. Продължаваме да изчисляваме нов индекс, докато намерим свободно място в таблицата. Новият индекс се изчислява чрез комбиниране на първоначалния индекс и стъпката, която се изчислява чрез втората хеш функция.
4. Вмъкване на елемента – когато намерим свободна позиция, поставяме новия елемент на този индекс.

Търсене(Search):

1. Хеширане с първата хеш функция – изчисляваме индекс в таблицата чрез първата хеш функция.
2. Проверяваме дали елементът със съответния индекс съвпада с търсения ключ. Ако елементът е намерен, операцията приключва успешно.
3. Ако индексът е зает от друг елемент, изчисляваме нов индекс, използвайки втората хеш функция. Продължаваме да търсим, докато намерим търсения ключ или свободна позиция.

Изтриване(Deletion):

1. Използваме процеса на търсене, за да намерим индекса на елемента.
2. Изтриваме елемента – ако елементът е намерен, изтриваме го от таблицата.
3. След изтриването на елемента, ако на неговия индекс има друг елемент с нов хеш, изчисляваме нов индекс, използвайки

втората хеш функция. Продължаваме да търсим, докато намерим свободна позиция или стигнем началния индекс.

Пример: Нека вмъкнем ключовете 27, 43, 692, 72 в хеш таблицата с размер 7, където първата хеш функция е **h1(k) = mod k 7**, а втората хеш функция е **h2(k) = 1 + (mod k 5).**

0	1	2	3	4	5	6

Стъпка 1: insert(27)

27 % 7 = 6, място 6 е свободно, така че вмъкваме 27 там.

0	1	2	3	4	5	6
						27

Стъпка 2: insert(43)

43 % 7 = 1, място 1 е свободно, така че вмъкваме 43 там.

0	1	2	3	4	5	6
	43					27

Стъпка 3: insert(692)

692 % 7 = 6, но място 6 вече е заето.

Така че трябва да разрешим тази колизия с помощта на двойно хеширане.

h = [h1(692) + i * (h2(692)] % 7

= [6 + 1 * (1 + 692 % 5)] % 7

= 9% 7 = 2, който е празен, затова вмъкваме в място 2.

0	1	2	3	4	5	6
	43	692				27

Стъпка 4: insert(72)

72 % 7 = 2, но място 2 вече е заето.

Така че трябва да разрешим тази колизия с помощта на двойно хеширане.

h = [h1(72) + i * (h2(72)] % 7

= [2 + 1 * (1 + 72 % 5)] % 7

= 5 % 7 = 5, който е празен, затова вмъкваме 72 в място 5.

0	1	2	3	4	5	6
	43	692			72	27

Time Complexity:

Insertion: O(n)

Search: O(n)

Deletion: O(n)
