Алгоритмы
и структуры данных

Массивы

Массив — структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов.

Свойства массива

1. Элементы массива принадлежат одному типу данных (все int/все float/..)
2. Хранится в памяти одним куском
3. Размер массива задается заранее, на этапе создания (?)

Массивы в python

```
:
# array
import array
a = array.array('f', [1,2,3])
```

```
:
# numpy array
import numpy as np
a = np.array([1,2,3])
```

```
:
# list
a = [1,2,3]
```

Оценка времени работы программы и сложности
Фактическая оценка времени работы

Запускаем алгоритм (можно несколько раз) и оцениваем время исполнения

```python
import time

t_start = time.time()
a = 0
for i in range(100000000):
    a += i
t_end = time.time()
print("Execution time, seconds: ", t_end - t_start)

```

Теоретическая оценка сложности алгоритмы
Точная оценка

Нужно оценить сложность вычисления функции:

f(x) = x^2 + 2x = x * x + 2 * x – 2 умножения, 1 сложение

f(x) = x^2 + 2x = x * (x + 2) – 1 умножение, 1 сложение

Точные оценки:

1. Не всегда возможно получить
2. Если возможно, то не всегда просто получить
3. Не всегда нужно получать

Асимптотические оценки. O-нотация

Асимптотическая оценка сложности используется для теоретической оценки сложности алгоритма при больших объемах входных данных.

Для таких оценок обычно используют O-нотацию. Если заранее не оговорено что-то другое, то сложность алгоритма оценивают ***сверху, асимптотически*** с ***точностью до константного***

Пусть какому-то алгоритму требуется в 99% случаев 4n^3 + 10n операций и в 1% 4n^2 операций. Тогда говорят что сложность алгоритма O(n^3). Произносится как O-большое от n в кубе

https://habr.com/ru/post/188010/ шпаргалка по сложности алгоритмов

Мотивация

n	log n	n*log n	n^2
1	0	0	1
16	4	64	256
256	8	2,048	65,536
4,096	12	49,152	16,777,216
65,536	16	1,048,565	4,294,967,296
1,048,576	20	20,969,520	1,099,301,922,576
16,775,616	24	402,614,784	281,421,292,179,456

Поиск числа в упорядоченном массиве. Наивный метод

Дан упорядоченный массив a_1…a_n.
Дано число (ключ). Найти номер места на котором находится это число

Пример.
Число 1001
Массив: 1 2 5 18 48 123 250 500 1000 1001 1245

Начинаем с первого элемента и сравниваем каждый элемент массива с искомым числом. В худшем варианте нам нужно сделать n сравнений

Сложность – O(n)

Алгоритм бинарного поиска

1. Сравниваем элемент массива, который стоит посередине с ключом
2. Если ключ меньше, то продолжаем поиск слева, если меньше, то справа. Если значение элемента равно значению ключа, алгоритм завершается
3. Если новый интервал размером хотя бы в 1 элемент, то переходим к пункту 1. Иначе алгоритм останавливается и ключ в массиве не найден

Бинарный поиск. Пример

Массив: 1 2 5 18 48 123 250 500 1000 1001 1245
Число, которое требуется найти (ключ): 1001

Массив: 1 2 5 18 48 ***123*** 250 500 1000 1001 1245
Индексы: 0 1 2 3 4 ***5*** 6 7 8 9 10

Массив: 250 500 ***1000*** 1001 1245
Индексы: 0 1 ***2*** 3 4

Массив: ***1001*** 1245
Индексы: ***0*** 1

Оценка затраченной памяти

Задача: Проверить что в последовательности длины N представлены все числа от 0 до N - 1 (включительно)

Пример:

5 4 2 3 1 0 – ок

5 4 4 2 1 3 – не ок

5 4 9 3 4 0 – не ок

Наивный алгоритм решения:

1. Для каждого числа i от 0 до N-1:
    a. Для каждого числа j от 0 до N:
        i. Если j == N:
            Число i не нашлось,
            значит последовательность не удовлетворяет условию.
            завершаем программу
        ii. Если i == a[j]:
            Завершаем цикл по j.
            Переходим к следующей итерации по i
2. Если цикл по i закончился без прерываний,
   значит последовательность удовлетворяет условию

Альтернативный алгоритм

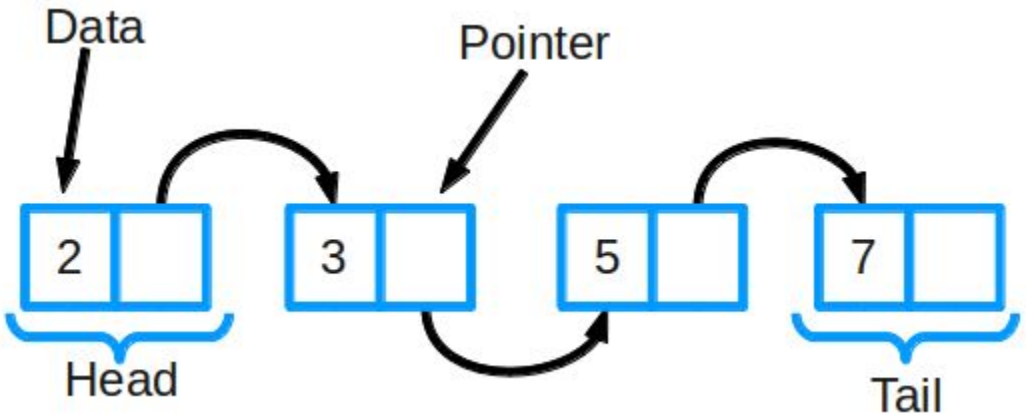
1. Заведем массив b и присвоим всем ее элементам значения -1
2. Для всех i от 0 до N-1:
   a. Если i < 0 или i > N-1:
      Завершаем программу с отрицательным результатом.
   b. j = a[i];
   c. Если b[j] != -1:
      Завершаем программу с отрицательным результатом.
   d. b[j] = i
3. Если цикл по i завершился без прерываний, значит последовательность удовлетворяет условию

Скорость выполнения операций в массиве

- Индексация (доступ к элементу) – O(1)
- Вставка/удаление элемента в конец/начало/середину – невозможно
- Поиск элемента – O(n)

Списки

Структура данных представляющая из себя последовательность элементов, каждый из которых состоит из значения и указателя, который ссылается на следующий элемент. Первый элемент – голова (head) списка, на него никто не указывает. Последний элемент – хвост (tail), у него нулевой указатель.

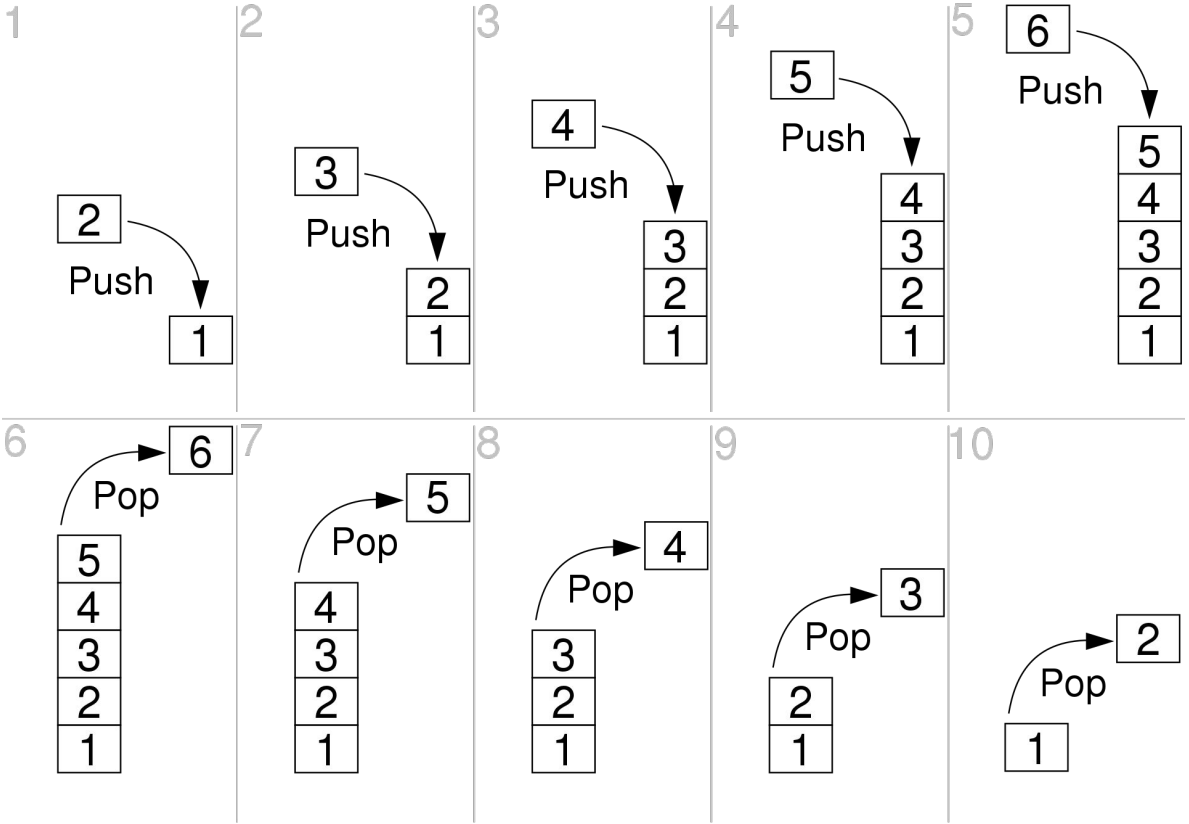


Операции со списком

1. Добавлять элемент в конец или в начало, O(1)
2. Добавлять элемент в середину, O(1) + время поиска
3. Удаление элемента из начала/конца, O(1)
4. Удаление из середины, O(1) + время на поиск
5. Поиск элемента, O(n)
6. Индексация

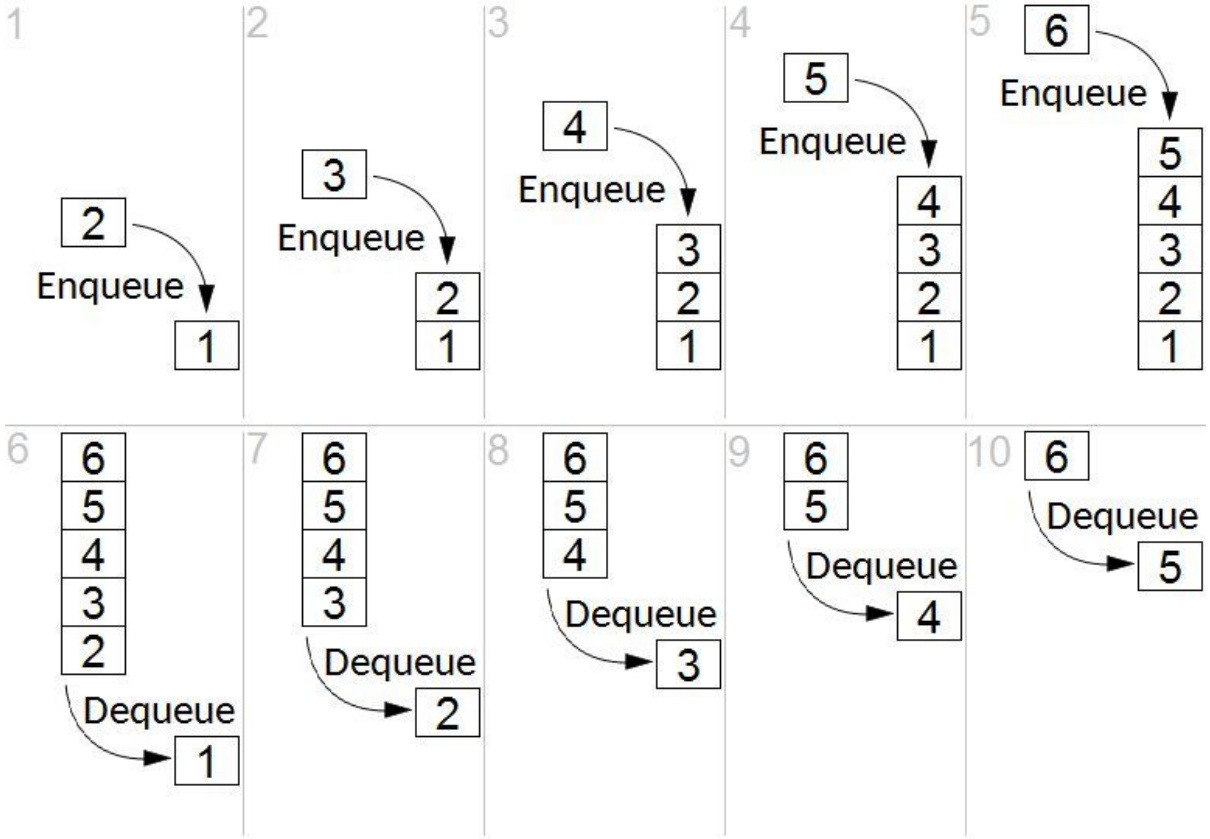
Стек

Хранилище данных, в котором определены операция Pop и Push.
Pop – кладет элемент в начало стека
Push – забирает элемент из начала стека
Говорят что эта структура данных работает по принципу LIFO:
Last In First Out



Очередь

Хранилище данных, для которого определены операции Enqueue и Dequeue
Enqueue добавляет элемент в начало очереди
Dequeue забирает элемент из конца очереди
Говорят что очередь работает по принципу FIFO:
First In First Out



Хеширование

Хеш-функция – преобразование битовой последовательности произвольной длины в последовательность фиксированного размера. При этом значение функции зависит только от входного аргумента.
Области применения:

- при построении ассоциативных массивов;
- при поиске дубликатов в последовательностях наборов данных;
- при построении уникальных идентификаторов для наборов данных;
- при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;
- при сохранении паролей *в системах защиты* в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции);
- при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

Примеры хеш функций:

- H(x) = x mod 100 (%)
- H(x) = int(len(x)) – число бит в x. Если число, которое обозначает длину исходной битовой последовательности не помещается в int, обрезаем так чтобы помещалось
- H(x) – функция которые берет первые 10 бит последовательности. Если в последовательности меньше 10 бит, функция дополняет ее нулями в начале

Примеры не хеш функций:

- f(x) = x
- f(x) = x mod 100 + random(100)

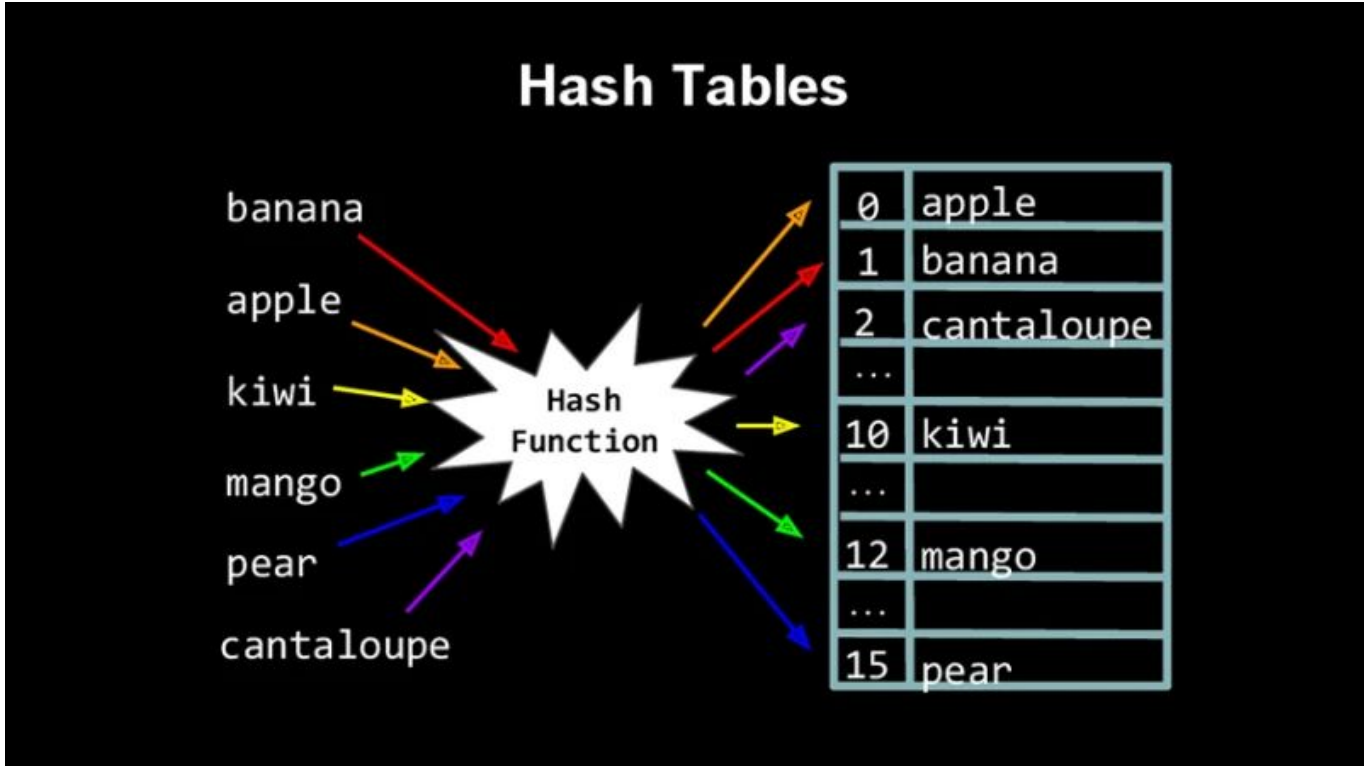
Требования к хеш функциям

Коллизией функции H(x) называется такая пара аргументов x и y, x != y, что H(x) = H(y)
Требования:

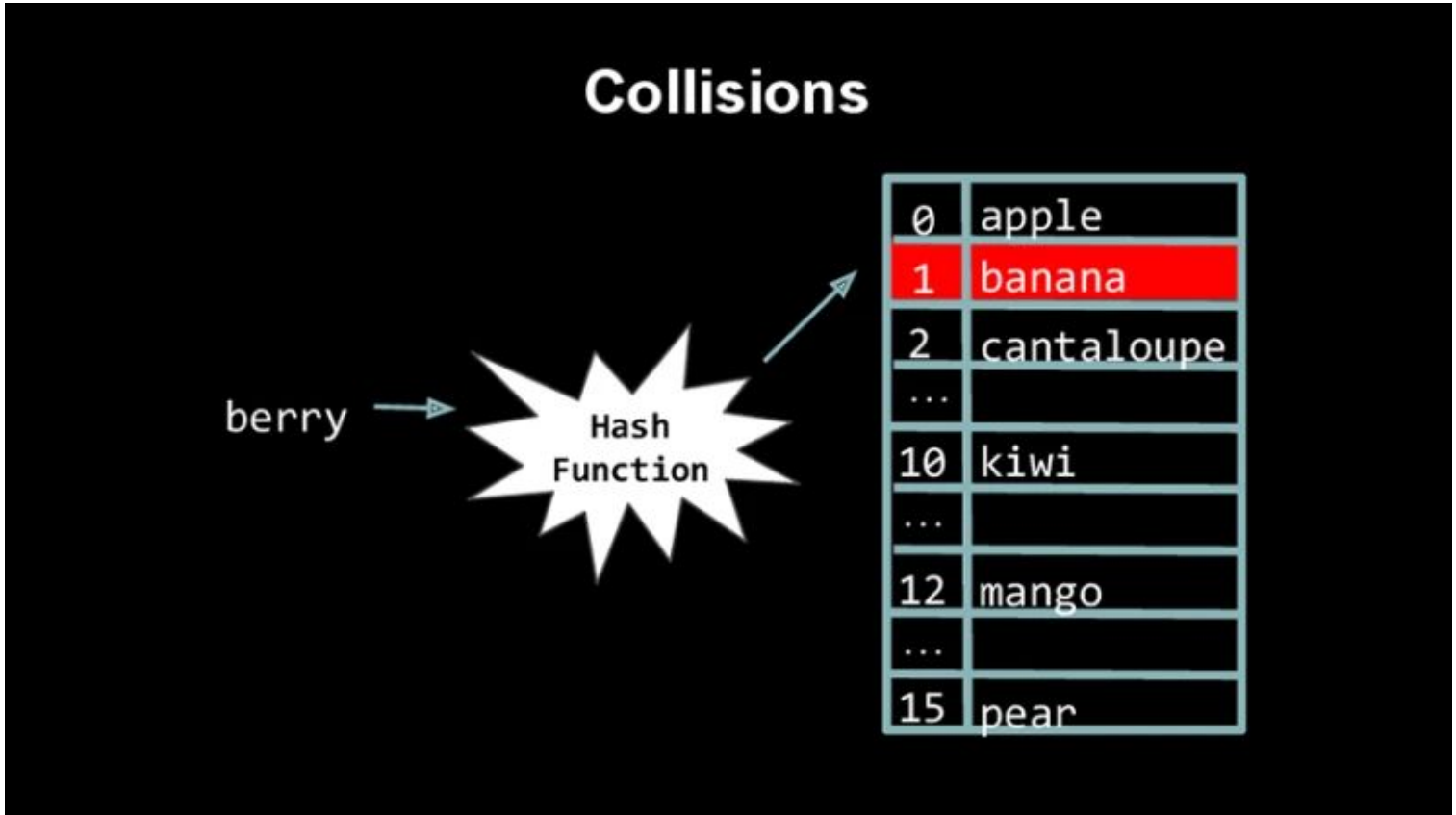
- Быстрое вычисление
- Малое число коллизий
- Специфические запросы под задачу

Хеш таблицы

Структура данных для хранения пар (ключ, значение). Так же известная как ассоциативный массив

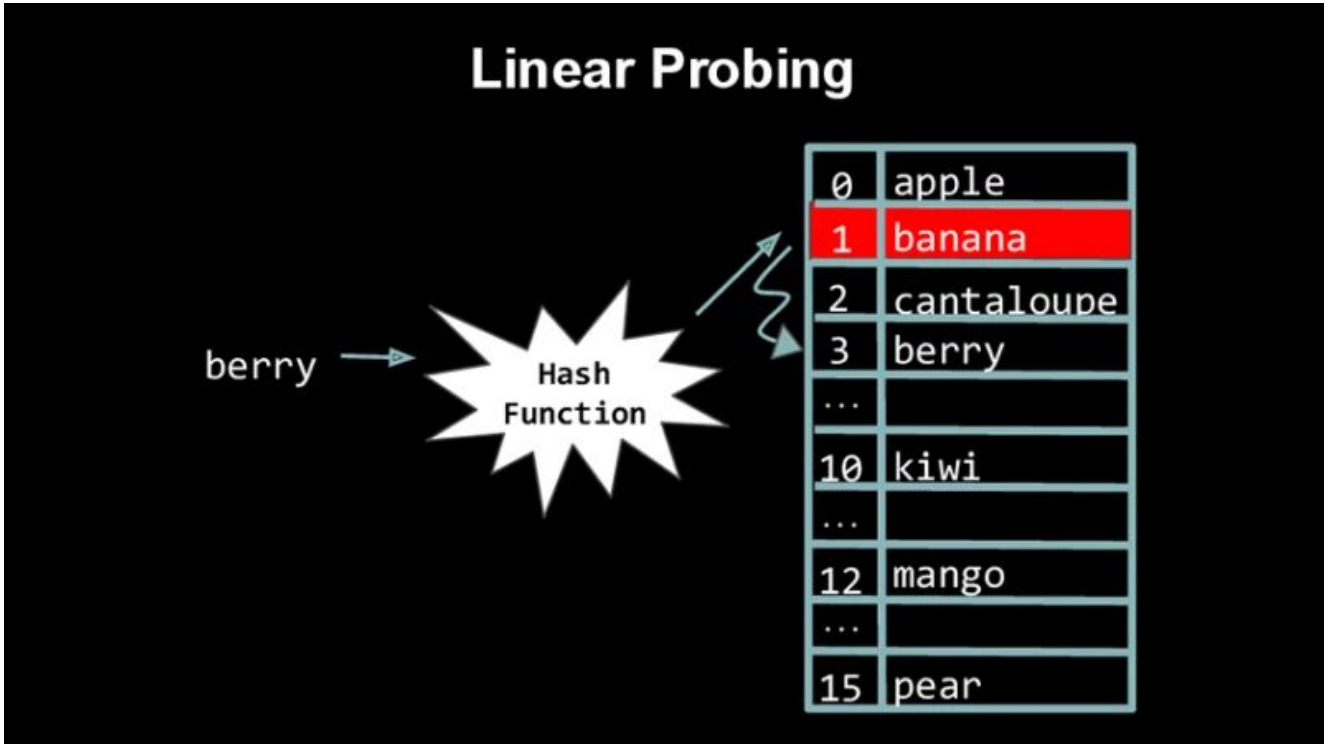


Проблема коллизий



Методы разрешения коллизий. Линейное зонирование

После того как коллизия случилась, можно идти по памяти и искать место куда положить или откуда забрать элемент. Шаг с которым это происходит не обязан быть равен единице. Можно подобрать другое число, такое чтобы поиск не закциклился



Методы разрешения коллизий. Метод цепочек

В каждом адресе лежит не один элемент, а ссылка на список, к которому при коллизии добавляются новые элементы