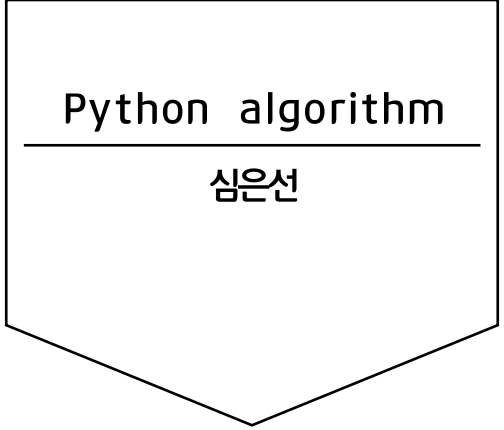

11. 해시 테이블

Contents

해시 테이블이란??

"성능 안 좋으면 해시 테이블 써~~!"

연산 대부분 성능: O(1) (분할상환분석)

HOW??!?!!
그전에 용어 정리 먼저!

Unit 00 | 해시 테이블

해시 테이블이란??

[용어 정리]

- 해시 함수(hash function)
  : 임의의 길이의 데이터를 고정된 길이로 매핑하는 함수 $h(x) = x \bmod m$
- 키(key)
  : 매핑 전 원래 데이터
- 해시 값(hash value)
  : 매핑 후 데이터의 값 (f(key)=hash_value)
- 해싱(hashing)
  : 매핑하는 과정
- 해시 테이블(hash table)=해시 맵(hash map)
  : 해시함수를 사용해 키를 해시 값으로 매핑하고,
  해시 값을 인덱스(주소) 삼아 데이터를 저장하는 자료구조.

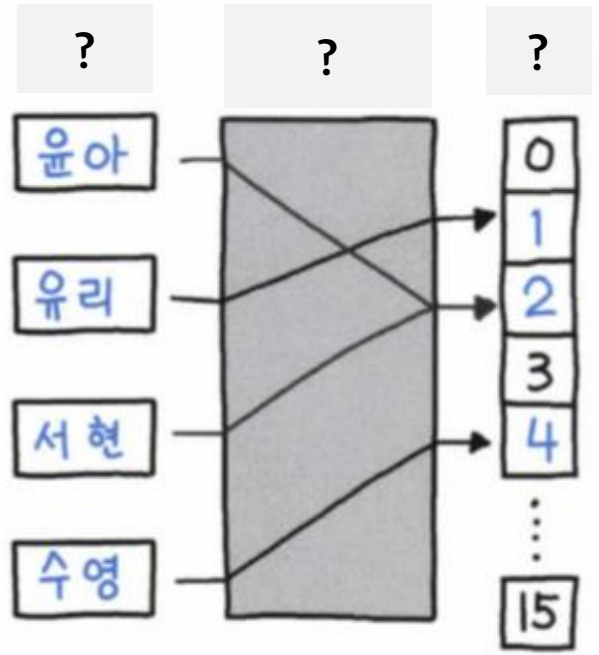

그림 11-3 해시 함수를 통해 키가 해시로 변경되는 과정[13]

Unit 00 | 해시 테이블

해시 테이블이란??

[용어 정리]
- 해시 충돌(collision)
  : 서로 다른 두개의 key(데이터)가 동일한 해시 값을 가지는 것.

해시 충돌이 별로 발생하지 않을 것 같지만, 생일 문제처럼 충돌 확률이 높음!

따라서 좋은 해시 함수는,
1) 해시 값의 충돌을 최소화 하고,
2) 연산이 쉽고 빠르며,
3) 해시 테이블 전체에 값이 균일하게 분포하고,
4) 사용할 키(원 데이터)의 모든 정보를 이용해 해싱하며,
5) 해시 테이블의 사용 효율이 높다.

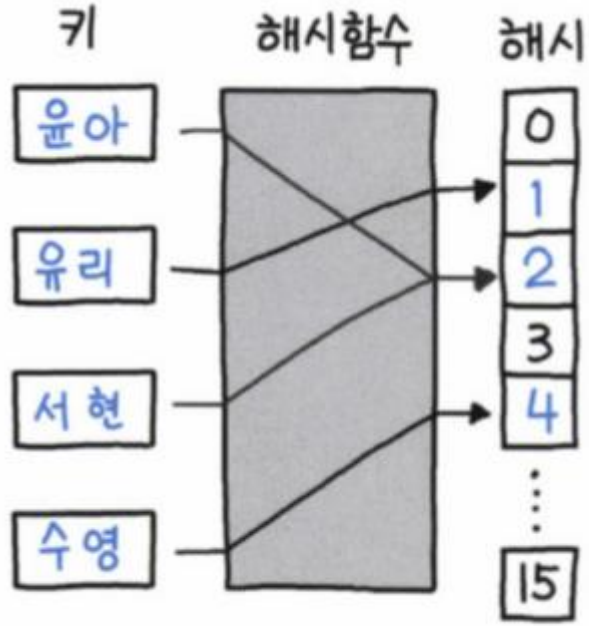

그림 11-3 해시 함수를 통해 키가 해시로 변경되는 과정[13]

해싱

Unit 00 | 해시 테이블

해시 테이블이란??

[용어 정리]
- 로드 팩터 (load factor)
  : 해시 테이블에 저장된 전체 데이터 개수(n)를
  버킷(해시 값으로 나올 수 있는 인덱스)의 개수(k)로 나는 것.

$$\text{load factor} = \frac{n}{k}$$

즉, 전체 데이터(key)가 몇 개의 공간(주소)로 분산되는 지,
해시함수가 키들을 잘 분산해 주는지 말하는 효율성 지표 (-> 검색의 효율성과 관련)

(단, load factor가 낮다고 항상 분산이 잘되는 것은 아님! 만약 30개의 key와 300개의 버킷, 한 버킷에 저장되는 경우)

Unit 00 | 해시 테이블

해시 테이블이란??

[용어 정리]

- 충돌 처리 방법 ① : 개별 체이닝
  : 충돌시 연결 리스트로 키를 연결함.
  즉, 해시 값을 배열의 인덱스로 이용하며
  같은 인덱스가 있으면 연결리스트로 연결하는 방식

- 특징:
  장: 무한대로 저장 가능
  단: 성능 (동적할당)

- 성능(탐색) : 개별 체이닝
  good: O(1)
  worst: O(N)

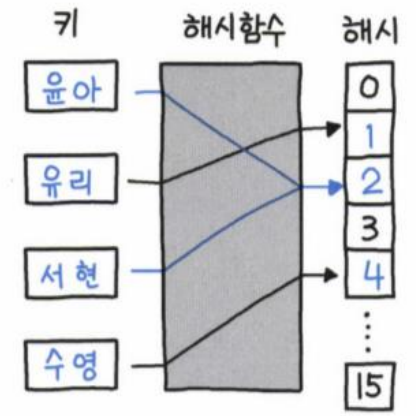

그림 11-5 해시 테이블의 충돌

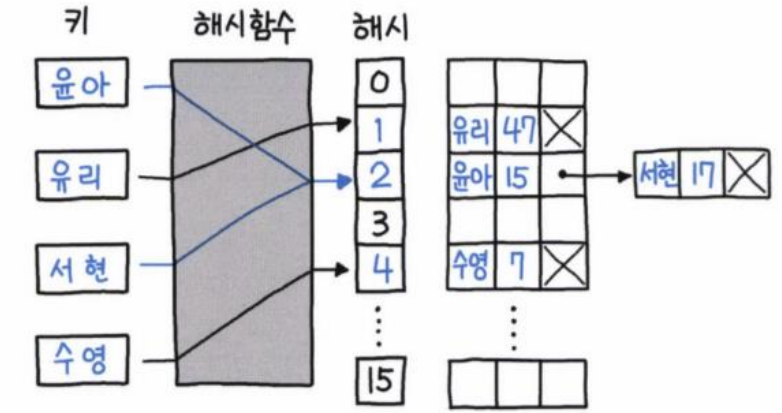

그림 11-6 개별 체이닝 방식

Unit 00 | 해시 테이블

해시 테이블이란??

[용어 정리]
- 충돌 처리 방법 ② : 오픈 어드레싱
  : 충돌시 탐사를 통해 빈 공간을 찾아 넣는 방식.
  따라서 자신의 해시 값과 일치하지 않는 공간에 저장될 수 있음

- 종류
  : 충돌시 해시 테이블 내 새로운 주소를 찾는 과정인 탐사에 따라 구분
  ex) 선형탐사 – 충돌 발생 위치부터 순차 탐사,
  가장 가까운 빈 위치에 key 삽입
  단점) 데이터의 클러스터링
  → 탐사 시간↑ , 해싱 효율↓

- 특징
  전체 버킷의 개수까지만 저장 가능
  기준 로드 팩터의 비율을 넘어가면, 새로운 버킷을 생성해 복사

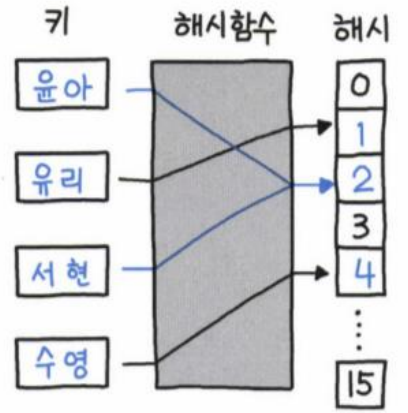

그림 11-5 해시 테이블의 충돌

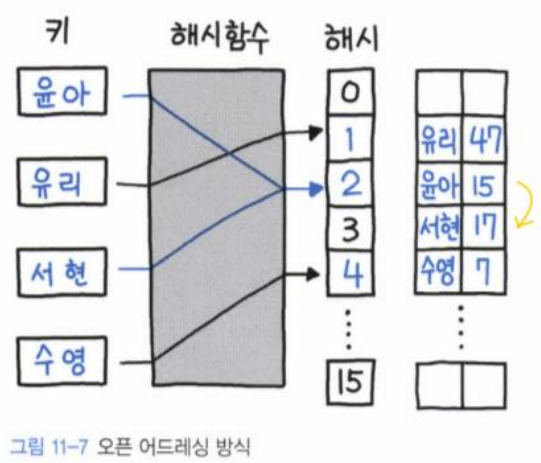

그림 11-7 오픈 어드레싱 방식

Ref: https://ratsgo.github.io/data%20structure&algorithm/2017/10/25/hash/

Unit 00 | 해시 테이블

해시 테이블이란??

해시 테이블 In 파이썬
- 딕셔너리가 해시 테이블로 구현 (오픈 어드레싱)
- 오픈 어드레싱으로 성능을 높임 (매번 메모리 할당 x)
- 대신 기준 로드팩터를 작게잡아
  로드팩터가 커질수록 낮아지는 성능 문제 해결 (0.66)

27. 해시맵 디자인은 가볍게 해보시면 좋을 것 같아요 ^_^

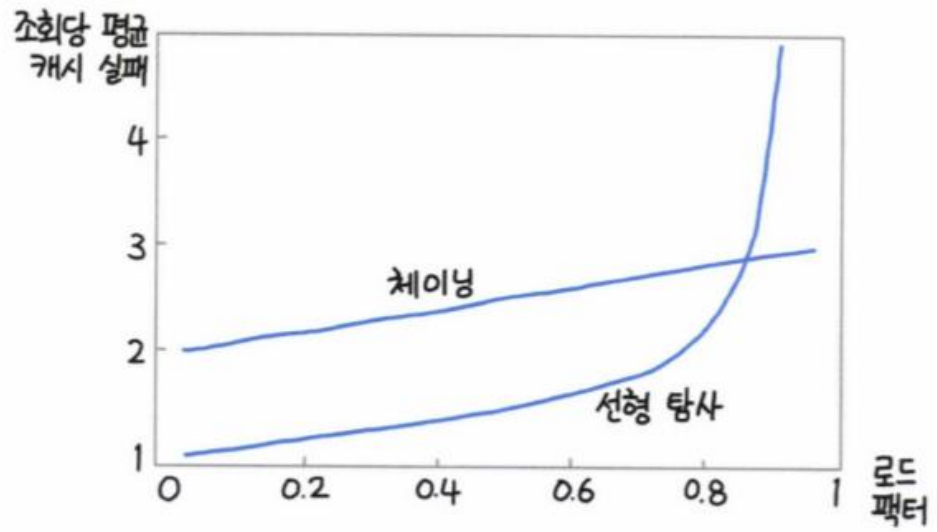

그림 11-8 체이닝과 선형 탐사 방식의 로드 팩터에 따른 성능 비교[24]

29. 보석과 돌

J(보석)에 있는 글자들이 S(돌)에 총 몇 번 나올까?

```
def numJewelsInStones(self, J: str, S: str) -> int:
    freqs = {}
    count = 0

    # 돌(S)의 빈도 수 계산
    for char in S:
        if char not in freqs:
            freqs[char] = 1
        else:
            freqs[char] += 1

    # 보석(J)의 빈도 수 합산
    for char in J:
        if char in freqs:
            count += freqs[char]

    return count
```

```
### 풀이2 ###
freqs = collections.defaultdict(int)
count = 0

#비교없이 돌(S) 빈도수 계산
for char in S:
    freqs[char] += 1

#비교없이 보석(J) 빈도수 계산
for char in J:
    count += freqs[char]

return count
```

defaultdict(value자료형)

존재하지 않는 key의 값을 조회하는 경우, 에러 대신 디폴트 값을 리턴함

29. 보석과 돌

J(보석)에 있는 글자들이 S(돌)에 총 몇 번 나올까?

```
### 풀이3 ###
freqs = collections.Counter(S) #돌(S)의 빈도수 계산
count = 0

#비교없이 보석(J)의 빈도수 계산
for char in J:
    count += freqs[char]

return count
```

Collections.Counter(list)

카운트 + defaultdict
most_common(k) 메서드

```
### 풀이4 ###
return sum(s in J for s in S)
```

리스트 컴프리헨션 + True/False의 sum

31. 상위 K 빈도 요소

주어진 리스트(nums)에서 빈도가 높은 상위 k개의 요소를 추출 (책 문제X)

```
### 풀이1 - 우선순위큐 모듈 heapq 이용 ###
freqs = collections.Counter(nums)
freqs_heap = []
#힙에 음수로 삽입
for i in freqs:
    heapq.heappush(freqs_heap, (-freqs[i], i))

topk = []
#k번 만큼 추출
for _ in range(k):
    topk.append(heapq.heappop(freqs_heap)[1])

return topk
```

<우선순위 큐로 상위 count 출력>

우선순위 큐 : heapq 모듈
[삽입]
1) 리스트로 삽입 후 heapq.heapify(리스트)
2) 매번 heapq.heappush(힙변수, (우선순위,데이터))
[삭제]
Heapq.heappop(큐이름)
-> (우선순위,데이터) 쌍 return

단, 파이썬은 기본 최소큐

31. 상위 K 빈도 요소

주어진 리스트(nums)에서 빈도가 높은 상위 k개의 요소를 추출 (책 문제X)

1) Counter의 most_common(k)
   - (값, count)의 리스트 (상위 k개) 리턴

2) Zip
   - 2개 이상의 시퀀스를 짧은 길이를 기준으로 새로운 튜플 시퀀스 생성
   - 동일한 인덱스에 있는 아이템들을 하나의 튜플로 만듦
   - 제너레이터가 리턴되므로, 출력하기 위해서는 list()로 묶어줌
   - immutable 객체

```
>>> a = [1,2,3,4,5]
>>> b = [2,3,4,5]
>>> c = [3,4,5]
>>> zip(a,b)
<zip object at 0x105b6d9b0>
```

```
>>> list(zip(a,b))
[(1, 2), (2, 3), (3, 4), (4, 5)]
>>> list(zip(a,b,c))
[(1, 2, 3), (2, 3, 4), (3, 4, 5)]
```

```
>>> a = ['a1', 'a2']
>>> b = ['b1', 'b2']
>>> c = ['c1', 'c2']
>>> d = ['d1', 'd2']
>>> list(zip(a, b, c, d))
[('a1', 'b1', 'c1', 'd1'), ('a2', 'b2', 'c2', 'd2')]
```

Unit 02 | 상위 K 빈도 요소

31. 상위 K 빈도 요소

주어진 리스트(nums)에서 빈도가 높은 상위 k개의 요소를 추출 (책 문제X)

3) *

- 튜플, 리스트를 언패킹 (하나의 객체가 아니라 여러 개로)

```
>>> fruits = ['lemon', 'pear', 'watermelon', 'tomato']
>>> fruits
['lemon', 'pear', 'watermelon', 'tomato']
```

```
>>> print(*fruits)
lemon pear watermelon tomato
```

- 함수의 파라미터로 들어가면 반대로 패킹

```
>>> def f(*params):
...     print(params)
```

```
>>> f('a', 'b', 'c')
('a', 'b', 'c')
```

31. 상위 K 빈도 요소

주어진 리스트(nums)에서 빈도가 높은 상위 k개의 요소를 추출 (책 문제X)

3) *
- 여러 개의 값을 한 변수에서 취함

```
>>> a, *b = [1,2,3,4]
>>> a
1
>>> b
[2, 3, 4]
```

```
>>> *a, b = [1,2,3,4]
>>> a
[1, 2, 3]
>>> b
4
```

cf) **
- 딕셔너리의 키/값 페어를 언패킹

```
>>> date_info = {'year': '2020', 'month': '01', 'day': '7'}
```

```
>>> new_info = {**date_info, 'day': "14"}
>>> new_info
{'year': '2020', 'month': '01', 'day': '14'}
```

31. 상위 K 빈도 요소

주어진 리스트(nums)에서 빈도가 높은 상위 k개의 요소를 추출 (책 문제X)

```
### 풀이2 ###    ②
return list(zip(*collections.Counter(nums).most_common(k)))[0]
       ③        ①                                          ④
```

① most_common(k) 결과 (값, count) 쌍을 언패킹
② 언패킹한 쌍을 각 인덱스 별로 하나로 묶음 -> (값1, 값2, …), (cnt1, cnt2, …)
③ 제너레이터 대신 바로 결과를 보기 위해 리스트화
④ 값들의 리스트만 추출

들어주셔서 감사합니다.