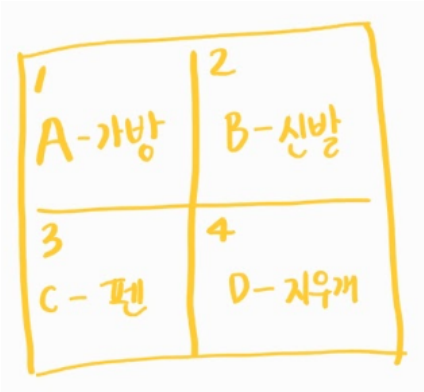
6. Hash

> linear & binary search는 index 검색이었다.
>
> 여기서 추가로 데이터의 검색 뿐만 아니라, 추가, 삭제 등에도 용이한 알고리즘이 해시법이다.

ex) git commit -m "blahblah"
하고 나서 git log 에 나오는 노란색 글씨들
이게 다 hash다.

- hash table
  ㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁㅁ
  이 칸 안에 data가 들어가 있는 상태.
- hashing 종류
  - chaining
  - open hashing
- hashing 방법
  - 앞에선 value를 주고 key를 찾아왔다.
  - 여기선 key를 주고 value를 찾아올 거다.

    ex) git switch 48c7... 여기서 [48c7] 이 값이 key가 되는 거다. 이 key가 있어야 사물함을 열 수 있다.
  - 가방 갖다줘 : 못 갖다줘, key가 없어서
    key(1) 주면서 물건 꺼내줘 : 갖고 올 수 있어.

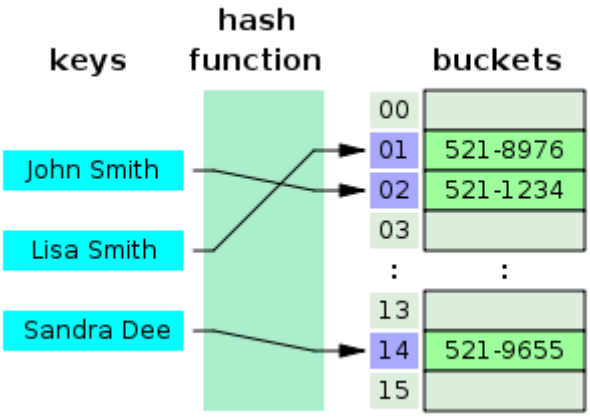


ex) value : 인증서, key : 열쇠
내 인증서를 갖고 오고 싶은데 개인정보 주면서 갖고 오라고 하는 게 아니라 key 주면서 갖고 오게 하는 것.

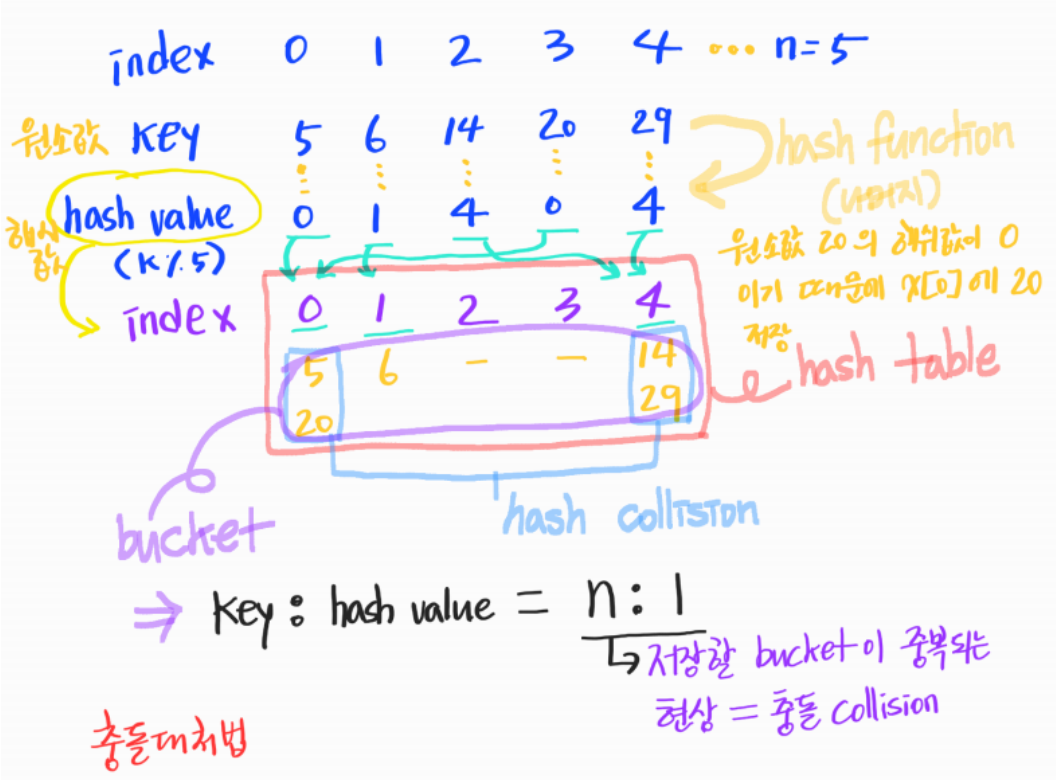
cf. key 값은 겹치지 않음.
git log처럼, 아파트 열쇠 처럼.
단, hash의 key는 다 int형 해쉬값이다.

- hash의 특징
  - 해싱된 키(Hash Key)를 가지고 배열의 인덱스로 사용하기 때문에 **삽입, 삭제, 검색이 매우 빠르다**.
  - 해시 함수(Hash Function)를 사용하는데 **추가적인 연산이 필요**하다.
  - 해시 테이블(Hash Table)의 크기가 유한적이고 해시 함수(Hash Function)의 특성상 **해시 충돌(Hash Collision)이 발생**할 수 밖에 없다.
  - 충돌이 없거나 적으면 O(1)의 상수 시간에 가까워지고, 충돌이 발생하면 할수록 성능은 점점 O(n)에 가까워진다. (극단적 경우)

- int 값들이 들어있는 key의 해시 함수는 `나누기` 함수가 대표적이다.

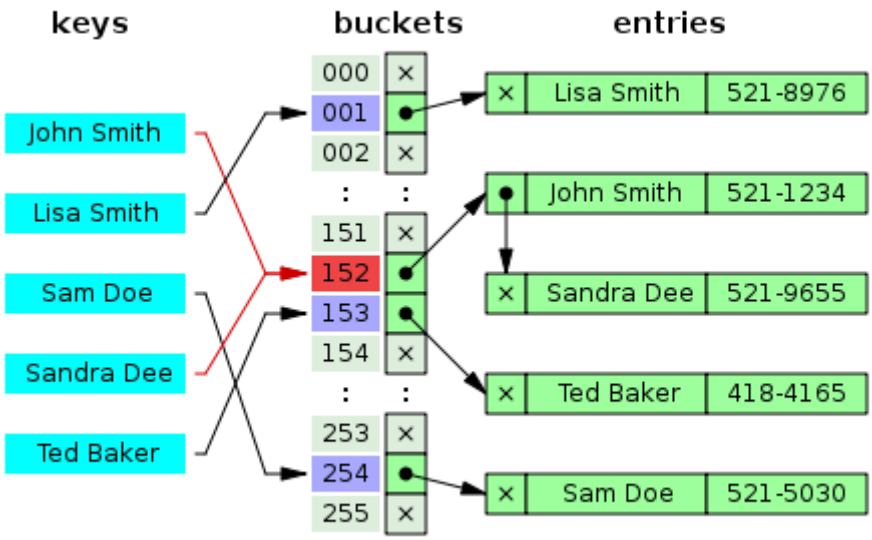


구조로 데이터를 저장하면 Key값으로 데이터를 찾을 때 해시 함수를 1번만 수행하면 되므로 매우 빠르게 데이터를 저장/삭제/조회할 수 있다. 해시테이블의 평균 시간복잡도는 O(1)이다.



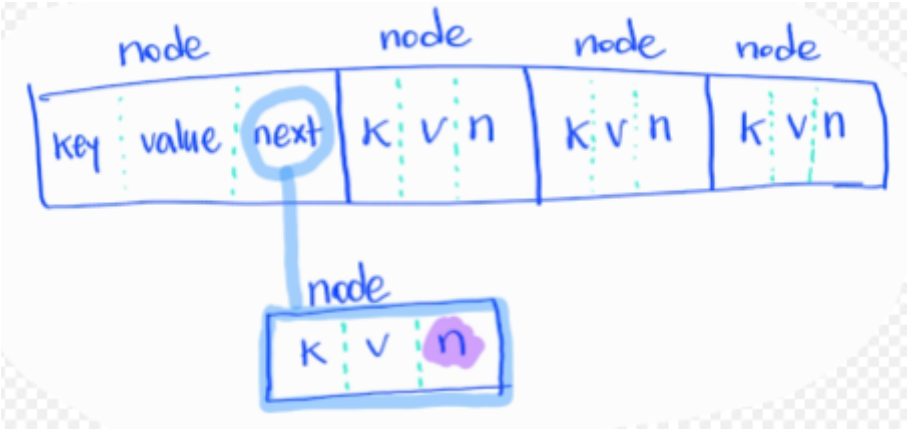
- n=5
  key 데이터 형태가 int일 때, 대부분 배열의 크기(capacity)로 나눈 나머지를 hash value=new index로 삼는다.
- capacity(용량) = table 크기 = 원소 갯수 = 배열 크기
- 충돌대처법
  1. 체인법
  2. 오픈주소법

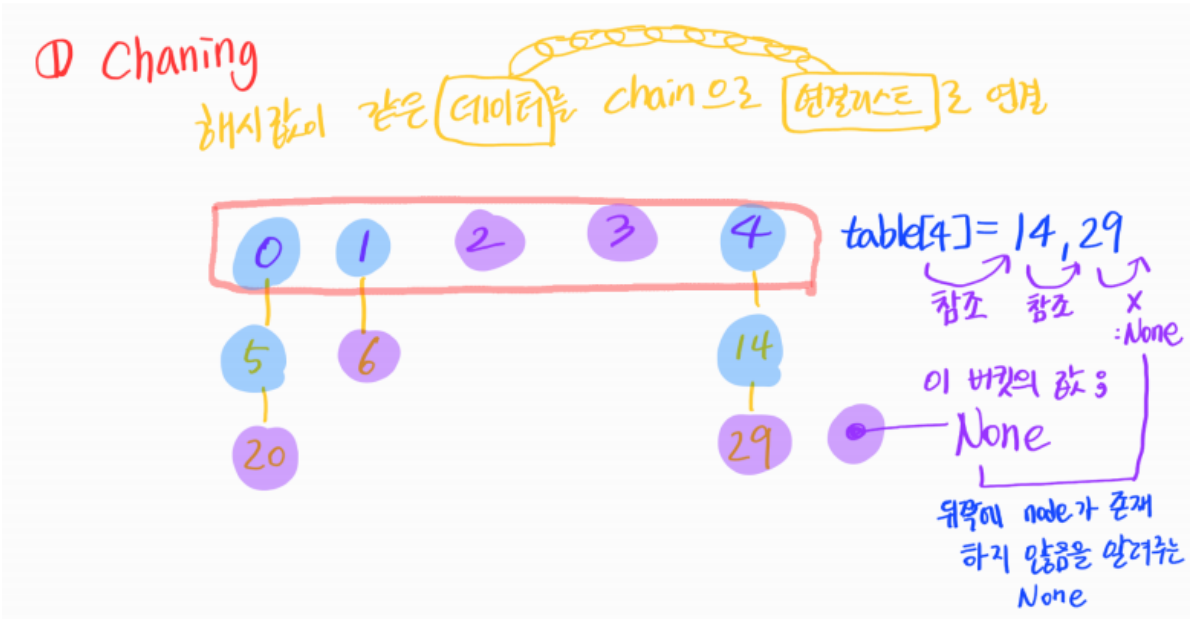
Chaning



[key-value-next]

- key와 value의 타입은 any
  : 숫자, 문자열 다 가능
- next : 뒤쪽 node를 참조 / node가 중요





- linked list 형태가 된다.

```
def hash_value(self, key: Any) -> int:
#     """해시값을 구함"""
     if isinstance(key, int):
         return key % self.capacity
     return(int(hashlib.sha256(str(key).encode()).hexdigest(), 16) %
self.capacity)
```

> int(hashlib.sha256(str(key).encode()).hexdigest(), 16

hashlib 통해서 string, int, float 형태가 될 수 있는 원소를 int형인 key 값으로 받겠다. 이 정도로만 이해하고 넘어가도 될 듯하다.

```
# value를 넣을 필요가 없다. 즉,해시에서 데이터를 찾을 때에는 'key'만을 이용해서 찾는다.
    def search(self, key: Any) -> Any:
        """키가 key인 원소를 검색하여 값을 반환"""
        hash = self.hash_value_int(key)  # 검색하는 키의 해시값을 구한다.
        # hash = self.hash_value(key\
        p = self.table[hash]         # 노드를 노드

        while p is not None: # 다음 노드(값)이 있으면 다음 계속 탐색
            if p.key == key:     # p의 key와
                return p.value  # 검색 성공
            p = p.next           # 뒤쪽 노드를 주목

        return None              # 검색 실패
```

> def search(self, key: Any) -> Any:

찾을 때는 key로.
return은 value로. 이 때, value는 어떤 형태든 될 수 있으므로 any.

//////////////코드이해필요//////////////

Open Addressing

- open addressing는
  hash collision이 발생 했을 때 rehashing(재해시)를 통해 빈 버킷을 찾아 데이터를 추가하는 방법이다.
- chaining처럼 지정한 메모리 외에 추가적인 저장공간이 필요가 없다.
- 삽입,삭제시 오버헤드가 적다.
  - **오버헤드**(overhead)는 어떤 처리를 하기 위해 들어가는 간접적인 처리 시간 · 메모리 등을 말한다.
- 저장할 데이터가 적을 때 더 유리하다.

  → 저장할 데이터가 많을 경우에는 chained hash가 더 유리할 수 있다.

출처 및 참고]

1. https://github.com/GyungBin-Park

2. https://jwoop.tistory.com/9
3. https://mangkyu.tistory.com/102
4. https://noahlogs.tistory.com/27