Hash | 해시

> 고급 탐색 구조

📚 해싱

킷값에 산술적인 연산을 적용해 레코드가 저장되어야 할 위치를 직접 계산하는 것임

▼ 기존의 탐색 방법들과의 차이점

기존: 탐색키와 각 레코드의 킷값을 비교해 원하는 항목을 찾음

!! 해시 함수를 사용해 키를 해시값으로 매핑하고, 이 해시값을 인덱스 혹은 주소로 삼아 데이터를 key와 함께 저장하는 자료구조

key-value로 이루어진 자료구조

📚 해시 함수 | Hash Function

해싱에서 킷값으로부터 레코드가 저장될 위치를 계산하는 함수

!! key를 고정된 길이의 hash로 변경해주는 역할(key로 hash를 만들어내는 함수)

* **해싱(hashing)**: key를 고정된 길이의 hash로 변경하는 과정
  - 해시 함수의 input: key
  - 해시 함수의 output: hash ⇒ hash는 저장위치가 됨

* **해시 주소(Hash Address)**: 해시 함수로 연산한 결과 `key(킷값)가 입력되면 해시 함수로 연산한 결과 h(key)가 해시 주소가 됨`

📚 해시 테이블 | Hash Table

해시 함수에 의해 계산된 위치에 레코드를 저장한 테이블

* **버킷(bucket)**: 해시 테이블을 구성하는 요소(해시 테이블은 M개의 버킷으로 이루어지는 테이블임)

* **슬롯(slot)**: 하나의 버킷은 여러 개의 슬롯을 가짐
  - 레코드: 하나의 슬롯에 하나의 레코드가 저장됨

```
hash: 색인 또는 인덱스
hash function: key를 hash로 만들어주는 함수
hash table: hash를 주소로 삼아 데이터를 저장하는 자료구조
```

🏷️ 해시의 이해

▼ ex1.

탐색키: 1~1000 사이의 정수 → 가장 빠르게 탐색할 수 있는 방법: 1000개의 항목을 가지는 배열(=우편함)을 생성

* 자료의 삽입 & 탐색: 탐색키 ⇒ 인덱스
* 해시 함수: h(key) = key
* 시간복잡도: O(1)

▼ ex2.

만약 탐색키가 문자열이거나 실수, 혹은 굉장히 큰 정수라면? (아파트 세대수가 너무 많아 세대별 우편함을 만들 공간이 부족한 상황)

메모리가 부족 → 보다 작은 배열을 사용해야 함 ⇒ 이는 탐색키를 더 이상 직접 배열의 인덱스로 사용할 수 없음

따라서, 해시 함수가 필요(탐색키를 작은 정수로 대응시키기 위함)

해시 함수: 탐색키를 입력받아 해시 주소(Hash Adderess)를 계산

📚 오버플로(overflow)

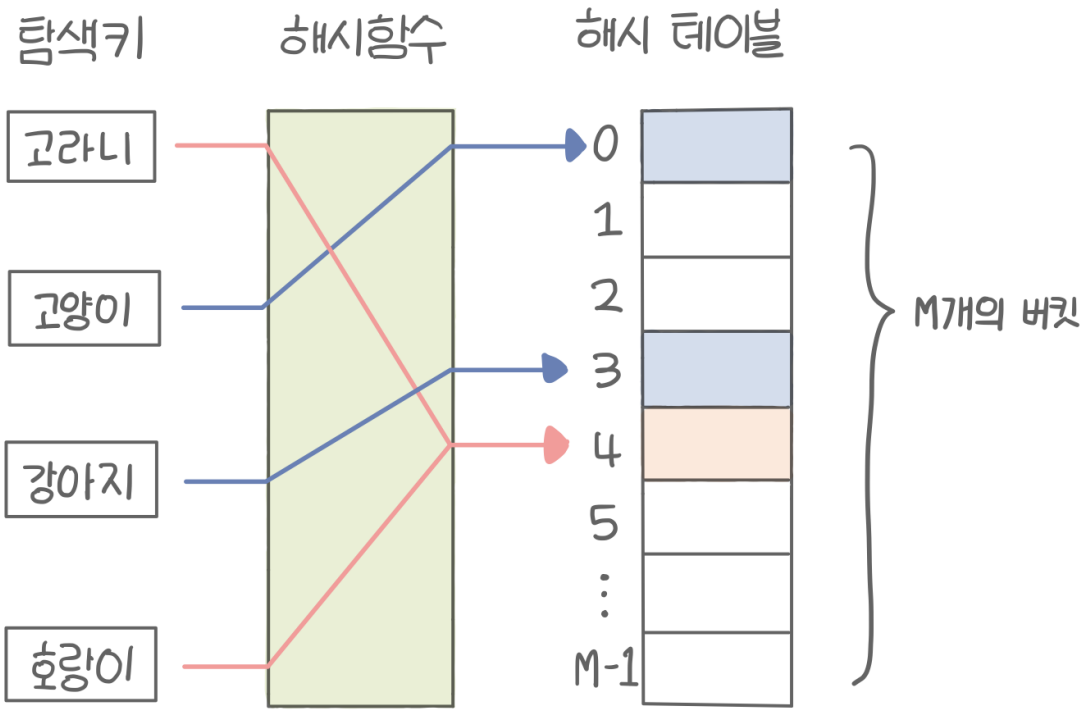
```
킷값 key가 입력되면 해시 함수로 연산한 결과 h(key)가 해시 주소가 되고, 이를 인덱스로 사용해 항목에 접근
```

▼ 💬 만약 버킷이 충분하지 않으면?

서로 다른 키가 해시함수에 의해 같은 주소로 계산되는 상황이 발생

* **해시의 충돌(collision)**: 버킷이 충분하지 않아 서로 다른 키가 해시함수에 의해 같은 주소로 계산되는 상황

* **동의어(synonym)**: 충돌을 일으키는 키들



▼ 해시 충돌 ex.

각 탐색키에 대한 해시 함수의 계산 결과: `h(고라니)⇒ 4` | `h(고양이)⇒ 0` | `h(강아지)⇒3` | `h(호랑이)⇒ 4`

* 해시 충돌: 고라니와 호랑이가 같은 주소로 계산되어 충돌 발생
* 동의어: 고라니, 호랑이

💬 해시 충돌이 발생하면?

만약 버킷에 여러 개의 슬롯이 존재하면 서로 다른 슬롯에 저장하면 됨

* **오버플로(overflow)**: 충돌이 슬롯 수보다 더 많이 발생한 경우
  - 해당 버킷에 더 이상 항목을 저장할 수 없음

📚 이상적인 해싱

충돌이 절대 일어나지 않는 경우

해시 테이블의 크기를 충분히 크게 하면 가능하지만 메모리가 지나치게 많이 필요함

따라서, 실제의 해싱에서는 테이블의 크기를 적절히 줄이고, 해시 함수를 이용해 주소를 계산

* 실제의 해싱: 시간 복잡도는 이상적인 경우 O(1)보다는 떨어짐
    - 실제의 해싱에서는 충돌과 오버플로가 빈번하게 발생하기 때문