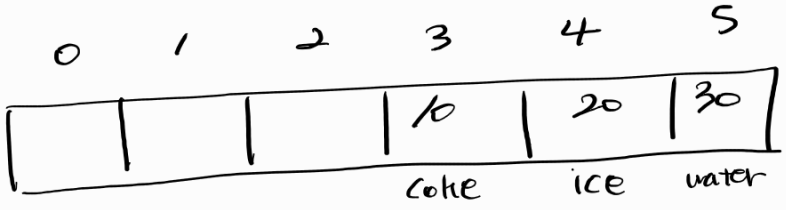
< 해시 테이블 > = hash map

- key 값을 입력 받아 해시 함수로 부터 반환된 hash code를 배열의 index로 환산해서 데이터에 접근하는 방식
- key 와 value를 저장, 관계를 모형화 할 수 있다

ex) key의 value를 저장하려한다.



→ hash table에서 hash function return value를 index로 잡고, 각 항목의(key) value를 추가한다.

만약 water의 가격(value)를 알고 싶다면, water를 hash function에 넣으면 index : 5를 반환하고 해당 index의 값이 가격(value) 이다.
따라서 time complexity 는 O(1) 임을 알 수 있다.

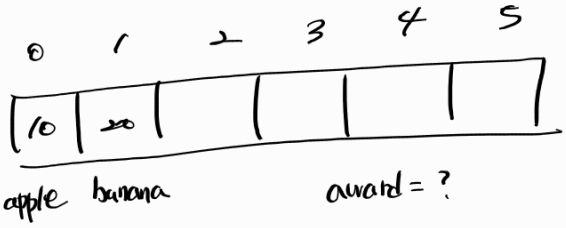
- 정보보호개론에서 배웠듯이, hash function은 서로 다른 데이터일때 서로 다른 hash value를 반환한다.
하지만, 딱딱 들어맞는 hash function을 만드는 것은 매우 힘들다.

→ "Collision" (충돌)

↓

hash function이 무한한 가짓수의 입력을 받아 유한한 가짓수의 출력값을 생성하는 경우

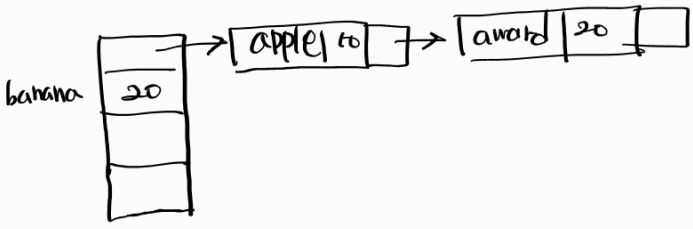
ex)



value를 반환하는 hash function을 만들었다치자

예를들어, 첫글자에 따라
a로 시작하면 0, b로 시작하면 1, c로 시작하면 2를 반환..
근데 award를 넣으려하는데, 이미 (index 0에 apple이 공간을 차지하고 있다.
→ collision 발생

* 충돌을 피하려면? → 가장 간단한 방법: 버킷에 여러 개의 키를 리스트로 만든다



: 이제 apple과 award가 같은 버킷에 리스트로 할당되었다.

⊕ 극단적인 예시) 모두 a로 시작하는 단어만 있다고 한다면...
→ hash table 한 버킷만 빼고 모두 비어있다. 한 버킷에는 거대한 linked list 존재.
결국 hash table 느려지게 된다 : O(n)
각 버킷에 고르게 분포할 수 있도록 만들어야 함.

⊕ 사용율 (load factor) = hash table 항목 수/hash table 공간수
(• 사용율이 1보다 크다는 것은 배열 공간의 수 보다 항목의 수가 더 많다는것
• 보통 사용율이 0.7보다 커지면 resizing 한다.

• collision 해결책 3가지

① separate chaining : 동일한 버킷의 데이터에 대해 추가 메모리 사용하여 연결 데이터의 주소를 지정. 간단함.
데이터 수 많아지면 효율성 떨어진다

② open addressing : 비어 있는 hash table 공간을 활용
ⓘ linear probing (선형탐색)
ⓘⓘ quadratic probing (제곱탐색)
ⓘⓘⓘ double hasing (이중탐색)

③ Resizing

* brute-force (완전탐색)으로는 시간초과에 빠지게 되는 문제에서는

hash를 적용시켜야 한다.

문제) N은 100,000의 값만큼 문자열이 입력된다
처음 입력되는 문자열은 "ok", 들어온적 있던 문자열은
"문자열 + index"로 출력한다.

ex)

input	output
5 // 5번 입력	ok
abc	ok
ab	abc 1
abc	abc 2
abC	ok
a	

똑같은 문자열이 들어왔는지 체크해보고, 들어온 문자열은 index 번호를 붙여해서 출력해주면 된다.
하지만, 현재 N값은 최대 10만인데, brute-force로 접근하면 총 100억번의 연산이 필요해서 시간초과에 빠질거다.
따라서, hash table 이용하자.

↓

hash table은 search를 최대한 줄여주기 위해 input에 대한 key값을 얻어내서 관리하는 방식.
N값 최대는 10만이므로, 2차원 배열로 1000 / 100으로 나누어 관리하면 더 효율적일 것이다.
collision을 고려해서, 두번째 배열 값에 4를 곱해서 선언한다(?)