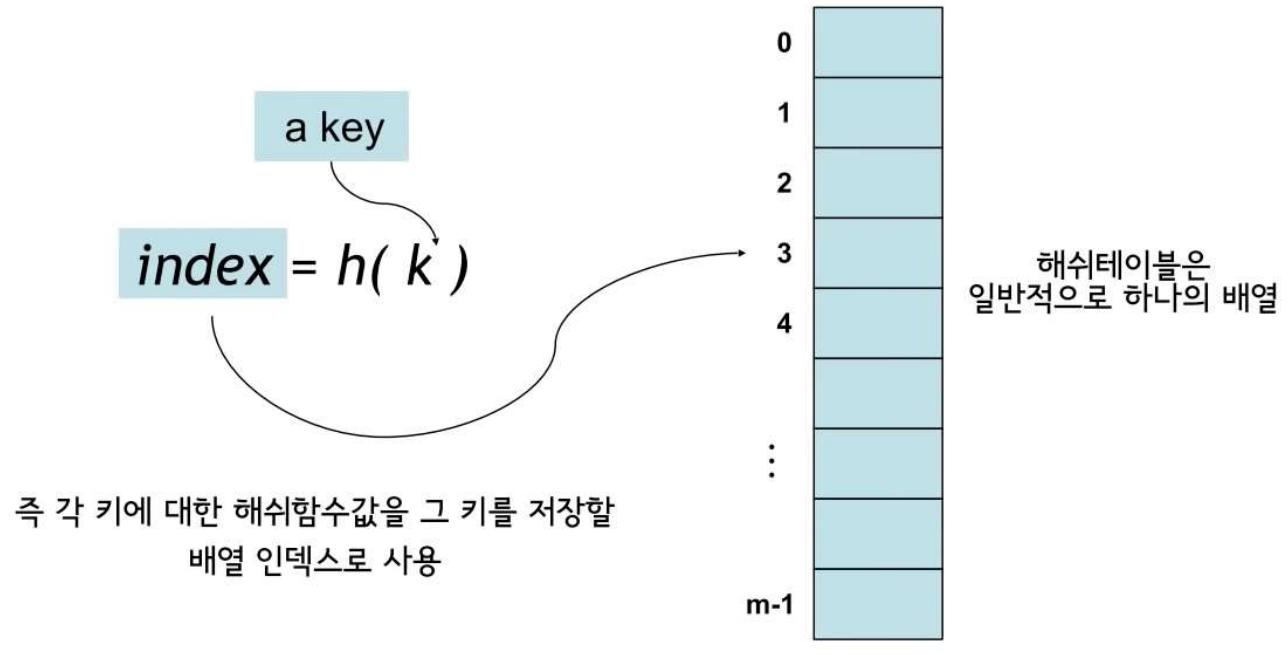
알고리즘9주차(해쉬)

Hash Table

해쉬테이블은 일반적으로 하나의 배열이라고 생각하면 된다.

해시 함수(hash function) h를 사용하여 키 k를 T[h(k)]에 저장

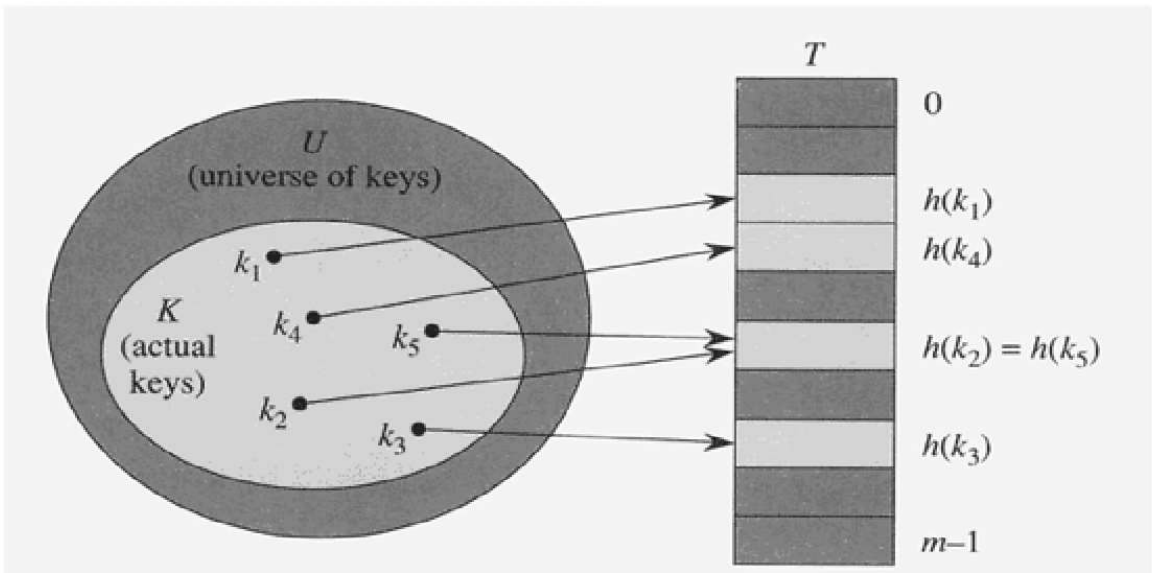
- h : U -> {0, 1, 2, … , m-1}
    - 여기서 m은 테이블의 크기, U는 모든 가능한 키들의 집합
- 키 k가 h(k)로 해싱되었다고 말한다.
- 즉, h() 해시 함수는 각 키에 대한 해시함수값을 그 키를 저장할 배열 인덱스로 사용한다.



해시 함수의 예

- 모든 키들을 자연수라고 가정(어떤 데이터든지 자연수로 해석하는 것이 가능하다)
- 예: 문자열
  - ASCII 코드 : C=67, L=76, R=82, S=83
  - 128진수로 표현하는 문자열 CLRS는(임의의 문자열을 자연수로 해석하기 위해)
  - (67*128^3) + (76*128^2) + (82*128^1) + (83*128^0) = 141,764,947
- 해시 함수의 간단한 예
  - h(k) = k % m
    - 즉, key를 하나의 자연수로 해석한 후 테이블의 크기 m으로 나눈 나머지를 데이터가 저장될 주소로 사용 한다.
    - 항상 0 ~ m-1 사이의 정수가 됨

충돌(collision)



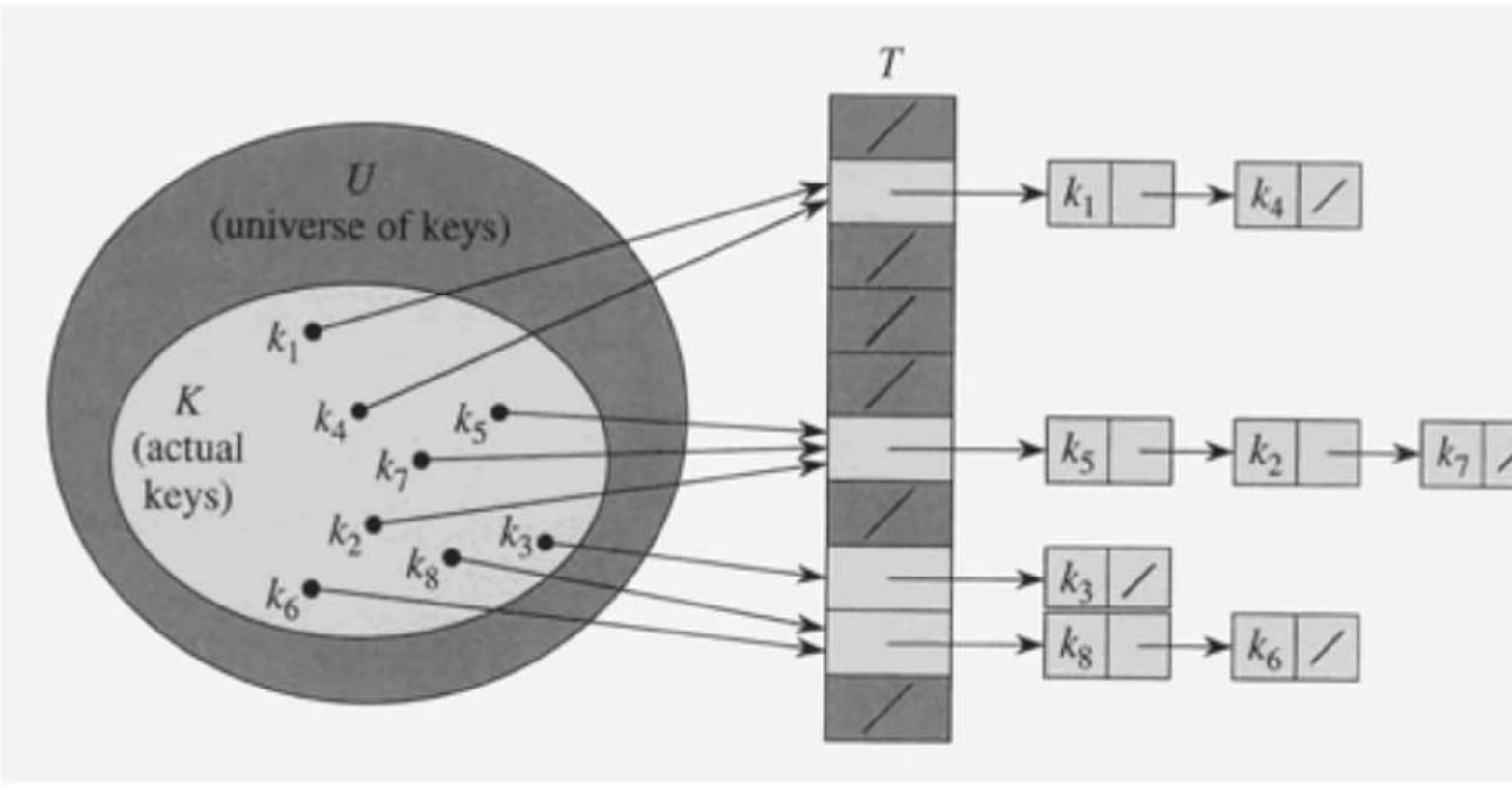
k_2와 k_5가 충돌

- 두 개 이상의 키가 동일한 위치로 해싱되는 경우
- 즉, 서로 다른 두 키 k1과 k2에 대해서 h(k1) = h(k2)인 상황
- **일반적으로|U| >> m 이므로 항상 발생 가능 (즉, 일반적으로 해시함수는 단사함수가 아님)**
- 만약 |K| > m 이라면 당연히 발생, 여기서 K는 실제로 저장된 키들의 집합
  - 임의의 정수를 저장하기 위한 배열의 크기를 무한정 키울수는 없다.

대표적인 두 가지 충돌 해결방법(아래)

Chaining에 의한 충돌 해결

동일한 장소로 해싱된 모든 키들을 하나의 연결리스트(Linked List)로 저장



- 키의 삽입(Insertion)
  - 키 k를 리스트 T[h(k)]의 맨 앞에 삽입 : 시간복잡도는 O(1)
  - 중복된 키가 들어올 수 있고 **중복 저장이 허용되지 않는다면** 삽입시 리스트를 검색해야 한다. 따라서 이런경우 삽입의 **시간복잡도는 리스트의 길이에 비례한다.**
- 키의 검색(Search)
  - 리스트 T[h(k)]에서 **순차검색**
  - 시간복잡도는 키가 **저장된 리스트의 길이에 비례한다**.
- 키의 삭제(Deletion)
  - 리스트 T[h(k)]로 부터 키를 검색 후 삭제
  - 일단 키를 검색해서 찾은 후에는 O(1)시간에 삭제가 가능하다.
- **최악의 경우**는 모든 키가 하느의 슬롯으로 해싱되는 경우이다.
  - 길이가 n인 하나의 연결리스트가 만들어지고
  - 따라서 최악의 경우 탐색시간은 O(n) + 해시함수 계산시간이 된다.
- **평균 시간복잡도**는 키들이 여러 슬롯에 얼마나 잘 분배되느냐에 의해서 결정 된다.

SUHA (Simple Uniform Hashing Assumption)

- 각각의 키가 모든 슬롯들에 균등한 확률로(eually likely) 독립적으로(independently) 해싱된다는 가정
  - **성능분석을 위해서 주로 하는 가정**임
  - hash함수는 deterministic(결정적 함수이므로)하므로 현실에서는 **불가능**하다.
    - 특정한 키 값의 해시함수 값은 정해져 있다.
- 위의 가정이 성립한다면(키가 해시테이블에 저장될 확률이 동일 하다면) Load factor를 정의할 수 있다.
- Load factor a = n(키의 개수)/m(테이블의 사이즈)
  - n : 테이블에 저정될 키의 개수
  - m : 해시테이블의 크기, 즉 연결리스트의 개수
  - **각 슬롯에 저장된 키의 평균 개수**

- 연결리스트 T[j]의 길이를 n-j이라고 하면 E[n-j] = a
- 만약 n=O(m)이면 평균검색시간은 O(1)
    - 강의 맨 처음에 "적절한 가정"하에 평균 탐색, 삽입, 삭제시간 O(1)이라는 이야기를 했었는데, '적절한 가정'이 성능분석을 위해 주로 하는 현실에서는 불가능한 가정이므로 가설이라고 이야기 할 수 있다.

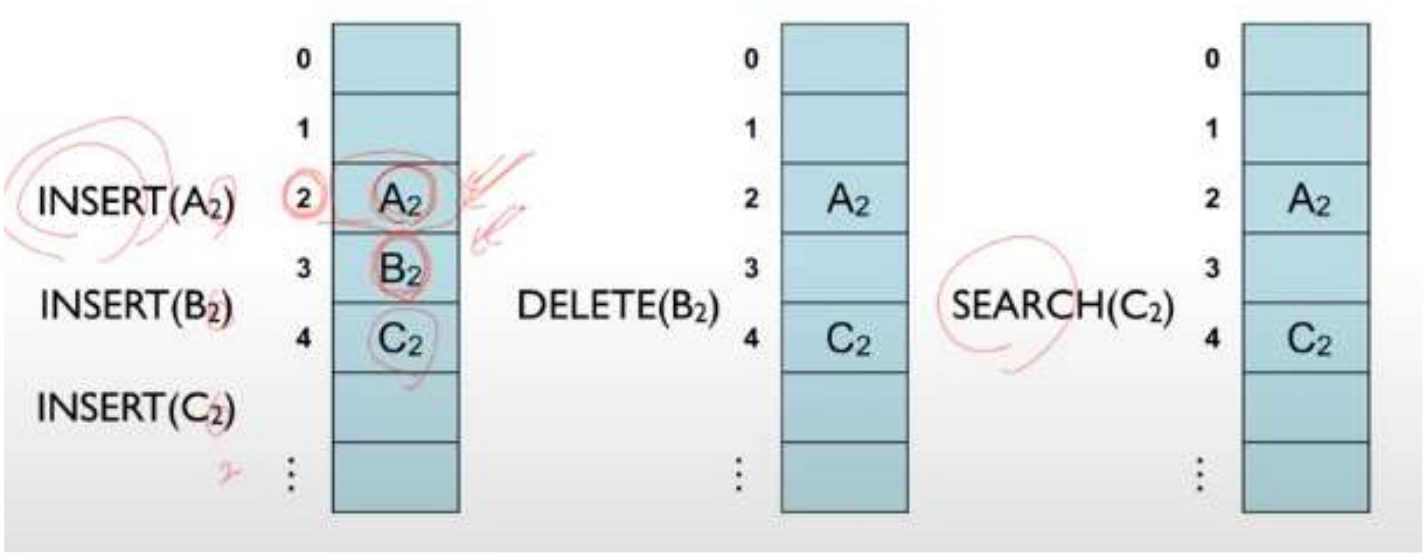
Open Addressing에 의한 충돌 해결

- 모든 키를 해시 테이블 자체에 저장
- 테이블의 각 칸(slot)에는 1개의 키만 저장
- 충돌 해결 기법
    - Linear probing
    - ▶ **Linear Probing(클릭)**
    - Quadratic probing
    - ▶ **Quadratic Probing(클릭)**
    - Double hashing
    - ▶ **Double Hashing(클릭)**

Open Addressing - 키의 삭제

- **단순히 키를 삭제할 경우 문제**가 발생한다. 가령 A2, B2, C2기 순서대로 모두 동일한 해시함수값을 가져서 linear probing으로 충돌을 해결했을 때, B2를 삭제한 후 C2를 검색하는 경우가 이에 해당한다.

- Linear Probing을 했을 경우 빈 슬롯에 도달하면 검색이 종료되기 때문에 검색에 문제가 생기게 된다.
- 이 문제를 해결하기 위해서 삭제된 슬롯에 예를 들면 DEL이라는 표시를 해둘 수도 있지만,
- Dynamic Set을 구현한 해시 테이블의 특성상 삽입, 삭제가 빈번하게 일어나므로 결국 거의 모든 슬롯이 DEL표시로 채워질 수 있다.
- 또한, 곳곳에 DEL 표시가 되어있으면 결국 배열과 같이 모든 슬롯을 검색하게 되므로 Hashing의 장점을 잃게 된다.



- **가장 적절한 해결책은 삭제될 슬롯에 있는 값과 같은 해시값을 가지는 클러스터의 마지막 슬롯을 삭제될 슬롯으로 가져와서 클러스터의 손상을 막는 방법이다.**

좋은 해시 함수란?

- 현실에서는 키들이 랜덤하지 않음
- 만약 키들의 통계적 분포에 대해 알고 있다면 이를 이용해서 해시 함수를 고안하는 것이 가능하겠지만 현실적으로 어려움

- 키들이 어떤 특정한 (가시적인) **패턴을 가지더라도 해시함수값이 불규칙적이 되도록 하는게 바람직하다**.
    - 해시함수값이 키의 특정 부분에 의해서만 결정되지 않아야 함

해시 함수

Division 기법

- h(k) = k mod m
- 예: m = 20 and k = 91 ==> h(k) = 11
- 장점: 한번의 mod연산으로 계산, 따라서 빠름
- 단점: 어떤 m값에 대해서는 해시 함수값이 키값의 특정 부분에 의해서 결정되는 경우가 있음, 가령 m = 2^p 이면 키의 하위 p비트가 해시 함수값이 됨

Multiplication 기법

- 0에서 1사이의 상수 A를 선택: 0 < A < 1
- k*A의 소수부분만을 택한다
- 소수 부분에 m을 곱한 후 소수점 아래를 버린다.
- 예 : m=8, word size = w = 5, k = 21
- A = 13/32를 선택
- kA = 21*13/32 = 273/32 = 8 + 17/32
- m (kA mod 1) = 8 * 17/32 = 17/4 = 4.xxx
- 즉, h(21) = 4
- 꼭 위의 기법을 사용하여 해시 함수를 만들어야 한다는 것은 아니다. 해시 함수를 만드는 방법들 중 하나일 뿐이다

Hashing in Java

- Java의 Object 클래스는 hashCode() 메서드를 가짐, 따라서 모든 클래스는 hashCode() 메서드를 상속받는다. 이 메서드는 하나의 32비트 정수를 반환한다. 32비트 정수라는 것은 음수일 수도 있다는 이야기이다.
- 만약 x.equals(y)이면 x.hashCode() == y.hashCode() 이다. 하지만 역은 성립하지 않는다.
- Object 클래스의 hashCode() 메서드는 객체의 메모리 주소를 반환하는것으로 알려져 있음(but it's implementation-dependent.)
- 필요에 따라 각 클래스마다 이 메서드를 override하여 사용한다.
    - 예) Integer 클래스는 정수값을 hashCode로 사용

해시함수의 예 : hashCode() for Strings in Java

```
public final class String { private final char[] s; ... public int hashCode() { int hash
= 0; for (int i = 0; i < length(); i++) hash = s[i] + (31 * hash); return hash; } }
```

사용자 정의 클래스의 예

- 모든 멤버들을 사용하여 hashCode를 생성한다.

```
public class Record { private String name; private int id; private double value; ...
public int hashCode() { int hash = 17; //nonzero constant hash = 31 * hash +
name.hachCode(); hash = 31 * hash + Integer.valueOf(id).hashCode(); hash = 31 * hash +
Double.valueOf(value).hashCode(); return hash; } }
```

hashCode와 hash 함수

- Hash code : -2^31에서 2^31사이의 정수
- Hash 함수 : 0에서 M-1까지의 정수 (배열 인덱스)
  - 0x7fffffff을 &연산 하는 이유는 음수일 경우 양수로 바꿔주는 작업이다.
  - 나머지 연산의 피연산자가 양수여야 하기 때문이다.

```
private int hash(Key key) {  return (key.hashCode() & 0x7fffffff) % M; }
```

HashMap in Java

- TreeMap 클래스와 유사한 인터페이스를 제공(둘 다 java.util.Map 인터페이스를 구현)
- 내부적으로 하나의 배열을 해시 테이블로 사용
- 해시 함수는 24페이지의 것과 유사함
- chaining으로 충돌 해결
- Load factor를 지정할 수 있음(0 ~ 1 사이의 실수)
- 저장된 키의 개수가 load factor를 초과하면 더 큰 배열을 할당하고 저장된 키들을 재배치(re-hashing)

HashSet in Java

```
HashSet<MyKey> set = new HashSet<MyKey>(); set.add(MyKey); if (set.contains(theKey)) ...
int k = set.size(); set.remove(theKey); Iterator<MyKey> it = set.iterator(); while (it.h
asNext()) {  MyKey key = it.next();  if (key.equals(aKey))    it.remove(); }
```