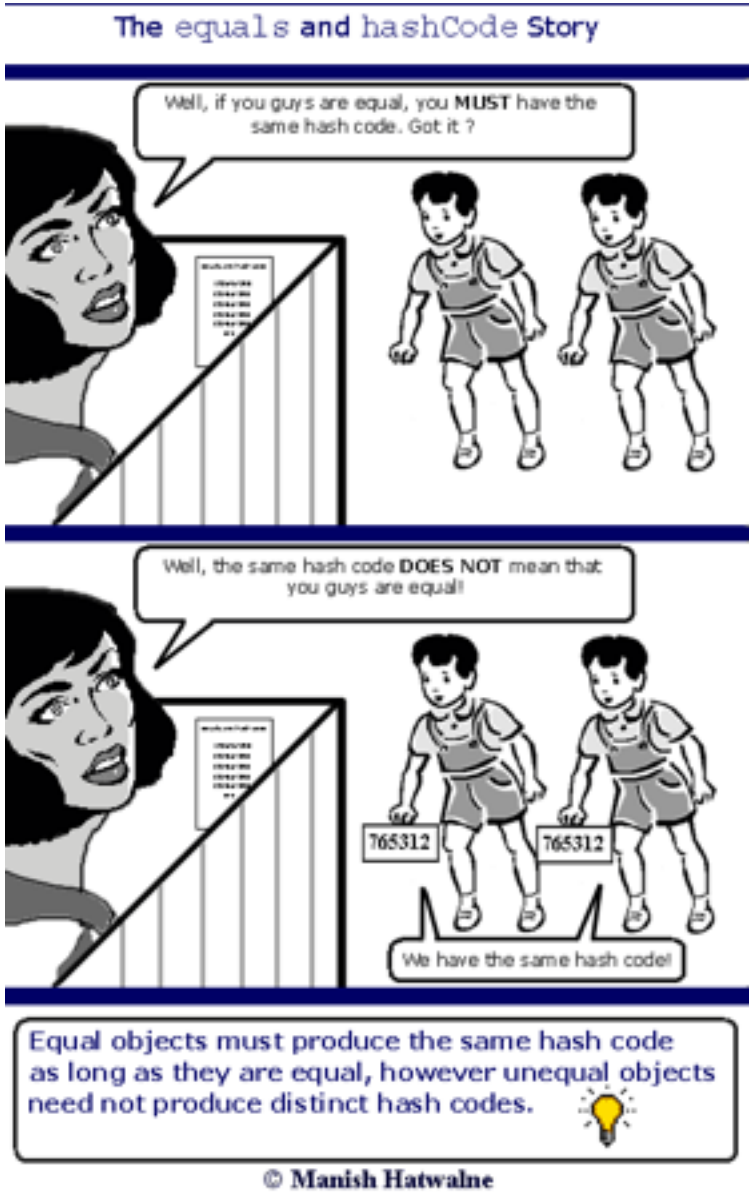


제5장

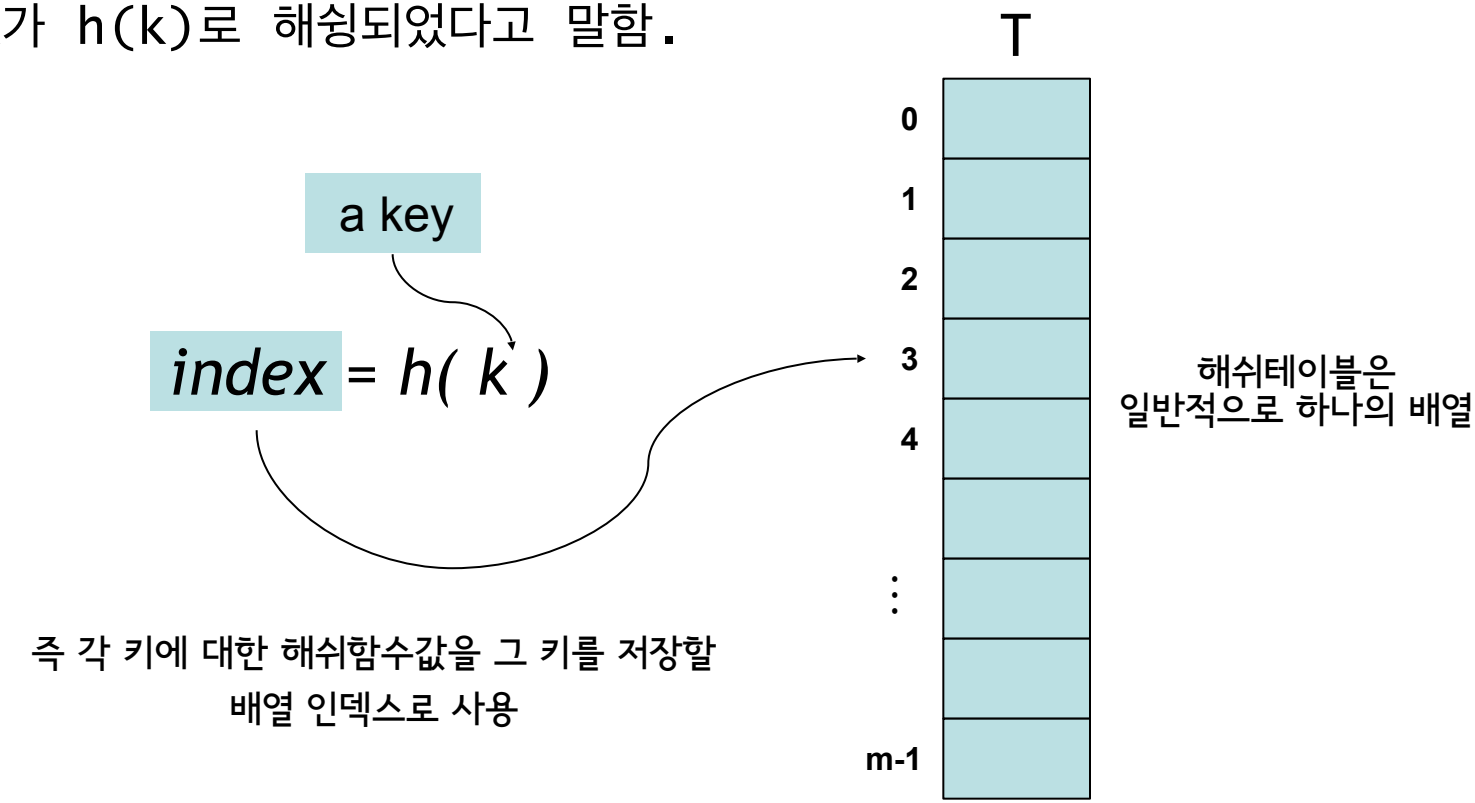
해쉬 (Hashing)

Hash Table

- 해쉬 테이블은 dynamic set을 구현하는 효과적인 방법의 하나
  - 적절한 가정하에서 평균 탐색, 삽입, 삭제시간 O(1)
  - 보통 최악의 경우 Θ(n)

Hash Table

- 해쉬 함수(hash function) h를 사용하여 키 k를 T[h(k)]에 저장
  - h : U → {0,1,...,m-1},
    여기서 m은 테이블의 크기, U는 모든 가능한 키들의 집합
  - 키 k가 h(k)로 해쉬되었다고 말함.



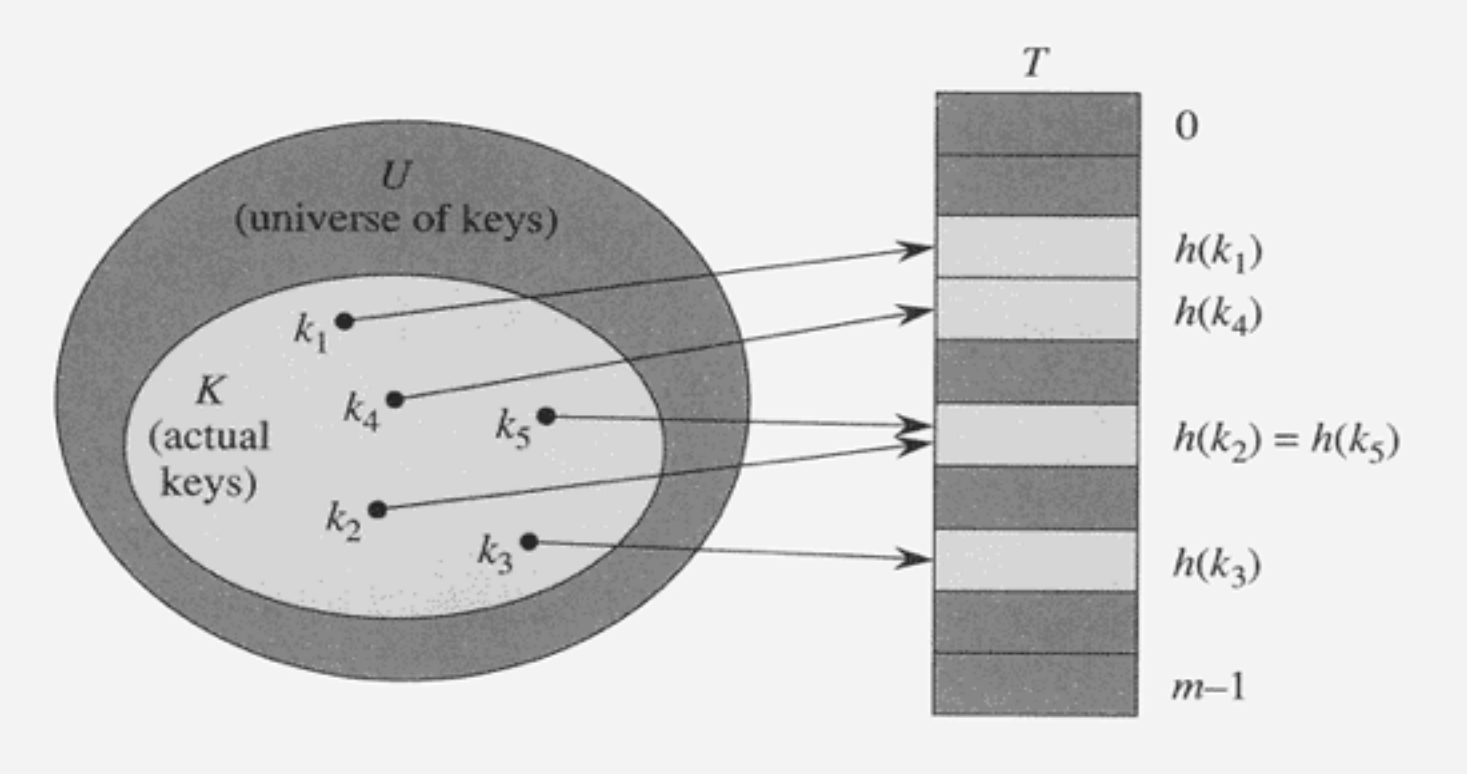
해쉬 함수의 예

- 모든 키들을 자연수라고 가정. 어떤 데이터든지 자연수로 해석하는 것이 가능
- 예: 문자열
  - ASCII 코드: C=67, L=76, R=82, S=83.
  - 문자열 CLRS는
    $(67 \cdot 128^3)+(76 \cdot 128^2)+(82 \cdot 128^1)+(83 \cdot 128^0)=141,764,947$
- 해쉬 함수의 간단한 예:
  - h(k) = k % m, 즉 key를 하나의 자연수로 해석한 후 테이블의 크기 m으로 나눈 나머지
  - 항상 0~m-1 사이의 정수가 됨

충돌(collision)

- 두 개 이상의 키가 동일한 위치로 해쉉되는 경우
- 즉, 서로 다른 두 키 k_1과 k_2에 대해서 $h(k_1)=h(k_2)$인 상황
- 일반적으로 $|U|>>m$이므로 항상 발생 가능 (즉 단사함수가 아님)
- 만약 $|K|>m$라면 당연히 발생, 여기서 K는 실제로 저장된 키들의 집합
- 충돌이 발생할 경우 대처 방법이 필요
- 대표적인 두 가지 충돌 해결 방법: chaining과 open addressing

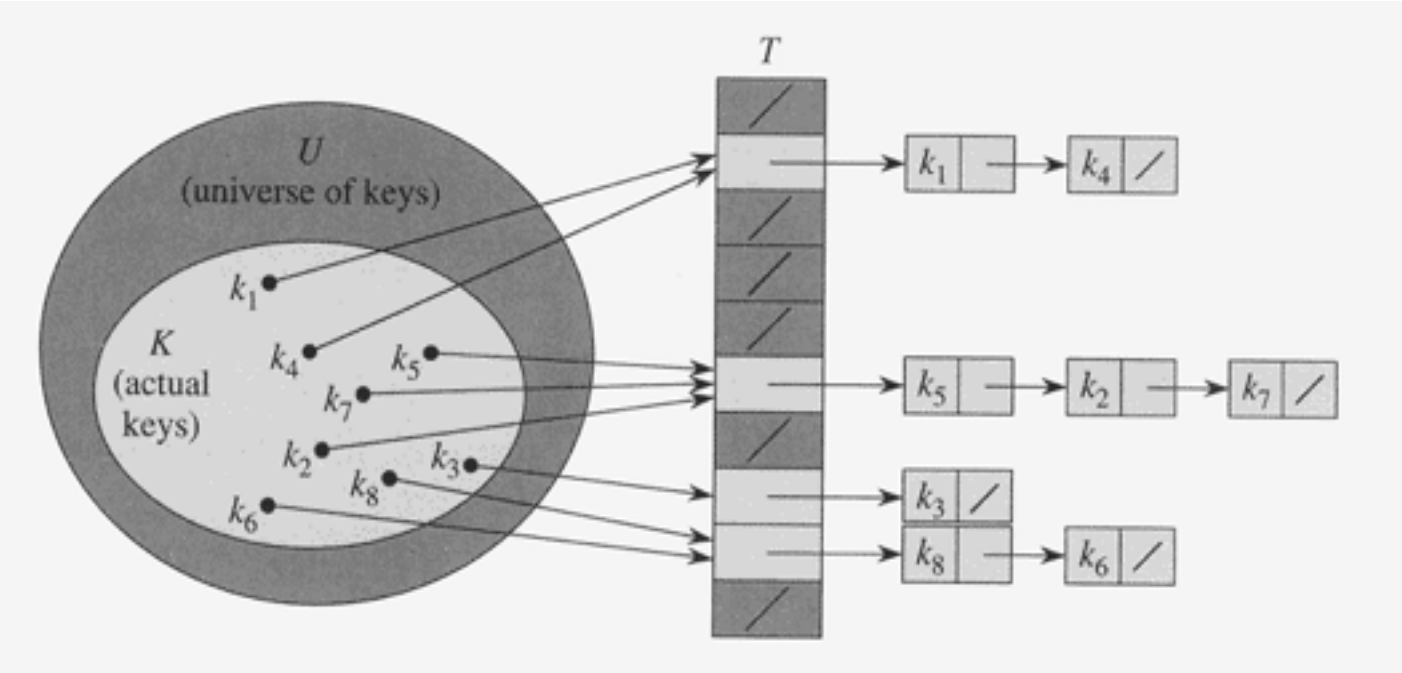
충돌(collision)



k_2와 k_5가 충돌

Chaining에 의한 충돌 해결

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



Chaining에 의한 충돌 해결

- 키의 삽입(Insertion)
  - 키 k를 리스트 T[h(k)]의 맨 앞에 삽입: 시간복잡도 O(1)
  - 중복된 키가 들어올 수 있고 중복 저장이 허용되지 않는다면 삽입시 리스트를 검색해야 함. 따라서 시간복잡도는 리스트의 길이에 비례
- 키의 검색(Search)
  - 리스트 T[h(k)]에서 순차검색
  - 시간복잡도는 키가 저장된 리스트의 길이에 비례.
- 키의 삭제(Deletion)
  - 리스트 T[h(k)]로 부터 키를 검색 후 삭제
  - 일단 키를 검색해서 찾은 후에는 O(1)시간에 삭제 가능

Chaining에 의한 충돌 해결

- 최악의 경우는 모든 키가 하나의 슬롯으로 해쉉되는 경우
  - 길이가 n인 하나의 연결리스트가 만들어짐
  - 따라서 최악의 경우 탐색시간은 Θ(n)+해쉬함수 계산시간
- 평균시간복잡도는 키들이 여러 슬롯에 얼마나 잘 분배되느냐에 의해서 결정

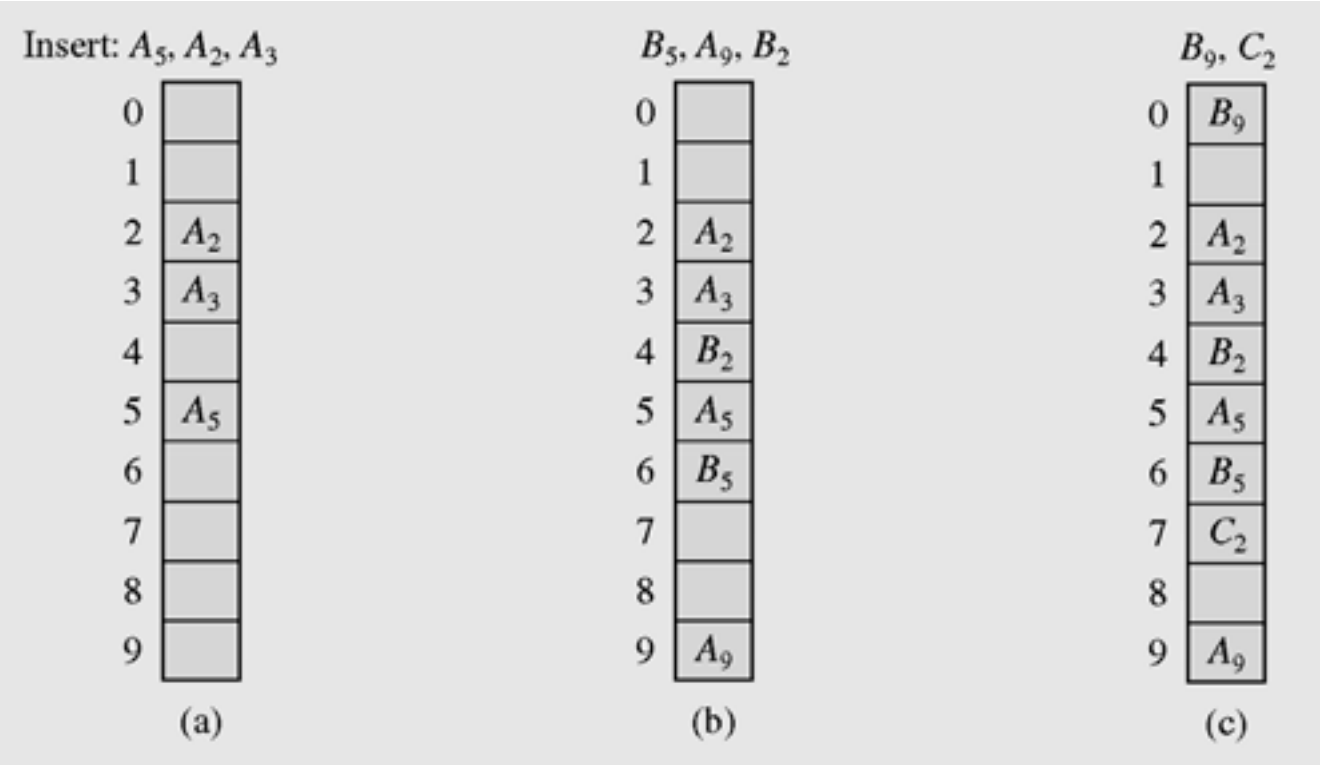
SUHA (Simple Uniform Hashing Assumption)

- 각각의 키가 모든 슬롯들에 균등한 확률로(eually likely) 독립적으로 (independently) 해쉥된다는 가정
  - 성능분석을 위해서 주로 하는 가정임
  - hash함수는 deterministic하므로 현실에서는 불가능
- Load factor $\alpha = n/m$:
  - n: 테이블에 저장될 키의 개수.
  - m: 해쉬테이블의 크기, 즉 연결리스트의 개수
  - 각 슬롯에 저장된 키의 평균 개수
- 연결리스트 T[j]의 길이를 n_j라고 하면 $E[n_j] = \alpha$
- 만약 n=O(m)이면 평균검색시간은 O(1)

Open Addressing에 의한 충돌 해결

- 모든 키를 해쉬 테이블 자체에 저장
- 테이블의 각 칸(slot)에는 1개의 키만 저장
- 충돌 해결 기법
  - Linear probing
  - Quadratic probing
  - Double hashing

Open Addressing - Linear Probing



h(k), h(k)+1, h(k)+2,… 순서로 검사하여 처음으로 빈 슬롯에 저장

테이블의 끝에 도달하면 다시 처음으로 circular하게 돌아감

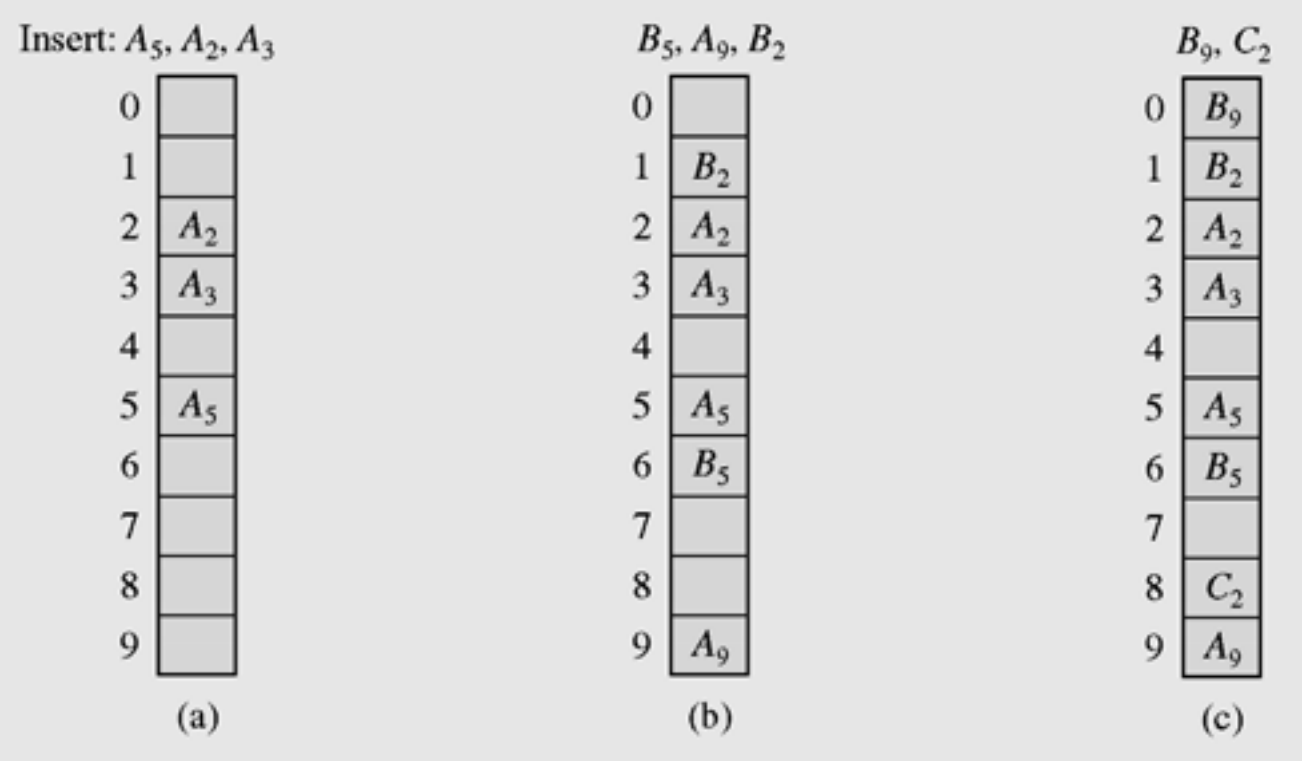
다른 방법들

- Linear probing의 단점
  - primary cluster: 키에 의해서 채워진 연속된 슬롯들을 의미
  - 이런 cluster가 생성되면 이 cluster는 점점 더 커지는 경향이 생김
- Quadratic probing
  - 충돌 발생시 $h(k)$, $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$,... 순서로 시도
- Double hashing
  - 서로 다른 두 해쉬 함수 h_1과 h_2를 이용하여

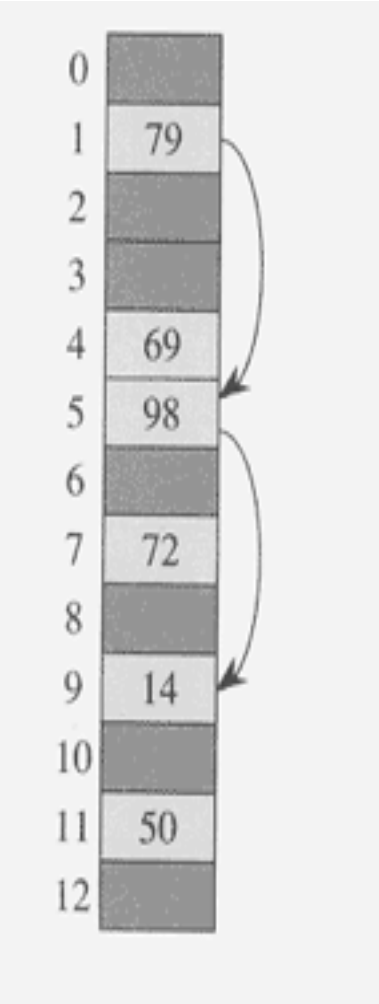
$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Open Addressing - Quadratic Probing

- 충돌 발생시 h(k), h(k)+1^2, h(k)+2^2, h(k)+3^2,... 순서로 시도



Open Addressing - Double hashing

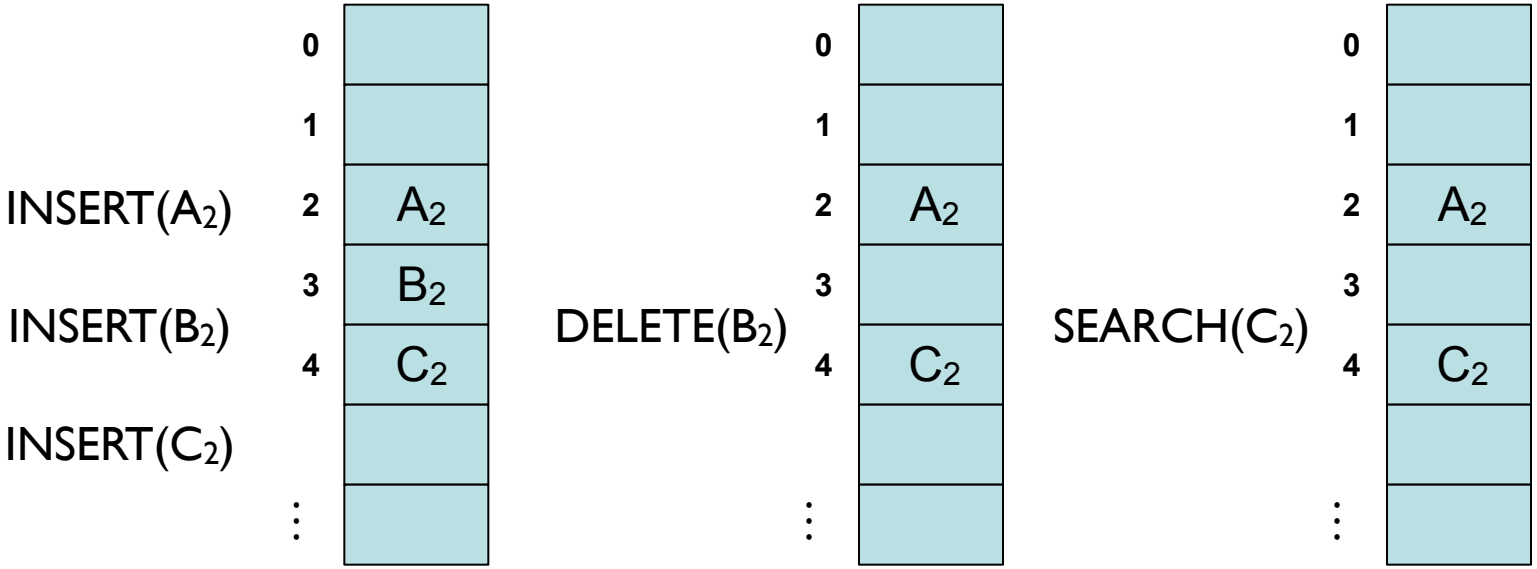


m=13
$h_1(k)$=k mod 13
$h_2(k)$ = 1 + (k mod 11)

k=14인 경우

Open Addressing - 키의 삭제

- 단순히 키를 삭제할 경우 문제가 발생
  - 가령 A_2, B_2, C_2가 순서대로 모두 동일한 해쉬함수값을 가져서 linear probing으로 충돌 해결(왼쪽 그림)
  - B_2를 삭제한 후(가운데 그림) C_2를 검색



좋은 해쉬 함수란?

- 현실에서는 키들이 랜덤하지 않음
- 만약 키들의 통계적 분포에 대해 알고 있다면 이를 이용해서 해쉬 함수를 고안하는 것이 가능하겠지만 현실적으로 어려움
- 키들이 어떤 특정한 (가시적인) 패턴을 가지더라도 해쉬함수값이 불규칙적이 되도록 하는게 바람직
  - 해쉬함수값이 키의 특정 부분에 의해서만 결정되지 않아야

해쉬 함수

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

해쉬 함수

- Multiplication 기법 :
  - 0에서 1사이의 상수 A를 선택: 0<A<1.
  - kA의 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.···
  - 즉, h(21) = 4

Hashing in Java

Hash code in Java

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

해쉬함수의 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;
   }
}
```

$$h(s) = 31^{L-1} \cdot s_0 + \cdots + 31^1 \cdot s_{L-2} + 31^0 \cdot s_{L-1}$$

사용자 정의 클래스의 예

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

모든 멤버들을 사용하여 hashCode를 생성한다.

hashCode와 hash 함수

- Hash code: -2^{31}에서 2^{31}사이의 정수
- Hash 함수: 0에서 M-1까지의 정수 (배열 인덱스)

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

HashMap in Java

- 4장에서 다룬 TreeMap 클래스와 유사한 인터페이스를 제공 (둘 다 java.util.Map 인터페이스를 구현)
- 내부적으로 하나의 배열을 해쉬 테이블로 사용
- 해수 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.hasNext()) {
    MyKey key = it.next();
    if (key.equals(aKey))
        it.remove();
}
```

unordered_set in C++

```
#include <unordered_set>
#include <iostream>

int main ()
{
    // initialize set
    std::unordered_set<std::string> s;
    s.insert("red");
    s.insert("green");
    s.insert("blue");
    s.erase("green");

    // search for membership
    if (s.find("red") != s.end())
        std::cout << "found red" << std::endl;
    return 0;
}
```