해 싱
(Hashing)

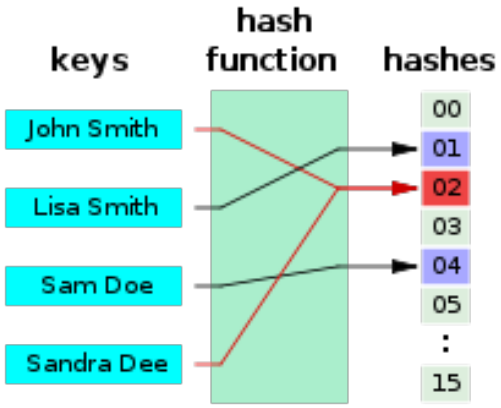
소프트웨어학과

이 의 종

개요
(Introduction)

"Hash" is really a broad term with different formal meanings in different contexts.

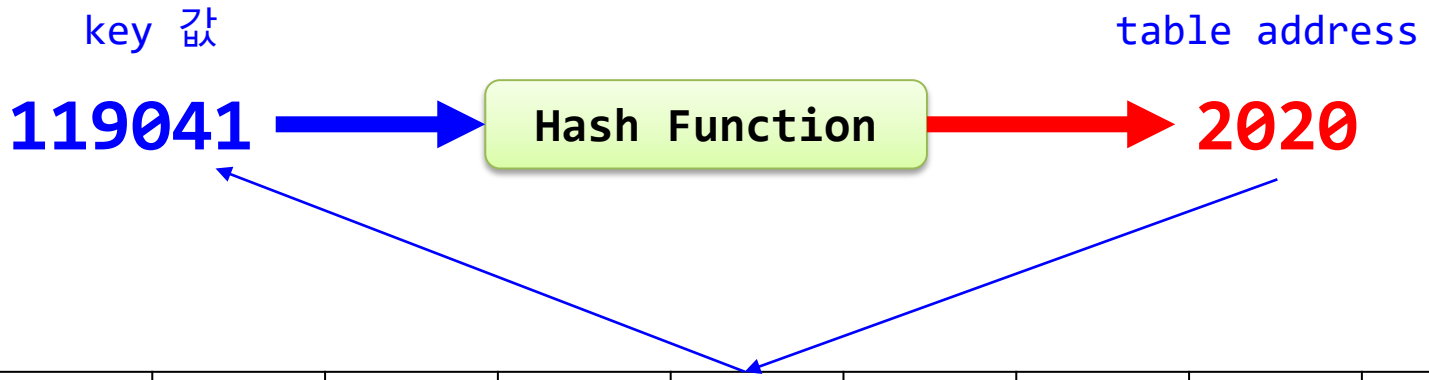
A "hash" is a function *h* referred to as hash function that takes as input objects and outputs a string or number. The input objects are usually members of basic data types like strings, integers, or bigger ones composed of other objects like user defined structures. The output is a typically a number or a string. The noun "hash" often refers to this output. The verb "hash" often means "apply a hash function".



문제 정의(Problem Definition)

배열에서 119041을 어떻게 찾을 것인가?

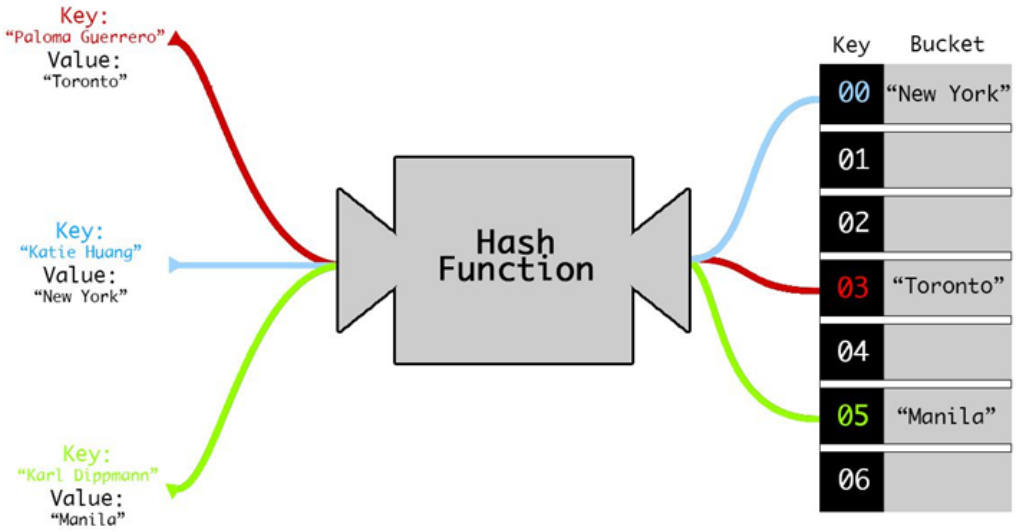
2	315	...	453902	...	119041	...	3	1	2019
table[0]	table[1]		table[1991]		table[2020]		[2997]	[2998]	[2999]



2	315	...	453902	...	119041	...	3	1	2019
table[0]	table[1]		table[1991]		table[2020]		[2997]	[2998]	[2999]

해싱의 정의(hashing)

- 데이터를 해시테이블(hash table)이라는 배열에 저장하는 것
- 데이터의 키 값을 적절한 해시함수(hash function)을 이용 해시테이블 주소로 변환하여 데이터를 찾는 방법
- 키 값 비교(comparison)를 통한 탐색보다 빠르게 탐색 가능
- 레코드의 수가 증가해도 키 값을 통해 바로 해시테이블의 주소 탐색

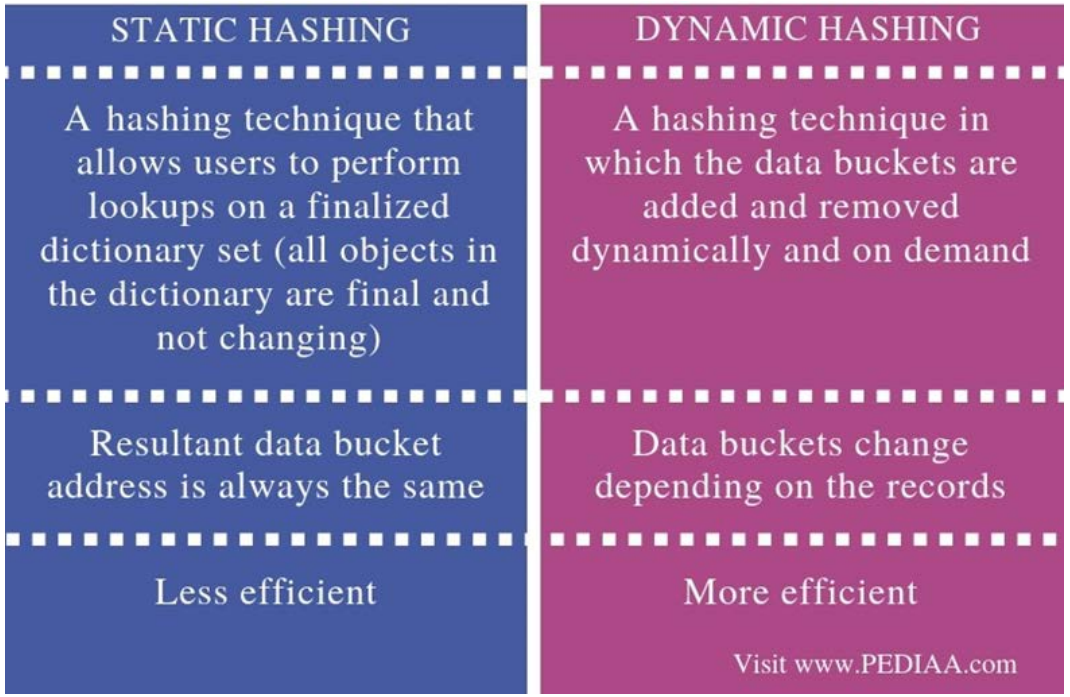


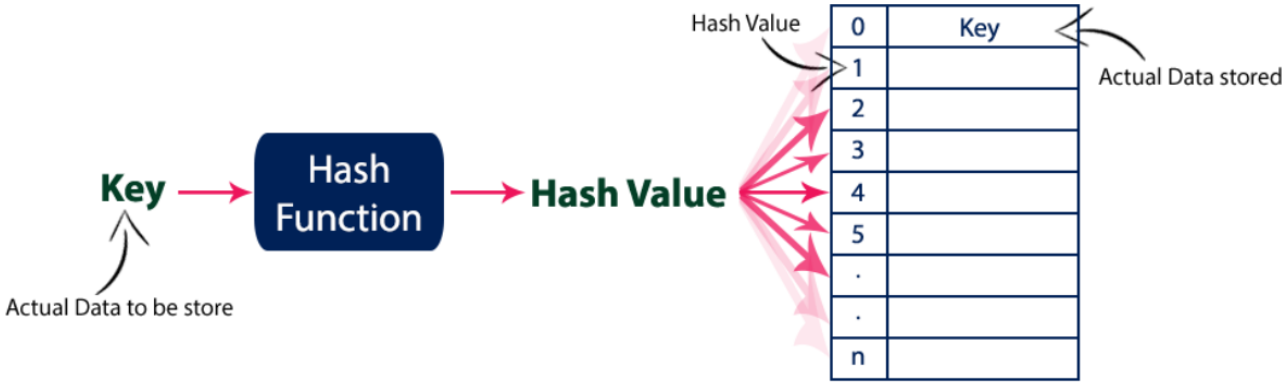
example: library

해싱의 정의

- **정적해싱(Static Hashing):**
  - the resultant data bucket address is always the same

- **동적해싱(Dynamic Hashing):**
  - the data buckets grow or shrink according to the increase and decrease of records.





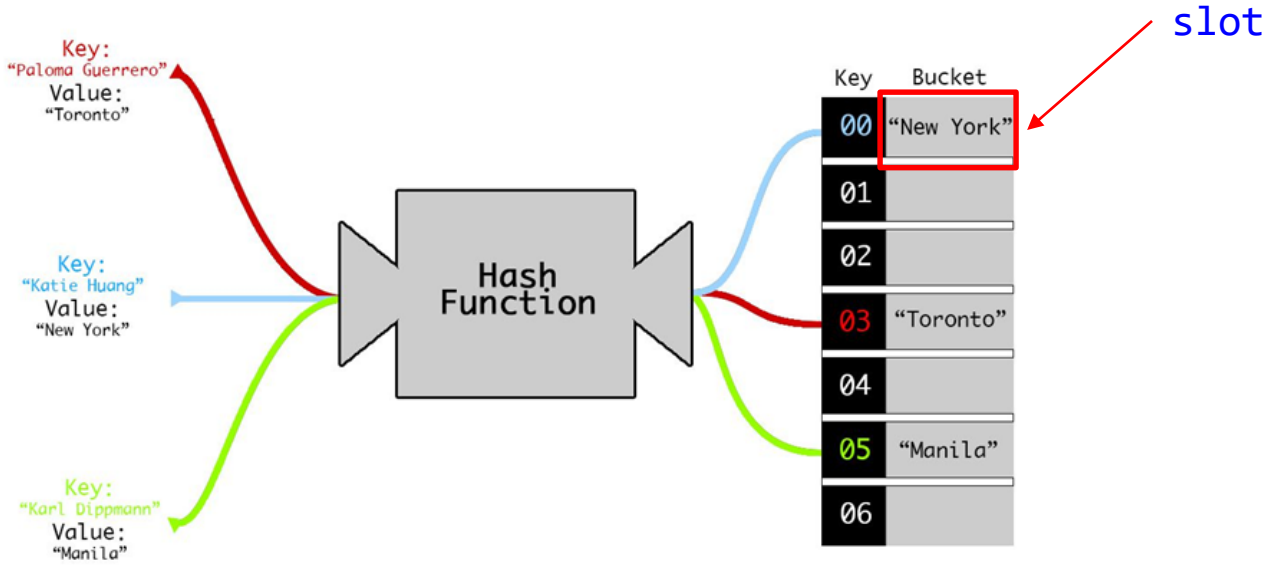
정적 해싱
(Static Hashing)

In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure. In all these search techniques, as the number of elements increases the time required to search an element also increases linearly.

Hashing is another approach in which time required to search an element doesn't depend on the total number of elements. Using hashing data structure, a given element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

해시 테이블(Hash Table)

- 데이터가 해시 테이블 ht에 저장
- ht[0], … , ht[b-1], 즉, b개의 버킷(bucket)으로 분할됨
- 한 버킷은 s개의 슬롯(slot)으로 구성됨
- 한 슬롯에는 하나의 사전 쌍이 저장됨
- 키 값이 k인 데이터의 주소 또는 위치는 해시 함수 h에 의해 결정됨
- 해시 함수는 키 값을 버킷으로 사상(mapping)시킴
- h(k) : k의 해시 또는 홈 주소(home address)



해시 테이블(Hash Table)

- 해시 테이블의 키 밀도(key density): n/T
  - n : 테이블에 있는 쌍의 수
  - T : 가능한 키의 총 개수

전체 키 값들 중에서 현재 해시 테이블에서 사용 가능한 키 값의 정도

- 해시 테이블의 적재 밀도(loading density): $\alpha = n/(sb)$
  - s : slot의 수
  - b : bucket의 수

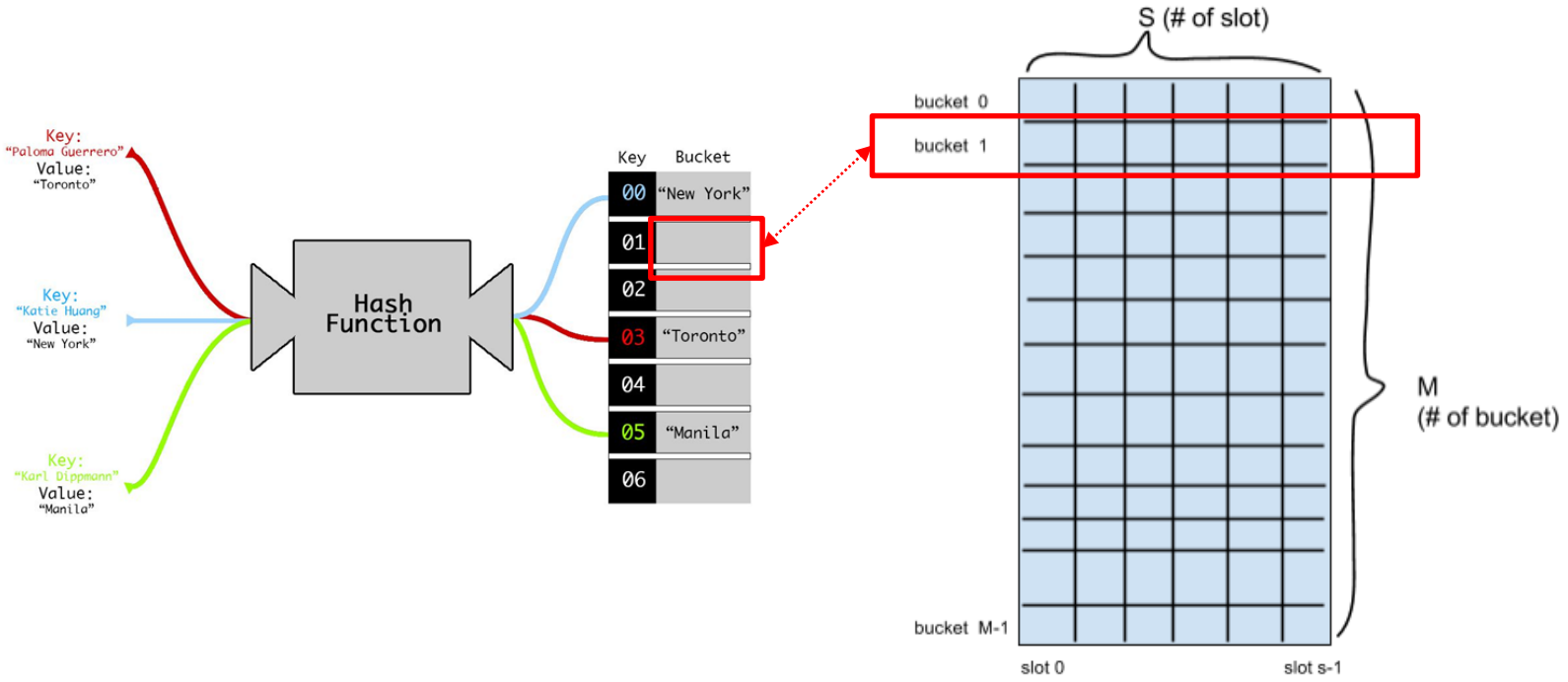
해시 테이블에 저장 가능한 키 값의 개수 중에서 현재 해시 테이블에 저장되어서 실제 사용되고 있는 키 값의 정도

- 키 밀도 n/T는 매우 작고 버킷 수 b도 T보다 작게 됨
- 동거자(synonym)
  - k_1와 k_2에 대해 $h(k_1) = h(k_2)$ 인 경우, k_1과 k_2를 h에 대한 동거자(synonym)라 함

해시 테이블(Hash Table)

- 충돌(collision)
  - 새로운 쌍의 삽입 시 홈 버킷에 비어 있지 않은 경우
  - k_1와 k_2에 대해 $h(k_1) = h(k_2)$ 인 경우 (hash 값이 같은 경우)

- 오버플로(overflow)
  - 버킷에 할당된 슬롯(slot)을 이미 소진하여 더 이상 버킷에 항목을 저장할 수 없는 경우 (여유 slot이 없어 저장할 공간이 없는 경우)



해시 테이블(Hash Table)

- b=26개의 버킷과 s=2인 해시 테이블
  - n=10개, α =10/52 = 0.19
  - 각 식별자를 0-25 중의 한 숫자로 사상시킴

clock이 버킷 ht[2]로 해시되면 overflow발생

clock을 테이블 어디에 위치시켜야 되는가?

	Slot 0	Slot 1
0	acos	atan
1		
2	char	ceil
3	define	
4	exp	
5	float	floor
6		
...		
25		

acos와 atan은 동거자(synonyms)

ceil과 char는 동거자

float과 floor는 동거자

26개의 버킷과 버킷 당 2개의 슬롯으로 구성된 해시 테이블

해시 테이블(Hash Table)

- 오버플로가 발생하지 않을 경우
  - 삽입, 삭제, 탐색 시간
    - 사전의 엔트리 수인 n에 무관
    - 해시 함수의 계산 시간과 한 버킷을 탐색하는데 소요되는 시간에만 좌우됨
  - 버킷 크기 s는 작기 때문에, 버킷 내에서 탐색을 위해 순차 탐색 기법을 사용할 수 있음
- 일반적으로 오버플로는 필연적으로 발생
- **해싱 기법**
  - 해시 함수 h를 사용하여 키를 해시 테이블 버킷에 사상시킴
  - 해시 함수는 충돌 수 최소화하도록 선택해야 함(hash값을 최대한 분산)
  - 오버플로 처리하는 기법이 필요함

해시 함수(Hash Function)

- **키를 해시 테이블 내의 버킷으로 사상시킴**
  - 계산이 쉽고 충돌이 적어야 함

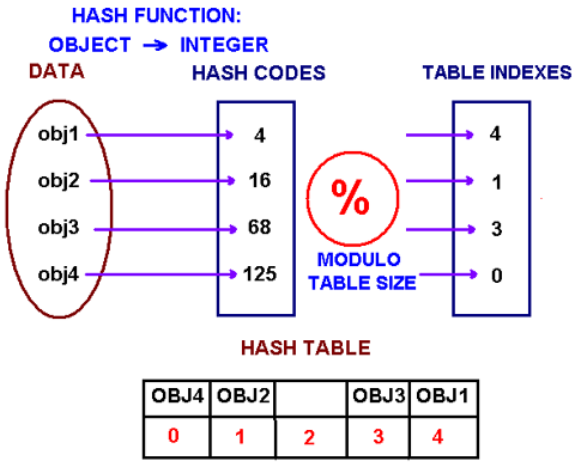
- **균일 해시 함수(uniform hash function)**
  - h(k) = i 가 될 확률은 모든 버킷 i에 대해 i/b이 됨
    - k = 키 공간에서 임의로 선택된 키
  - b개의 버킷 각각에 임의의 k가 대응될 확률은 모두 같게 됨

- **다양한 종류의 균일 해시 함수가 사용됨**
  - 키를 산술적인 연산이 가능한 데이터 타입(예: 정수)으로 변환
    - 많은 응용에서 산술적인 연산을 할 수 없는 경우도 있음

해시 함수(Hash Function)

- **해시 함수의 조건**
  - **해시 함수는 계산이 쉬워야 함**
    - 비교 검색 방법의 키 값 비교연산 수행 시간보다 해싱 함수를 사용하여 계산하는 시간이 빨라야 해싱 검색을 사용하는 의미가 있음
  - **해시 함수는 충돌이 적어야 함**
    - 충돌이 많다는 것은 동일 버킷을 할당 받는 키 값이 많다는 것
    - 비어있는 버킷이 많은데도 어떤 버킷은 오버플로가 발생할 수 있는 상태가 되므로 좋은 해시 함수가 될 수 없음
  - 해시 테이블에 고르게 분포할 수 있도록 주소를 만들어야 함

해시 함수(Hash Function)



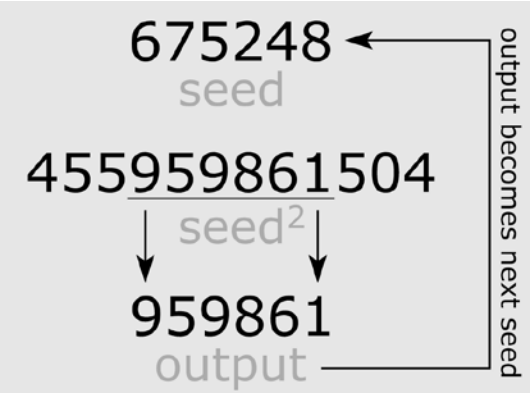
- **제산(division) 함수**
  - 실제로 가장 많이 쓰이는 해시 함수
  - 키 값이 음이 아닌 정수라고 가정
  - 홈 버킷은 모드(%) 연산자에 의해 결정
    - 키 k를 정해진 수 D로 나눈 나머지를 k의 홈 버킷으로 사용
      - h(k) = k % D
      - 버킷 주소의 범위 : 0 ~ (D-1), 최소 b = D개의 버킷이 있어야 함
  - D의 선택이 오버플로 발생 수에 영향 미침
  - D가 홀수가 되도록 제한하고 b = D가 되도록 함
  - 배열의 크기를 두 배로 만들면 버킷의 수가 b에서 **2b+1**로 증가됨

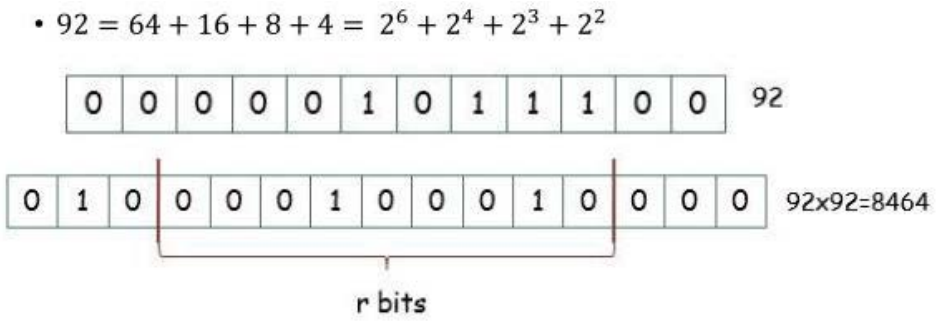
to make it prime number

The number 2p + 1 associated with a Sophie Germain prime is called a safe prime.

해시 함수(Hash Function)

- **중간 제곱(mid-square) 함수**
  - 키를 제곱한 후에 결과의 중간에 있는 적절한 수의 비트를 취함

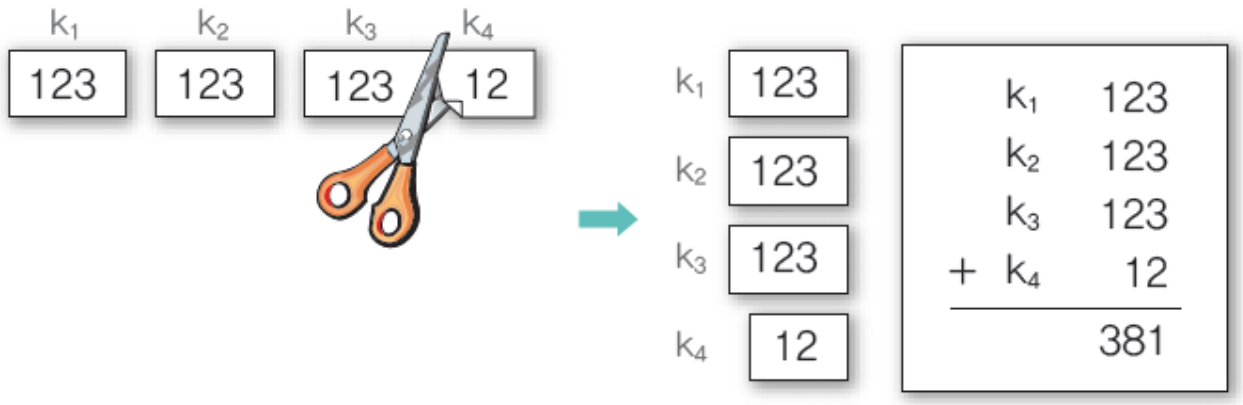




  - 제곱 수의 중간 비트는 그 키의 모든 비트에 의존하므로 서로 다른 키들은 서로 다른 해시 주소를 갖게 될 확률이 높게 됨
  - 사용되는 비트의 수는 테이블 크기에 달려 있음
    - r개의 비트가 사용되면 각 값들의 범위는 0에서 2^r-1까지가 됨
    - 해시 테이블의 크기는 2의 제곱이 되게 선정

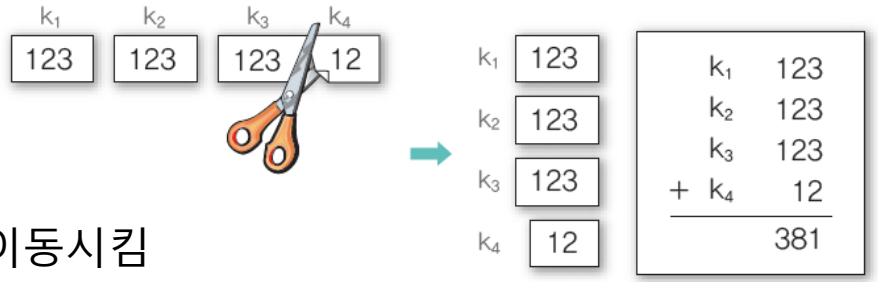
해시 함수(Hash Function)

- **접지(folding) 함수**
  - 숫자로 된 키 k를 몇 부분으로 나누는데, 마지막 부분을 제외하고는 모두 길이가 같음
  - 각 부분들을 서로 더하여 k에 대한 해시 주소 만듦

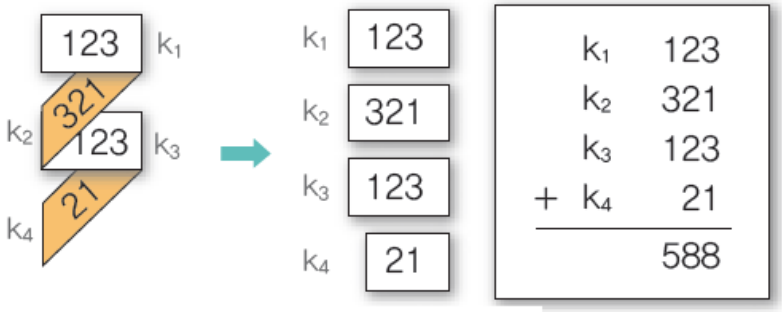


해시 함수(Hash Function)

- **두 가지 접지(folding) 방식**
  - 이동 접지 (shift folding)
    - 마지막을 제외한 모든 부분들을 이동시킴
    - 최하위 비트가 마지막 부분의 자리와 일치하도록 맞춤
    - 서로 다른 부분들을 더하여 h(k)를 얻음

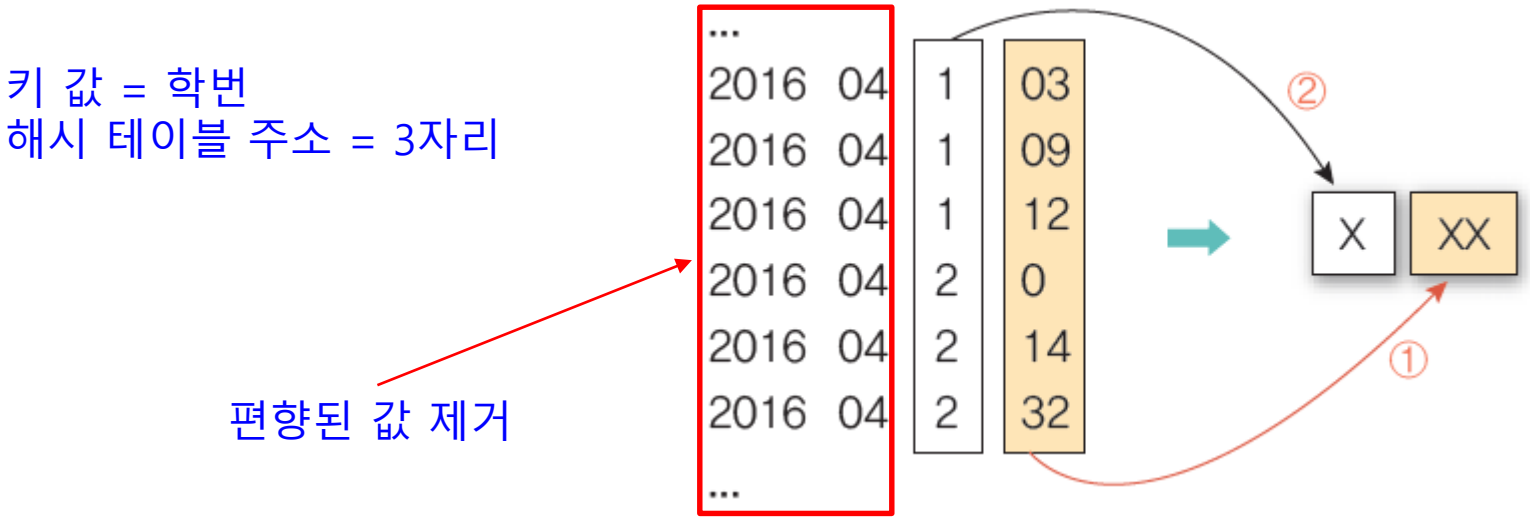


  - 경계 접지 (folding at the boundaries)
    - 키의 각 부분들을 경계에서 겹치게 함
    - 같은 자리에 위치한 수들을 더하여 h(k)를 얻음



해시 함수(Hash Function)

- **숫자 분석 함수(digital analysis)**
  - 키 값의 각 자릿수의 분포를 분석하여 해시 주소로 사용
  - 각 키를 어떤 기수(radix) r을 이용해 하나의 숫자로 바꿈
  - 이 기수를 이용해 각 키의 숫자들을 검사함
  - 가장 편향된(skewed) 분산을 가진 숫자 생략
  - 남은 숫자만으로 해시 테이블 주소를 결정함



정적 해싱(Static Hashing)

- **키를 정수로 변환**
  - 키를 음이 아닌 정수로 변환함
  - 유일한 정수로 변환시킬 필요는 없음
  - 스트링을 동일한 정수로 변환시키기만 하면 됨

```
unsigned int stringToInt(char *key) {
        int number = 0;
        while (*key)
             number += *key++;
        return number;
}
```

하나의 스트링을 음이 아닌 정수로 변환하는 방법

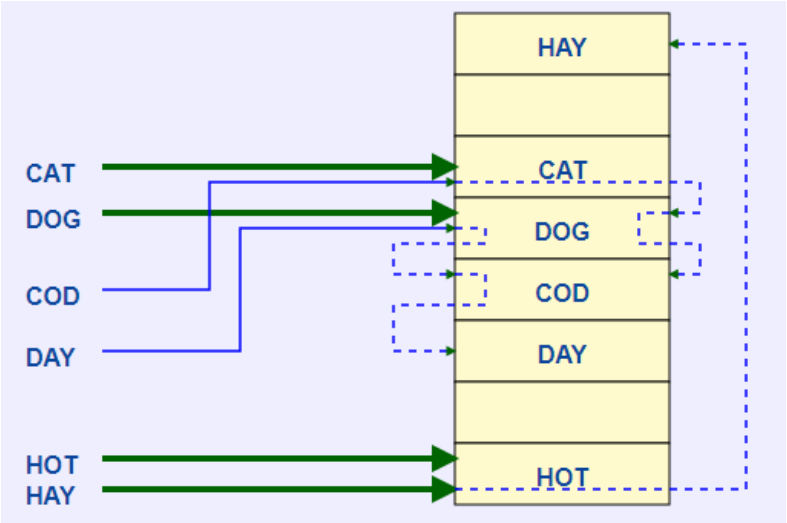
하나씩 건너뛴 문자에 대응하는 정수를 8비트 이동시켜 합산함.

정수의 범위를 넓게 만듦

```
unsigned int stringToInt(char *key) {
       int number = 0;
       while (*key)       {
              number += *key++;
              if(*key) number += ((int) *key++) << 8;
       }
       return number;
}
```

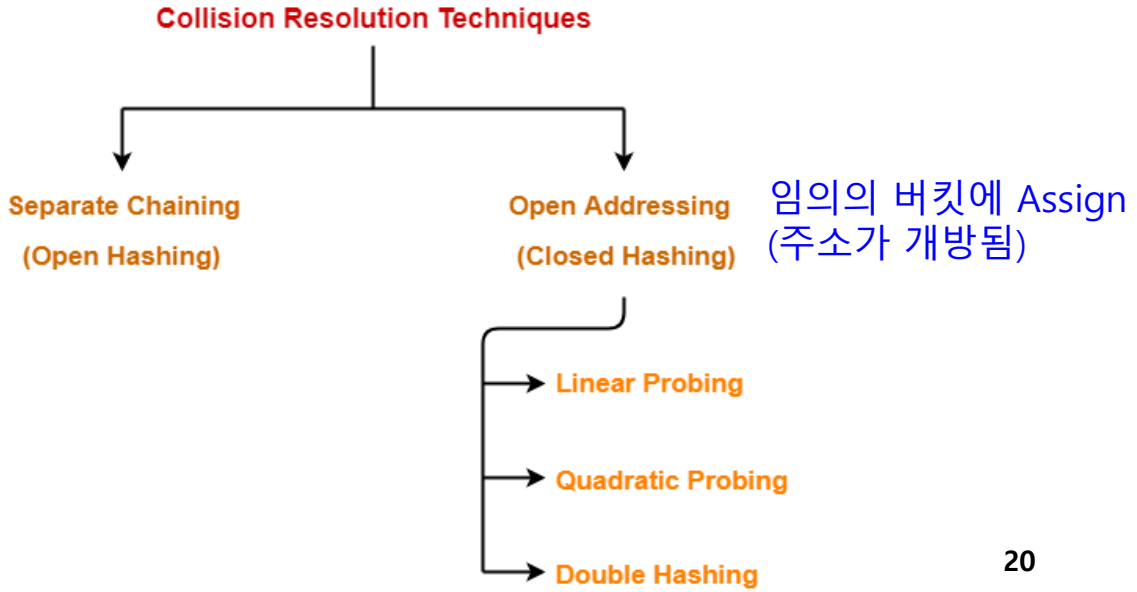
오버플로 처리(Overflow Handling)

- **개방 주소법(open addressing)**
  - 선형 조사법(linear probing)
  - 이차 조사법(quadratic probing)
  - 재해싱(double hashing)
  - 임의 조사법(random probing)



- **체인법 (chaining)**

해당 버킷에 Assign
(다른 버킷의 주소가 개방되지 않음)



임의의 버킷에 Assign
(주소가 개방됨)

오버플로 처리(Overflow Handling)

- **선형 조사법 (Linear Probing)**
  - 오버플로가 발생하면, 그 다음 버킷에 빈 슬롯이 있는지 조사
  - ht[h(k)+i] % b의 순서에 따라 해시 테이블 버킷 검색
    - h : 해시 함수, b : 버킷의 수, 0 <= i <= b - 1
  - 빈 버킷(슬롯)이 있으면 데이터가 그 버킷(슬롯)으로 삽입
  - 빈자리가 있는 버킷이 없다면 해시 테이블이 만원인 것
    - 테이블 크기를 늘려야 함
    - 적재 밀도 = 0.75와 같은 경계 값을 넘을 때 테이블 크기 늘림
  - 해시 테이블의 크기를 다시 정할 때
    - 해시 함수 바꿔야 함
    - 모든 사전 엔트리들은 새로 커진 테이블에 다시 사상되어야 함

static hashing의 문제점

오버플로 처리(Overflow Handling)

- **선형 조사법 (Linear Probing) 예제 1**
  - 13-버킷 테이블, 버킷 당 1슬롯
  - 데이터: **for, do, while, if, else, function**
  - 제산(division) 해시 함수 이용
  - **function**과 **if:** 같은 버킷으로 해시

ASCII Code 값, f = 102, o = 111, r = 114

Identifier	Additive Transformation	x	Hash
for	102+111+114	327	2
do	100+111	211	3
while	119+104+105+108+101	537	4
if	105+102	207	12
else	101+108+115+101	425	9
function	102+117+110+99+116+105+111+110	870	12

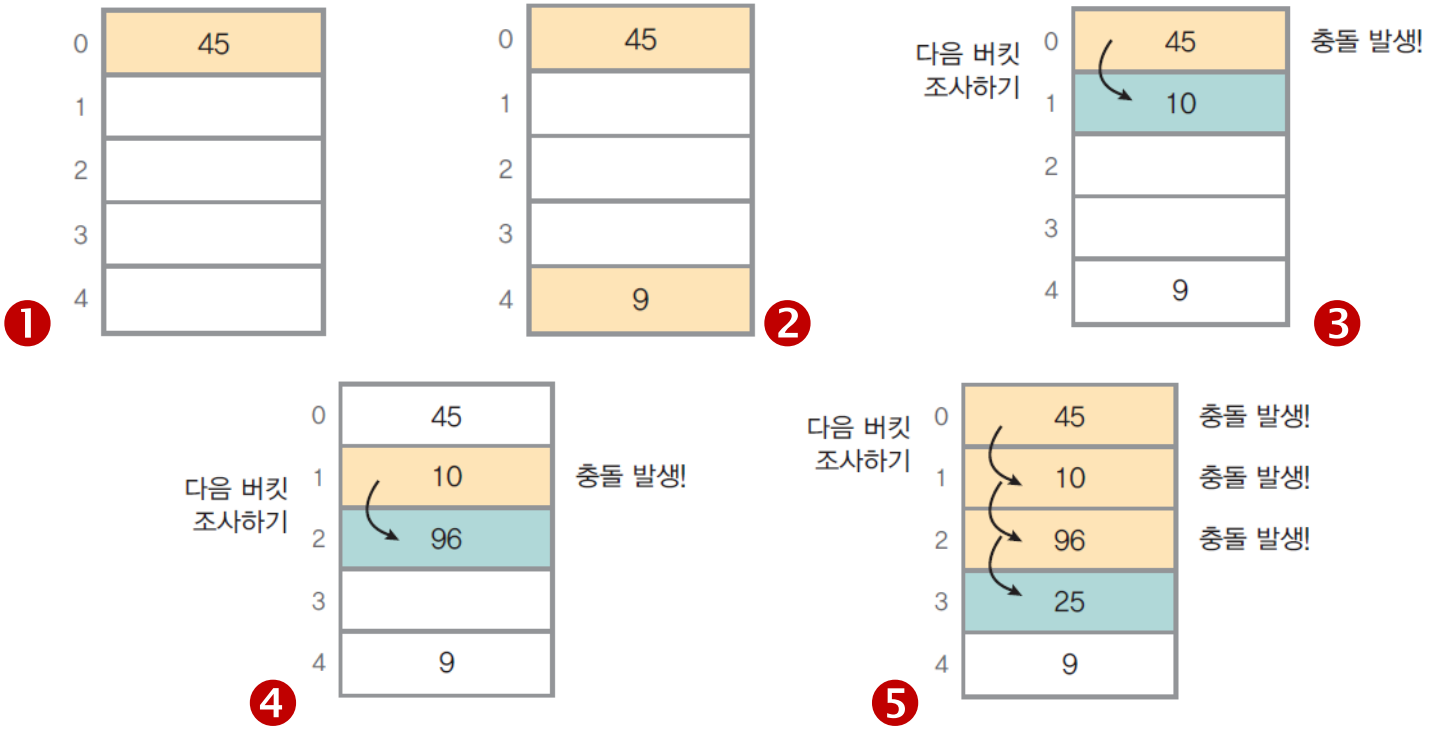
합산 변환

[0]	function
[1]	
[2]	for
[3]	do
[4]	while
[5]	
[6]	
[7]	
[8]	
[9]	else
[10]	
[11]	
[12]	if

선형 조사법을 사용하는 해시 테이블
(13개의 버킷, 버킷 당 1슬롯)

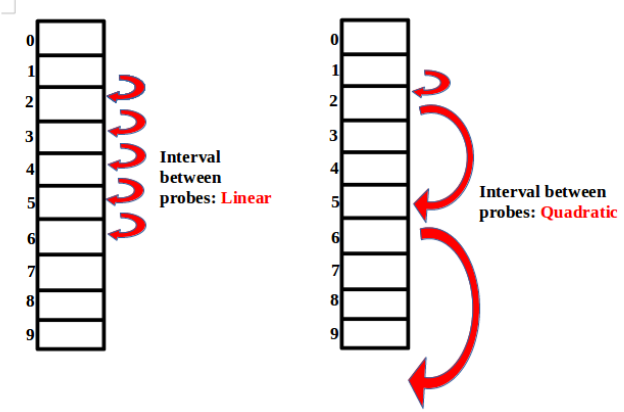
오버플로 처리(Overflow Handling)

- **선형 조사법 (Linear Probing) 예제 2**
  - 해시 테이블의 크기 : 5
  - 해시 함수 : 제산함수 사용. 해시 함수 h(k) = k mod 5
  - 저장할 키 값 : {45, 9, 10, 96, 25}

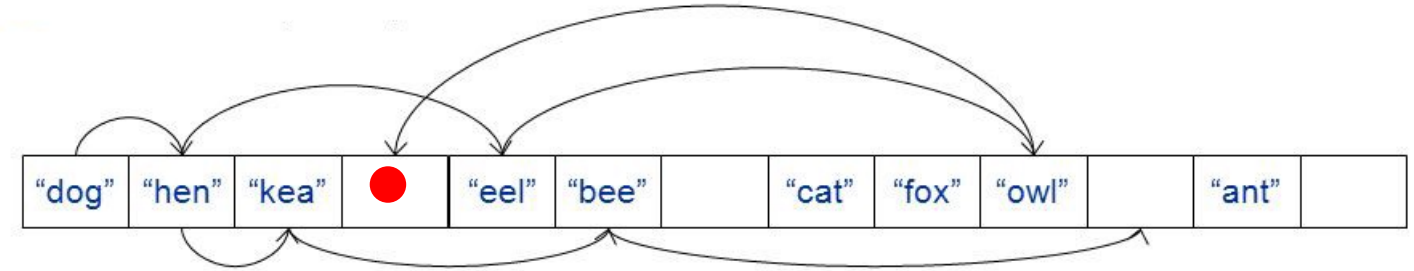


오버플로 처리(Overflow Handling)

- **2차 조사법 (Quadratic Probing)**
  - 선형 조사법에서 발생하는 집중문제를 해결하기 위한 방법
  - 특정한 수만큼 떨어진 곳을 순환적으로 빈 공간을 찾아 저장
  - $h(k), (h(k)+i)\%b, (h(k)+i^2)\%b, (0 \le i \le (b-1)/2$




```
h(bat) = 0
(h(bat) + 1)%13 = 1
(h(bat) + 4)%13 = 4
(h(bat) + 8)%13 = 8
(h(bat) + 16)%13 = 3
```



오버플로 처리(Overflow Handling)

- **재해싱(Rehashing, Double Hashing)**
  - 선형 조사법에서 발생하는 집중문제를 해결하기 위한 방법
  - 키 값에 대해 여러 개의 해시 함수 h_1, h_2, ..., h_m 사용
  - A popular second hash function = R - (key % R)
    - R = table size보다 작은 a prime number

table size = 10
$h_1(k) = k \% 10$
$h_2(k) = 7 - (k \% 7)$

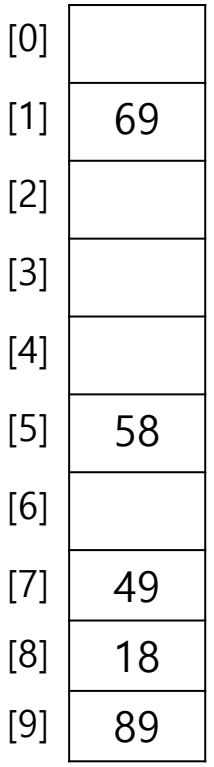
insert keys:
89, 18, 49, 58, 69

$h_1(89) = 89 \% 10 = 9$
$h_1(18) = 18 \% 10 = 8$

$h_1(49) = 49 \% 10 = 9$ (collision)
$h_2(49) = 7 - (49 \% 7) = 7$

$h_1(58) = 58 \% 10 = 8$ (collision)
$h_2(58) = 7 - (58 \% 7) = 5$

$h_1(69) = 69 \% 10 = 9$ (collision)
$h_2(69) = 7 - (69 \% 7) = 1$



오버플로 처리(Overflow Handling)

- **선형 조사법이 효율이 좋지 않은 이유**
  - 키를 탐색할 때 서로 다른 해시 값을 갖는 키와 비교

- **해결 방안**
  - 각 버킷에 대해 동거자(synonym)들을 키 리스트로 구성
    - 불필요한 비교 제거
    - 탐색 시 해시 주소 h(k)를 계산하여 h(k)에 대한 리스트만 검사

- **체인법(chaining)**
  - 체인을 사용할 때 배열 ht[0:b-1]을 이용
  - ht[i]는 버킷 i에 연결된 체인 중 첫 번째 노드를 가리킴

오버플로 처리(Overflow Handling)

chaining

```
[0] → acos atoi atol
[1] → NULL
[2] → char ceil cos ctime
[3] → define
[4] → exp
[5] → float floor
[6] → NULL
...
[25] → NULL
```

linear probing

bucket	*x*	buckets searched
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
7	float	3
8	atol	9
9	floor	5
10	ctime	9
...		
25		

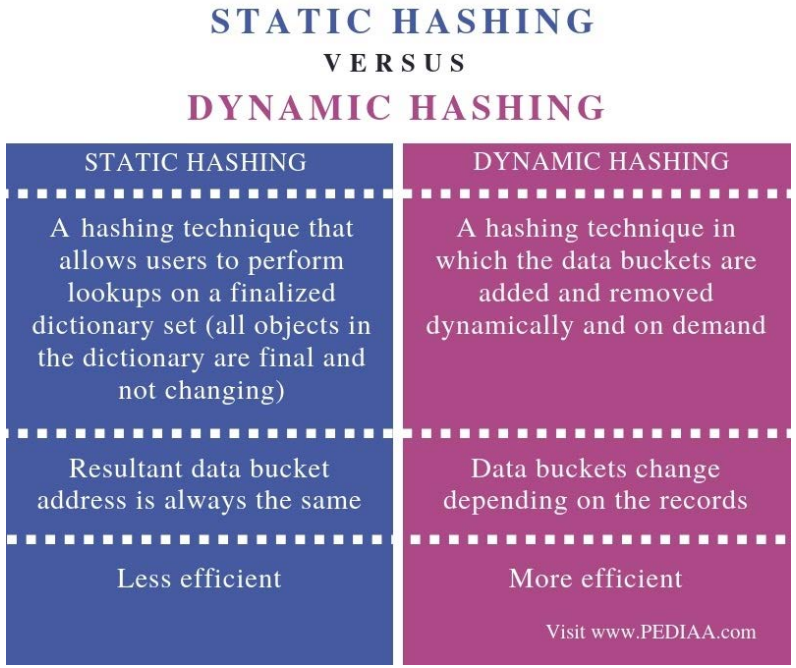
비교 횟수

- acos, char, define, exp, float → 1번
- atoi, ceil, floor → 2번
- atol, cos → 3번
- ctime → 4번
- 평균 비교 횟수 = 21/11 = 1.91

(1x5 + 2x3 + 3x2 + 4x1)/(11 words)

동적 해싱
(Dynamic Hashing)

The main difference between static and dynamic hashing is that, in static hashing, the resultant data bucket address is always the same while, in dynamic hashing, the data buckets grow or shrink according to the increase and decrease of records.



동적 해싱의 동기(Motivation for Dynamic Hashing)

- **적재 밀도가 경계값을 넘을 때마다 해시테이블의 크기를 증가**
  - D = b → new D = 2b + 1
  - 확장성 해싱(extendible hashing)이라고도 불림

- **원래의 해시테이블의 값을 새로운 해시테이블로 재조정이 필요**
  - 원래 해시테이블의 엔트리를 새로운 해시테이블로 단순 복사해서는 안됨
  - 재조정을 한번 할 때마다 오직 하나의 버킷 안에 있는 엔트리들에 대해서만 홈 버킷을 변경하게 하여 재조정 시간을 줄임

A = 100
B = 101
C = 110

0 = 000
1 = 001
2 = 010
3 = 011
4 = 100
5 = 101

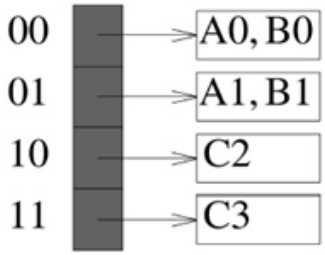
k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

해시 함수

디렉토리를 사용하는 동적 해싱
(Dynamic Hashing using Directories)

- 버킷들에 대한 포인터를 저장하고 있는 디렉터리 d를 이용
- 디렉터리의 크기 : h(k)의 비트 수에 좌우됨
- 디렉터리 깊이(directory depth)
  - 디렉터리를 인덱싱하는 h(k)의 비트 수
  - t : 디렉터리 깊이, 2^t : 디렉터리 크기
    - (ex) h(k,2)를 사용하여 인덱싱 → 디렉터리의 크기 = $2^2 = 4$

      h(k,5) 를 사용하여 인덱싱 → 디렉터리의 크기 = $2^5 = 32$
  - 버킷 수 ≤ 디렉터리 크기

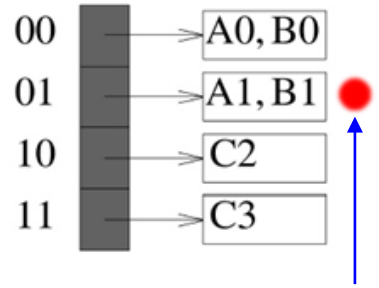


directory depth = 2

디렉토리를 사용하는 동적 해싱
(Dynamic Hashing using Directories)

- A0, B0, A1, B1, C2, C3을 포함하고 있는 동적 해시 테이블

directory depth = 2



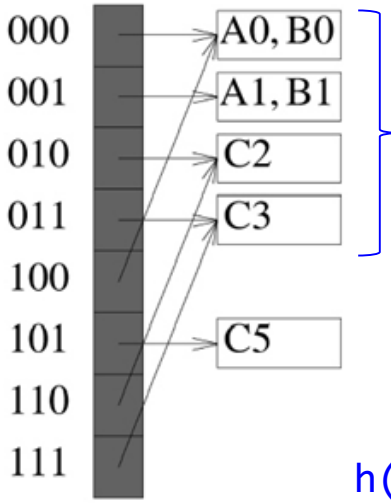
collision!
overflow

h(C5, 2) = 01

A = 100
B = 101
C = 110

0 = 000
1 = 001
2 = 010
3 = 011
4 = 100
5 = 101

directory depth = 3



버켓 복사 x
링크 조정

C5가 A1, B1과 다른 h(k, u)를 갖게 하는 u = 3
A1[:3] = 001
B1[:3] = 001
C5[:3] = 101

doubling directory

h(C1, 3) = 001

C1이 A1, B1과 다른 h(k, u)를 갖게 하는 u = 4
A1[:4] = 0001
B1[:4] = 1001
C1[:4] = 0001

directory depth = 4