

## Chap7 HASHING

- 1) Hashing 정의
- 2) Hashing Function (해쉬 함수)
- 3) Overflow handling
- 4) Hashing algorithm: Insert, delete, find, etc.

### 1. 정의

- 응용분야: 예) DBMS 의 Data dictionary, Word processor 의 spelling checker, **Symbol Tables** in Loaders, Assemblers, Compilers.

- **Symbol Table**

- 1) 이름-값 (name-attribute)으로 이루어진 쌍의 집합.  
(예: compiler: name -> 변수명, attribute-> 초기값 및 그 변수를 사용하는 line list 등 정보를 포함)
- 2) Operations on any symbol table  
특정 이름의 존재여부, 그 이름의 속성 검색 및 변경, 새 이름의 값 과 속성 삽입, 삭제등

- **Symbol Table 의 표현 시 고려할 점**

- searching, inserting, deleting 을 효과적으로 해야 함.
- ⇒ HASHING 은 searching, inserting, deleting 을 효과적으로 할 수 있다.
- ⇒ 대부분의 검색기법은 키값의 비교에 의존하는 반면, Hashing 은 특수한 Hashing Function에 의존한다.
- ⇒ Hashing 구성방법: 배열(hash table), 이진 트리(BST)

## 1.1 Hashing

특정 키를 검색하기 위해 일련의 비교를 수행 하는 대신, Hashing 은 키 k에 대하여 임의의 함수 F를 적용하여 k의 주소나 색인을 계산하여 비교 절차 없이 직접 검색하는 방법이다.

(적용함수 F를 -> **hashing function**

계산된 주소 -> **hash address** / home address )

- **Hash Table:**

Symbol table 을 메모리 상에서 유지할 때 이를 Hash Table 이라 한다. (즉, 변수가 저장되는 장소)

. Hashing function 'F'를 사용하여 어떤 변수의 hash table 내 주소/장소를 결정한다.

- Hash Table 은 b 개의 bucket 으로 구성된다.

(예: ht[0], ht[1],...ht[b])

- 1 개의 bucket 은 s 개의 slot 으로 구성되고, 1 개의 slot 은 1 개의 record 를 저장할 수 있다.

- **hash function**,  $f(x)$  는 변수를 mapping 하여 정수값으로 변환한다. (변환된 정수값은  $0 \sim (b-1)$  이다).

- **Identifier (변수) density**  $\Rightarrow n/T$ ,

$n$  = number of identifiers in the HT,

$T$  = possible values for identifier

**Loading density/loading factor**  $\Rightarrow \alpha = n/(sb)$

( $s$  = number of slot,  $b$  = bucket size)

\*  $f(k_1) = f(k_2)$  인 경우,  $k_1$  과  $k_2$  는 동의어라 하고  $k_1$  과  $k_2$  는 동일한 bucket 에 저장되어야 한다.

- **Overflow** - 해당 bucket 이 한개의 slot 으로 구성될때, 이미  $k_1$  이 저장되어 있을때,  $k_2$  를 저장하면, overflow 발생  
(when we HASH a new identifier into a FULL bucket)
- **Collision** - 서로 다른 2 개의 키 (ex:  $k_1, k_2$ ) 가 동일한 bucket 으로 hash 되는 것을 collision 이라 하며, slot =1 일 경우는 collision and overflow 이다.  
(When bucket size is 1, collision and overflow occurs simultaneously)

ex) HT with bucket = 26, slot =2,  
n = 10 identifiers (GA, D,A,G ,L,A2,A1,A3,A4,E)

⇒ loading factor =  $10/(26 \times 2) = 10/52 = 0.19$

⇒ HASH function => associate letters a-z with the numbers 0-25 즉,  **$f(x) = \text{first character of } x$**

0	1	2	3	4	5	6	.....	...	25
A1			D	E		GA		L	
A2						G			

A3

A4 => overflow 발생

- Hashing function 은 계산이 간편하고, 모든 입력에 대하여 Hash Table 에 균등하게 분포하여야 함
- ⇒ **uniform hashing function** (random 하게 input x 를 선택 했을 때, 임의의 b bucket 에 동일하게 hashing 될 수 있다).  
그러나 함수 이름들은 같은 character 로 시작하는 경우가 많기 때문에 collision 을 피하기 어렵다.

## 2. HASHING FUCTIONS

### 1) Division (나눗셈)

키 값을 특정소수(prime number)로 나누어서 나머지 값을 address 로 한다.

Ex)  $H(K) = K \bmod M$  : produce 0 ~ (M-1) address  
 $H(357) = 357 \bmod 31 = 16$        $H(124) = 124 \bmod 31 = 0$

- ⇒ M 의 선택이 중요 (소수이용)  
 (예: M 이 짝수이면, K 가 짝수일때, H(K) 도 짝수,  
       M 이 짝수이면, K 가 홀수일때, H(K) 도 홀수  
 이므로, 충돌 가능성이 많다)

if M=25,       $52 = H(52) = 2 \Rightarrow HT[2]$       충돌  
                   $77 = H(77) = 2 \Rightarrow HT[2]$

- ⇒ 20 이상되는 소수(prime number) 선택 권고.

## 2) Mid-Square (중간제곱법)

키 값을 제곱하여 얻어진 수의 중간 위치값을 추출하여 (중간에서 적절히 몇개의 bit 선택) bucket 주소로 한다.

예)

key K	$K^2$	address
327	10 <u>69</u> 29	69
184	03 <u>38</u> 56	38
185	39 <u>94</u> 24	94

## 3) Folding (접지법)

Key 를 같은 길이의 여러 부분으로 나눈다. 나눈 부분의 각 숫자를 더하여 그 결과치를 address 로 이용한다.

Ex) 3 8 4 4 2 2 2 4 1

83	=> reverse
142	=> reverse
+ 4422	=> no reverse
4647	=> 자릿수 변경도 가능

key	address
384422241	4647

## 4) Digit Analysis (숫자분석법)

불필요/중복부분 삭제 후 address 선택

ex) 384-42-2241=>그대로사용시, bucket 이 10 억개 필요.

384-81-3678 분석 => 384 는 동일하므로 버림.

6,7,9 column 은 균일-> 선택.

Key	address
384-42- <u>2241</u>	221
384-81- <u>3678</u>	368

### 3. Overflow handling

- **Linear open addressing / Linear Probing**
- **Chaining**

#### 3.1 Linear Open Addressing (Linear Probing)

⇒ Collision 발생시에 Table search 해서 비어있는 가장 가까운 bucket 을 찾아 그곳에 저장하는 방법

예)

bucket	x	bucket searched
0	acos	1
1	atoi	2
2	char	1
3	define	1
4	exp	1
5	ceil	4
6	cos	5
..		
25		

ex)  $32 \bmod 13 \Rightarrow 6$   
 $19 \bmod 13 \Rightarrow 6$

0	
..	
6	32
7	19

⇒ List 의 끝에 도달하면, 처음으로 되돌아가서 빈 영역 search

\* 특징: 삭제가 처리가 어렵다 (부가적 flag 이용)  
Clustering 현상 발생 (탐색시간 길어짐)

## < Variations of Linear Probing >

- **Quadratic Probing** (이차조사법)

⇒ (Reduce average number of probing and curtail the growth of these clusters)

· 선형 조사법:  $(f(k) + I) \bmod b$

· 이차 조사법:  $(f(k) + i^2) \bmod b$  or  $(f(k) - i^2) \bmod b$

- **Random Probing**

$f(k) + S(I)$  :  $s(i)$  is random number

- **Rehashing** (여러개의 해쉬함수를 적용함)

$f_i(K)$ 로 overflow 발생시  $\rightarrow f_{i+1}(k)$ 로 계산  $\rightarrow$  overflow  
 $\rightarrow f_{i+2}(K)$ 로 계산  $\rightarrow \dots$

⇒ Clustering 문제를 해소하기 위해 Overflow 발생시 Linear Probing 에 series of hash function ( $f_1, f_2, \dots, f_b$ ) 을 적용하는 기법

- **Linked Method** (연결방법)

⇒ 기억장소를 prime/overflow 영역으로 구분,

⇒ 각 record 는 key, data, link 로 구성.

⇒ 처음엔 prime 에 할당, 충돌시에는 overflow 영역에 저장

Ex)

$$h(x) = (k \bmod 7) + 1$$

$$22 \bmod 7 + 1 \Rightarrow 2$$

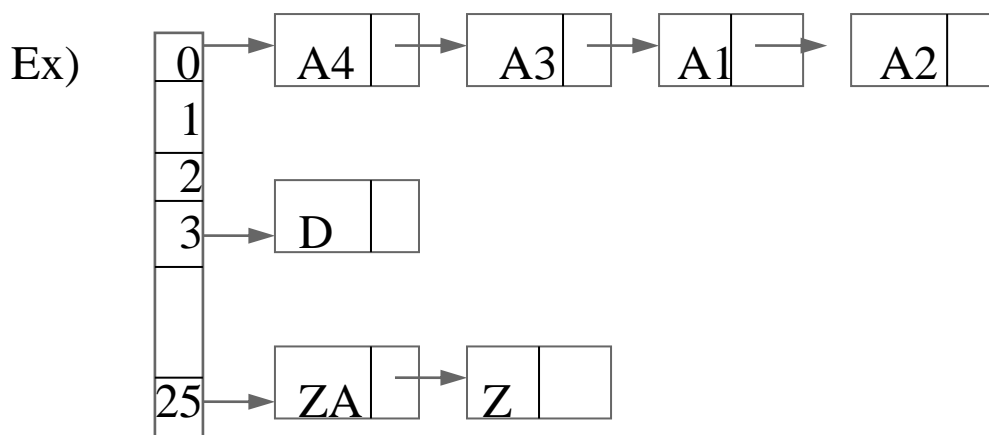
$$36 \bmod 7 + 1 \Rightarrow 2 \Rightarrow 7$$

$$29 \bmod 7 + 1 \Rightarrow 2 \Rightarrow 8$$

	Key	Data	Link	
0	0		nil	prime area
1	0		nil	
2	22		7	
	<u>space</u>			
7	36		8	overflow area
8	29		nil	

### 3.2 Chaining

- ⇒ linear probing 은 삽입 시 다른 값 들과의 비교를 해야 한다. (불필요한 비교도 해야 함)
- ⇒ chaining 은 삽입시 단순히 해쉬 함수값만 계산하고 그 리스트에 있는 변수들을 조사하면 된다.
- ⇒ Chaining 은 Linked List 구조를 가진다.  
즉, 노드당 key field 와 link field 가 필요하고, 또한 n 개의 리스트를 위한 Headnode 필요.





## **4. HASHING Algorithm**

### **1) Data Structure**

```
typedef struct
{ int key;    int empty; }
hashtable;
hashtable Htable[MaxSize];
```

### **2) ADT 함수**

```
int  findKey(key)           // static HASHing
{ .  index  HASH(key)  // hashing function
  . if (key = Htable(key)) found = true
    Found = false
  Return found
}
```

```
int  findKey(key)  // Linear Probing 으로 확장
{
  while ( 1. Check hashtale is empty for the Key &&
          2. Check key is not Found yet           &&
          3. Finish Checking entire hashtable)
  {
    if (key is in Hashtable) then found = true
    else
      . increase index by 1
      . if (finish checking entire Htable) then  loop=true

  return found
}
```

```

int insertkey(key)
{
    check = findkey(Key)

    if (check = true)    return false // no duplicated key

    if (Htable is occupied) return false

    else { . Save key in Hable
           . Mark the slot is occupied
           . return true
        }
    }
}

```

```

int deletekey(key)
{
    check = findkey(Key)
    if (check = false)    return false    //can't find
    else { . mark the slot in Htable
           . return true
        }
    }
}

```

```

void printtable()
{.....}

}

```

● **HASH function**     // if hashing function is **DIVISION**

```

int hash(int key)   { return   key%MaxSize;   }

```