Chap7 HASHING

1) Hashing 정의
2) Hashing Function (해쉬함수)
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 Function 에 의존한다.

⇨ Hashing 구성방법: 배열, Linked List

1.1 Hashing

특정 키를 검색하기 위해 일련의 비교를 수행 하는 대신, Hashing 은 **키 k** 에 대하여 임의의 **함수 H** 를 적용하여 k 의 **주소나 색인을** 계산하여 비교 절차 없이 직접 검색하는 방법.

(적용함수 F 를 -> **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 을 사용하여 어떤 변수의 hash table 내 주소/장소를 결정한다.

h(key) = index → Hash Table[index]

- Hash Table 은 b 개의 bucket 으로 구성된다.

(예: ht[0], ht[1],...ht[b-1])

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

Ex)hash table

b[0]	b[1]	b[2]	….	b[n-1]

- **hash function**, h(x) 는 변수를 mapping 하여 정수값으로 변환한다. (변환된 정수값은 0 ~ (n-1) 이다).

- **Identifier (변수) density(밀도)** => n/T,

n = number of identifiers in the HT,
T = possible values for identifier

Loading density(적재밀도)/loading factor

$\alpha = n/(sb)$: (s = number of slot, b = bucket size)

* h(k1) = h(k2) 인 경우, k1 과 k2 는 동의어라 하고 k1 과 k2 는 동일한 bucket 에 저장되어야 한다.

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

⇨ loading factor = 10/(26 x 2) = 10/52 = 0.19

⇨ HASH function => associate letters a-z with the numbers 0-25 즉, **h(x) = first character of x**

b[0]	b[1]	b[2]	b[3]	b[4]	b[5]	b[6]	.....	b[25]
A1 A2			D	E	F	GA G		

A3

A4 => overflow 발생

- **Overflow** - 해당 bucket 이 한개의 slot 으로 구성될때, 이미 k1 이 저장되어 있을때, k2 를 저장하면, overflow 발생 (when we HASH a new identifier into a FULL bucket)

- **Collision** – 서로 다른 2 개의 키 (ex: k1, k2) 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)

2. HASHING FUCTIONS

- Key 를 hash table 내의 bucket 으로 mapping
Hashing function 은 계산이 간편하고, 모든 입력에 대하여 Hash Table 에 균등하게 분포하여야 함

*** uniform hashing function** (균일 해쉬함수)
(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.

1) Division (나눗셈) - 가장 많이 사용되는 해쉬함수

키 값을 특정소수로 나누어서 나머지 값을 address 로 한다.

H(K) = K mod M : produce 0 ~ (M-1) address
최소 M 개의 bucket 필요

ex) H(357) = 357 mod 31 = 16 H(124) = 124 mod 31 = 0

⇨ M 의 선택이 중요 (소수이용)
if M=25, 52 = H(52) = 2 => HT[2] 충돌
77 = H(77) = 2 => HT[2]

⇨ M 을 20 이상되는 **소수(prime number)** 로 선택 권고.

2) Mid-Square (중간 제곱법)

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

예)

key K	K^2	address
327	10**69**29	69
184	03**38**56	38
185	39**94**24	94

3) Folding (접지법)

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

● Shift folding (이동접지)

. 마지막을 제외한 모든 부분을 이동

. 최 하위 비트가 마지막 부분의 자리와 일치

Ex) k = 12320324111220, 길이가 세 자리로 나눈다.

P1 = 123, P2 = 203, P3 = 241, P4 = 112, P5 = 20

h(k) = 123 + 203 + 241 + 112 + 20 = 699

● 자릿수 변경도 가능

P2 와 P4 를 역순으로 302 와 211 을 각각 구한 후
각 부분을 더하여 h(k) = 123 + 302 + 241 + 211 + 20 = 897

4) Digit Analysis (숫자 분석법)

불필요/중복부분 삭제 후 address 선택

ex) 384-42-2241=>그대로사용시, bucket 이 10 억개 필요.
384-81-3678 분석결과 => 384 는 동일하므로 버림.
6,7,9 column 은 균일-> 선택.

Key	address
384-42-2241	221
384-81-3678	368

● 기타 (안전 해시함수)

- For message authentication (메시지 인증)
- 보안 해시알고리즘 (SHA: Secure Hash Algorithm)
  미국 국립표준기술연구소 개발

3. Overflow handling

- **Open addressing (개방 주소법)**
  Linear probing(선형 조사법), quadratic probing, double hashing, rehashing, random probing.
- **Chaining (체인법)**

3.1 Open Addressing (Linear Probing)

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

예)

bucket	key	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 mod 13 => 6
19 mod 13 => 6

0	
..	
6	32
7	19

⇨ List 의 끝에 도달하면, 처음으로 되돌아가서 빈 영역 search

* 특징: 삭제시 처리가 어렵다
- 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: (h(k) + i) mod b

. **이차 조사법: (h(k) + i^2) mod b**

i 번째 해쉬함수를 h(x)로부터 i^{*2} 만큼 떨어진 자리로 insert. 즉, h(x), h(x)+1, h(x)+4, h(x)+9,…

Ex) $h_i(x) = ((h(x)+ i^2) \bmod 13$, Input: 15, 18, 43, 37, 45, 30

0	1	2	3	4	5	6	7	8	9	10	11	12
		15		**43**	18	45		30			37	

h(15) =2, h(18) = 5, h(43) = 4, h(37) = 11, h(45) = 6,
h(30) = 4+4 = 8

Ex) $h_i(x) = ((h(x)+ i^2) \bmod 7$ input: 76, 40, 5, 55

0	1	2	3	4	5	6
		5	55		40	76

h(76) =6 h(40)= 5
h(5) =5+4=9%7=2 h(55) = 6+4=10%7=3

- **Rehashing** (여러개의 해쉬함수를 적용함)

$h_i(K)$로 overflow 발생시 → $h_{i+1}(k)$로 계산 → overflow → $h_{i+2}(K)$로 계산 → .....

⇨ Clustering 문제를 해소하기 위해 Overflow 발생시 Linear Probing 에 series of hash function (h_1,h_2,...h_n) 을 적용하는 기법

● Double Hashing

Double hashing 은 두개의 함수 사용.

$h_i(x) = (h(x) + i*f(x)) \bmod m$

// h(x)와 f(x)는 서로다른 함수, 충돌이 생겨 다음 주소를 계산할 때 두 번째 해쉬 함수값 만큼씩 점프를 한다.

ex) 입력 데이터: 15, 19, 28, 41, 67

h(x) = x mod 13, f(x)= (x mod 11)+1

hi(x) = (h(x) + if(x)) mod 13

0	1	2	3	4	5	6	7	8	9	10	11	12
		15		**67**		19			**28**		**41**	

$h_0(15) =$ **$h_0(28) = h_0(41) = h_0(67)$** $= 2$, $h_0(19) = 19\%13 = 6$

$h_1(28) = 2+ ((28\%11)+1 =7) =9$

$h_1(41)= 2+ ((41\%11)+1=9) =11$ **$h_1(67) = (2+((67\%11)+1=2) = 4$**

● Linked Method (연결방법)

⇨ 기억장소를 prime/overflow 영역으로 구분,

⇨ 각 record 는 key, data, link 로 구성.

⇨ 처음엔 prime 에 할당, 충돌시에는 overflow 영역에 저장

h(x)=(k mod 7)+1

22 mod 7 +1 = 2

36 mod 7 +1 = 2

⇨ Overflow area

⇨ 7

29 mod 7 +1 =2

⇨ Overflow area

⇨ 8

	Key	Link	Prime area
0			
1			
2	22	7	
3			
4			
			Overflow area
7	36	8	
8	29	NIL	
9			

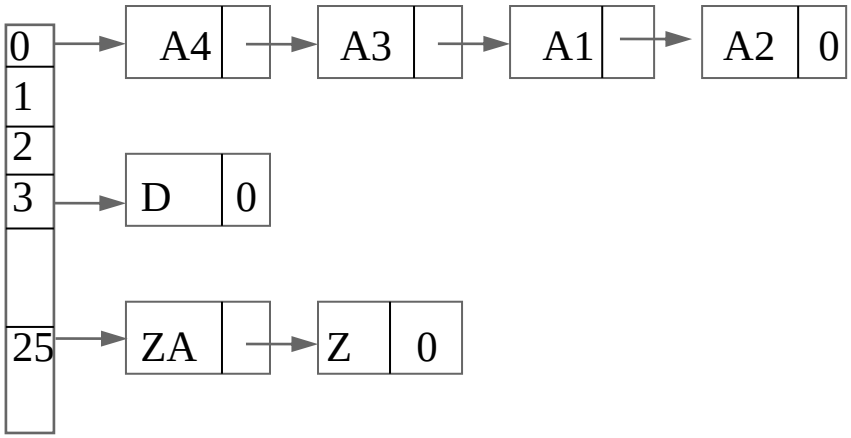
3.2 **Chaining**

⇨ 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.

Ex)



4. HASHING Algorithm

● 기본 연산

- create table, - search - Insert, Delete

1) Data Structure

```
class Node {
private:
        int data;
        Node *link;
        friend class Htable;
};

class Htable {
private:
        Node * head;
public:
        int findkey(int);
        int insertkey(int);
        int deletekey(int);
        void printable();
};
```

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)
{
- Get index value for Key        // index = Hash(key);
- Get head node from HashTable    // p = HT[index];
- If (p = NULL)
                return false;
else {                                  // q= p;
   Search the table for the Key // while ((q!=null)&&(q->data!=key))
                                        //      q= q->link
   If (q = null) return false;
   else return true
  }

int insertkey(key)
{    index = hash(key)                   // get HT address
     check = findkey(Key)               // check if ht key exists
     if (check = true)    return false    // no duplicated key can insert
     create a new node q;

     if (HT[index] == NULL)     HT[index] = q;      // make head node
     else {
          get head node from HT[index];        // p = head
          while (p->link!= NULL) p = p->link; // move to the end
          p->link = q;      // insert new node
  }

void printtable()   {
 for (i= 0; i < maxsize;    i++)    {
       print "HT [i]"
       Get head node for "i"
       for (head; head!= NULL; head= head->link)
             Print "node
  }
```

```
int deletekey(key)
{
   check = findkey(Key)
   if (check = false)    return false        //can't find
   else {
      . get index for the key
      . get head node for the key
      . if (head = key) delete head node &
            move head= head->next;      // update head node
      . else
          find node and delete the node for the key
                  // same as singly linked list
   }
}
```

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

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