Chap8 HASHING

● 목차

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) Loaders, Assemblers, Compiler 에서 사용.
2) 이름-값 (name-attribute)으로 이루어진 쌍의 집합.
   (예: compiler: name -> 변수명,
   attribute-> 초기값 및 그 변수를 사용하는 line list 등 정보를 포함)
3) operations on any symbol table

■ 특정 이름의 존재여부, 그 이름의 속성 검색 및 변경, 새 이름의 값 과 속성 삽입, 새 이름의 값 과 속성 삭제

- 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(hash table), 이진 트리(BST))

1.1 Hashing

● 개요: 특정 키를 검색하기 위해 일련의 비교를 수행 하는 대신, Hashing 은 키 k 에 대하여 임의의 함수 F 를 적용하여 k 의 주소나 색인을 계산하여 비교 절차 없이 직접 검색하는 방법이다.

(적용함수 F 를 -> **hashing function**
계산된 주소 -> **hash address** / home address)

- **Hash Table:**

Symbol table 을 순차적으로 메모리 상에서 유지할 때 이를 Hash Table 이라 한다. (즉, 변수가 저장되는 장소) 그리고 함수 F 를 사용하여 어떤 변수 x 의 hash table 내의 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 ~ (b-1) 이다).

- **Identifier (변수) density** => n/T,
n = number of identifiers in the HT,
T =possible values for identifier

Loading density/loading factor => $\alpha = n/(sb)$
(s = number of slot, b = bucket size)

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

- **Overflow** - 해당 bucket 이 slot1 이고, 이미 k1 이 저장되어 있을때, k2 를 저장하려고 하면, 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, s=1 일 경우는 collision and overflow 이다.
(When bucket size is 1, collision and overflow occurs simultaneously) slot=2일때 collision이지만 overflow는 아니다

ex) HT with bucket = 26, slot =2,
n = 10 identifiers (GA, D,A,G ,L,A2,A1,A3,A4,E)
A무시 잘못나옴
⇨ loading factor = 10/(26 x 2) = 10/52 = 0.19 <– 이거 무조건 시험문제!!!!

⇨ HASH function => associate letters a-z with the numbers 0-25 즉, f(x) = 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 (easy to compute) 모든 입력에 대하여 HT 에 균등하게 분포하여야 함 (모든 bucket I 에 대하여 f(k) = I 가 될 확율이 1/b 가 되는 것이다.

⇨ 만약, 임의의 키 K 가 b 개의 bucket 에 동일한 확률로 mapping 된다면, 이에 적용되는 함수는 uniform hashing function 이다 (즉, random 하게 input x 를 선택 했을 때, any of b bucket 에 equally hashing 될 수 있다).

⇨ 그러나 함수 이름들은 같은 character 로 시작하는 경우가 많고, loading factor 가 일반적으로 작기 때문에 collision 을 피할 수 없다)

2. HASHING FUCTIONS

(mid-square, division, folding, digit analysis)

1) 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**69**29	69
184	03**38**56	38
2)	39**94**24	94

2) Division (나눗셈) 디비전 약간 중요

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

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

⇨ M 의 선택이 중요 (bucket 의 size 보다 작게, 소수 이용)

(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 => HT[2] 충돌
77 =H(77) = 2 =>HT[2]

⇨ 따라서, 20 이상되는 소수를 선택하기를 권고한다.

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	4142
“	4647

ex) Key 를 여러자리수로 나눈다.

123 203 241 112 20

123
203
241
112
+ 20
} => 699

4) Digit Analysis (숫자분석법) 그닥 중요 x

키를 분석하여 불필요 부분/중복부분 삭제 후 address 선택

ex)

384-42-2241=>그대로 사용할 경우, bucket 이 10 억개

384-81-3678 이상필요=> 낭비, 속도는 빠름

384-38-4569

분석 => 384 는 동일하므로 discard

6,7,9 column 은 분포가 균일하므로 선택.

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

3. Overflow handling

- 2 methods to detect Overflow and Collision

1) Linear open addressing / Linear Probing
2) Chaining

선형 물어보기?

1) Linear Open Addressing

⇨ 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 mod 13 => 6
19 mod 13 => 6
23 mode 13 => 10

0	
..	
6	32
7	19

⇨ List 의 끝에 도달하면, 처음으로 되돌아가서 빈 영역 search

* 특징: 삭제시 처리가 어렵다 (부가적 flag 이용, 표시?) Clustering 현상 발생 (탐색시간 길어짐)

무리, 송이

< 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 $(f(k) - i^2) \bmod b$, $(1 \le I \le (b-1)/2)$

- **Random Probing**

$f(k) + S(I)$ 이건잘안씀

random number

- **Rehashing**

$f_i(K)$로 overflow 발생시 -> $f_{i+1}(k)$로 계산 -> overflow -> $f_{i+2}(K)$로 계산 ->.....

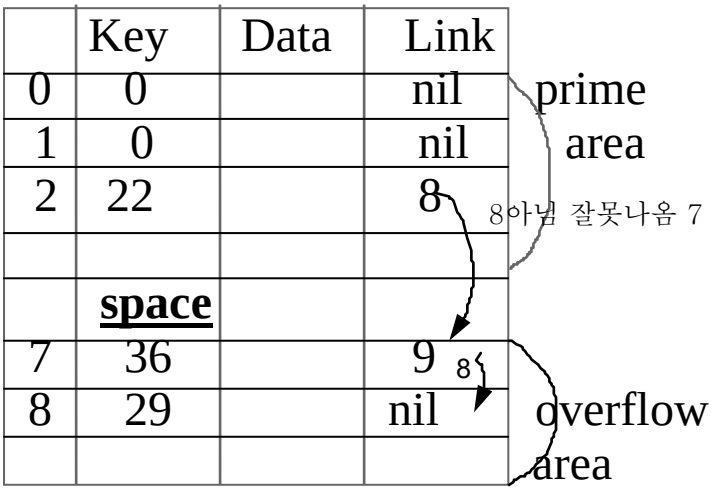
⇨ Clustering 문제를 해소하기 위해 Overflow 발생시 Linear Probing에 series of hash function (f_1,f_2,...f_b) 을 적용하는 기법

- **Linked Method (연결방법)**

⇨ 기억장소를 prime/overflow 영역으로 구분, 각 record 는 key, data, link 로 구성. 처음엔 prime 에 할당, 충돌시에는 overflow 영역에 삽입.

Ex)
h(x) = (k mod 7) + 1
22 mod 7 + 1 => 2
36 mod 7 + 1 => 2=>7
29 mod 7 + 1 => 2=>8

1더하는건 내맘



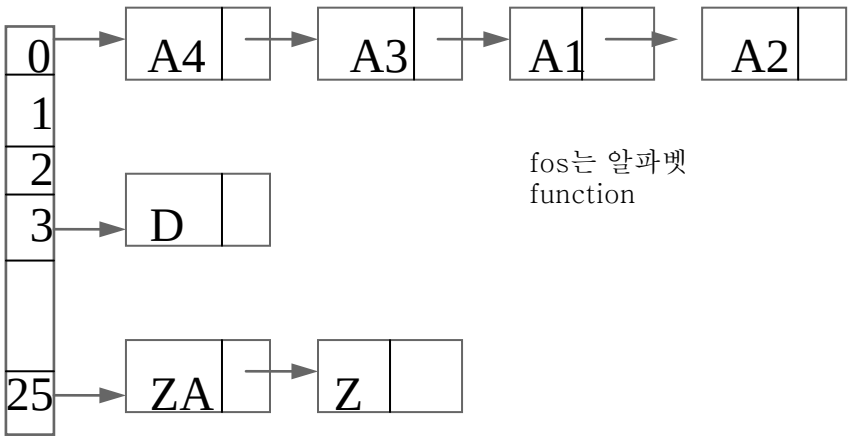
2) Chaining

⇨ linear probing 은 삽입 시 다른 값 들과의 비교를 해야 한다. (불필요한 비교도 해야 함)

⇨ 만약, 각 bucket 에 synonym list 를 유지한다면, 삽입시 단순히 해쉬 함수값만 계산하고 그 리스트에 있는 변수들을 조사하면 된다. 이 세줄 중요 x

⇨ Chaining 은 Linked List 구조를 가진다.
즉, 노드당 key field 와 link field 가 필요하고, 또한 n 개의 리스트를 위한 Headnode (Link field 만 필요) 필요.

Ex)



fos는 알파벳 function

4. HASHING Algorithm

1) Data Structure

```
typedef struct {
        int key;      int empty;        }hashtable;
hashtable htable[13];
```

2) Main Algorithm

```
do { Enter command (i,f,d,q):    read(ch);
    if (ch != 'q') {
       print("Enter key ");
       read(inkey);

       switch (ch){
          case 'i':    check= insertkey(inkey);
                       if (check==false)
                          print("Cannot insert key");    break;

          case 'f':    check= findkey(inkey, index);
                       if (check==false)
                            print("key not found");        break;

          case 'd':    check=deletekey(inkey);
                       if (check==false)
                            print("key not found ");    break;
                       default : print("Bad Command");
          }
       }
       printtable();
  }while (ch == 'q' );
}
```

3) ADT 함수

- **findkey**

```
int    findKey(key, index)
{    index= HASH(key)
     if (HasTable[index].key == key) then    found = true
          else found = false
     return found
}
```

- **insertKey**

```
int insertkey(key)
{    check = findkey(HasTable, Key, index)
     if    (check==true)    return false
     else    (HashTable[index].key = key
               HashTable[index].empty = false}
      return true
  }
```

- **deleteKey**

```
int deletekey(key)
{    check = findkey(HasTable, Key, index)
     if (check == false)    return false
     else { HasTable[index].empty = true
            return true}
}
```

- **HASH function**

```
int hash(int key)    {    //hashing function is DIVISION
  return      key%MaxtableSize;
}
```