CHAPTER 8

Hashing

Contents

- ADT dictionary

ADT *Dictionary* is
objects: a collection of $n > 0$ pairs, each pair has a key and an associated item
functions:
for all $d \in$ *Dictionary*, $item \in$ *Item*, $k \in$ *Key*, $n \in$ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(d, n)	::=	**if** ($n > 0$) **return** *FALSE* **else return** *TRUE*
Element Search(d, k)	::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k)	::=	delete and return item (if any) with key k;
void Insert(d, *item*, k)	::=	insert *item* with key k into d.

ADT 5.3: Abstract data type *dictionary*

사전

- Applications of dictionaries
  - Spelling checker
  - Data dictionary
  - Symbol tables

- Representation of dictionary
  - Binary search tree (chapter 5)
  - Balanced BST (chapter 10)
- Operations: search, insert, delete
  - O(n) time: for a BST → skewed
  - O(log n) time using a balanced BST
- Hashing
  - A technique that performs the dictionary operations search, insert and delete in O(1) expected time
  - Static hashing, dynamic hashing

8.2 STATIC HASHING

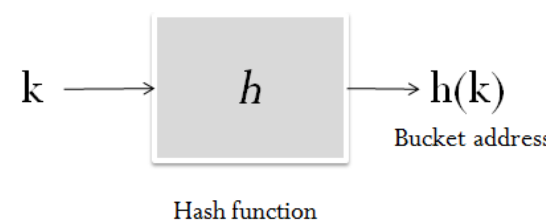
8.2.1 Hash Tables

- Dictionary pairs
  - Stored in a table *ht*, called the **hash table**
- Hash table
  - Partitioned into ***b*** **buckets**: *ht*[0], …, *ht*[b-1]
  - Each bucket consists of has ***s*** **slots**
    - Each slot holds one dictionary pair
  - The address or location of a pair whose key is *k* is determined by a hash function, *h*

hash function에 key를 넣으면
hash table의 address가 나옴

- Hash function
  - $h(k)$: integer in the range 0 through b-1
    - hash or home address of key *k*

      ↳ 테이블의 index

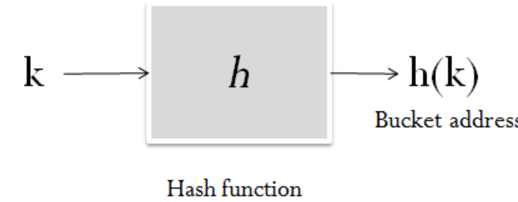


- Definition:
  - The key density of a hash table: n/T
    - n: # of pairs in the table   table에 있는 key의 개수
    - T: The total # of possible keys   모든 가능한 key의 개수

    용량
  - The loading density (factor) of a hash table: $\alpha = n/(sb)$

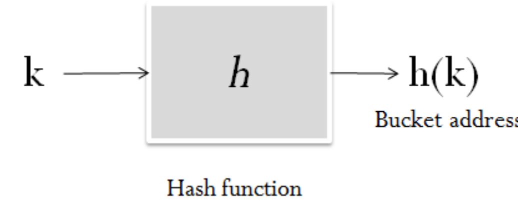
    데이터의 개수
    - s : # of slots
    - b : # of buckets

→ 전체 버킷에서 사용하고있는 버킷의 개수



- Suppose that keys are at most six characters long
  - The first character: a letter
  - The remaining characters: letters or digits
  - The # of possible keys : $T = \Sigma_{i=0}^{5}\ 26 \times 36^{i} > 1.6 \times 10^{9}$
  - But, most applications use only very small fraction of it;
  - Key density n/T is usually very small
- The number of of buckets b is also much less than T
  - Hash function h maps several different keys into the same bucket → $h(k1) = h(k2)$
- If $h(k_1) = h(k_2)$
  - Two keys k_1 and k_2 are said to be *synonyms*

key 값은 다른데, 해쉬함수의 값은같음
이때 두 key를 synonyms 라고 함



- Overflow
  - Home bucket is full at the time we wish to insert a new pair into the dictionary
- Collision

bucket 이 이미 찼을때..

  - Occurs when the home bucket for a new pair is not empty at the time of insertion
- If each bucket has 1 slot
  - collisions and overflows occur at the same time

- Ex 8.1) Hash table

(key)
데이터 10개

  - b=26 buckets and s=2 slots, distinct identifiers n=10
    - Loading factor α = 10 / 52 = 0.19
    - distinct identifiers: ‘acos’, ‘define’, ‘float’, ‘exp’, ‘char’, ‘atan’, ‘ceil’, ‘floor’, ‘clock’, ‘ctime’
  - Define a hash function, ***h*(*x*)**, as the first character of *x*
    - *h*(“acos”) = ‘a’
    - Associate the letters, a-z, with the numbers, 0-25, respectively
  - acos and atan, float and floor, ceil and char
    - synonyms
  - *h*(“clock”)
    - overflow

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

Figure 8.1: Hash table with 26 buckets and two slots per bucket

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

Figure 8.1: Hash table with 26 buckets and two slots per bucket

- The time complexity of insert, delete or search if no overflow occurs : …

  $O(1)$

- But, overflows occur for most cases

- Hashing Schemes

  어떤 hash function을 쓸것인가

  – Use a hash function to map keys into hash-table buckets

  – Desirable a hash function to use that is both easy to compute and minimizes the number of collisions

  hash function을 잘 구현하자

  – A mechanism to handle overflows is needed

  overflow를 잘 해결하자

8.2.2 Hash Functions

- Hash function : $h(k) = i$
  - Maps a key k into a bucket i in the hash table

- The desired properties
  - Easy to compute
  - Minimize the # of collisions
  - Unbiased;
  - **Uniform** hashing function
    - Probability of $h(k) = i$ is $1/b$ for all buckets i
    - k: a key chosen at random from the key space

해쉬 함수에 값은 넣으면 출력값이 골고루 나옴

- Popular Hashing Functions
    - Division
    - Mid-square
    - Folding
    - Digit Analysis

Hash Fuctions : Division

- The most widely used hash function in practice
- Home bucket $h(k) = k \% \boxed{D}$ bucket의 개수
  - k: nonnegative, D: some number
  - Bucket : $0 \sim D$-1
  - Hash table must have at least $D(=b)$ buckets

- The choice of *D* is critical
  - If *D* is divisible by 2, then odd (even) keys are mapped to odd (even) buckets; biased
    - 20%14 = 6, 30%14 = 2, 8%14 = 8
    - 15%14 = 1, 3%14 = 3, 23%14 = 9
  - The distribution is biased whenever *D* has small prime factors 2,3,5,7, ..

    unbiased 한 $h(k)$를 내기위함
  - Ideally, choose *D* so that it is a prime number
  - Alternatively, choose *D* so that it has no prime factor smaller than 20

- Is it practical? …

- The relaxed requirement on *D*
  - Use odd *D* and set *b* equal to *D*
  - As the size of the dictionary grows, it will be necessary to increase the size of the hash table *ht* dynamically;
  - *Array doubling* results in increasing the # of buckets (and hence divisor D) from *b* to 2*b* +1

Hash Fuctions : Mid-Square

- $h(k)=middle(k^2)$
  - Square the key and then use an appropriate # of bits from the middle of the square
  - The middle bits of the square usually depend upon all the characters in an identifier
    - Different identifiers will produce different hash addresses
  - If r bits are used, then the size of hash tables
    - …

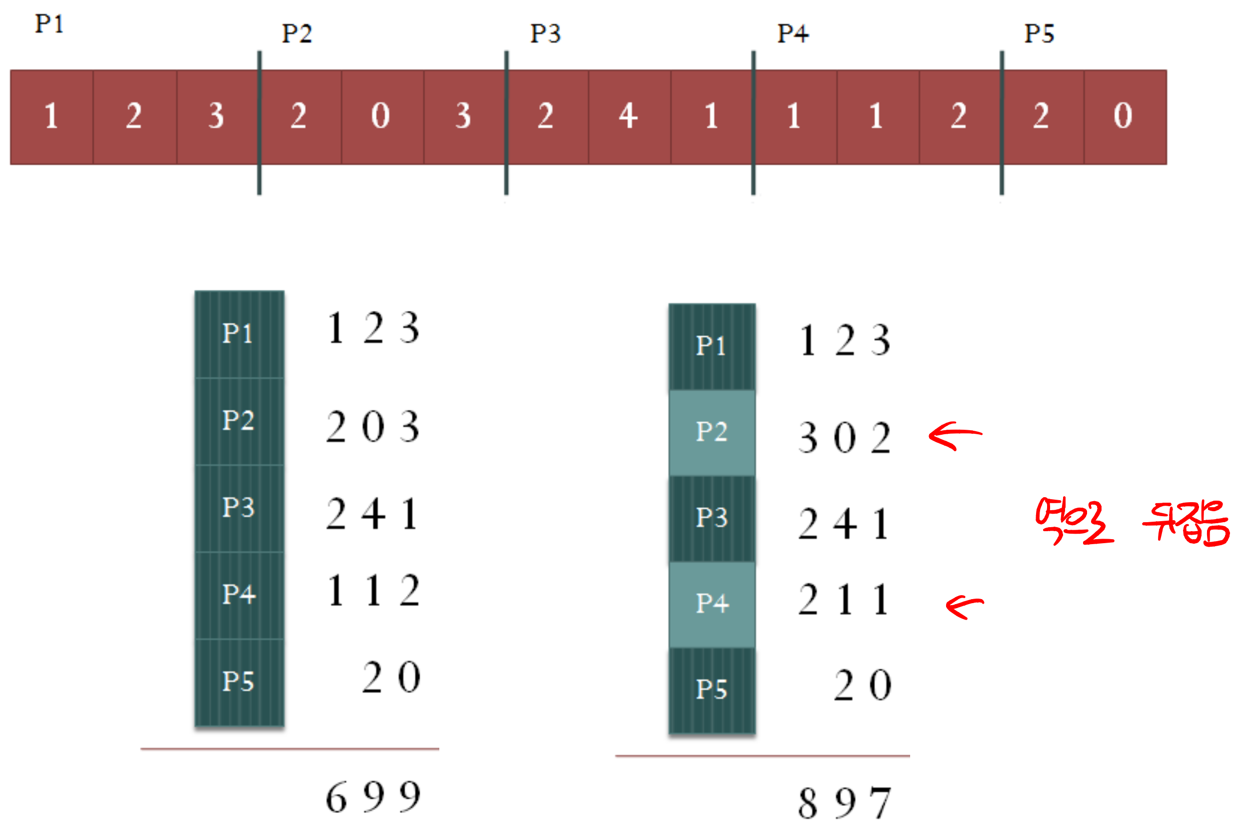
```
      10100     20
                 ×
      10100     20
-----------------
  00110010000   400
```

Hash Fuctions : Folding

- The key *k*
  - **Partitioned** into several parts, all but possibly the last being of the same length
  - Then **added** together to obtain the hash address for *k*

- Two schemes
  - Shift folding
  - Folding at the boundaries

- Ex 8.2) k= 12320324111220
  - partition it into parts that are three decimal digits long



Shift folding:
$h(k)$=699

Folding at the boundaries:
$h(k)$=897

Hash Fuctions : Digit Analysis

- Useful in a static file
  - where all the keys in the table are known in advance

- Method
  - Each key is interpreted as a number using some radix *r*
  - The digits of each key are examined
  - Digits having the most skewed distributions are deleted
    - remaining digits: an address of the hash table

542-42-2241
542-81-3678
542-22-8171
542-38-9671
542-54-1577
542-88-5376
542-19-3552

422, 836, 281,
396, 515, 853,
135.

고른 분포를 나타내는 곳을
찾아서 배열 주소로 활용
(index)

Converting Keys to Integers

- Hash keys
  - Need to be converted to nonnegative integer

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

key: 8 characters
→ integer up to 11 bits long

```
unsigned int stringToInt(char *key)
{/* alternative additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
   {
      number += *key++;
      if (*key) number += ((int) *key++) << 8;
   }
   return number;
}
```

Program 8.2: Alternative way to convert a string into a non-negative integer

Hashing

- 8.1 Introduction
- 8.2 Static Hashing
  - Hash Tables
  - Hashing Functions
  - Overflow Handling
    - Open addressing : Linear probing , Quadratic probing, Rehashing, Random probing
    - Chaining
- 8.3 Dynamic Hashing

Overflow Handling

↳ 이미 bucket이 차있어

- Linear Probing
  - Inserting a new pair (k)
    - Search the hash table buckets in the order, $ht[h(k) + i] \% b$, where $0 \le i \le b$-1

      해쉬한 결과값

      ht[(h(k)+i) % b]

      ↳ bucket의 사이즈

      ↳ hash table의 size
    - This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket

      번 bucket을 만나면 종료

    끝까지 못 찾으면?
    - In case no such bucket is found: hash table is full 완전 꽉찼지..
      - It is necessary to increase the table size → size를 늘리자
      - Table size is increased when the loading density exceeds a pre-specified threshold (ex: 0.75)
  - Resizing the hash table
    - We must change the hash function; ⇒ hash table을 바꾼다는 것은 hash function을 바꾼다는 것
    - All dictionary entries need to be remapped into the new larger table

- Ex 8.4)
  - 13-bucket table with one slot per bucket
    - Using $h(k) = k \% D$ 13 : Division
  - words: for, do, while, if, else, function

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	$\boxed{12}$

아스키 코드값

Overflow

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

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a n
    that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

Using a circular rotation, the next available bucket is at *ht*[0] 결국에 0이 비어있었네..

- Hash Table Search
  - when s = 1 and linear probing is used to handle overflows

(1) Compute $h(k)$.

(2) Examine the hash table buckets in the order $ht[h(k)]$, $ht[(h(k)+1)\ \%\ b]$, $\cdots$, $ht[(h(k)+j)\ \%\ b]$ until one of the following happens:

  (a) The bucket $ht[(h(k)+j)\ \%\ b]$ has a pair whose key is k; in this case, the desired pair has been found.

  (b) $ht[h(k)\quad]$ is empty; k is not in the table.

  (c) We return to the starting position $ht[h(k)]$; the table is full and k is not in the table.

870 (function)

```
element* search(int k)
{/* search the linear probing hash table ht (each bucket has exactly one slot)  for k,
 if a pair with key k is found, return a pointer to this pair;
 otherwise, return NULL */
     int homeBucket, currentBucket;
     homeBucket = h(k);   12
     for(currentBucket = homeBucket; ht[currentBucket]
                    && ht[currentBucket]->key != k;) {
        currentBucket = (currentBucket + 1) % b;
                        /* treat the table as circular */
        if(currentBucket == homeBucket)
             return NULL;  /* back to start point */
     }
     if(ht[currentBucket]->key == k)
        return ht[currentBucket];
     return NULL;
}
```

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

Program 8.3: Linear probing

- Linear probing
  - Keys tend to cluster together

- Suppose input sequence:
  - acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime
- Hash function
  - $h(x)$: the first character of x
- When we try to enter "atol"
  - …

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		

Figure 8.4: Hash table with linear probing (26 buckets, one slot per bucket)

- Input : acos, atoi, char, define, exp, ceil, cos, float, atol, floor, ctime
- Average # of key comparisons = 41/11= 3.73
- Keys tend to cluster together
  - Increase the search time
- Improvements …

Enter:

bucket	x	buckets searched
0	[redacted]	
1	[redacted]	
2	[redacted]	
3	[redacted]	
4	[redacted]	
5	[redacted]	
6	[redacted]	
7	[redacted]	
8	[redacted]	
9	[redacted]	
10	[redacted]	
. . .		
25		

Fig 8.4: Hash table with linear probing
(26 buckets, 1 slot/bucket)

- If using linear probing + **uniform hash function**
  - The expected average number of key comparisons $p = (2-\alpha)/(2-2\alpha)$
    - α: Loading density
  - In Fig 8.4:

    $\alpha=11/26 = 0.42$

    $p = 1.36$
  - The worst-case number of comparisons: $O(n)$

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		

- Quadratic Probing
  - Search $h(k)$, $(h(k) + i^2)$ % b, $(h(k) - i^2)$ % b
    - For $1 \le i \le (b - 1)/2$
  - b: a prime number of the form $4j+3$, where j is an integer
    → every buckets are examined

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Figure 8.5: Some primes of the form $4j + 3$

- Rehashing
  - Use a series of hash functions $h_1, h_2, \ldots, h_m$
  - Buckets $h_i(k)$, $1 <= i <= m$ are examined in that order

- Random Probing
  - Search for a key k by examining the buckets in the order $h(k)$, $(h(k)+\mathbf{s(i)}) \% b$, $1 \le i \le b-1$
    - $s(i)$: a pseudo random number

Hashing

- 8.1 Introduction
- 8.2 Static Hashing
  - Hash Tables
  - Hashing Functions
  - Overflow Handling
    - Open addressing : Linear probing , Quadratic probing, Rehashing, Random probing
    - Chaining
- 8.3 Dynamic Hashing