Chap 8. Hashing

Hash Table *ht*

(Hash) Bucket (버켓)

Bucket size, 1 slot

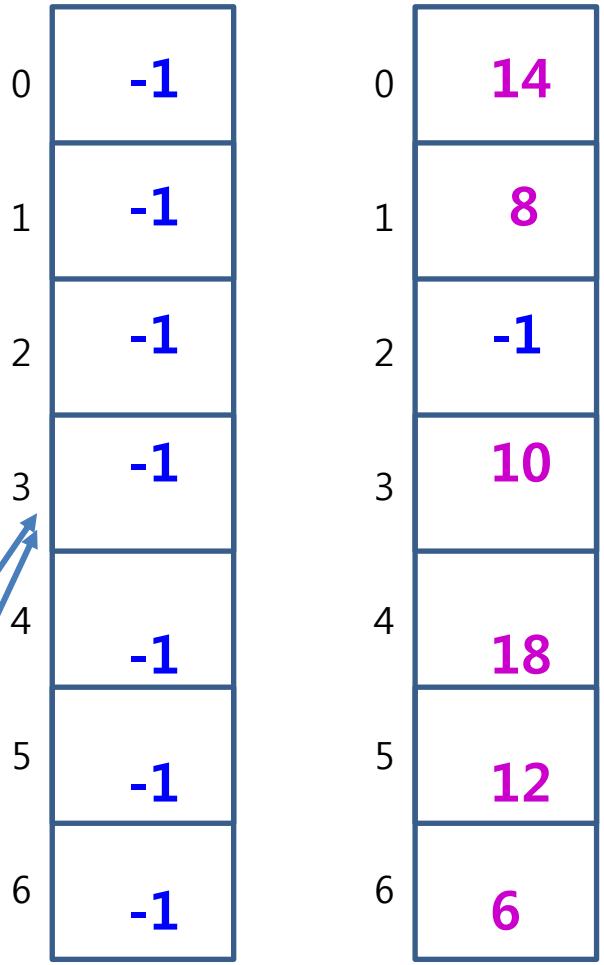
k

h(k) = k mod 7

hash function
(해쉬 함수)

Home Bucket

Hash Address
(Home Address)

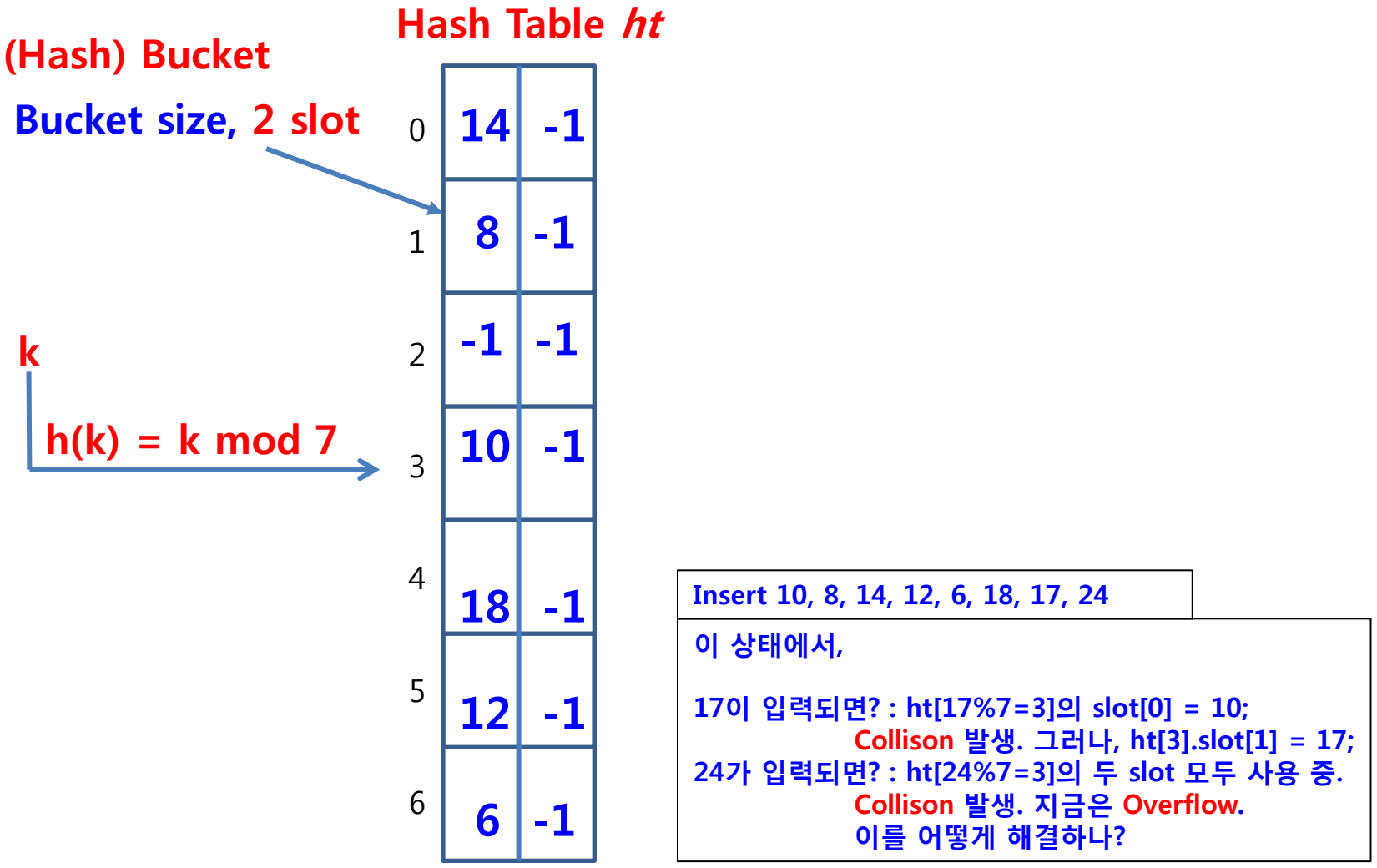


초기화 상태

Insert 10, 8, 14, 12, 6, 18, 17, 24

초기화 상태에서,
10이 입력되면? : ht[10%7 = 3] = 10;
8이 입력되면? : ht[8%7=1] = 8;
14가 입력되면? : ht[14%7 = 0] = 14;
12가 입력되면? : ht[12%7=5] = 12;
6이 입력되면? : ht[6%7=6] = 6;
18이 입력되면? : ht[18%7=4] = 18;

17이 입력되면? : ht[17%7=3]
에 이미 다른 key가 존재한다.
So, Collison 발생. 지금 이는 Overflow 이다.
이를 어떻게 해결하나?

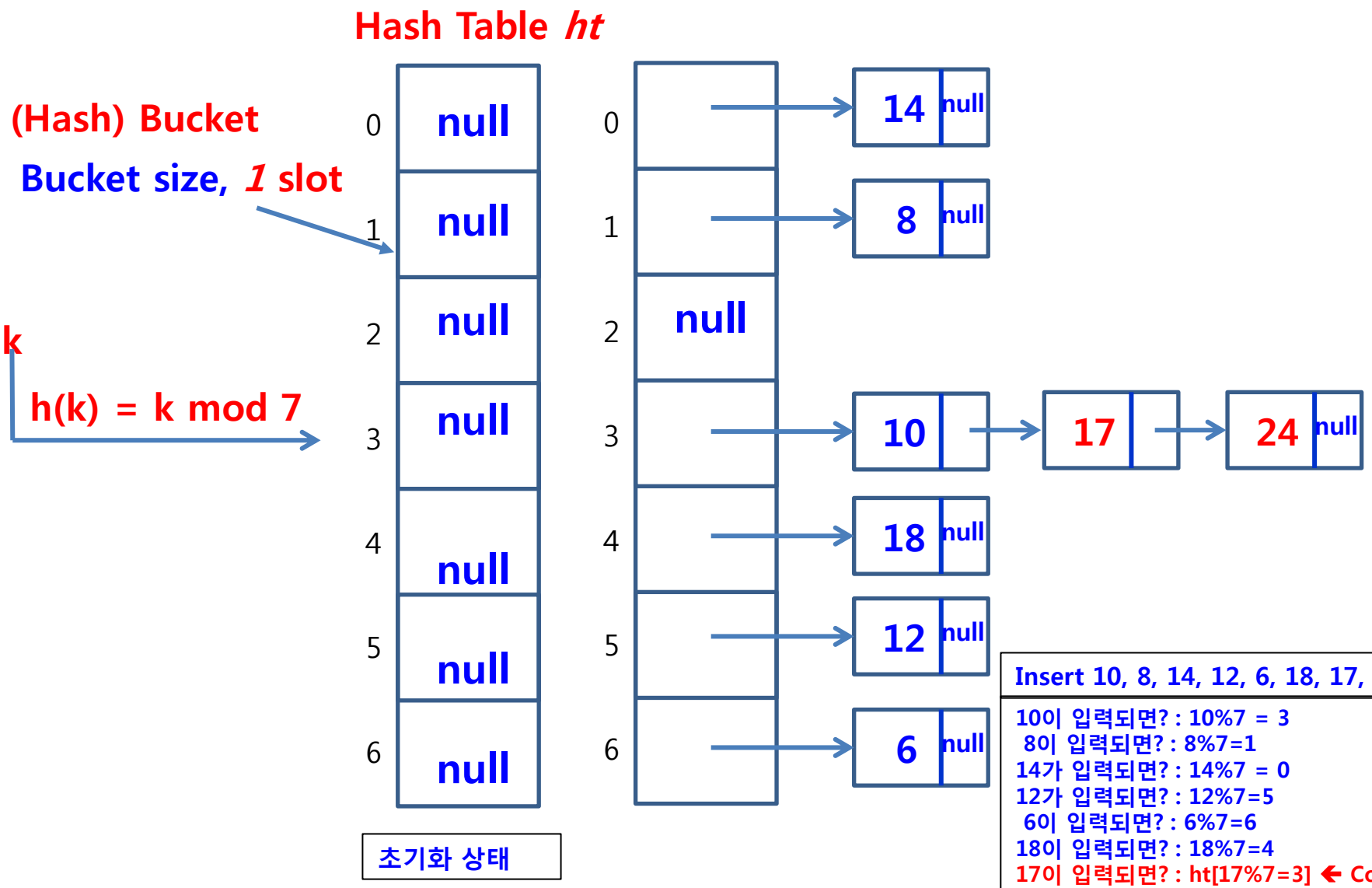


Insert 10, 8, 14, 12, 6, 18, 17, 24

이 상태에서,

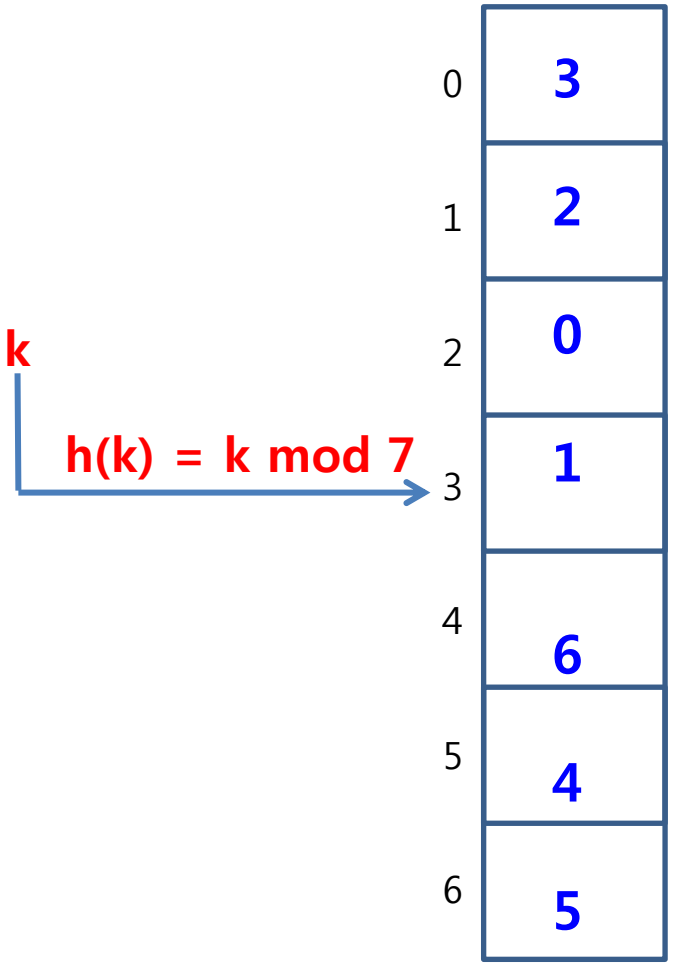
17이 입력되면? : ht[17%7=3]의 slot[0] = 10;
Collison 발생. 그러나, ht[3].slot[1] = 17;
24가 입력되면? : ht[24%7=3]의 두 slot 모두 사용 중.
Collison 발생. 지금은 Overflow.
이를 어떻게 해결하나?

Overflow Chaining
(오버플로우 연결법)



Overflow Chaining

Hash Table *ht*



Overflow Table

FreeList = 9

0		
1	10	7
2	8	0
3	14	0
4	12	0
5	6	0
6	18	0
7	17	8
8	24	0
9		10
10		11
11		0

초기화 상태에서,
10이 입력되면? : 10%7 = 3
8이 입력되면? : 8%7=1
14가 입력되면? : 14%7 = 0
12가 입력되면? : 12%7=5
6이 입력되면? : 6%7=6
18이 입력되면? : 18%7=4
17이 입력되면? : ht[17%7=3] ← Collison
24가 입력되면? : ht[24%7=3] ← Collison

- ***key density*** of a hash table = ***n/T***,
  where ***n*** is the number of pairs in the table and
  T is the total number of possible keys.
- ***loading density*** or ***loading factor*** of a hash table is
  $\alpha = n / (sb)$.

- U_n: ***n*** 개의 키를 가진 hash table에서 그 테이블에 없는 key를 찾을 경우, 평균 몇 번 비교해야 하는가?
- S_n: ***n*** 개의 키를 가진 hash table에서 그 테이블에 있는 key를 찾을 경우, 평균 몇 번 비교해야 하는가?
- When chaining is used along with a uniform hash function, For Overflow Chaining, Un ≈ α, Sn ≈ 1 + α/2

k

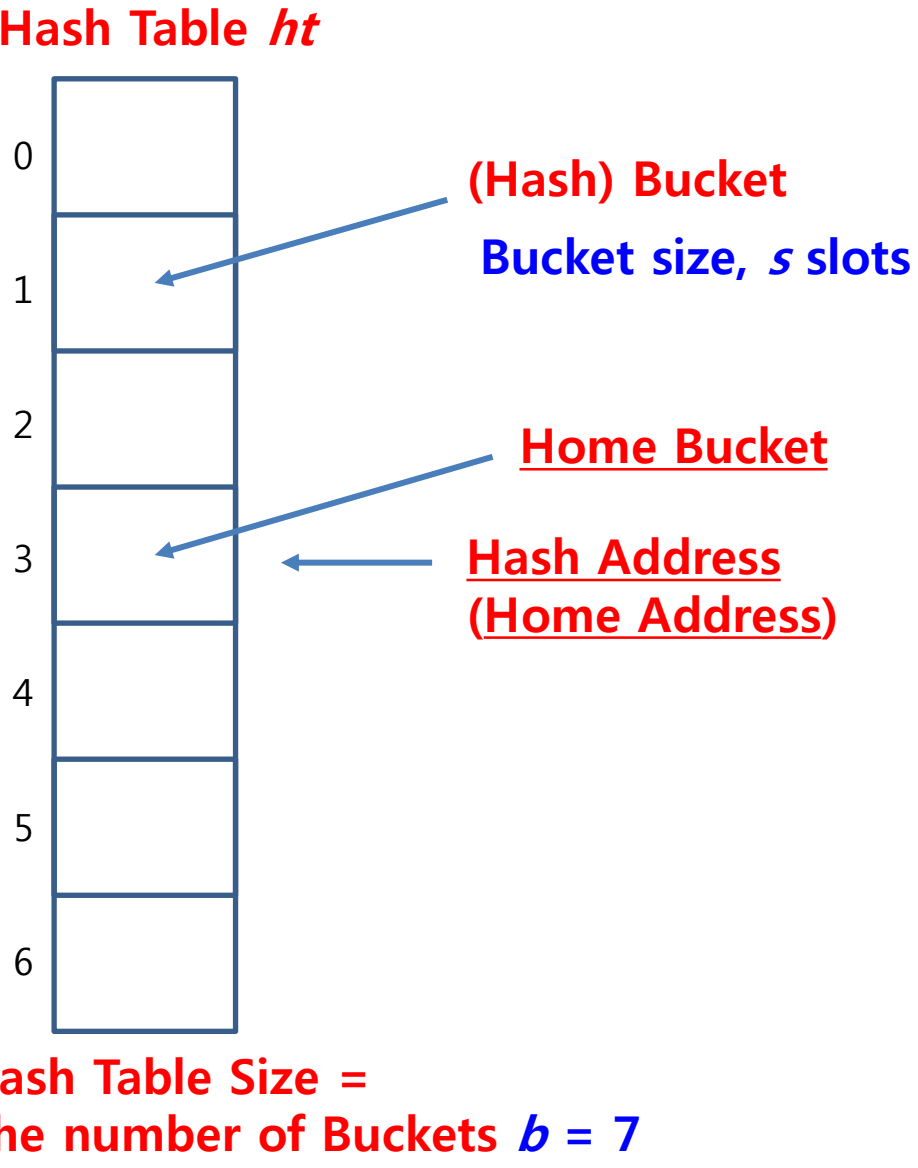
h(k) = k mod D

hash function

- ← Uniform hash function?
- ← b = 7 = D
- ← 7 is a prime number.
- ← *D* should be odd to avoid the bias caused by an even D.
- ← Array Doubling : b → 2b + 1

The hash function *h* maps several different keys into the same bucket.

- ✓ *Synonym*
- ✓ *Collision*
- ✓ *Overflow*
- ✓ *Overflow Handling*



Contents

8.1 Introduction

In this chapter, we again consider the ADT **dictionary** that was introduced in Chapter 5 (ADT 5.3). Examples of dictionaries are found in many applications, including the spelling checker, the thesaurus, the index for a database, and the symbol tables generated by loaders, assemblers, and compilers. When a dictionary with n entries is represented as **a binary search tree** as in Chapter 5, the dictionary operations ***search***, ***insert*** and ***delete*** take O(n) time. These dictionary operations may be performed in O(log n) time using **a balanced binary search tree** (Chapter 10). In this chapter, we examine a technique, called **hashing**, that enables us to perform the dictionary operations ***search***, ***insert*** and ***delete*** in **O(1)** expected time. We divide our discussion of hashing into two parts: ***static hashing*** and ***dynamic hashing***.

8.2 Static Hashing

8.2.1 Hash Tables

In ***static hashing***, the dictionary pairs are stored in a table *ht*, called the ***hash table***. The hash table is partitioned into ***b buckets***, *ht*[0], …, *ht*[*b*-1]. Each bucket is capable of holdings ***s*** dictionary pairs (or pointers to this many pairs). Thus, a bucket is said to consist of ***s slots***, each slot being large enough to hold one dictionary pair. **Usually *s* = 1**, and each bucket can hold exactly one pair. The address or location of a pair whose key is *k* is determined by a ***hash function, h,*** which maps keys into buckets. Thus, for any key *k*, ***h*(*k*)** is an integer in the range 0 through *b*- 1. *h*(*k*) is the ***hash*** or ***home address*** of *k*. Under ideal conditions, dictionary pairs are stored in their ***home buckets***.

Definition: The ***key density*** of a hash table is the ratio ***n/T***, where *n* is the number of pairs in the table and *T* is the total number of possible keys. The ***loading density*** or ***loading factor*** of a hash table is $\alpha = n / (sb)$. □

Suppose our keys are at most six characters long, where a character may be a decimal digit or an uppercase letter, and that the first character is a letter. Then the number of possible keys is $T = \Sigma_{i=0}^{5} 26 \times 36^i > 1.6 \times 10^9$. Any reasonable application, however, uses only a very small fraction of these. So, the key density, n/T, is usually very small. Consequently, the number of buckets, b, which is usually of the same magnitude as the number of keys, in the hash table is also much less than T. Therefore, the hash function h maps several different keys into the same bucket. Two keys, k_1 and k_2, are said to be ***synonyms*** with respect to h if $h(k_1) = h(k_2)$.

As indicated earlier, under ideal conditions, dictionary pairs are stored in their ***home buckets***. Since many keys typically have the same home bucket, it is possible that the home bucket for a new dictionary pair is full at the time we wish to insert this pair into the dictionary. When this situation arises, we say that an ***overflow*** has occurred. A ***collision*** occurs when the home bucket for the new pair is not empty at the time of insertion. When each bucket has 1 slot (i.e., $s = 1$), collisions and overflows occur simultaneously.

Example 8.1: Consider the hash table *ht* with $b = 26$ buckets and $s = 2$. We have $n = 10$ distinct identifiers, each representing a C library function. This table has **a loading factor, α, of 10/52 = 0.19.** The hash function must map each of the possible identifiers onto one of the numbers, 0-25. We can construct a fairly simple hash function by associating the letters, *a*-*z*, with the numbers, 0-25, respectively, and then defining the hash function, $f(x)$, as the first character of x. Using this scheme, the library functions **acos**, **define**, **float**, **exp**, **char**, **atan**, **ceil**, **floor**, **clock**, and **ctime** hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively. Figure 8.1 shows the first 8 identifiers entered into the hash table.

The identifiers **acos** and **atan** are **synonyms**, as are **float** and **floor**, and **ceil** and **char**. The next identifier, **clock**, hashes into the bucket *ht*[2]. Since this bucket is full, we have an **overflow**. Where in the table should we place **clock** so that we may retrieve it when necessary? We consider various solutions to the overflow problem in Section 8.2.3. □

k

h(k) = int(the first character of k – 'a')

ht

	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

- Insert, delete, find
  - O(s) if no overflow occurs
  - O(1) if no collision occurs
- But, collision occurs for most cases, since the ratio b/T is usually very small.

- Hash Table Issues
  - Choice of hash function.
  - Overflow handling method.
  - Size (number of buckets) of hash table.

8.2.2 Hash Functions

A **hash function** maps a key into a bucket in the hash table. As mentioned earlier, the desired properties of such a function are that it be easy to compute and that it minimize the number of collisions. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs; that is, if k is a key chosen at random from the key space, then we want the probability that $h(k) = i$ to be $1/b$ for all buckets i. With this stipulation, a random key has an equal chance of hashing into any of the buckets. A hash function satisfying this property is called ***a uniform hash function***.

Several kinds of uniform hash functions are in use in practice. Some of these compute the home bucket by performing arithmetic (e.g., multiplication and division) on the key. Since, in many applications, the data type of the key is not one for which arithmetic operations are defined (e.g., string), it is necessary to first convert the key into an integer (say) and then perform arithmetic on the obtained integer. In the following subsections, we describe four popular hash functions as well as ways to convert strings into integers.

8.2.2.1 Division

This hash function, which is the most widely used hash function in practice, assumes the keys are non-negative integers. The home bucket is obtained by using **the modulo (%) operator**. The key k is divided by some number D, and the remainder is used as the home bucket for k. More formally,

$$h(k) = k \% D$$

- This function gives bucket addresses in the range 0 through D - 1, so the hash table must have at least $b = D$ buckets. Although for most key spaces, every choice of D makes h **a uniform hash function**, the number of overflows on real-world dictionaries is critically dependent on the choice of D.
- **If D is divisible by two, then odd keys are mapped to odd buckets (as the remainder is odd), and even keys are mapped to even buckets.**
- Since real-world dictionaries tend to have a bias toward either odd or even keys, the use of an even divisor D results in a corresponding bias in the distribution of home buckets.
- In practice, it has been found that for real-world dictionaries, the distribution of home buckets is biased whenever D has small prime factors such as 2, 3, 5, 7 and so on. However, the degree of bias decreases as the smallest prime factor of D increases. **Hence, for best performance over a variety of dictionaries, you should select D so that it is *a prime number*. With this selection, the smallest prime factor of D is D itself.** For most practical dictionaries, a very uniform distribution of keys to buckets is seen even when we choose D such that it has no prime factor smaller than 20.
- When you write hash table functions for general use, the size of the dictionary to be accommodated in the hash table is not known. This makes it impractical to choose D as suggested above.
- So, we relax the requirement on D even further and **require only that D be odd to avoid the bias caused by an even D**. In addition, **we set b equal to the divisor D.**
- As the size of the dictionary grows, it will be necessary to increase the size of the hash table *ht* dynamically. **To satisfy the relaxed requirement on D, array doubling results in increasing the number of buckets (and hence the divisor D) from b to $2b + 1$.**

8.2.2.2 Mid-Square

The **mid-square hash function** determines the home bucket for a key by **squaring the key and then using an appropriate number of bits from the middle of the square** to obtain the bucket address; the key is assumed to be an integer. Since the middle bits of the square usually depend on all bits of the key, different keys are expected to result in different hash addresses with high probability, even when some of the digits are the same. The number of bits to be used to obtain the bucket address depends on the table size. If r bits are used, the range of values is 0 through $2^r - 1$. So the size of hash tables is chosen to be a power of two when the mid-square function is used.

ex)

```
        10100
   X    10100
----------------
00110010000
```

5 bits are used.

```
      00
    10100
+   10100
----------------
 00110010000
```

8.2.2.3 Folding

In this method the key k is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for k. There are two ways of carrying out this addition.

In the first, all but the last partition are shifted to the right so that the least significant digit of each lines up with the corresponding digit of the last partition. The different partitions are now added together to get $h(k)$. This method is known as ***shift folding***.

In the second method, ***folding at the boundaries***, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain $h(k)$. This is equivalent to reversing every other partition and then adding.

Example 8.2: Suppose that k = 123203241112 20, and we partition it into parts that are three decimal digits long. The partitions are $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$, and $P_5 = 20$.

Using **shift folding**, we obtain

$$h(k) = P_1 + P_2 + P_3 + P_4 + P_5 = 123 + 203 + 241 + 112 + 20 = 699$$

When **folding at the boundaries** is used, we first reverse P_2 and P_4 to obtain 302 and 211, respectively. Next, the five partitions are added to obtain

$$h(k) = P_1 + \text{reverse}(P_2) + P_3 + \text{reverse}(P_4) + P_5 = 123 + 302 + 241 + 211 + 20 = 897.$$

8.2.2.4 Digit Analysis

This method is particularly useful in the case of a static file where all the keys in the table are known in advance. Each key is interpreted as a number using some **radix *r***. The same radix is used for all the keys in the table. Using this radix, the digits of each key are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

Digit Analysis(숫자 분석 또는 Digit Extraction(숫자 추출)) 방법

키 값을 구성하는 각 digit의 분포를 이용.

키들의 모든 자릿수에 대한 빈도 테이블을 만들고, 어느 정도 균등한 분포를 갖는 자릿수를 주소로 사용.

- 예) 주민등록번호의 경우 (YyMmDd에서, y, m, d는 Y, M, D에 비해 균등한 성질을 가짐)
- 키 값을 구성하는 각 digit의 분포를 미리 알고 있을 경우에 유용

예제: 키 값의 9th, 7th, 5th, 3rd 자리로 주소를 구성

- **h(123456789) = 9753**
- **h(987654321) = 1357**
- **h(000000472) = 2400**

8.2.2.5 Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to nonnegative integers. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers. It is ok for us to convert the strings data, structures, and algorithms into the same integer (say, 199). In this section, we consider only the conversion of strings into non-negative integers. Similar methods may be used to convert other data types into non-negative integers to which the described hash functions may be applied.

Example 8.3: [Converting String to Integers]

Since it is not necessary to convert strings into unique nonnegative integers, we can map every string, no matter how long, into an integer. Programs 8.1 and 8.2 show you two ways to do this.

```
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

Program 8.1 converts each character into a unique integer and sums these unique integers. Since each character maps to an integer in the range 0 through 255, the integer returned by the function is not much more than 8 bits long. For example, strings that are eight characters long would produce integers 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

Program 8.2 shifts the integer corresponding to every other character by 8 bits and then sums. This results in a larger range for the integer returned by the function. □

8.2.3 Overflow Handling

8.2.3.1 Open Addressing

There are two popular ways to handle overflows: ***open addressing*** and ***chaining***. In this section, we describe four open addressing methods - ***linear probing***, which also is known as ***linear open addressing***, ***quadratic probing***, ***rehashing*** and ***random probing***.

In linear probing, when inserting a new pair whose key is k, we search the hash table buckets in the order, $ht\ [h(k) + i]\ \%\ b$, $0 \leq i \leq b - 1$, where h is the hash function and b is the number of buckets. This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket. In case no such bucket is found, the hash table is full and it is necessary to increase the table size.

In practice, to ensure good performance, table size is increased when **the loading density** exceeds a pre-specified threshold such as **0.75** rather than when the table is full. Notice that when we resize the hash table, we must change the hash function as well. For example, when the division hash function is used, the divisor equals the number of buckets. This change in the hash function potentially changes the home bucket for each key in the hash table. So, all dictionary entries need to be remapped into the new larger table.

Example 8.4: Assume we have a 13-bucket table with one slot per bucket. As our data we use the words **for**, **do**, **while**, **if**, **else**, and **function**. Figure 8.2 shows the hash value for each word using the simplified scheme of Program 8.1 and the division hash function. Inserting the first five words into the table poses no problem since they have different hash addresses. However, the last identifier, **function**, hashes to the same bucket as **if**. Using a circular rotation, the next available bucket is at *ht*[0], which is where we place *function* (Figure 8.3).

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

Figure 8.2: Additive transformation

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

Figure 8.3: Hash table with linear probing (13 buckets, one slot per bucket)

Linear probing

When $s = 1$ and linear probing is used to handle overflows, a hash table search for the pair with key k proceeds as follows:

(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) $ht[h(k)+j]$ 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.

Program 8.3 is the resulting search function. This function assumes that the hash table *ht* stores pointers to dictionary pairs. The data type of a dictionary pair is *element* and data of this type has two components *item* and *key*.

```
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);
    for (currentBucket = homeBucket; ht[currentBucket]
                                  && ht[currentBucket]->key != k;) {
        currentBucket = (currentBucket + 1) % b; /* treat the table as circular */
        if (currentBucket == homeBuket)          /* back to start point */
            return NULL;
    }
    if (ht[currentBucket)->key == k)
        return ht[currentBucket];
    return NULL;
}
```

Program 8.3: Linear probing

When linear probing is used to resolve overflows, keys tend to cluster together.

Example :

- Input sequence: **acos, atoi, char, define, exp, ceil, cos, float, atol, floor,** and **ctime**.
- For illustrative purposes, we assume that the hash function uses the first character in each function name.
- What happens when we try to enter "atol" ?
- If we retrieved each of the identifiers in *ht* exactly once, the average number of buckets examined would be 41/11 = 3.73 per identifier.

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)

- With uniform hash function, the expected average number of key comparisons *p* is (2-α)/(2-2α), where α is the loading density.
  - Figure 8.4 : α=11/26 = 0.42, *p* =1.36
- Even though the average number of comparisons is small, the worst case can be quite large. The worst-case number of comparisons: O(n)

Some **improvement** in the growth of clusters and hence in the average number of comparisons needed for searching can be obtained by ***quadratic probing***. Linear probing was characterized by searching the buckets $(h(k) + i) \% b$, $0 \leq i \leq b - 1$, where b is the number of buckets in the table.

In quadratic probing, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets $h(k)$, $(h(k) + i^2) \% b$, and $(h(k) - i^2) \% b$ for $1 \leq i \leq (b - 1)/2$. When b is a prime number of the form $4j + 3$, for j an integer, the quadratic search described above examines every bucket in the table. Figure 8.5 lists some primes of the form $4j + 3$.

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 $4j + 3$

An alternative method to retard the growth of clusters is to use a series of hash functions $h_1, h_2, \ldots, h_m$. This method is known as ***rehashing***. Buckets $h_i(k)$ $(1 \leq i \leq m)$ are examined in that order.

Yet another alternative, random probing, is explored in the exercises.

8.2.3.2 Chaining

Linear probing and its variations perform poorly because the search for a key involves comparison with keys that have different hash values. In the hash table of Figure 8.4, for instance, searching for the key **atol** involves comparisons with the buckets *ht* [0] through *ht* [8], even though only the keys in *ht* [0] and *ht* [1] had a **collision** with **atol**; the remainder cannot possibly be **atol**.

Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address $h(k)$ and examining only those keys in the list for $h(k)$. Although the list for $h(k)$ may be maintained using any data structure that supports the search, insert and delete operations (e.g., arrays, chains, search trees), chains are most frequently used. We typically use an array $ht[0:b\text{-}1]$ with $ht[i]$ pointing to the first node of the chain for bucket i. Program 8.4 gives the search algorithm for **chained hash tables**.

```
element* search(int k)
{
    /* search the chained hash table ht for k, if a pair with this key is found,
       return a pointer to this pair; otherwise, return NULL. */
    nodePointer current;
    int homeBucket = h(k);
    /* search the chain ht[homeBucket] */
    for (current= ht[homeBucket]; current; current = current->link)
        if (current->data.key == k) return &current->data;
    return NULL;
}
```

Program 8.4: Chain search

Figure 8.6 shows the chained hash table corresponding to the linear table found in Figure 8.4. The number of comparisons needed to search for any of the identifiers is now one each for **acos**, **char**, **define**, **exp** and **float**; two each for **atoi**, **ceil**, and **float**; three each for **atol** and **cos**; and four for **ctime**. The average number of comparisons is now 21/11 = 1.91.

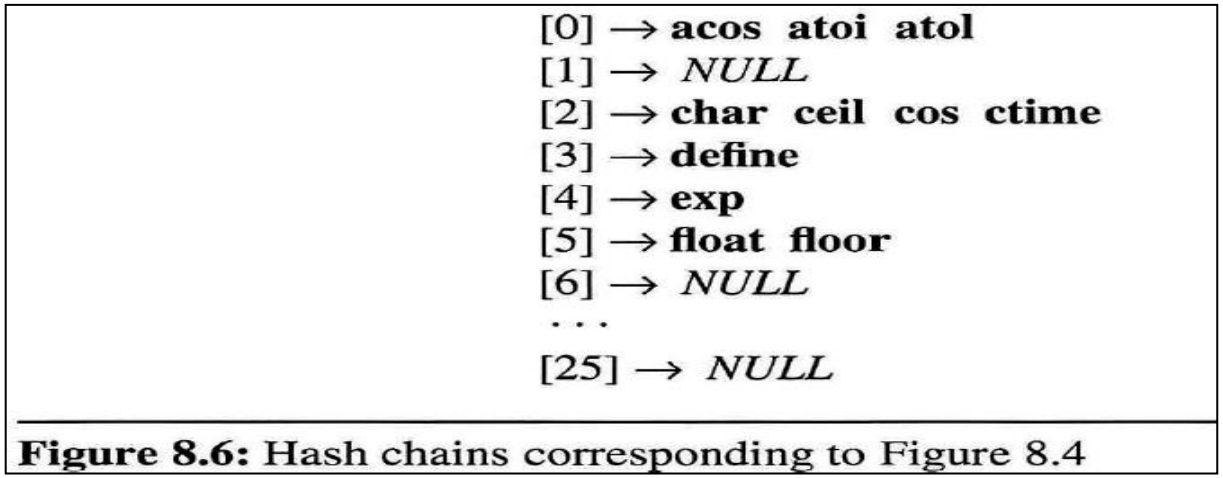


Figure 8.6: Hash chains corresponding to Figure 8.4

- To **insert** a new key, k, into a chain, we must first verify that it is not currently on the chain. Following this, k may be inserted at any position of the chain.

- **Deletion** from a chained hash table can be done by removing the appropriate node from its chain.

- When chaining is used along with a uniform hash function, the expected average number of key comparisons for a successful search is $\approx 1 + \alpha/2$, where α is the loading density n/b (b = number of buckets). For $\alpha = 0.5$ this number is 1.25, and for $\alpha = 1$ it is 1.5. The corresponding numbers for linear probing are 1.5 and b, the table size.

- The performance results cited in this section tend to imply that provided we use a uniform hash function, performance depends only on the method used to handle overflows. Although this is true when the keys are selected at random from the key space, it is not true in practice. In practice, there is a tendency to make a biased use of keys. Hence, in practice, different hash functions result in different performance. **Generally, the division hash function coupled with chaining yields best performance.**

- The worst-case number of comparisons needed for a successful search remains $O(n)$ regardless of whether we use open addressing or chaining. The worst-case number of comparisons may be reduced to $O(\log n)$ by storing synonyms in a balanced search tree (see Chapter 10) rather than in a chain.

8.2.4 Theoretical Evaluation of Overflow Techniques

The experimental evaluation of hashing techniques indicates a very good performance over conventional techniques such as balanced trees. The worst-case performance for hashing can, however, be very bad. In the worst case, an insertion or a search in a hash table with n keys may take $O(n)$ time. In this section, we present a probabilistic analysis for the expected performance of the chaining method and state without proof the results of similar analyses for the other overflow handling methods. First, we formalize what we mean by expected performance.

Let ht [0:b-1] be a hash table with b buckets, each bucket having one slot. Let h be a uniform hash function with range [0, b - 1]. If n keys kl> k2, ···, kn are entered into the hash table, then there are b^n distinct hash sequences h(k1), h(k2), · · ·, h(kn). Assume that each of these is equally likely to occur. Let Sn denote the expected number of key comparisons needed to locate a randomly chosen ki, $1 \leq i \leq n$. Then, Sn is the average number of comparisons needed to find the jth key kj, averaged over $1 \leq j \leq n$, with each j equally likely, and averaged over all b^n hash sequences, assuming each of these also to be equally likely. Let Un be the expected number of key comparisons when a search is made for a key not in the hash table. This hash table contains n keys. The quantity Un may be defined in a manner analogous to that used for Sn.

Theorem 8.1: Let $\alpha = n/b$ be the loading density of a hash table using a uniform hashing function h. Then

(1) for linear open addressing

$$U_n \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

$$S_n \approx \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha)$$

$$S_n \approx -\left[\frac{1}{\alpha}\right]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha$$

$$S_n \approx 1 + \alpha/2$$

Un: n 개의 키를 가진 hash table에서 그 테이블에 없는 key를 찾을 경우, 평균 몇 번 비교해야 하는가?

Sn: n 개의 키를 가진 hash table에서 그 테이블에 있는 key를 찾을 경우, 평균 몇 번 비교해야 하는가?

Proof: Exact derivations of U_n and S_n are fairly involved and can be found in Knuth's book *The Art of Computer Programming: Sorting and Searching* (see the References and Selected Readings section). Here we present a derivation of the approximate formulas for chaining. First, we must make clear our count for U_n and S_n. If the key k being sought has $h(k) = i$, and chain i has q nodes on it, then q comparisons are needed if k is not on the chain. If k is in the jth node of the chain, $1 \le j \le q$, then j comparisons are needed.

When the n keys are distributed uniformly over the b possible chains, the expected number in each chain is $n/b = \alpha$. Since U_n equals the expected number of keys on a chain, we get $U_n = \alpha$.

When the ith key, k_i, is being entered into the table, the expected number of keys on any chain is $(i-1)/b$. Hence, the expected number of comparisons needed to search for k_i after all n keys have been entered is $1 + (i-1)/b$ (this assumes that new entries will be made at the end of the chain). Thus,

$$S_n = \frac{1}{n}\sum_{i=1}^{n}\{1 + (i-1)/b\} = 1 + \frac{n-1}{2b} \approx 1 + \frac{\alpha}{2} \quad \square$$

EXERCISE 1 (for Hash Tables with Overflow Chaining) [4점]

Hash table ht[0:6] (즉, ht[7])을 이용하며 Overflow Chaining 방법을 이용하며 Hash Function은 h(k) = k mod 7 을 이용하여 아래 hash 작업들을 수행하시오.

(1) Hash table을 초기화 하시오.

(2) Keyboard로 부터 10, 8, 14, 12, 6, 18, 17, 24를 차례로 입력 받아 hash table에 삽입하시오.

(3) Hash table의 내용을 출력하시오. 예를 들면 다음과 같다.

Ht[0]: 14
Ht[1]: 8
Ht[2]:
Ht[3]: 10, 17, 24
Ht[4]: 18
Ht[5]: 12
Ht[6]: 6

(4) Keyboard로 부터 8, 12, 40, 17, 60을 차례로 입력 받아 hash table에서 검색하시오. 예를 들면 다음과 같다.

탐색할 키를 입력하시오: 8
8은 hash table의 1번째 엔트리의 레코드 1입니다.
…
탐색할 키를 입력하시오: 40
40은 hash table에 존재하지 않습니다.
…
탐색할 키를 입력하시오: 17
17은 hash table의 3번째 엔트리의 레코드 2입니다.
…

(5) Keyboard로 부터 8, 12, 40, 17, 60을 차례로 입력 받아 hash table에서 삭제하시오. 예를 들면 다음과 같다.

삭제할 키를 입력하시오: 8
8을 hash table의 1번째 엔트리의 레코드 1에서 삭제합니다.
…
삭제할 키를 입력하시오: 40
40은 hash table에 존재하지 않습니다.
…
삭제할 키를 입력하시오: 17
17을 hash table의 3번째 엔트리의 레코드 2에서 삭제합니다.
…

(6) Hash table의 내용을 출력하시오. 예를 들면 다음과 같다.

Ht[0]: 14
Ht[1]:
Ht[2]:
Ht[3]: 10, 24
Ht[4]: 18
Ht[5]:
Ht[6]: 6

EXERCISE 2 (for Hash Tables with Overflow Chaining) [6점]

Student 배열 a[40]에 40명의 학생 정보를 생성하여 저장한다. 각 학생의 정보는 <소속, 학번, 성명>으로 구성된다. 학생의 성명을 이용하여 학생정보를 탐색하고자 <성명, 배열 index> 정보를 hash table ht[13] (즉, ht[0:12])에 저장하고자 한다. Overflow Chaining 방법을 이용하며 Hash Function은 Program 8.1과 Division 방법을 이용하여 아래 hash 작업들을 수행하시오.

(1) a[40]에 40명의 학생 정보를 생성한 후, 그를 출력하시오.

(2) a[40]에 저장된 40명의 학생 정보의 <성명, 배열 index>를 모두 ht[13]에 삽입하시오. 삽입 알고리즘을 개발하여야 한다.

1) 각 삽입에 몇 번의 비교를 하였는가? 전체 평균은?

2) hash entry의 각 chain의 length는? 전체 평균은?

(3) a[40]에 저장된 40명의 학생 정보들을 차례로 hash table에서 탐색하시오. 이는 Program 8.4: Chain search를 이용하면 된다.

1) 각 탐색에 몇 번의 비교를 하였는가? 전체 평균은?

2) 알고리즘의 정확성을 검사하기 위하여 아래 작업과 같은 작업을 수행하시오.

1) 탐색할 학생 성명 입력: 일지매
   - 학생정보 : index = 3, <컴퓨터학부, 2015012345, 일지매>
   - 학생정보 : index = 7, <컴퓨터학부, 2014021435, 일지매>
2) 탐색할 학생 성명 입력: 홍길동
   - 학생정보 : 없음

(4) a[40]에 저장된 40명의 학생 정보들을 차례로 hash table에서 삭제하시오. 삭제 알고리즘을 개발하여야 한다.

1) 각 삭제에 몇 번의 비교를 하였는가? 전체 평균은?

8.3 Dynamic Hashing

8.3.1 Motivation for Dynamic Hashing

- To ensure good performance, **it is necessary to increase the size of a hash table whenever its loading density exceeds a prespecified threshold**. So, for example, if we currently have b buckets in our hash table and are using the division hash function with divisor $D = b$, then, when an insert causes the loading density to exceed the prespecified threshold, we use **array doubling** to increase the number of buckets to $2b + 1$. At the same time, the hash function divisor changes to $2b + 1$. This change in divisor requires us to rebuild the hash table by collecting all dictionary pairs in the original smaller size table and reinserting these into the new larger table. We cannot simply copy dictionary entries from the smaller table into corresponding buckets of the bigger table as **the home bucket for each entry has potentially changed**. For very large dictionaries that must be accessible on a 24/7 basis, the required rebuild means that dictionary operations must be suspended for unacceptably long periods while the rebuild is in progress.

- ***Dynamic hashing***, which also is known as ***extendible hashing***, aims to **reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket**. In other words, although table doubling increases the total time for a sequence of n dictionary operations by only $O(n)$, the time required to complete an insert that triggers the doubling is excessive in the context of a large dictionary that is required to respond quickly on a per operation basis. The objective of dynamic hashing is to provide acceptable hash table performance on a **per operation basis**.

We consider two forms of dynamic hashing - one uses a **directory** and the other does not - in this section. For both forms, we use a hash function h that maps keys into non-negative integers. The range of h is assumed to be sufficiently large and we use ***h*(*k*, *p*)** to denote **the integer formed by the *p* least significant bits of *h*(*k*).**

For the examples of this section, we use a hash function $h(k)$ that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation. Figure 8.7 shows 8 possible 2 character keys together with the binary representation of $h(k)$ for each. For our example hash function, $h(A0, 1) = 0$, $h(A1, 3) = 1$, $h(B1, 4) = 1001 = 9$, and $h(C1, 6) = 110\ 001 = 49$.

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

- letters A ~ C → 100, 101, 110
- digits 0 ~ 7 → 000, 001, … ,111

Figure 8.7: An example hash function

8.3.2 Dynamic Hashing using Directories

We employ a directory, d, of pointers to buckets. The size of the directory depends on the number of bits of $h(k)$ used to index into the directory. When indexing is done using, say, $h(k, 2)$, the directory size is $2^2 = 4$; when $h(k, 5)$ is used, the directory size is $2^5 = 32$. The number of bits of $h(k)$ used to index the directory is called the ***global depth*** (or ***directory depth***). The size of the directory is 2^{gd}, where gd is the global depth and the number of buckets is at most equal to the directory size. Figure 8.8(a) shows a ***dynamic hash table*** that contains the keys A0, B0, A1, B1, C2, and C3. This hash table uses a directory whose depth is 2 and uses buckets that have 2 slots each. In Figure 8.8, the directory is shaded while the buckets are not. In practice, the bucket size is often chosen to match some physical characteristic of the storage media. For example, when the dictionary pairs reside on disk, a bucket may correspond to a disk track or sector.

- To **search** for a key k, we merely examine the bucket pointed to by d[h(k,gd)], where gd is the global depth.

- Suppose we **insert** C5 into the hash table of Figure 8.8 (a). Since, h(C5,2) = 01, we follow the pointer, d[01], in position 01 of the directory. This gets us to the bucket with A1 and B1. This bucket is full and we get a bucket overflow. To resolve the overflow, we **determine the least u such that h(k,u) is not the same for all keys in the overflowed bucket.** In case the least u is greater than the directory depth, we increase the directory depth to this least u value. This requires us to increase the directory size but not the number of buckets. When the directory size doubles, the pointers in the original directory are duplicated so that the pointers in each half of the directory are the same. A quadrupling of the directory size may be handled as two doublings and so on. For our example, the least u for which h(k,u) is not the same for A1, B1, and C5 is 3. So, the directory is expanded to have depth 3 and size 8. Following the expansion, **d[i] = d [i + 4], $0 \leq i < 4$**.

- Following the resizing (if any) of the directory, we split the overflowed bucket using h(k,u). In our case, the overflowed bucket is split using h(k, 3). For Al and Bl, h(k, 3) = 001 and for C5, h(k, 3) = 101. So, we create a new bucket with C5 and place a pointer to this bucket in d[101]. Figure 8.8 (b) shows the result. Notice that each dictionary entry is in the bucket pointed at by the directory position h(k, 3), although, in some cases the dictionary entry is also pointed at by other buckets. For example, bucket 100 also points to A0 and B0, even though h(A0,3) = h(B0,3) ≠ 100.

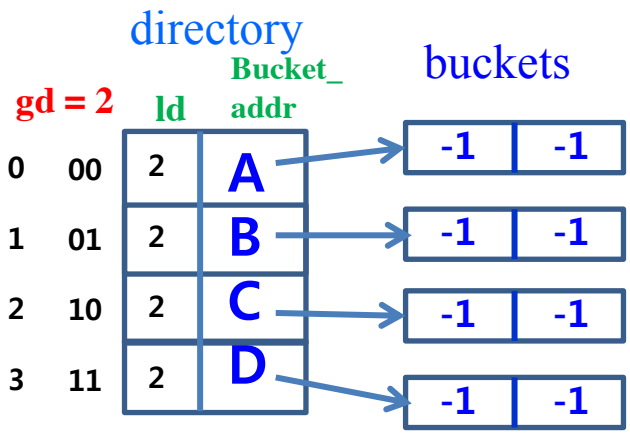
- Suppose that instead of C5, we were to **insert** Cl. The pointer in position h(Cl,2) = 01 of the directory of Figure 8.8 (a) gets us to the same bucket as when we were inserting C5. This bucket overflows. **The least u for which h (k,u) isn't the same for A1, B1 and C1 is 4.** So, the new directory depth is 4 and its new size is 16. The directory size is quadrupled and the pointers d[0:3] are replicated 3 times to fill the new directory. When the overflowed bucket is split, A1 and C1 are placed into a bucket that is pointed at by d[0001] and B1 into a bucket pointed at by d[1001].

- When the current directory depth is greater than or equal to u, some of the other pointers to the split bucket also must be updated to point to the new bucket. Specifically, the pointers in positions that agree with the last u bits of the new bucket need to be updated. The following example illustrates this. Consider **inserting** A4 (h (A4) = 100 100) into Figure 8.8 (b). Bucket d [100] overflows. The least u is 3, which equals the directory depth. So, the size of the directory is not changed. Using h(k, 3), A0 and B0 hash to 000 while A4 hashes to 100. So, we create a new bucket for A4 and set d[100] to point to this new bucket.

- As a final **insert** example, consider inserting C1 into Figure 8.8 (b). h(C1,3) = 001. This time, bucket d [001] overflows. The minimum u is 4 and so it is necessary to double the directory size and increase the directory depth to 4. **When the directory is doubled, we replicate the pointers in the first half into the second half**. Next we split the overflowed bucket using h(k, 4). Since h(k, 4) = 0001 for A1 and Cl and 1001 for B1, we create a new bucket with B1 and put C1 into the slot previously occupied by B1. A pointer to the new bucket is placed in d[1001]. Figure 8.8 (c) shows the resulting configuration. For clarity, several of the bucket pointers have been replaced by lowercase letters indicating the bucket pointed to.

- **Deletion** from a dynamic hash table with a directory is similar to insertion.
- Although dynamic hashing employs array doubling, the time for this array doubling is considerably less than that for the array doubling used in static hashing. This is so because, in dynamic hashing, **we need to rehash only the entries in the bucket that overflows rather than all entries in the table**. Further, savings result when the directory resides in memory while the buckets are on disk.
- A search requires only 1 disk access; an insert makes 1 read and 2 write accesses to the disk, the array doubling requires no disk access.

Extendible Hashing



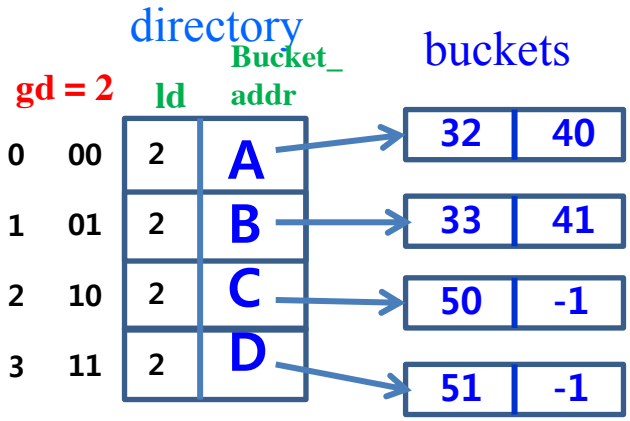
초기화 상태

- global depth (or directory depth) *gd*.
- local depth *ld*.

h(key, gd) = key의 LSB의 gd 비트들.

To search for a key *key*, examine the bucket in d[h(key, gd)].

Insert 32, 33, 50, 51, 40, 41, 53, 49.



초기화 상태에서,
32가 입력되면? : h(32, 2) = h(0010 0000, 2) = 00 = 0 : d[0]의 bucket에 삽입
33이 입력되면? : h(33, 2) = h(0010 0001, 2) = 01 = 1 : d[1]의 bucket에 삽입
50이 입력되면? : h(50, 2) = h(0011 0010, 2) = 10 = 2 : d[2]의 bucket에 삽입
51이 입력되면? : h(51, 2) = h(0011 0011, 2) = 11 = 3 : d[3]의 bucket에 삽입
40이 입력되면? : h(40, 2) = h(0010 1000, 2) = 00 = 0 : d[0]의 bucket에 삽입
41이 입력되면? : h(41, 2) = h(0010 1001, 2) = 01 = 1 : d[1]의 bucket에 삽입
53이 입력되면? : h(53, 2) = h(0011 0101, 2) = 01 = 1 : d[1]의 bucket에 OVERFLOW
어떻게 하나?

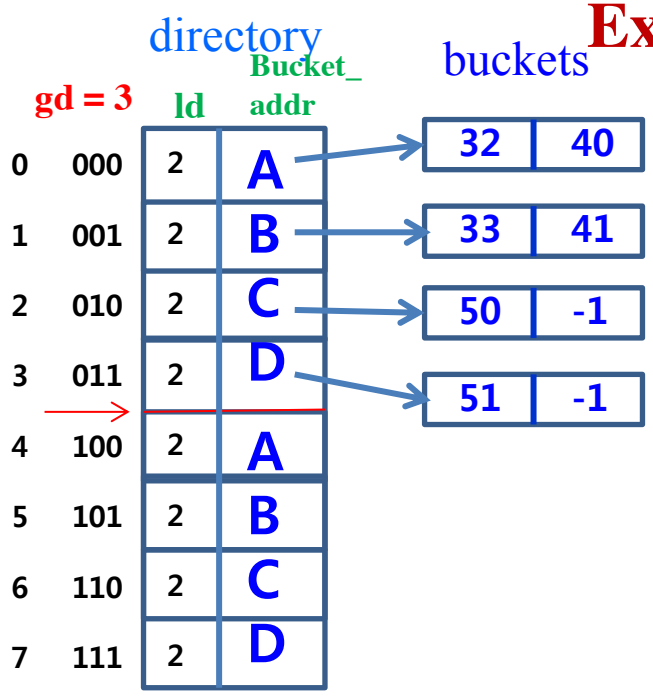
```
int get_index(int key, int gd)
{
    int index, mask;

    mask = get_mask(gd);
    index = key & mask;
    return index;
}
```

```
int get_mask(int gd)
{
    int mask = 0x01;
    int count;

    for (count = 1; count < gd; count++)
        mask = (mask << 1) + 0x01;
    return mask;
}
```

Extendible Hashing

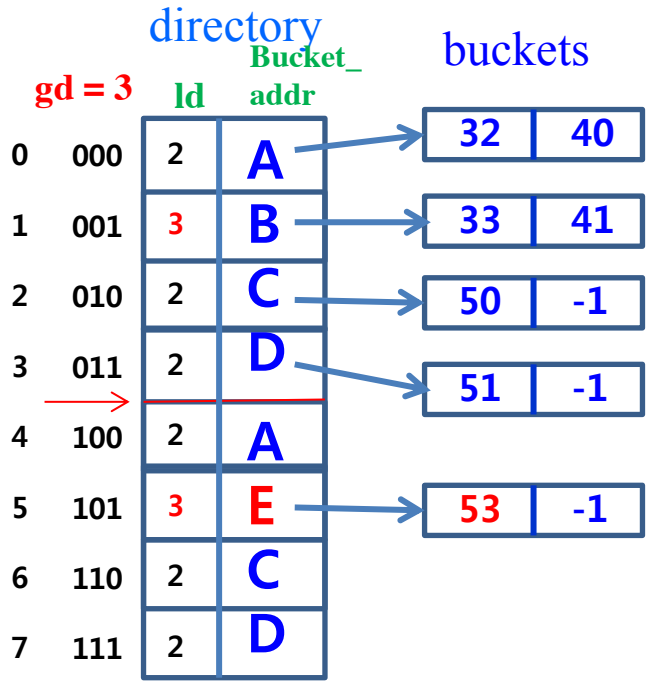


앞 상태에서, 53이 입력되면,
1. h(53, 2) = 1. Overflow in bucket *B*.
2. gd = d[01].ld. So, double the directory and 전반부 내용을 후반부에 copy한다.
   gd +=1. Now, gd becomes 3.
   Now, d[01] becomes d[001] and d[101] such that they are buddies.
3. Rehash (entries in bucket *B* + 53) with h(k,3).
   h(33,3)=001, h(41,3)=001, h(53,3)=101
   With gd=3 and ld=3, we need two buckets, one of the original *B* and a new bucket *E*.
   d[1] -> B: [33, 41] ld = 3;
   d[5] -> E: [53, -] ld = 3;

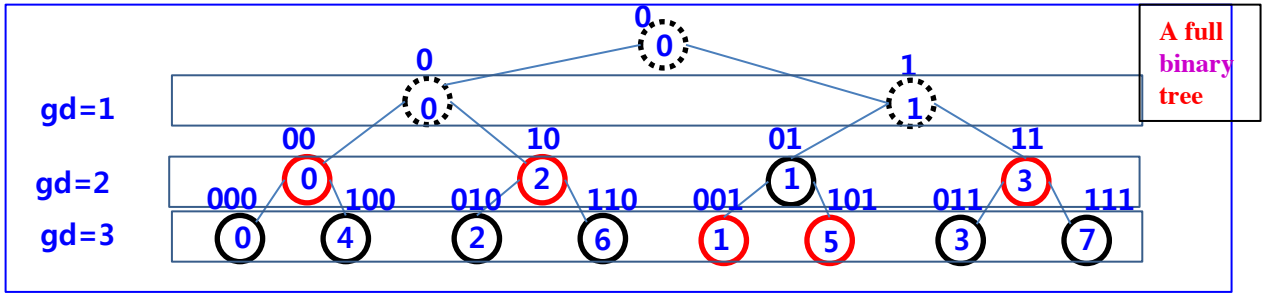
33 and 41 go to the bucket *B* in d[001].
Get a new bucket *E* and put it in d[5] with ld=3.
53 goes to the bucket *E* in d[101].

Note that d[4] also points to A, even though h(32,3) = h(40,3) ≠ 4.

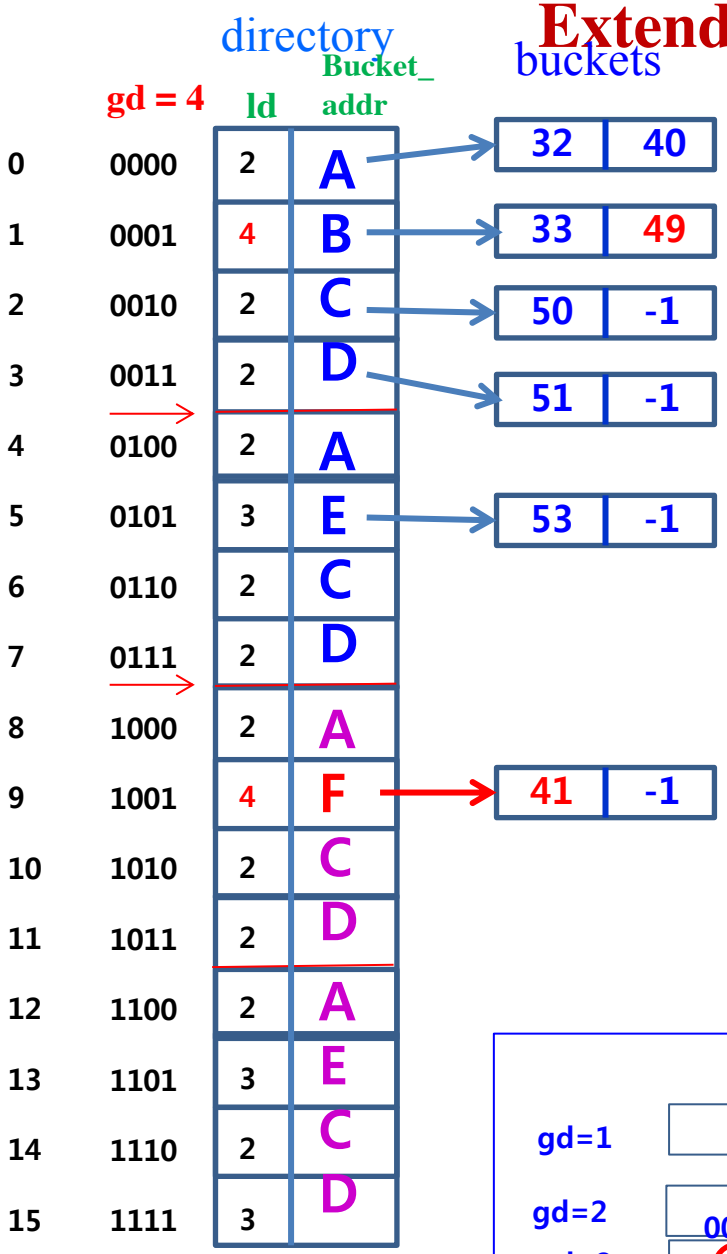
참고: Directory를 두 배로 확장에서,
w = 2^{gd}; // w = (0x01 << gd);
Directory를 realloc()을 사용하여 두 배(2*w)로 확장;
for (k = 0; k < w; k++) {
    // dr[k]는 그대로;
    dr[k + w] = dr[k];
}



확장 전에 OVERFLOW를 일으킨 entry의 index가 ov이면, 확장 후,
(d[ov]의 bucket의 키들 + 삽입할 키) 들은 d[ov]와 d[ov + w]로 split된다.
위 예의 경우, (d[1]의 bucket의 키들 + 삽입할 키인 53) 들은 d[1]과 d[1 + 4]로 split된다.



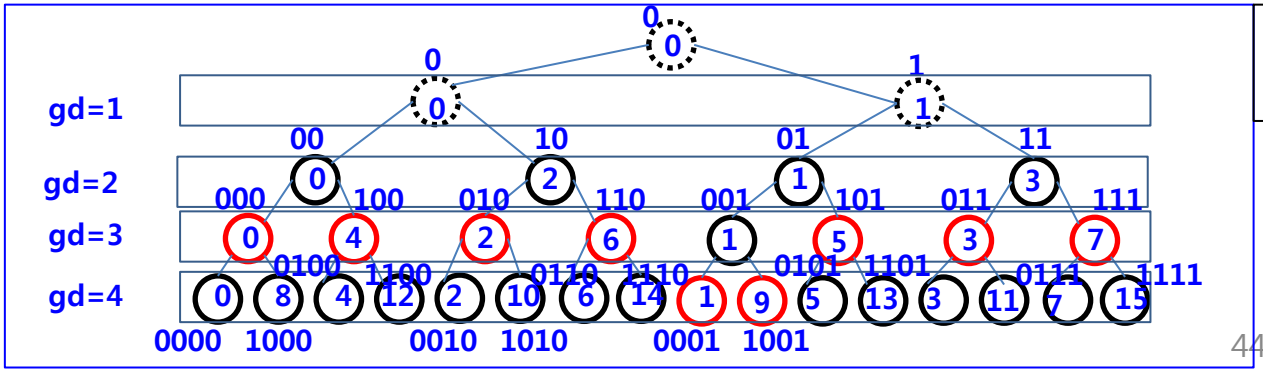
Extendible Hashing



앞 상태에서, 49가 입력되면
1. h(49, 3) = h(0011 0001, 3) = 001 = 1. OVERFLOW in bucket B.
2. gd = d[01].ld. So, double the directory and 전반부 내용을 후반부에 copy한다.
   gd +=1. Now, gd becomes 4.
   Now, d[001] becomes d[0001] and d[1001] such that they are buddies.
3. Rehash (entries in bucket *B* + 49) with h(k,4).
   h(33,4)=0001, h(41,4)=1001, h(49,4)=0001
   With gd=4 and ld=4, we need two buckets, one of the original *B* and a new bucket *F*.
   d[1] -> B: [33, 49] ld = 4;
   d[9] -> F: [41, -] ld = 4;

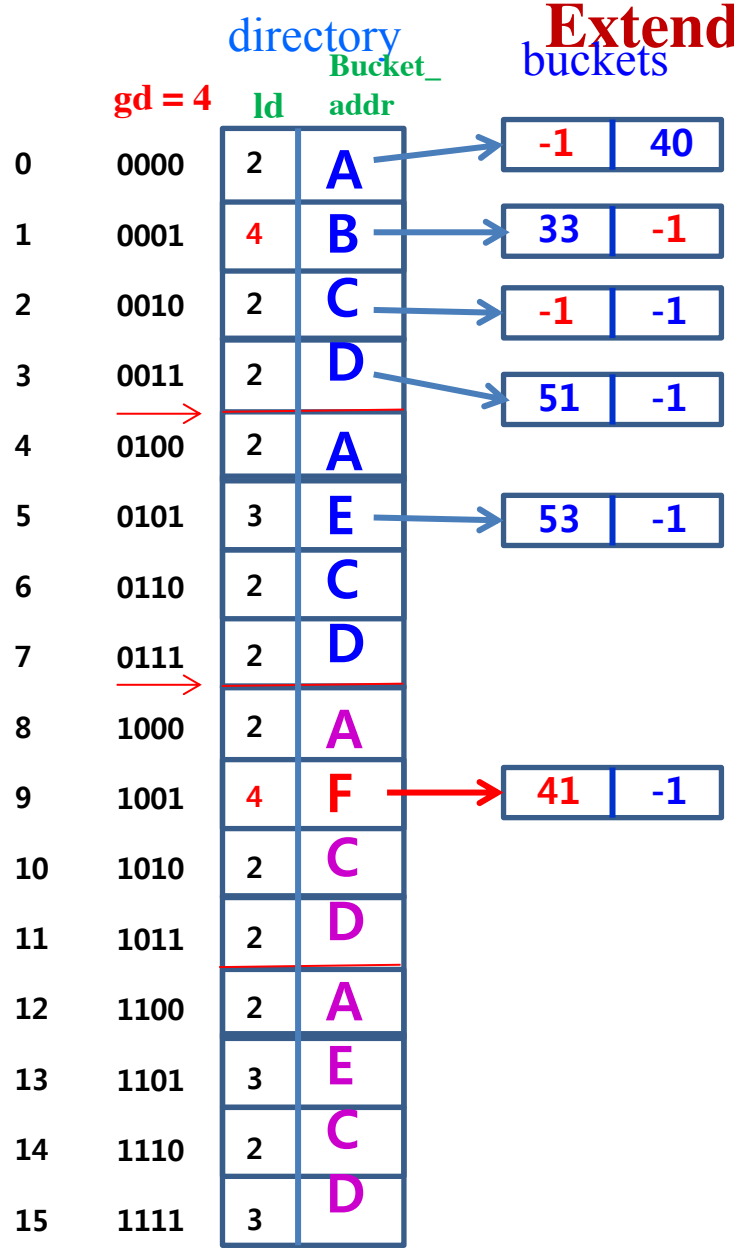
33 and 49 go to the bucket *B* in d[0001].
Get a new bucket *F* and put it in d[9] with ld=4.
41 goes to the bucket *F* in d[1001].

참고로, d[5]의 내용은 건드리지 않는다. 왜 그럴까?
53 : h(53, 4) = h(0011 0101, 4) = 0101 = 5



A full binary tree

Extendible Hashing



앞 상태에서,
Delete 50, 32, 49, 41, 33.

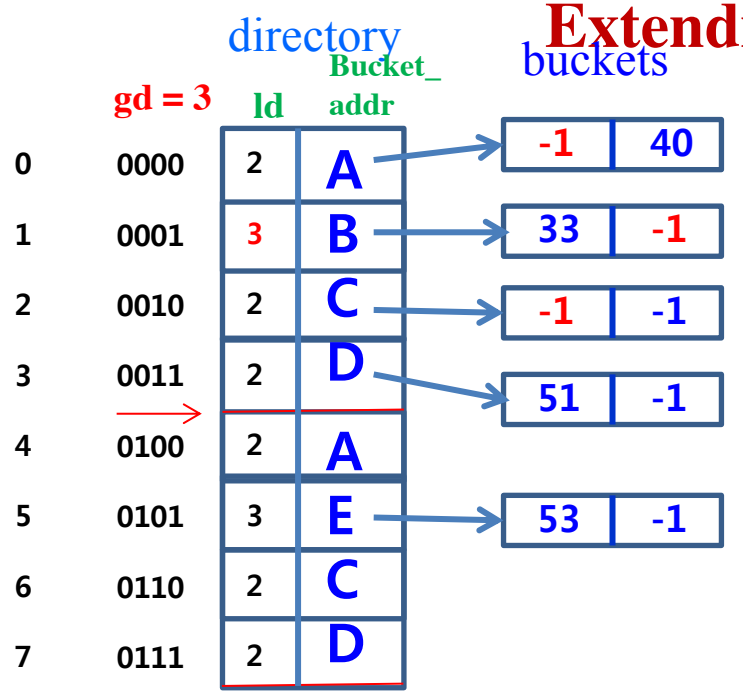
Delete 50:
h(50, 4) = h(0011 0010, 4) = 0010 = 2 : d[2]의 bucket C에서 50을 확인 후 삭제.
d[2]의 bucket의 모든 slot이 -1이 되었나?
Yes.
dr[2]의 ld=2 즉, 초기값인가?
Yes. (그 bucket을 삭제하지 않고 그냥 둔다.)

Delete 32:
h(32, 4) = h(0010 0000, 4) = 0000 = 0 : d[0]의 bucket A에서 32를 확인 후 삭제.
dr[0]의 bucket의 모든 slot이 -1이 되었나?
No.

Delete 49:
h(49, 4) = h(0011 0001, 4) = 0001 = 1 : d[1]의 bucket B에서 32를 확인 후 삭제.
d[1]의 bucket의 모든 slot이 -1이 되었나?
No.

Delete 41:
h(41, 4) = h(0010 1001, 4) = 1001 = 9 : d[9]의 bucket F에서 41을 확인 후 삭제.
d[9]의 bucket의 모든 slot이 -1이 되었나?
Yes.
d[9]의 ld=2 즉, 초기값인가?
No.
dr[9].ld는 4이다.
그 bucket을 삭제한다.
gd=4에서 d[9]의 sibling은? $9 - 2^3 = 9 - 8 = 1$.
dr[9].bucket_addr = dr[1].bucket_addr;
dr[9].ld -= 1; dr[1].ld -1=1;
dr[9]의 변경전 ld값이 gd와 같았으므로 다음을 수행한다.
dr[0] ~ dr[7]의 모든 ld를 검사한다. 만약 모두가 gd보다 작다면, gd -= 1;
이 경우, gd = 3이 되고, directory의 후반부를 제거한다.
그 결과를 다음 페이지에 나타내었다.

Extendible Hashing



Delete 33:

h(33, 3) = h(0010 0001, 3) = 001 = 1 : d[1]의 bucket B에서 33을 확인 후 삭제.

bucket B의 모든 slot이 -1이 되었나?

Yes.

d[1]의 ld=2 즉, 초기값인가?

No.

d[1].ld는 현재 3이다.

bucket B를 삭제한다.

gd=3에서 d[1]의 sibling은? $1 + 2^2 = 1 + 4 = 5$.

d[1].bucket_addr = d[5].bucket_addr;

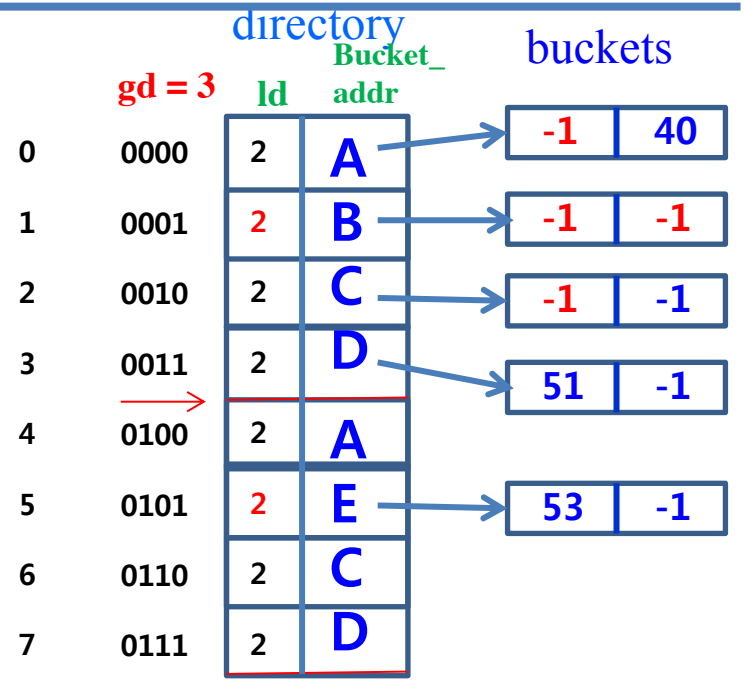
d[1].ld -= 1; d[5].ld -1=1;

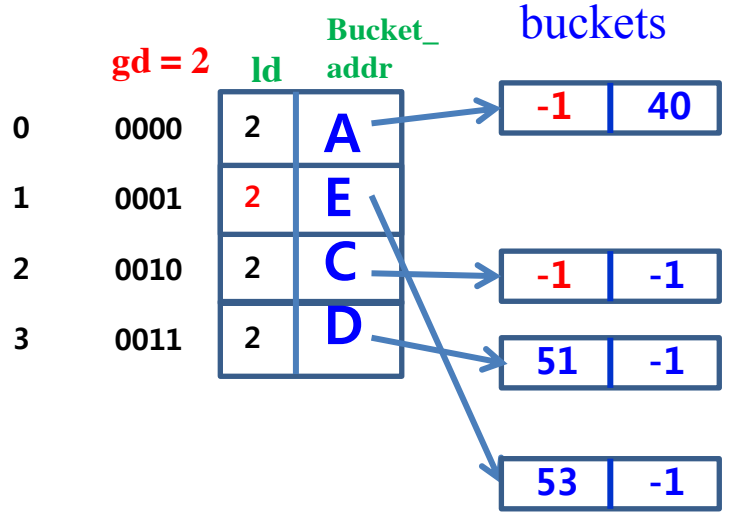
d[1]의 변경전 ld값이 gd와 같았으므로 다음을 수행한다.

dr[0] ~ dr[3]의 모든 ld를 검사한다. 만약 모두가 gd보다 작다면, gd -= 1;

이 경우, gd = 2가 되고, directory의 후반부를 제거한다.

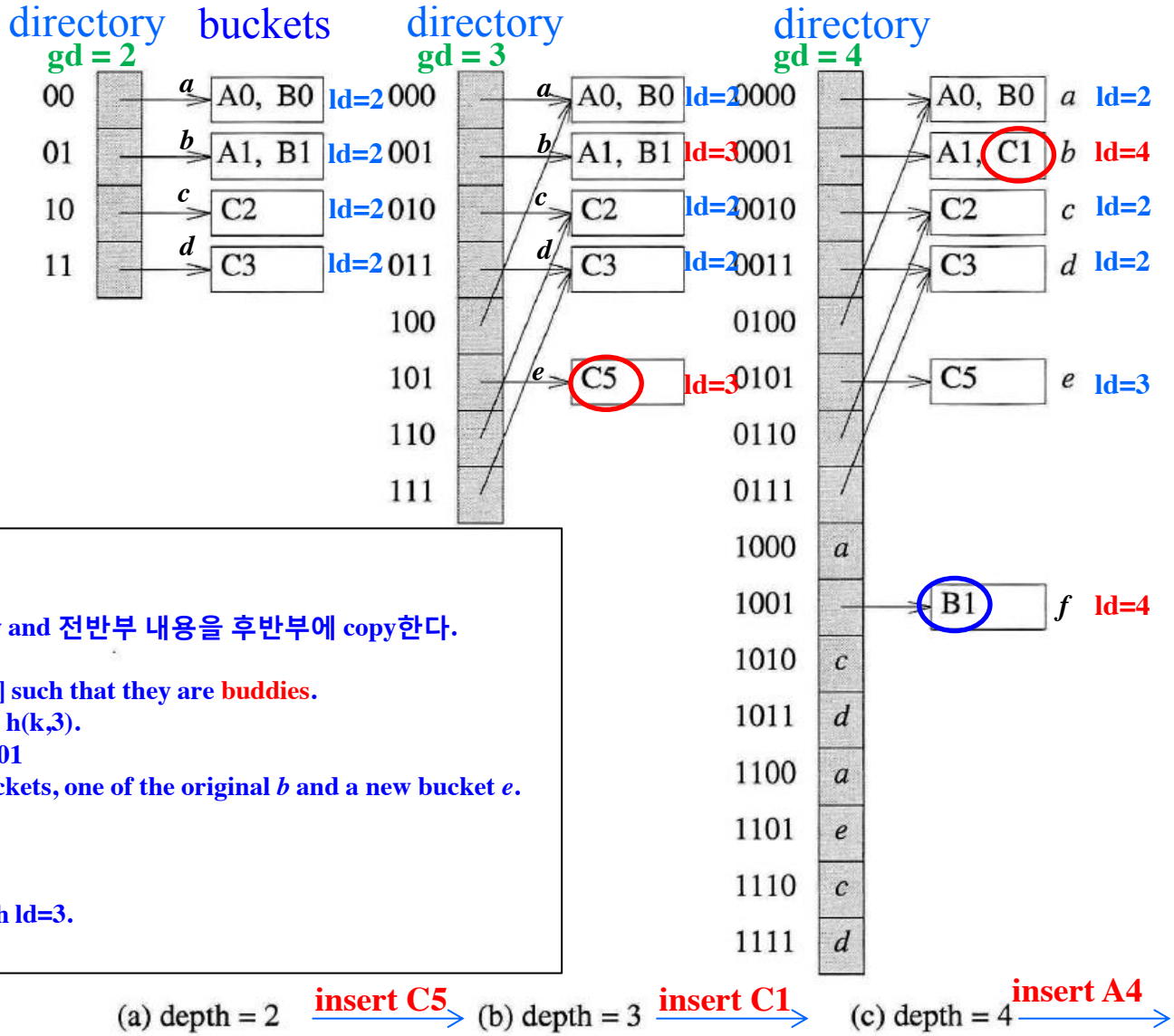
그 결과는 아래 우측에 나타내었다.





Extendible Hashing

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



insert C5 into (a).

1. h(C5,2) = 01. Overflow in bucket *b*.
2. gd = d[01].ld. So, double the directory and 전반부 내용을 후반부에 copy한다.
   gd +=1. Now, gd becomes 3.
   Now, d[01] becomes d[001] and d[101] such that they are buddies.
3. Rehash (entries in bucket *b* + C5) with h(k,3).
   h(A1,3)=001, h(B1,3)=001, h(C5,3)=101
   With gd=3 and ld=3, we need two buckets, one of the original *b* and a new bucket *e*.
   d[1] -> *b*: [A1, B1] ld = 3;
   d[5] -> *e*: [C5, -] ld = 3;

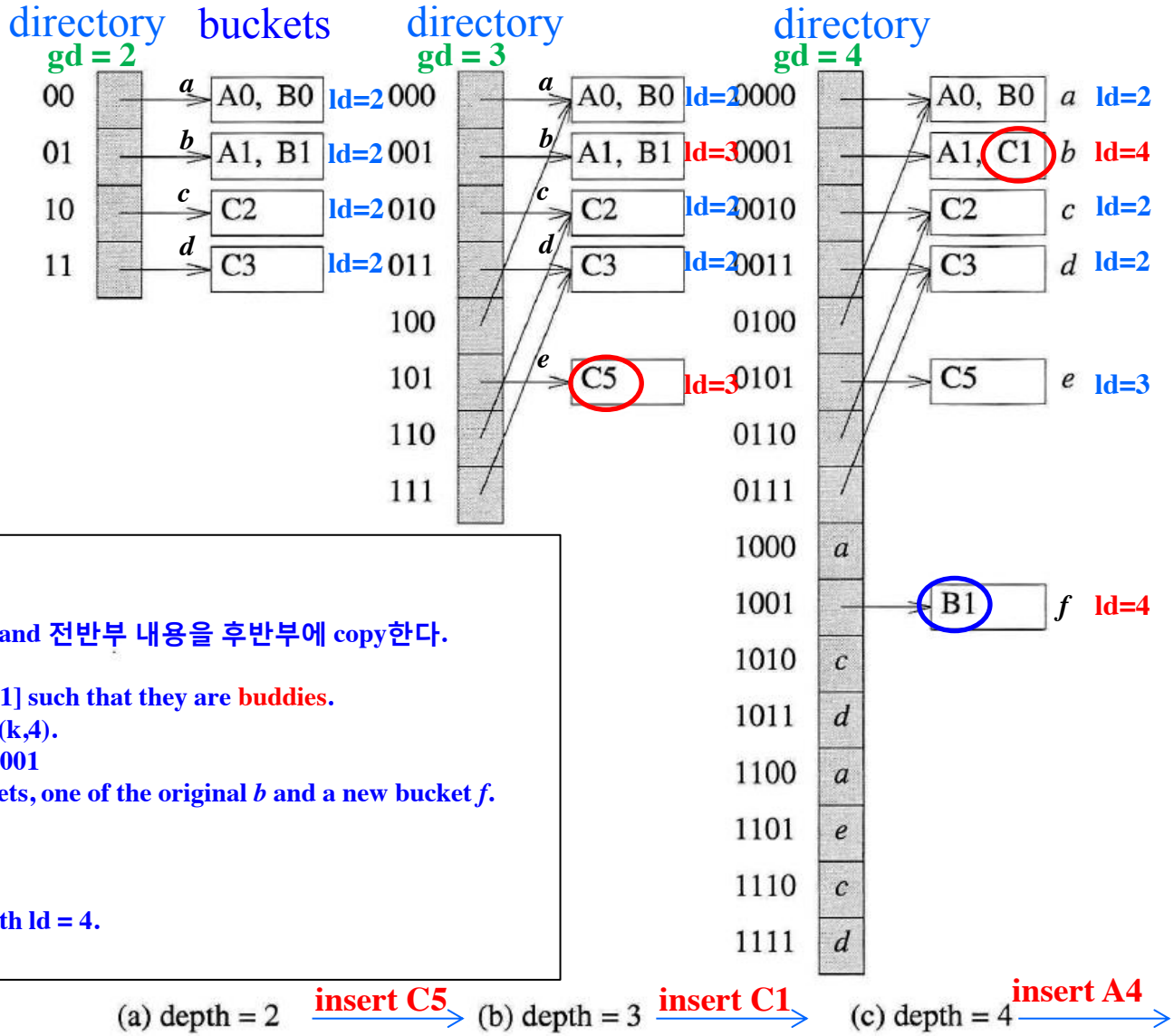
A1 and B1 go to the bucket *b* in d[001].
Get a new bucket *e* and put it in d[5] with ld=3.
C5 goes to the bucket *e* in d[101].

Figure 8.8: Dynamic hash tables with directories

buckets with 2 slots

Extendible Hashing

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



insert C1 into (b).

1. h(C1,3) = 001. Overflow in bucket *b*.
2. gd = d[001].ld. So, double the directory and 전반부 내용을 후반부에 copy한다.
   gd +=1. Now, gd=4.
   Now, d[001] becomes d[0001] and d[1001] such that they are buddies.
3. Rehash (entries in bucket *b* + C1) with h(k,4).
   h(A1,4)=0001, h(B1,4)=1001, h(C1,4)=0001
   With gd=4 and ld=4, we need two buckets, one of the original *b* and a new bucket *f*.
   d[1] = *b*: [A1, C1] ld = 4;
   d[9] = *f*: [B1, -] ld = 4;

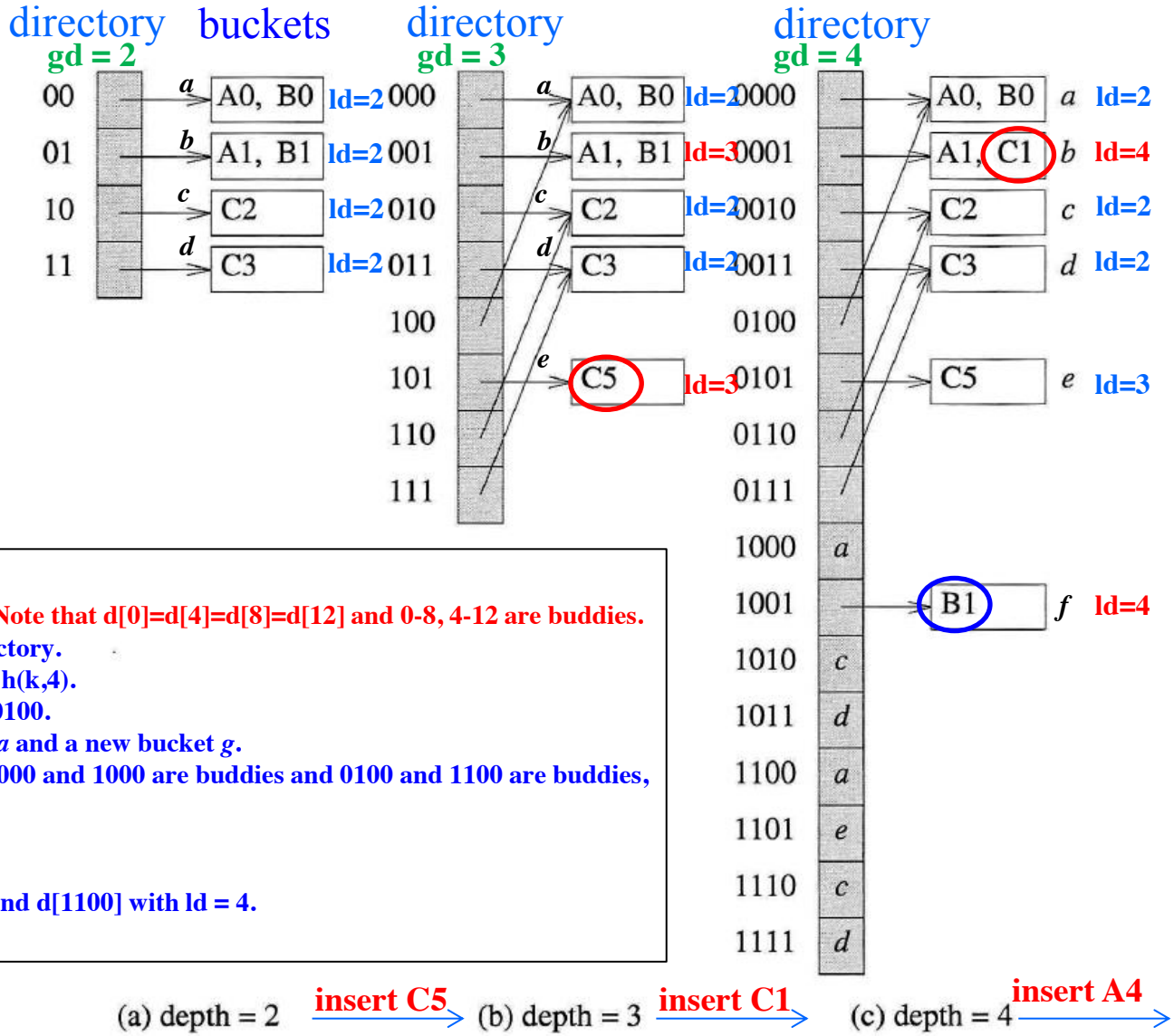
A1 and C1 go to the bucket *b* in d[0001].
Get a new bucket *f* and put it in d[1001] with ld = 4.
B1 goes to the bucket *f* in d[1001].

Figure 8.8: Dynamic hash tables with directories

buckets with 2 slots

Extendible Hashing

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



insert A4 into (c).

1. h(A4,4) = 0100. Overflow in bucket *a*. Note that d[0]=d[4]=d[8]=d[12] and 0-8, 4-12 are buddies.
2. gd > d[4].ld. So, do not double the directory.
3. Rehash (entries in bucket *a* + A4) with h(k,4).
   h(A0,4)=0000, h(B0,4)=1000, h(A4,4)=0100.
   We need 2 buckets, one of the original *a* and a new bucket *g*.
   Since ld of the bucket in d[4] is 2 and 0000 and 1000 are buddies and 0100 and 1100 are buddies,
   d[0] = d[8] = *a*: [A0, B0], ld = 3;
   d[4] = d[12] = *g*: [A4, -], ld = 3;

A0 and B0 goes to the bucket *a*.
Get a new bucket *g* and put it in d[0100] and d[1100] with ld = 4.
A4 goes to the bucket g in d[0100].

Figure 8.8: Dynamic hash tables with directories

buckets with 2 slots

EXERCISE 1 (Extendible Hashing)

For the examples of this section, we use a hash function h(k) that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation. Figure 8.7 shows 8 possible 2 character keys together with the binary representation of h(k) for each. For our example hash function, h(A0, 1) = 0, h(A1,3) = 1, h(B1,4) = 1001 = 9, and h (C1 ,6) = 110 001 = 49.

문제 1: key A0, A1, A2, …, A7, B0, B1, …, C5, C6, C7에 대한 hash value를 생성하는 hash function h(k)를 만들고 그 결과를 Figure 8.7과 같이 생성하시오. [2점]

문제 2: h(k)를 대상으로 h(k, p)를 생성하는 함수를 만들고 그 결과를 p = 1, p = 2, p = 3, p = 4, p = 5를 대상으로 수행하시오. [2점]

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

Figure 8.7: An example hash function

- letters A ~ C → 100, 101, 110
- digits 0 ~ 7 → 000, 001, … ,111

EXERCISE 2 (Extendible Hashing)

아래 요구를 만족하는 Extendible Hash 프로그램을 작성하시오.

1) The initial global depth for the directory is 2. In other words, the initial directory has 4 entries.
2) Each bucket takes 2 slots.
3) Initially we have four empty buckets and each entry in the initial directory points to one of them.
4) We use $h(k, p)$ as in Exercise 1.

(1) Insert the following keys in the order: A0, B0, A1, B1, C2, C3, C5, C1, A4, B7. [4점]
   1) 삽입을 완료한 후, global depth는 얼마인가? Bucket의 수는 얼마인가?
   2) 삽입을 완료한 후, 각 bucket의 내용과 local depth를 출력하시오.

(2) Search the keys in the order: C1, A4. [1점]
   1) 모두 성공하였는가? 하나라도 실패하였다면 문제가 있다.

(3) Delete the keys in the order: C3, C5, B1, A1, A4, C2, A0, B0, C1, A4, B7. [3점]
   1) 각 삭제를 수행한 후, global depth는 얼마인가? Bucket의 수는 얼마인가?
   2) 모든 삭제를 완료한 후, 각 bucket의 내용과 local depth를 출력하시오.

EXERCISE 3 (for Extendible Hashing) [10점]: homework

정수 배열 a[1000]에 함수 rand()를 이용하여 random 하게 생성한 정수 1000개를 저장한 후, 그 숫자들을 차례로 hash table에 저장하고자 한다. 아래 요구를 만족하는 프로그램을 작성하시오.

1) The initial global depth for the directory is 4. In other words, the initial directory has 8 entries.
2) Each bucket takes 8 slots.
3) h(k) = k mod m, where m might be either the global depth or the local depth.
4) We use $h(k, p)$ to denote the integer formed by the p least significant bits of $h(k)$.

(1) a[1000]에 저장된 1000개의 정수들을 모두 hash table에 삽입하시오.
   1) 삽입을 완료한 후, global depth는 얼마인가? Bucket의 수는 얼마인가?
   2) 삽입을 완료한 후, 각 bucket의 내용과 local depth를 출력하시오.

(2) a[1000]에 저장된 1000개의 정수들을 차례로 hash table에서 탐색하시오.
   1) 모두 성공하였는가? 하나라도 실패하였다면 문제가 있다.

(3) a[1000]에 저장된 980개의 정수들을 차례로 hash table에서 삭제하시오.
   1) 삭제를 완료한 후, global depth는 얼마인가? Bucket의 수는 얼마인가?
   2) 삭제를 완료한 후, 각 bucket의 내용과 local depth를 출력하시오.