Hashing

Introduction to Data Structures

Kyuseok Shim

SoEECS, SNU.

8.1 INTRODUCTION

- Binary search tree (Chapter 5)
  - GET, INSERT, DELETE – O(n)
- Balanced binary search tree (Chapter 10)
  - GET, INSERT, DELETE – O(log n)
- Hashing
  - GET, INSERT, DELETE – O(1)

- Static hashing, dynamic hashing

8.2 STATIC HASHING
8.2.1 Hash Tables

- Static hashing
  - Dictionary pairs – stored in a hash table
  - Hash table
    - Partitioned into b buckets ht[0],…,ht[b−1]
    - Each bucket – holding s dictionary pairs (or pointers to this many pairs)
  - Key
    - Address of a pair
    - Determined by a hash function, h
    - h(k) – integer in the [0, b−1]
  - Under ideal condition
    - Dictionary pairs are stored in their home buckets.

8.2 STATIC HASHING
8.2.1 Hash Tables

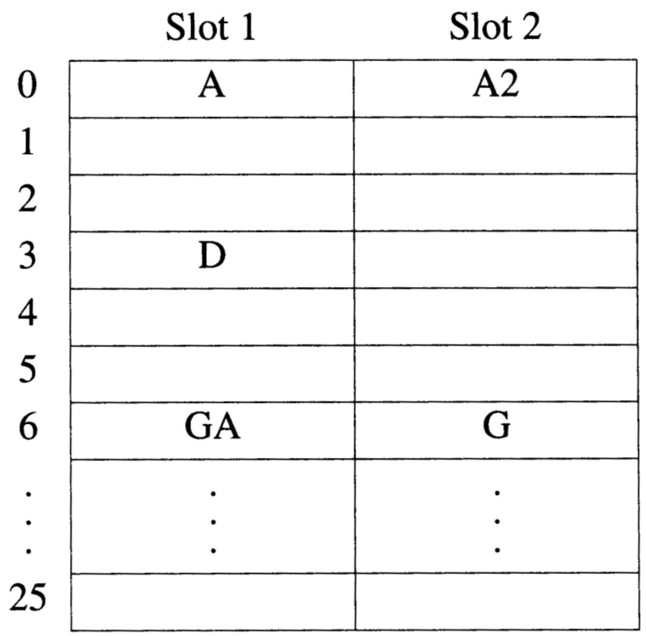
- Definition
  - The key density of a hash table is the ratio n/T.
    - n–number of pairs in the table
    - T–total number of possible keys
  - The loading density or loading factor of a hash table is $\alpha = n/(sb)$ (s – # of each bucket's slots, b – # of buckets)

8.2 STATIC HASHING
8.2.1 Hash Tables

- Synonyms
  - Two identifiers I1 and I2 if h(I1) = h(I2)
- Overflow
  - A new identifier is hashed into a full bucket.
- Collision
  - Two nonidentical identifiers are hashed into the same bucket. When s = 1, collisions and overflows occur simultaneously.

8.2 STATIC HASHING

8.2.1 Hash Tables



	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
⋮	⋮	⋮
25		

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

8.2 STATIC HASHING
8.2.2 Hash Functions

- Hash function
  - Easy to compute
  - Minimize collisions
- Uniform hash function gives 1/b probability of $h(x) = i$ to x.

8.2 STATIC HASHING
8.2.2 Hash Functions

- Division
  - We divide the identifier k by some number *D* and use the remainder as the hash address for *k*.

    $$h(\mathrm{k}) = \mathrm{k} \% D$$

    - This gives bucket addresses that range 0 to *D* – 1, where *D* = that table size.
  - The choice of *D* is critical.
    - If *D* is divisible by 2, then odd keys are mapped to odd buckets and even keys are mapped to even buckets.
    - When many identifiers are permutations of each other, a biased use of the table results.
    - In practice, choose D such that it has no prime divisors less than 20.

8.2 STATIC HASHING
8.2.2 Hash Functions

- Mid-Square
  - Used in many cases.
  - Square the identifier and use an appropriate number of bits from the middle.

```
ex)          10100
             10100
        ------------------
          110010000
```

$$11\underbrace{00100}_{}00 \quad 2^5 \text{ bits used.}$$

  - r bits used – table size = 2^r

8.2 STATIC HASHING
8.2.2 Hash Functions

- Folding
  - We partition the identifier *x* into several parts. All parts, except for the last one have the same length. We then add the parts together to obtain the hash address for *x*.

8.2 STATIC HASHING
8.2.2 Hash Functions

- Folding
  - There are two ways of carrying out this addition.
    - *Shift folding*: We shift all parts except for the last one, so that the least significant bit of each part lines up with corresponding bit of the last part. We then add the parts together to obtain $f(x)$.
      - Ex: suppose that we have divided the identifier k=12320324111220 into the following parts: x_1 = 123, x_2 = 203, x_3 = 241, x_4 = 112, and x_5 = 20. We would align x_1 through x_4 with x_5 and add. This gives us a hash address of 699.

…………………………….

8.2 STATIC HASHING
8.2.2 Hash Functions

- Folding
  - There are two ways of carrying out this addition.

…………………………

  - *Folding at the boundaries*: reverses every other partition before adding.
    - Ex: suppose the identifier *x* is divided into the same partitions as in shift folding. We would reverse the second and forth partitions, that is $x_2 = 302$ and $x_4 = 211$, and add the partitions. This gives us a hash address of 897.

8.2 STATIC HASHING
8.2.2 Hash Functions

- Digit Analysis
  - Digital analysis is used with static files. A *static file* is one in which all the identifiers are known in advance.
  - Using this method,
    - We first transform the identifiers into numbers using some radix, *r*.
    - We then examine the digits of each identifier, deleting those digits that have the most skewed distribution.
    - We continue deleting digits until the number of remaining digits is small enough to give an address in the range of the hash table.

8.2 STATIC HASHING
8.2.2 Hash Functions

- Converting Keys to Integers
  - Keys need to first be converted to nonnegative integers.

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

```
unsigned int StringToInt (string s)
{//Convert s into a nonnegative int that
 //depends on all characters of s.
     int length=(int)s.length(); //number of characters in s
     unsigned int answer=0;
     if (length%2==1)
     {//length is odd
          answer=s.at(length-1);
          length--;
     }

     //length is now even
     for (int i=0; i<length; i+=2)
     {//do two characters at a time
          answer+=s.at(i);
          answer+=((int)s.at(i+1))<<8;
     }

     return answer;
}
```

8.2 STATIC HASHING
8.2.2 Hash Functions

- Converting Keys to Integers
  - C++ STL provides specializations of the STL template class hash<T>
  - Transform instances of type T into a nonnegative integer of type size_t.

Program 8.2:The specialization hash<string>

```
template<>
class hash<string>{
public:
    size_t operator()(const string theKey) const
    {//Convert theKey to a nonnegative integer.
        unsigned long hashValue=0;
        int length=(int)theKey.length();
        for (int i=0l i<length; i++)
            hashValue=5*hashValue+theKey.at(i);

        return size_t(hahsValue);
    }
};
```

8.2.3 Secure Hash Functions

- One–way property
  - For a given c, it is computationally difficult to find a k such that h(k)=c
- Weak collision resistance
  - When given input x and h, It is computationally difficult to find a x’ such that h(x’)=h(x) (x’ is a synonym for x)
- Strong collision resistance
  - It is computationally difficult to find a pair (x,y) such that h(x)=h(y)

8.2.3 Secure hash Functions

- More practical
  1. h can be applied to a block of data of any size
  2. h produces a fixed-length hash code
  3. h(k) is relatively easy to compute for any given k, making both hardware and software implementations practical.

8.2.3 Secure hash Functions

- Secure hash Algorithm(SHA)
  - Developed at the NIST
  - Input : any message with maximum length less than 2^{64} bits
  - Output : 160-bit code

Program 8.3: SHA algorithm

Step 1: Preprocess the message so that its length is q*512 bits for some integer q.

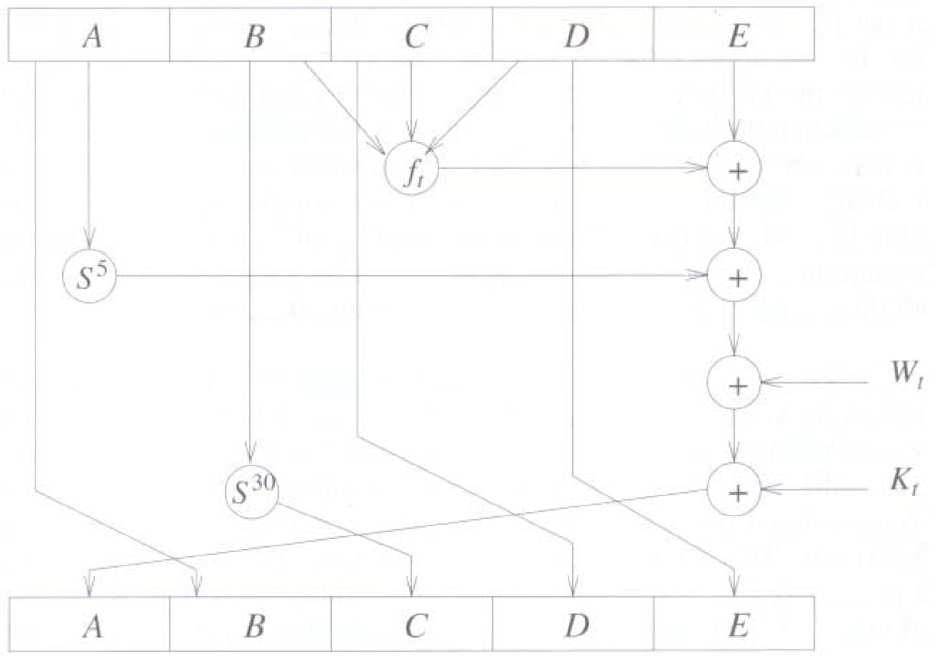
Step 2: Initialize the 160-bit output buffer OB, which comprises five 32-bit registers A, B, C, D, E with appointed value.

Step 3: for (int i=0; i<=q; i++){
Let B_i = ith block of 512 bits of the message; OB=F(OB,Bi); }

Step 4: Output OB

Program 8.3: SHA algorithm

- The function F in Step 3 itself consists of 4 rounds of 20 steps each.



t	=	step number; $0 \le t \le 79$
$f_t(B,C,D)$	=	primitive (bitwise) logical function for step t; e.g., $(B \wedge C) \vee (B \wedge D)$
S^k	=	circular left shift of the 32-bit register by k bits
W_t	=	a 32-bit value derived from B_i
K_t	=	a constant

8.2.4 Overflow Handling

- Two method of overflow handling
  - Open addressing (linear probing, linear open addressing)
  - Chaining

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

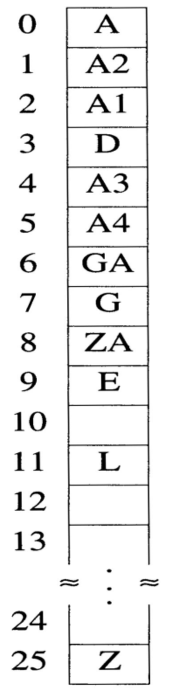
- Linear probing
  - Inserting a new pair with key k
  - Search hash table buckets in the order, ht[h(k)+i]%b, $0 \le i \le b-1$
  - Terminate when reach the first unfilled bucket
  - No unfilled bucket, hash table full and increase the table size => Must change hash function as well.

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

- Ex 8.6)
  - 26-bucket table, one slot per bucket
  - Hash function h(x) = first character of x
  - Identifiers : GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

- identifiers : GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E



0	A
1	A2
2	A1
3	D
4	A3
5	A4
6	GA
7	G
8	ZA
9	E
10	
11	L
12	
13	
⋮	⋮
24	
25	Z

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

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

- Hash table search for the pair with key k

1. Compute h(k)
2. Examine the hash table buckets in the order ht[h(k)], ht[h(k)+1]%b], …,ht[(h(k)+j)%b] until one of the following happens:
    1. The bucket ht[(h(k)+j)%b] has a pair whose key is k; in this case, the desired pair has been found.
    2. ht[h(k)+j] is empty; k is not in the table.
    3. We return to the starting position ht[h(k)]; the table is full and k is not in the table.

Program 8.4: Linear probing

```
template <class K, class E>
pair <K, E>* LinearProbing <K,E>::Get(const K& k)
{//Search the linear probing hash table ht(each bucket has exactly one slot) for k.
    //If a pair with this key is found, return a pointer to this pair; otherwise, return 0.
    int i=h(i); //home bucket
    int j;
    for (j=i; ht[j]&&ht[j]->first!=k;){
        j=(j+1)%b;  //treat the table as circular
        if (j==i) return 0; //back to start point
    }
    if (ht[j]->fisrt==k) return ht[j];
    return 0;
}
```

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

- Analysis of Ex 8.6)
  - The number of buckets examined
    - A – 1, A2 – 2, A1 – 2, D – 1, A3 – 5, A4 – 6, GA – 1, G – 2, ZA – 10, E – 6, L – 1, Z – 1
    - total : 39 buckets examined
    - average : 39/12 = 3.25
  - Approximation of average number of identifier comparison : P
    - $P = (2 - \alpha) / (2 - 2\alpha)$, α = loading density
    - in Ex 8.6) $\alpha = 12/26 = 0.47$
    - $P = 1.5$
    - but in the real case, the average was 3.25

8.2.4 Overflow Handling
8.2.4.1 Open Addressing

- Quadratic probing
  - Quadratic function of i is used as the increment
  - Examining buckets h(k), $(h(k)+i^2)\%b$, and $(h(k)-i^2)\%b$ for $1 \le i \le (b-1)/2$
  - When b is a prime number of the form 4j+3, the quadratic search described above examines every bucket in the table.

8.2.4 Overflow Handling
8.2.4.2 Chaining

- Each bucket has one list of synonyms.
- head node + linked list

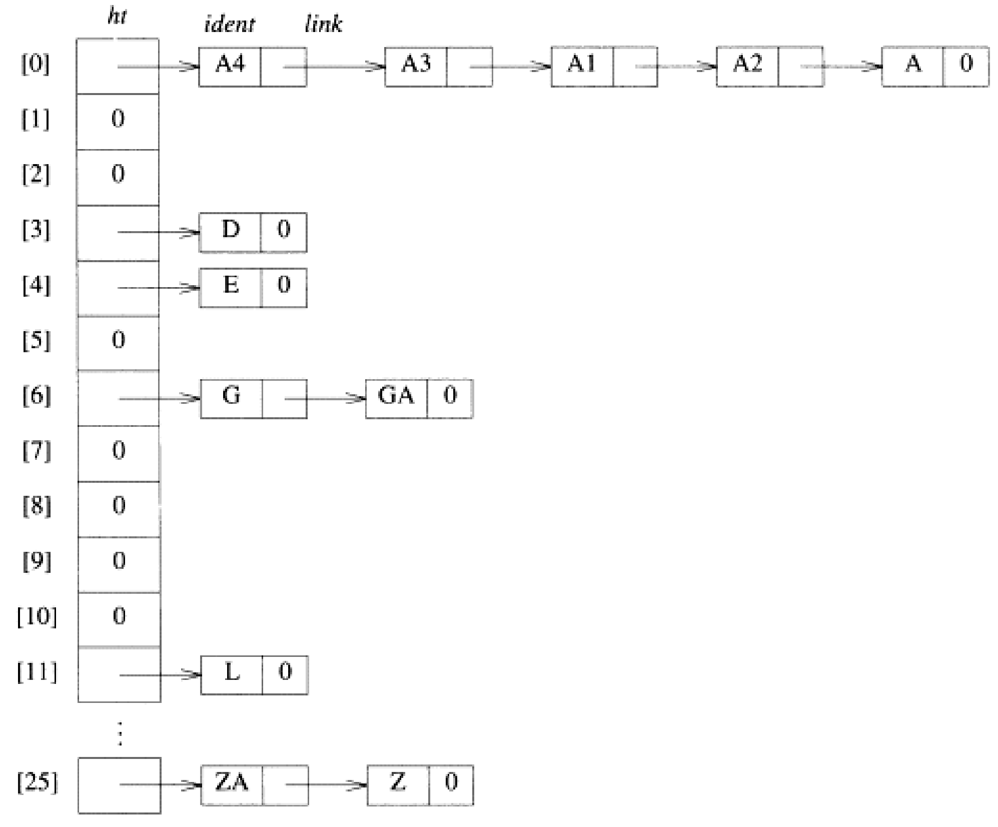
- Array ht[0:b-1] of type ChainNode <pair<K,E>>* : ht[i] pointing to the first node of the chain for bucket i.

Program 8.5: Chain Search

```
template <class K, class E>
pair<K,E>* Chaining<K,E>::Get(const K& 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 0.
    int i=h(i); //home bucket
    //search the chain ht[i]
    for (ChaoinNode <pair<K,E>>* current=ht[i]; current;
                                        current=current->link)
        if (current->data.first==k) return &current->data;
    return 0;
}
```

8.2.4 Overflow Handling
8.2.4.1 Chaining

- identifiers : GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E



Hash table with 26 buckets; each bucket can hold a link.

Figure 8.6: Hash chains corresponding to Figure 8.4

8.2.5 Theoretical Evaluation of Overflow Techniques

- Worst case performance can be very bad. – insertion or search 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.

8.2.5 Theoretical Evaluation of Overflow Techniques

- ht[0:b−1] each bucket having one slot.
- h : uniform hash function with range [0,b−1] a
- If n keys k_1, k_2,⋯,k_n are entered into the hash table, there are b^n distinct hash sequences h(k_1), h(k_2),⋯,k(k_n).
- S_n : expected number of key comparisons needed to locate a randomly chosen k_i, $1 \le i \le n$
- U_n : expected number of key comparisons when a search is made for a key not in the hash table.

8.2.5 Theoretical Evaluation of Overflow Techniques

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

  (1) for linear open addressing

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

8.2.5 Theoretical Evaluation of Overflow Techniques

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

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

$$U_n \approx \frac{1}{1-\alpha} \qquad S_n \approx -\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$$

8.2.5 Theoretical Evaluation of Overflow Techniques

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

  (3) for chaining

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

8.2.5 Theoretical Evaluation of Overflow Techniques

- Proof:
  - When n keys are distributed uniformly over the b possible chains, the expected number in each chain is $n/b=\alpha$. U_n equals the expected number of keys on a chain, α
  - When ith key, k_i, is being entered into the table, the expected number of keys on any chain is $(i-1)/b$.
  - The expected number of comparisons needed to search for k_i after all n keys have been entered is $1+(i-1)/b$

$$S_n = \frac{1}{n}\sum_{i=1}^{n}\left\{1+\frac{i-1}{b}\right\} = 1+\frac{n-1}{2b} \approx 1+\alpha/2$$

8.3 DYNAMIC HASHING
8.3.1 Motivation for Dynamic Hashing

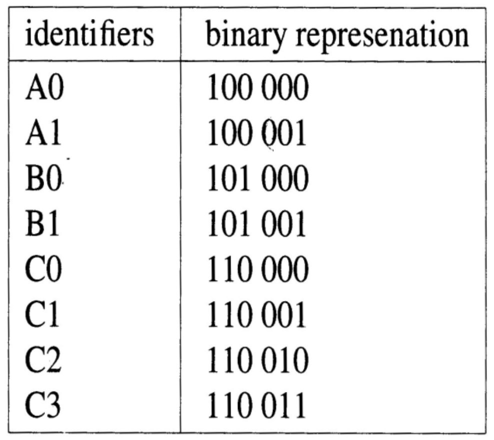
- When an insert causes the loading density to exceed the prespecified threshold, we use array doubling to increase the number of buckets to 2b+1. => hash function divisor changes to 2b+1 => rebuild the hash table by collecting all dictionary pairs.
- Dynamic Hashing (extendible hashing)
  - Reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket

8.3 DYNAMIC HASHING
8.3.1 Motivation for Dynamic Hashing

- Two forms of dynamic hashing
  - use a directory
  - don't use a directory
- Hash function h map keys into non-negative integers
- Range of h is assumed to be sufficiently large
- h(k,p) : Denote the integer formed by the p least significant bits of h(k)

8.3 DYNAMIC HASHING
8.3.1 Motivation for Dynamic Hashing

- Ex) h(k) : transform key into 6-bit non-negative integers.
- A,B,C -> 100, 101, 110
- Digit 0 through 7 -> 000, 001,…, 111



identifiers	binary represenation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011

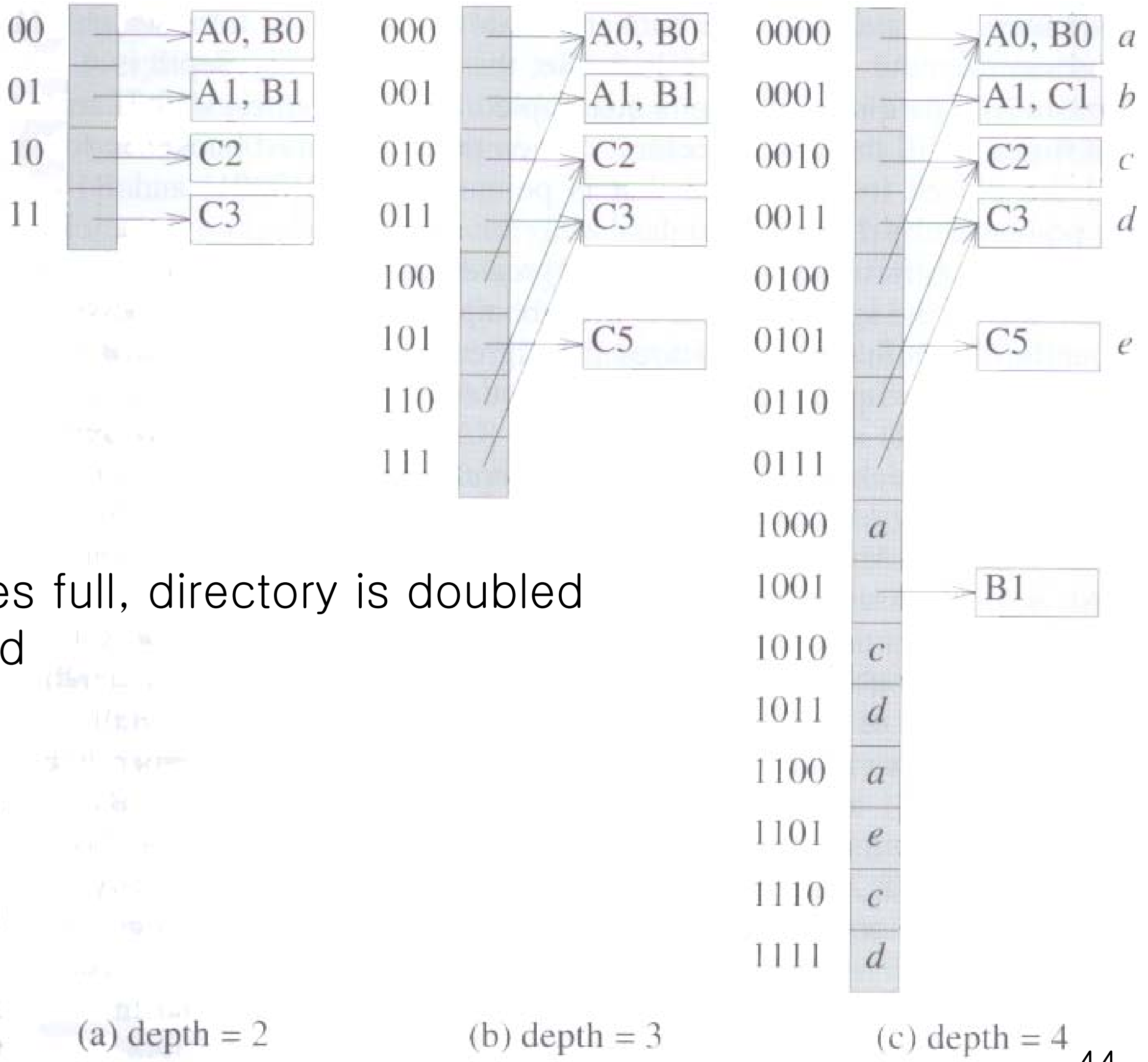
Figure 8.8: Some identifiers that require three bits per character

8.3 DYNAMIC HASHING
8.3.2 Dynamic Hashing Using Directory

- Directory d – pointers to bucket
- Directory depth – number of bits of h(k) used to index the directory
- size of the directory depends on the directory depth
  - ex) h(k,2) – directory size=2^2=4
  - h(k,5) – directory size=2^5=32
- To search for a key k, we merely examine the bucket pointed to by d[h(k,t)], t – directory depth.

Figure 8.7: Dynamic hash tables with directories

identifiers	binary represenation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011



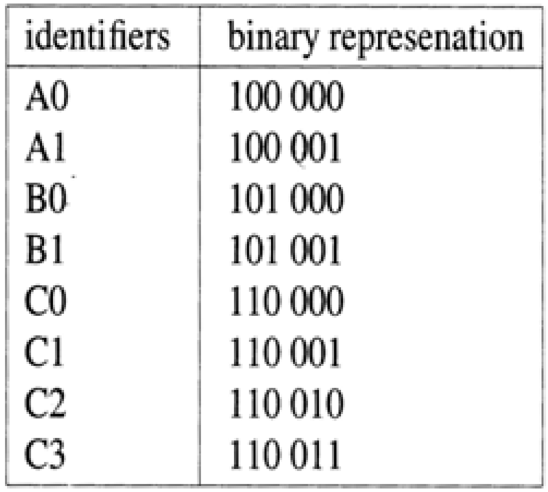
- When bucket becomes full, directory is doubled and pointers are copied

8.3 DYNAMIC HASHING
8.3.3 Directoryless Dynamic Hashing

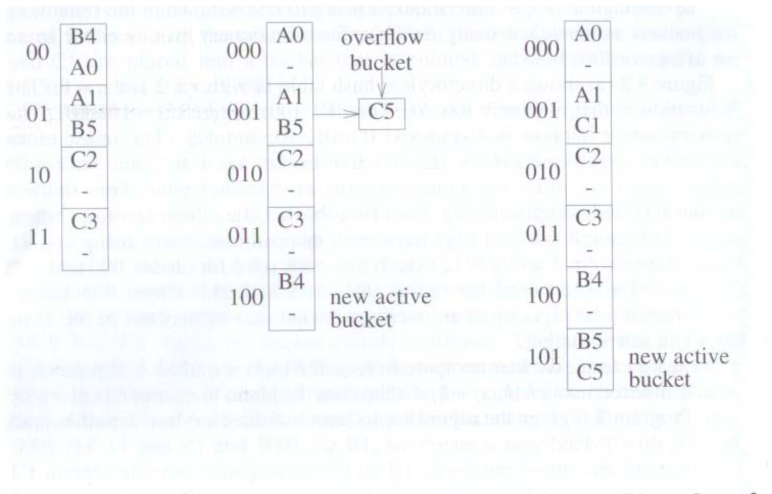
- Array ht of buckets is used.
- Assume this array is as large as possible and so there is no possibility of increasing its size dynamically.
- Only buckets 0 through 2^r+q-1 are active. (q, r – variable to keep track of active bucket, $0 \le q < 2^r$)
- Indexing
  - active buckets $[0, q-1]$, $[2^r, 2^r+q-1]$ are indexed $h(k, r+1)$
  - remaining buckets are indexed $h(k, r)$

8.3 DYNAMIC HASHING
8.3.3 Directoryless Dynamic Hashing

- Searching
  - if (h(k,r)<q) search the chain that begins at bucket h(k, r+1);
  - else search the chain that begins at bucket h(k, r);



identifiers	binary represenation
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C0	110 000
C1	110 001
C2	110 010
C3	110 011



(a) $r = 2, q = 0$ (b) Insert C5, $r = 2, q = 1$ (c) Insert C1, $r = 2, q = 2$

- An overflow is handled by activating bucket 2^r+q
  - incrementing q by 1
  - In case q becomes 2^r, increment r by 1 and reset q to 0

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Consider an application
  - maintain an indexed file
  - there is only one index – just a single key
  - dense index : has an entry for each record in the file
  - updates to the file (insert, delete, change) permitted
  - It is necessary to keep a backup copy of index and file

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Transaction log
  - Log of all updates made since the backup copies were created
- Time needed to recover
  - Function of the sizes of the backup index and file and the size of the transaction log
- More frequent backups reduce time but it is not practical

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Only the file is very large – keeping updated records in a separate file (called differential file) reduce recovery time
  - Master file is unchanged
  - Master index is changed to reflect the position of the most current version of the record with a given key

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- When a differential file is used, the backup file is an exact replica of the master file.
- It is necessary to backup only master index and differential file frequently (relatively small)

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Both the index and the file are very large – differential file scheme does not work.
- Using differential file and differential index
- master index and master file remain unchanged as update are performed
  - differential file contain current version of all changed record
  - differential index – index to differential file

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Access Step

  (a) No differential file
  - Step 1: Search master index for record address.
  - Step 2: Access record from this master file address.
  - Step 3: If this is an update, then update master index, master file, and transaction log.

  (b) Differential file in use
  - Step 1: Search master index for record address.
  - Step 2: Access record from either the master file or the differential file, depending on the address obtained in Step 1.
  - Step 3: If this is an update, then update master index, master file, and transaction log.

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

(c) Differential index and file in use

- Step 1: Search differential index for record address. If the search is unsuccessful, then search the master index.
- Step 2: Access record from either the master file of the differential file, depending on the address obtained in Step 1.
- Step 3: If this is an update, then update differential index, differential file, and transaction log.

8.4 BLOOM FILTERS
8.4.2 Bloom Filter Design

(d) Differential index and file and Bloom filter in use

- Step 1: Query the Bloom filter. If the answer is "maybe," then search differential index for record address. If the answer is "no" or if the differential index search is unsuccessful, then search the master index.
- Step 2: Access record from either the master file or the differential file, depending on the address obtained in Step 1.
- Step 3: If this is an update, then update Bloom filter, differential index, differential file, and transaction log.

8.4 BLOOM FILTERS
8.4.1 An Application–Differential Files

- Comparing with (a),(c)
  - additional disk access are frequently needed, as we will often first query the differential index and then the master index.
  - Differential file is much smaller than the master file – most request are satisfied from the master file.
- Differential index and file are used – backup both high frequency (both are relatively small)

8.4 BLOOM FILTERS
8.4.2 Bloom Filter Design

- Bloom filter
  - m bits of memory, h uniform and independent hash functions $f_1,\cdots,f_h$
  - f_i hash a key k to an integer [1,m]
  - Initially all m filter bits are zero and differential index and file are empty
  - When key k is added to the differential index, bits $f_1(k)$, ⋯, $f_h(k)$ of the filter are set to 1.
  - "Is key k in the differential index?" – bits $f_1(k)$, ⋯, $f_h(k)$ are examined.
    - "maybe" if all these bits are 1
    - "no" otherwise

8.4 BLOOM FILTERS
8.4.2 Bloom Filter Design

- Probability of a filter error
  - n records, u updates (none of these is insert or delete)
  - record keys – uniformly distributed
  - probability update request for record i = 1/n, $1 \le i \le n$
  - probability particular update does not modify record i = $1-1/n$
  - probability none of the u update modify record i = $(1-1/n)^u$
  - expected number of unmodified record = $n(1-1/n)^u$
  - probability (u+1)'st update is for an unmodified record = $(1-1/n)^u$

8.4 BLOOM FILTERS
8.4.2 Bloom Filter Design

- Probability of a filter error
  - bit i of the Bloom filter, hash function f_j(uniform hash function) $1 \le j \le h$
  - k : the key corresponding to one of the u update
  - probability $f_j(k) \neq i$ => $1-1/m$
  - probability $f_j(k) \neq i$ for all h hash functions => $(1-1/m)^h$
  - If only update, probability bit i of the filter is zero => $(1-1/m)^h$
  - probability bit i is zero following u updates => $(1-1/m)^{uh}$

8.4 BLOOM FILTERS
8.4.2 Bloom Filter Design

- Probability of a filter error
  - probability of a filter = $(1-(1-1/m)^{uh})^h$
  - P(u) : probability arbitrary query made after u updates results in a filter error => $(1-1/n)^u(1-(1-1/m)^{uh})^h$
  - $(1-1/x)^q \sim e^{-q/x}$
  - $P(u) \sim e^{-u/n}(1-e^{-uh/m})^h$ (large x,n,m)
  - In most application, m, n fixed but h can be designed.
- The fittest h = $(\log_e 2)m/u \sim 0.693m/u$