Chapter 10

Hashing
(one session)



## Objectives

- Linear Searching: O(n)
- Binary Searching: O(logn)
- Is there a more efficient way for searching?

**Discuss the following topics:**

- Hashing
- Hash Functions
- Collision Resolution
- Deletion
- Perfect Hash Functions



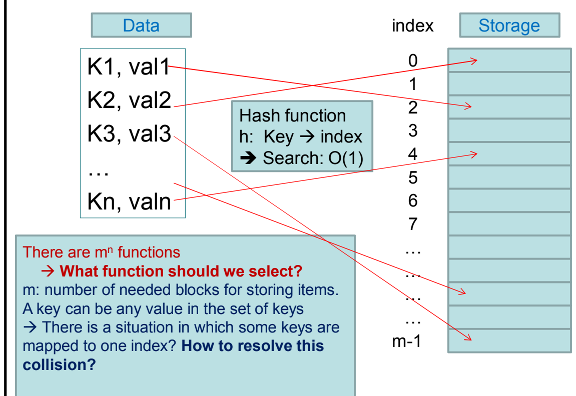
## Objectives (continued)

Discuss the following topics:

- Hash Functions for Extendible( extensible) Files
- Hashing in `java.util`
- Case Study: Hashing with Buckets



## Hashing



## Hashing

- To find a function (*h*) that can transform a particular key (*K*) (a string, number or record) into an index in the table used for storing items of the same type as *K*, the function *h* is called a **hash function**
- If *h* transforms different keys into different numbers, it is called a **perfect hash function**
- To create a perfect hash function, the table has to contain at least the same number of positions as the number of elements being hashed



## 10.1- Common Hash Functions

- The **division** method is the preferred choice for the hash function if **very little** is known about the keys

  *TSize =sizeof(table)*, as in *h(K) = K mod TSize*
- In the **folding** method, the key is divided into several parts which are combined or folded together and are often transformed in a certain way to create the target address: h(K) = sum of parts *mod* TSize

Example1: K= 123-45-6789 → sum 3 parts:123 + 45 + 6789= 1368

→ h(K)=1367 mod TSize

Example2: K= 123-45-6789 → Sum 5 parts:12 + 34+56+78+9 = 189

→ h(K) = 189 mod TSize

Example3: K="abcd" → h(K)= "abcd" xor "efgh"



## Hash Functions (continued)

- In the **mid-square** method, the key is *squared* and the middle or *mid* part of the result is used as the address

Example: TSize= 1024= $2^{10}$, K= 3121 → $K^2$=9 740 641

$K^2$= 1001010 0101000010 1100001 → h(K) = $0101000010_2$ = 32

- In the **extraction** method, only a part of the key is used to compute the address

Example: Concatenate two beginning digits and two ending digits:

K = 123-45-6789, h(K) = 1289 *mod* TSize



## Hash Functions (continued)

- Using the **radix transformation**, the key *K* is transformed into another number base; *K* is expressed in a numerical system using a different radix

Example: Use the base of 9

K = $345_{10}$ = $423_9$ → , f(K) = 423 *mod* Tsize ( =23 if Tsize=100 )

# 10.2- Collision Resolution

**Open Addressing Method**

- when a key collides with another key, the collision is resolved **by finding an available table entry** other than the position (address) to which the colliding key is originally hashed. Common methods:
  - **Linear probing**(dò tuyến tính), *t*he simplest method.
    Probing p(i) with i=1, 2, 3,…→ norm(h(K) + p(i))
    norm is a function as a normalizer. A common normalizer is h(h(K) + p(i))
    *Example:* p(i)= i, h(K) = (h(K) + i) mod TSize
  - ***Quadratic probing*** *(dò bậc 2)* , p(i) = ai$^2$ norm(h(K) + ai$^2$)
    *Example:* p(i)= ± i$^2$, h(K) = (h(K) ± i$^2$) mod TSize



# Collision Resolution (continued)

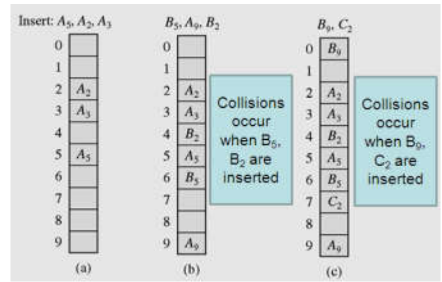



**Figure 10-1 Resolving collisions with the linear probing method. Subscripts indicate the home positions of the keys being hashed.**



# Collision Resolution (continued)

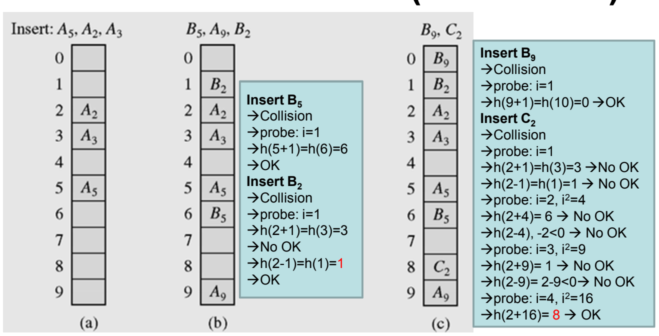



**Figure 10-2 Using quadratic probing for collision resolution**
h(K) = (h(K) ± i$^2$) mod 10



# Collision Resolution (continued)

| | Linear Probing | Quadratic Probing[a] | Double Hashing |
|---|---|---|---|
| successful search | $\frac{1}{2}\left(1+\frac{1}{1-\mathrm{LF}}\right)$ | $1-\ln(1-\mathrm{LF})-\frac{\mathrm{LF}}{2}$ | $\frac{1}{\mathrm{LF}}\ln\frac{1}{1-\mathrm{LF}}$ |
| unsuccessful search | $\frac{1}{2}\left(1+\frac{1}{(1-\mathrm{LF})^2}\right)$ | $\frac{1}{1-\mathrm{LF}}-\mathrm{LF}-\ln(1-\mathrm{LF})$ | $\frac{1}{1-\mathrm{LF}}$ |

Load Factor $\mathrm{LF} = \frac{\text{number of elements in the table}}{\text{table size}}$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

**Figure 10-3 Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches (Knuth, 1998)**

# Ý nghĩa Load Factor

- Tham số **α** $\alpha = \frac{N}{SIZE}$
- Băm đ/c mở: mức độ đầy (load factor)
  - **α** tăng thì khả năng va chạm tăng
  - Khi thiết kế, cần đánh giá max của N để lựa chọn SIZE
  - **α** không nên vượt quá 2/3
  - Băm dây chuyền: độ dài trung bình của một dây chuyền

| | | Băm đ/c mở, Thăm dò tuyến tính | Băm đ/c mở, Thăm dò bình phương | Băm dây chuyền |
|---|---|---|---|---|
| Thời gian trung bình | Tìm kiếm thành công | $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ | $\frac{-\ln(1-\alpha)}{\alpha}$ | $1+\frac{1}{\alpha}$ |
| | Tìm kiếm thất bại | $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ | $\frac{1}{1-\alpha}$ | α |



# Collision Resolution (continued)

**Chaining Method**

- *Keys do not have to stored in table itself,* each position of the table is associated with a linked list or **chain** of structures whose `info` fields store keys or references to keys
- This method is called **separate chaining**, and a table of references (pointers) is called a **scatter table**



# Collision Resolution (continued)

h(K) → index of a linked list of elements having the same value of hask function.

K→ h(K) → index →traverse the appropriate list to find the element having this key.

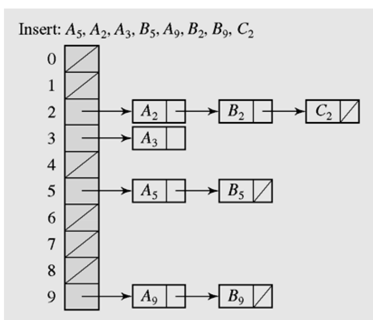



**Figure 10-5 In chaining, colliding keys are put on the same linked list**



# Collision Resolution (continued)

- A version of chaining called **coalesced hashing**- băm theo nhóm sử dụng array- (or **coalesced chaining**) combines linear probing with chaining
- An overflow area known as a **cellar** can be allocated to store keys for which there is no room in the table

## Collision Resolution (continued)

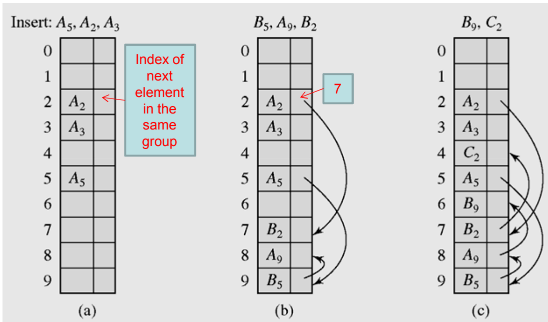



**Figure 10-6 Coalesced hashing puts a colliding key in the last available position of the table**



## Collision Resolution (continued)

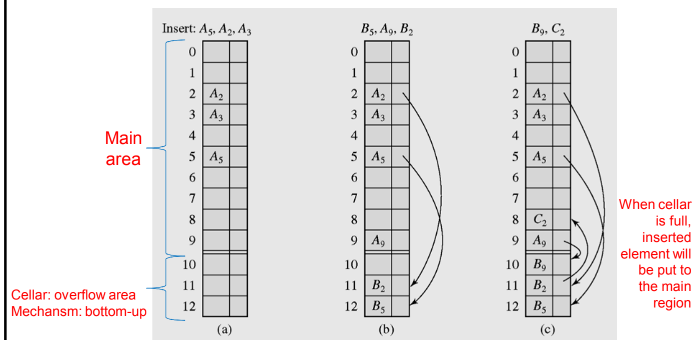



**Figure 10-7 Coalesced hashing that uses a cellar**



## Collision Resolution (continued)

### Bucket Addressing

- To store colliding elements in the same position in the table can be achieved by associating a bucket with each address
- A **bucket** is a block of space large enough to store multiple items



## Collision Resolution (continued)

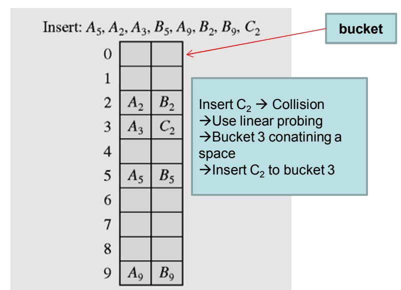



**Figure 10-8 Collision resolution with buckets ( bucket=2) and linear probing method**

## Collision Resolution (continued)

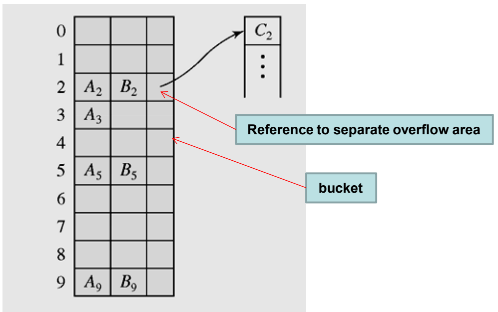



**Figure 10-9 Collision resolution with buckets and overflow area**



## Ý nghĩa Load Factor

- Tham số **α** $\alpha = \frac{N}{SIZE}$
- Băm đ/c mở: mức độ đầy (load factor)
  - **α** tăng thì khả năng va chạm tăng
  - Khi thiết kế, cần đánh giá max của N để lựa chọn SIZE
  - **α** không nên vượt quá 2/3
  - Băm dây chuyền: độ dài trung bình của một dây chuyền

| | | Băm đ/c mở, Thăm dò tuyến tính | Băm đ/c mở, Thăm dò bình phương | Băm dây chuyền |
|---|---|---|---|---|
| Thời gian trung bình | Tìm kiếm thành công | $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ | $\frac{-\ln(1-\alpha)}{\alpha}$ | $1+\frac{1}{\alpha}$ |
| | Tìm kiếm thất bại | $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ | $\frac{1}{1-\alpha}$ | $\alpha$ |



## 10.3- Deletion

- Structure and collision resolution of the hash table will decide the way by which its elements are deleted. Thay can be
  - Linear search for deletion
  - Linear search to locate the linked list of the subgroup then delete an element in this linked list.
  - Linear search to locate the subgroup then delele an element in this subgroup, update references to next elements in the same subgroup.



## Deletion

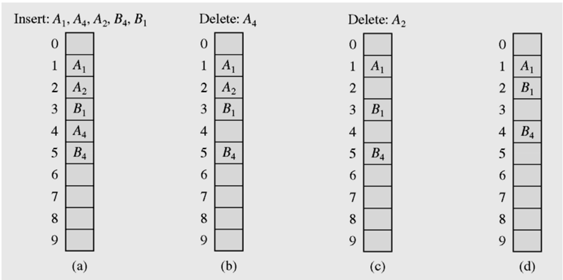
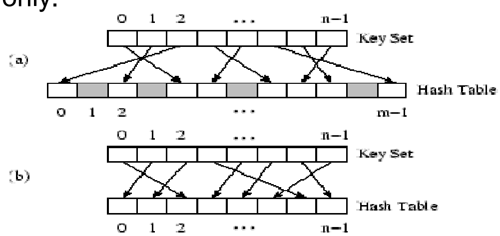



**Figure 10-10 Linear search in the situation where both insertion and deletion of keys are permitted**

## 10.4- Perfect Hash Functions (*)

- If a function requires only as many cells in the table as the number of data so that no empty cell remains after hashing is completed, it is called a **minimal perfect hash function**
- (*) You need to know essential features on this part only.






## Perfect Hash Functions

### Cichelli's method

- **Cichelli's method** is an algorithm to construct a minimal perfect hash function
- It is used to hash a relatively small number of reserved words
- Where *g* is the function to be constructed

*h*(*word*) = (*length*(*word*) + *g*(*firstletter*(*word*)) + *g*(*lastletter*(*word*))) mod *TSize*

- The algorithm has three parts:
  – Computation of the letter occurrences
  – Ordering the words
  – Searching



## Perfect Hash Functions

### Cichelli's method

- Choose a value of *max;*
- Compute the number of occurrences of each first letter and last letter in the set of all words.
- Order all words in accordance to the frequency of occurrence of yhe first and the last letters;



### Cichelli's Method



This is used primarily when it is necessary to hash a relatively small collection of keys, such as the set of reserved words for a programming language.

The basic formula is:

```
h(S) = S.length() + g(S[0]) + g(S[S.length()-1])
```

where `g()` is constructed using Cichelli's algorithm so that h() will return a different hash value for each word in the set.

The algorithm has three phases:

- computation of the letter frequencies in the words
- ordering the words
- searching

## Cichelli's Method



Suppose we need to hash the words in the list below:

| |
|---|
| calliope |
| clio |
| erato |
| euterpe |
| melpomene |
| polyhymnia |
| terpsichore |
| thalia |
| urania |

Determine the frequency with which each first and last letter occurs:

| letter: | e | a | c | o | t | m | p | u |
|---|---|---|---|---|---|---|---|---|
| freq: | 6 | 3 | 2 | 2 | 2 | 1 | 1 | 1 |

Score the words by summing the frequencies of their first and last letters, and then sort them in that order:

| | |
|---|---|
| calliope | 8 |
| clio | 4 |
| erato | 8 |
| euterpe | 12 |
| melpomene | 7 |
| polyhymnia | 4 |
| terpsichore | 8 |
| thalia | 5 |
| urania | 4 |

| |
|---|
| euterpe |
| calliope |
| erato |
| terpsichore |
| melpomene |
| thalia |
| clio |
| polyhymnia |
| urania |

## Cichelli's Method



Finally, consider the words in order and define `g(x)` for each possible first and last letter in such a way that each of the words will have a distinct hash value:

| word | g_value assigned | h(word) | table slot |
|---|---|---|---|
| euterpe | e-->0 | 7 | 7 ok |
| calliope | c-->0 | 8 | 8 ok |
| erato | o-->0 | 5 | 5 ok |
| terpsichore | t-->0 | 11 | 2 ok |
| melpomene | m-->0 | 9 | 0 ok |
| thalia | a-->0 | 6 | 6 ok |
| clio | none | 4 | 4 ok |
| polyhymnia | p-->0 | 10 | 1 ok |
| urania | u-->0 | 6 | 6 reject |
| | u-->1 | 7 | 7 reject |
| | u-->2 | 8 | 8 reject |
| | u-->3 | 9 | 0 reject |
| | u-->4 | 10 | 1 reject |

## Cichelli's Method



Cichelli's method imposes a limit on the search at this point (we're assuming it's 5 steps), and so we back up to the previous word and redefine the mapping there:

| word | g_value assigned | h(word) | table slot |
|---|---|---|---|
| polyhymnia | p-->0 | 10 | 1 reject |
| | p-->1 | 11 | 2 reject |
| | p-->2 | 12 | 3 |
| urania | u-->0 | 6 | 6 reject |
| | u-->1 | 7 | 7 reject |
| | u-->2 | 8 | 8 reject |
| | u-->3 | 9 | 0 reject |
| | u-->4 | 10 | 1 ok |

So, if we define `g()` as determined above, then `h()` will be a minimal perfect hash function on the given set of words.

The primary difficulty is the cost, because the search phase can degenerate to exponential performance, and so it is only practical for small sets of words.

# Perfect Hash Functions

## Cichelli's method

Set of nine Muses – 9 nhà thơ
Calliope, Clio, Erato, Euterpe, Melpomene, Polyhymia, Terpsichore, Thalia, Urania

Occurrence of each letter at the first and the last position:
E (6), A(3), C(2), O(2), T(2), M(1), P(1), U(1).

According to frequencies, list may be ordered as :
**Euterpe, Calliope, Erato, Terpsichore, Melpomene, Thalia, Clio, Polyhymnia, Urania**

| | | | reserved hash values |
|---|---|---|---|
| Euterpe | E = 0 | h = 7 | {7} |
| Calliope | C = 0 | h = 8 | {7 8} |
| Erato | O = 0 | h = 5 | {5 7 8} |
| Terpsichore | T = 0 | h = 2 | {2 5 7 8} |
| Melpomene | M = 0 | h = 0 | {0 2 5 7 8} |
| Thalia | A = 0 | h = 6 | {0 2 5 6 7 8} |
| Clio | | h = 4 | {0 2 4 5 6 7 8} |
| Polyhymnia | P = 0 | h = 1 | {0 1 2 4 5 6 7 8} |
| Urania | U = 0 | h = 6* | {0 1 2 4 5 6 7 8} |
| Urania | U = 1 | h = 7* | {0 1 2 4 5 6 7 8} |
| Urania | U = 2 | h = 8* | {0 1 2 4 5 6 7 8} |
| Urania | U = 3 | h = 0* | {0 1 2 4 5 6 7 8} |
| Urania | U = 4 | h = 1* | {0 1 2 4 5 6 7 8} |
| Polyhymnia | P = 1 | h = 2* | {0 2 4 5 6 7 8} |
| Polyhymnia | P = 2 | h = 3 | {0 2 3 4 5 6 7 8} |
| Urania | U = 0 | h = 6* | {0 2 3 4 5 6 7 8} |
| Urania | U = 1 | h = 7* | {0 2 3 4 5 6 7 8} |
| Urania | U = 2 | h = 8* | {0 2 3 4 5 6 7 8} |
| Urania | U = 3 | h = 0* | {0 2 3 4 5 6 7 8} |
| Urania | U = 4 | h = 1 | {0 1 2 3 4 5 6 7 8} |

**Figure 10-11 Subsequent invocations of the searching procedure with *Max* = 4 in Cichelli's algorithm assign the indicated values to letters and to the list of reserved hash values. The asterisks indicate failures.**

## Perfect Hash Functions

**The FHCD Algorithm**

- The FHCD algorithm, an extension of Cechelli's approach, devised by Thomas Sager, searches for a minimal perfect hash function of the form (modulo *TSize*), where *g* is the function to be determined by the algorithm

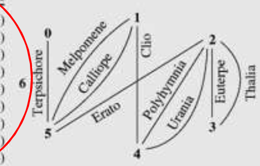
  $h(word) = h_0(word) + g(h_1(word)) + g(h_2(word))$



## The FHCD Algorithm (continued)

| Value of: | $h_0$ | $h_1$ | $h_2$ |
|---|---|---|---|
| Calliope | (0 | 1 | 5) |
| Clio | (7 | 1 | 4) |
| Erato | (3 | 2 | 5) |
| Euterpe | (6 | 2 | 3) |
| Melpomene | (3 | 1 | 5) |
| Polyhymnia | (8 | 2 | 4) |
| Terpsichore | (8 | 0 | 5) |
| Thalia | (8 | 2 | 3) |
| Urania | (0 | 2 | 4) |

(a)




(b)

| Level | Node | Arcs |
|---|---|---|
| 0 | 2 | |
| 1 | 5 | Erato |
| 2 | 1 | Calliope, Melpomene |
| 3 | 4 | Clio, Polyhymnia, Urania |
| 4 | 3 | Euterpe, Thalia |
| 5 | 0 | Terpsichore |

(c)

| Level | Vertex | g-value | | |
|---|---|---|---|---|
| 0 | 2 | 2 | | |
| 1 | 5 | 6 | h(Erato) | = (3 + 2 + 6) % 9 = 2 |
| 2 | 1 | 4 | h(Calliope) | = (0 + 4 + 6) % 9 = 1 |
| 2 | 1 | 4 | h(Melpomene) | = (3 + 4 + 6) % 9 = 4 |
| 3 | 4 | 2 | h(Clio) | = (7 + 6 + 2) % 9 = 6 |
| 3 | 4 | 2 | h(Polyhymnia) | = (8 + 6 + 2) % 9 = 7 |
| 3 | 4 | 2 | h(Urania) | = (0 + 6 + 2) % 9 = 8 |
| 4 | 3 | 4 | h(Euterpe) | = (6 + 2 + 4) % 9 = 3 |
| 4 | 3 | 4 | h(Terpsichore) | = (8 + 2 + 4) % 9 = 5 |
| 4 | 3 | 4 | h(Thalia) | = (8 + 4 + 6) % 9 = 0 |

(d)

| Function g | |
|---|---|
| 0 | 4 |
| 1 | 4 |
| 2 | 2 |
| 3 | 4 |
| 4 | 2 |
| 5 | 6 |

(e)

**Figure 10-12 Applying the FHCD algorithm to the names of the nine Muses**



## Review

- Static Hashing
- Dynamic Hashing
  - Extendible hashing
  - Linear Hashing



## 10.5- Hash Functions for Extendible Files (*)

- **File=table.**
- **Expandable hashing**, **dynamic hashing**, and **extendible hashing methods** distribute keys among buckets in a similar fashion
- Data → h(Data) → index → buckets[index]
- The main difference is the structure of the index (directory)
- In expandable hashing and dynamic hashing, a binary tree is used as an index of buckets
- In extendible hashing, a directory of records is kept in a table

## Extendible Hashing

- **Extendible hashing** accesses the data stored in buckets indirectly through an index that is dynamically adjusted to reflect changes in the file
- Extendible hashing allows the file to expand without reorganizing it, but requires storage space for an index
- Values returned by such a hash function are called **pseudokeys**

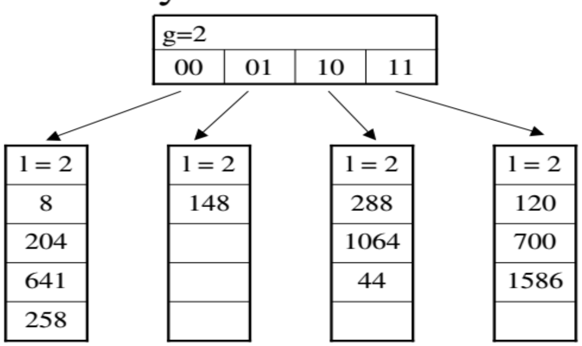


## Extendible Hashing Example

- Suppose that g=2 and bucket size = 4.
- Suppose that we have records with these keys and hash function h(key) = key mod 64:

| key | h(key) = key mod 64 | bit pattern |
|---|---|---|
| 288 | 32 | 100000 |
| 8 | 8 | 001000 |
| 1064 | 40 | 101000 |
| 120 | 56 | 111000 |
| 148 | 20 | 010100 |
| 204 | 12 | 001100 |
| 641 | 1 | 000001 |
| 700 | 60 | 111100 |
| 258 | 2 | 000010 |
| 1586 | 50 | 110010 |
| 44 | 44 | 101010 |

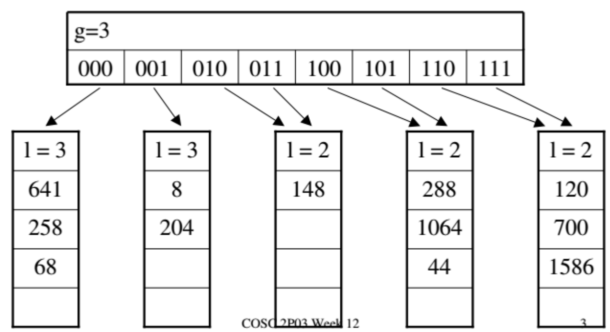
## Extendible Hashing Example – directory and bucket structure





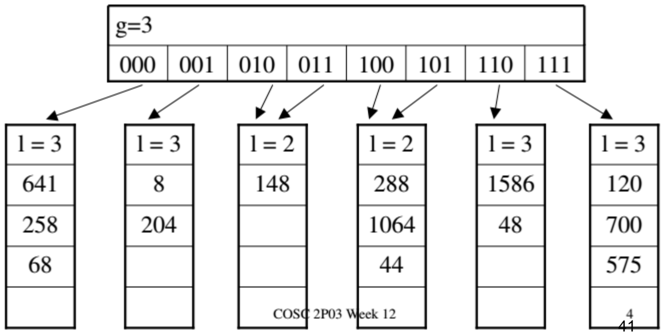

## Bucket and directory split

- Insert 68
- 68 mod 64 = 4 = 000100



# Bucket split – no directory split

- Insert 48 and 575
- 48 mod 64 = 48 = 110000
- 575 mod 64 = 63 = 111111



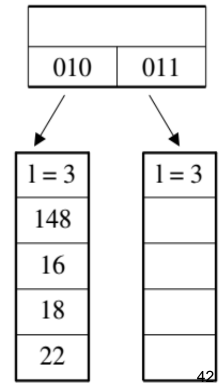



# Multiple splits

- Insert 16, 18, 22, 23
- 16 mod 64 = 16 = 010000
- 18 mod 64 = 18 = 010010
- 22 mod 64 = 22 = 010110
- 23 mod 64 = 23 = 010111

Setting l=3 gives this intermediate (partial) picture…

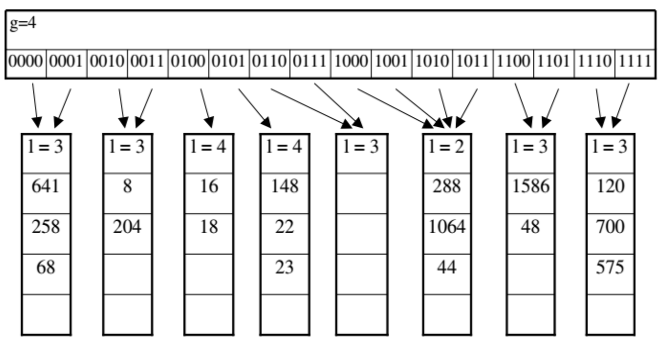
Continue to next page…






# Multiple splits, continued

- Setting l=4 (and thus g=4) gives this final result…






# Extendible Hashing (continued)

Function h generates patterns of 5 bits

2 leftmost bits present the position in the directory containing the reference to the bucket containing the key. Number of bits is called **local depth (2 in this example).**

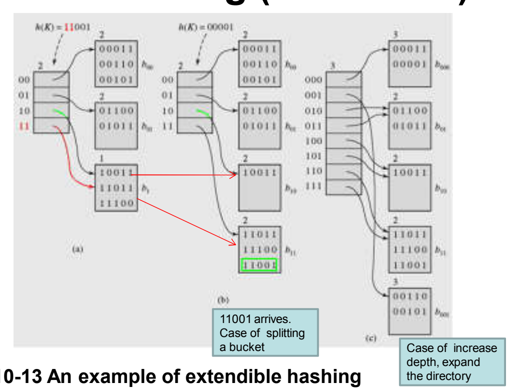



**Figure 10-13 An example of extendible hashing**

# Linear Hashing

- With this method, no index is necessary because new buckets generated by splitting existing buckets are always added in the same linear way, so there is no need to retain indexes
- A bucket is full when its loading factor exceeds a certain level. This bucket will be splitted.
- A reference **split** indicates which bucket is to be split next
- **After the bucket is divided, the keys in this bucket are distributed between this bucket and the newly created bucket, which is added to the end of the table**



# Linear Hashing (continued)

TSize=3

Some hash functions may be applied.

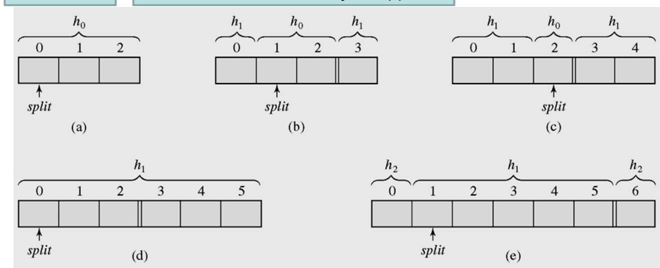


**Figure 10-14 Splitting buckets in the linear hashing technique**



# Linear Hashing (continued)

TSize=3

$H_0(K)$ = K mod TSize

$H_1(K)$ = K mod 2*Tsize

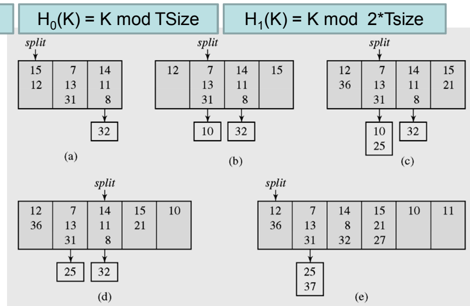


**Figure 10-15 Inserting keys to buckets and overflow areas with the linear hashing technique**



# `10.6- Hashing in java.util` `The HashMap class`

- `HashMap` is an implementation of the interface `Map`
- A map is a collection that holds pairs (key, value) or entries
- A **hash map** is a **collection of singly linked lists** (buckets); that is, chaining is used as a collision resolution technique
- In a hash map, both null values and null keys are permitted

# HashMap (continued)

| Method | Operation |
|---|---|
| `void clear()` | Remove all the objects from the hash map. |
| `Object clone()` | Return a copy of the hash map without cloning its elements. |
| `boolean containsKey(Object key)` | Return `true` if the hash map contains the object `key`. |
| `boolean containsValue(Object val)` | Return `true` if the hash map contains the object `val`. |
| `Set entrySet()` | Return a set containing all the pairs (key, value) in the hash map. |
| `boolean equals(Object ob)` | Return `true` if the current hash map and object `ob` are equal (inherited). |
| `int hashCode()` | Return the hash code for the hash map (inherited). |
| `Object get(Object key)` | Return the object associated with `key`. |
| `HashMap()` | Create an empty hash map with initial capacity equal to 16 and the load factor equal to .75. |
| `HashMap(int ic)` | Create an empty hash map with initial capacity `ic` and the load factor equal to .75; throw `IllegalArgumentException` `if ic < 0`. |
| `HashMap(int ic, float lf)` | Create an empty hash map with initial capacity `ic` and the load factor `lf`; throw `IllegalArgumentException` if `ic < 0` or `lf ≤ 0`. |

**Figure 10-16 Methods in class `HashMap` including three inherited methods**



# HashMap (continued)

| | |
|---|---|
| `HashMap(Map m)` | Create a hash map with copies of elements from map `m`; throw `NullPointerException` if `m` is null. |
| `boolean isEmpty()` | Return `true` if the hash map contains no elements, `false` otherwise. |
| `Set keySet()` | Return a set containing all the keys of the hash map. |
| `Object put(Object key, Object val)` | Put the pair (`key`, `val`) in the hash map; return a value associated with `key` if there is any in the hash map, `null` otherwise. |
| `void putAll(Map m)` | Add objects from `map` to the current hash map; throw `NullPointerException` if `m` is null. |
| `void rehash()` | A protected method to increase the capacity of the hashtable; the method is called automatically when the number of keys in the hashtable is greater than the product of the load factor and the current capacity. |
| `Object remove(Object key)` | Remove the pair (`key`, corresponding value) from the hash map and return the value associated currently with `key` in the hash map. |
| `int size()` | Return the number of objects in the hash map. |
| `String toString()` | Return a string representation of the hash map that contains the string representation of all the elements (inherited). |
| `Collection values()` | Return a collection with all the values contained in the hash map. |

**Figure 10-16 Methods in class `HashMap` including three inherited methods (continued)**



# HashMap (continued)

This example (in textbook) demonstates how to use the **HashMap** class to manage a list of **person** < name, age,hashCode> in which the **hasCode is the sum of charcter codes in the field name**.

Click to go the HashSet class

```
import java.io.*;
import java.util.HashMap;

class Person {
    private String name;
    int age;
    private int hashcode = 0;
    public Person(String n, int a) {
        name = n; age = a;
        for (int i = 0; i < name.length(); i++)
            hashcode += name.charAt(i);
    }
    public Person() {
        this("",0);
    }
    public boolean equals(Object p) {
        return name.equals(((Person)p).name);
    }
    public int hashCode() {
        return hashcode;
    }
    public String toString() {
        return "(" + name + "," + age + ")";
    }
}
```

**Figure 10-17 Demonstrating the operation of the methods in class `HashMap`**



# HashMap (continued)

```
class TestHashMap {
    public static void main(String[] a) {
        HashMap cities = new HashMap();
        cities.put(new Person("Gregg",25),"Pittsburgh");
        cities.put(new Person("Ann",30),"Boston");
        cities.put(new Person("Bill",20),"Belmont");
        System.out.println(cities);
        // {(Ann,30)=Boston, (Gregg,25)=Pittsburgh, (Bill,20)=Belmont}
        cities.put(new Person("Gregg",30),"Austin");
        System.out.println(cities);
        // {(Ann,30)=Boston, (Gregg,25)=Austin, (Bill,20)=Belmont}
        System.out.println(cities.containsKey(new Person("Ann",30)));
        // true
        System.out.println(cities.containsValue("Boston"));
        // true
        System.out.println(cities.size());                    // 3
```

**Figure 10-17 Demonstrating the operation of the methods in class `HashMap` (continued)**

## `HashMap` (continued)

```
        System.out.println(cities.get(new Person("Ann",30)));   // Boston
        System.out.println(cities.entrySet());
        // [(Ann,30)=Boston, (Gregg,25)=Austin, (Bill,20)=Belmont]
        System.out.println(cities.values());
        // [Boston, Austin, Belmont]
        System.out.println(cities.keySet());
        // [(Ann,30), (Gregg,25), (Bill,20)]
        System.out.println(cities.remove(new Person("Bill",20)));
        // Belmont
        System.out.println(cities);
        // [(Ann,30), (Gregg,25)]
        cities.put(null,"Nashville");
        cities.put(new Person("Kay",44),null);
        System.out.println(cities);
        // [(Ann,30)=Boston, (Gregg,25)=Austin, (Kay,44)=null,
        // null=Nashville]
        System.out.println(cities.get(new Person("Kay",44)));
        // null
        System.out.println(cities.get(new Person("Stan",55)));
        // null
        System.out.println(cities.containsKey(new Person("Kay",44)));
        // true
        System.out.println(cities.containsKey(new Person("Stan",55)));
        // false
    }
}
```

**Figure 10-17 Demonstrating the operation of the methods in class `HashMap` (continued)**



## The `HashSet` class

- `HashSet` is another implementation of a set (an object that stores **unique elements**)
- Class hierarchy in `java.util` for `HashSet` is:

`Object → AbstractCollection → AbstractSet → HashSet`

- `HashSet` is implemented in terms of `HashMap`

```
public HashSet() {
    map = new HashMap();
}
```



## `HashSet` (continued)

| Method | Operation |
|---|---|
| `boolean add(Object ob)` | Add `ob` to the hash set if it is not already there; return `true` if `ob` was added. |
| `boolean addAll(Collection c)` | Add all the elements from the collection `c` to the hash set; return `true` if the hash set was modified`()`; throw `NullPointerException` if `c` is null (inherited). |
| `void clear()` | Remove all the objects from the hash set. |
| `Object clone()` | Return a copy of the hash set without cloning its elements. |
| `boolean contains(Object ob)` | Return `true` if the hash set contains the object `ob`. |
| `boolean containsAll(Collection c)` | Return `true` if the hash set contains all elements in the collection `c`; throw `NullPointerException` if `c` is null (inherited). |
| `boolean equals(Object ob)` | Return `true` if the current hash set and object `ob` are equal (inherited). |
| `int hashCode()` | Return the hash code for the hash set (inherited). |
| `HashSet()` | Create an empty hash set with initial capacity equal to 16 and the load factor equal to .75. |
| `HashSet(int ic)` | Create an empty hash set with initial capacity `ic` and the load factor equal to .75; throw `IllegalArgumentException` if `ic < 0`. |

**Figure 10-18 Methods in class `HashSet` including some inherited methods**



## `HashSet` (continued)

| | |
|---|---|
| `HashSet(int ic, float lf)` | Create an empty hash set with initial capacity `ic` and the load factor `lf`; throw `IllegalArgumentException` if `ic < 0` or `lf ≤ 0`. |
| `HashSet(Collection c)` | Create a hash set with copies of elements from `c`; throw `NullPointerException` if `c` is null. |
| `boolean isEmpty()` | Return `true` if the hash set contains no elements, `false` otherwise. |
| `boolean iterator()` | Return an iterator over the elements in the hash set; the iteration order can change over time. |
| `boolean remove(Object ob)` | Remove `ob` from the hash set and return `true` if `ob` was in the hash set. |
| `boolean removeAll(Collection c)` | Remove from the hash set all elements contained in collection `c`; return `true` if any element was removed; throw `NullPointerException` if `c` is null (inherited). |
| `boolean retainAll(Collection c)` | Remove from the hash set all elements that are not in the collection `c`; return `true` if any element was removed; throw `NullPointerException` if `c` is null (inherited). |
| `Object[] toArray()` | Copy all elements from the hash set to a newly created array and return the array (inherited). |

**Figure 10-18 Methods in class `HashSet` including some inherited methods (continued)**

## HashSet (continued)

| | |
|---|---|
| `Object[] toArray(Object a[])` | Copy all elements from the hash set to the array **a** if **a** is large enough or to a newly created array and return the array; throw `ArrayStoreException` if the class type of any element in the hash set is not the same as or does not extend the class type of **a**; throw `NullPointerException` if **a** is null (inherited). |
| `int size()` | Return the number of objects in the hash set. |
| `String toString()` | Return a string representation of the hash set that contains the string representation of all the elements (inherited). |

**Figure 10-18 Methods in class `HashSet` including some inherited methods (continued)**



## HashSet (continued)

This example demonstates how to use the **HashSet** class to manage a list of **person** < name, age,hashCode> in which the **hasCode is the sum of charcter codes in the field name**.

Click to go to the HashTable class

```
import java.io.*;
import java.util.*;

class Person implements Comparable {
    private String name;
    public int age;
    private int hashcode = 0;
    public Person(String n, int a) {
        name = n; age = a;
        for (int i = 0; i < name.length(); i++)
            hashcode += name.charAt(i);
```

**Figure 10-19 Demonstrating the operation of the methods in class `HashSet`**



## HashSet (continued)

```
    }
    public Person() {
        this("",0);
    }
    public boolean equals(Object p) {
        return name.equals(((Person)p).name);
    }
    public int compareTo(Object p) {
        return name.compareTo(((Person)p).name);
    }
    public int hashCode() {
        return hashcode;
    }
    public String toString() {
        return "(" + name + "," + age + ")";
    }
}
```

**Figure 10-19 Demonstrating the operation of the methods in class `HashSet` (continued)**



## HashSet (continued)

```
class TestHashSet {
    public static void main(String[] ar) {
        HashSet hashset1 = new HashSet();
        hashset1.add(new Integer(40));
        hashset1.add(new Integer(60));
        System.out.println(hashset1);      // [40, 60]
        hashset1.add(new Integer(50));
        System.out.println(hashset1);      // [40, 50, 60]
        hashset1.add(new Integer(50));
        System.out.println(hashset1);      // [40, 50, 60]
        System.out.println(hashset1.contains(new Integer(50)));  // true
        System.out.println(hashset1.contains(new Integer(70)));  // false
        HashSet hashset2 = new HashSet();
        hashset2.add(new Integer(30));
        hashset2.add(new Integer(40));
        hashset2.add(new Integer(50));
        System.out.println(hashset2);      // [30, 40, 50]
        hashset1.addAll(hashset2);
        // union: [40, 50, 60] and [30, 40, 50] ==> [30, 40, 50, 60]
        System.out.println(hashset1);      // [30, 40, 50, 60]
        hashset1.remove(new Integer(30));
        System.out.println(hashset1);      // [40, 50, 60]
        hashset1.retainAll(hashset2);      // [40, 50]
        // intersection: [40, 50, 60] and [30, 40, 50] ==> [40, 50]
        System.out.println(hashset1);      // [40, 50]
```

**Figure 10-19 Demonstrating the operation of the methods in class `HashSet` (continued)**

## HashSet (continued)

```
hashset1.add(new Integer(60));     // [40, 50, 60]
hashset1.removeAll(hashset2);
// difference: [40, 50, 60] and [30, 40, 50] ==> [60]
System.out.println(hashset1);      // [60]
hashset1.add(null);
System.out.println(hashset1);      // [60, null]

HashSet pSet = new HashSet();
Person[] p = {new Person("Gregg",25), new Person("Ann",30),
              new Person("Bill",20), new Person("Gregg",35),
              new Person("Kay",30)};
for (int i = 0; i < p.length; i++)
     pSet.add(p[i]);
System.out.println(pSet);
// [(Ann,30), (Gregg,25), (Kay,30), (Bill,20)]
java.util.Iterator it = pSet.iterator();
((Person)it.next()).age = 50;
System.out.println(pSet);
// [(Ann,50), (Gregg,25), (Kay,30), (Bill,20)]
pSet.add(new Person("Craig",40));
```

**Figure 10-19 Demonstrating the operation of the methods in class `HashSet` (continued)**



## HashSet (continued)

```
        System.out.println(pSet);
        // [(Ann,50), (Gregg,25), (Kay,30), (Craig,40), (Bill,20)]
        for (int i = 0; i < p.length; i++)
            System.out.println(p[i] + " " + pSet.contains(p[i]));
        // (Gregg,25) true
        // (Ann,50) true
        // (Bill,20) true
        // (Gregg,35) true
        // (Kay,30) true
        // Using an array to sort elements in the hashset:
        Person[] pArray = (Person[]) pSet.toArray(new Person[0]);
        for (int i = 0; i < p.length; i++)
            System.out.print(pArray[i] + " ");
        System.out.println();
        // (Ann,50) (Gregg,25) (Kay,30) (Craig,40) (Bill,20)
        Arrays.sort(pArray);
        for (int i = 0; i < p.length; i++)
            System.out.print(pArray[i] + " ");
        System.out.println();
        // (Ann,50) (Bill,20) (Craig,40) (Gregg,25) (Kay,30)
        System.out.println(pSet);
    }
}
```

**Figure 10-19 Demonstrating the operation of the methods in class `HashSet` (continued)**



## The Hashtable class

- A `Hashtable` is roughly equivalent (gần tương đương) to a `HashMap` except that it is synchronized and does not permit null values with methods to operate on hash tables
- The class `Hashtable` is considered a legacy class, just like the class Vector
- Class hierarchy in `java.util` is:

  `Object → Dictionary → Hashtable`



## Hashtable (continued)

| Method | Operation |
|---|---|
| `void clear()` | Remove all the objects from the hashtable. |
| `Object clone()` | Return a copy of the hashtable without cloning its elements. |
| `boolean contains(Object val)` | Return **`true`** if the hashtable contains the object **`val`**; throw **`NullPointerException`** if **`val`** is null. |
| `boolean containsKey(Object key)` | Return **`true`** if the hashtable contains the object **`key`**; throw **`NullPointerException`** if **`key`** is null. |
| `boolean containsValue(Object val)` | Return **`true`** if the hashtable contains the object **`val`**; throw **`NullPointerException`** if **`val`** is null. |
| `Enumeration elements()` | Return an enumeration of the values in the hashtable. |
| `Set entrySet()` | Return a set containing all the pairs (key, value) in the hashtable. |

**Figure 10-20 Methods of the class `Hashtable` including three inherited methods**

# `Hashtable` (continued)

| | |
|---|---|
| `boolean equals(Object ob)` | Return `true` if the current hashtable and object `ob` are equal. |
| `Object get(Object key)` | Return the object associated with `key`; throw `NullPointerException` if `key` is null. |
| `int hashCode()` | Return the hash code for the hashtable. |
| `Hashtable()` | Create an empty hashtable with initial capacity equal to 11 and the load factor equal to .75. |
| `Hashtable(int ic)` | Create an empty hashtable with initial capacity `ic` and the load factor equal to .75; throw `IllegalArgumentException` if `ic < 0`. |

**Figure 10-20 Methods of the class `Hashtable` including three inherited methods (continued)**



# `Hashtable` (continued)

| | |
|---|---|
| `Hashtable(int ic, float lf)` | Create an empty hashtable with initial capacity `ic` and the load factor `lf`; throw `IllegalArgumentException` if `ic < 0` or `lf ≤ 0`. |
| `Hashtable(Map m)` | Create a hashtable with copies of elements from map `m`; throw `NullPointerException` if `m` is null. |
| `boolean isEmpty()` | Return `true` if the hashtable contains no elements, `false` otherwise. |
| `Enumeration keys()` | Return an enumeration containing all the keys of the hashtable. |
| `Set keySet()` | Return a set containing all the keys of the hashtable. |
| `Object put(Object key, Object val)` | Put the pair (`key`, `val`) in the hashtable; return a value associated with `key` if there is any in the hashtable, `null` otherwise; throw `NullPointerException` if `key` or `val` is null. |
| `void putAll(Map m)` | Add objects from map `m` to the current hashtable; throw `NullPointerException` if `m` is null. |

**Figure 10-20 Methods of the class `Hashtable` including three inherited methods (continued)**



# `Hashtable` (continued)

| | |
|---|---|
| `void rehash()` | A protected method to increase the capacity of the hashtable; the method is called automatically when the number of keys in the hashtable is greater than the product of the load factor and the current capacity. |
| `Object remove(Object key)` | Remove the pair (`key`, corresponding value) from the hashtable and return the value associated currently with `key` in the `hashtable`; throw `NullPointerException` if `key` is null. |
| `int size()` | Return the number of objects in the hashtable. |
| `String toString()` | Return a string representation of the hashtable that contains the string representation of all the objects. |
| `Collection values()` | Return a `Collection` object with all the values contained in the hashtable. |

**Figure 10-20 Methods of the class `Hashtable` including three inherited methods (continued)**



# `Hashtable` (continued)

```
import java.io.*;
import java.util.*;

class Person {
    private String name;
    public int age;
    public Person(String s, int i) {
        name = s; age = i;
    }
    Person() {
        this("",0);
    }
    public String toString() {
        return "(" + name + "," + age  + ")";
    }
    public boolean equals(Object p) {
        return name.equals(((Person)p).name);
    }
}
```

This example demonstates how to use the **HashTable** class to manage a list of **person** < name, age> with two cases:
**Case 1** : Key of a persion is an integer.
**Case 2** : Key of a persion is an object that belongs to the SSN class, a pre-defined class.

Click to go to the case study

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods**

# Hashtable (continued)

```
class SSN {
    private int value;
    private hashValue;
    public SSN(int i) {
        value = i;
        hashValue = (value & 0x0000ffff) + (value >>> 16)
    }
    public boolean equals(Object ob) {
        return value == ((SSN)ob).value;
    }
    public int hashCode() {
        return hashValue;
    }
    public String toString() {
        return "" + value;
    }
}

class testHashtable {
    static void print(Iterator it) {
        if (it.hasNext()) {
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**



# Hashtable (continued)

```
            System.out.print(it.next());
            while (it.hasNext())
                System.out.print(", " + it.next());
        }
        System.out.println();
    }
    static void print(Enumeration e) {
        if (e.hasMoreElements()) {
            System.out.print(e.nextElement());
            while (e.hasMoreElements())
                System.out.print(", " + e.nextElement());
        }
        System.out.println();
    }
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**



# Hashtable (continued)

```
public static void main(String[] ar) {
    Hashtable hashTable1 = new Hashtable(4);
    hashTable1.put(new Integer(123456789),new Person("Larry",25));
    hashTable1.put(new Integer(111111111),new Person("Kathy",30));
    System.out.println(hashTable1);
    // {111111111=(Kathy,30), 123456789=(Larry,25)}
    hashTable1.put(new Integer(222222222),new Person("Kathy",20));
    System.out.println(hashTable1);
    // {111111111=(Kathy,30), 222222222=(Kathy,20), 123456789=(Larry,25)}
    print(hashTable1.entrySet().iterator());
    // 111111111=(Kathy,30), 222222222=(Kathy,20), 123456789=(Larry,25)
    print(hashTable1.keySet().iterator());
    // 111111111, 222222222, 123456789
    print(hashTable1.elements());
    // (Kathy,30), (Kathy,20), (Larry,25)
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**



# Hashtable (continued)

```
print(hashTable1.keys());
// 111111111, 222222222, 123456789
print(hashTable1.values().iterator());
// (Kathy,30), (Kathy,20), (Larry,25)
Iterator it = hashTable1.values().iterator();
((Person)it.next()).age = 28;
System.out.println(hashTable1);
// {111111111=(Kathy,28), 222222222=(Kathy,20), 123456789=(Larry,25)}
hashTable1.put(new Integer(111111111),new Person("Jerry",20));
System.out.println(hashTable1);
// {111111111=(Jerry,20), 222222222=(Kathy,20), 123456789=(Larry,25)}
hashTable1.put(new Integer(111111113),new Person("Frank",30));
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**

## `Hashtable` (continued)

```
System.out.println(hashTable1);
// {111111113=(Frank,30), 123456789=(Larry,25), 222222222=(Kathy,20),
// 111111111=(Jerry,20)}
System.out.println(hashTable1.get(new Integer(111111111)));    // (Jerry,20)
System.out.println(hashTable1.contains(new Person("Jerry",20)));      // true
System.out.println(hashTable1.containsValue(new Person("Jerry",20))); // true
System.out.println(hashTable1.remove(new Integer(111111111))); // (Jerry,20)
System.out.println(hashTable1);
// {222222222=(Kathy,20), 123456789=(Larry,25)}
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**



## `Hashtable` (continued)

```
        Hashtable hashTable2 = new Hashtable();
        hashTable2.put(new SSN(123456789),new Person("Larry",28));
        hashTable2.put(new SSN(111111111),new Person("Kathy",30));
        System.out.println(hashTable2);
        // {111111111=(Kathy,30), 123456789=(Larry,28)}
        hashTable2.put(new SSN(222222222),new Person("Kathy",20));
        System.out.println(hashTable2);
        // {222222222=(Kathy,20), 111111111=(Kathy,30), 123456789=(Larry,28)}
        hashTable2.put(new SSN(111111111),new Person("Jerry",25));
        System.out.println(hashTable2);
        // {222222222=(Kathy,20), 111111111=(Jerry,25), 123456789=(Larry,28)}
        hashTable2.put(new SSN(111111113),new Person("Frank",30));
        System.out.println(hashTable2);
        // {222222222=(Kathy,20), 111111113=(Frank,30), 111111111=(Jerry,25),
        // 123456789=(Larry,28)}
    }
}
```

**Figure 10-21 A program demonstrating operations of the `Hashtable` methods (continued)**



## Bài tập

- Xây dựng chương trình tự điển, dữ liệu được tải lên từ file như sau
  – hello,xin chao
  – study,hoc



## Case Study: Hashing with Buckets

This example(in the textbook) depicts how to implement hashing data in a file.
Do yourself

Click to go to the Summary

```
import java.io.*;
import java.io.File;

public class FileHashing {
    private final int bucketSize = 2, tableSize = 3, strLen = 20;
    private final int recordLen = strLen;
    private final byte empty = '*', delMarker = '#';
    private long[] positions;
    private InputStreamReader isr = new InputStreamReader(System.in);
    private BufferedReader buffer = new BufferedReader(isr);
    private RandomAccessFile outfile;
    private RandomAccessFile sorted;
    private RandomAccessFile overflow;
    public FileHashing() {
    }

    private void print(byte[] s) { // print a byte array;
        for(int k = 0; k < s.length; k++)
            System.out.print((char)s[k]);
    }

    private long hash(byte[] s) {
        long xor = 0, pack;
        int i, j, slength; // exclude trailing blanks:
        for (slength = s.length; s[slength-1] == ' '; slength--);
```

**Figure 10-22 Implementation of hashing using buckets**

## Case Study: Hashing with Buckets (continued)

```
    for (i = 0; i < slength; ) {
        for (pack = j = 0; ; j++, i++) {
            pack |= (long) s[i];  // include s[i] in the rightmost
            if (j == 3 || i == slength - 1) { // byte of pack;
                i++;
                break;
            }
            pack <<= 8;
        }             // xor at one time 8 bytes from s;
        xor ^= pack;  // last iteration may put less
    }                 // than 8 bytes in pack;
    return (xor % tableSize) * bucketSize * recordLen;
}// return byte position of home bucket for s;

private byte[] getName() throws IOException {
    System.out.print("Enter a name & phone#: ");
    String s = buffer.readLine();
    for (int i = s.length(); i < recordLen; i++)
        s += ' ';
    return s.getBytes(); // s => line
}
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



## Case Study: Hashing with Buckets (continued)

```
private int comparesTo(byte[] s1, byte[] s2) {  // same length
    for (int i = 0; i < s1.length; i++) // of s1 and s2
        if (s1[i] != s2[i])           ]  // is assumed;
            return s1[i] - s2[i];
    return 0;
}

private void insert() throws IOException {
    insertion(getName());
}

private void insertion(byte[] line) throws IOException {
    byte[] name = new byte[recordLen];
    boolean done = false, inserted = false;
    int counter = 0;
    long address = hash(line);
    outfile.seek(address);
    while (!done && outfile.read(name) != -1) {
        if (name[0] == empty || name[0] == delMarker) {
             outfile.seek(address+counter*recordLen);
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



## Case Study: Hashing with Buckets (continued)

```
             outfile.write(line);
             done = inserted = true;
        }
        else if (comparesTo(name,line) == 0) {
             print(line);
             System.out.println(" is already in the file");
             return;
        }
        else counter++;
        if (counter == bucketSize)
             done = true;
        else outfile.seek(address+counter*recordLen);
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



## Case Study: Hashing with Buckets (continued)

```
    }
    if (!inserted) {
        done = false;
        counter = 0;
        overflow.seek(0);
        while (!done && overflow.read(name) != -1) {
            if (name[0] == delMarker)
                 done = true;
            else if (comparesTo(name,line) == 0) {
                 print(line);
                 System.out.println(" is already in the file");
                 return;
            }
            else counter++;
        }
        if (done)
             overflow.seek(counter*recordLen);
        else overflow.seek(overflow.length());
        overflow.write(line);
    }
}
```

**Figure 10-22 Implementation of hashing using buckets (continued)**

# Case Study: Hashing with Buckets (continued)

```
private void delete() throws IOException {
    byte[] line = getName();
    long address = hash(line);
    outfile.seek(address);
    int counter = 0;
    boolean done = false, deleted = false;
    byte[] name = new byte[recordLen];
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
    while (!done && outfile.read(name) != -1) {
        if (comparesTo(line,name) == 0) {
            outfile.seek(address+counter*recordLen);
            outfile.write(delMarker);
            done = deleted = true;
        }
        else counter++;
        if (counter == bucketSize)
            done = true;
        else outfile.seek(address+counter*recordLen);
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
    }
    if (!deleted) {
        done = false;
        counter = 0;
        overflow.seek(0);
        while (!done && overflow.read(name) != -1) {
            if (comparesTo(line,name) == 0) {
                overflow.seek(counter*recordLen);
                overflow.write(delMarker);
                done = deleted = true;
            }
            else counter++;
            overflow.seek(counter*recordLen);
        }
    }
    if (!deleted) {
        print(line);
        System.out.println(" is not in database");
    }
}
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
private void swap(long[] arr, int i, int j) {
    long tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;
}

private int partition(int low, int high) throws IOException {
    byte[] rec = new byte[recordLen];
    byte[] pivot = new byte[recordLen];
    int i, lastSmall;
    swap(positions,low,(low+high)/2);
    outfile.seek(positions[low]*recordLen);
    outfile.read(pivot);
```

**Figure 10-22 Implementation of hashing using buckets (continued)**

# Case Study: Hashing with Buckets (continued)

```
        for (lastSmall = low, i = low+1; i <= high; i++) {
            outfile.seek(positions[i]*recordLen);
            outfile.read(rec);
            if (comparesTo(rec,pivot) < 0) {
                lastSmall++;
                swap(positions,lastSmall,i);
            }
        }
        swap(positions,low,lastSmall);
        return lastSmall;
    }
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
private void sort(int low, int high) throws IOException {
    if (low < high) {
        int pivotLoc = partition(low, high);
        sort(low, pivotLoc-1);
        sort(pivotLoc+1, high);
    }
}

private void sortFile() throws IOException {
    byte[] rec = new byte[recordLen];
    sort(1,(int)positions[0]); // positions[0] contains the # of elements;
    for (int i = 1; i <= positions[0]; i++) { // put data from
        outfile.seek(positions[i]*recordLen); // outfile in sorted order
        outfile.read(rec);
        sorted.write(rec);                    // in file sorted;
    }
}
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
// data from overflow file and outfile are all stored in outfile and
// prepared for external sort by loading positions of the data to an array;

private void combineFiles() throws IOException {
    byte[] rec = new byte[recordLen];
    int counter = bucketSize*tableSize;
    outfile.seek(outfile.length());
    overflow.seek(0);
    while (overflow.read(rec) != -1) { // transfer from
        if (rec[0] != delMarker) {     // overflow to outfile only
            counter++;                 // valid (undeleted) items;
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
                outfile.write(rec);
            }
        }
        positions = new long[counter+1];
        outfile.seek(0);          // load to the array positions
        int arrCnt = 1;           // of valid data stored in output file;
        for (int i = 0; i < counter; i++) {
            outfile.seek(i*recordLen);
            outfile.read(rec);
            if (rec[0] != empty && rec[0] != delMarker)
                positions[arrCnt++] = i;
        }
        positions[0] = --arrCnt; // store the number of data in position 0;
    }
```

**Figure 10-22 Implementation of hashing using buckets (continued)**

# Case Study: Hashing with Buckets (continued)

```
public void processFile(String fileName) {
    char command = '1';
    byte[] line = new byte[recordLen];
    String commandLine;
    try {
        (new File(".\\","outfile")).delete();
        (new File(".\\","overflow")).delete();
        (new File(".\\","sorted")).delete();
        RandomAccessFile fIn = new RandomAccessFile(fileName,"rw");
        outfile = new RandomAccessFile("outfile","rw");
        sorted = new RandomAccessFile("sorted","rw");
        overflow = new RandomAccessFile("overflow","rw");
        for (int i = 1; i <= tableSize*bucketSize*recordLen; i++)
            outfile.write(empty);      // initialize outfile;
        while (fIn.read(line) != -1)    // load fIn to outfile;
            insertion(line);
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
while (command != '3') {
    System.out.print("Enter your choice "
                   + "(1. insert, 2. delete, 3. exit): ");
    commandLine = buffer.readLine();
    command = commandLine.charAt(0);
    if (command == '1')
         insert();
    else if (command == '2')
         delete();
    else if (command != '3')
         System.out.println("Wrong command entered, please retry.");
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
            }
            combineFiles();
            sortFile();
            outfile.close();
            sorted.close();
            overflow.close();
            fIn.close();
            (new File(".\\","names")).delete();
            (new File(".\\","sorted")).renameTo(new File(".\\","names"));
        } catch (IOException ioe) {
        }
    }
```

**Figure 10-22 Implementation of hashing using buckets (continued)**



# Case Study: Hashing with Buckets (continued)

```
    static public void main(String args[]) {
        String fileName = "";
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader buffer = new BufferedReader(isr);
        FileHashing fClass = new FileHashing();
        try {
            if (args.length == 0) {
                 System.out.print("Enter a file name: ");
                 fileName = buffer.readLine();
            }
            else fileName = args[0];
        } catch(IOException io) {
            System.err.println("Cannot open " + fileName);
        }
        fClass.processFile(fileName);
    }
}
```

**Figure 10-22 Implementation of hashing using buckets (continued)**

## Summary

- Common hash functions include the division, folding, mid-square, extraction and radix transformation methods.
- Collision resolution includes the open addressing, chaining, and bucket addressing methods.
- Cichelli's method is an algorithm to construct a minimal perfect hash function



## Summary (continued)

- The FHCD algorithm searches for a minimal perfect hash function of the form (modulo *TSize*), where *g* is the function to be determined by the algorithm
- In expandable hashing and dynamic hashing, a binary tree is used as an index of buckets
- In extendible hashing, a directory of records is kept in a table



## Summary (continued)

- A hash map is a collection of singly linked lists (buckets); that is, chaining is used as a collision resolution technique
- `HashSet` is another implementation of a set (an object that stores unique elements)
- A `Hashtable` is roughly equivalent to a `HashMap` except that it is synchronized and does not permit null values with methods to operate on hash tables



## Notices about using hash tables

- When should hashtables be used:
  - Elements in a group are different and insertion and search are main operations.
- What are things to be concerned before a hashtable is implemented?
  - Choose a key for each element: number/string?
  - Choose a hash function
  - Choose a collision resolution

  because these things will affect on algorithms that will be selected in our hashtable.

This lecture has no tutorial. Re-code samples in the textbook.

**Thank you.**