FPT UNIVERSITY
DREAM OF INNOVATION

7. Hashing

Objectives

- Why Hashing?
- Hash Table
- Hash Functions
- Collision Resolution
- Deletion
- Perfect Hash Functions
- Hash Functions for Extendable files
- Hash code
- Maps
- Hashing in java.util

Why hashing?

- If data collection is sorted array, we can search for an item in O(log n) time using the binary search algorithm.
- However with a sorted array, inserting and deleting items are done in O(n) time.
- If data collection is balanced binary search tree, then inserting, searching and deleting are done in O(log n) time.
- Is there a data structure where inserting, deleting and searching for items are more efficient?
- The answer is “Yes”, that is a Hash Table.

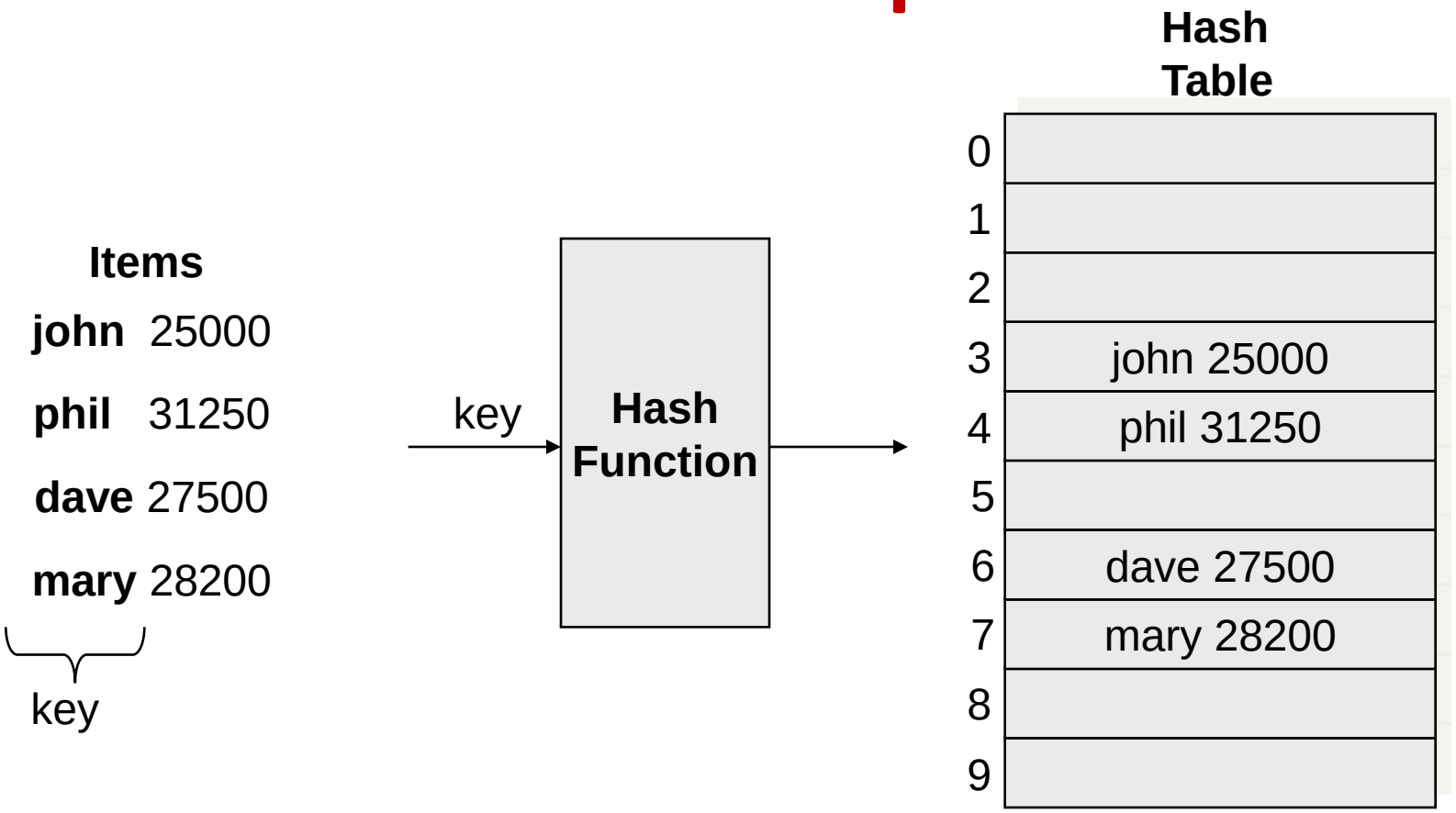
Hash Tables

- We'll discuss the *hash table* ADT which supports only a subset of the operations allowed by binary search trees.
- The implementation of hash tables is called **hashing**.
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function*.

Hash Table Example - 1



Hash Function - 1

- The hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but in many cases we don't.

Hash Function - 2

Problems:

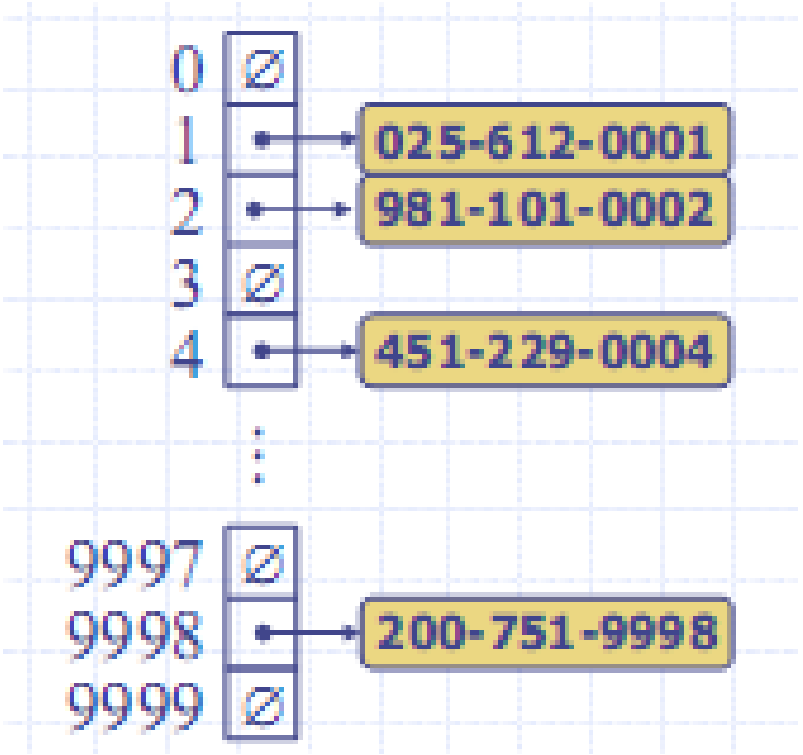
- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

Hash Function - 3

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Hash table example - 2

- We design a hash table for a map storing entries as (ID, Name), where ID is a nine-digit positive integer
- Our ***hash table*** uses an array of size N = 10,000 and the hash function h(x) = last four digits of x



How to select Hash Functions?

- We want a hash function that is easy to compute and that minimizes the number of collisions.
- Hashing functions should be unbiased.
  - That is, if we randomly choose a key, *x*, from the key space, the probability that *f*(*x*) = i is 1/M, where M is the size of the hash table.
  - We call a hash function that satisfies unbiased property a *uniform hash function*.

FPT UNIVERSITY
DREAM OF INNOVATION

Division Hash Functions

- **Division** $h_D(x) = x \% M$:
  - Using the modulus (%) operator.
  - We divide the key *x* by some number *M* and use the remainder as the hash index for *x*.
    - This gives indexes that range from 0 to *M* - 1,

      where *M* = that table size (hash table).
- The choice of *M* is critical.
  - If *M* is divisible by 2, then odd keys to odd indexes and even keys to even ones. (biased!!)
  - If M is a power of 2, i.e. m = 2^p , then h(k) is just the p lowest-order bits of k. (biased!!)
  - If M = pH, then keys in the set {H, 2H, 3H, …, (p-1)H, pH, (p+1)H,…, kH,…} map to p positions {H, 2H, 3H, …, (p-1)H, 0} only (biased!!)
  - A good choice for *M* would be : *M* a prime number such that *M* does not divide $r^k \pm a$ for small *k* and *a*.

Folding Hash Functions

- **Folding**
  - Partition key *x* into several parts.
  - All parts except for the last one have the same length.
  - Add the parts together to obtain the value y, the hash index then is $h(x)$ = y % *M.*
- Two possibilities (divide *x* into several parts)
  - ***Shift folding***:
    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. Suppose x = 72320354121324
    - x_1=723, x_2=203, x_3=541, x_4=213, x_5=24,

    index= $(x_1 + x_2 + x_3 + x_4 + x_5)$ % 1000 = 1704%1000 = 704
  - ***Boundary folding*** (folding at the boundaries):
    reverses every other partition before adding
    - x_1=723, x_2=302, x_3=541, x_4=312, x_5=24, index=1902%1000 = 902

Other Hash Functions

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

  Exp: key = 3121^2 = 9 740 641 ->index = 406 mod Tsize

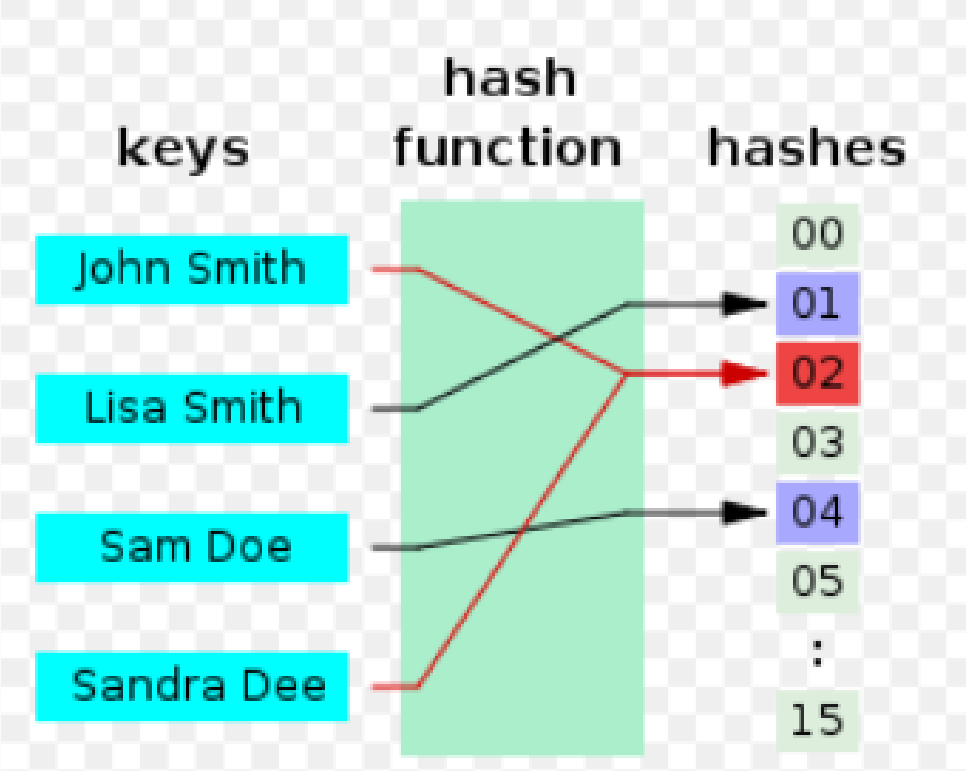
  Since the middle bits of the square usually depend upon all the characters in a key, there is high probability that different keys will produce different hash indexes.
- In the **extraction** method, only a part of the key is used to compute the address

  Exp: 123-45-6789->1234-5-6789 -> index = 1289 mod TSize
- Using the **radix transformation**, the key *K* is transformed into another number base; *K* is expressed in a numerical system using a different radix.

  Exp: $345_{10} = 423_9$ -> index = 423 mod TSize

Collision

- Collisions occur when different elements are mapped to the same cell



Collision Resolution

Open addressing method

In the **open addressing** method, when a key x 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. Thus, if the position k = h(x) is used, then the following postions are tried:

$$k = h_i(x) = h(x) + p(i) \bmod M, \quad i = 1, 2, \ldots \ (M = Tsize)$$

- The simplest method is **linear probing**, for which $p(i) = i$, and for the *ith* probe, the position to be tried is $(h(x) + i) \bmod M$, $i = 1,2,\ldots$

- **Quadratic**: $p(i) = i^2$ thus the position to be tried is $(h(x) + i^2) \bmod M$, $i = 1,2,\ldots$

Search an item in hash tables using linear Probing

- Consider a hash table A that uses linear probing
- get(k)
  - We start at cell h(k)
  - We probe consecutive locations until one of the following occurs
  - An item with key k is found, or
  - An empty cell is found, or
  - N cells have been unsuccessfully probed

```
Algorithm get(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

Factors affecting Search perfomance

- Quality of hash function
  - how uniform?
  - depends on actual data
- Collision resolution strategy used
- Load factor of the HashTable
  - N/Tsize
  - The lower the load factor the better the search performance

Quadratic Probing example

if the position k = h(x) is used, then the following postions are tried:

$$k = h_i(x) = h(x) + i^2 \bmod M, \quad i = 1, 2, \ldots \ (M = Tsize)$$

h(x) = x mod 10

Insert keys 89, 18, 49, 58, 69 in this order

Content	49		58	69					18	89
Index	0	1	2	3	4	5	6	7	8	9

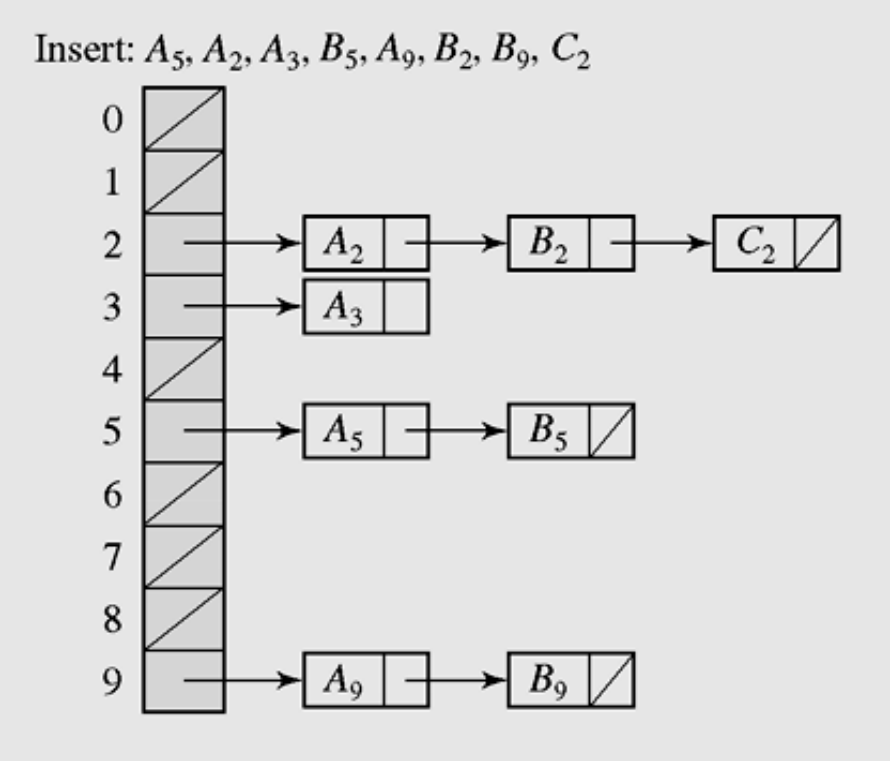
Advantages and disadvantages of quadratic probing

One problem with quadratic probing is that probe sequences do not probe all locations in the table. For example, if M=11, k = h(x) = x%11. Then for those key x-s, where h(x) = 3 and collision occurs, only positons 3, 4, 7, 1, 8, 6 are probed.
when M is prime, we can make the following guarantee.
Theorem. If M is prime and the table is at least half empty, then quadratic probing will always find an empty location. Furthermore, no locations are checked twice.

Collision Resolution

Chaining method

- Keys do not have to be stored in the table itself. In **chaining**, 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**. In this method, the table can never overflow, because the linked list is extendible.

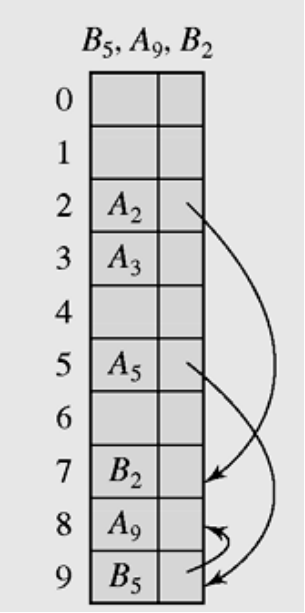


In chaining, colliding keys are put on the same linked list

Collision Resolution

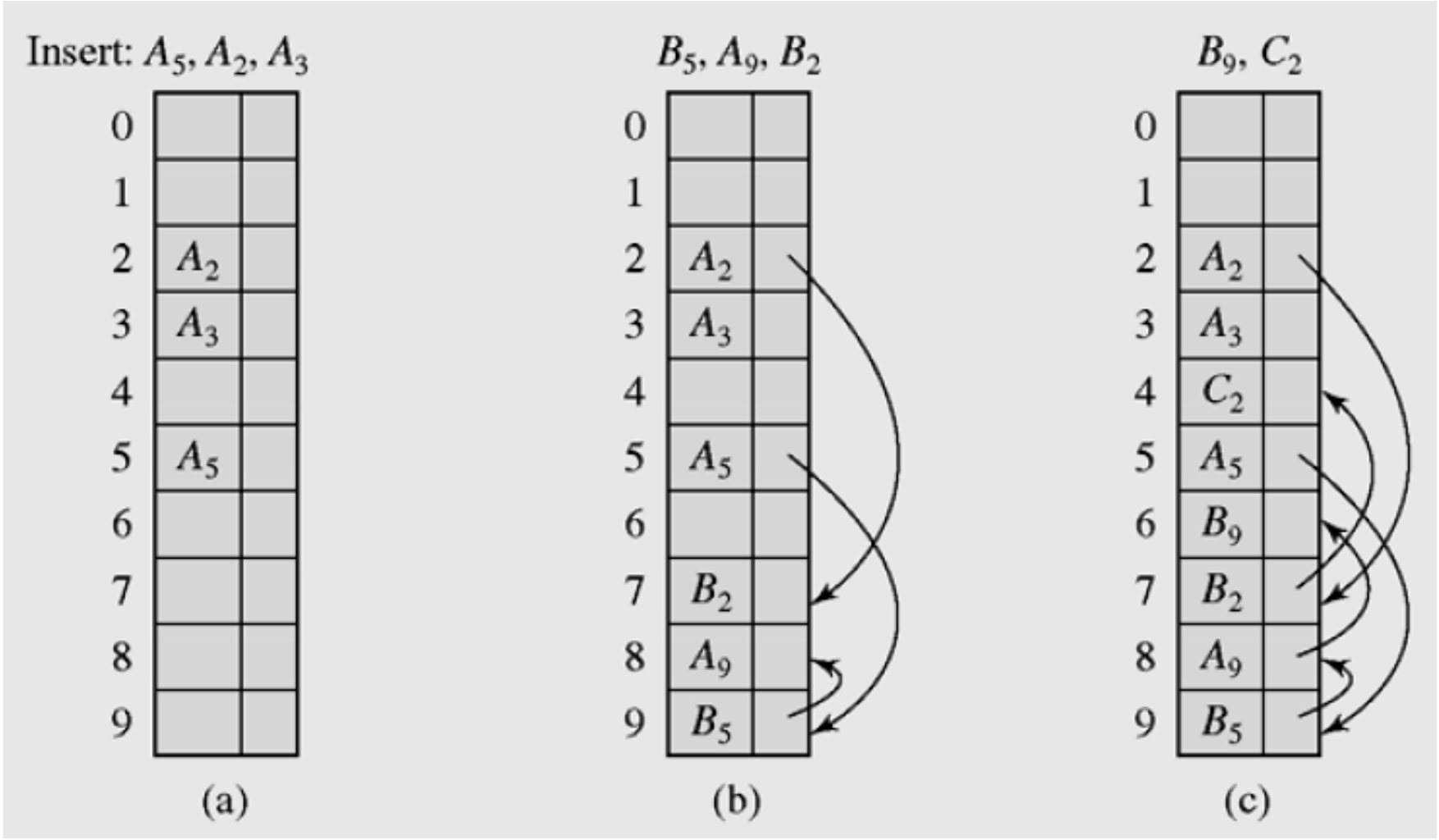
Coalesced hashing or coalesced chaining method

- A version of chaining called **coalesced hashing** (or **coalesced chaining**) combines linear probing with chaining. Each position pos in the table contains 2 fields: info and next. The next field contains the index of the next key that is hashed to pos. By this way, a sequential search down the table can be avoided by directly accessing the next element on the linked list.
- An overflow area known as a **cellar** can be allocated to store keys for which there is no room in the table



Coalesced hashing puts a colliding key in the last available position of the table

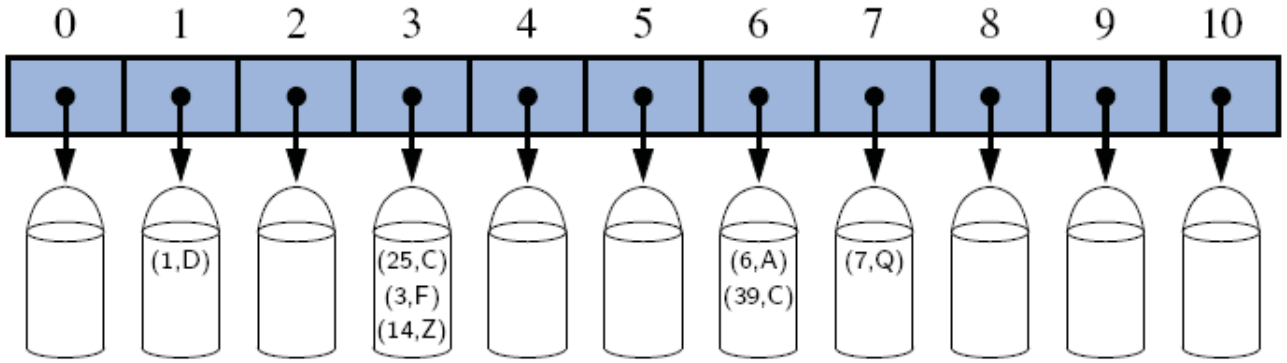
Coalesced hashing example



Coalesced hashing puts a colliding key in the last available position of the table

Bucket Addressing

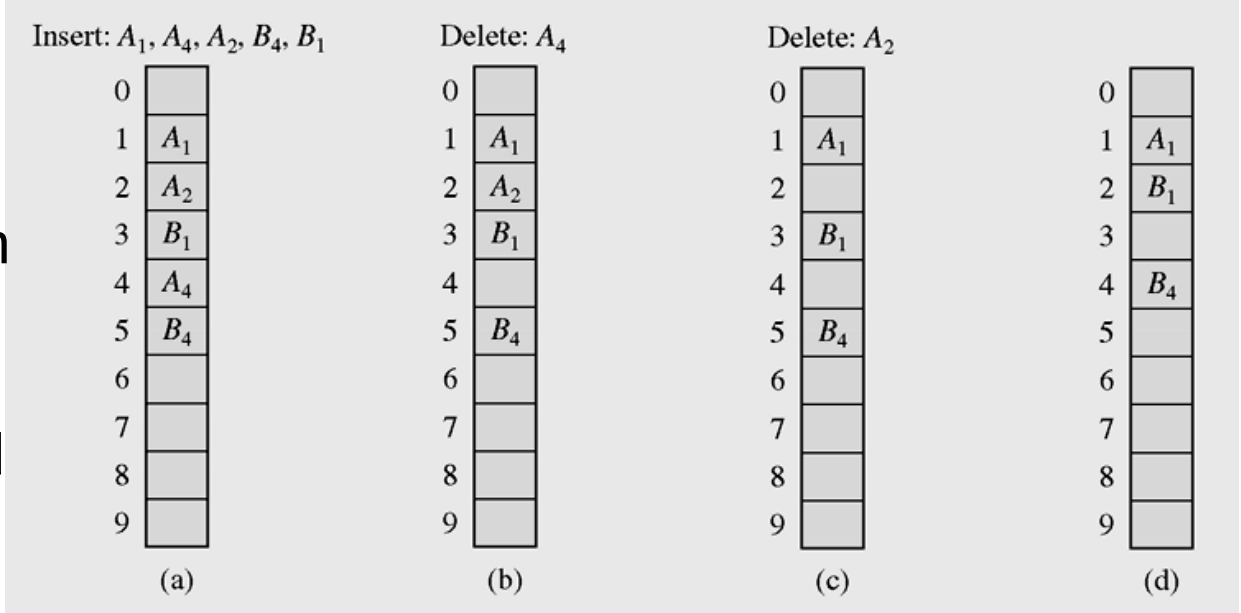
- To store colliding elements in the same position in the table can be achieved by associating a bucket with each address.
- A **bucket** is a block of space large enough to store multiple items (a block consists of slots, each slot contains one item).
- By using buckets, *the problem of collisons is not totally avoided*. By incorporating the open addressing approach, the colliding item can be stored in the next bucket is it has an available slot when using linear probing, or it can be stored in some other bucket when, say, quadratic probing is used. The colliding items can also be stored in an overflow area. *In this case, each bucket includes a field that indicates whether the search should be continued in this area or not*.



Buckets of a hash table with size 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a modulo-division hash function.

Deletion

Consider the table in which the keys are stored using linear probing. Suppose we delete A_4 and then then try to find B_4. Because when searching B we hash it to position 4 and see that this position is empty and conclude that B4 is not found (which is not true).



Linear search in the situation where both insertion and deletion of keys are permitted

To avoid this situation, we mark the deleted positions only. When inserting new element to this position, we update information for new element. When there too many marked deleted elements in the table, the table is refresh (d).

Perfect Hash Functions

- If hash function h transforms different keys into different numbers, it is called a ***perfect hash function***.
- 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**
- **Cichelli's method** is an algorithm to construct a minimal perfect hash function

Hash Functions for Extendible Files

- There are two categories of hashing: Static hashing (the hash table is fixed-sized), and Dynamic/Extendible hashing.
- Dynamic/Extendible hashing: splits and coalesces buckets appropriately with the database size.
  - i.e. buckets are added and deleted on demand.
  - The hash function typically produces a large number of hash values, uniformly and randomly.
  - Only part of the value k = h(x) is used depending on the size of the database.
  - Hash indices are typically a prefix of the entire hash value.
  - More than one consecutive index can point to the same bucket.

Cryptographic Hash Functions

A **cryptographic hash function** is a hash function which takes an input (or message) and returns a fixed-size alphanumeric string, which is called the **hash value** (sometimes called a **message digest**, a **digital fingerprint**, a **digest** or a **checksum**).

The ideal hash function has three main properties:

1. It is extremely easy to calculate a hash for any given data.
2. It is extremely computationally difficult to calculate an alphanumeric text that has a given hash.
3. It is extremely unlikely that two slightly different messages will have the same hash.
4. Functions with these properties are used as hash functions for a variety of purposes, not only in cryptography. Practical applications include message integrity checks, digital signatures, authentication, and various information security applications.

Common hash functions: MD5, SHA-1, SHA-2 , SHA-3

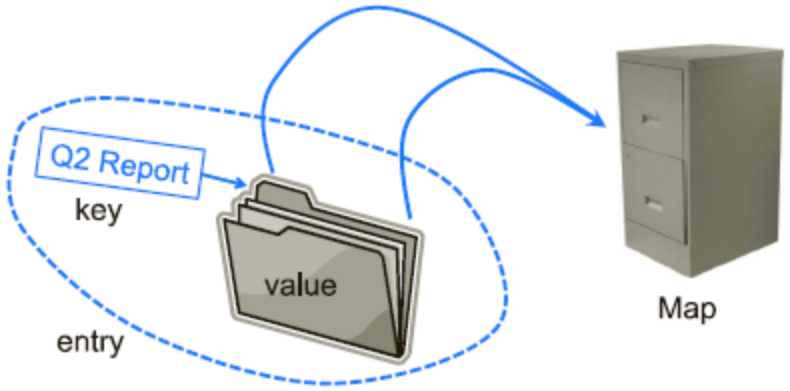
Hash Code

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are not integers, hash function needs more care.
  - The first action that a hash function performs is to convert an arbitrary key *k* to an integer that is called the ***hash code*** for *k*; this integer need not be in the range [0,M −1], and may even be negative. For example, If the keys are real numbers between 0 and 1, we might just multiply by M and round off to the nearest integer to get an index between 0 and M-1

Maps - 1

A ***map*** is an abstract data type designed to efficiently store and retrieve values based upon a uniquely identifying ***search key*** for each. Specifically, a map stores keyvalue pairs (*k*,*v*), which we call ***entries***, where *k* is the key and *v* is its corresponding value. Keys are required to be unique, so that the association of keys to values defines a mapping.

The figure below provides a conceptual illustration of a map using the file-cabinet metaphor. For a more modern metaphor, think about the web as being a map whose entries are the web pages. The key of a page is its URL (e.g., http://datastructures.net/) and its value is the page content.



Maps - 2

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are **not** allowed
- Applications:
  - address book
  - student-record database

The methods of Map ADT

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

Map application example

Counting Word Frequencies

As a case study for using a map, consider the problem of counting the number of occurrences of words in a document. This is a standard task when performing a statistical analysis of a document, for example, when categorizing an email or news article. A map is an ideal data structure to use here, for we can use words as keys and word counts as values.

Hashing in java.util - HashSet and HashMap classes

- HashSets store objects
  - supports adding and removing in constant time
- HashMaps store a pair (key,object)
  - this is an implementation of a **Map**
  - **HashMap is more useful and standard**
  - **HashMap's main methods are:**
    - **put(Object key, Object value)**
    - **get(Object key)**
    - **remove(Object key)**
  - **All done in expected O(1) time.**
  - **The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls**.
  - An instance of HashMap has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

A **hash map** is a collection of singly linked lists (buckets); that is, chaining is used as a collision resolution technique

HashSet example

```
import java.util.*;
public class Main
  { public static void main(String args[])
      { Set s = new HashSet();
        String [] a = {"i", "came", "i", "came", "i", "conquered"};
        for(int i=0; i<a.length;i++)
          { if(!s.add(a[i]))
            System.out.println("Duplicate detected : " + a[i]);
          }
        System.out.println(s.size() + " distinct words detected : " + s );
       }
  }
```

```
Duplicate detected : i
Duplicate detected : came
Duplicate detected : i
3 distinct words detected : [i, conquered, came]
```

HashMap class

Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (**The HashMap class is roughly equivalent to HashTable, except that it is unsynchronized and permits nulls.**)

Some main methods	
void	**clear**() Removes all mappings from this map.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Object	**get**(Object key) Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
Object	**remove**(Object key) Removes the mapping for this key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.

HashMap class example

```
import java.util.*;
public class Main
  { public static void main(String args[])
      { HashMap hMap = new HashMap();
        hMap.put("One", new Integer(1));
        hMap.put("Two", new Integer(2));
        Object obj = hMap.get("One");
        System.out.println(obj);
      }
  }
```

Output is: 1

Applications of Hashing

There are many areas where hashing is applicable.Here are common ones:

1. **Databases**: Efficient retrieval of records.
2. **Compilers**: Symbol tables.
3. **Games**: Lookup board configuration to find the move that goes with it.
4. **UNIX shell**: Quick command lookup.
5. **IP Routing**: Fast IP address lookup.

Summary

- Hash functions include the division, folding, mid-square, extraction and radix transformation methods
- Collision resolution includes the open addressing, chaining, and bucket addressing methods
- A HashMap is a is an implementation of a **Map** ADT using hashing 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

FPT UNIVERSITY
DREAM OF INNOVATION

Reading at home

Text book: Data Structures and Algorithms in Java

10 Maps, Hash Tables, and Skip Lists - 401
10.2 Hash Tables - 410
10.2.1 Hash Functions - 411
10.2.2 Collision-Handling Schemes - 417
10.2.3 Load Factors, Rehashing, and Efficiency - 420
10.2.4 Java Hash Table Implementation - 422