Introduction to Hashing

- References: Text book : Chapter 21 and Chapter 22

1. Hashing

- Support ADT Dictionary operations without searching (or with minimum searching)

- Not good for sorted order traversal

- Given a Table T with m slots : 0, 1, …, m-1 and n records (or objects or data) each with a unique search key k

- Direct Address

  If we have a small set of records, i.e. n <= m with each search key < m, then store record with key k into slot k of Dictionary, i.e. T[k]

  Example : Table T

Index		Record
0		
1		
…		
i	→	Record with key = i
…		
m-1		

  All operations : O(1)

  Insertion() : Insert record with key k into Table[k]
  Deletion() : Delete record R of key k from Table[k]
  Retrieval() : Return record R of key k from Table[k]

- Hash Table

  If the range of keys is big (can be bigger than m-1). It is not practical to build a table with m = maximum key value.

  Solution : Use hash function h()

  - Given a search key, it returns the index of an element in an array called the hash table, i.e. given any key k, h(k) ∈ {0,1,2,…, m-1}

  - The index is known as the hash index

  - A perfect hash function maps each search key into a different integer suitable as an index to the hash table

  - Typically, hash function performs two steps
    Convert the search key into an integer called the hash code
    Compress the hash code into the range of indices for the hash table

  - Typical hash functions are not perfect
    They can allow more than one search key to map into a single index
    This is known as a collision

  - Example1 : h(k) = k mod 5

    h(7) = 7 mod 5 = 2 /* put this record into slot 2 */
    h(14) = 14 mod 5 = 4 /* put this record into slot 4 */

Index		Record
0		
1		
2	→	record with key = 7
3		
4	→	record with key = 14

    Note : two keys may map into same Dictionary index → collision.
    e.g. key 7 and 12

- We would like to study two major issues in hashing

Collision resolution

Designing a good hash function

2. Collision resolution

(i) Open addressing

Each slot in the Table can only hold 1 record only, i.e. all records are stored in the Table. To resolve collision, need to locate alternate slot.

In this scheme, n <= m, but the key numbers may be (>= m)

Idea :

- Each Table entry (or record) has additional field to indicate "empty", "occupied" and "deleted"

- To insert – if the slot is occupied, try another slot until an empty or deleted slot is found (need to define the probing strategy), set indicator to "occupied"

- To delete – follow the defined probe sequence.
  If the Table entry is "deleted", continue to search.
  If the Table entry is "empty", return "unsuccessful".
  If the record is found, delete the record from the Table, set indicator to "deleted".

- To retrieve – follow the defined probe sequence.
  If the Table entry is "deleted", continue to search.
  If the Table entry is "empty", return "unsuccessful".
  If the record is found, return the record information.

Example: h(k) = k mod 8 and use linear probing for the following operations (a) insert 2, (b) insert 10, (c) insert 3, (d) insert 13, (e) search 10, (f) search 18, (f) insert 18, (g) delete 3, (h) search 18, (i) insert 26
// To avoid error (duplicate keys) in insert, need to probe until "empty

Probing strategies

- Linear probing : search sequentially until the desired entry is found i.e. T(k) = u, probe sequence is u, (u+1) mod m, (u+2) mod m. …. etc

  Problem : primary clustering – Table contains groups of consecutively occupied entries

- Quadratic probing : use probe sequence u, $(u+1^2)$, $(u+2^2)$, $(u+3^2)$, … etc

  Avoid primary clustering. May not probe all entries in array
  Problem : secondary clustering – same probe sequence for T(k') = T(k'') = u

- Double hashing : use two different hash functions
  h1(k) = a, h2(k) = b.

  The probe sequence is a, a+b, a+2b, a+3b, … etc
  This method avoids both primary and secondary clustering. May not probe all entries in array

(ii) Separate chaining

Each slot (bucket) in the Table holds a linked list, i.e. Elements in the same slot are linked together (actually, a bucket can be any container such as array, vector, linked list, sorted list etc)

In this scheme, n can be bigger than m and the key numbers may be >= m

Idea : assume h(k) = u, get the bucket of T[u]

- To insert – insert record into the front of the list
- To delete – search and delete the record (if it is found)
- To retrieve – search and retrieve the record (if it is found)

(iii) Analysis :

Assume each key is equally likely to be hashed into any slot

Let n = total number of keys and m = number of slots in the Table

load factor $\alpha = n/m$
i.e. how full the Table is.

Note : it may be more than 1 (using separate chaining)

From the book "The Art of Computer Programming, Vol. 3" by D. Knuth, the following analysis is provided :

- Linear probing : average number of comparisons

$0.5 * [1 + 1/(1-\alpha)]$ for a successful search
$0.5 * [1 + 1/(1-\alpha)^2]$ for an unsuccessful search

For example :	$\alpha = 2/3$,	$\alpha = 0.9$ (90%)
successful search	2	5.5
unsuccessful search	5	50.5

- Quadratic probing and double hashing :

$[-\log_e(1-\alpha)]/\alpha$ for a successful search
$1/(1-\alpha)$ for an unsuccessful search

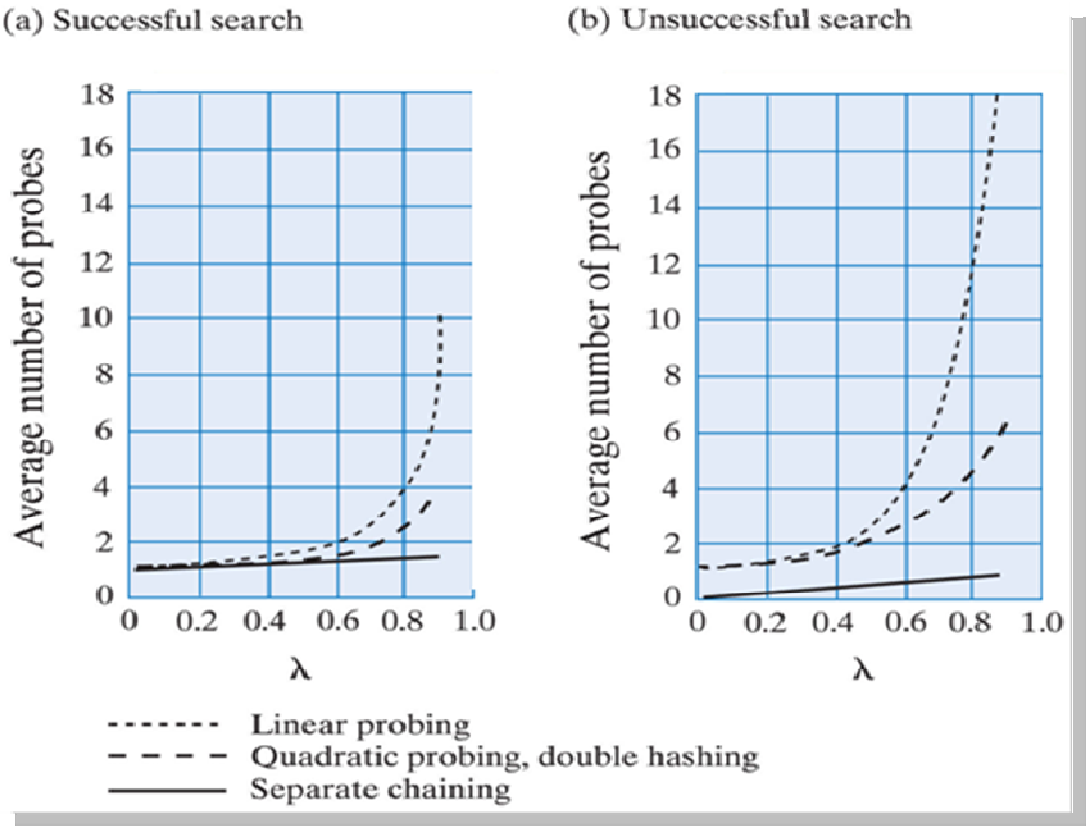
For example :	$\alpha = 2/3$,	$\alpha = 0.9$ (90%)
successful search	2	2.6
unsuccessful search	3	10

- Separate chaining :

  $1 + \alpha/2$ for a successful search
  α for an unsuccessful search

  note : 1 for checking if a bucket is not empty)

For example :	$\alpha = 2$	$\alpha = 0.5$
successful search	2	1.3
unsuccessful search	2	0.5



(iv) Rehashing: To improve performance

- When load factor becomes too large
  Expand the hash table
- Double present size, increase result to next prime number
- Use method **add** to place current entries into new hash table

3. Designing a good hash function

- What is a good hash function?

  - Should be easy and fast to compute
  - Should distribute keys uniformly into slots
  - Regularity in key distribution should not affect uniformity
    i.e. map number 1-100 into slot 1, 101-200 into slot 2 →bad functions

  Consider integer search keys only. Note : if a key is a string, it can be converted to integer by using the ASCII numbers, then apply the hash function

- String hash functions. Let string be $s_0 s_1 s_2 s_3 \ldots s_{n-1}$

  Note: Compute hash code c only…. User needs to c % m (table size)

  - Example 1: Let string hash function be :

    $$S_{n-1} + 32^1 s_{n-2} + \ldots + 32^{n-2} s_1 + 32^{n-1} s_0$$

    ```
    h = 0
    for (int i = 0; i < n; i--) {
       h = 32*h + s_i   // same as shift 5 bits to left
    }
    ```

    we end up keeping only the bits from last 7 bytes because the shift operation causes bits from other bytes to be thrown away (since an h can only hold 32 bits).

- Example 2: Let string hash function be :

$$S_{n-1} + 31^{1}s_{n-2} + \ldots. + 31^{n-2}s_1 + 31^{n-1}\ s_0$$

```
h = 0
for (int i = 0; i < n; i--) {
   h = 31*h + s_i
}
```

"hello" = 104,101,108,108,111

$111 + 31^{1}*108 + 31^{2}*108 + 31^{3}*101 + 31^{4}*104 = 99162322$

Note: overflow bits are ignored

This is a good algorithm and it is used Java

4.Java class library: java.util.HashMap<K,V>

- http://download.oracle.com/javase/6/docs/api/java/util/HashMap.html
- Several constructor available to specify capacity (table size) and load factor (to rehash)
- Use separate chaining strategy
- Search key class has a method to compute hash code, i.e. key.hashCode()

- Example:

Note: Integer & String have hashCode() method

```
// Statistics.java
// Simple demonstration of HashMap.
// From 'Thinking in Java, 3rd ed.' (c) Bruce Eckel 2002
// www.BruceEckel.com. See copyright notice in CopyRight.txt.
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Random;

// value class
class Counter {
      int i = 1;
      public String toString() {
               return Integer.toString(i);
      }
}

// <K,V> = <Integer, Counter>
public class hashMapExample{
         private static Random rand = new Random();

         public static void main(String[] args) {
              Map<Integer, Counter> hm =
                       new HashMap<Integer, Counter> ();
              for (int i = 0; i < 1000; i++) {
                   // Produce a number between 0 and 20:
                   Integer r = new Integer(rand.nextInt(20));
                   if (hm.containsKey(r))  // key already in table
                            ((Counter) hm.get(r)).i++;
                   else       // new key
                            hm.put(r, new Counter());
              }
              System.out.println(hm);
         }
}
```

```
$ java hashMapExample

{0=63, 1=57, 2=44, 3=42, 4=43, 5=55, 6=56, 7=46, 8=45, 9=36, 10=51, 11=56,
12=64, 13=44, 14=50, 15=46, 17=58, 16=52, 19=47, 18=45}
```