Topic 3

MAPs and Hash Tables

Aims

- To present the ***Map*** model that is defined to solve problems of dynamic search efficiently
- To study the ***Hash Table*** as an efficient epresentation of the *Map* model, taking especially into consideration the following aspects:
  - The concepts related to its definition : *hash* function, conflicts (*collisions*) and their solution
  - The analysis of its efficiency, measured as its *load factor*
  - Implementation of the class *TablaHash* with separate chaining

Contents

1. The *Map* model
Introduction

- The *Map* model is designed to ease data search in a collection (in general repeated data are not allowed)
- The data that are stored in a *Map* are key-value pairs, where:
  - The search is carried out depending on the key: the method *equals* will have to allow to check whether two keys are equals or not
  - The value is the information associated at the key that we aim at retrieving
- The basic operation of a *Map* is searching by key (or name) in a collection of entries

1. The *Map* model
Methods

- The functionality of the *Map* model can be observed via the following interface Java:

```java
public interface Map<C, V> {
  // Add the entry (c,v) and return the old value that this key had
  // (or null if it had no associated value)
  V insertar(C c, V v); // insert

  // Delete the entry with key c and return its associated value
  // (or null if there is no key with the key c)
  V eliminar(C c); // delete

  // Search for the key c and return its associated value
  // (or null if there is no key with the key c)
  V recuperar(C c); // retrieve

  // Return true the Map ia empty
  boolean esVacio(); // isEmpty

  // Return the number of entries of Map
  int talla(); // size

  // Return a List with Point of INterest with the keys of all entries
  // of the Map
  ListaConPI <C> claves(); // keys
}
```

1. The *Map* model

Using the model (I)

- There exist many applications that use *Maps* and of them is the translation of texts. A simple example is the design of a word to word translator from Spanish to English.

- Exercise: implement the following method :

```
public static String translate(String textSpanish,
   Map<String,String> map)
```

Considering that the key in *map* is the word in Spanish and the value is its translation in English. The method translate returns a chain with the translation in English, word by word, of the chain *textSpanish*. If a word is not in *map* the method will have to substitute it with “<error>” in the out put chain.

1. The *Map* model

Using the model (II)

- Another application is computing the frequency of occurrence of the elements of a collection.

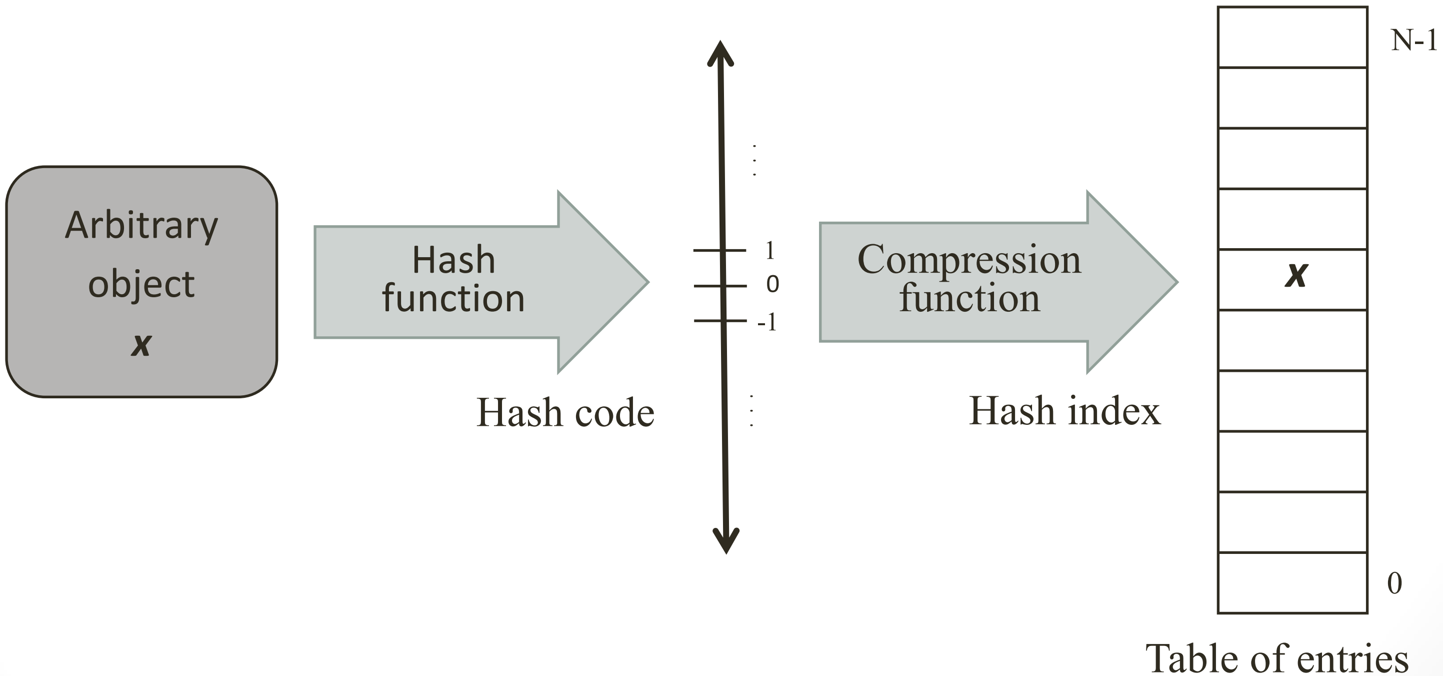
- Exercise: using the interface *Map,* and having the class *TablaHash* that implements it, design a program that reads a text from the keyboard and returns the number of different words of the text togethr with their frequency of occurrence.

  Note: the frequency is calculated as the number of times that a word appears in the text divided by the total number of words of the text.

2. Hash table

2.1. The concept of hash

- Data structure designed for the implementation of *Maps* (operations: *retrieve*, *insert* and *delete* in a time O(1))



2. Hash table

2.2. Hash function– Simple method

- Definition: function that converts an entry in an int (*hash code*) appropiate to index the table in which this entry will be stored

- Simple method: sum of components. Example:
  - An entry is a word in Spanish (key) together with its transation in English (value)
  - We aim to store the collection of entries in an array
  - In order to know in what position of the array to store each entry, we can sum the ASCII codes of the characters of its key:

key of the entry		**hash code**
casa	→ $99 + 97 + 115 + 97$	$= 408$
hola	→ $104 + 111 + 108 + 97$	$= 420$

2. Hash table

2.2. Polynomial hash function

- the sum of components is not a good hash function since it is easy that two different entries have the same hash code (**collision**):

hola → $104 + 111 + 108 + 97 = 420$

teja → $116 + 101 + 106 + 97 = 420$ **collision**

- Polynomial functions : to improve the quality of the hash function, it is possible to weight the position of each character of the key:

$f(c) = c_0 \cdot a^{k-1} + c_1 \cdot a^{k-2} + \ldots + c_{k-2} \cdot a^1 + c_{k-1}$, with a>1.

Example with a=2

hola → $104 \cdot 2^3 + 111 \cdot 2^2 + 108 \cdot 2 + 97 = 1589$

teja → $116 \cdot 2^3 + 101 \cdot 2^2 + 106 \cdot 2 + 97 = 1641$

2. Hash table

2.2. The method hashCode of Java

```
public int hashCode();    // defined in the class Object
```

- Every class that is going to be used as key in a *Map* must overwrite properly the above method.
- The class *String* implements a polynomial hash function with base 31:

$$hashCode = \sum_{i=0}^{length-1} charAt(i) \bullet base^{length-1-i}$$

2. Hash table

2.3. Compression functions

- The *hash* code can be a value greater than the size of the *array*. It can be also a negative number.
- Compression function: it converts an *hash* code in an **hash index** between 0 and the size of the *array* minus one.
- Method of the division:

hashIndex = hashCode % sizeOfArray

if (hashIndex < 0) hashIndex += sizeOfArray;

When the hash code is negative

2. Hash table
2.4. Collisions

- The hash function returns always the same value for the same entry (or for two entries which are equal according to *equals* method).
- If two entries are different, then the hash function should return two different values. Although this is not strictly necessary, this feature improves the efficiency of hash tables.
- Even with a good hash function, collisions can occur $\Rightarrow$ we need efficient methods to solve collisions:
  - Open addressing
  - Separate chaining

2. Hash table

2.4. Collisions – Open addressing

- If we are going to insert an element in a specific position which is already taken, we search for an alternative position.
- The **linear exploration** solves a collision searching sequentially, starting from *hashIndex* until the next free position in the table:

  hashIndex + i , i-th collision.

  - Problem of primary clustering

- The **quadratic exploration** solves a collision checking the positions (implementing circularity) :

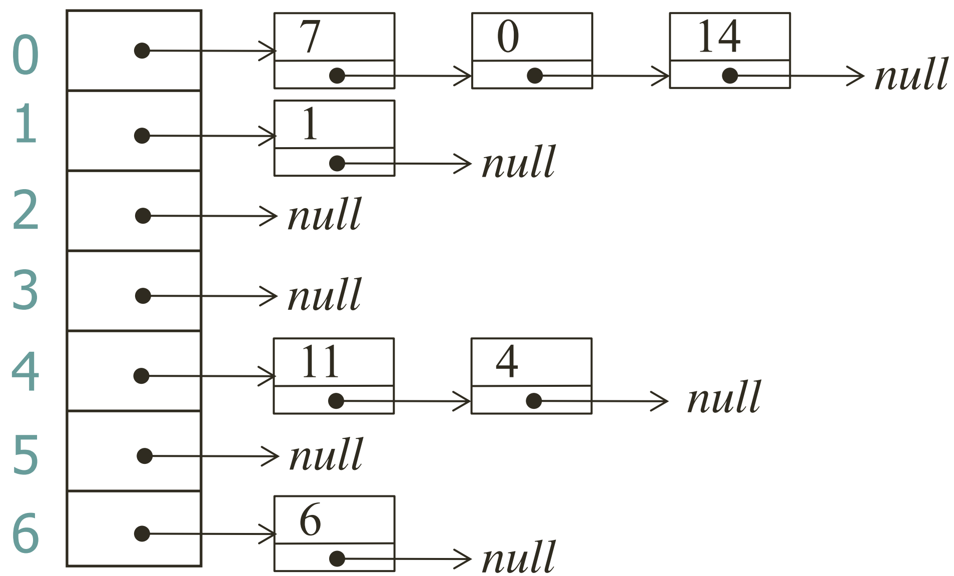
  hashIndex$+1^2$, *hashIndex*$+2^2$, ..., *hashIndex*$+i^2$

  - No primary clustering, although there is secondary clustering

2. Hash table

2.4. Collisions – Separate chaining

- All the entries which collide in the same position are stored in a linked list.
  - Each list is called **bucket**



2. Hash table

2.5. Load factor

- The performance of an hash table is measured in terms of its **load factor**, which is defined as the average length of its buckets: LF = actualSizeOfTable / sizeOfArray
- Therefore, the efficiency of a hash table depends on:
  - The quality of its **hash function**:

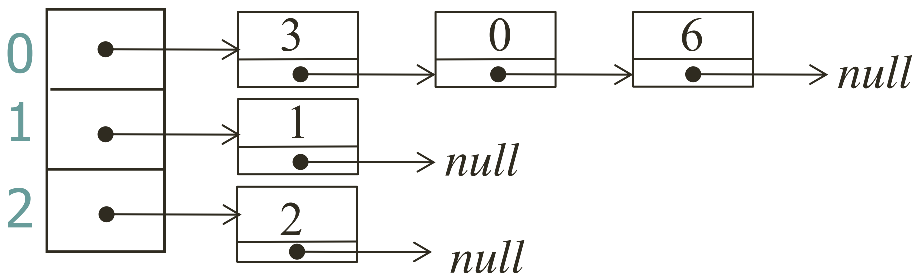
    Better hashing → less collisions
  - Its **load factor**:

    Fuller table→ more collisions
  - Its method to **solve collisions**

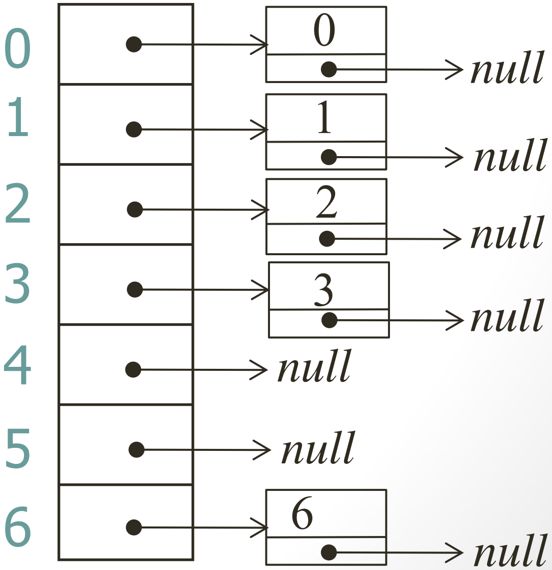
2. Hash table
2.5. Rehashing

- The number of collisions can grow too much if the Load Factor (or occupancy rate) is very high. The ***rehashing*** consists in increasing the size of the hash table, reducing its occupancy rate

LF = 5/3 = 1.67



LF = 5/7 = 0.71



3. Implementation

Entries for the Hash table

- It is necessary to define a generic class that stores together the key and the value of an entry:

```
class EntradaHash<C, V> {
  C clave;                                    // Key of the entry
  V valor;                                    // Value of the entry

  public EntradaHash(C clave, V valor) {
    this.clave = clave;
    this.valor = valor;
  }
}
```

3. Implementation

The class TablaHash: attributes and constructor

- The constructor receives the estimated number of elements to allocate and saves space to store them with a LF of 75%
- It is highly recommended that the **size** of the *array* is a **prime number** in order to improve the hashing of the elements

```
public class TablaHash<C, V> implements Map<C, V> {
  // Array of LPIs
  private ListaConPI<EntradaHash<C,V>> elArray[];
  // Number of elements stored in the table
  private int talla;

  public TablaHash(int tallaMaximaEstimada) {
    int capacidad = siguientePrimo((int)
                    (tallaMaximaEstimada/0.75));
    elArray = new LEGListaConPI[capacidad];
    for (int i = 0; i < elArray.length; i++)
       elArray[i] = new LEGListaConPI<EntradaHash<C,V>>();
    talla = 0;
  }
```

3. Implementation

Search of the position of an element in the table

```
/** It calculates the bucket for an element with key c.
 *  First it obtains the hash value (hashCode) and
 * after its hash index
 *  @param c   Key of the element to search
 *  @return    Bucket where the element is
 */
private int indiceHash(C c) {
  int indiceHash = c.hashCode() % this.elArray.length;
  if (indiceHash < 0)
      indiceHash += this.elArray.length;
  return indiceHash;
}
```

3. Implementation

Inserting an entry in the table

```
// It adds the entry(c,v) and returns the old value
// of the given key (or null if the key dis not have
// any associated value)
public V insertar(C c, V v) {
  V oldValue = null;
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> bucket= elArray[pos];
  //Search for the entry of key c in the bucket
  for (bucket.inicio(); !bucket.esFin() &&
  !bucket.recuperar().clave.equals(c); bucket.siguiente());
    if (bucket.esFin()) {// Insert the entry if there is not
     bucket.insertar(new EntradaHash<C,V>(c, v));
      talla++; // Rehashing depending on LF
  } else {//If the entry was in the bucket, update its value
      oldValue = bucket.recuperar().valor;
      bucket.recuperar().valor = v;
  }
  return oldValue;
}
```

3. Implementation

Deleting an entry form the table

```
// It deletes the entry with key c and returns its
// associated value (or null if there is no entry
// with this key)
public V eliminar(C c) {
 int pos = indiceHash(c);
 ListaConPI<EntradaHash<C,V>> bucket = elArray[pos];
 V value = null;
 // Search for the entry of key c in the bucket
 for (bucket.inicio(); !bucket.esFin() &&
 !bucket.recuperar().clave.equals(c);bucket.siguiente());
  if (!bucket.esFin()) {// If we find it, we delete it
      value = bucket.recuperar().valor;
      bucket.eliminar();
      talla--;
  }
  return value;
}
```

3. Implementation

Search of entries, isEmpty and size

```
// It searches tfor he key c and returns its associated
// info or null an entry with such a key does not exist
public V recuperar(C c) {
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> bucket= elArray[pos];
  // Search for the entry of key c in the bucket
  for (bucket.inicio(); !bucket.esFin() &&
     !bucket.recuperar().clave.equals(c);
      bucket.siguiente());
  if (bucket.esFin()) return null; // Not found
  else return bucket.recuperar().valor; // Found
}

// It returns true if the Map is empty
public boolean esVacio() { return talla == 0; }

// It returns the number of entries in the Map
public int talla() { return talla; }
```

References

- Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java (4th edition)*. John Wiley & Sons, Inc., 2005.
  - Chapter 9, section 1 and 2.
- Weiss, M.A. Data Structures *in Java*. Adisson-Wesley, 2000.
  - Chapter 6, section 7, and chapter 19.