DATA STRUCTURES AND ALGORITHMS

Practice 2: Hash Tables

2021-2022

1. Introduction

With this practice it is expected to achieve the following objectives:

- Implement a hash table with collisions resolution by chaining.
- Implement a mechanism to dynamically expand hash tables.
- Learn how to use inner classes to improve encapsulation.
- Implement an array that dynamically expands.
- Read a text file and register in which positions appears each word.

Warning:

In the source code of the practice there will be a Map interface and a HashMap class. Both the interface and the class already exist in the Java API package 'java.util'. Therefore, it is important not to import them from this package to avoid confusion.

2. ADT dictionary (*Map*)

A dictionary is a data structure that stores a collection of key-value pairs where duplicated keys are not allowed. This ADT, when presented with a key, will return the associated value.

The following code snippet contains the Map interface which you must implement. This interface inherits from Iterable to allow looping through the elements of the set with a for-each loop.

```
package p1;

public interface Map<K,V> extends Iterable<K>
{
    void put(K key, V value);
    V get(K key);
    V remove(K key);
    boolean contains(K key);
    int size();
    boolean isEmpty();
    void clear();
}
```

An example of using a dictionary can be seen in the following code snippet:

```
Map<Integer,String> m = ...   // create a dictionary
m.put(2, "dos");
m.put(3, "tres");
m.put(5, "cinco");
...
System.out.println(m.get(2)); // prints "dos"
System.out.println(m.get(3)); // prints "tres"
System.out.println(m.get(5)); // prints "cinco"
...
System.out.println(m); // prints "{2=dos, 3=tres, 5=cinco}"
```

The previous example shows that it is possible to program against an interface without any problem, but you cannot create an object to work with if there is no implementation of that interface. For this reason, there are ... in the first sentence.

With the method toString() you can override the default implementation, inherited from the Object class, so that it returns a text string that represents the elements of the dictionary in sequential order (i.e. "{2=dos, 3=tres, 5=cinco}").

3. HashTable

To implement the Map interface the following schema can be used:

```
package p1;

import java.util.Iterator;

public class HashMap<K,V> implements Map<K,V>
{
    public HashMap()
    {
        this(16, 0.75f);
    }

    public HashMap(int capacity, float loadFactor)
    {
        ...
    }

    @Override public Iterator<K> iterator()
    {
        return new CIterator();
    }
    ...

    private static class Node<K,V>
    {
        ...
    }

    private class CIterator implements Iterator<K>
    {
        ...
        @Override public boolean hasNext() { ... }
        @Override public K next()          { ... }
        ...
    }
}
```

As the `Map interface inherits from Iterable`, the iterator function must be implemented, which simply must return a new object of the CIterator class to allow looping through the elements of the dictionary.

3.1. Inner Classes

The Node and CIterator classes are defined within the HashMap class. Bothc lasses are inner classes. The first one has the static keyword, while the second does not.

When an inner class is static, its objects cannot access the enclosing class, unless it has a reference. Also, an object of a static inner class can exist on its own without the need for an object of the enclosing class.

If an inner class is not static, its objects must be created from an object from the enclosing class. In addition, the objects of the inner class will have a reference to the object of the enclosing class, so they will be able to access that object. This behavior is appropriate to the CIterator class, because it needs to access the HashMap data to be able to looping through its elements.

When an inner class does not need to access the data of the enclosing class, it is convenient to make it static to prevent its objects from having a reference to the enclosing class, and thus save memory

4. Bucket Array (nodes)

As the nodes of the hash table are generic, it will be necessary to create the array in a special way so that the compiler does not give an error:

```
@SuppressWarnings("unchecked")
private Node<K,V>[] newArray(int length)
{
    return (Node<K,V>[])new Node[length];
}
```

The @SuppressWarnings annotation lets you tell the compiler that it doesn't need to warn us of a possible incorrect casting because it is known to be okay.

4.1. Bucket array indexes

The index of a bucket can be calculated from the hash code as follows:

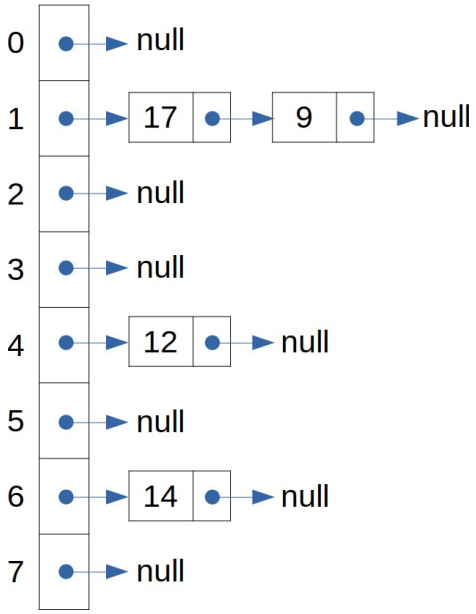
```
hasCode % buckets.length
```

Where '%' is the modulo operation. This calculation is expensive, but can be optimized if the size of the array is a power of 2 using the bitwise operator '&' like this:

```
hashCode & (buckets.length - 1)
```

4.2. Collision resolution by chaining

In case of collision when adding an item, the new item will be added to the corresponding bucket list, if it is not already present.



In this example, the values 9, 12,and 14 were added first.

When 17 was added, a collision occurred and new node 17 was added ahead of 9.

5. Hash Table expansion

One of the main problems when working with a hash table is to choose the appropriate number of buckets m with respect to the forecast of the number of elements it will store, to minimize the number of collisions and guarantee a constant cost in insert and search operations.

To decide when to increase the number of buckets you should use the concept of the load factor α of the hash table:

$$\alpha = n / m$$

where n is the number of elements contained in the table, and m is the number of available buckets.

So we can calculate the maximum number of elements for the hash table to work properly:

$$n = m * \alpha$$

If before adding an element the total number exceeds the calculated maximum value, then it is time to expand the number of buckets.

5. Array

The Array class must behave similarly to ArrayList (from Java API). The inner array, which stores the elements, must expand when its full.

Following methods must be implemented:

```
public class Array<E> implements Iterable
{
    ...
    public void add(E element)            { ... }
    public E    get(int index)            { ... }
    public void set(int index, E element) { ... }
    public int  size()                    { ... }
    public void clear()                   { ... }
    @Override public String toString()    { ... }
    @Override public Iterator iterator()  { ... }
    ...
}
```

6. Reading a text file

The next snippet can be used to read a text file line by line:

```
BufferedReader in = new BufferedReader(
                    new FileReader("fichero.txt"));
```

In order to fulfill this practice is needed to design an algorithm that reads a text file word by word and calculates their positions.

To read a line:

```
String line = in.readLine();
```

To check is a character is a letter:

```
Character.isLetter(char ch);
```

The algorithm must transform to lowercase (toLowerCase) every word to count them equally despite their capitalization.

7. Javadoc

Classes must be documented using Javadoc.

In case a function of a subclass has the same documentation as the function of the superclass that it overwrites, the following comment can be used to avoid repeating the text:

```
/**
 * {@inheritDoc}
 */
public V put(K key, V value)
{
    ...
}
```

The following document explains how to document code with Javadoc:

[How to Write Doc Comments for the Javadoc Tool]

6. Deliverables

To obtain the highest mark, the following work must be done:

- `Implementation of the HashMap<K,V> class`.
- Implementation of the `Array<E> class`.
- Implementation of the `toString` method for the previous classes.
- Executable class `Test` that checks the correct behaviour of the previous classes
- Executable class `Main` which takes a filename as an parameter, analyzes the file and shows for every word in which rows and columns appears. It is mandatory to use a HashTable with the followinf type:

  ```
  HashMap<String,Array<Posicion>> map = ...
  ```

- The `Posicion class is optional`. The position can be stored as a string "(row:column)".
- Document every class using JavaDoc.

The source code (.java files) must be delivered packaged in a single compressed file (.zip, .tgz, etc.) through a Poliformat task.

All classes (.java files) must be in the package `p1`:

```
package p1;  // lowercase p!
...
```

6.1. Execution example

As HashTable does not guarantee any order while looping over the keys they result may be unordered as in the following example:

```
$ java p1.Main file.txt

private : [(5:1)]
package : [(1:1)]
public : [(3:1), (7:1), (13:11)]
column : [(5:25), (7:31), (10:10), (10:20), (15:28)]
tostring : [(13:25)]
override : [(13:2)]
int : [(5:15), (7:17), (7:27)]
return : [(15:5)]
p : [(1:9)]
string : [(13:18)]
final : [(5:9)]
class : [(3:8)]
row : [(5:19), (7:21), (9:10), (9:20), (15:17)]
position : [(3:14), (7:8), (18:6)]
this : [(9:5), (10:5)]
```