Estructura de Datos

UNIVERSITAT ROVIRA i VIRGILI

Práctica 1

Lista doblemente encadenada y

Tabla de Hash

Alex García Amezcua

ÍNDICE

Enunciado

Esta práctica consiste en la comprensión, implementación y análisis de dos tipos de estructuras de datos.

Las estructuras que se han de implementar son una lista doblemente encadenada y una tabla de hash, ambas haciendo uso de memoria dinámica salvo en el caso específico de la tabla de hash donde se utilizará memoria estática y dinámica “a la vez”.

También es necesario que sean capaces de tratar con un tipo de datos genérico T que implementará la interfaz Comparable, que nos permitirá hacer búsquedas independientemente del tipo.

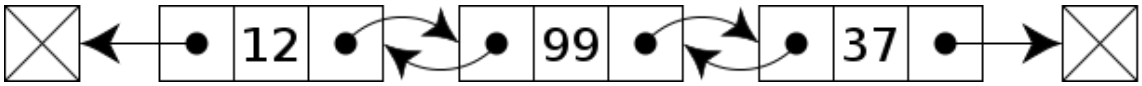
Por último, una vez validado el funcionamiento de ambas estructuras con diferentes juegos de pruebas, haciendo uso de un TAD propio, será necesario realizar un estudio del coste computacional de la operación de búsqueda y comparar los resultados obtenidos con los esperados.

Análisis estructura del proyecto

Dentro de la carpeta del proyecto se encuentra una carpeta para toda la primera práctica que se utiliza de esta manera para poder reutilizar la carpeta del proyecto en la siguiente práctica.

Una vez dentro encontramos una organización muy básica, hay 3 carpetas más en las que se encuentran todos los juegos de pruebas y “mains” del proyecto, otra donde se encuentran todas las estructuras programadas tanto para la primera parte como para la segunda y por último las excepciones programadas para esta práctica.

Análisis Lista doblemente encadenada



La lista doblemente encadenada esta compuesta de cuatro archivos fuente:

- DLL, describe el contrato base proporcionado en el enunciado para esta práctica y que estará implementado por la lista doblemente encadenada.

- Node, tiene las referencias a los nodos siguiente y anterior (como se define a una lista doblemente encadenada, en caso de ser sólo encadenada, tendría una única referencia al siguiente nodo) y el valor genérico T que se quiera guardar en ese nodo.

- DoublyLinkedList, tiene las referencias al primer y último nodo de la estructura, además de implementar todas las operaciones definidas en el contrato DLL.

- TIterator, implementa la interfaz Iterator<T> que nos permitirá iterar por la estructura de manera más cómoda.

Código:

insert(pos, data): de manera parecida a insertar al final, salvo que en esta operación es necesario atravesar la lista hasta la posición especificada, si esta posición es más grande que el tamaño de la lista, salta una de las excepciones programadas, sino se añadirá de manera similar a insertar al final pero actualizando las referencias “prev” y “next” de los dos nodos entre los que se encuentre.

get(pos): atraviesa la lista de manera similar a insert, comprobando si la posición en la que se encuentra es la que se pide, a la vez que comprobando si está en el final de la lista. Cuando sale de este bucle solo hay dos posibilidades, ha llegado a la posición que se pedía o ha atravesado toda la lista y no ha llegado a dicha posición. Si no se ha encontrado el elemento, salta una de las excepciones programadas.

remove(pos): atraviesa la lista hasta la posición pedida, seguidamente se hacen una serie de comprobaciones dependiendo de la posición en la que se encuentre para reasignar la referencia “next” del anterior nodo al siguiente y la referencia “prev” del siguiente nodo al anterior.

search(data): devuelve la cantidad de nodos accedidos para encontrar el valor pedido, esto se resuelve atravesando la lista y sumando a un contador cost a cada iteración hasta encontrar el elemento deseado o llegar al final de la lista. Si no se encuentra el elemento o se llega al final de la lista, se lanza una excepción que recibe por parámetro el contador cost – 1 dado que sabemos que para llegar a una de estas condiciones ha sido necesario acceder a un supuesto último nodo de valor "null" el cual no es necesario contar.

Juego de pruebas LDE

Para el juego de pruebas de la lista doblemente encadenada, he decidido que la mejor manera de testear todos los métodos era con diferentes ejemplos del TAD Ciutadà y utilizando todos los métodos de distintas formas para comprobar la tolerancia a fallos de la estructura.

Para ello, ejecuto distintos métodos de la estructura sobre diferentes "ciutadans" intentando que éstos den algún tipo de fallo y comprobando que el uso de las excepciones es el correcto.

Cabe mencionar el método estático showlist que facilita y acorta el juego de pruebas para mostrar por pantalla el contenido de la lista utilizando un foreach dado que la lista es iterable.

Ejemplos:

```
Ciutada c1 = new Ciutada( n: "Ruby",  c: "Lawrence",  d: "19632780A");
Ciutada c2 = new Ciutada( n: "Elena",  c: "Lawrence",  d: "35970581F");
Ciutada c3 = new Ciutada( n: "Jean",  c: "Cooper",  d: "12408979L");
Ciutada c4 = new Ciutada( n: "Ruby",  c: "Campbell",  d: "12408979L");
Ciutada c5 = new Ciutada( n: "Troy",  c: "Alvarez",  d: "89703330H");
Ciutada c6 = new Ciutada( n: "Luna",  c: "Wong",  d: "19632780A");

ciutadans.insert(c1);
ciutadans.insert(c2);
ciutadans.insert(c3);
ciutadans.insert(c4);

try{
    ciutadans.insert( pos: 2, c5);
    ciutadans.insert( pos: 1, c6);
} catch (SizeException e){
    System.out.println(e.getMessage());
}

System.out.println("List initially...");

showList(ciutadans);
```

Añade los primeros 4 ciutadans al final y los dos últimos en las posiciones 2 y 1 respectivamente, como resultado se obtiene lo siguiente:

```
List initially...
Nom: Ruby   Cognom: Lawrence    DNI: 19632780A
Nom: Luna   Cognom: Wong    DNI: 19632780A
Nom: Elena  Cognom: Lawrence    DNI: 35970581F
Nom: Troy   Cognom: Alvarez DNI: 89703330H
Nom: Jean   Cognom: Cooper  DNI: 12408979L
Nom: Ruby   Cognom: Campbell    DNI: 12408979L
```

Aún repitiendo nombre, apellido o dni, ambos métodos de inserción funcionan correctamente.

Si se intenta añadir en una posición de la lista inexistente se lanza la excepción que se puede captar con la sentencia catch, como en la siguiente imagen:

```
try{
    ciutadans.insert( pos: 8, c4);
}
catch (SizeException s){
    System.out.println(s.getMessage());
}
```

Resultado:

```
Trying to add to a position bigger than the list size...
Error : No es pot afegir l'element
Size after failed insertion: 6
```

Eliminar nodos puede ser un caso problemático en esta estructura, pero es aun más delicado si hablamos del primero o último nodo dado que son las únicas referencias que tenemos para acceder a cualquier otro nodo de la lista.

```
List after removing first element...
Nom: Luna   Cognom: Wong    DNI: 19632780A
Nom: Elena  Cognom: Lawrence    DNI: 35970581F
Nom: Troy   Cognom: Alvarez DNI: 89703330H
Nom: Jean   Cognom: Cooper  DNI: 12408979L
Nom: Ruby   Cognom: Campbell    DNI: 12408979L
Size after removing: 5
```

Si intentamos obtener elementos en posiciones existentes (4) y otras más grandes que el tamaño de la lista (7):

```
Element at position = 4: Nom: Ruby  Cognom: Campbell    DNI: 12408979L
Error : No s'ha pogut trobar l'element
```

Si intentamos buscar elementos en la estructura que existen y otro que no existe obtenemos lo siguiente:

```
Searching for citizens Elena, Ruby and other citizen...
Items checked: 2
Items checked: 4
Error : No s'ha trobat l'element
S'han accedit a 5 elements
```

Análisis Tabla de Hash

La tabla de hash está compuesta de tres archivos fuente:

- HashTableContract, describe el contrato base proporcionado en el enunciado para esta práctica y que estará implementado por la tabla de hash.

- HashNode, tiene la referencia al siguiente nodo y los valores genérico K y T que se quiera guardar en ese nodo.

- HashTable, tiene la estructura estática de nodos, además de implementar todas las operaciones definidas en el contrato HashTableContract.

Código:

hashFunc(size, key): función de hashing no inyectiva que suma de 4 bytes en 4 bytes y que devuelve una posición válida de la estructura “hashTable”.

insert(key, data): esta función primero comprueba si es necesario redimensionar la tabla estática según el valor del factor de carga y después hace uso de la función de hashing para obtener una posición válida de la tabla de hash y llama al método privado “insertion” el cuál tiene el código principal para añadir un nuevo nodo en un índice determinado de una lista que se pasa todo por parámetro. La existencia de este método tiene como objetivo no repetir código en la inserción simple o redimensionamiento.

remove(key): como en la mayoría de los métodos, lo primero es obtener el índice de la tabla de hash relativo a la clave que se quiere eliminar. A partir de aquí se realizan varias comprobaciones, si en esa posición no existe ningún nodo, si justamente el primer nodo de esa posición es el elemento buscado se reasigna el valor de esa posición al siguiente nodo. Por último, si encontramos varias colisiones, iteramos hasta llegar al final o hasta encontrar la clave, al encontrar la clave se asigna la referencia next del anterior nodo que el eliminado al siguiente del eliminado.

Juego de pruebas TH

De manera similar al juego de pruebas de la Lista doblemente encadenada, se crean varias instancias de “Ciutada” que más adelante se utilizan para comprobar todas las funciones de la Tabla de Hash y testear su tolerancia a fallos y uso de excepciones.

Es necesario mencionar también la existencia del método auxiliar showTable() implementado en la clase HashTable para facilitar la visualización del contenido de la lista.

De la misma manera que en la validación de la Lista doblemente encadenada, lo primero es crear ciutadans y añadirlos a la tabla de hash. En este caso en cuanto a problemas solo tendríamos un posible fallo y se daría cuando se excediera el factor de carga de la tabla en el cual o saltaría la excepción adecuada o en mi caso se haría un redimensionamiento que se explica más adelante.

Es por esta misma razón que aun inicializando la tabla de hash a tamaño 5, al añadir 6 elementos no resulta en ningún problema y lo realiza correctamente.

```
HashTable<String, Ciutada> ciutadans = new HashTable<>( dim: 5);

Ciutada c1 = new Ciutada( n: "Kratos",  c: "Lawrence",  d: "19632780A");
Ciutada c2 = new Ciutada( n: "Elena",  c: "Lawrence",  d: "35970581F");
Ciutada c3 = new Ciutada( n: "Jean",  c: "Cooper",  d: "12408979L");
Ciutada c4 = new Ciutada( n: "Ruby",  c: "Campbell",  d: "54368005F");
Ciutada c5 = new Ciutada( n: "Troy",  c: "Alvarez",  d: "89703330H");
Ciutada c6 = new Ciutada( n: "Luna",  c: "Wong",  d: "29309153T");

try {
    ciutadans.insert(c1.getDNI(), c1);
    ciutadans.insert(c2.getDNI(), c2);
    ciutadans.insert(c3.getDNI(), c3);
    ciutadans.insert(c4.getDNI(), c4);
    ciutadans.insert(c5.getDNI(), c5);
    ciutadans.insert(c6.getDNI(), c6);
}
catch(SizeException e){
    System.out.println(e.getMessage());
}
System.out.println("\nList initially...");

ciutadans.showTable();

System.out.println("\n----------------------------------------\n");
```

Como se ha mencionado previamente, se usa el método showTable para visualizar el contenido de la tabla, este método va accediendo primero de arriba abajo y después de izquierda a derecha, es decir, primero accede a una posición de la tabla y después a los nodos de la posición y así sucesivamente hasta el final de la tabla. Por lo tanto, como en una tabla de hash no se inserta ordenadamente sino según la posición que "manda" la función de hash, su contenido no estará ordenado de la misma manera que en su inserción. Vemos el ejemplo en la siguiente imagen:

```
List initially...
Key [19632780A] Value [Nom: Kratos  Cognom: Lawrence    DNI: 19632780A]
Key [12408979L] Value [Nom: Jean    Cognom: Cooper  DNI: 12408979L]
Key [54368005F] Value [Nom: Ruby    Cognom: Campbell    DNI: 54368005F]
Key [29309153T] Value [Nom: Luna    Cognom: Wong    DNI: 29309153T]
Key [35970581F] Value [Nom: Elena   Cognom: Lawrence    DNI: 35970581F]
Key [89703330H] Value [Nom: Troy    Cognom: Alvarez DNI: 89703330H]
```

Se utiliza el método implementado size, el cual devuelve únicamente la cantidad de elementos en la estructura y no el tamaño de esta. En este caso la estructura tiene un tamaño de 10 (dado al redimensionamiento en la inserción anterior) pero la cantidad de elementos en la estructura solo es de 6.

```
----------------------------------------

Size: 6

----------------------------------------
```

Una de las problemáticas al insertar un elemento también podía ser que añadiésemos un elemento cuya clave ya existía, en dicho caso solo seria necesario actualizar el valor en vez de añadir un nuevo nodo en la estructura. Vemos también que esto funciona correctamente no solo por el contenido sino porque el tamaño de la estructura continúa siendo de 6.

```
----------------------------------------

Trying to add a citizen with a same dni...
Key [19632780A] Value [Nom: John    Cognom: Cena    DNI: 19632780A]
Key [12408979L] Value [Nom: Jean    Cognom: Cooper  DNI: 12408979L]
Key [54368005F] Value [Nom: Ruby    Cognom: Campbell    DNI: 54368005F]
Key [29309153T] Value [Nom: Luna    Cognom: Wong    DNI: 29309153T]
Key [35970581F] Value [Nom: Elena   Cognom: Lawrence    DNI: 35970581F]
Key [89703330H] Value [Nom: Troy    Cognom: Alvarez DNI: 89703330H]

----------------------------------------

Size: 6

----------------------------------------
```

Si intentamos buscar un elemento en la estructura por una clave existente y otra no existente obtenemos lo siguiente:

```
----------------------------------------

Getting key = 29309153T
Nom: Luna   Cognom: Wong    DNI: 29309153T

----------------------------------------

Getting unknown key = 12345678T
Error : No s'ha pogut trobar l'element

----------------------------------------
```

Al intentar eliminar un elemento existente de la estructura, visualizar su contenido y comprobar su tamaño obtenemos el siguiente correcto resultado:

```
----------------------------------------

Trying to remove Ruby...
Key [19632780A] Value [Nom: John    Cognom: Cena    DNI: 19632780A]
Key [12408979L] Value [Nom: Jean    Cognom: Cooper  DNI: 12408979L]
Key [29309153T] Value [Nom: Luna    Cognom: Wong    DNI: 29309153T]
Key [35970581F] Value [Nom: Elena   Cognom: Lawrence    DNI: 35970581F]
Key [89703330H] Value [Nom: Troy    Cognom: Alvarez DNI: 89703330H]

----------------------------------------

List size: 5

----------------------------------------
```

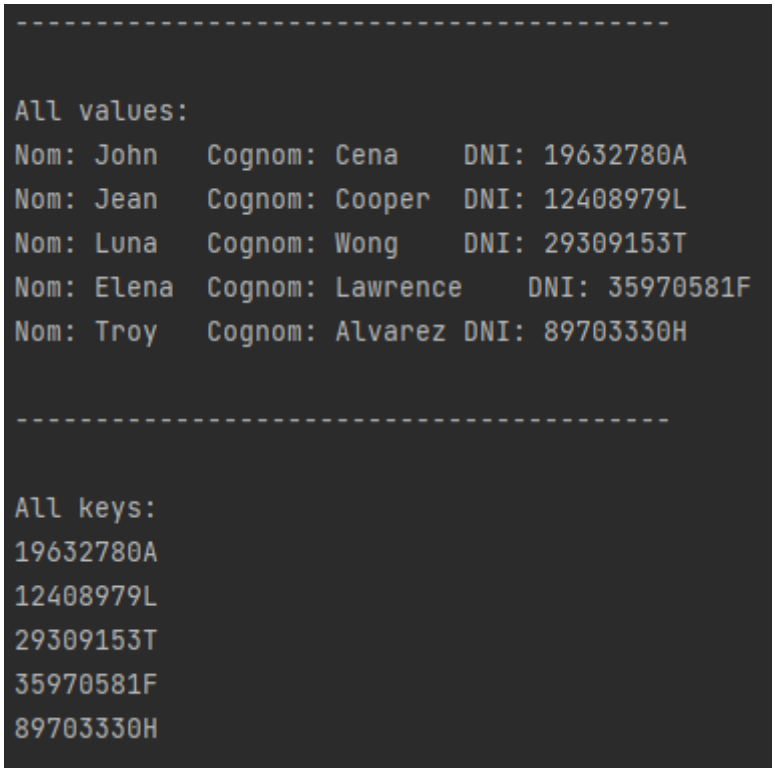
Por la formula del calculo del factor de carga (# elementos / tamaño tabla), sabemos que el factor de carga debería ser de 0,5 (size ha devuelto una cantidad de elementos de 5 y por el redimensionamiento al inicio tenemos una tabla de tamaño 10).

```
----------------------------------------

Load Factor: 0.5

----------------------------------------
```

Si buscamos 2 DNIs, el primero sabemos que existe y el segundo no existe, este método debería devolver la cantidad de elementos accedidos para encontrarlos o por el contrario para saber con certeza que no existe en la estructura.

```
----------------------------------------

Cost of searching for Luna's DNI: 2

----------------------------------------

Cost of searching for unknown DNI:
Error : No s'ha trobat l'element
S'han accedit a 3 elements

----------------------------------------
```

Por ultimo los métodos mas sencillos que devuelven un lista de valores, en este caso cada Ciutada que se encuentre en la estructura y una lista de claves, en este caso los DNIs.



Redimensión Tabla de Hash

En esta práctica es necesario aumentar el tamaño de la tabla estática una vez se sobrepasa el 75% de carga de la estructura, para hacerlo he implementado el método privado resize el cual crea una tabla igual a la ya existente, pero con el doble de tamaño.

Al implementarlo, me di cuenta de que se repetía gran parte del código de insertar, dado que es necesario obtener cada elemento existente de la tabla pequeña e insertarlo en la tabla redimensionada, así que se crea un método privado mas para insertar un nodo en un índice de una tabla, todo pasado por parámetro.

En resize, se itera sobre la estructura vieja buscando por elementos que relocalizar en la nueva estructura. Una vez encontrado uno, se crea un nodo, un índice relativo al tamaño de la nueva tabla y se llama a insertion que lo inserta en esta nueva estructura y así para todos los elementos. Una vez no hay mas elementos, se asigna esta nueva estructura a la vieja y así se finaliza el redimensionamiento.

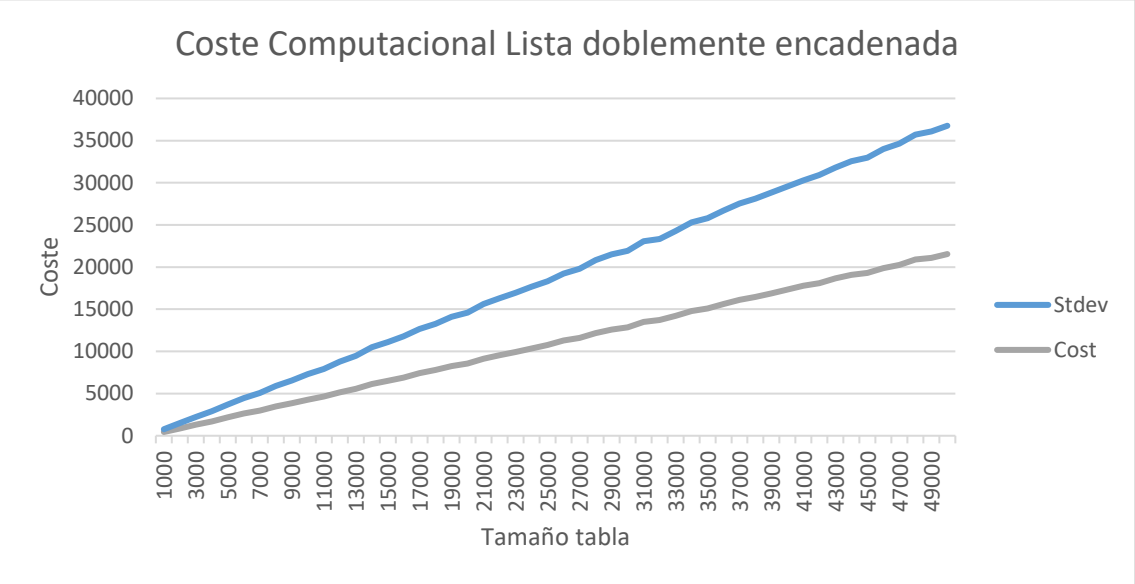
Este método es privado dado que solo se debe llamar desde al método insert y solo y únicamente cuando se sobrepase el 75% de la carga de la estructura.

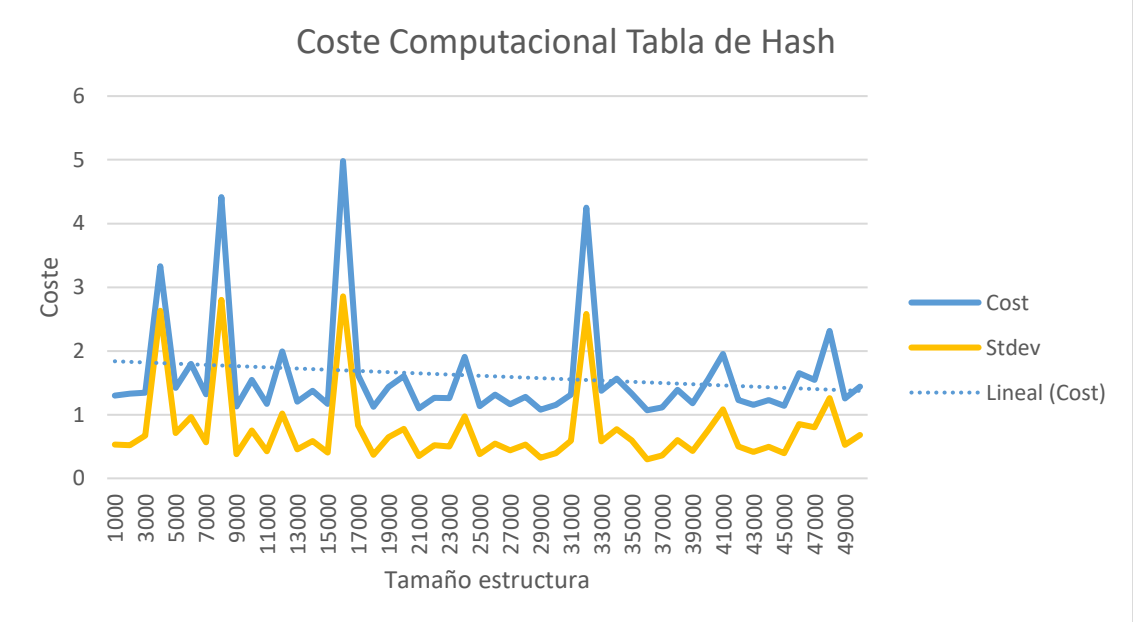
Análisis del Coste computacional

Este análisis se ha de realizar sobre ambas estructuras implementadas haciendo búsquedas de enteros.

Antes de realizar el análisis se pueden realizar ciertas “predicciones” sobre cuales van a ser los resultados obtenidos.

Si todo ha estado bien implementado, por teoría sabemos que la lista doblemente encadenada debería tener un coste de búsqueda lineal, es decir, O(n), dado que no esta ordenada y se tiene que buscar desde el inicio hasta el peor caso que seria el ultimo. Por otro lado la tabla de hash debería tener un coste constante, O(1), gracias a la función de hash que nos lleva directamente a la posición en la que se debería encontrar el elemento y teniendo únicamente en cuenta una cantidad pequeña de colisiones.





Como se ve en las gráficas, la diferencia de coste es abismal. Donde el coste de búsqueda de la lista doblemente encadenada llega hasta los 21543 elementos accedidos en comparación a 1 elemento accedido de media en la tabla de hash para el tamaño máximo de este análisis.

Se pueden sacar varias conclusiones de estos resultados, el primero y obvio es que la tabla de hash es mucho mejor para tiempos de búsqueda en comparación a la lista doblemente encadenada.

También se ha visto que la teoría no siempre es la regla en la práctica, dado que esperábamos un coste constante para la tabla de hash y en ciertas ocasiones se ha visto que el coste sobrepasaba al elemento accedido. Aunque esto último se da en pocas ocasiones y normalmente en tamaños mas pequeños, puede deberse a una mala distribución de las claves dentro de la estructura por parte de la función de hash además de la aleatoriedad de los elementos insertados en la estructura para este estudio. Aun así, se ve claramente que la tendencia de coste se queda en un rango de entre 1 o 2 de coste, nunca llegando al último.

Y por último también podemos ver claramente que la gráfica de coste de la lista doblemente encadenada representa una linealidad que esperábamos en el estudio previo, aun siendo este menos inclinado de lo esperado.

Código enganchado

SearchCost

```
package Aplicacio;

import EstructuraDeDades.Part1.DoublyLinkedList;
import EstructuraDeDades.Part2.HashTable;
import Exceptions.*;
import com.opencsv.CSVWriter;

import java.io.*;
import java.util.Random;

public class SearchCost {
   public static void main(String[] args){
      final int MAX_SIZE = 50000;

      DoublyLinkedList<Integer> dll;
      HashTable<Integer, Integer> ht;

      Random rand = new Random();

      File file1 = new File("resultdll.csv");
      File file2 = new File("resultht.csv");
      try {
         FileWriter fwdll = new FileWriter(file1);
         FileWriter fwht = new FileWriter(file2);

         CSVWriter writerdll = new CSVWriter(fwdll);
         CSVWriter writerht = new CSVWriter(fwht);

         String[] header = {"Size", "Cost","Stdev"};
```

```
writerdll.writeNext(header);
writerht.writeNext(header);

for (int i = 1000; i <= MAX_SIZE; i += 1000) {
   double dllsum = 0;
   double htsum = 0;

   // create lists of i elements
   dll = new DoublyLinkedList<>();
   ht = new HashTable<>(i);

   System.out.println("Creating lists of " + i + " size...");

   // add i random integers from 1 to i/2 to the lists
   int j = 0;
   while (j < i) {
      dll.insert(rand.nextInt(1, i / 2));
      try {
         ht.insert(rand.nextInt(1, i / 2), i);
      } catch (SizeException e) {
         //System.out.println(e.getMessage());
      }
      j++;
   }

   DoublyLinkedList<Integer> dllsearch = new DoublyLinkedList<>();
   DoublyLinkedList<Integer> htsearch = new DoublyLinkedList<>();

   System.out.println("Searching for " + i + " elements...");
```

```
            // search for i random integers
            j = 0;
            while (j < i) {

               // if it's found add it to the total sum for that i size
               try {
                  int dllres = dll.search(rand.nextInt(1, i / 2));
                  dllsearch.insert(dllres);
                  dllsum += dllres;

               } catch (SearchNotFound s) { // if it can't be found add the total cost of searching the table
                  dllsum += s.getCost();
                  //System.out.println(s.getMessage());
               }

               // if it's found
               try {
                  int htres = ht.search(rand.nextInt(1, i / 2));
                  htsearch.insert(htres);
                  htsum += htres;
               } catch (SearchNotFound s) { // if it can't be found
                  htsum += s.getCost();
                  //System.out.println(s.getMessage());
               }

               j++;
            }

            System.out.println("Saving results...");
```

```
            // saving the results
            double dllmean = dllsum / i;
            double htmean = htsum / i;

            double dllstdev = calculateSD(dllsearch, dllsum);
            double htstdev = calculateSD(htsearch, htsum);

            System.out.println("Doubly Linked List: ");
            System.out.println("Size: " + i + "\tMean: " + dllmean + "\tStandard Deviation: " + dllstdev + "\t");

            System.out.println("Hash Table: ");
            System.out.println("Size: " + i + "\tMean: " + htmean + "\tStandard Deviation: " + htstdev + "\t");

            String[] data1 = {String.valueOf(i), String.valueOf(dllmean), String.valueOf(dllstdev)};
            writerdll.writeNext(data1);

            String[] data2 = {String.valueOf(i), String.valueOf(htmean), String.valueOf(htstdev)};
            writerht.writeNext(data2);
        }
        writerdll.close();
        writerht.close();
    } catch (IOException e){
        e.printStackTrace();
    }

    System.out.println("Done!!!");
  }

  public static double calculateSD(DoublyLinkedList<Integer> list, double sum){
    double stdev = 0.0;
```

```
        int length = list.size();

        double mean = sum / length;

        for (Integer n : list){
            stdev += Math.pow(n - mean, 2);
        }

        return Math.sqrt(stdev/length);
    }
}
```

ValidationDLL

```
package Aplicacio;

import EstructuraDeDades.*;
import EstructuraDeDades.Part1.DoublyLinkedList;
import Exceptions.*;

public class ValidationDLL {
    public static void main(String[] args){
        DoublyLinkedList<Ciutada> ciutadans = new DoublyLinkedList<>();

        Ciutada c1 = new Ciutada("Ruby", "Lawrence", "19632780A");
        Ciutada c2 = new Ciutada("Elena", "Lawrence", "35970581F"); // same lastname as c1
        Ciutada c3 = new Ciutada("Jean", "Cooper", "12408979L");    // all different
        Ciutada c4 = new Ciutada("Ruby", "Campbell", "12408979L");  // same dni as c2
        Ciutada c5 = new Ciutada("Troy", "Alvarez", "89703330H");   // all different
        Ciutada c6 = new Ciutada("Luna", "Wong", "19632780A");      // same dni as c1

        ciutadans.insert(c1);
```

```
ciutadans.insert(c2);
ciutadans.insert(c3);
ciutadans.insert(c4);

try{
   ciutadans.insert(2, c5);
   ciutadans.insert(1, c6);
} catch (SizeException e){
   System.out.println(e.getMessage());
}

System.out.println("List initially...");

showList(ciutadans);

System.out.println("Size: " + ciutadans.size()); // size = 6

System.out.println("Trying to add to a position bigger than the list size...");

try{
   ciutadans.insert(8, c4);
}
catch (SizeException s){
   System.out.println(s.getMessage());
}

System.out.println("Size after failed insertion: " + ciutadans.size()); // size = 6

System.out.println("Removing first... ");

try{
```

```
            ciutadans.remove(0);
        }
        catch (NotFound n){
            System.out.println(n.getMessage());
        }

        System.out.println("List after removing first element...");

        showList(ciutadans);

        System.out.println("Size after removing: " + ciutadans.size()); // size = 5

        try {
            System.out.println("Element at position = 4: " + ciutadans.get(4)); // Ruby Campbell 12408979L
            System.out.println("Element at position = 7: " + ciutadans.get(7)); // Error
        }
        catch(NotFound e){
            System.out.println(e.getMessage());
        }

        Ciutada cout = new Ciutada("Doesn't", "Belong", "Here");
        System.out.println("Searching for citizens Elena, Ruby and other citizen...");
        try{
            System.out.println("Items checked: " + ciutadans.search(c2)); // 2
            System.out.println("Items checked: " + ciutadans.search(c4)); // 4 as Ruby has the same DNI as Jean
            System.out.println("Items checked: " + ciutadans.search(cout)); // 5 and error as it doesn't belong to the citizens list
        } catch (SearchNotFound e){
            System.out.println(e.getMessage());
        }
```

```
    }

    public static <T extends Comparable<T>> void showList(DoublyLinkedList<T> l){
        // using foreach thanks to the iterator
        for (T elem:l){
            System.out.println(elem);
        }
    }
}
```

ValidationHT

```
package Aplicacio;

import EstructuraDeDades.Ciutada;
import EstructuraDeDades.Part1.DoublyLinkedList;
import EstructuraDeDades.Part2.HashTable;
import Exceptions.NotFound;
import Exceptions.SearchNotFound;
import Exceptions.SizeException;

public class ValidationHT {
    public static void main(String[] args){
        HashTable<String, Ciutada> ciutadans = new HashTable<>(5);

        Ciutada c1 = new Ciutada("Kratos", "Lawrence", "19632780A");
        Ciutada c2 = new Ciutada("Elena", "Lawrence", "35970581F");
        Ciutada c3 = new Ciutada("Jean", "Cooper", "12408979L");
        Ciutada c4 = new Ciutada("Ruby", "Campbell", "54368005F");
        Ciutada c5 = new Ciutada("Troy", "Alvarez", "89703330H");
        Ciutada c6 = new Ciutada("Luna", "Wong", "29309153T");
```

```
try {
   ciutadans.insert(c1.getDNI(), c1);
   ciutadans.insert(c2.getDNI(), c2);
   ciutadans.insert(c3.getDNI(), c3);
   ciutadans.insert(c4.getDNI(), c4);
   ciutadans.insert(c5.getDNI(), c5);
   ciutadans.insert(c6.getDNI(), c6);
}
catch(SizeException e){
   System.out.println(e.getMessage());
}
System.out.println("\nList initially...");

ciutadans.showTable();

System.out.println("\n-----------------------------------------\n");

System.out.println("Size: " + ciutadans.size()); // size = 6

System.out.println("\n-----------------------------------------\n");

System.out.println("Trying to add a citizen with a same dni...");

Ciutada c7 = new Ciutada("John", "Cena", "19632780A");
try{
   ciutadans.insert(c7.getDNI(), c7);
}
catch (SizeException s){
   System.out.println(s.getMessage());
}
```

```
ciutadans.showTable();

System.out.println("\n----------------------------------------\n");

System.out.println("Size: " + ciutadans.size()); // size = 6

System.out.println("\n----------------------------------------\n");

System.out.println("Getting key = 29309153T");
try {
   System.out.println(ciutadans.get("29309153T")); // Should print citizen Luna
}
catch (NotFound e){
   System.out.println(e.getMessage());
}

System.out.println("\n----------------------------------------\n");

System.out.println("Getting unknown key = 12345678T");
try {
   System.out.println(ciutadans.get("12345678T")); // Should give an error
}
catch (NotFound e){
   System.out.println(e.getMessage());
}

System.out.println("\n----------------------------------------\n");

System.out.println("Trying to remove Ruby...");
try{
   ciutadans.remove("54368005F");
```

```
        }
        catch (NotFound e){
            System.out.println(e.getMessage());
        }

        ciutadans.showTable();

        System.out.println("\n----------------------------------------\n");

        System.out.println("List size: " + ciutadans.size()); // size = 5

        System.out.println("\n----------------------------------------\n");

        System.out.println("Load Factor: " + ciutadans.getLoadFactor()); // 5 / 100 = 0.05

        System.out.println("\n----------------------------------------\n");

        try {
            System.out.println("Cost of searching for Luna's DNI: " +
ciutadans.search("29309153T")); // 2 at most
        }
        catch (SearchNotFound e){
            System.out.println(e.getMessage());
        }

        System.out.println("\n----------------------------------------\n");

        System.out.println("Cost of searching for unknown DNI: ");
        try {
            System.out.print(ciutadans.search("12345678A")); // Error message
        }
```

```
        catch (SearchNotFound e){
            System.out.println(e.getMessage());
        }

        System.out.println("\n----------------------------------------\n");

        System.out.println("All values: ");
        DoublyLinkedList<Ciutada> l1 = ciutadans.getValues();
        for (Ciutada c:l1){
            System.out.println(c);
        }

        System.out.println("\n----------------------------------------\n");

        System.out.println("All keys: ");
        DoublyLinkedList<String> l2 = ciutadans.getKeys();
        for (String str:l2){
            System.out.println(str);
        }
    }
}
```

DLL

```
package EstructuraDeDades.Part1;

import Exceptions.*;

public interface DLL<T extends Comparable<T>> {

    void create();
```

```
    void insert(T data);

    void insert(int pos, T data) throws SizeException;

    T get(int pos) throws NotFound;

    int size();

    void remove(int pos) throws NotFound;

    int search(T data) throws SearchNotFound;
}
```

DoublyLinkedList

```
package EstructuraDeDades.Part1;

import Exceptions.*;

import java.util.Iterator;

public class DoublyLinkedList<T extends Comparable<T>> implements DLL<T>, Iterable<T> {
    private Node<T> first, last;

    /**
     * Constructor de classe
     */
    public DoublyLinkedList(){
        create();
    }

    /**
```

```
 * Crear: constructor per inicialitzar la llista
 *
 */
public void create() {
   first=last=null;
}

/**
 * Inserir: insereix un element al final de la llista
 *
 * @param data T que es vol inserir a la llista
 */
public void insert(T data) {
   Node<T> node = new Node<>(data);
   // look if the list is empty
   if (last == null) { // same as comparing if first == null
      first = node;
      last = node;
      node.prev = null;
      node.next = null;
   }
   else { // if list is not empty, add at the end of the list
      node.prev = last;
      node.next = null;
      last.next = node;
      last = node;
   }
}

/**
 * Inserir: insereix un element a la llista en la posició indicada
```

```
 *
 * @param pos enter on es es vol inserir l'element
 * @param data T que es vol inserir a la llista
 * @throws SizeException excepció en cas que no es pugui fer l'operació
 */
public void insert(int pos, T data) throws SizeException{
    Node<T> node = first;
    Node<T> newNode = new Node<>(data);

    if (pos > this.size()) {
        throw new SizeException();
    }
    else {
        int n = 0;
        while (n < pos) {
            node = node.next;
            n++;
        }

        newNode.next = node;
        if (node.prev == null) {
            newNode.prev = null;
            first = newNode;
        } else {
            newNode.prev = node.prev;
            node.prev.next = newNode;
        }
        node.prev = newNode;
    }
}
```

```
/**
 * Obtenir: retorna l'element que hi ha en una determinada posició
 *
 * @param pos posició on es troba l'element
 * @return l'element que hi ha en una determinada posició
 * @throws NotFound excepció en cas que no es pugui obtenir
 */
public T get(int pos) throws NotFound {
   Node<T> node = first;
   int count = 0;
   while((count < pos) && (node != null)){
      count++;
      node = node.next;
   }

   if ((count == pos) && (node != null)){
      return node.data;
   }else throw new NotFound();
}

/**
 * Nombre d'elements que conté la llista
 *
 * @return el nombre d'elements que conté la llista en aquest moment
 */
public int size() {
   Node<T> node = first;
   int size = 0;
   while(node != null){
      size++;
      node = node.next;
```

```
    }
    return size;
  }

  /**
   * Esborrar: esborra un element de la llista en una posició determinada
   *
   * @param pos integer on es troba l'element a esborrar
   * @throws NotFound llença l'excepció en cas que no es pugui eliminar
   */
  public void remove(int pos) throws NotFound{
    Node<T> target = first;
    int n = 0;
    while(n < pos){ // get target node in position == pos
      n++;
      target = target.next;
    }

    if ((n == pos) && (target != null)){
      if (target.prev == null)    // if target position is the first node then we don't
        first = target.next;    // need to reassign prev node's next pointer
      else
        target.prev.next = target.next; // any other position will reassign prev node's next
pointer

      if (target.next == null)    // if target position is the last node then we don't need to
        last = target.prev;     // reassign next node's prev pointer
      else
        target.next.prev = target.prev; // any other position will reassign next node's prev
pointer
    } else throw new NotFound();
  }
```

```
  /**
   * Buscar: comprova si un element està a la llista
   *
   * @param data T que es vol buscar
   * @return cost de l'operació, nombre d'elements que s'han accedit per tal de comprovar si l'element existeix o no.
   * @throws SearchNotFound excepció en cas que l'element no s'hagi trobat, contindrà informació del nombre d'elements que s'han accedit.
   */
  public int search(T data) throws SearchNotFound {
    Node<T> node = first;
    int cost = 1;

    T val = node.data;

    while((val.compareTo(data) != 0) && (node != null)) {
      node = node.next;
      if(node != null)
        val = node.data;
        cost++;
    }

    if(node == null)
      throw new SearchNotFound(cost - 1);

    if (val.compareTo(data) == 0){
      return cost;
    } else throw new SearchNotFound(cost - 1);
  }

  @Override
```

```
    public Iterator<T> iterator(){
        return new TIterator<>(this);
    }
}
```

Node

```
package EstructuraDeDades.Part1;

public class Node<T extends Comparable<T>> {
    Node<T> next, prev;
    T data;

    public Node (T data){
        this.data = data;
    }

    @Override
    public String toString() {return "Node [data= " + data + "]";}
}
```

TIterator

```
package EstructuraDeDades.Part1;

import Exceptions.NotFound;

import java.util.Iterator;

public class TIterator<T extends Comparable<T>> implements Iterator<T> {
    private DoublyLinkedList<T> list;
    private int posIterator;
```

```
    public TIterator(DoublyLinkedList<T> list){
        this.list = list;
        posIterator = 0;
    }

    @Override
    public boolean hasNext() {
        return posIterator < list.size();
    }

    @Override
    public T next() {
        T aux = null;
        try {
            aux = list.get(posIterator);
        } catch (NotFound e) {
            e.printStackTrace();
        }
        posIterator++;
        return aux;
    }
}
```

HashNode

```
package EstructuraDeDades.Part2;

public class HashNode<K extends Comparable<K>, T extends Comparable<T>> {
    K key;
    T value;
    HashNode<K, T> next;
```

```
    public HashNode(K key, T value){
        this.key = key;
        this.value = value;
    }

    @Override
    public String toString(){
        return "Key [" + key + "]\tValue [" + value + "]";
    }
}
```

HashTable

```
package EstructuraDeDades.Part2;

import EstructuraDeDades.Part1.DoublyLinkedList;
import Exceptions.*;

import java.util.Arrays;

public class HashTable<K extends Comparable<K>, T extends Comparable<T>> implements HashTableContract<K, T>{
    private HashNode<K, T>[] hashTable;
    private static final float MAX_LOAD = 0.75F;

    /**
     * Constructor de classe
     *
     * @param dim dimensió de la taula de hashing
     */
    public HashTable(int dim){
        create(dim);
```

```
}

/**
 * Crear: Constructor per inicialitzar la taula
 */
@SuppressWarnings("unchecked")
public void create(int dim) {
   hashTable = new HashNode[dim];
   Arrays.fill(hashTable, null);
}

/**
 * Funció de hashing (String folding)
 *
 * @param key K per obtenir un índex de la taula de hashing
 * @return índex de la taula de hashing
 */
private int hashFunc(int size, K key){
   long sum = 0;
   long mul = 1;

   String keyAsString = key.toString();

   // fold a string, summing 4 bytes at a time
   for(int i = 0; i < keyAsString.length(); i++){
      mul = (i % 4 == 0) ? 1 : mul * 256;
      sum += keyAsString.charAt(i) * mul;
   }
   return (int) (Math.abs(sum) % size);
}
```

```
/**
 * Inserir: insereix un element a la taula de Hash
 * si l'element ja existeix, actualitza el seu valor
 *
 * @param key clau que indexa l'informació
 * @param data informació a emmagatzemar
 * @throws SizeException excepció en cas que no es pugui inserir
 */
public void insert(K key, T data) throws SizeException {
   float lf = getLoadFactor();

   if (lf > MAX_LOAD) {
      /*throw new SizeException();*/
      resize();
   }

   // call to these functions after possible resizing so that the
   // hashtable length and index calculations are correct
   int index = hashFunc(hashTable.length, key);
   HashNode<K, T> node = new HashNode<>(key, data);

   insertion(hashTable, index, node);
}

/**
 * Obtenir: retorna l'element que té la clau K
 *
 * @param key clau que indexa l'element que es vol obtenir
 * @return l'element que té la clau K
 * @throws NotFound excepció en cas que no es pugui obtenir
 */
```

```
    public T get(K key) throws NotFound{
        int index = hashFunc(hashTable.length, key);

        HashNode<K, T> node = hashTable[index];

        // traverse list as long as it doesn't reach the end, or it finds
        // the node that contains the same key as the argument
        while((node != null) && (node.key.compareTo(key) != 0)){
            node = node.next;
        }

        // while statement might have reached the end of the collision
        // without finding the key, or the index of the hash table was already null
        if(node == null) {
            throw new NotFound();
        }

        return node.value;
    }

    /**
     * Buscar: comprova si un element està a la taula
     *
     * @param key clau que indexa l'element a buscar
     * @return Cost de l'operació. Nombre d'elements que s'hagin accedit
     * @throws SearchNotFound excepció en cas que l'element no s'hagi trobat, conté
     * informació del nombre d'elements que s'han accedit per comprovar si l'element existeix o
no
     */
    public int search(K key) throws SearchNotFound {
        int index = hashFunc(hashTable.length, key);
```

```
    HashNode<K, T> node = hashTable[index];

    int cost = 1; // just by visiting first node we'll count as one access

    while(node != null) {
      // add until to the total cost if it's not the key we are searching for
      if (node.key.compareTo(key) != 0) {
        cost++;
      }
      else {
        return cost;
      }
      node = node.next;
    }

    throw new SearchNotFound(cost); // reaching here means it didn't find the key and reached the end of the list
  }

  /**
   * Mida: nombre d'elements que conté la taula
   *
   * @return el nombre d'elements que conté la taula en aquest moment
   */
  public int size() {
    int size = 0;
    // each existing node different from null will count as an element in the hash table
    // so traversing the table and each linked list will be necessary
    for (HashNode<K, T> ktHashNode : hashTable) {
      if (ktHashNode != null) {
        size++;
```

```
            HashNode<K, T> node = ktHashNode;
            while (node.next != null) {
                node = node.next;
                size++;
            }
        }
    }
    return size;
}

/**
 * Esborrar: esborra un element de la taula
 *
 * @param key clau que indexa l'element a esborrar
 * @throws NotFound excepció en cas que l'element no s'hagi trobat
 */
public void remove(K key) throws NotFound {
    int index = hashFunc(hashTable.length, key); // get index to remove

    HashNode<K, T> node = hashTable[index];

    if (node == null) // if position at index is null, there's no node to remove
        throw new NotFound();

    // if node at position index happens to have the key searched for
    // if it was the only collision, it will be equal to null
    // if it wasn't, it will be equal to the next node reference
    if (node.key.compareTo(key) == 0)
        hashTable[index] = node.next;
    else
```

```
        // traverse list in position until last element of list or the key is found
        while((node.next != null) && (node.next.key.compareTo(key) != 0)){
            node = node.next;
        }
        // if reached here it can only be either end of list or key found
        // so throw exception if end of list reached
        if(node.next == null)
            throw new NotFound();
        // reassign "next" reference to the to be removed node's next
        if(node.next.key.compareTo(key) == 0) {
            node.next = node.next.next;
        }
    }

    /**
     * ObtenirValors: obtenir una llista amb tots els valors de la taula
     *
     * @return una llista amb tots els valors de la taula
     */
    public DoublyLinkedList<T> getValues() {
        DoublyLinkedList<T> values = new DoublyLinkedList<>();

        // for every position in the hash table, get its initial node
        // and add it to the list if it's not null
        for (HashNode<K, T> ktHashNode : hashTable) {
            HashNode<K, T> node = ktHashNode;

            while (node != null) {
                values.insert(node.value);
                node = node.next;
            }
```

```
    }
    return values;
}

/**
 * ObtenirClaus: obtenir una llista amb totes les claus de la taula
 *
 * @return una llista amb totes les claus de la taula
 */
public DoublyLinkedList<K> getKeys() {
    DoublyLinkedList<K> keys = new DoublyLinkedList<>();

    // for every position in the hash table, get its initial node
    // and add it to the list if it's not null
    for (HashNode<K, T> ktHashNode : hashTable) {
        HashNode<K, T> node = ktHashNode;

        while (node != null) {
            keys.insert(node.key);
            node = node.next;
        }
    }
    return keys;
}

/**
 * ObtenirFactorDeCàrrega: factor de càrrega actual
 *
 * @return el factor de càrrega actual
 */
public float getLoadFactor() {
```

```
    return ((float) size()) / hashTable.length;
  }


  private void insertion(HashNode<K, T>[] table, int index, HashNode<K, T> node){
    if (table[index] == null) { // not a collision
      table[index] = node;
    } else {
      HashNode<K, T> temp = table[index];

      // comparing temp.next to null instead of temp with null to get
      // the last node so that we can assign its ".next" to the new node if needed
      while ((temp.next != null) && (temp.key.compareTo(node.key) != 0)) {
        temp = temp.next;
      }
      // in case the key already exists, just update its value
      if (temp.key.compareTo(node.key) == 0) {
        temp.value = node.value;
      } else {
        temp.next = node;
        node.next = null;
      }
    }
  }

  @SuppressWarnings("unchecked")
  private void resize(){
    HashNode<K, T>[] resized = new HashNode[hashTable.length * 2]; // double the original
size
    Arrays.fill(resized, null);
```

```
        for (HashNode<K, T> node : hashTable) {
            while (node != null) {
                // for every node existing in the original table get a new index
                // relative to the resized table and insert it
                HashNode<K, T> temp = new HashNode<>(node.key, node.value);
                int index = hashFunc(resized.length, temp.key);

                insertion(resized, index, temp);

                node = node.next;
            }
        }

        hashTable = resized;
    }

    public void showTable(){
        for (HashNode<K, T> ktHashNode : hashTable) {
            if (ktHashNode != null) {
                HashNode<K, T> node = ktHashNode;
                while (node != null) {
                    System.out.println(node);
                    node = node.next;
                }
            }
        }
    }
}
```

HashTableContract

```
package EstructuraDeDades.Part2;

import EstructuraDeDades.Part1.DoublyLinkedList;
import Exceptions.*;

public interface HashTableContract<K extends Comparable<K>, T extends Comparable<T>> {

    void create(int dim);

    void insert(K key, T data) throws SizeException;

    T get(K key) throws NotFound;

    int search(K key) throws SearchNotFound;

    int size();

    void remove(K key) throws NotFound;

    DoublyLinkedList<T> getValues();

    DoublyLinkedList<K> getKeys();

    float getLoadFactor();
}
```

Ciutada

```
package EstructuraDeDades;


public class Ciutada implements Comparable<Ciutada> {
   private final String Nom;
   private final String Cognom;
   private final String DNI;

   public Ciutada(String n, String c, String d){
      Nom = n;
      Cognom = c;
      DNI = d;
   }

   public String getNom() {
      return Nom;
   }

   public String getCognom() {
      return Cognom;
   }

   public String getDNI() {
      return DNI;
   }

   @Override
   public int compareTo(Ciutada o) {
      return getDNI().compareTo(o.getDNI());
   }
```

```
    @Override
    public String toString(){return "Nom: " + Nom + "\tCognom: " + Cognom + "\tDNI: " + DNI;}
}
```

NotFound

```
package Exceptions;

public class NotFound extends Exception{
    private static final long serialVersionUID=1L;

    public NotFound() { super("Error : No s'ha pogut trobar l'element");}
}
```

SearchNotFound

```
package Exceptions;

public class SearchNotFound extends Exception{
    private static final long serialVersionUID=1L;
    private int cost = 0;

    public SearchNotFound(int n) {
        super("Error : No s'ha trobat l'element\nS'han accedit a " + n + " elements");
        cost = n;
    }

    public int getCost() {
        return cost;
    }
}
```

SizeException

```
package Exceptions;


public class SizeException extends Exception{
   private static final long serialVersionUID=1L;


   public SizeException(){ super("Error : No es pot afegir l'element");}
}
```