ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

[DΣIM]

Estructura de Dades

Pràctica 1: Doubly-linked list i Taula de Hash

curs 2022-23

Estudiant: Matías Larrosa

Professor/a: Marc Ruiz

Data de lliurament: 21/04/2021

Decisions de disseny

Doubly-linked List (DLL)

La doble llista enllaçada que he implementat no fa servir de cap classe extra de tipus node, en canvi, ella mateixa es el seu propi "node". Penso que així es mes simple.

```
public class DLL<T extends Comparable<T>>

    private DLL<T> fwd;
    private DLL<T> bkw;
    private T data;
```

crear()

Crear només inizialitza tots els valors a null. Tots els constructors criden a crear() per tal de no haber de cridar aquest mètode redundantment i, si al constructor se li passa alguna dada aplicarà els canvis necessaris.

```
public DLL(T data, DLL<T> bkw){
    crear();
    this.data = data;
    this.bkw = bkw;
}
```

```
public void crear() {
    fwd = null;
    bkw = null;
    data = null;
}
```

inserir(T)

El mètode inserir afegeix un element al final de la llista, l'única cosa que cal destacar d'aquest mètode es el canvi que he agut d'aplicar a la implementació. En un principi, utilitzava un mètode recursiu en el qual si poi.getFwd() != null cridava a inserir de fwd (fwd.inserir(T)). Aquest codi funciona bé però el problema es el desbordament de la pila (stack overflow) que succeeix quan hi han molts elements per recorre. Això, hem va passar al fer el anàlisi de cost computacional al inserir ~40_000 elements. El mètode ara es iteratiu.

```
if(data != null){
    if (this.data != null) {
        if(fwd != null)
            fwd.inserir(data);
        else
            fwd = new DLL<>(data, this);
    } else
        this.data = data;

}
```

```
if (this.data != null) {
    DLL<T> poi = this;
    while(poi.getFwd() != null){
        poi = poi.getFwd();
    }
    poi.setFwd(new DLL<>(data, poi));
} else
    this.data = data;
```

inserir(int, T)

Aquest mètode ja es una mica més complicat, comencem decidint quin rang de valors admetrem. En el meu cas [1, longitud() + 1], qualsevol valor que estigui fora d'aquest rang provocarà una excepció en temps d'execució (operacioImpossible(int)). La posició 1 correspon a la primera de la llista (head) i la posició corresponent a la longitud() + 1 de la llista correspon a la última (tail). A partir d'aquí ja es tractar els diferents casos.

```
public void inserir(int posicio, T data) throws operacioImpossible {
    if(posicio > longitud() + 1 || posicio < 1) throw new operacioImpossible(posicio);
    DLL<T> poi = this;

    for(int i = 1; i < posicio;i++){
        if(poi.getFwd() != null)
            poi = poi.fwd;
    }

    if(poi.getBkw() == null) {
        DLL<T> newNode = new DLL<>(this.data,  bkw: this);
        newNode.setFwd(fwd);
        fwd = newNode;
        this.data = data;
    } else {
        /* Al final */
        if(posicio == longitud() + 1){
            DLL<T> newNode = new DLL<>(data, poi);
            newNode.setFwd(poi.getFwd());
            if(poi.getFwd() != null)
                poi.getFwd().setBkw(newNode);
            poi.setFwd(newNode);
        } else {
            DLL<T> newNode = new DLL<>(data, poi.getBkw());
            newNode.setFwd(poi);
            poi.getBkw().setFwd(newNode);
            poi.setBkw(newNode);
        }
    }
}
```

obtenir(int)

Obtenir retorna l'element a la posició demanada. Si la posició esta fora del rang, en aquest cas [1, longitud()], es llençarà una excepció en temps de execució. Per recorre la llista i trobar la posició demanada la recorro utilitzant un for-each.

```
public T obtenir(int posicio) throws operacioImpossible {
    if(posicio > longitud() || posicio < 1) throw new operacioImpossible(posicio);
    int i = 0;
    for(T data : this){
        i++;
        if(i == posicio) return data;
    }
    throw new operacioImpossible(posicio);
}
```

`longitud()`

Retorna la longitud del la llista, utilitzant un bucle for-each per recorre-la. Es podria heber obtat per guardar la longitud de la llista i actualitzarla segons s'afegeixen o s'eliminen elements.

```
public int longitud() {
    if(data == null)
        return 0;
    int i = 0;
    for(T data : this)
        i++;
    return i;
}
```

`esborrar(int)`

Esborra la posició demanada sempre que estigui dins del rang correcte [1, longitud()]. Per esborrar s'ha de tenir en compte quin cas s'està tractant; si es tracta de l'últim element (fwd == null), si es l'únic element, si es el primer element o si es un del mig.

```
public void esborrar(int posicio) throws operacioImpossible {
    if(posicio > longitud() || posicio < 1) throw new operacioImpossible(posicio);

    DLL<T> poi = this;
    for(int i = 1; i < posicio;i++){
        poi = poi.fwd;
    }
    if(poi.fwd == null){    /* Ultim */
        if(poi.getBkw() != null){
            poi.getBkw().setFwd(null);
            poi.setBkw(null);
        }
        poi.setData(null); /* Unic */
    } else{
 } else{
     if(poi.getBkw() == null){   /* Primer */
         T data = poi.getFwd().getData();
         setData(data);
         if(poi.getFwd().getFwd() != null)
             poi.getFwd().getFwd().setBkw(this);
         DLL<T> tmp = poi.getFwd();
         setFwd(poi.getFwd().getFwd());
         poi = tmp;
     }
     else{ /* Mig */
         poi.getBkw().setFwd(poi.getFwd());
         poi.getFwd().setBkw(poi.getBkw());
     }
     poi.setData(null);
     poi.setFwd(null);
     poi.setBkw(null);
 }
```

`buscar(T)`

Retornarà el nombre d'elements que s'han agut de comprovar per trobar o no trobar l'element. Recorro la llista amb un bucle for-each.

```java
@Override
public int buscar(T data) throws elementNoExisteix {
    int elem = 0;
    for(T d : this){
        elem++;
        if(d.compareTo(data) == 0){
            return elem;
        }
    }
    throw new elementNoExisteix(elem, longitud());
}
```

DLLIterator

Aquesta es la classe que es fa servir per poder iterar sobre la llista. Es guarda una referencia de la llista sencera que s'inicialitza al constructor se si hi ha un següent element si la referencia no està apuntant a null i per apuntar al següent només cal fer dll = dll.getFwd(). Aquest és un dels avantatges de no tenir nodes.

```java
private DLL<T> dll;

public DLLiterator(DLL<T> dll) { this.dll = dll; }

@Override
public boolean hasNext() { return dll != null; }

@Override
public T next() {
    T data = dll.getData();
    dll = dll.getFwd();
    return data;
}
```

Joc de proves DLL

Per fer el joc de proves he fet servir la llibreria de JUnit per fer més simple el testeig i la comprovació. He deixat un main preparat per si es vol fer comprovacions extres.

Taula de hash (TaulaHash)

La taula de hash si que utilitza un node extern a la seva estructura i emmagatzema aquestos nodes en un ArrayList inicialitzat a null en totes les seves posicions (tamany = capacity). També té un valor constant anomenat límit que correspon al valor de tall per saber si s'ha de redimensionar l'array o no. I el booleà repetits existeix per saber si es permeteixen repetits o no (a la especificació posava que si hi ha una clau repetida s'actualitza el seu valor però això crea un problema a l'hora de fer el càlcul del cost computacional).

```
public class TaulaHash<K extends Comparable<K>

    ArrayList<NodeTaulaHash<K, T>> taula;
    private int capacity;
    private static final float limit = 0.75f;
    private boolean repetits;
```

```
public K clau;
public T valor;
public NodeTaulaHash<K, T> seg;

public NodeTaulaHash(K key, T data) {
    seg = null;
    clau = key;
    valor = data;
}
```

+ getters & setters

Hasher

Hasher es la classe (amb mètode estàtics) que s'encarrega de transformar cualsevol input en un valor de hash. Per als enters retorna l'enter directament que seria lo mateix a dir hash(k) = k si k es un integer. Per a les cadenes de caràcters aplica una XOR amb el valor numèric de cada caràcter de forma alternativa (mètode de les permutacions) i per els objectes crida al toString() de l'objecte i retorna el hash de l'String.

Un dels problemes d'aquest mètode es la repetició de secuencies, al estar aplicant una XOR de cada caràcter si només hi ha un 1 la XOR de 0 i cualsevol altre cosa es aquesta cosa. Per tant, hash("a") = 'a' (97) si hi ha dos "a" hash("aa") = 0, ja que, la XOR de dos coses iguals es 0. Però en el cas de que n'hi hagin 3 "a" o cualsevol nombre imparell la hash("aaa") = "a" (97) = hash("a").

Encara així, he triat aquest mètode dels oferits a la teoria ja que hem donava els millors resultats per a claus com Patata | patata. Però no, te en compte claus com Matias Larrosa | Larrosa Matias.

```java
public static int getHash(String toHash){
    int result = 0;
    //toHash = toHash.replaceAll(" ", "").trim();
    char[] chars = new char[toHash.length()];
    toHash.getChars( srcBegin: 0, toHash.length(), chars, dstBegin: 0);

    for(int i = 0;i < toHash.length(); i++){
        result = result^(int) chars[i];
    }
    return result;
}
```

```java
public static int getHash(Object toHash){
    if (toHash instanceof Integer)
        return getHash((Integer) toHash);
    return getHash(toHash.toString());
}
```

```java
public static int getHash(int toHash) { return toHash; }
```

crear()

Crear inizialitza l'array a null amb una capacitat inicial de 10 i no admet repetits. El constructor base crida a crear i, en canvi, el complexe que modifica la capacitat inicial i si es permeten repetits no el crida.

```java
@Override
public void crear() {
    capacity = 10;
    taula = new ArrayList<>(capacity);
    for(int i = 0; i < capacity; i++){
        taula.add(null);
    }
    repetits = false;
}
```

```java
public TaulaHash(int icapacity, boolean permetreRepetits){
    capacity = icapacity;
    taula = new ArrayList<>(capacity);
    for(int i = 0; i < capacity; i++){
        taula.add(null);
    }
    repetits = permetreRepetits;
}
```

inserir(K, T)

Inserir demana una clau i l'objecte al cual li pertany aquesta clau. Doncs l'index corresponent s'obté del hash % capacitat de la taula. Si aquesta no esta utilitzada s'assigna el valor de la posició a la data (T) i, si ho està significa que s'ha donat una colisió i per tant, cal afegir-lo al final si no existeix la mateixa clau o si es permeteixen repetits o, sino si existeix la clau es modificaran les dades.
El metode només llençara una excepció si les dades eren les mateixes a les d'entrada i no s'acceptaven repetits.
Per acabar, si el factor de carrega supera el limit caldrà redimensionar la taula.

```java
public void inserir(K key, T data) throws noInsercio {
    int index = Hasher.getHash(key) % capacity;
    if(taula.get(index) != null){ /* Colisió */
        NodeTaulaHash<K, T> node = taula.get(index);
        while(node.seg != null){
            if(repetits || key.compareTo(node.getClau()) != 0)
                node = node.seg;
            else break;
        }
        if(repetits || key.compareTo(node.getClau()) != 0){
            NodeTaulaHash<K, T> nnode = new NodeTaulaHash<>(key, data);
            node.setSeg(nnode);
        }else {
            if(node.getValor().equals(data))
                throw new noInsercio(index);
            node.setValor(data);
        }
```

```java
} else { /* No colisió */
    NodeTaulaHash<K, T> nnode = new NodeTaulaHash<>(key, data);
    taula.set(index, nnode);
}
if(obtenirFactorDeCarrega() > limit){
    redimensionar();
}
```

`redimensionar()`

Redimensionar és un metode privat que redimensiona l'array amb una capacitat de 1.5 vegades més gran. Per fer-ho cal obtenir tots els elements i afegir-los a les posibles noves posicions.

```
private void redimensionar() {
    capacity *= 1.5;
    ArrayList<NodeTaulaHash<K,T>> nouarray = new ArrayList<>(capacity);
    for(int i = 0; i < capacity; i++){
        nouarray.add(null);
    }
    for(NodeTaulaHash<K, T> elem : taula){
            while(elem != null){
                int index = Hasher.getHash(elem.getClau()) % capacity;
                NodeTaulaHash<K, T> nnode = new NodeTaulaHash<>(elem.getClau(), elem.getValor());
                NodeTaulaHash<K, T> node = nouarray.get(index);
                if(node == null){
                    nouarray.set(index, nnode);
                } else {
                    while(node.getSeg() != null){
                        node = node.getSeg();
                    }
                    node.setSeg(nnode);
                }
                elem = elem.getSeg();
            }
    }
    taula = nouarray;
}
```

`obtenir(K)`

Obtenir retorna el valor de una clau determina només si existeix la seva posició del hash i si aquest posició te la mateixa clau passada.

```
public T obtenir(K key) throws noObtenir {
    int index = Hasher.getHash(key) % capacity;
    if(taula.get(index) == null) /* No existeix la posició */
        throw new noObtenir(key);
    NodeTaulaHash<K, T> node = taula.get(index);
    while(node != null && node.getClau().compareTo(key) != 0){
        node = node.getSeg();
    }
    /* La clau no es correspon amb qualsevol guardada */
    if(node == null) throw new noObtenir(key);
    return node.getValor();
}
```

esborrar(K)

Esborrar elimina un valor que tingui aquesta clau. Cal tenir en compte en quina posició i situació es troba aquest valor (col·lisió no col·lisió, inici, final o mig de la col·lisió). Aquest metode llençarà una excepció si la clau no es troba a la taula.

```
public void esborrar(K key) throws elementNoExisteix {
    int index = Hasher.getHash(key) % capacity;
    if(taula.get(index) == null)
        throw new elementNoExisteix( e: 0, mida());
    NodeTaulaHash<K, T> node = taula.get(index);
    if(node.seg == null){ /* No col·lisió */
        if(key.compareTo(node.getClau()) != 0) throw new elementNoExisteix( e: 1, mida());
        taula.set(index, null);
    } else { /* Col·lisió */
        NodeTaulaHash<K, T> ant = taula.get(index);
        int i = 0;
        if(node.getClau().compareTo(key) != 0){
            node = node.getSeg();
            while(node != null && node.getClau().compareTo(key) != 0){
                node = node.getSeg();
                ant = ant.getSeg();
                i++;
            }
            i++;
        }
        if(node == null)
            throw new elementNoExisteix( e: 1 + i, mida());
        if(ant == node){
            taula.set(index, node.getSeg());
        } else {
            ant.setSeg(node.getSeg());
        }
    }
```

TaulaHashIterator

Aquesta es la classe que es fa servir per iterar sobre la taula de hash. Emmagatzema una referencia a la taula de hash, un node per recorre en cas de que n'hi hagin col·lisions i un índex per saber en quina posició del arrayList estic. Se que hi han més elements si l'índex es més petit que la capacitat de la taula de hash. I per retornar el següent retorno el node actual i l'avanço al següent, si hi ha alguna col·lisió incremento l'índex fins a trobar un altre element o fins a acabar. L'índex l'inicialitzo a la primera posició on hi hagi un valor.

```java
private TaulaHash<K, T> th;
private NodeTaulaHash<K, T> nth;
private int index;

public TaulaHashIterator(TaulaHash<K, T> th){
    this.th = th;
    index = 0;
    for(NodeTaulaHash<K, T> node : th.getAl()){
        if(node == null){
            index++;
        } else break;
    }
    this.nth = th.getAl().get(index);
}
```

```java
@Override
public boolean hasNext() {
    return index < th.getCapacity();
}

@Override
public T next() {
    ArrayList<NodeTaulaHash<K, T>> al = th.getAl();
    T data = nth.getValor();
    nth = nth.getSeg();
    if(nth == null){
        for(index =  index + 1; index < th.getCapacity(); index++){
            if(al.get(index) != null){
                nth = al.get(index);
                break;
            }
        }
    }
    return data;
}
```

Joc de proves Taula de Hash

Per fer el joc de proves he fet servir la llibreria de JUnit per fer més simple el testeig i la comprovació. He deixat un main preparat per si es vol fer comprovacions extres.

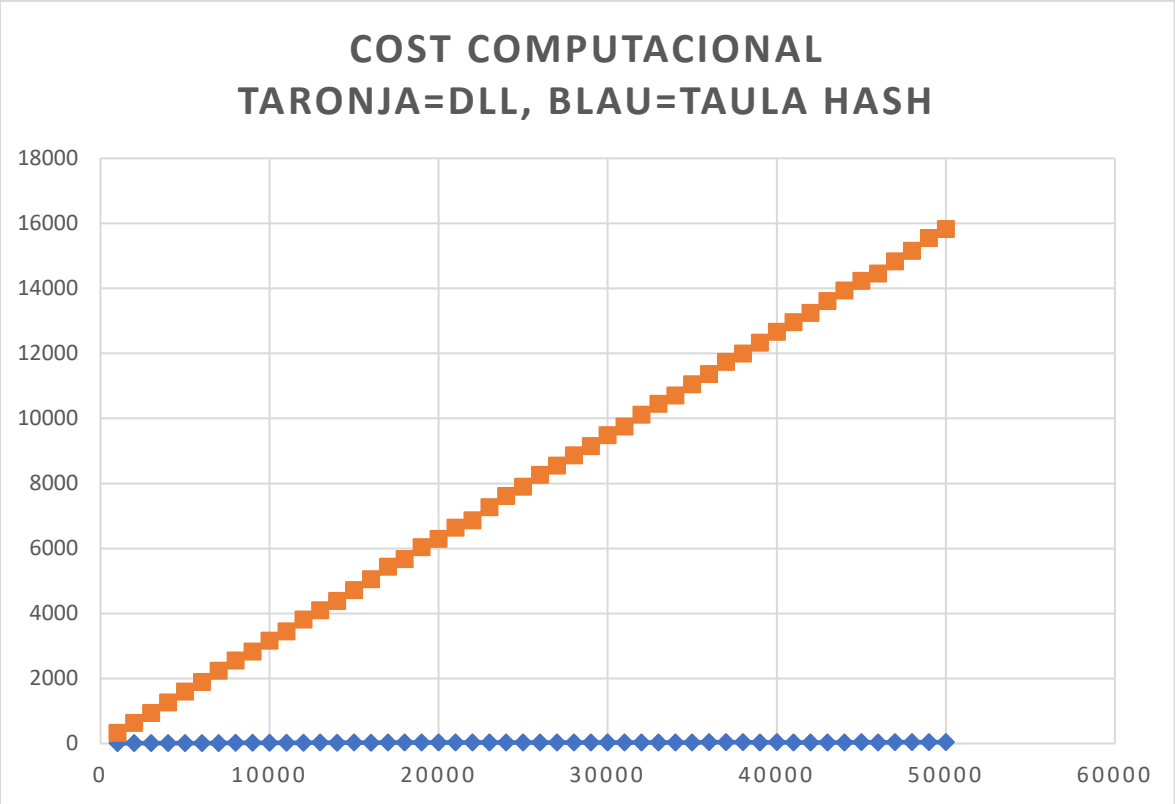
Cost Computacional

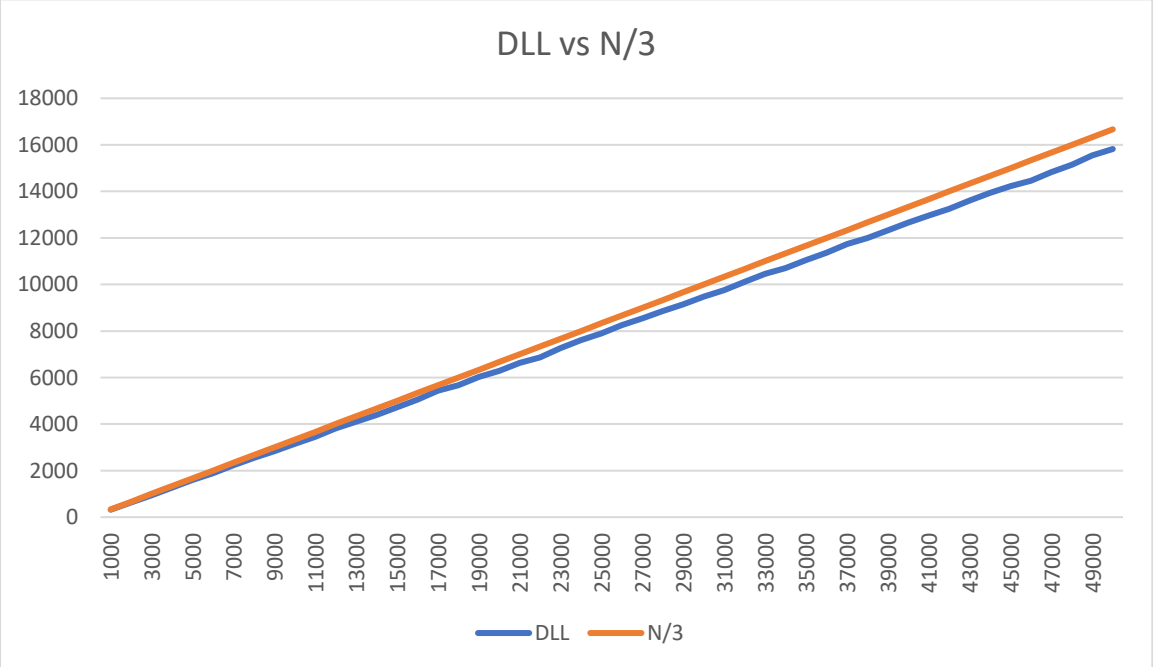
Com a part opcional de aquesta practica és demana fer un anàlisi comparatiu de l'operació buscar, aquesta, retorna el número de elements que s'han agut de recorre per comprovar l'existència o no existència d'un element. Hi ha un problema ocasionat per dos motius principals:

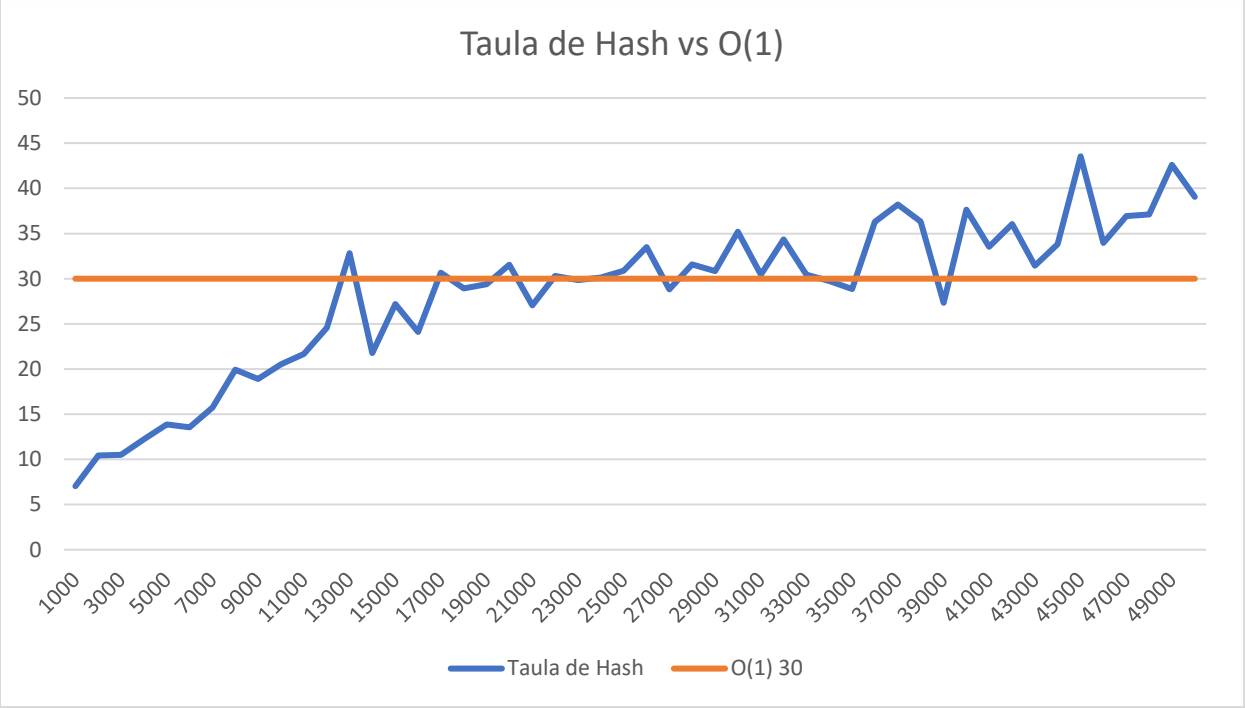
- A la especificació de inserir diu "Si l'element ja existia, actualitza el seu valor", això porta a que en una llista de N elements ([1000,2000,..,49000,50000]), on es guardaran valors aleatoris de 1-N/2, en la taula de hash es guardaran com a màxim N/2 elements. Aquesta taula llavors, sempre tindira menys de la meitat que la llista doblement enllaçada (DLL).
- El meu mètode de Hash per a enters retorna el mateix nombre al ser una funció lo suficientment exhaustiva no veia inconvenient a fer servir aquest mètode.

Per tant, la taula de hash que utilitzaré en aquest anàlisi es compondra de <String, Integer> onstring serà l'enter en forma de cadena. I permetré la repetició de valors per a tenir la taula amb N elements.

La classe que fa aquest anàlisi i crea el .csv s'anomena CostTest i està al Package CostComputacional.







La taula de hash te un cost de cerca gairebé constant ja que gracies als codis de hash i han poques col·lisions, si hagués utilitzat el mètode de hash dels int i afegit elements aleatoris de l'1 – N l'acces a la busqueda hauria sigut 1 ja que no hi haurien col·lision.

També m'he percatat que la DLL te un accés de busqueda de O(n) = n/3.