# Documentació Pràctica 1 Estructures de Dades

Iván Menacho Domínguez  
GTDAWiM  
2021/22  
18/04/22  
Estructures de Dades

## Índex:

1. Decisions de disseny
2. Joc de proves
3. Conclusions
4. Codi font

## 1.- Decisions de disseny

En aquesta pràctica tinc tres classes que son comuns per les dues parts. La primera seria la de PersonalException que es amb la que controlo les excepcions de tota la pràctica, després la classe Ciutada i la classe CiutadaIterator que son amb les que es fa el joc de proves.

Per la part 1.1 de la pràctica, és a dir, la llista doblement encadenada tinc la classe MainLinkedList, en la que faig el joc de proves, la classe DoublyLinkedList, en la que genero la llista amb les funcions especificades i la classe Node, en la que tinc les dades i els punters al següent i a l’anterior node.

Finalment, per la part 1.2 de la pràctica tinc altres 3 classes, les quals són la classe MainHashTable, en la que faig el joc de proves, la classe HashTable en la que genero la taula de hash encadenada indirecta i finalment una altra classe anomenada NodeHash, he de dir que he fet servir uns nodes diferents, ja que a les col·lisions de aquesta taula de hash he utilitzat una llista enllaçada simple per no complicar-me amb els punters al node anterior ja que em va semblar més fàcil.

Per fer la primera part de la pràctica no vaig tenir molt problemes, ja que l’any passat a Fonaments de Programació II vam treballar amb llistes enllaçades simples però també vam parlar de les dobles, per tant el concepte ja ho coneixia amb bastants exactitud i vaig poder posar-me a treballar bastant aviat. Primer vaig treballar amb enters i vaig anar fent proves de totes les funcions, fins que al final vaig acabar afegint els genèrics, el comparable i l’iterator.

Per la part de la taula de hash encadenada indirecta no he tingut molts problemes ja que la he fet després de l’examen de teoria i tenia bastant assolit el concepte. Aquí ja he treballar directament amb els genèrics utilitzant la classe Ciutada per les proves i ha sigut bastant ràpid de crear aquesta part de la pràctica.

En quant al cost computacional he realitzat una altre classe amb un main en la que creo 25 llistes de 2000 fins a 50000 elements, de 2000 en 2000 cada llista i calculo les mitges de cost que hi ha a cada estructura, el que passant-lo a un full de càlcul i veient la gràfica es pot observar el següent:

Per la línia blava, que es la de la llista enllaçada tenim un cost superior a la de la taula de hash, en aquest cas la línia taronja, en el que els costs van entre 4 i 6 nodes, per que podríem dir que la taula de hash te una cerca en cost O(1), ja que insertant la clau, com hi haurà menys col·lisions, haurà de recórrer una llista enllaçada simple molt petita. En canvi, la llista enllaçada haurà d’anar-se recorrent de node en node fins trobar el que interessa, per tant podríem dir que té un cost O(n).

També he calculat la desviació estàndard de cada estructura i a la de la taula de hash he obtingut una molt petita, en aquest cas de 0,627162924, i la de la llista, molt més gran, en aquest cas de 14719,0595.

## 2.- Joc de proves

Els jocs de proves de cada part de la pràctica esta especificat amb comentaris en cada main de cada part i es proven les funcions de cada apartat en totes les seves possibilitats.

En el cas de la llista doblement enllaçada, comencem creant la llista i els ciutadans que utilitzarem com a tests. Més tard els afegim i mostrem la llista per pantalla, després afegim un ciutadà a la llista però en aquest cas en una posició concreta i obtenim el valor d’aquesta posició per tal de veure que s’ha afegit correctament, després per tractar les excepcions intentem afegir el ciutadà però aquesta vegada en una posició inexistent, el que farà saltar l’excepció. Això ho farem amb un número més gran del nombre d’elements i amb un número negatiu. Mirarem la llargada de la llista, després crearem un nou ciutadà que encara no afegirem i utilitzarem la funció de buscar per veure si està i controlarem l’excepció ja que encara no l’haurem afegit. En canvi si ho fem amb un ciutadà que si hem afegit a la llista podrem veure que si que ens dona un resultat. Finalment fem unes comparacions entre dos ciutadans que ens diran la resposta de si tenen els DNI’s iguals gracies a la funció del compareTo.

Per la part de la taula de hash encadenada indirecta comencem també creant la taula de hash, en la que passarem per clau una String que serà el DNI del ciutadà i per valor un ciutadà. Després fem la creació dels ciutadans, però abans fem dues proves de les excepcions que poden saltar a la hora d’inserir, en aquest cas que la clau sigui nul·la o que el valor sigui nul. Ja inserim els ciutadans correctament i provem d’afegir un ciutadà amb la mateixa clau però amb diferent valor, el que farà que aquesta clau actualitzi el seu valor. Això ho comprovarem amb la funció d’obtenir. Després, si féssim un debug, podríem veure com s’han anat guardant els nodes a la taula, amb les seves col·lisions, per tant podríem veure el que tarda en buscar un element a partir dels salts. En el cas de la llista enllaçada simplement podríem fer-ho amb mostrar-la i contar quants nodes hi ha abans de trobar-la. Després utilitzem la funció en la que podem veure quants elements hi ha, borrem un i tornem a mostrar i podrem veure que el nombre d’elements haurà disminuït. Ja quasi al final provem la funció del factor de càrrega i podem veure que com hi ha més elements que posicions a la taula(degut a les col·lisions) el factor de càrrega és superior a 1, en el cas de que hi hagués menys elements que posicions a la taula, seria menor que 1. Ara sí finalment creem les llistes enllaçades amb les funcions d’obtenir claus i valors i les mostrem per pantalla amb la funció de mostrar les llistes feta a la part 1.1 de la pràctica.

## 3.- Conclusions

Amb aquesta pràctica he aconseguit assolir els coneixements necessaris per poder fer una llista doblement enllaçada la qual pot ser molt útil tan com el de utilitzar una bona taula de hash encadenada indirecta. També he pogut assolir els coneixements sobre el cost computacional sobre cada estructura de dades per poder escollir millor.

## 4.- Codi font

public class Ciutada implements Comparable<Ciutada>{  
 private final String name, surname, dni;  
  
 public Ciutada(String name, String surname, String dni){  
 this.name = name;  
 this.surname = surname;  
 this.dni = dni;  
 }  
  
 @Override  
 public String toString(){  
 return name+" "+surname+" "+dni;  
 }  
  
 @Override  
 public int compareTo(Ciutada other){  
 if(this.dni.equals(other.dni))  
 return 0;  
 else  
 return -1;  
 }  
  
 public String getDni(){  
 return dni;  
 }  
}

import java.util.Iterator;  
  
public class CiutadaIterator<T extends Comparable<T>> implements Iterator<T>{  
 private Node<T> current;  
  
 public CiutadaIterator(DoublyLinkedList<T> list){  
 this.current = list.getFirst();  
 }  
  
 @Override  
 public boolean hasNext() {  
 return current != null;  
 }  
  
 @Override  
 public T next() {  
 if(this.hasNext()){  
 T data = current.getData();  
 this.current = this.current.getNext();  
 return data;  
 }  
 return null;  
 }  
}

public class PersonalException extends Exception{  
 public PersonalException(String msg){  
 super(msg);  
 }  
}

public class Node<T> {  
 private final T data;  
 private Node<T> next;  
 private Node<T> prev;  
  
 public Node(T data){  
 this.data = data;  
 this.next = null;  
 this.prev = null;  
 }  
  
 public T getData(){  
 return data;  
 }  
  
 public Node<T> getNext(){  
 return next;  
 }  
  
 public Node<T> getPrev(){  
 return prev;  
 }  
  
 public void setNext(Node<T> n){ next = n; }  
  
 public void setPrev(Node<T> n){  
 prev = n;  
 }  
}

import java.util.Iterator;  
  
public class DoublyLinkedList<T extends Comparable<T>> implements Iterable<T>{  
 private Node<T> tail;  
 private Node<T> head;  
 private int nElem;  
  
 //Constructor  
 public DoublyLinkedList(){  
 head = null;  
 tail = null;  
 nElem = 0;  
 }  
  
  
 //Add node at the end of the list  
 public void addNode(T data){  
 Node<T> temp = new Node<>(data);  
 if(this.head == null){  
 head = temp;  
 tail = temp;  
 head.setPrev(null);  
 tail.setNext(null);  
 }else{  
 tail.setNext(temp);  
 temp.setPrev(tail);  
 tail = temp;  
 tail.setNext(null);  
 }  
 nElem++;  
 }  
  
 //Add node at a given position  
 public void addNodePos(int pos, T data) throws PersonalException {  
 Node<T> newNode = new Node<>(data);  
 newNode.setNext(null);  
 newNode.setPrev(null);  
  
 if(pos < 1) {  
 throw new PersonalException("Couldn't add the node in this position");  
 }  
 else if(pos > nElem)  
 throw new PersonalException("Invalid position");  
 else if(pos == 1){  
 newNode.setNext(head);  
 head.setPrev(newNode);  
 head = newNode;  
 }else{  
 Node<T> temp = head;  
 for(int i = 1; i < pos-1; i++) {  
 if (temp != null)  
 temp = temp.getNext();  
 }  
 if(temp != null){  
 newNode.setNext(temp.getNext());  
 newNode.setPrev(temp);  
 temp.setNext(newNode);  
 if(newNode.getNext() != null){  
 newNode.getNext().setPrev(newNode);  
 }  
 }else  
 System.*out*.println("The previous node is null");  
 }  
 nElem++;  
 }  
  
 //Return the data of the node of a specific position  
 public T getNode(int pos) throws PersonalException{  
 Node<T> aux = head;  
 int cont = 1;  
 T data = null;  
 if(head == null)  
 throw new PersonalException("The list is empty");  
 else if(pos < 1)  
 throw new PersonalException("Invalid postion");  
 else if(pos > nElem)  
 throw new PersonalException("Invalid postition");  
 else{  
 while(cont <= pos){  
 if(cont == pos)  
 data = aux.getData();  
 cont++;  
 aux = aux.getNext();  
 }  
 }  
 return data;  
 }  
  
 //Return the length of the list  
 public int listLength(){  
 return nElem;  
 }  
  
 //2 functions for delete a node and delete the node at given position  
 private void deleteNode(Node<T> del){  
 if(head == null || del == null)  
 System.*out*.println("test");  
 if(head == del)  
 head = del.getNext();  
 if(del.getNext() != null)  
 del.getNext().setPrev(del.getPrev());  
  
 if(del.getPrev() != null)  
 del.getPrev().setNext(del.getNext());  
  
 del = null;  
 }  
  
 public void deleteNodeAtGivenPos(int pos) throws PersonalException{  
 if(head == null || pos <= 0 || pos > nElem)  
 throw new PersonalException("Couldn't delete that node");  
  
 Node<T> current = head;  
  
 for(int i = 1; current != null && i < pos; i++)  
 current = current.getNext();  
  
 if(current == null)  
 return;  
  
 deleteNode(current);  
 nElem--;  
 }  
  
 //Function to search a node with the given data  
 public int search(T data) throws PersonalException{  
 int count = 1;  
 Node<T> aux = head;  
 if(head == null){  
 throw new PersonalException("The list is empty");  
 }else{  
 while(aux != null){  
 if(aux.getData() == data){  
 return count;  
 }else if(count == nElem)  
 throw new PersonalException("Node with the given data not found");  
 else{  
 count++;  
 aux = aux.getNext();  
 }  
 }  
 }  
 return count;  
 }  
  
 //Show all elements from the doubly linked list  
 public void showAll(DoublyLinkedList<T> list){  
 Iterator<T> i =list.iterator();  
 while(i.hasNext()){  
 System.*out*.println(i.next());  
 }  
 }  
  
 public Node<T> getFirst(){  
 return head;  
 }  
  
 @Override  
 public Iterator<T> iterator() {  
 return new CiutadaIterator<>(this);  
 }  
}

public class MainLinkedList {  
  
 public static void main(String[] args){  
 //Constructor de la llista  
 DoublyLinkedList<Ciutada> list = new DoublyLinkedList<>();  
  
 //Creació dels elements de la llista  
 Ciutada test1 = new Ciutada("Ivan", "Menacho", "48275497S"); //Mateix DNI que Joel  
 Ciutada test2 = new Ciutada("Joel", "Marti", "48275497S"); //Mateix DNI que Iván  
 Ciutada test3 = new Ciutada("Antonio", "Matas", "46275497S");  
 Ciutada test4 = new Ciutada("Ivan", "Gonzalez", "45275497S");  
 Ciutada test5 = new Ciutada("Josema", "Ramos", "44275497S");  
 Ciutada test6 = new Ciutada("Adria", "Chipre", "43275497S");  
 Ciutada test7 = new Ciutada("Adrian", "Izquierdo", "42275497S");  
 Ciutada test8 = new Ciutada("Juan", "Menacho", "41275497S");  
 Ciutada test9 = new Ciutada("Montse", "Dominguez", "40275497S");  
 Ciutada test10 = new Ciutada("Patri", "Menacho", "49275497S");  
  
 list.addNode(test1);  
 list.addNode(test2);  
 list.addNode(test3);  
 list.addNode(test4);  
 list.addNode(test5);  
 list.addNode(test6);  
 list.addNode(test7);  
 list.addNode(test8);  
 list.addNode(test9);  
 //list.addNode(test10); Aquest l'afegirem més tard en una posició concreta per comprovar que funciona  
 //Mostrem la llista per consola  
 list.showAll(list);  
  
 try{  
 //Afegim el ciutadà a la posició número 3  
 list.addNodePos(3, test10);  
 }catch(PersonalException e){  
 System.*err*.println("No s'ha pogut afegir");  
 }  
  
 //Utilitzem la funció obtenir per veure si a la 3ra posició s'ha afegit la Patri correctament, primer salt de linia perque quedi més clar  
 try {  
 System.*out*.println(list.getNode(3));  
 }catch(PersonalException e){  
 System.*err*.println("No s'ha pogut obtenir la posició");  
 }  
 //Efectivament s'haura afegit a la tercera posició  
 //Ara l'afegirem pero a una posició inexistent perque sali l'excepció  
 try{  
 //Afegim el ciutadà a la posició número 15, que es mes gran del nombre d'elements  
 list.addNodePos(15, test10);  
 }catch(PersonalException e){  
 System.*err*.println("No s'ha pogut afegir");  
 //Salta l'excepció, ho controla correctament  
 }  
  
 //Ara intentarem obtenir una posició de la llista inexistent, com la posició -5  
 try{  
 System.*out*.println(list.getNode(-5));  
 }catch(PersonalException e){  
 System.*err*.println("No s'ha pogut obtenir la posició");  
 }  
 //En aquest cas saltarà l'excepció  
 //Ara obtindrem la longitud de la llista, que en aquest cas hauria de ser 10, ja que estan afegits tots els ciutadans.  
 System.*out*.println("La longitud de la llista es de: "+list.listLength()+" elements"); //Funciona correctament  
  
 //Ara esborrarem de la llista a la Patri, que era a la posició 3 si recordem  
 try {  
 list.deleteNodeAtGivenPos(3);  
 }catch(PersonalException e){  
 System.*err*.println("No es pot esborrar en aquesta posicio");  
 }  
 //Imprimim la llista i ens sortirà que ja no hi és i que la longitud de la llista serà de 9 elements  
 list.showAll(list);  
 System.*out*.println("La longitud de la llista es de: "+list.listLength()+" elements"); //Funciona correctament  
  
 //Ara guardarem en una variable unes dades, i les passarem a la funció de buscar per veure si existeix algun element de la llista  
 //amb les dades introduides  
 Ciutada dadesTest = new Ciutada("test", "test", "dniTest");  
 try{  
 System.*out*.println("Les dades es troben a la posició: "+list.search(dadesTest));  
 }catch(PersonalException e){  
 System.*err*.println("Les dades no es troben a la llista");  
 }  
 //Donara error ja que les dades no son a la llista, pero si probem amb un ciutada afegit abans podrem veure que si ens dona resultat.  
 try{  
 System.*out*.println("Les dades es troben a la posició: "+list.search(test4));  
 }catch(PersonalException e){  
 System.*err*.println("Les dades no es troben a la llista");  
 }  
  
 //Finalment compararem dos ciutadans per veure si són el mateix, en aquest cas, si els DNI són els mateixos, seran el mateix.  
 //Si la funció en retorna 0, són el mateix DNI, si retorna -1 vol dir que tenen DNI diferents.  
 //Primer probarem amb l'Iván i el Joel, que tenen el mateix DNI i hauria de retornar 0  
 System.*out*.println("Comparacio DNI Iván i Joel: "+test1.compareTo(test2)); //Retorna 0 són iguals  
 //Ara probarem amb l'Iván i la Patri, que com tenen DNI diferents, haura de retornar -1  
 System.*out*.println("Comparacio DNI Iván i Patri: "+test1.compareTo(test10));//Retorna -1, són diferents  
 }  
}

public class NodeHash<K extends Comparable<K>, V> {  
 K key;  
 V value;  
 NodeHash<K, V> next;  
  
 public NodeHash(K key, V value, NodeHash<K,V> next){  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
}

public class HashTable<K extends Comparable<K>, V extends Comparable<V>> {  
 private NodeHash[] hashTable;  
 private int nElem;  
 private double loadFactor;  
  
 //Constructor  
 public HashTable(int size) {  
 this.hashTable = new NodeHash[size];  
 this.nElem = 0;  
 loadFactor = 0.75;  
 }  
  
 //Add function  
 public void add(K key, V value) throws PersonalException {  
 boolean added = false;  
 if (key == null)  
 throw new PersonalException("Null key");  
 else if (value == null)  
 throw new PersonalException("Null value");  
  
 int index = hashFunction((String) key) % hashTable.length;  
  
 if (hashTable[index] == null) {  
 hashTable[index] = new NodeHash<>(key, value, null);  
 nElem++;  
 } else {  
 NodeHash temp = hashTable[index];  
 while (temp != null && added == false) {  
 if (temp.key.compareTo(key) == 0) { //mateixa clau, actualitzem valor  
 temp.value = value;  
 break;  
 }  
 if (temp.next == null) {  
 temp.next = new NodeHash<>(key, value, null);  
 nElem++;  
 }  
 temp = temp.next;  
 }  
 }  
 }  
  
 //Function to get a value by passing the key  
 public V get(K key) throws PersonalException{  
 int index = hashFunction((String) key) % hashTable.length;  
  
 if(key == null)  
 throw new PersonalException("Null key");  
 if (hashTable[index] == null)  
 return null;  
 else {  
 NodeHash temp = hashTable[index];  
 while (temp != null) {  
 if (temp.key.compareTo(key) == 0) {  
 return (V) temp.value;  
 }  
 temp = temp.next;  
 }  
 return null;  
 }  
 }  
  
 //Function to get the nElems of the hash table  
 public int getnElem() {  
 return nElem;  
 }  
  
  
 //Function to remove a node by passing the key  
 public void remove(K key) throws PersonalException{  
 int index = hashFunction((String) key) % hashTable.length;  
  
 if(key == null)  
 throw new PersonalException("Null key");  
  
 if (hashTable[index] == null) {  
 return;  
 } else {  
 NodeHash previous = null;  
 NodeHash current = hashTable[index];  
  
 while (current != null) {  
 if (current.key.compareTo(key) == 0) {  
 if (previous == null) {  
 hashTable[index] = current.next;  
 } else {  
 previous.next = current.next;  
 }  
  
 nElem--;  
 return;  
 }  
 previous = current;  
 current = current.next;  
 }  
 }  
 }  
  
 //Function that check if an element is at the table  
 public int search(K key) throws PersonalException{  
 int count = 1;  
 int index = hashFunction((String) key) % hashTable.length;  
  
 if(key == null)  
 throw new PersonalException("Null key");  
  
 if(hashTable[index] == null){  
 throw new PersonalException("Empty table");  
 }else{  
 NodeHash temp = hashTable[index];  
 while (temp != null) {  
 if (temp.key.compareTo(key) == 0) {  
 return count;  
 }  
 temp = temp.next;  
 count++;  
 }  
 }  
 return count;  
 }  
  
 //Function that returns a doubly linked list with the values of the elements of the table  
 public DoublyLinkedList<V> getValues(){  
 DoublyLinkedList<V> temp = new DoublyLinkedList<V>();  
  
 for(int i = 0; i < hashTable.length; i++){  
 if(hashTable[i] != null){  
 temp.addNode((V) hashTable[i].value);  
 NodeHash current = hashTable[i];  
 while(current.next!=null){  
 temp.addNode((V) current.next.value);  
 current = current.next;  
 }  
 }  
 }  
 return temp;  
 }  
  
 public DoublyLinkedList<K> getKeys(){  
 DoublyLinkedList<K> temp = new DoublyLinkedList<K>();  
  
 for(int i = 0; i < hashTable.length; i++){  
 if(hashTable[i] != null){  
 temp.addNode((K) hashTable[i].key);  
 NodeHash current = hashTable[i];  
 while(current.next!=null){  
 temp.addNode((K) current.next.key);  
 current = current.next;  
 }  
 }  
 }  
 return temp;  
 }  
  
 //Function that returns the load factor  
 double loadFactor(){  
 return (double)nElem/hashTable.length;  
  
 }  
  
 //Hash function  
 public int hashFunction(String key){  
 int k = key.length();  
 int sum = 0;  
 for(int i = 0; i < k-1; i++)  
 sum += key.charAt(i)<<(5\*i); //desplaça bits del operand cap a l'esquerra les posicions indicades  
 return Math.*abs*(sum);  
 }  
  
  
  
  
}

public class MainHashTable {  
 public static void main(String[] args){  
  
 //Creació de la taula  
 HashTable<String, Ciutada> table = new HashTable<>(3);  
  
 //Creacio dels ciutadans  
 Ciutada test1 = new Ciutada("Antonio", "Rodriguez", "48275497S");  
 Ciutada test2 = new Ciutada("Juan", "Martin", "48275497S");  
 Ciutada test3 = new Ciutada("Nuria", "Quesada", "46275497S");  
 Ciutada test4 = new Ciutada("Paula", "Gonzalez", "45275497S");  
 Ciutada test5 = new Ciutada("Mireia", "Ramos", "44275497S");  
 Ciutada test6 = new Ciutada("Xavi", "Chipre", "43275497S");  
 Ciutada test7 = new Ciutada("Jose", "Izquierdo", "42275497S");  
 Ciutada test8 = new Ciutada("Miguel", "Hernandez", "42005497S");  
  
 //Insercio dels ciutadans amb excepció, una clau es nula, salta excepció  
 try {  
 table.add(null, test1);  
 }catch (PersonalException e){  
 System.*err*.println(e);  
 }  
  
 //Insercio dels ciutadans amb excepció, un valor es nul, salta excepció  
 try {  
 table.add(test1.getDni(), null);  
 }catch (PersonalException e){  
 System.*err*.println(e);  
 }  
  
 //Inserció correcta dels ciutadans sense que salti l'excepcio  
 try{  
 table.add(test1.getDni(), test1);  
 table.add(test2.getDni(), test2);  
 table.add(test3.getDni(), test3);  
 table.add(test4.getDni(), test4);  
 table.add(test5.getDni(), test5);  
 table.add(test6.getDni(), test6);  
 table.add(test7.getDni(), test7);  
 //Afegim un altre amb la clau del DNI de test1 per comprovar que actualitzem el valor  
 table.add(test1.getDni(), test8);  
 }catch (PersonalException e){  
 System.*err*.println(e);  
 }  
  
 //Si ara obtenim el valor de la clau del DNI de test1, veurem que el valor es test7 i no test1, es a dir  
 //Miguel Hernandez, no Antonio Rodriguez  
 try{  
 System.*out*.println("El valor de la clau del DNI del test1 és: "+table.get(test1.getDni()));  
 }catch (PersonalException e){  
 System.*err*.println(e);  
 }  
  
 //Ara si debuguessim veuriem per quants nodes hem de passar abans de arribar al que volem  
 try {  
 System.*out*.println("El numero de nodes per el que s'ha de passar per trobar test4 es: "+table.search(test4.getDni()));  
 }catch(PersonalException e){  
 System.*err*.println(e);  
 }  
  
 //La mida actual de la taula es 6 ja que hem actuializat un, no hem afegit 7  
 System.*out*.println("La mida de la taula es: "+table.getnElem());  
 //Si esborrem un, hauria de baixar a 5  
 try{  
 table.remove(test1.getDni());  
 }catch (PersonalException e){  
 System.*err*.println(e);  
 }  
  
 System.*out*.println("La mida de la taula ara es de: "+table.getnElem());  
  
 //Ara podem obtenir el factor de carrega, en aquest superior a 1 ja que hi ha mes elements  
 //que posicions a la taula, perque s'han emmagatzemat en colisions  
 System.*out*.println("El factor de carrega es de: "+table.loadFactor());  
  
 //Ara creem les dues llistes doblement enllaçades de la part 1.1 i les mostrem amb les seves funcions  
 DoublyLinkedList claus = table.getKeys();  
 DoublyLinkedList valors = table.getValues();  
  
 System.*out*.println();  
 System.*out*.println("Claus: ");  
 claus.showAll(claus);  
 System.*out*.println();  
 System.*out*.println("Valors: ");  
 valors.showAll(valors);  
 }  
}

public class CostComputacional {  
 public static void main(String[] args) throws PersonalException{  
 for(int size = 2000; size < 50001; size = size + 2000){  
 DoublyLinkedList<Integer> llista = new DoublyLinkedList<>();  
 HashTable<Integer,Integer> taula = new HashTable<>(size);  
  
 int[] num = new int[size];  
 for(int i = 0; i < taula.getSize(); i++){  
 num[i] = (int) Math.*floor*(Math.*random*()\* taula.getSize()/2+1);  
 taula.add(num[i], num[i]);  
 llista.addNode(num[i]);  
 }  
 long mitjanaLlista;  
 long mitjanaTaula;  
 long totalLlista = 0;  
 long totalTaula = 0;  
 for(int i = 0; i < taula.getSize(); i++){  
 totalTaula += taula.search(num[i]);  
 totalLlista += llista.search(num[i]);  
 }  
 mitjanaLlista = totalLlista / taula.getSize();  
 mitjanaTaula = totalTaula / taula.getSize();  
 System.*out*.println("Mida llista: "+size+", mitjana llista: "+mitjanaLlista);  
 System.*out*.println("Mida taula: "+size+", mitjana taula: "+mitjanaTaula);  
 }  
  
 }  
}