Fecha

09-11-2015

Peio Valle  
Sergio Erlantz Tobal

GestIon imdb

De Actores y Peliculas

Contenidos

[Introducción y descripción general 2](#_Toc432336952)

[Estructuras de datos principales y alternativas examinadas 2](#_Toc432336953)

[Diseño de las clases 3](#_Toc432336954)

[Diseño e implementación de los algoritmos principales 4](#_Toc432336955)

[Método buscarElemento 4](#_Toc432336956)

[Código 5](#_Toc432336957)

[Clase Actor 5](#_Toc432336958)

[Clase Pelicula 5](#_Toc432336959)

[Clase ListaActores 6](#_Toc432336960)

[Clase ListaPeliculas 7](#_Toc432336961)

[Clase CatalogoActores 8](#_Toc432336962)

[Clase CatalogoPeliculas 8](#_Toc432336963)

[Clase Fichero 9](#_Toc432336964)

[Clase Stopwatch 10](#_Toc432336965)

[Prueba Actor 10](#_Toc432336966)

[Prueba Pelicula 11](#_Toc432336967)

[Prueba ListaActores 12](#_Toc432336968)

[Prueba ListaPeliculas 13](#_Toc432336969)

[Prueba CatalogoActores 14](#_Toc432336970)

[Prueba CatalogoPeliculas 15](#_Toc432336971)

[Conclusiones y resultados empíricos de las pruebas 16](#_Toc432336972)

# Introducción y descripción general

Se nos pide un programa en el cual recojamos de un fichero una cantidad enorme de nombres, tanto de actores/actrices como de películas para guardarlos en una estructura de datos y trabajar con ellos. En el archivo estarán juntos tanto los nombres como los títulos de las películas separados por ###. Se deberá poder buscar actores, películas, borrar, insertar… Basicamente todas aquellas funciones que se suelen ver en una base de datos y se llaman CRUD (Create, Read, Update, Delete).

# Estructuras de datos principales y alternativas examinadas

En un principio se pensó en usar usar estructuras de datos lo mas eficientes posibles, asi pues estuvimos mirando las 3 colecciones mas conocidas en Java, las listas, los sets y los maps. Veámoslos en una comparativa.

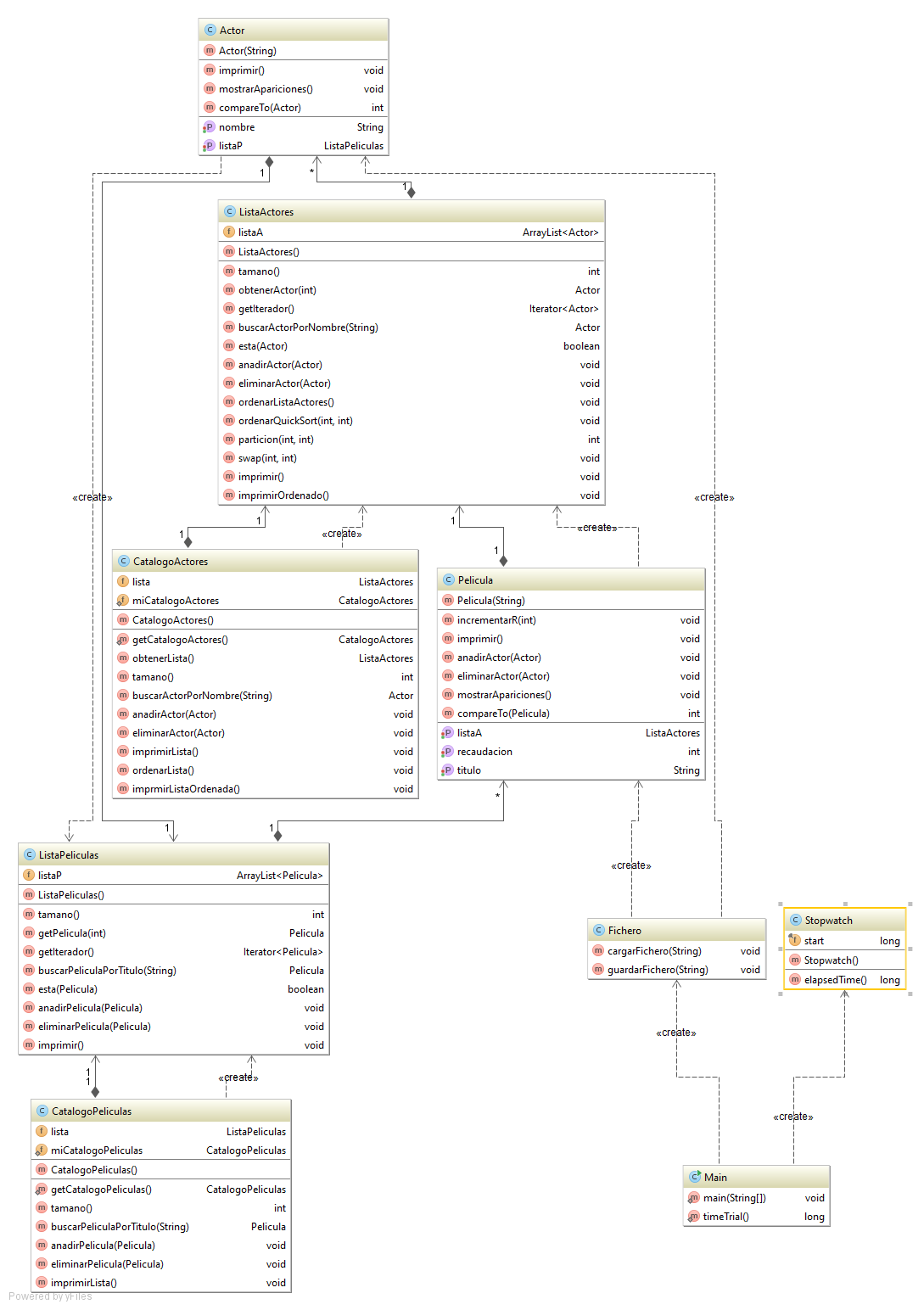
|  |  |  |  |
| --- | --- | --- | --- |
|  | Lists | Maps | Sets |
| Admiten Duplicidad | Si | Keys únicas, valores Si | No |
| Permitir elementos null | Si | Solo 1 key, valores todos los que se quiera | Solo 1 debido a la no duplicidad |
| Ordenadas | Si | No, porque va por keys | No ordenadas |
| Implementaciones | Arraylist, Linkedlist, Vector | Hashmap, LinkedHashmap, Hashtable, Treemap… | Hashset, LinkedHashset, TreeSet, SortedMap, SortedSet… |

Aun asi algunas implementaciones no cumplen lo mostrado en la tabla, por ejemplo Hashtable no admite keys o valores nulos o TreeMap y TreeSet que mantienen un orden impuesto por usando Comparator o Comparable.

Tambien lo hemos estado mirando en función del tiempo que tardarían las distintas operaciones en cada tipo de estructuras, y aunque queríamos implementar arboles ya que nunca habíamos trabajado con ellos al final se nos convenció que lo mejor era empezar con un Arraylist que la hemos usado anteriormente y sabemos su funcionamiento.

Por otro lado estuvimos mirando en algoritmos de ordenación cual seria mas eficiente y por lo tanto cual usar, el primero que nos planteamos era el Timsort, el cual se creo en 2002 con ideas de mergeSort (crear subarrays y ordenarlos cuando se juntan) y de insertionSort (cojer un elemento y lo mueve mirando los demás elementos y colocándolo ordenadamente en el array), y que su fundamental diferencia es que busca subarrays ordenados para que el tiempo sea menor, aunque si el array es demasiado pequeño se tiende a usar directamente InsertionSort. Desechamos este algoritmo ya que la forma de usarlo es llamar a clases ya hechas de Java y no sabíamos si esto estaba permitido y nos decantamos por el QuickSort que es el que vimos en clase que más rendimiento nos daba, aunque el mergesort es el mas estable.

# Diseño de las clases



Como se puede ver hemos creado clases para los modelos, las listas y los catalogos, asi como una clase para el cronometro y otra para la gestión del fichero. Los catalogos son clases Singleton que tienen listas de objetos que a su vez tienen los modelos. Por ultimo estarían las clases auxiliares de Fichero, que se encarga de las funciones de guardar y cargar archivos, y el Stopwatch que es el cronometro para el tiempo.

# Diseño e implementación de los algoritmos principales

En este apartado se presentarán los métodos principales, indicando por cada uno su especificación (precondiciones y postcondiciones), además de los casos de prueba planteados. Por cada método no trivial, se presentará la descripción del algoritmo implementado. Finalmente, se deberá presentar, de manera razonada, el cálculo del coste de cada algoritmo.

Presentamos un ejemplo de una posible manera de presentar un método de ejemplo.

## Método buscarPeliculaPorTitulo

public Pelicula buscarPeliculaPorTitulo(String titulo){

/\* Precondición: el array no está vacío

/\* Postcondición: devuelve la película buscada, en caso de no estar devuelve null

Casos de prueba:

* Búsqueda de un elemento en lista vacía
* Búsqueda de un elemento en lista de un solo elemento:
  + El elemento buscado es igual al de la lista
  + El elemento buscado no es el de la lista
* Búsqueda en una lista no vacía
  + El elemento buscado no está en la lista
  + El elemento buscado es el primero de la lista
  + El elemento buscado es el último de la lista
  + El elemento buscado está en la mitad de la lista

Implementación del algoritmo:

Pelicula seBusca = null;  
boolean enc = false;  
Iterator<Pelicula> itr = this.getIterador();  
while (!enc && itr.hasNext()){  
 seBusca = itr.next();  
 if (seBusca.getTitulo().equals(titulo)){  
 enc = true;  
 }  
}  
if (!enc){  
 seBusca=null;  
}  
return seBusca;

Coste: el algoritmo, en el caso peor, examina todos los elementos de la lista. Para una lista de n elementos, el coste será O(n).

## Método buscarActorPorNombre

public Pelicula buscarActorPorNombre(String nombre){

/\* Precondición: el array no está vacío

/\* Postcondición: devuelve el actor buscado, en caso de no estar devuelve null

Casos de prueba:

* Búsqueda de un elemento en lista vacía
* Búsqueda de un elemento en lista de un solo elemento:
  + El elemento buscado es igual al de la lista
  + El elemento buscado no es el de la lista
* Búsqueda en una lista no vacía
  + El elemento buscado no está en la lista
  + El elemento buscado es el primero de la lista
  + El elemento buscado es el último de la lista
  + El elemento buscado está en la mitad de la lista

Implementación del algoritmo:

Actor seBusca = null;  
boolean enc = false;  
Iterator<Actor> itr = this.getIterador();  
while (!enc && itr.hasNext()){  
 seBusca = itr.next();  
 if (seBusca.getNombre().equals(nombre)){  
 enc = true;  
 }  
}  
if (!enc){  
 seBusca=null;  
}  
return seBusca;

Coste: el algoritmo, en el caso peor, examina todos los elementos de la lista. Para una lista de n elementos, el coste será O(n).

## Método cargarFichero

public Pelicula cargarFichero(String path){

/\* Precondición: el fichero existe

/\* Postcondición: carga en los catalogos todo lo que el fichero contiene separado por strings

Casos de prueba:

* Cargar un fichero vacío
* Intentar cargar un fichero que no existe
* Cargar un fichero con muchos elementos

Implementación del algoritmo:

Pelicula objPeli;  
Actor objActor;  
CatalogoActores catActores = CatalogoActores.getCatalogoActores();  
String[] a;  
HashMap<String, Pelicula> hm;  
  
int cont = 0;  
  
try {  
 hm = new HashMap<String,Pelicula>();  
 BufferedReader br = new BufferedReader(new FileReader(path));  
 String line = br.readLine();  
 Pattern patron = Pattern.compile("\\s+###\\s+");  
 while (line != null){  
 a = patron.split(line);//Por cada linea corta nombres y mete en posiciones del array  
 objActor = new Actor(a[0]);//Creamos objeto Actor con su nombre  
 catActores.anadirActor(objActor);  
  
 for (int i = 1; i < a.length; i++) {  
 objPeli = hm.get(a[i]);  
 if (objPeli == null){  
 objPeli = new Pelicula(a[i]);  
 hm.put(a[i], objPeli);  
 }  
 objPeli.anadirActor(objActor);  
 objActor.getListaP().anadirPelicula(objPeli);  
 }  
 cont++;  
 line = br.readLine();//lectura siempre al final  
 if(cont % 10000==0){System.out.println(cont);}//Imprime los numeros divisores de 10.000 para saber como va haciendolo  
 }  
 br.close();  
  
 for (Pelicula p :hm.values()) {  
 CatalogoPeliculas.getCatalogoPeliculas().anadirPelicula(p);  
 }  
} catch (FileNotFoundException e) {  
 e.printStackTrace();  
} catch (IOException e) {  
 System.out.println("Archivo no encontrado");  
}

Coste:

## Método guardarFichero

public Pelicula guardarFichero(String path){

/\* Precondición: el fichero existe

/\* Postcondición: Guarda todo en un fichero

Casos de prueba:

* Guardar en un fichero vacío
* Intentar guardar un fichero que no existe
* Guardar en un fichero que contiene elementos

Implementación del algoritmo:

CatalogoActores catAct = CatalogoActores.getCatalogoActores();  
try {  
 BufferedWriter bw = new BufferedWriter(new FileWriter(path));  
 Actor a;  
 for (int i = 0; i < catAct.tamano(); i++) {  
 a = CatalogoActores.getCatalogoActores().obtenerLista().obtenerActor(i);  
 bw.write(a.getNombre());  
  
 for (int j = 0; j < a.getListaP().tamano(); j++) {  
 bw.append(" ### ");  
 bw.write(a.getListaP().getPelicula(j).getTitulo());  
 }  
 bw.flush();  
 bw.newLine();  
 }  
 bw.close();  
  
} catch (IOException e) {  
 e.printStackTrace();  
}

Coste:

## Método ordenar

# Código

Clase Actor



Clase Pelicula



Clase ListaActores

Clase ListaPeliculas



Clase CatalogoActores



Clase CatalogoPeliculas

Clase Fichero

Clase Stopwatch



A continuación, se presentan las JUnit donde algunas están hechas con JUnit4 y otras con TestNG que probaba los métodos más rápido, asi que hemos podido ver lo que dice la lógica, que estas pruebas no están pensadas para saber los tiempos empíricos sino para probar que el método funcione.

Prueba Actor



Prueba Pelicula



Prueba ListaActores



Prueba ListaPeliculas



Prueba CatalogoActores



Prueba CatalogoPeliculas



# Conclusiones y resultados empíricos de las pruebas

Hemos aprendido a trabajar con distintas estructuras de datos asi como calcular los costes de tiempo de distintas funciones, además de repasar como se usaban los ficheros y las listas.

Hemos usado un Hashmap para la carga de los actores aunque estuvimos tentados de usar un Hashtable por su particularidad de no permitir duplicados y ser prácticamente iguales. En las listas usamos ArrayList, de búsqueda usamos una búsqueda lineal con el Iterator ya que tiene ventaja sobre las colleciones que no están ordenadas, y por ultimo en el método de ordenación usamos el QuickSort ya que implementar el TimSort no era posible.

Nuestro principal problema ha sido el algoritmo de ordenación y las pruebas unitarias ya que estas ultimas hacían que el tiempo de nuestro programa aumentara bastante, lo comprobamos al hacer las llamadas desde un Main.java y al cambiar de librería de tests.

Con este trabajo se podría seguir e implementar todos los actores películas que contiene IMDB pero seria una carga bastante grande pero al haberlo hecho por objetos no seria muy diferente de lo que se hace con las bases de datos no relacionales y se podría seguir esta línea de trabajo y asi aprender como funcionan a la vez que se usan las bases de datos para permitir mayor cantidad de datos.

Resultados empíricos

* Cargar Fichero:
* Guardar Fichero:
* Busqueda de actor por nombre Desordenado:
* Listar Desordenado:
* Listar Ordenado:
* Busqueda de actor por nombre Ordenado: