Fecha

09-11-2015

Peio Valle  
Sergio Erlantz Tobal

GestIon imdb

De Actores y Películas

Contenidos

[Introducción y descripción general 2](#_Toc432612717)

[Estructuras de datos principales y alternativas examinadas 2](#_Toc432612718)

[Diseño de las clases 3](#_Toc432612719)

[Diseño e implementación de los algoritmos principales 4](#_Toc432612720)

[Método buscarPeliculaPorTitulo 4](#_Toc432612721)

[Método buscarActorPorNombre 5](#_Toc432612722)

[Método cargarFichero 5](#_Toc432612723)

[Método guardarFichero 6](#_Toc432612724)

[Método ordenar 7](#_Toc432612725)

[Código 8](#_Toc432612726)

[Clase Actor 8](#_Toc432612727)

[Clase Pelicula 8](#_Toc432612728)

[Clase ListaActores 9](#_Toc432612729)

[Clase ListaPeliculas 10](#_Toc432612730)

[Clase CatalogoActores 11](#_Toc432612731)

[Clase Catalogo Películas 11](#_Toc432612732)

[Clase Fichero 12](#_Toc432612733)

[Clase Stopwatch 13](#_Toc432612734)

[Prueba Actor 13](#_Toc432612735)

[Prueba Pelicula 14](#_Toc432612736)

[Prueba Lista Actores 15](#_Toc432612737)

[Prueba Lista Películas 16](#_Toc432612738)

[Prueba Catalogo Actores 17](#_Toc432612739)

[Prueba Catalogo Películas 18](#_Toc432612740)

[Conclusiones y resultados empíricos de las pruebas 18](#_Toc432612741)

# 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… Básicamente 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 más eficientes posibles, así pues, estuvimos mirando las 3 colecciones más conocidas en Java, las listas, los sets y los maps. Veámoslos en una comparativa.

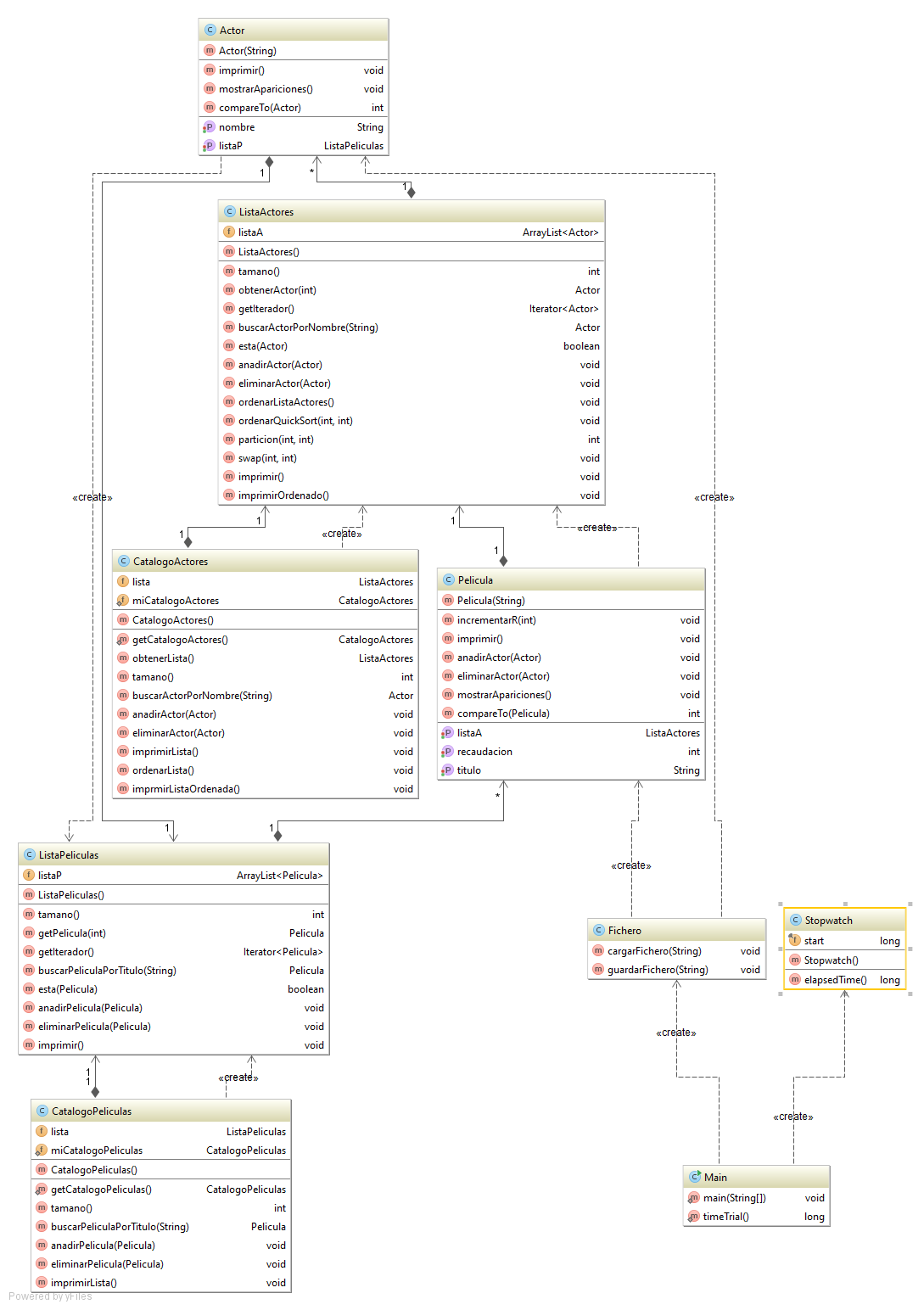
|  |  |  |  |
| --- | --- | --- | --- |
|  | Lists | Maps | Sets |
| Admiten Duplicidad | Si | Keys únicas, valores Si | No |
| Permitir elementos null | Si | Solo 1 key, valores todos los que se quiera | Solo 1 debido a la no duplicidad |
| Ordenadas | Si | No, porque va por keys | No ordenadas |
|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

Aun así, 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.

También 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 sería más eficiente y por lo tanto cual usar, el primero que nos planteamos era el Timsort, el cual se creó en 2002 con ideas de mergeSort (crear subarrays y ordenarlos cuando se juntan) y de insertionSort (coger 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 más estable.

# Diseño de las clases



Como se puede ver hemos creado clases para los modelos, las listas y los catálogos, así como una clase para el cronometro y otra para la gestión del fichero. Los catálogos 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

## 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:

Declaramos un objeto Pelicula llamado seBusca

Inicializamos el iterador itr

Mientras itr tenga elementos o no se haya encontrado

seBusca = siguiente elemento  
 Si el titulo es igual al que tenemos en seBusca  
 se ha encontrado y se sale de bucle  
 }  
fin mientras  
Si no existe el objeto de seBusca se inicializa a null  
devolver 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:

Declaramos el objeto Actor seBusca

Declaramos la variable boolean enc = false;  
inicializamos el iterador itr  
mientras que no se haya encontrado o haya un siguiente se repite  
 seBusca = siguiente;  
 si el nombre que nos dan y el nombre del seBusca es igual

Se encuentra y salimos del bucle

fin mientras  
si no se ha encontrado seBusca es null  
devolver 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:

Declaramos el objeto Pelicula objPeli;  
Declaramos el objeto Actor objActor;  
Inicializamos el CatalogoActores catActores

Declaramos la variable String[] a;  
HashMap<String, Pelicula> hm;  
Declaramos la variable int cont = 0;  
 hm = new HashMap<String,Pelicula>();  
 Busca el fichero del path que le pasamo como dato  
 Introcuimos en String line la linea  
 Creamo el patrón Pattern patron = Pattern.compile("\\s+###\\s+");

mientras haya líneas se repite  
 metemos en a la línea sin el patrón anterior y separado  
 Creamos objeto Actor con su nombre  
 Añadimos el actor al Catalogo  
  
 por cada elemento en a empezando desde posición 1 repetimos  
 creamos los objetos película con el elem de la posicion  
 le añadimos el actor a la pelicula  
 añadimos la película a la lista de películas del actor  
 incrementamos cont  
 convertimos la línea en la siguiente línea del fichero  
 Imprime los numeros divisores de 10.000 para saber cómo va  
 repetir por cada elemento película que tengamos  
 añadimos las películas al catalogo

fin repetir

fin repetir

fin mientras

Coste: Teniendo en cuenta que los actores sean n y las películas m el coste sería O(n\*m), pero si lo miramos con detenimiento al tener que crear el catálogo de las películas que podríamos decir que su coste es p el coste final sería O(n\*m+p).

## 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:

Inicializamos el CatalogoActores catAct  
Declaramos el objeto Actor a;  
 por cada actor en la lista repetimos  
 añadimos el nombre del actor al fichero  
 repetimos por cada película del acotr  
 añadimos el separador (" ### ")  
 escribimos el título de la pelicula  
 fin repetir  
 nueva linea  
 fin repetir

Coste: Teniendo en cuenta que el número de actores sea n y el número de películas de cada actor el coste sería O(n\*m).

## Método ordenar

public void ordenarListaActores(){

/\* Precondición: el array esta lleno

/\* Postcondición: ordena la lista por apellido

Casos de prueba:

* Ordenar un array vacío
* Ordenar un array de un solo elemento
* Ordenar un array con muchos elementos

Implementación del algoritmo:

llamada al método ordenarQuickSort desde el elemento 0 hasta el tamaño de la lista -1 porque si no nos saldríamos del array

Coste: En el peor de los casos deberíamos de recorrer todo con lo que el coste es O(n\*log2n).

Los otros métodos que lo complementan:

private void ordenarQuickSort(int pInicio, int pFin){  
 si la resta del valor inicial y el final es mayor que 0 entonces  
 llamamos al método partición y que devuelve un número  
 llamamos a este metodo desde el pivote en adelante  
 llamamos a este metodo desde la parte de atrás del pivote   
  
private int particion(int i, int f){  
 Declaramos el objeto Actor actorPivote = this.listaA.get(i);  
 Declaramos la variable int izq = i;  
 Declaramos la variable int der = f;  
  
 mientras que el valor de la izquierda < derecha repetir  
 mientras que la comparación del actor con el pivote sea <=0 y que el valor de la izquierda < derecha repetir incrementamos izq

mientras que la comparación del actor con el pivote sea >0 repetir incrementamos der  
 si la izquierda menor que la derecha intercambiamos

fin mientras  
 metemos en el índice el valor de la posición der  
 metemos en la posición der el pivote  
 return der;  
  
private void swap(int one, int two) {

cambiamos el elemento primero por el segundo

# Código

Clase Actor



Clase Pelicula



Clase ListaActores

Clase ListaPeliculas



Clase CatalogoActores



Clase Catalogo Películas

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, así 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 Lista Actores



Prueba Lista Películas



Prueba Catalogo Actores



Prueba Catalogo Películas



# Conclusiones y resultados empíricos de las pruebas

Hemos aprendido a trabajar con distintas estructuras de datos, así 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 colecciones que no están ordenadas, y por último 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 últimas 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.

Resultados empíricos

Haciendo pruebas entre distintos equipos hemos visto que hay bastante diferencia entre generaciones de procesadores, así que hemos usado un i7 para las mediciones. El cargar Fichero nos tarda 3,6s, guardar Fichero 6s, la búsqueda de actor por nombre desordenado 76ms, el listar Desordenado 4,7s, el listar Ordenado 6s y la búsqueda de actor por nombre ordenado 84ms. Como hemos comentado estos tiempos no son exactos ya que, al hacerlos desde las pruebas, estas hacen que el tiempo aumente, aunque hemos intentado que sea lo menor posible usando una librería algo más compleja de usar pero que nos ha dado mejores resultados.