Universidad Diego Portales
Escuela de Informática &
Telecomunicaciones

Estructuras de Datos & Análisis de Algoritmos

Laboratorio 3: Algoritmos de ordenamiento y búsqueda en Listas enlazadas

---

Autores: David Fuentes
Xavier Oyarzun
Profesor: Marcos Fantoval

Índice

1. Introducción **2**

2. Implementación y Experimentación **2**
2.1. Implementación 2
2.2. Experimentación 7
2.2.1. Generación de Datos 7
2.3. Benchmarks 8
2.3.1. Medición del tiempo de ordenamiento 8
2.3.2. Medición del tiempo de búsqueda 11

3. Analisis: busqueda lineal e binaria, Counting Sort y Generics en Java **11**
3.1. Busqueda lineal y binaria 11
3.1.1. Comparación entre Búsqueda Lineal y Binaria 11
3.1.2. Implementación de Counting Sort 12
3.1.3. Uso de Generics en Java para estructuras reutilizables 12

4. conclusion y posibles extensiones **13**

1. Introducción

En este laboratorio se creo un sistema de organizacion de juegos basados en su precio, categoria o nivel de calidad y la busqueda eficiente de estos mismos. Para ello, se desarrolló en lenguaje Java el sistema denominado GAME, el cual permite gestionar nombre, categoría, precio y calidad. A continuación, se profundiza en el desarrollo y funcionamiento de este sistema.

2. Implementación y Experimentación

2.1. Implementación

Para empezar esta parte se basa en el codigo que esta github en el siguente link: https://github.com/xavieroyarzun/laboratorio-3-.git

En la figura 1 tenemos la clase Game la y su constructor en el cual tenemos como atributos name, category, price y por último quiality.

```
public class Game {  47 usages
    private String name;  2 usages
    private String category;  2 usages
    private int price;  2 usages
    private int quality;  2 usages

    public Game(String name, String category, int price, int quality) {  6 usages
        this.name = name;
        this.category = category;
        this.price = price;
        this.quality = quality;
    }

    public String getName() { return name; }  4 usages
    public String getCategory() { return category; }  4 usages
    public int getPrice() { return price; }  7 usages
    public int getQuality() { return quality; }  4 usages
```

Figura 1: Clase game

- name: Es nombre del juego establecido
- category: Es la categoria del juego (acción,aventura).
- price: Es el valor del juego, este valor se establece entre 0-70000 (pesos chilenos) en numeros enteros .
- quality: Es una puntuación a la calidad percibida basada en reseñas y evaluaciones.

Ademas de sus get's.

La figura 2 podemos observar la clase Dataset con 2 atributos que son un arreglo llamado data y un string llamado sortedbyAttribute, ademas de sus respectivos constructores.

```
public class Dataset {  2 usages
    private ArrayList<Game> data;  68 usages
    private String sortedByAttribute;  6 usages

    public Dataset(ArrayList<Game> data) {  1 usage
        this.data = new ArrayList<>(data);
        this.sortedByAttribute = null;
    }
```

Figura 2: Clase dataset

- data: Es un arreglo de que guarda objetos tipo Game.
- sortedbyAttribute: Indica por cuál campo está ordenado actualmente el dataset.

A continuación se presentan los metodos de la clase Dataset.

```
public ArrayList<Game> getGamesByPrice(int price) {
        ArrayList<Game> result = new ArrayList<>();
        if ("price".equals(sortedByAttribute)) {
            int cant = binarySearchByPrice(price);
            if (cant != -1) {
                result.add(data.get(cant));
                int left = cant - 1;
                while (left >= 0 && data.get(left).getPrice() == price) {
                    result.add(data.get(left));
                    left--;
                }
                int right = cant + 1;
                while (right < data.size() && data.get(right).getPrice() == price) {
                    result.add(data.get(right));
                    right++;
                }
            }
        } else {
            for (int i = 0; i < data.size(); i++) {
                Game game = data.get(i);
                if (game.getPrice() == price) {
                    result.add(game);
                }
            }
        }
        return result;
    }
```

Figura 3: Clase getgamesbyprice

En este caso, figura 3, el método busca los juegos con el precio solicitado utilizando dos enfoques distintos: emplea búsqueda lineal si el arreglo está desordenado y búsqueda binaria si está ordenado. Los resultados encontrados se almacenan en una lista, que es finalmente retornada como salida del proceso.

En la figura 4, el metodo indaga los juegos que estan dentro de un rango de precios siendo lowerprice es precio menor y higherPrice el precio mayor, añadiendo estos valores a una lista la cual retorna al finalizar.

```java
public ArrayList<Game> getGamesByPriceRange(int lowerPrice, int higherPrice) {  1 usage
    ArrayList<Game> result = new ArrayList<>();
    if ("price".equals(sortedByAttribute)) {
        int start = primero(lowerPrice);
        int end = ultimo(higherPrice);

        if (start != -1 && end != -1) {
            for (int i = start; i <= end; i++) {
                result.add(data.get(i));
            }
        }
    } else {
        for (int i = 0; i < data.size(); i++) {
            Game game = data.get(i);
            int gamePrice = game.getPrice();
            if (gamePrice >= lowerPrice && gamePrice <= higherPrice) {
                result.add(game);
            }
        }
    }
    return result;
}
```

Figura 4: Clase getgamesbyrange

En la imagen 5 se observa un método que retorna una lista con la categoría buscada, utilizando búsqueda binaria si el arreglo está ordenado y búsqueda lineal si está desordenado.

```java
public ArrayList<Game> getGamesByCategory(String category) {
        ArrayList<Game> result = new ArrayList<>();
        if ("category".equals(sortedByAttribute)) {
            int cant = binarySearchByCategory(category);
            if (cant != -1) {
                result.add(data.get(cant));

                int left = cant - 1;
                while (left >= 0 && data.get(left).getCategory().equals(category)) {
                    result.add(data.get(left));
                    left--;
                }

                int right = cant + 1;
                while (right < data.size() && data.get(right).getCategory().equals(category)) {
                    result.add(data.get(right));
                    right++;
                }
            }
        } else {
            for (int i = 0; i < data.size(); i++) {
                Game game = data.get(i);
                if (game.getCategory().equals(category)) {
                    result.add(game);
                }
            }
        }
        return result;
    }
```

Figura 5: Clase getgamesbycatogory

La Figura 6 muestra un algoritmo que filtra los elementos del arreglo según las calidades buscadas. Implementa búsqueda binaria para arreglos ordenados y recurre a búsqueda lineal cuando el arreglo no está ordenado.

```java
public ArrayList<Game> getGamesByQuality(int quality) {
        ArrayList<Game> result = new ArrayList<>();
        if ("quality".equals(sortedByAttribute)) {
            int cant = binarySearchByQuality(quality);
            if (cant != -1) {
                result.add(data.get(cant));

                int left = cant - 1;
                while (left >= 0 && data.get(left).getQuality() == quality) {
                    result.add(data.get(left));
                    left--;
                }

                int right = cant + 1;
                while (right < data.size() && data.get(right).getQuality() == quality) {
                    result.add(data.get(right));
                    right++;
                }
            }
        } else {
            for (int i = 0; i < data.size(); i++) {
                Game game = data.get(i);
                if (game.getQuality() == quality) {
                    result.add(game);
                }
            }
        }
        return result;
    }
```

Figura 6: Clase getgamesbyquality

En este caso, la Figura 7 muestra un método diseñado para ordenar un arreglo según el algoritmo y el atributo especificados.

```
public ArrayList<Game> sortbyalgorithm(String algorithm, String attribute) {
        Comparator<Game> comparator = getComparator(attribute);

        String alg = algorithm.toLowerCase();
        if (alg.equals("bubblesort")) {
            bubbleSort(comparator);
        }
        else if (alg.equals("insertionsort")) {
            insertionSort(comparator);
        }
        else if (alg.equals("selectionsort")) {
            selectionSort(comparator);
        }
        else if (alg.equals("mergesort")) {
            mergeSort(comparator);
        }
        else if (alg.equals("quicksort")) {
            quickSort(comparator);
        }
        else {
            Collections.sort(data, comparator);
        }

        sortedByAttribute = attribute;
        return new ArrayList<>(data);
    }
```

Figura 7: Clase sortbyalgoritm

Estos son los metodos principales para el funcionamiento del codigo, pero para flexibilidad los metodos auxiliares estan en el github https://github.com/xavieroyarzun/laboratorio-3-.git.

2.2. Experimentación

2.2.1. Generación de Datos

En esta sección presentan los datos y las reglas de estos mismos para su posterior uso.

Los datos estan compuestos por juegos los cuales tienen los atributos calidad, precio y categoria, ademas de reglas las cuales son las siguentes.

- Calidad: Un numero aleatorio entre 0 a 100.
- Precio: Un precio (CLP) aleatorio entre 0 a 70000.
- categoria: Son categorias reales de los videojuegos.

2.3. Benchmarks

2.3.1. Medición del tiempo de ordenamiento

En esta parte usan los datos anteriormente dichos para su representación en tablas. Realizado en https://www.jdoodle.com/online-java-compiler

Algoritmo	Tamaño del dataset	Tiempo (milisegundos)
bubbleSort	10^2	1629
bubbleSort	10^4	162900
bubbleSort	10^6	300(seg)+
insertionSort	10^2	1770,6
insertionSort	10^4	177060
insertionSort	10^6	300(seg)+
selectionSort	10^2	1810
selectionSort	10^4	181000
selectionSort	10^6	300(seg)+
mergeSort	10^2	1787,6
mergeSort	10^4	178760
mergeSort	10^6	300(seg)+
quickSort	10^2	2405,6
quickSort	10^4	240560
quickSort	10^6	300(seg)+
collectionsSort	10^2	1812,6
collectionsSort	10^4	181260
collectionsSort	10^6	300(seg)+

Cuadro 1: precio

Algoritmo	Tamaño del dataset	Tiempo (milisegundos)
bubbleSort	10^2	2001,6
bubbleSort	10^4	200160
bubbleSort	10^6	300(seg)+
insertionSort	10^2	1789,6
insertionSort	10^4	178960
insertionSort	10^6	300(seg)+
selectionSort	10^2	2041
selectionSort	10^4	204100
selectionSort	10^6	300(seg)+
mergeSort	10^2	2199,3
mergeSort	10^4	219930
mergeSort	10^6	300(seg)+
quickSort	10^2	1979
quickSort	10^4	197900
quickSort	10^6	300(seg)+
collectionsSort	10^2	1799
collectionsSort	10^4	179900
collectionsSort	10^6	300(seg)+

Cuadro 2: calidad

Algoritmo	Tamaño del dataset	Tiempo (milisegundos)
bubbleSort	10^2	2020
bubbleSort	10^4	202000
bubbleSort	10^6	300(seg)+
insertionSort	10^2	1825,3
insertionSort	10^4	182530
insertionSort	10^6	300(seg)+
selectionSort	10^2	1981,3
selectionSort	10^4	198130
selectionSort	10^6	300(seg)+
mergeSort	10^2	1975,3
mergeSort	10^4	197530
mergeSort	10^6	300(seg)+
quickSort	10^2	2162,6
quickSort	10^4	215260
quickSort	10^6	300(seg)+
collectionsSort	10^2	2405,3
collectionsSort	10^4	240503
collectionsSort	10^6	300(seg)+

Cuadro 3: categoria

2.3.2. Medición del tiempo de búsqueda

Método	Algoritmo	Tiempo (milisegundos)
getGamesByPrice	linearSearch	1889
getGamesByPrice	binarySearch	2322
getGamesByPriceRange	linearSearch	2206,3
getGamesByPriceRange	binarySearch	2407,6
getGamesByCategory	linearSearch	2029,3
getGamesByCategory	binarySearch	1801
getGamesByquality	inearSearch	1923,3(ms)
getGamesByquality	binarySearch	2101,6(ms)

Cuadro 4: Tiempos de ejecución de búsqueda

3. Analisis: busqueda lineal e binaria, Counting Sort y Generics en Java

3.1. Busqueda lineal y binaria

En este espacio se presenta la comparación entre la búsqueda binaria y la lineal, con relación a los datos de forma empírica, así como una reflexión sobre estos dos métodos de búsqueda.

3.1.1. Comparación entre Búsqueda Lineal y Binaria

La principal diferencia en su funcionamiento radica en cómo estos dos métodos realizan la búsqueda. La búsqueda lineal recorre todo el arreglo elemento por elemento hasta encontrar el objetivo, mientras que la búsqueda binaria divide el arreglo en mitades sucesivas, reduciendo el espacio de búsqueda hasta hallar el elemento deseado. En términos de complejidad temporal, la búsqueda lineal tiene un orden de O(n), mientras que la búsqueda binaria opera en O(nlog n), y esto es crucial en relacion con los casos anteriormente vistos, dado que el arreglo debe ordenarse primero para realizar búsquedas binarias, en 3 de 4 casos como se puede observar en los siguente.

- getGamesByPrice — linearSearch — 1889(ms)
- getGamesByPrice — binarySearch — 2322(ms)
- getGamesByPriceRange — linearSearch — 2206,3(ms)
- getGamesByPriceRange — binarySearch — 2407,6(ms)
- getGamesByCategory — linearSearch — 2029,3(ms)
- getGamesByCategory — binarySearch — 1801(ms)
- getGamesByquality — linearSearch — 1923,3(ms)
- getGamesByquality — binarySearch — 2101,6(ms)

Si bien el requisito de ordenamiento previo representa un obstáculo para la búsqueda de elementos, cuando el arreglo ya está ordenado, la búsqueda binaria sería significativamente más rápida. En tales escenarios, la búsqueda lineal solo sería recomendable para casos con arreglos pequeños o desordenados.

3.1.2. Implementación de Counting Sort

La implementación de Counting Sort no presentó mayor dificultad, requiriendo solo la codificación del algoritmo y su integración a la lista 'sortByAlgorithm'. Al aplicarlo para ordenar la calidad de los juegos, se observó un cambio notable en el tiempo de ejecución, como puede apreciarse en el siguiente esquema.

- Counting Sort 1775(ms)
- bubbleSort 1999,3(ms)
- insertionSort 1970(ms)
- selectionSort 2493,6(ms)
- mergeSort 2027(ms)
- quickSort 1996(ms)
- collectionsSort 2493,6(ms)

Para la medición segun el tiempo por el tamaño de valores fueron de 1775(ms),177500(ms) y mas de 300(s)+. Este algoritmo es eficaz cuando se conoce el tamaño del arreglo, pero resulta inaplicable sin esta información previa, como se evidencia en los resultados mostrados.

3.1.3. Uso de Generics en Java para estructuras reutilizables

Para que la clase Dataset funcione con cualquier tipo de objeto, se puede modificar usando generics (Dataset¡T¿), reemplazando Game por el tipo genérico T. Esto permite que la estructura maneje diferentes tipos de datos de forma segura y sin repetir

código. Los generics en Java ofrecen reutilización, verificación de tipos en tiempo de compilación (evitando errores en ejecución) y claridad en el código, mejorando la flexibilidad y mantenibilidad de la clase sin sacrificar robustez

4. conclusion y posibles extensiones

Para finalizar, en este laboratorio se realizó un experimento que consistió en realizar pruebas de ordenamiento con distintos algoritmos y la búsqueda de elementos en arreglos ordenados y desordenados. Se logró ordenar y buscar dichos elementos, lo que resultó en una gran satisfacción.

Para concluir, fue un laboratorio muy interesante, ya que se trabajó con algo bastante cotidiano. Aunque fue un poco repetitivo realizar tantas pruebas con los diferentes algoritmos, fue de gran ayuda para entender mejor lo que son los arreglos, los métodos de ordenamiento y la diferencia entre búsqueda binaria y lineal. Estos conocimientos no solo son relevantes en el ámbito académico, sino también en aplicaciones reales donde la eficiencia en el procesamiento de datos es crucial.