



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Tista García Edgar Ing.

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 5

No de Práctica(s): 4

Integrante(s): Téllez González Jorge Luis

*No. de Equipo de
cómputo empleado:* 37

No. de Lista o Brigada: X

Semestre: 2020-2

Fecha de entrega: 04/03/2020

Observaciones:

CALIFICACIÓN: _____



Índice

1. Introducción	2
2. Objetivos	4
3. Las listas en Java	4
3.1. Operaciones de una lista en Java	6
4. Implementaciones de búsqueda en Java	7
4.1. Búsqueda lineal	7
4.2. Búsqueda binaria	9
5. Listas de objetos	11
5.1. Caso lineal	13
5.2. Caso binario	13
6. Conclusiones	16

La *Búsqueda lineal* representa una de las formas más básicas de concretar un proceso de búsqueda: recorrer una lista de forma sucesiva hasta hallar la clave deseada en alguno de los índices de la colección recibida. Debido a su sencillez de implementación y su relativa 'tosquedad' a la hora de establecer una solución al problema de la búsqueda de claves, este algoritmo es un ejemplo clásico de un algoritmo de *fuerza bruta*.

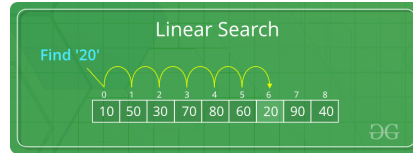


Figura 2: *Búsqueda particionada de la clave deseada en un arreglo.*

Existe otro algoritmo que establece una solución más elegante al problema de la búsqueda: la *Búsqueda binaria*. Sin embargo, es muy importante remarcar que este algoritmo *únicamente* funciona para listas **ordenadas**.

La búsqueda binaria es un algoritmo muy eficiente para encontrar un elemento en una lista ordenada: funciona dividiendo de forma repetitiva a la mitad una porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una. Su gran bondad radica en su complejidad logarítmica de $O(\log(n))$, lo cual le brinda una notable ventaja frente a la complejidad lineal del primer algoritmo.

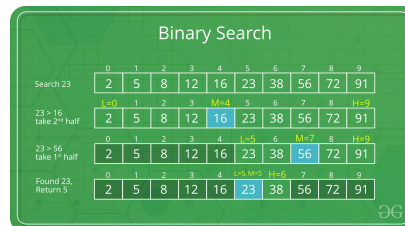


Figura 3: *Búsqueda sucesiva de la clave deseada en un arreglo.*

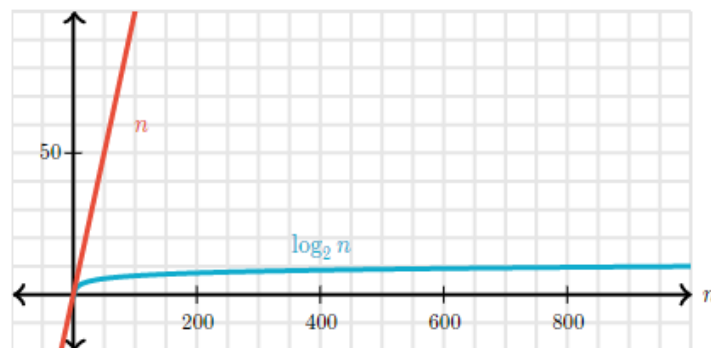


Figura 4: *Comparación de rendimiento entre la búsqueda lineal y binaria.*

2. Objetivos

- El estudiante identificará el comportamiento y características de los principales algoritmos de búsqueda por comparación de llaves.
- El alumno aplicará la búsqueda por comparación de llaves mediante la implementación de listas de tipos de datos primitivos y de tipos de datos abstractos.

3. Las listas en Java

El manejo de las listas en **Java**, se realiza por medio de la interfaz *List* $< E >$ que cuenta con diversas implementaciones. Las más utilizados son las siguientes:

- *ArrayList*: Utiliza arreglos dinámicos, es decir, que cambian de tamaño total conforme se introducen elementos en el arreglo.
- *LinkedList*: Se implementa la ED *Lista ligada doble*.

A continuación, se crea un proyecto en **NetBeans** que implementa el uso de listas.

La clase principal contiene los siguientes métodos:

1. El método **main** inicia creando una *LinkedList* de wrappers *Integers* llamada **lista1**. A continuación, se agregan elementos a la lista usando el método **add** y finalmente se imprimen los elementos presentes en la lista creada.

```
public static void main(String[] args) {  
    List<Integer> lista1 = new LinkedList<>();  
  
    lista1.add(15);  
  
    lista1.add(12);  
    lista1.add(20);  
    lista1.add(45);  
    lista1.add(96);  
    lista1.add(74);  
    lista1.add(80);  
}
```

Figura 5: Llenado de la lista de enteros.

Posteriormente, se vuelven a agregar enteros a la lista. Sin embargo, en esta ocasión se envían dos parámetros: uno que indique el índice donde se desea colocar el elemento y el elemento a insertar en cuestión.

```
lista1.add(1,300);  
lista1.add(3,500);  
lista1.add(5,700);
```

Figura 6: Método **add** sobrecargado con 2 parámetros.

Las siguientes líneas hacen uso del método **set** con el fin de reemplazar el elemento presente en cierto índice de la lista (indicado como primer parámetro) con otro entero especificado como segundo parámetro.

```
lista1.set(0, 4);  
lista1.set(2, 6);  
lista1.set(7, 8);
```

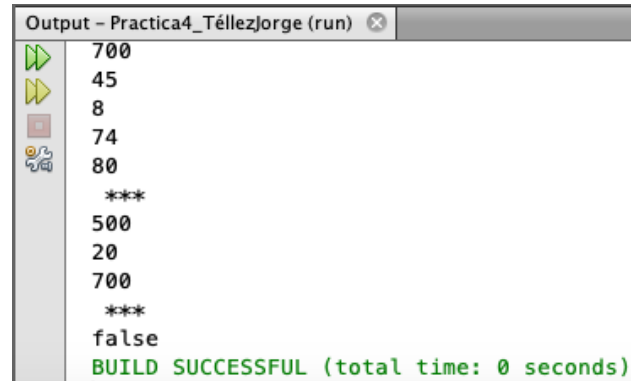
Figura 7: El método **set** reemplaza un elemento en cierto índice con un valor especificado.

Finalmente, se crea otra lista del mismo tipo denominada *lista2* y, después, a esta lista se le asignan los elementos presentes en los índices 3 y 6 de la *lista1* y se imprime el contenido de la nueva *sublista*.

```
List<Integer> lista2;  
lista2 = lista1.subList(3, 6);  
imprimirLista(lista2);  
System.out.println(" *** ");  
System.out.println(lista1.equals(lista2));
```

Figura 8: Sublista creada a partir de la *lista1*.

El método **equals** verifica si 2 listas del mismo tipo tienen exactamente el mismo tamaño e idénticos elementos; teniendo como salida un valor booleano **true** o **false**. En este caso particular, se imprime la salida de comparación entre la *lista1* y la *lista2*, la cual evidentemente resulta falsa.



```

Output - Practica4_TéllezJorge (run)
700
45
8
74
80
***
500
20
700
***
false
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figura 9: Salida *false* de *equals*: simplemente indica que las listas son diferentes.

2. El método **imprimirLista** únicamente recibe como parámetro una lista de tipo *Integer*. Iterando en ella por medio de un ciclo **for-each**, se imprimen los elementos presentes en la lista recibida.

```

public static void imprimirLista(List<Integer> listaPrint){
    for(Integer var : listaPrint){
        System.out.println(var);
    }
}

```

Figura 10: Estructura general del método *imprimirLista*.

3.1. Operaciones de una lista en Java

Java posee métodos implementados para realizar las operaciones básicas definidas en una lista:

- **Borrar:** El método **remove()** elimina un elemento de un *ArrayList*; teniendo como parámetro el índice del elemento a eliminar. Por medio de sobrecarga, también acepta como parámetro un determinado objeto a remover.

```

Se eliminó el elemento en la posición: 2
Sublista actual
***
500
20
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figura 11: Instrucciones añadidas para eliminar el número 700 de la sublista.

```
lista2.remove(posicion);
System.out.println("Se eliminó el elemento en la posición: "+posicion);
System.out.println("Sublista actual");
System.out.println(" *** ");
imprimirLista(lista2);
```

Figura 12: Impresión del índice del elemento eliminado y la lista.

- **Verificación:** El método `isEmpty()` de `ArrayList` retorna un **true** si la lista a la que le fue aplicada el método no contiene elementos. Caso contrario, retorna un **false**.

```
System.out.println("La sublista está vacía?");
boolean res=lista2.isEmpty();
System.out.println("Resultado: " +res);
```

Figura 13: Líneas añadidas para verificar si la lista está vacía.

```
La sublista está vacía?
Resultado: false
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 14: El resultado anterior evidentemente es **false**.

- **Búsqueda:** El método `indexOf()` de `ArrayList` regresa el primer índice donde se encuentra el elemento especificado en la lista. En caso de no encontrarlo, retorna un -1 .

```
int posicion=lista2.indexOf(700);
System.out.println("Posicion del 700 en la sublista: "+posicion);
```

Figura 15: Instrucciones añadidas para encontrar el índice del número 700 en la sublista.

```
Posicion del 700 en la sublista: 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 16: Impresión del índice buscado anteriormente.

4. Implementaciones de búsqueda en Java

En el proyecto anterior se han agregado dos clases que corresponden a las implementaciones de *Búsqueda Lineal* y *Búsqueda Binaria* usando Java, las cuales cubren las operaciones básicas de búsqueda e incidencia de claves en una lista de objetos de tipo **Integer**; envoltorios especializados para números enteros.

4.1. Búsqueda lineal

La clase *BusquedaLineal* contiene en su interior tres métodos que implementan 3 variantes distintas de la búsqueda lineal:

- El método **busquedaIncidencia** recibe una lista de envoltorios **Integer** y una clave, que corresponde al elemento a buscar. Por medio de un contador y un ciclo *for-each* que itera sobre la lista recibida, se contabilizan las apariciones de la clave ingresada como parámetro de búsqueda. Finalmente, se regresa a la variable contadora para recuperar su valor almacenado que corresponde al número total de apariciones de la clave en la lista.

```
int contadorClave = 0;
for (Integer i : lista) {
    if (clave == i) {
        contadorClave++;
    }
}
return contadorClave;
```

Figura 17: Estructura de *busquedaIncidencia*.

- La implementación **busquedaIndice** recibe los mismos parámetros que el método anterior. En comparación, este devuelve la primera posición en la que se encuentra la clave introducida como parámetro de búsqueda en la lista de **Integers**. En caso de no encontrar nada, devolverá un -1 ; indicando que la clave buscada no se encontró en la lista.

```
for (int i = 0; i < lista.size(); i++) {
    if (clave == lista.get(i)) {
        return i;
    }
}
return -1;
```

Figura 18: Condición principal de la búsqueda lineal.

- La implementación de **busquedaCentinela** hace uso de una variable intermedia que toma el valor final de una lista y, por medio de un ciclo **while**, se comienza a contar la incidencia de la clave sobre la lista. Una vez que se imprime la incidencia, la función devuelve un *true* si el valor fue hallado e incrementó al contador. En caso contrario, se retorna un *false*.

```
int contador = 0;
while (lista.get(contador) != clave) {
    contador++;
}
System.out.println(contador);
lista.set(lista.size() - 1, centinela);
if (contador < lista.size() - 1 || list
    return true;
} else {
    return false;
```

Figura 19: El valor retornado es un boolean que indica si tal clave existe en la lista.

A continuación, se muestra la llamada a los métodos de búsqueda lineal para verificar su correcta implementación:

```
imprimirLista(lista2);
System.out.println("El 600 se encuentra en la lista?: " + BusquedaLineal.busquedaCentinela(lista2, 600));
System.out.println("El valor 700 tiene el indice: " + BusquedaLineal.busquedaIndice(lista2, 700));
System.out.println("¿Cuántas veces se encuentra el 20?: " + BusquedaLineal.busquedaIncidencia(lista2, 20));
```

Figura 20: Líneas de código añadidas.

```
500
20
700
El 600 se encuentra en la lista?: false
El valor 700 tiene el indice: 2
¿Cuántas veces se encuentra el 20?:1
```

Figura 21: Salida del código anterior.

4.2. Búsqueda binaria

La clase *Busqueda Binaria* contiene en su interior dos métodos que realizan la búsqueda de la primer incidencia de la clave buscada y la incidencia misma de la clave en toda la lista:

- El primer método **busquedaValor** recibe una lista, la clave a buscar, así como el intervalo de inicio y fin de la lista recibida. Esta implementación es recursiva y, por tanto, se considera el caso base previo a realizar cualquier operación.

Una vez que se verifique que tal caso no ocurre, se obtiene el índice medio redondeado y convertido por medio de *casting* a un entero. Hecho esto se verifica: ¿La clave se encuentra en el índice medio de la lista? De ser cierto, el proceso termina devolviendo *true*. Caso contrario, se pregunta si la clave es mayor o menor al valor buscado, retornando el mismo método con el intervalo de búsqueda modificado para buscar únicamente en un sub-intervalo de la lista.

```
if (inicio > fin) {  
    return false;  
}
```

Figura 22: Caso base del proceso recursivo.

```
if (lista.get(mitad) < clave) {  
    return busquedaValor(lista, clave, mitad + 1, fin);  
} else {  
    return busquedaValor(lista, clave, inicio, mitad - 1);  
}
```

Figura 23: Intervalos de búsqueda modificados.

- El método **busquedaIncidencia** aplica un enfoque iterativo, a diferencia del método anterior. Mientras la lista sea de tamaño superior a 1, se obtiene el índice medio truncado y convertido a entero por *casting*.

Si se cumple que el elemento medio de la lista corresponde a la clave buscada, se usa una variable *auxiliar* a la que se le asigna el índice medio. Por medio de los ciclos **while** implementados, se recorre la lista por ambas partes, incrementado el valor del contador si se encuentra repetida la clave deseada en la lista. Finalmente, se regresa el contador con el valor numérico de la incidencia de la clave deseada en la lista.

```
if (lista.get(mitad) == clave) {  
    int auxiliar = mitad;  
  
    while (Objects.equals(lista.get(auxiliar), lista.get(auxiliar + 1)))  
        auxiliar++;  
    contador++;  
}  
  
while (Objects.equals(lista.get(auxiliar), lista.get(auxiliar - 1)))  
    auxiliar--;  
contador++;  
}  
return contador;
```

Figura 24: Contador de incidencia.

En caso de que la búsqueda haya resultado negativa, se modifican los parámetros de *inicio* o *fin* dependiendo si la clave es mayor al elemento ubicado en el índice medio de la lista. De esta forma, se garantiza que la complejidad se encuentre en el orden deseado y no se realicen operaciones innecesarias.

```
else {  
  
    if (lista.get(mitad) < clave) {  
  
        inicio = mitad + 1;  
    } else {  
  
        fin = mitad - 1;  
    }  
}
```

Figura 25: *Modificación iterativa de los intervalos de búsqueda.*

Finalmente, se ponen a prueba los métodos mediante la modificación de la lista creada al inicio:

```
lista2.set(0, 200);  
lista2.set(1, 500);  
lista2.add(500);  
lista2.add(700);  
lista2.add(900);  
imprimirLista(lista2);
```

Figura 26: *Lista de Integers modificada.*

```
; "+BusquedaBinaria.busquedaValor(lista2, 500, 0, lista2.size()));  
00 en la lista?: " +BusquedaBinaria.busquedaIncidencia(lista2, 500, 0, lista2.size()));
```

Figura 27: *Llamada a los métodos de búsqueda binaria.*

```
200  
500  
500  
700  
900  
índice de la clave: 2  
El 500 se encuentra en la lista?: true  
Se ha contabilizado otro valor  
¿Cuántas veces se encuentra el 500 en la lista?: 2
```

Figura 28: *Salida obtenida para la clave 500.*

5. Listas de objetos

Las implementaciones analizadas previamente pueden ser utilizadas para realizar búsquedas sobre listas con diferentes tipos de objetos, por ejemplo... ¡Una lista de perros! A continuación, se muestra la clase *Perro*,

la cuál modela las características esenciales de un perro común de toda la vida: su nombre, raza y su color de pelo.

```
public class Perro {  
  
    private String nombreDelPerro;  
    private String razaDelPerro;  
    private String colorDePelaje;  
  
    public Perro(String nombreDelPerro, String razaDelPerro, String colorDePelaje){  
        this.nombreDelPerro=nombreDelPerro;  
        this.razaDelPerro=razaDelPerro;  
        this.colorDePelaje=colorDePelaje;  
    }  
}
```

Figura 29: Atributos privados de la clase *Perro* y su respectivo constructor.

La clase *Perro* fue creada con el concepto de encapsulamiento en mente. Por tanto, fueron incluidos los métodos *get* para cada uno de los atributos, con el fin de recuperarlos y utilizarlos posteriormente. Así mismo, fueron incluidos dos métodos más: **ladrar** y **pelota**. Estos muestran mensajes en pantalla relacionados con los ladridos de un perro dado y con el objeto *Perro* 'corriendo' para ir por una pelota que le fue lanzada.

```
public static void ladrar(List<Perro> lista, int clave){  
    String nombrePerro=lista.get(clave).getNombre();  
    System.out.println("El perro " +nombrePerro+ " tiene unas palabras  
    System.out.println(";Guau!");  
}
```

Figura 30: Método **ladrar** de la clase *Perro*.

```
public String getNombre(){  
    return nombreDelPerro;  
}  
  
public String getRaza(){  
    return razaDelPerro;  
}
```

Figura 31: Métodos **get** establecidos para los atributos.

```
lomitos.add(new Perro("Johnny", "Mezcla", "Negro")); //Posicion 0
lomitos.add(new Perro("Cookie", "Pool", "Cafe")); //Posicion 1.
lomitos.add(new Perro("Rocky", "Doberman", "Cafe")); //Posicion 2.
lomitos.add(new Perro("Jake", "Pug", "Blanco")); //Posicion 3.
```

Figura 32: Lista de perros creada en el método main de la clase principal.

5.1. Caso lineal

Con el fin de adaptar las implementaciones realizadas al tipo de dato *Perro*, se modificó el parámetro de entrada del método **busquedaIndice** con el fin de especificar la entrada de una lista de 'perros' y la clave de búsqueda. Se crearon dos métodos separados para especificar la búsqueda deseada (Por *Nombre* o por *Raza*); en ambos casos se especifica la frecuencia de aparición de la clave ingresada como parámetro:

```
for (int i = 0; i < lista.size(); i++) {
    if (clave == lista.get(i).getNombre()) {
        int contadorClave = 0;
        for (int j = 0; j < lista.size(); j++) {
            if (clave == lista.get(j).getNombre()) {
                contadorClave++;
            }
        }
        System.out.println("Frecuencia de aparición: " + contadorClave);
        return i;
    }
}
```

Figura 33: Búsqueda lineal por Nombre de un objeto de tipo *Perro*.

```
System.out.println("Donde se encuentra Jake?");
System.out.println("Indice de Jake: " + BusquedaLineal.busquedaLinealNombre(lomitos, "Jake"));
```

Figura 34: Uso del método de búsqueda lineal por nombre.

```
Donde se encuentra Jake?
Frecuencia de aparición: 1
Indice de Jake: 3
```

Figura 35: Salida obtenida del código anterior.

5.2. Caso binario

Con el fin de facilitar la tarea, se eligió utilizar la variante iterativa del algoritmo de búsqueda binaria. La adaptación del algoritmo a la búsqueda de objetos toma en cuenta una mayor cantidad de factores con respecto a la búsqueda lineal:

- El proceso de comparación y verificación de la clave consiste en observar en primer lugar si el primer carácter de la clave coincide con el primer carácter de la clave introducida como parámetro de

búsqueda. En caso de cumplirse la condición, se utiliza una variable auxiliar a la que se le asigna el índice medio de la lista. Posteriormente, se verifica si la clave del elemento medio de la lista coincide idénticamente con la clave buscada. De cumplirse lo anterior, la función devuelve el índice *medio*.

```
int mitad = (int) (Math.floor(inicio + fin)/2);

if ((int) lista.get(mitad).getNombre().charAt(0) == (int) clave.charAt(0)) {
    int auxiliar = mitad;

    if (clave.equals(lista.get(mitad).getNombre())) {
        return mitad;
    }
}
```

Figura 36: Consideración básica de la búsqueda binaria en objetos.

En caso de no cumplirse lo anterior, el método procederá a recorrer por medio de la variable *auxiliar* la lista de izquierda a derecha, verificando aquellos elementos con coincidencia en la primer letra. Finalmente, verificará si son idénticos para devolver como índice la variable *auxiliar*.

```
auxiliar++;
while (clave != lista.get(auxiliar).getNombre() && (int)
    auxiliar++);
}
if (clave.equals(lista.get(auxiliar).getNombre())) {
    return auxiliar;
}
```

Figura 37: Recorrido hacia la derecha.

Por último, en el caso de que el elemento en su letra inicial sea diferente a la inicial de la clave buscada, se realiza una comparación del valor ASCII de la primera letra del nombre del elemento medio con la primera letra de la clave. Con el fin de realizar la verificación ASCII, se utiliza un *cast* para transformar los datos de comparación en números. En el caso de que el valor buscado sea menor, la búsqueda será reducida al sub-arreglo de rango $[inicio, mitad - 1]$. Caso contrario, el nuevo intervalo será $[mitad + 1, fin]$.

```
else {

    if ((int) lista.get(mitad).getNombre().charAt(0) < (int) clave.charAt(0)) {
        inicio = mitad + 1;
    }
    else
        fin = mitad - 1;
}
```

Figura 38: Cambio de los intervalos de búsqueda binaria en objetos.

Con el fin de poner a prueba la validez de las implementaciones de búsqueda binaria para ambos parámetros, se utilizó una versión modificada y ordenada de la lista de perros creada previamente. Así mismo, para obtener la frecuencia de aparición, se implementó un ciclo **for** y una variable contadora, como en el caso de búsqueda lineal, en cada caso donde se regresa un índice de coincidencia:

```
if ((int) lista.get(mitad).getNombre().charAt(0) == (int) clave.charAt(0)) {
    int auxiliar = mitad;

    if (clave.equals(lista.get(mitad).getNombre())) {
        int contadorClave = 0;
        for (int j = 0; j < lista.size(); j++) {
            if (clave == lista.get(j).getNombre()) {
                contadorClave++;
            }
        }
        System.out.println("Frecuencia de aparición: " + contadorClave);

        return mitad;
    }
}
```

Figura 39: Contador de frecuencia en la búsqueda binaria.

```
lomitos.add(new Perro("Arnold", "Boxer", "Cafe")); //Posicion 0.
lomitos.add(new Perro("Cookie", "Cavalier", "Cafe")); //Posicion 1.
lomitos.add(new Perro("Iggy", "Doberman", "Negro")); //Posicion 2.
lomitos.add(new Perro("Jake", "Pug", "Negro")); //Posicion 3.
lomitos.add(new Perro("Kike", "Mastin", "Negro")); //Posicion 4.
lomitos.add(new Perro("Iggy", "Doberman", "Negro"));
```

Figura 40: Nueva lista de perros.

```
Perro.pelota(lomitos, 3);
System.out.println("Jake se emociona!");
Perro.ladraz(lomitos, 3);
System.out.println("Olvide la raza de Cookie...");
System.out.println("En que posicion estara?: " + BusquedaBinaria.busquedaBinariaRaza(lomitos, "Cavalier", 0, lomitos.size()));
System.out.println("Un momento...");
System.out.println("¿Y a donde se fue Iggy?: " + BusquedaBinaria.busquedaBinariaNombre(lomitos, "Iggy", 0, lomitos.size()));
```

Figura 41: Líneas de código en el main de la clase principal para probar los métodos implementados.

```
¡El perro Jake se ha ido corriendo por la pelota!
¡Wow! *Se ve muy emocionado*
Jake se emociona!
El perro Jake tiene unas palabras para tí:
¡Guau!
Olvide la raza de Cookie...
Frecuencia de aparición: 1
En que posicion estara?: 1
Un momento...
Frecuencia de aparición: 2
¿Y a donde se fue Iggy?: 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figura 42: Salida obtenidas con las líneas anteriores.

6. Conclusiones

El trabajo desarrollado en la presente práctica demostró ser una actividad retadora intelectualmente. Durante el desarrollo de mis implementaciones me enfrenté a dificultades relacionadas a pequeños detalles que, en un inicio, pasaron desapercibidos:

- Escritura de los *cast*: los paréntesis fueron responsables de muchos errores asociados al índice intermedio utilizado en los métodos de la *Búsqueda binaria*.
- Vigilancia de los índices de trabajo en los métodos: una gran dificultad que tuve que resolver fue arreglar los múltiples errores obtenidos debido a un incorrecto manejo interno de los índices.
- Validación de datos de tipo String: esta práctica requirió un uso más generalizado y cuidadoso del *casting* de cadenas de caracteres a su respectiva conversión en ASCII con el fin de realizar las comparaciones correspondientes.

El poder llegar a implementaciones funcionales requirió una fuerte tarea de depuración en el código que implementé, lo cual tomó una buena parte del tiempo de desarrollo inicial. Una vez que se lograron resolver tales dificultades, los problemas fueron menores y se logró llegar a uno de los objetivos principales planteados al inicio de la presente práctica: La programación de métodos de búsqueda lineal y binaria para *Integers* y *Datos abstractos*.

Debido al tamaño de las listas empleadas, la diferencia de rendimiento entre ambos algoritmos de búsqueda no fue apreciable en absoluto. Sin embargo, considerando la complejidad computacional de la *Búsqueda binaria* de $O(\log(n))$ en comparación con su contraparte lineal, resulta evidente el mejor desempeño de la búsqueda binaria en listas de un tamaño considerable.

El paradigma orientado a objetos no es nuevo para mí: el lenguaje con el cuál me inicié en el mundo de la programación fue Java. En su momento fue una actividad tremendamente complicada y frustrante debido a las nulas nociones de programación estructurada que poseía en ese momento. Sin embargo, con el tiempo, comencé a entender de mejor forma al lenguaje y sus conceptos; aunque a un nivel muy básico. Tras iniciar en el mundo de C en *Fundamentos de Programación*, las lagunas que tenía respecto a la programación estructurada fueron solventadas, así mismo, mi capacidad de comprensión y análisis fue solidificada en *EDA I*. Finalmente, mi regreso a Java se encuentra marcado por una relativa facilidad para maniobrar con el lenguaje; a diferencia de mis primeros pasos marcados por constantes tropiezos y frustraciones.

A pesar de eso, considero que todavía me hace falta mucho por aprender y mejorar, por lo que considero que el uso aplicado de Java en el curso es la oportunidad perfecta para consolidar mi formación en el lenguaje así como mi formación de Ingeniero en Computación.



Referencias

- [1] Búsqueda binaria. Recuperado de: <https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search>. Fecha de consulta: 22/02/2020.
- [2] Linear Search. Recuperado de: <https://www.geeksforgeeks.org/linear-search/>. Fecha de consulta: 22/02/2020.
- [3] Linear Search vs Binary Search. Recuperado de: <https://www.geeksforgeeks.org/linear-search-vs-binary-search/>. Fecha de consulta: 22/02/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©