



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS II

Grupo: 5

Número de Práctica(s): Guía práctica de estudio 4: Algoritmos de Búsqueda I

Integrante(s): MURRIETA VILLEGAS ALFONSO

*Núm. De Equipo de
cómputo empleado :* 38

Semestre : 2019 - 1

Fecha de entrega: 4 DE SEPTIEMBRE DE 2018

Observaciones:

CALIFICACIÓN: _____

ALGORITMOS DE BÚSQUEDA. PARTE I

INTRODUCCIÓN

Recuperar información de una computadora es una de las actividades más útiles e importantes, es muy comúnmente se tiene un nombre o llave que se quiere encontrar en una estructura de datos tales como una lista u arreglo.

Al proceso de encontrar un dato específico llamado “CLAVE” en alguna estructura de datos, se denomina búsqueda. Este proceso termina exitosamente cuando se localiza el elemento que contiene la llave buscada, o termina sin éxito cuando no aparece ningún elemento con esa llave.

Clasificación de algoritmos de ordenamientos

Los algoritmos de búsqueda se pueden clasificar en:

- 1) Algoritmos de comparación
- 2) Algoritmos de transformación de llaves

OBJETIVOS DE LA PRÁCTICA

- El estudiante identificará el comportamiento y características de algún algoritmo de búsqueda por comparación de llaves.

DESARROLLO

Actividad 1. Análisis del manual de prácticas

Es esta práctica los 2 métodos que se abarcarán serán Búsqueda lineal y Búsqueda Binaria.

1.1 Búsqueda Lineal

BÚSQUEDA LINEAL

Este algoritmo empieza con la búsqueda desde la primera posición de la lista, comparando el elemento con la llave, en caso de no ser continúa verificando las siguientes posiciones hasta encontrarlo o terminar con toda la estructura de datos. Es de esta forma que podemos encontrar a el elemento ya sea en la primera, última o en medio de la estructura donde se ha guardado.

Ejemplificación

```
BúsquedaLineal (A,n,x)
Inicio
    encontrado=-1
    Para k=0 hasta n-1
        Si x==A[k]
            encontrado = k
        Fin Si
    Fin Para
    Retorna encontrado
Fin
```

Imagen 1: Pseudocódigo del Algoritmo de búsqueda “Búsqueda Lineal”

Análisis de complejidad

Como se puede observar en el pseudocódigo el número de iteraciones no cambia a lo largo de toda la ejecución del algoritmo por lo que inmediatamente podemos notar que este algoritmo de búsqueda a pesar de ser fácil de programar realmente nos da una complejidad lineal $O(n)$ al momento de buscar algún elemento.

BÚSQUEDA LINEAL MEJORADA

Sin embargo, una de las mejoras que se le puede hacer a este algoritmo es por ejemplo no esperar a que revise todos los elementos para ello se puede utilizar un elemento auxiliar donde a través de la comparación de este elemento es cómo es posible el reducir se existe o no el elemento en la estructura. A continuación, el pseudocódigo del algoritmo:

Ejemplificación

```
BúsquedaLinealMejorado
Inicio
    encontrado=-1
    Para k=0 hasta n-1
        Si A[k]==x
            encontrado= k
            Salir de la estructura de repetición
        Fin Si
    Fin Para
    retorna encontrado
Fin
```

Imagen 2: Pseudocódigo del Algoritmo de búsqueda mejorado “Búsqueda Lineal”

BÚSQUEDA SECUENCIAL

Es aquella que asegura encontrar el elemento a buscar a través de un “centinela” el cual a través de la búsqueda del final del arreglo o lista va guardando los elementos en el paso, de esa forma es como se va revisando todos los elementos y además se obtiene un índice K empleado en la versión mencionada anteriormente. A continuación, el pseudocódigo del algoritmo:

```
BusquedaLinealCentinela(A,n,x )
tmp=A[n-1]
A[n-1]=x
k=0
Mientras A[k] sea diferente de x
    k=k+1
Fin Mientras

A[n-1]=tmp

Si k < n-1 o A[n-1]==x
    retorna k
En otro caso
    retorna -1
Fin Si
Fin
```

Imagen 3: Pseudocódigo del Algoritmo de búsqueda “Búsqueda Secuencial”

NOTA: es necesario mencionar que en este caso la complejidad de búsqueda puede en el mejor caso mejorar debido a que sería del tipo $O(1)$ mientras que en el peor caso simplemente regresaríamos a $O(n)$.

BÚSQUEDA LINEAL (VERSIÓN RECURSIVA)

En la versión recursiva se tiene como caso base el caso donde se da el fracaso de la búsqueda el cual es directamente cuando se rebasa el número de elementos de la lista al momento de hacer la búsqueda y un caso de éxito cuando, después de utilizar la recursividad cierta cantidad de veces es cuando se encuentra el elemento que se busca. A continuación, el pseudocódigo del algoritmo:

```
BusquedaLinealRecursiva (A,x,ini,fin)
Inicio
  Si ini>fin
    encontrado=-1
  Si no
    Si A[ini]==x
      encontrado=ini
    Si no
      encontrado = BusquedaLinealRecursiva(A, x,ini+1,fin)
  Fin Si no
  Fin Si no
  retorna encontrado
Fin
```

Imagen 4: Pseudocódigo del Algoritmo de búsqueda “Búsqueda Lineal” en su forma recursiva

Actividad (Python)

Para esta actividad se proporcionó la versión recursiva e iterativa de búsqueda binaria, a continuación, se muestra la salida de ambos programas:

```
In [9]: runfile('C:/Users/pon_c/OneDrive/Documentos/Universidad/3° Semestre/EDA II/
Prácticas_LAB/4_practice/Códigos Manual/busquedalineal.py', wdir='C:/Users/pon_c/
OneDrive/Documentos/Universidad/3° Semestre/EDA II/Prácticas_LAB/4_practice/Códigos
Manual')
A continuación se muestra la salida de todas las versiones
Elemento a buscar:
4
Posición del elemento:
2
```

Imagen 5: Búsqueda de un elemento a través de “Búsqueda lineal”, salida del programa.

1.2 Búsqueda Binaria

BÚSQUEDA BINARIA

También conocido como dicotómica, es eficiente para encontrar un elemento en una lista ordenada ya ordenada y además es un ejemplo de divide y vencerás. Para poder demostrar como se lleva este tipo de búsqueda podemos hacer la analogía con un diccionario donde la primera letra es la que nos reduce enormemente la búsqueda, sin embargo, en este caso a pesar de reducir la búsqueda aún tenemos dos casos, uno es donde casi está al inicio el elemento y el otro es donde está hasta el otro extremo de donde se busca.

Dicho de otra forma, este algoritmo se encarga de partir el arreglo o lista inicial en 2 partes y así sucesivamente hasta llevar a listas o arreglos de un solo elemento. La estrategia consiste sobre todo en comparar la llave o elemento a buscar con las sub-listas. Con base a lo mencionado anteriormente, la forma en que podría verse la lista una vez partida podría ser la siguiente (Ver imagen 5)

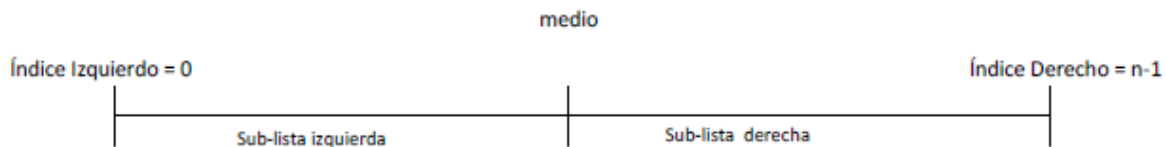


Imagen 6: Lista partida para la búsqueda de un elemento a través del algoritmo de búsqueda “Búsqueda Binaria”

Una vez que se determina la mitad del arreglo o lista, se compara la mitad con el valor que se está buscando si son iguales entonces ahí termina la búsqueda, en caso contrario nuevamente se realiza la misma lógica para ambos sub-arreglos.

NOTA: Es importante mencionar que tanto en el sub-arreglo izquierdo como derecho se realizan comparaciones de valor para poder limitar o mejor dicho acotar la búsqueda del elemento.

Por último, en dado caso que las sub-listas lleguen a tener dimensión 0 significa que el elemento que se quería buscar no estaba disponible en la lista o arreglo original. A continuación, el pseudocódigo del algoritmo:

```

BusquedaBinariaIterativa(A,x,indiceIzq,indiceDer)
Inicio
    Encontrado= -1
    Mientras indiceIzq <= indiceDer
        Medio=[(indiceIzq + IndiceDer)/2]
        Si x == A[Medio] entonces
            Encontrado=Medio
        Si no
            Si A[medio] < x
                indiceIzq=medio +1
            Si no
                indiceDer=medio-1
            Fin Si no
        Fin Si no
    Fin Mientras
    Retorna Encontrado
Fin
    
```

Imagen 7: Algoritmo de búsqueda “Búsqueda Binaria”.

BÚSQUEDA BINARIA (Versión Recursiva)

Este algoritmo también tiene un enfoque recursivo, donde el caso base es cuando se ha pasado de la cantidad de elementos contenidos en el arreglo o lista, mientras que en el caso exitosos es aquel que después de llamar algunas veces a la función a través de la recursividad. A continuación, el pseudocódigo del algoritmo:

```

BusquedaBinariaRecursiva(A,x,indiceIzq,indiceDer)
Inicio
    Si indiceIzquierdo > indiceDerecho y x es diferente de A[indiceDerecho]
        Retorna -1
    Fin Si
    medio = [(indiceIzq + IndiceDer)/2]
    Si x==A[medio]
        Retorna medio
    En otro caso
        Si x < A[medio]
            Retorna BusquedaBinariaRecursiva(A,x,indiceIzquierdo,medio)
        En otro caso
            Retorna BusquedaBinariaRecursiva(A,x,medio, indiceDerecho)
        Fin Si
    Fin Si
Fin

```

Imagen 8: Algoritmo de búsqueda “Búsqueda Binaria”. Versión Recursiva

Análisis de Complejidad

Tanto la versión iterativa como la versión recursiva retornan la posición donde se encuentra el elemento o bien x -1 si es el caso de no encontrarse en el arreglo o lista.

Debido a esto es que el tiempo de ejecución en ambos casos es “ $O(\log(n))$ ”. El cual resulta mucho mejor respecto a la búsqueda lineal.

Actividad (Python)

Para esta actividad se proporcionaron todos los códigos de las distintas versiones de búsqueda Lineal, a continuación, se muestran las salidas de los programas.

```

In [14]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/4_practice/
Códigos Manual/busquedaBinaria_Iterativa.py', wdir='C:/
Users/pon_c/OneDrive/Documentos/Universidad/3° Semestre/
EDA II/Prácticas_LAB/4_practice/Códigos Manual')
Elemento a buscar:
21
Posición del elemento:
1

```

Imagen 9: Algoritmo de búsqueda “Búsqueda Binaria”. Versión Iterativa

```

In [15]: runfile('C:/Users/pon_c/OneDrive/Documentos/
Universidad/3° Semestre/EDA II/Prácticas_LAB/4_practice/
Códigos Manual/busquedaBinaria_Recursiva.py', wdir='C:/
Users/pon_c/OneDrive/Documentos/Universidad/3° Semestre/
EDA II/Prácticas_LAB/4_practice/Códigos Manual')
3
Elemento a buscar:
40
Posición del elemento:
3

```

Imagen 10: Algoritmo de búsqueda “Búsqueda Binaria”. Versión Recursiva

Actividad 2. Ejercicios de laboratorio

2.1 Lista en Java

Para este problema se proporcionó un código previamente programado en Java donde se tiene como propósito general aplicar y conocer tanto los conocimientos de EDA II como los conocimientos de POO.

a) Diferencia del método “set” y “add”

Los métodos set y add son los relacionados con las listas en Java, a continuación, se explica a detalle cada uno de los métodos:

ADD

El método add es aquel que agrega elementos en una lista conforme se van ingresando los datos, cabe destacar que además en Java se le pueden dar ya sea solamente el valor que se quiere ingresar o también la posición donde se quiere agregar y el valor que se quiere agregar.

```
Estado punto 1:
15
16
17
18
19
20
80
***
```

Imagen 11: Salida de la lista solamente utilizando add

NOTA: Es importante destacar que en caso de que la posición donde se quiera agregar un elemento ya esté ocupada, simplemente el elemento que se encuentra ahí lo mueve a la siguiente posición, o que significa que realmente no se pierde ninguno de los valores.

```
Estado punto 2:
15
300
16
500
17
700
18
19
20
80
```

Imagen 12: Salida de la lista solamente utilizando add

Como se puede ver en la imagen 12 al utilizar el método add en la lista inicial (imagen 11) lo único que hace es recorrer los elementos, debido a que se están agregando elementos nuevos.

SET

El método set es aquel que agrega elementos en una lista conforme se van ingresando los datos, cabe destacar que además en Java se le pueden dar ya sea solamente el valor que se quiere ingresar o también la posición donde se quiere agregar y el valor que se quiere agregar.

La diferencia respecto al método add es que en caso de que en la posición donde se quiere insertar un elemento ya está previamente ocupada, lo que se realiza es la eliminación del elemento previo para posteriormente insertar el elemento que se está mandando.

```
Estado punto 3:
4
300
6
500
17
700
18
8
20
80
```

Imagen 13: Salida de la lista una vez utilizado el método set

b) Método “sublist”

El método sublist es aquel encargado de como dice su nombre crea una sublista a partir de una lista inicial, los parámetros que se le mandan a este método son tanto el índice donde empezará la sublista como el índice donde terminará la sublista. Como se puede ver en la siguiente línea

```
lista2= lista1.subList(3, 6);
```

la **lista2** es la lista donde se guardarán los datos acotados por el método **sublist**, por otro lado, **lista1** es la lista inicial y por último, **subList** es el método encargado de partir la lista principal.

A continuación, se muestra la salida del programa empleando el método subList:

```
***
500
17
700
***
false
```

Imagen 14: Salida de la lista una vez utilizado el método subList.

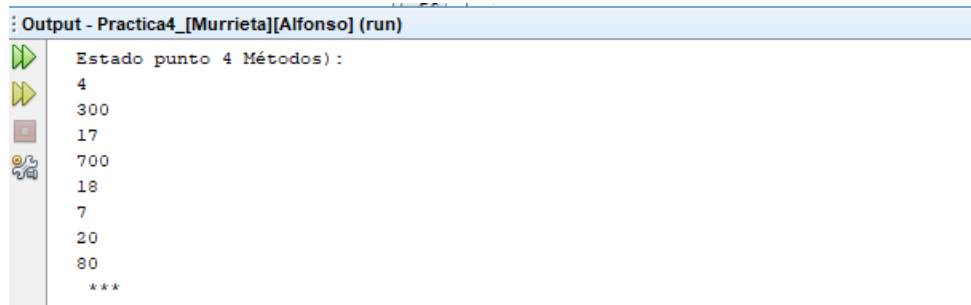
NOTA EXTRA: La última línea lo único que hace es comparar la lista inicial con la lista creada a partir del método sublist, al no ser iguales es por ello que la salida nos arroja “false”, cabe destacar que para la comparación entre listas se empleó otro método que fue “.equals”.

c) Método en Java; “Borrar”, “Está Vacía” y “Buscar un elemento”

Borrar – remove();

El método para eliminar un elemento en una lista en Java es “.remove();” donde el parámetro que se le pasa es el índice de la lista que se quiere eliminar, cabe denotar que en caso de que no se le pase parámetro al método por automático remueve el último elemento de la lista.

A continuación, se muestra la salida de un programa empleando este método:



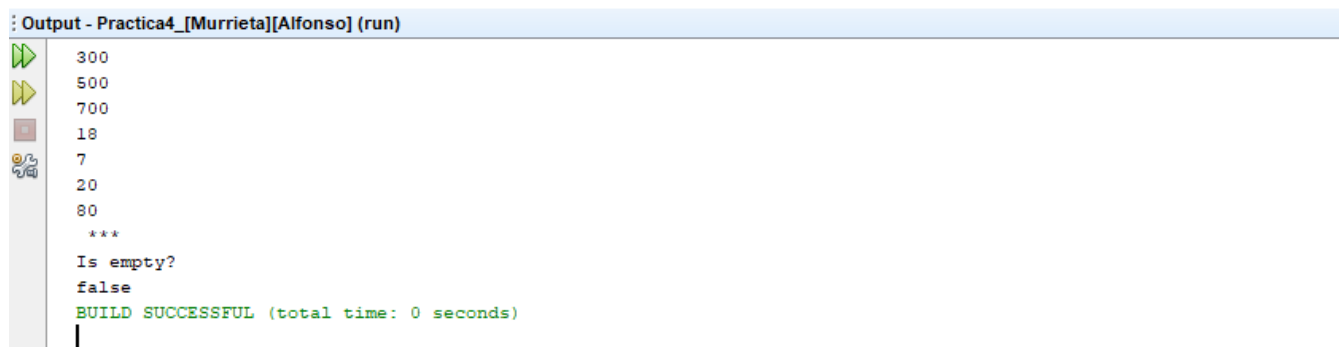
```
Output - Practica4_[Murrieta][Alfonso] (run)
Estado punto 4 Métodos):
4
300
17
700
18
7
20
80
***
```

Imagen 15: Salida de la lista una vez utilizado el método remove (Comparar con el estado punto 3).

Está Vacío – isEmpty();

El método saber si está o no vacía una lista en Java es .isEmpty(); donde no se le pasan ningún elemento o parámetro, sin embargo regresa a través de un valor booleano true en caso de que la lista tenga elementos o false en caso de que no tenga elementos o sea que esté vacía.

A continuación, se muestra la salida de un programa empleando este método:



```
Output - Practica4_[Murrieta][Alfonso] (run)
300
500
700
18
7
20
80
***
Is empty?
false
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 16: Salida de la lista indicando si tiene o no elementos.

Buscar elemento – contains();

Para poder saber si se encuentra un elemento en una lista o arreglo se puede hacer de varias formas, la primera forma es solamente indicando si existe o no el elemento dentro de la lista, esto a través de un valor booleano, sin embargo, también existe la posibilidad de saber dónde se encuentra el elemento o sea el índice de este, en el caso concreto de listas en Java existen varios métodos, que a continuación serán mencionados y mostrados en la imagen número 17.

```

59 //BUSCAR UN ELEMENTO
60 System.out.println("ELEMENTOS (BÚSQUEDA) : ");
61 System.out.println(listal.contains(11));
62 listal.add(7);
63 System.out.println(listal.indexOf(7));
64 System.out.println(listal.lastIndexOf(7));

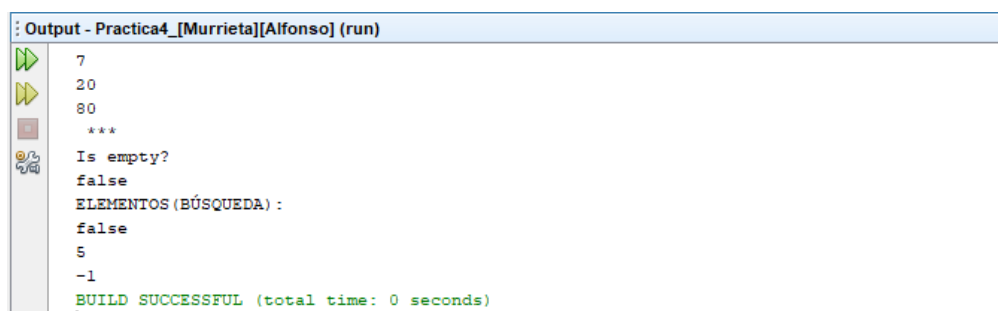
```

Imagen 17: Parte encargada de búsqueda de elementos

El primer método que se puede ver es el método “contains();” el cual se encarga de indicar si existe o no un valor dentro de la lista, lo que nos da como resultado es un valor booleano, por otro lado, tenemos los métodos indexOf(); y lastIndexOf(); los cuales son métodos encargados de indicar el índice del elemento que se les pasa como parámetro, en caso de que estos encuentren el elemento nos indican el índice de este, en caso de que no se encuentre simplemente nos dan como resultado un “-1” esto debido a que el valor 0 es una posición en una lista.

Cabe destacar que la diferencia de ambos métodos es que el primero nos dice el índice independientemente de la posición mientras que lastIndexOf como dice su nombre en inglés nos indica el último índice del elemento que se está buscando.

A continuación, se muestra la salida de un programa empleando este método:



```

Output - Practica4_[Murrieta][Alfonso] (run)
7
20
80
***
Is empty?
false
ELEMENTOS (BÚSQUEDA) :
false
5
-1
BUILD SUCCESSFUL (total time: 0 seconds)

```

Imagen 18: Salida del programa empleando los métodos de búsqueda de elementos en Java.

2.2 Búsqueda Lineal

Para este apartado se pidió programar una nueva clase llamada “Búsqueda Lineal” en la cual se implementarían 3 métodos principales donde estos abarcaran los 3 casos de una búsqueda. A continuación, se mostrarán y mencionarán cada uno de estos métodos

a) Método de verdadero o falso de una lista

En el caso de este método realmente resulta sencillo de programar lo único que debemos hacer es crear un método donde se le pase tanto el valor que se quiere buscar como la lista en la que se empezará ese valor, al solo querer saber si se encuentra o no el valor lo que podemos hacer es que el método sea del tipo booleano y de esa forma regresar true o falso a través de una variable. En la parte inferior se muestra la salida del programa (Imagen 19).

b) Método que devuelva el índice donde se encuentra el elemento (clave)

A diferencia del método anterior en este caso se quiere saber en qué posición de la lista se encuentra el elemento a buscar es por ello que lo que se debe primero hacer es utilizar un ciclo for para poder recorrer la lista, sin embargo, a su vez se debe emplear una variable externa que nos pueda guardar el valor del elemento a buscar en este caso se le denominó como index que al ser una posición se declaró del tipo entero.

En caso de no encontrar el elemento en la lista arroja el valor con el que fue inicializado que en mi caso fue -1, cabe mencionar que este valor puede depender del contexto con el que se trabajó o incluso con el punto de vista con el que se quiera trabajar.

En la parte inferior se muestra la salida del programa (Imagen 19).

c) Método que regresa la cantidad de veces que aparece un elemento en una lista

Por último, para este método se empleó la misma lógica que en el anterior método tanto desde el ciclo de búsqueda como los parámetros que se le mandaban al método, al igual que el tipo de método (Ambos métodos de tipo entero), sin embargo, la única diferencia radica en que este método debe llevar algo como un contador para poder de esta forma regresar el valor de veces que se repite un valor en la lista, es por ello que para esto lo único que se realizó fue utilizar un valor auxiliar que fuera incrementando durante el recorrido de la lista.

En la parte inferior se muestra la salida del programa (Imagen 19), siendo los elementos por buscar:

```
System.out.println(BusquedaLineal.True_False_Lineal(500,listal));  
System.out.println(BusquedaLineal.Index_Here_Lineal(300,listal));  
System.out.println(BusquedaLineal.Count_Lineal(7,listal));
```

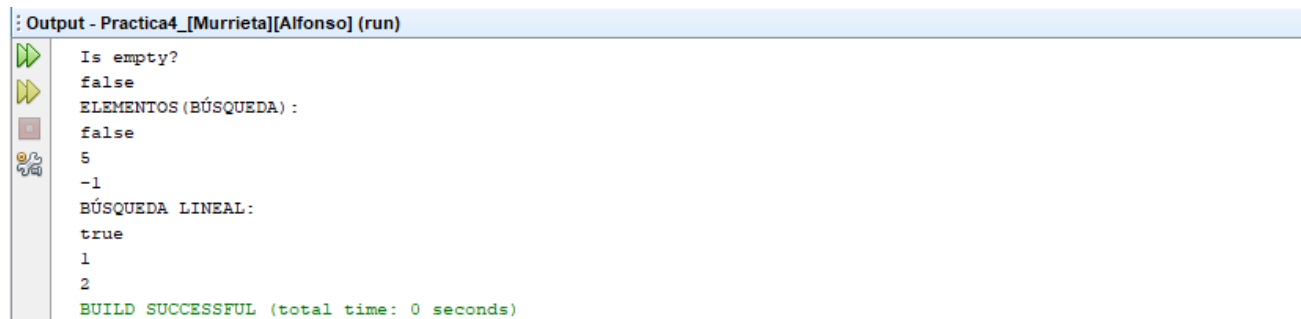


Imagen 19: Salida del programa empleando los métodos anteriormente mencionados de búsqueda lineal

Como se puede apreciar en la captura de pantalla en el caso de la primera búsqueda al existir por eso obtenemos un true, por otro lado, el elemento 300 está en la posición 1 de la lista siendo 0 la primera posición y por último el elemento 7 se repite 2 veces en la lista.

2.3 Búsqueda Binaria

Para este apartado se pidió programar una nueva clase llamada “Búsqueda Binaria” en la cual se implementarían 2 métodos principales donde estos abarcaran 2 de los 3 casos de una búsqueda. A continuación, se mostrarán y mencionarán cada uno de estos métodos

NOTA: Es importante mencionar que, para poder llevar a cabo este algoritmo de búsqueda, primero debe estar ordenada la lista o arreglo donde se llevará a cabo los métodos programados, sobre todo porque al momento de hacer las particiones de la lista en sub-listas, de un lado deben encontrar elementos mayores

al elemento intermedio de la lista mientras que del otro lado deben encontrarse elementos menores, esto con el fin de hacer la búsqueda lo más rápido posible.

a) Método de verdadero o falso de una lista

En el caso de este método realmente resultó sencillo de programar, lo único que se consideró previamente fue la versión recursiva analizada anteriormente en esta práctica para poder facilitar el manejo y programación de este método en Java.

Lo primero que se consideró es que el límite de búsqueda fuera al momento de que la lista tuviera un tamaño de 0 elementos (lo cual es más que lógico), por otro lado, dentro de esta condición se pusieron los diferentes casos que pueden darse al momento de buscar el elemento dentro de la lista:

- 1) Que el elemento sea directamente el punto intermedio de la lista, si es el caso directamente se regresa un true caso contrario pasa al segundo caso.
- 2) El segundo caso es que el elemento no se encuentre a la mitad por lo tanto a través de la recursividad (llamado de la misma función) es como se puede nuevamente buscar al elemento a la mitad de la lista, sin embargo, aquí es cuando se busca tanto del lado derecho como del izquierdo, en un caso donde los elementos son mayores al valor intermedio de la lista mientras que en el otro caso son menores.
- 3) EL tercer y último caso es que en caso de que no exista el elemento en la lista, simplemente se regresa un false

En la parte inferior se muestra la salida del programa (Imagen 20),

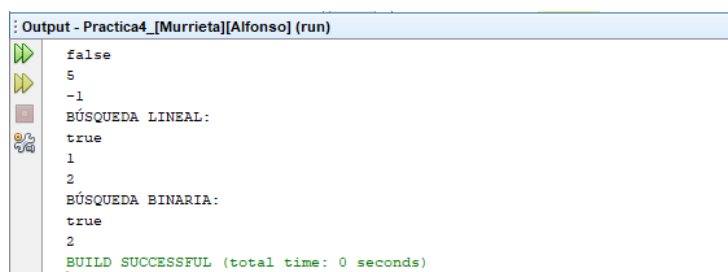
b) Método que regresa la cantidad de veces que aparece un elemento en una lista

Para este método se tomó prácticamente la misma lógica que el anterior método sin embargo, entre los cambios principales que se le hizo a este método fueron cambiar el tipo de método siendo este del tipo entero en vez del booleano como era el otro, a su vez en vez de regresar un valor de tipo booleano en este caso se regresaba una variable encargada de contar las veces que se repetía un elemento, es por ello que los parámetros que se le mandaban a este método son tanto el valor a buscar como la lista donde se quiere buscar un elemento.

NOTA: Es importante destacar que en este método a diferencia del otro al querer saber cuántas veces se repite un elemento en la lista lo que se debe hacer es sumar las veces de cada uno de los casos para de esa forma obtener el total de veces que se repite el elemento, es por ello que el mayor cambio de este método respecto al otro es en el primer if donde el contador se le suman todos los casos que existen dentro del algoritmo de búsqueda binaria.

En la parte inferior se muestra la salida del programa (Imagen 20), siendo los elementos a buscar:

```
System.out.println(BusquedaBinaria.True_False_Binaria(5,lista3));  
System.out.println(BusquedaBinaria.Count_Binario(6,lista3));
```



```
Output - Practica4_[Murrieta][Alfonso] (run)
false
5
-1
BÚSQUEDA LINEAL:
true
1
2
BÚSQUEDA BINARIA:
true
2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 20: Salida del programa empleando los métodos anteriormente mencionados de búsqueda binaria

2.4 Búsqueda en Lista de Objetos

Para este apartado realmente se realizó algo realmente distinto a lo que comúnmente se hacía en EDA, sobre todo porque comúnmente se trabajaba con tipos de datos primitivos o en dado caso con arreglo, sin embargo, al ser un semestre donde se está viendo programación orientada a Objetos se decidió llevar a cabo los algoritmos de búsqueda solamente que a entidades como son los objetos.

Requisitos y análisis previo

Para poder llevar este programa, lo primero que se necesita ver y analizar son todos los requisitos que se necesitan, a continuación, se muestran los requisitos del programa:

- 1) Agregar una nueva clase al proyecto anterior donde se definan atributos y métodos de un objeto denominado “EquipoFutbol”.
- 2) Dentro de las clases de búsqueda lineal y binaria se deben definir métodos dedicados a la búsqueda de un objeto del tipo equipo futbol a través del **nombre del equipo** y por el **nombre del estado**.
- 3) Llevar a cabo la prueba de todo lo mencionado anteriormente a través de la clase principal del proyecto.
-> Contemplar el funcionamiento de todos los apartados.

Creación de los métodos y clases

A continuación, se mencionarán cada uno de los métodos empleados para llevar a cabo la última actividad de la práctica.

A) Clase EquipoFutbol – Constructores

Como se ha visto en clase de POO, al momento de programar una clase esta puede programársele constructores los cuales son métodos encargados de definir qué datos tendrá el objeto al momento de crearlo, en este caso solamente he hecho 3 constructores, el estándar que solamente asigna valores nulos, uno donde se le asigna todos los valores a los atributos menos la cantidad de jugadores y por último el constructor donde se asignan todos los valores a un objeto (Este el que se empleó para el programa).

B) Clase EquipoFutbol – Setters y Getters

Los setter de cada uno de los atributos los había hecho con el propósito de volver los atributos de la clase privados, sin embargo, por alguna razón que desconozco todavía, al momento de emplear los setters y getters en las clases de ordenamiento binario y lineal, me marcaban error de privacidad, es por ello que finalmente tuve que volver los atributos públicos para poder corregir este pequeño detalle.

```
//Atributos no son privados :({
    String nombreEquipo;
    String estadoEquipo;
    String paisEquipo;
    String divisionEquipo;
    int numJugadores;

//Constructores

EquipoFutbol() {
    this.nombreEquipo = "No name";
    this.estadoEquipo = "No data";
    this.paisEquipo = "No data";
    this.divisionEquipo = "No data";
    this.numJugadores = 12;
}
```

Imagen 21: Parte del programa dedicado en la declaración de los atributos del objeto y su respectivo constructor estándar.

C) Clase EquipoFutbol – Método de impresión

Por último, para poder llevar a cabo de forma más eficiente y rápida la impresión de todos los elementos de la lista de objetos lo que realicé fue una función de impresión a la que solamente se le pasaba la lista de objetos y como resultado imprimía todos los datos de los objetos de la lista. A continuación, se muestra la salida del programa empleando este método.

```
: Output - Practica4_[Murrieta][Alfonso] (run)

BÚSQUEDA DE OBJETOS:

=====
Equipo: Real Madrid
Estado: Madrid
País: España
Division: UEFA
Jugadores: 20
=====

Equipo: Barcelona
Estado: Barcelona
País: España
Division: UEFA
Jugadores: 21
=====

Equipo: Valencia
Estado: Madrid
País: España
Division: UEFA
Jugadores: 22
=====

Equipo: Paris
Estado: Paris
País: Francia
Division: UEFA
Jugadores: 18
```

Imagen 22: Salida del programa, indicando los datos de cada uno de los objetos creados.

D) Clase Búsqueda Lineal – Método “T_F_L_” de búsqueda

En este método he utilizado exactamente el mismo método empleado para la búsqueda lineal de enteros en una lista de enteros, sin embargo, para poder adaptar el método tuve que primero cambiar los parámetros que se le pasaban siendo así de un método booleano que se le pasaba un entero y como valor a buscar también un entero, ahora iba a ser un método de tipo booleano (Debido a que solo se quiere saber si existía o no el objeto dentro de la lista), donde lo que se buscaba era ya sea el atributo del nombre o el del estado, además la lista que se le pasaba era de objetos.

E) Clase Búsqueda Lineal – Método “Index_Here_L_Nombre” de búsqueda

En este método he utilizado exactamente el mismo método empleado para la búsqueda lineal de enteros en una lista de enteros, sin embargo, para poder adaptar el método tuve que primero cambiar el tipo de método y los parámetros que se le pasaban siendo así de un método entero que se le pasaba un entero y como valor a buscar también un entero, ahora iba a ser un método de tipo entero (Debido a que solo se quiere saber la posición del objeto dentro de la lista), donde lo que se buscaba era ya sea el atributo del nombre o el del estado, además la lista que se le pasaba era de objetos.

Realmente el adaptar esta función no fue ningún problema sobre todo porque el mismo lenguaje te da las herramientas necesarias para poder llevar a cabo la lógica como por ejemplo el método equals para comparar cadenas (Los nombres).

En este caso lo que se regresaba era el índice con la posición del objeto que se quería buscar.

F) Clase Búsqueda Lineal – Método “Count_Lineal_” de búsqueda

Al igual que el método anterior, es del tipo entero donde solamente se le pasa una cadena y una lista de objetos, sin embargo, en este método no se manda la posición si no una variable encargada de almacenar las veces que se repite un elemento en la lista.

```
public static int Count_Lineal_Estado(String estado, List<EquipoFutbol> lista){  
    int count =0;  
    for (EquipoFutbol clave: lista){  
        if ((clave.estadoEquipo.equals(estado))){  
            count = count +1;  
        }  
    }  
    return count;  
}
```

Imagen 23: Método de búsqueda lineal , cantidad de elementos, en este caso particular a través de la clave Estado

G) Clase Búsqueda Binaria – Método “T_F_B_” de búsqueda

Esta clase al igual que la empleada para buscar enteros en una lista de enteros (Análisis en parte previa del reporte), como parámetros necesita tanto el objeto que se quiere buscar en específico la clave de nombre o estado y la lista correspondiente.

Sin embargo, para poder llevar a cabo de forma más cómoda el apartado de búsqueda fue necesario emplar algunos métodos incluidos en la api de Java.util como fue .size() , .compareTo, subList y .get(). Por ejemplo el primero fue utilizado para llevar a cabo la parte del tamaño de la lista ya que al haber programado la versión recursiva es necesario conocer el tamaño de la lista para poder parar la recursividad en el caso base (Qué es cuando la lista tiene 0 elementos), por otro lado, compareTo para hacer las debidas comparaciones con los índices de la lista esto con el fin también de determinar los valores encontrados respecto al valor que se buscaba, por último, .subList y .get se emplearon primero para poder llevar a cabo la división de la lista principal en 2 sub-listas secundarias y para conseguir el tamaño de la lista en el momento de la comparación respecto a los diferentes casos que existen dentro del algoritmo.

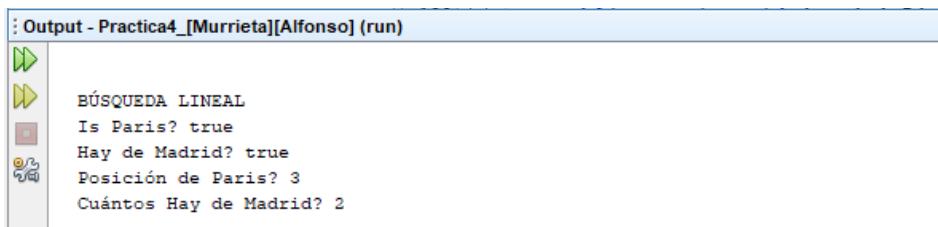
H) Clase Búsqueda Binaria – Método “Count_B_de búsqueda

Por último, el método para poder contar la cantidad de objetos a buscar que existe en la lista, este método realmente fue el que más problemas tuve sobre todo porque al momento de llevarlo al caso recursivo lamentablemente no me había fijado y había colocado erróneamente los casos recursivos, sin embargo, en este momento ya es funcional y trabaja adecuadamente.

Al igual que su versión con lista de enteros trabaja con la misma lógica y regresa exactamente una variable de tipo entera encargada de contar la cantidad de veces que un objeto en concreto se encuentra en la lista de objetos que se le pasa.

Resultados y salida del programa:

A continuación, se muestra la salida del programa empleando todos los métodos anteriormente mencionados:



```
Output - Practica4_[Murrieta][Alfonso] (run)
BÚSQUEDA LINEAL
Is Paris? true
Hay de Madrid? true
Posición de Paris? 3
Cuántos Hay de Madrid? 2
```

Imagen 24: Salida del programa empleando los métodos de búsqueda lineal para la búsqueda de objetos

Como se puede apreciar en la imagen anterior, en el primer y segundo resultado vemos la búsqueda lineal en su versión de true o false de un objeto en una lista, posteriormente se ve el método encargado de darnos el índice del objeto que se está buscando y por último se ve el método que cuenta cuantas veces está presente un objeto en una lista.


```
Output - Practica4_[Murrieta][Alfonso] (run)
BÚSQUEDA BINARIA
Is Valencia? true
Cuántos hay de Berlín? 0
Is Madrid? true
Cuántos Hay de Madrid? 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 25: Salida del programa empleando los métodos de búsqueda binaria para la búsqueda de objetos

Como se puede apreciar en la imagen anterior, en el primer resultado vemos la búsqueda binaria en su versión de true o false de un objeto en una lista mientras que el segundo resultado vemos el método encargado de indicar cuántos elementos hay en la lista, ambos a través de la clave “nombreEquipo”, por otro lado en el tercer resultado vemos la búsqueda binaria en su versión de true o false de un objeto en una lista mientras que el cuarto resultado vemos el método encargado de indicar cuántos elementos hay en la lista, sin embargo esta vez a través de su clave “estadoEquipo”.

Errores corregidos de la versión 2

La única corrección que se realizó a los códigos respecto al a versión anterior de envío (La versión 2), es que para llevar a cabo búsqueda binaria es necesario tener la lista de los objetos o datos primitivos ordenada, por lo que al comentarlo en clase directamente se decidió emplear algún método ya establecido en Java para poder llevar a cabo el ordenamiento de estas listas, cabe destacar que otra opción pudo haber sido crear un propio método pero para mayor comodidad se decidió emplear el .sort de Java para llevar a cabo los ordenamientos tanto de la lista de enteros como la lista de objetos.

Sort para lista de enteros

En el caso de la lista de enteros solamente se tuvo que llamar el método .sort() y pasarle directamente la lista que se quería ordenar, realmente este apartado no fue nada complicado e incluso es muy común encontrar ayuda en foros o blogs de programación como stackoverflow para poder ver cómo funciona o cómo se emplea.

Sort para lista de objetos

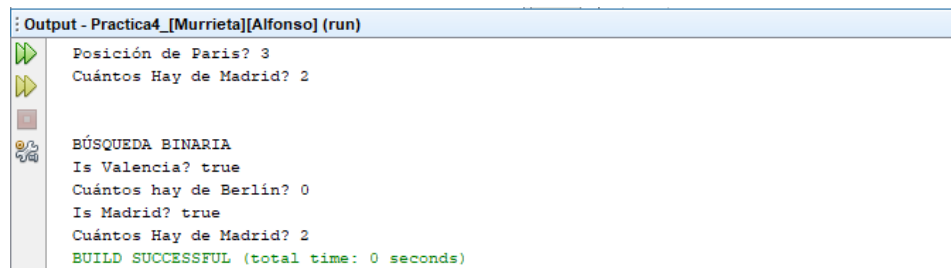
En el caso particular de la lista de objetos realmente resultó muy difícil encontrar un método o la manera adecuada de llevar un método común a una tarea más específica y sobre todo poco común. Con ayuda de un amigo de la clase y sobre todo por la ayuda en la página web de stackoverflow fue como directamente conocí y empleé el método .sort() pero para ordenar una lista de objetos. A continuación, se muestra las líneas de código que se usaron para llevar a cabo el ordenamiento:

```
//BÚSQUEDA BINARIA
System.out.println("\n\nBÚSQUEDA BINARIA");
//La parte mediante el nombre
Collections.sort(EquiposDeFutbol, (o1, o2) -> o1.getNombreEquipo().compareTo(o2.getNombreEquipo()));
```

Lo primero que vemos es que empleamos un collections sobre el cual aplicamos o llamamos el método sort para poder llevar a cabo el ordenamiento de la lista, los parámetros que se le pasan al sort son tanto la lista y a su vez la expresión conocida como “lambda 8”

Lambda 8 es una función sin nombre. Es un atajo que permite declarar un método en el que se va a utilizar, sin especificar el nombre del método, el tipo de cambio ni el acceso, es de esta forma que permite tratar el código como los datos y usar funciones como argumentos de método.

Es por ello que a través de los getters del objeto y a través de dos objetos (o1 y o2) es como se lleva a cabo la lógica de comparación, también es destacable que debido a que lo que queremos comparar son cadenas es por ello que se emplea el método `compareTo`.

A screenshot of a terminal window titled "Output - Practica4_[Murrieta][Alfonso] (run)". The output shows several lines of text: "Posición de Paris? 3", "Cuántos Hay de Madrid? 2", a separator line, "BÚSQUEDA BINARIA", "Is Valencia? true", "Cuántos hay de Berlín? 0", "Is Madrid? true", "Cuántos Hay de Madrid? 2", and "BUILD SUCCESSFUL (total time: 0 seconds)".

```
Output - Practica4_[Murrieta][Alfonso] (run)
Posición de Paris? 3
Cuántos Hay de Madrid? 2

BÚSQUEDA BINARIA
Is Valencia? true
Cuántos hay de Berlín? 0
Is Madrid? true
Cuántos Hay de Madrid? 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 26: Salida del programa empleando los métodos de búsqueda binaria para la búsqueda de objetos y además de emplear el ordenamiento en la lista

Como se puede ver (Comparando la imagen 26 respecto a la 25) en este caso nos indica que existen 2 elementos con el estado Madrid, dando como resultado la salida correcta del programa.

Errores o puntos por mejorar y optimizar

Como cualquier buena práctica de programación siempre hay que ser sincero con nuestros resultados y sobre todo con lo que podemos mejorar al momento de programar o re-programar algo, en este caso y a mi parecer hay 2 aspectos principales que quedarían pendientes, el primero es que a pesar de que se hace la búsqueda ya sea lineal o binaria se repiten prácticamente 2 veces los mismos métodos esto debido a que lo único que cambia para buscar ya sea a través del nombre o estado es el atributo asociado, lo que realmente quisiera es que a través de un mismo método se pudiera realizar ambas búsquedas sobre todo para reducir líneas de código, por otro lado, un error o aspecto que realmente quisiera checar es el por qué no pude utilizar los getters y setters dentro de las clases de búsqueda lineal y binaria sobre todo porque no tiene caso tener esos métodos si realmente voy a dejar los atributos de la clase `EquipoFutbol` como públicos.

Conclusiones

Para poder llevar a cabo mis conclusiones de una manera más concreta y ordenada, he decidido partir en 2 secciones principales mis conclusiones, la primera dedicada a la parte previa a la segunda y última entrega de mi práctica y la segunda enfocada solamente a la parte principal del desarrollo de los ejercicios del laboratorio 2 – 4.

Los ejercicios del manual de la práctica realmente me resultaron muy sencillos de hacer, creo que conforme uno va programando constantemente cada vez ciertas cosas se vuelven más sencillas, o tal vez sea que en verdad Python facilita muchísimo las cosas al momento de programar algo simple, porque siendo sincero no me imagino programar todo lo que realicé en Java en Python.

También quiero aclarar y destacar que el enfoque que se le dio al primer ejercicio del laboratorio me pareció muy llamativo ya que en verdad muchas veces utilizamos muchísimos métodos en Java (Es mi

caso en POO) que realmente no tenemos idea alguna de que es lo que hay detrás de todo eso, el caso más simple es como se utiliza descaradamente el método sort o el método add o remove en una lista de Java.

Por otro lado, los ejercicios del laboratorio sobre todo el 2 y 3 me parecieron realmente didácticos de llevar a cabo en EDA sobre todo porque comúnmente trabajamos con datos de tipo primitivo lo cual resulta muy fácil asimilar las cosas al momento de programarlas, sin embargo, el ejercicio 4 en verdad fue lo más pesado y largo de programar en todo lo que llevo de EDA 1 y 2 y es que en verdad el llevar tus conocimientos de una materia externa como es POO a una materia como es EDA no es del todo sencillo, idear la forma de realizar búsquedas de objetos en listas de objetos en verdad suena realmente algo extraño o incluso poco común, sin embargo, y analizando bien la situación actual creo que puede ser de lo más común al momento de programar algo de la vida real, no creo que realmente al momento de escoger un arma o un objeto en un videojuego esta se escoja a través de solo variables primitivas sino puede que se hagan búsquedas o incluso ordenamiento de objetos, sin duda alguna algo fascinante que falta por descubrir.

REFERENCIAS

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). Problem Solving with Algorithms and Data Structures using Python. Segunda Edición.

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.

Elba Karen Saenz García. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos II*. UNAM, Facultad de Ingeniería.

Recuperado el 10 de septiembre de 2018, de <https://es.stackoverflow.com/questions/27816/ordenar-un-list-de-objetos-en-java>