	<b>Carátula para entrega de prácticas</b>	
Facultad de Ingeniería	Laboratorio de docencia	

# Laboratorios de computación salas A y B

---

*Profesor: M.I. EDGAR TISTA GARCÍA*

*Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS 2*

*Grupo: 8*

*No de Práctica(s): 4] ALGORITMOS DE BUSQUEDA PARTE 1*

*Integrante(s): ADOLFO ROMÁN JIMENEZ*

*No. de Equipo de cómputo empleado: TRABAJO EN CASA*

*No. de Lista o Brigada:*

*Semestre: 2022 - 1*

*Fecha de entrega: 13 DE OCTUBRE DE 2021*

**CALIFICACIÓN:** \_\_\_\_\_

# Practica 4

## Algoritmos de Busqueda Parte 1

Adolfo Roman Jimenez

October 14, 2021

### **1 Objetivo**

El estudiante identificará el comportamiento y características de los principales algoritmos de búsqueda por comparación de llaves.

### **2 Objetivo de Clase**

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 Desarrollo

### 3.1 Ejercicio 1: Listas en Java

La diferencia basica entre los metodos *set()* y *add()* tiene que ver con la forma en la que insertan los elementos a una lista, pues ambos de estos metodos se usan como parte de las funciones internas de las clases tipo Lista.

El metodo *add()* emplea dos parametros, que son en primera el lugar del indice en donde se va a colocar el elemento que se pretende agregar a la lista y en segundo lugar, el elemento que se pretende agregar como tal.

En caso de que el parametro que corresponde al indice se deje en blanco, entonces simplemente el elemento se agregara al final de la lista.

Si el area de indice no se deja en blanco y se indica donde se requiere el elemento, entonces lo que este metodo hara, es insertar al elemento dentro del indice requerido y empujar a los demas elementos a su derecha para crear espacio para el nuevo elemento.

En cambio, el metodo *set()* tambien toma 2 parametros pero este necesariamente debe de tomar un indice primero y despues el elemento a agregar, pero a diferencia de *add()*, este metodo reemplaza al elemento en el indice en el que se posiciona, por lo que no hace ningun arrastre de elementos ni mucho menos.

### 3.2 Ejercicio 2: Busqueda Lineal

#### 3.2.1 Desarrollo

1. Al primer metodo de la clase **BusquedaLineal** lleva por nombre *boolSearch()*. Este metodo utiliza 2 parametros de entrada que es una lista de datos de tipo de entero y una clave a buscar.

La implementacion es muy sencilla, primeramente se declara una variable booleana llamaba **search** cuyo valor inicial se iguala a *false*. Despues de esto se declara un *for-loop* que itera sobre cada uno de los elementos de esta lista con un iterador de tipo entero, el cual, dentro del cuerpo del *loop* se utiliza un metodo de control *if()* que evalua si el valor del elemento de la lista en ese momento es igual al valor que estamos buscando y cuando esto es verdadero, entonces se modifica la variable **search**, se iguala a *true* y se ejecuta un **break** para salir en ese momento del *loop* debido a que significa que ya se encontro el elemento buscado. Si no se llega a encontrar el elemento, entonces simplemente el *loop* finaliza al momento de llegar al ultimo elemento de la lista.

Al final, el metodo a traves de la funcion *println()* imprime el valor en **search** que sera verdadero si el elemento fue encontrado y falso si no lo fue.

2. El siguiente metodo que lleva por nombre *indexSearch()* es similar al metodo anterior, pero contiene ciertas modificaciones para que cumpla con lo requerido por la practica.

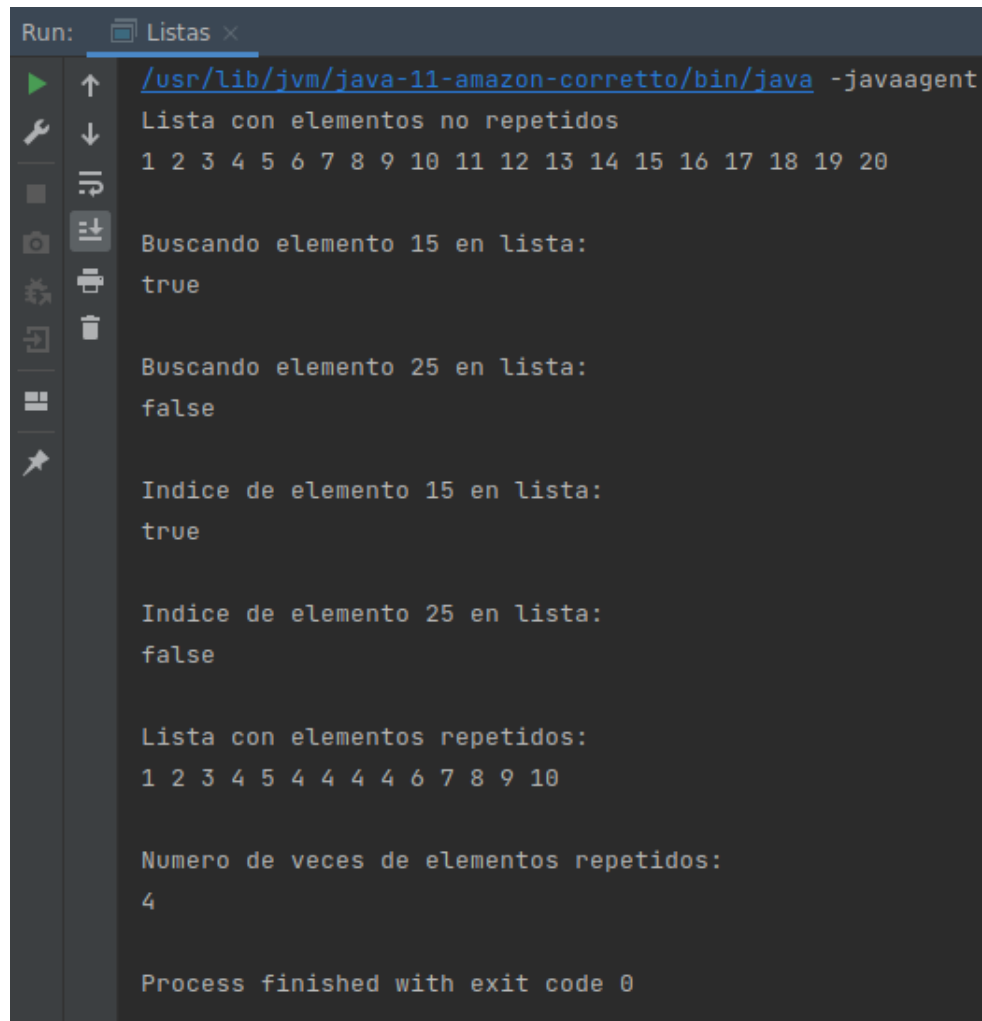
Este metodo de igual forma toma como parametros una lista de enteros y la clave a encontrar, pero a diferencia del metodo anterior, al momento de iterar a traves de la lista, no se hace por medio de los elementos, sino por medio de un *for-loop* regular que itera desde 0 hasta el tamano de la lista menos uno.

Al momento de encontrar el elemento, entonces el *loop* imprime el indice en el que fue encontrado ese elemento, de lo contrario no imprime ningun numero.

3. Esta funcion tiene el nombre de *timesSearch()* y usa la misma estructura que el metodo *boolSearch()* pero con la diferencia de que se declara una variable entera **count** que se iguala a 0 para poder contar el numero de veces que un elemento aparecera dentro de la lista.

El metodo acepta los mismos parametros que las funciones anteriores e imprime 0 en caso de que el elemento no se encuentre en la lista. Para determinar si el elemento se encuentra o no en la lista, se usa el iterador de elementos y no por medio de valores como en el *loop* anterior.

### 3.2.2 Ejecucion



```
Run: Listas x
/usr/lib/jvm/java-11-amazon-corretto/bin/java -javaagent
Lista con elementos no repetidos
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Buscando elemento 15 en lista:
true

Buscando elemento 25 en lista:
false

Indice de elemento 15 en lista:
true

Indice de elemento 25 en lista:
false

Lista con elementos repetidos:
1 2 3 4 5 4 4 4 4 6 7 8 9 10

Numero de veces de elementos repetidos:
4

Process finished with exit code 0
```

Figure 1: Ejecucion de metodos de la clase BusquedaLineal

### 3.3 Ejercicio 3: Búsqueda Binaria

#### 3.3.1 Desarrollo

1. Primeramente se creo la clase *BusquedaBinaria* para poder hacer este ejercicio y despues se le agrego el primer algoritmo que la practica solicita llamado *binBoolSearch()*.

El metodo *binBoolSearch()* toma como parametros una lista de enteros y la clave a encontrar dentro de esta lista. Despues de esto, se declaran 3 variables enteras, que son **right** que se inicializa con el valor del tamano de la lista menos 1, **mid** y **left** la cual se inicializa en 0, asi como una variable booleana que es **found** y que se inicializa en falso.

Acto seguido, se declara un *while-loop* que al momento de ingresar al ciclo, asigna a la variable **mid** el valor de las variables **right** + **left** entre 2, esta variable sera determinante para el indice a evaluar en la lista.

Despues de esto, se presentan 3 bloques *if()* de codigo, el primero evaluara si la posicion de **mid** es igual al valor que se esta buscando, para esto se usa el metodo *equals()* y si esto es verdadero, la variable **found** se convierte a *true* e inmediatamente se ejecuta un *break* para romper el ciclo.

En cambio si el valor en el indice que se evalua, no es el que se esta buscando, entonces, los siguientes 2 bloques de codigo, evaluan si el valor en el indice es mayor o menor que el valor que se esta buscando. Si es menor entonces a **left** se le suma 1 y si es mayor entonces a **right** se le resta 1 y se comienza con un ciclo distinto que no terminara hasta que exista una "colision" entre los indices y **left** sea mayor que **right**.

2. Para este metodo llamado *binTimesSearch()* tambien se usaron como parametros la lista de enteros y la clave que se pretende encontrar dentro de la lista.

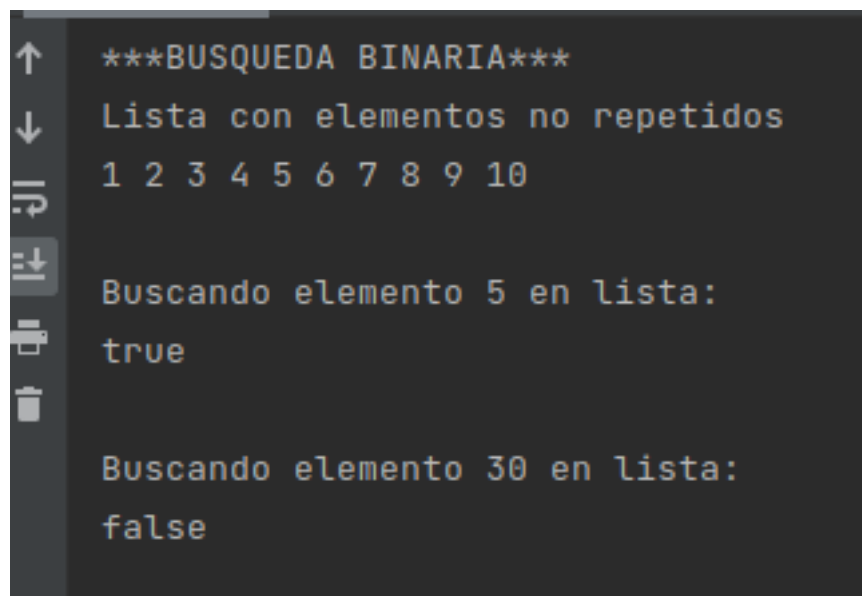
La estructura de este metodo es la misma que la de *binBoolSearch()* pero se agregan un par de variables adicionales que se llaman **left** y **right** que servirán para llevar indices y ademas una variable **times** que sirve para contar las veces en las que un elemento se encuentra en la lista.

El funcionamiento es casi el mismo que en el metodo pasado, pero con la diferencia de que cuando el metodo encuentra el elemento, se ingresa se iguala el valor de la variable **mid** que lleva el conteo de los indices y en **left** se deposita ese mismo valor menos 1, en **right** ese valor mas 1 y se comienza el conteo en la variable **times** al mismo tiempo que continua la

iteracion en el subciclo.

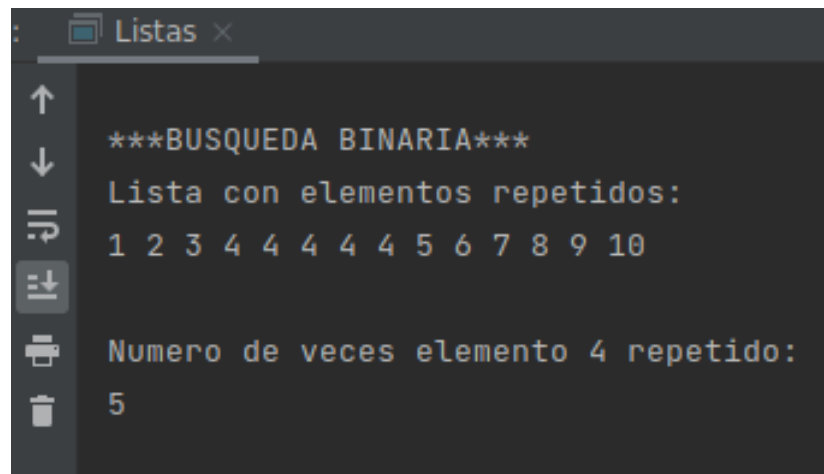
A medida que el subciclo va progresando, evalua tanto si a la derecha como a la izquierda del indice encontrado originalmente, se encuentra ese mismo valor y va contando las veces que lo va encontrado a medida que progresa de un lado o del otro y el ciclo se rompe cuando deja de encontrar el valor en cualquier direccion ejecutando un *break* para salir y imprimiendo en pantalla el valor de **times** para ese momento.

### 3.3.2 Ejecucion

A screenshot of a terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for navigation (up, down), search, and other standard terminal functions. The main area of the terminal displays the following text:

```
***BUSQUEDA BINARIA***  
Lista con elementos no repetidos  
1 2 3 4 5 6 7 8 9 10  
  
Buscando elemento 5 en lista:  
true  
  
Buscando elemento 30 en lista:  
false
```

Figure 2: Ejecucion de boolBinSearch()



```
: Listas x
***BUSQUEDA BINARIA***
Lista con elementos repetidos:
1 2 3 4 4 4 4 4 5 6 7 8 9 10
Numero de veces elemento 4 repetido:
5
```

Figure 3: Ejecucion de binTimesSearch()



## 3.4 Ejercicio 4: Búsqueda en Listas de Objetos

### 3.4.1 Desarrollo

1. Para este ejercicio lo que hice fue crear una nueva clase llamada *Computadora* e inserte diversos atributos privados a la clase los cuales fueron:
  - (String) Marca
  - (double) Precio
  - (String) Tipo
  - (int) RAM
  - (String) OS
  - (String) Procesador

Despues de esto escribi, los constructores correspondientes, cada uno con 3 atributos como parametros distintos, uno para los atributos que no son especificaciones tecnicas de la computadora y otro para cosas mas generales de ella.

Al finalizar cree automaticamente los *getters* y los *setters* e inserte un par de metodos adiciones llamados *toStringStore()* y *toStringSpecs()* las cuales, cada una, utilizan el metodo *String.format()* para convertir en cadenas de caracteres los diferentes atributos que correspondan de acuerdo a los metodos.

2. Al par de metodos de creados para la búsqueda lineal, corresponden los nombres de *searchBrandPC()* y *searchRamPC()*, la primera busca, como su nombre lo indica, una computadora por marca, lo que implica que tiene que buscar un tipo de dato *String* y la segunda un tipo de dato *int* pues la cantidad de memoria RAM corresponde a un numero entero.

Para los dos metodos los parametros de entrada son los mismos y corresponden en esta ocasion a una lista de instancias de la clase *Computadora* y tambien se agrega, para el caso de *searchBrandPC* un parametro tipo *String* que representa la marca de la computadora a buscar y para el caso de *searchRamPC()* se incluye un parametro de tipo entero para buscar la computadora que contenga el valor deseado de memoria RAM.

Para ambos casos la estructura del metodo iterador es la misma, se declara un *enhanced for-loop* que itera a traves de los elementos de la lista y evalua con el metodo *Objects.equals()* si el atributo ya sea de marca o de memoria, es el mismo que el que se esta buscando. Si es asi, entonces se imprimen los valores de la computadora encontrada a traves de los metodos *toStringStore()* y *toStringSpecs()*, en caso de ser falso, el metodo solo termina su

ejecucion.

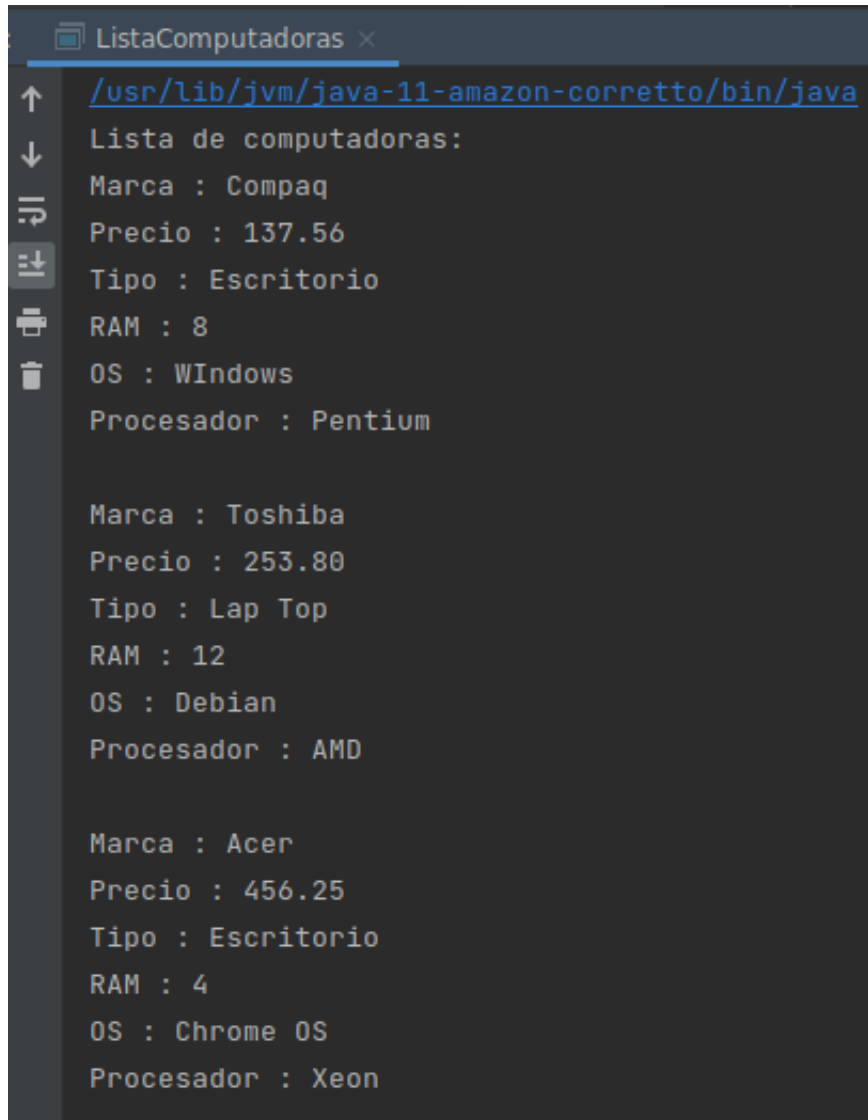
3. Para el caso de la busqueda binaria por marca, decidi hacer una busqueda tipo *mergeSort()* que utiliza la recursividad para iterar en toda la lista y encontrar el atributo del elemento requerido.

El metodo lleva el nombre de *binBrandSearch()* y recibe como parametros una lista de clase *Computadora* que se llama **listaPC**, una cadena string **brand** un numero entero **start** y otro numero entero **end**.

El metodo comienza utilizando el metodo *subList()* para crear una sublista en **listaPC** que va desde **start** hasta **end** - 1 y despues de esto un entero **mid** evalua la mitad de la lista para hacer el corte recursivo en las subsecuentes funciones, las cuales son 2 y reciben la sublista creada al principio de la funcion pero una obtiene parametros desde 0 hasta la mitad de la lista mientras que la otra funcion obtiene parametros desde la mitad de la lista hasta el final de esta.

El caso base se da cuando la longitud de la lista es 1, para lo cual cuando eso sucede, se evalua si **brand** es igual a la marca que contiene el objeto, si esto es asi, entonces se imprime el valor **true** si no entonces el programa no imprime nada y simplemente sale de su ciclo recursivo con el return al final de los comandos.

### 3.4.2 Ejecucion



```
ListaComputadoras x
/usr/lib/jvm/java-11-amazon-corretto/bin/java
Lista de computadoras:
Marca : Compaq
Precio : 137.56
Tipo : Escritorio
RAM : 8
OS : WIndows
Procesador : Pentium

Marca : Toshiba
Precio : 253.80
Tipo : Lap Top
RAM : 12
OS : Debian
Procesador : AMD

Marca : Acer
Precio : 456.25
Tipo : Escritorio
RAM : 4
OS : Chrome OS
Procesador : Xeon
```

Figure 4: Ejecucion de Clase Computadora y Busqueda Binaria

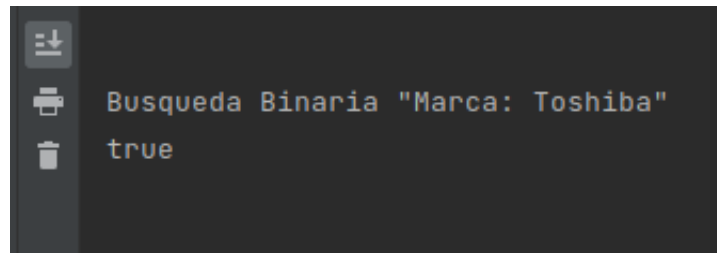
```
Busqueda lineal "Compaq"  
Marca : Compaq  
Precio : 137.56  
Tipo : Escritorio  
RAM : 8  
OS : WIndows  
Procesador : Pentium
```

```
Busqueda lineal "RAM: 4"  
Marca : Acer  
Precio : 456.25  
Tipo : Escritorio  
RAM : 4  
OS : Chrome OS  
Procesador : Xeon
```

Figure 5: Ejecucion de Clase Computadora y Busqueda Binaria

```
Busqueda Binaria "RAM: 12"  
Marca : Toshiba  
Precio : 253.80  
Tipo : Lap Top  
RAM : 12  
OS : Debian  
Procesador : AMD
```

Figure 6: Ejecucion de Clase Computadora y Busqueda Binaria

A terminal window with a dark background and a light gray sidebar on the left. The sidebar contains three icons: a list with a downward arrow, a printer, and a trash can. The terminal text is as follows:

```
Busqueda Binaria "Marca: Toshiba"
true
```

Figure 7: Ejecucion de Clase Computadora y Busqueda Binaria

## 4 Conclusiones

Esta practica estuvo muy interesante porque me permitio aplicar los algoritmos de busqueda binaria de diferntes formas a como lo habia hecho anteriormente.

Una de esas formas que mas interesantes me parecieron y que me deja la practica fue como se puede modificar la busqueda binaria para buscar linealmente dentro del mismo objeto como se hizo en el caso de los numeros repetidos dentro de un arreglo.

Por el momento ya estoy mas interesando en aprender a implementar estos algoritmos no solamente para numeros sino para tipos de datos abstractos e instancias de objetos porque me gustaria conocer como le hacen para ordenar datos cuando vienen por grandes cantidades.

Me parece que se cumplieron los objetivos de la practica y espero pronto la siguiente.

Gracias por leer mi practica! :)