



# Universidad Nacional Autónoma de México

FACULTAD DE INGENIERÍA  
ESTRUCTURAS DE DATOS Y ALGORITMOS II

*Grupo: 07 - Semestre: 2024-1*

## PROYECTO 2 – ALGORITMOS DE BÚSQUEDA

FECHA DE ENTREGA: 08/10/2023

### **Profesor:**

Edgar Tista Garcia

### **Equipo 10 - Alumno(s):**

Hernandez Gallardo Daniel Alonso

Perez Osorio Luis Eduardo

Valle Chavez Anton Yael

## **Abstract**

This project is based on prior research that focuses on the main collections in Java and their class hierarchies. In this research, implementations of lists, hash tables, and sets in Java are explored, along with their key differences.

The class hierarchy in Java starts with the `java.util.Collection` interface, which has subinterfaces such as `List`, `Set`, and `Map`. Each of these subinterfaces represents different types of collections with unique characteristics. For instance, `List` allows duplicates and maintains a specific order, `Set` disallows duplicates, and `Map` represents a collection of key-value pairs.

Within these categories, specific implementations were investigated. Regarding lists, `ArrayList`, `LinkedList`, and `Vector` were examined, each with its advantages and specific use cases. In the context of hash tables, `HashMap`, `LinkedHashMap`, and `TreeMap` were considered, offering varying levels of performance and ordering. Additionally, in the realm of sets, `HashSet`, `LinkedHashSet`, and `TreeSet` were explored, each with its own duplication and ordering characteristics.

This research lays the foundation for understanding data structures and collections in Java, which is essential for the development of search and chaining applications. The programs implement key comparison search algorithms such as linear search and binary search, in the context of comparable objects like `Students` and `Subjects`. Furthermore, collision resolution is explored through chaining in a simulated hash table, and an interactive interface is created to allow users to add elements to the table and view its contents.

## Resumen

Este proyecto se basa en una investigación previa que se centra en las principales colecciones en Java y sus jerarquías de clases. En esta investigación, se exploran las implementaciones de listas, tablas hash y conjuntos (Set) en Java, así como las diferencias clave entre ellas.

La jerarquía de clases en Java comienza con la interfaz `java.util.Collection`, que tiene subinterfaces como `List`, `Set` y `Map`. Cada una de estas subinterfaces representa diferentes tipos de colecciones con características únicas. Por ejemplo, `List` permite duplicados y mantiene un orden específico, `Set` no permite duplicados y `Map` representa una colección de pares clave-valor.

Dentro de estas categorías, se investigaron las implementaciones específicas. En el caso de listas, se examinaron `ArrayList`, `LinkedList` y `Vector`, cada una con sus ventajas y casos de uso particulares. En cuanto a las tablas hash, se consideraron `HashMap`, `LinkedHashMap` y `TreeMap`, que ofrecen diferentes niveles de rendimiento y ordenación. Además, en el contexto de conjuntos, se exploraron `HashSet`, `LinkedHashSet` y `TreeSet`, cada uno con sus propias características de duplicación y orden.

Esta investigación sienta las bases para comprender las estructuras de datos y las colecciones en Java, lo que es esencial para el desarrollo de las aplicaciones de búsqueda y encadenamiento. Los programas implementan algoritmos de búsqueda por comparación de llaves, como búsqueda lineal y búsqueda binaria, en el contexto de objetos comparables, como `Alumnos` y `Asignaturas`. Además, se explora la solución de colisiones mediante el encadenamiento en una tabla hash simulada y se crea una interfaz interactiva para permitir a los usuarios agregar elementos a la tabla y ver su contenido.

## Índice general

<b>1. Introducción</b>	<b>3</b>
<b>2. Desarrollo</b>	<b>20</b>
<b>3. Conclusiones</b>	<b>58</b>
Referencias . . . . .	61

## **Objetivo**

Que el alumno desarrolle aplicaciones para la búsqueda por comparación de llaves y la transformación de llaves junto con la solución de colisiones

## **Investigación**

### **Framework**

Un framework es una agrupación de clases e interfaces que proporcionan una arquitectura lista para desarrollar software. Por ende, si se quiere implementar una nueva característica o clase no es necesario definir un nuevo framework si ya existe uno. Es por esto que, una buena práctica en el paradigma orientado a objetos es incluir un framework con una colección de clases tal que todas las clases realicen el mismo tipo de operaciones.

### **Colecciones en Java**

Cualquier grupo de objetos individuales que se representa como una sola unidad se le conoce como una colección de objetos. En Java Development Kit 1.2 por ejemplo, se definió un modelo denominado Collection Framework que contiene todas las clases de colección con sus respectivas interfaces. De ahí que la interfaz `java.util.Collection` y la interfaz de mapa `java.util.Map` son las dos interfaces principales de las clases de colección en Java.

### **Collection Framework en Java**

Antes de que existiera el Collection Framework, es decir, antes de JDK 1.2, los métodos estándar para agrupar objetos en Java, o colecciones, eran Arrays, Vectors o Hashtables. Todas estas colecciones no tenían una interfaz en común.

Por consiguiente, aunque el objetivo principal de todas las colecciones es el mismo, la implementación se definió de forma independiente y en consecuencia no había ninguna relación entre ellas. Además, era muy difícil para los programadores recordar los diferentes métodos, sintaxis y constructores existentes para cada clase de colección.

## Ventajas del Collection Framework en Java

La falta de un framework dio a lugar a las desventajas descritas en la sección anterior. Sin embargo, luego de su declaración se comenzó a observar un panorama favorable.

- **API consistente:** La API tiene un conjunto básico de interfaces como Collection, Set, List o Map, donde todas las clases, ArrayList, LinkedList, Vector, que implementan estas interfaces tienen métodos en común.
- **Reduce la complejidad al programar:** Un programador ya no tiene que preocuparse por el diseño de la Colección, lo cual le permite priorizar el resto de su programa. Es por esto que, la abstracción, uno de los principales aspectos del paradigma orientado a objetos se logró implementar satisfactoriamente.
- **Aumenta la eficiencia y la calidad del programa:** La eficiencia se ve incrementada gracias a la proporción de implementaciones de alto rendimiento para las estructuras de datos junto con algunos algoritmos útiles, ya que, en este caso, el programador no necesita preocuparse por elegir la mejor implementación de una estructura de datos particular. Simplemente puede utilizar la implementación predefinida y así aumentar el rendimiento de su algoritmo/programa.

## Jerarquía del Collection Framework en Java

El paquete `java.util` contiene todas las clases e interfaces que requiere el Collection Framework. Asimismo, el Collection Framework contiene una interfaz conocida como `Iterable`, la cual proporciona un iterador para recorrer todas las colecciones. Todas las colecciones que amplían la interfaz, aumentan al mismo tiempo el rango del iterador y sus métodos, tal como se ilustra en la siguiente imagen.

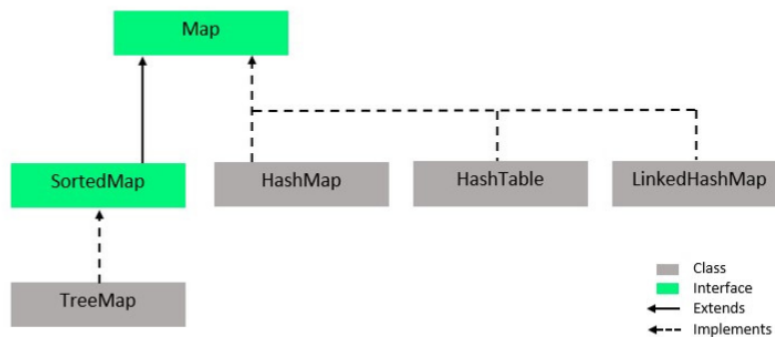


Figura 1.1: GeeksforGeeks. (2023). Jerarquía del Collection Framework en Java [PNG]. GeeksforGeeks <https://media.geeksforgeeks.org/wp-content/cdn-uploads/20200811210521/Collection-Framework-1.png>.

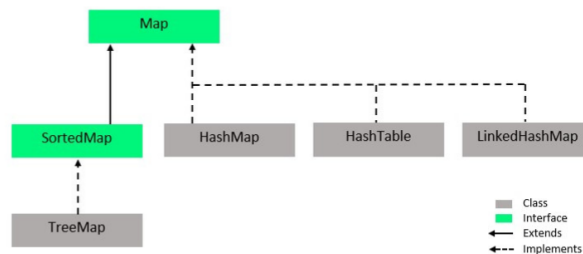


Figura 1.2: GeeksforGeeks. (2023). Jerarquía del Collection Framework en Java [PNG]. GeeksforGeeks <https://media.geeksforgeeks.org/wp-content/cdn-uploads/20200811210611/Collection-Framework-2.png>.

Antes de profundizar en los diferentes componentes del modelo, es importante recordar los conceptos de clase e interfaz.

- **Clase:** Una clase es un modelo o prototipo definido por el usuario a partir del cual se van a crear objetos. En esta, se representa un conjunto de atributos y métodos que serán comunes para todos los objetos de la clase.
- **Interfaz:** Al igual que una clase, una interfaz puede tener métodos y atributos, pero los métodos declarados en una interfaz son abstractos por defecto. Entonces, las interfaces especifican qué debe hacer una clase más no el cómo, es decir, son la plantilla de la clase.

### Clase contra Interfaz

Clase	Interfaz
Una clase es un prototipo definido por el usuario para construir objetos en Java.	Una interfaz es un modelo definido por el usuario que describe la estructura de cada clase que la implementa.
Una interfaz es un modelo definido por el usuario que describe la estructura de cada clase que la implementa.	No se puede utilizar para instanciar objetos.
Una clase puede tener modificadores de acceso públicos y predeterminados.	Una Interfaz puede tener modificadores de acceso públicos y predeterminados.
Las clases pueden ser concretas o abstractas.	Todas las interfaces son abstractas.
Una clase consta de constructores, métodos y atributos. Los métodos están definidos en una clase.	Una interfaz consta de atributos y métodos. Los métodos no están definidos en una interfaz, sólo contiene sus prototipos.

Cuadro 1.1: Comparación entre Clase e Interfaz en Java

### Métodos de la interfaz Collection

Esta interfaz contiene varios métodos que pueden ser utilizados por todas las colecciones que implementan esta interfaz. Los cuales son:

- **add(Object)** se utiliza para agregar un objeto a la colección.
- **addAll(Collection c)** agrega todos los elementos de la colección que se recibe como parámetro a la colección.
- **clear()** elimina todos los elementos de la colección.
- **contains(Object o)** devuelve verdadero si la colección contiene el objeto especificado.
- **containsAll(Collection c)** devuelve verdadero si la colección contiene todos los elementos de la colección que recibe como parámetro.



- **equals(Object o)** compara el objeto especificado con la colección para determinar la igualdad.
- **hashCode()** se utiliza para devolver el valor del código hash para esta colección, es decir, el identificador de 32 bits que se almacena en un Hash en la instancia de la clase.
- **isEmpty()** devuelve verdadero si la colección no contiene elementos.
- **iterator()** devuelve un iterador sobre los elementos de esta colección.
- **max()** se utiliza para devolver el valor máximo presente en la colección.
- **parallelStream()** devuelve un Stream paralelo con esta colección como fuente, dicho de otra manera, este genera un Stream donde cada elemento no depende de otro para ser procesado.
- **remove(Object o)** se utiliza para eliminar el objeto dado de la colección. Sin embargo, si hay valores duplicados, este método elimina la primera aparición del objeto.
- **removeAll(Collection c)** se utiliza para eliminar de la colección todos los objetos contenidos en la colección que recibe como parámetro.
- **removeIf(Predicate filter)** se utiliza para eliminar todos los elementos de esta colección que satisfacen el predicado dado.
- **retainAll(Collection c)** se utiliza para conservar sólo los elementos de la colección que están contenidos en la colección especificada.
- **size()** se utiliza para conocer la cantidad de elementos de la colección.
- **splititerator()** se utiliza para crear un Spliterator sobre los elementos de esta colección, el cual nos permite recorrer y dividir una secuencia de elementos.
- **stream()** devuelve un Stream secuencial con la colección como fuente, el cual permite operar con la colección y hacer que el procesamiento masivo de datos sea rápido y fácil de leer.
- **toArray()** devuelve un array que contiene todos los elementos de la colección.

## Interfaces y clases del Collection Framework

### Interfaz Iterable

Esta interfaz es la raíz de todo el Collection Framework. Como se observa en la imagen, la interfaz Collection amplía la interfaz Iterable. Por lo tanto, todas las interfaces y clases implementan esta interfaz. La función principal de esta interfaz es proporcionar un iterador para las colecciones. De ahí que, esta interfaz contiene sólo un método abstracto que es el iterador.

## Interfaz Collection

Como ya se mencionó, esta interfaz amplía la interfaz Iterable y a su vez amplía la implementación de todas las clases en el Collection Framework. Esta interfaz contiene todos los métodos básicos que tiene cada colección, como agregar datos, eliminar datos o borrarlos. Todos estos métodos se implementan en esta interfaz porque son usados por todas las clases independientemente de su lógica. Además, tener estos métodos en esta interfaz garantiza que los nombres de los métodos sean universales para todas las colecciones. Por consiguiente, esta interfaz construye una base sobre la cual se implementan las clases del resto de interfaces.

## Interfaz List

Esta es una interfaz secundaria de la interfaz de Collection. Se enfoca en las clases que son una colección secuencial en la que el usuario de la interfaz tiene control sobre cualquier elemento que es insertado a la lista. Además, el usuario puede acceder a sus elementos por un índice entero o buscar algún elemento en la lista. Por otra parte, a diferencia de la interfaz Set, la interfaz List si permite que haya elementos repetidos junto con varios elementos nulos. De ahí que esta interfaz está implementada por clases como ArrayList, Vector, Stack y LinkedList. En consecuencia, podemos crear una instancia de un objeto de List con cualquiera de las clases mencionadas.

**Clase ArrayList** Esta clase implementa una lista como un arreglo al que se le puede cambiar el tamaño, junto con todas las operaciones opcionales de la lista e igual permite todos los elementos, incluido null. Asimismo, provee métodos que manipulan el tamaño del arreglo interno usado para almacenar la lista. Por lo que, esta clase es similar a la clase Vector, solo que no está sincronizada, es decir, que múltiples subprocesos pueden operar en un ArrayList al mismo tiempo para hacer un ArrayList seguro para subprocesos. Sin embargo, se puede sincronizar externamente usando el método Collections.synchronizedList().

**Clase Vector** Esta clase implementa una variedad creciente de objetos. Al igual que una matriz, contiene elementos a los que se puede acceder mediante un índice entero. Sin embargo, el tamaño de un Vector puede aumentar o disminuir según se requiera para permitir las operaciones de adición y eliminación de elementos una vez que ya se creó un objeto de la clase. Al igual que la clase ArrayList, esta clase también permite todos los elementos, incluido null. Además, como ya se mencionó anteriormente, la clase Vector si es sincronizada, lo cual implica que sólo un único subproceso puede operar en un método de Vector a la vez.

**Clase Stack** La clase Stack representa una pila de objetos donde el último en entrar, es el primero en salir (LIFO). Por ende, cuando se crea una pila por

primera vez, no contiene elementos. Asimismo, amplía la clase `Vector` con cinco operaciones que permiten tratar un vector como una pila.

- **`empty()`** prueba si la pila está vacía.
- **`peek()`** mira el objeto en la parte superior de la pila sin sacarlo de la pila.
- **`pop()`** elimina el objeto en la parte superior de la pila y devuelve ese objeto como el valor de esta función.
- **`push(E item)`** agrega un elemento a la parte superior de la pila.
- **`search(Object o)`** devuelve la posición basada en 1 de donde se encuentra un objeto en la pila.

**Clase `LinkedList`** Esta clase es la implementación de una lista doblemente ligada de las interfaces `List` y `Deque`. Incluye todas las operaciones de lista opcionales y permite todos los elementos (incluido `null`). Además, todas las operaciones se realizan como se podría esperar en una lista doblemente ligada. Por otro lado, las operaciones que indexan la lista recorrerán la lista desde el principio o el final, de acuerdo con lo que esté más cerca del índice especificado. Hay que tener en cuenta que esta implementación no está sincronizada, es decir, si varios subprocesos acceden a una `LinkedList` al mismo tiempo y al menos uno de los subprocesos modifica la lista estructuralmente, debe sincronizarse externamente.

### Aspectos a tomar en cuenta para elegir alguna de las clases de `List`

Como se vio anteriormente, las clases `ArrayList`, `LinkedList`, `Vector` y `Stack` son todas implementaciones de la interfaz `List`, pero se utilizan en diferentes situaciones debido a sus características y rendimiento. Por ende, conviene tener en cuenta algunos aspectos para saber cuando conviene usar cada una.

- **`ArrayList`:**
  - Se suele utilizar cuando se necesita una lista dinámica que se puede redimensionar y acceder rápidamente a sus elementos con un índice. En consecuencia, la clase `ArrayList` es la elección más común.
  - En cuanto al rendimiento, esta clase ofrece un acceso rápido a elementos por índice, sin embargo, puede ser menos eficiente al insertar o eliminar elementos que se encuentran al medio de la lista debido a que se tienen que desplazar elementos.
- **`LinkedList`:**
  - Se recomienda utilizar cuando vas a insertar o eliminar de manera frecuente elementos al medio de la lista, en este aspecto la clase `LinkedList` puede ser más eficiente que la clase `ArrayList`. También es

útil cuando se requiere iterar sobre la lista en ambas direcciones, es decir del inicio al final o del final al inicio.

- Ahora bien, el rendimiento en cuanto a insertar y eliminar elementos en la clase `LinkedList` es más rápido que en la clase `ArrayList`, pero el acceso aleatorio es más lento.

■ **Vector:**

- Se utiliza con menos frecuencia en las aplicaciones modernas. Es similar a la clase `ArrayList`, pero esta clase si es sincronizada, lo que significa que es segura para el uso de hilos múltiples. Por ello, si necesitas una lista sincronizada, puedes considerar usar la clase `Vector`. De lo contrario, es preferible usar `ArrayList`.
- En lo que al rendimiento se refiere, a causa de la sincronización, la clase `Vector` puede ser más lenta que la clase `ArrayList` en operaciones sin hilos múltiples.

■ **Stack:**

- Se recomienda usar si necesitas una estructura de datos tipo LIFO (Last In, First Out). Sin embargo, es preferible usar la interfaz `Deque` con la clase `LinkedList` o `ArrayDeque` en lugar de la clase `Stack`, ya que el uso de esta clase no se aconseja desde Java 1.5.
- El rendimiento suele ser aceptable para operaciones de apilar y desapilar, no obstante, se debe tener en cuenta que la interfaz `Deque` ofrece mayores opciones y flexibilidad.

## **Interfaz Queue**

La interfaz `Queue` mantiene la lógica FIFO donde el primero en entrar, es el primero en salir, lo cuál es similar a una cola de espera en el mundo real. Por ello, esta interfaz está dedicada a almacenar elementos donde el orden de los elementos sí importa. Por ejemplo, cuando queremos reservar un boleto, los boletos se venden por orden de llegada, entonces, la persona cuya solicitud llega primero a la cola obtiene el billete antes que los demás. En este sentido, hay clases como `PriorityQueue`, `ArrayDeque` y `LinkedList` para implementar la interfaz `Queue`, por lo que, se puede crear una instancia con cualquiera de estas clases.

**Clase `PriorityQueue`** Esta clase permite usar la lógica de una cola de prioridad ilimitada, la cual se basa en un Heap de prioridad. Además, los elementos de la cola de prioridad se ordenan según su orden natural, o de acuerdo con un parámetro `Comparator` proporcionado en el momento de la construcción de la cola, conforme al constructor que se utilice. Lamentablemente, la `PriorityQueue` no permite elementos null. Asimismo, como se basa en el orden natural, tampoco permite la inserción de objetos no comparables, ya que hacerlo puede resultar en `ClassCastException`.

**Interfaz Deque** Esta es una variación muy ligera de la interfaz Queue. Por ende, la interfaz Deque, también conocida como cola doble, es una interfaz con la que podemos agregar y eliminar elementos de ambos extremos de la cola. Como ya se mencionó, la interfaz Deque amplía la interfaz Queue. Por otro lado, las clases que implementan esta interfaz son ArrayDeque y LinkedList, por lo cual, podemos crear una instancia de alguna de estas clases.

**Clase ArrayDeque** Esta clase implementa un array redimensionable con la interfaz Deque que a su vez amplía la interfaz Queue. Un ArrayDeque no tiene restricciones de capacidad, porque crece según sea necesario para soportar el uso. Sin embargo, no son seguros para subprocesos, ya que en ausencia de sincronización externa, no admiten el acceso simultáneo de varios subprocesos. Además, los elementos nulos están prohibidos en esta clase. A pesar de lo anterior, es probable que esta clase sea más rápida que la clase Stack cuando se usa como pila y más rápida que la clase LinkedList cuando se usa como cola.

**Clase LinkedList** Esta clase es la implementación de una lista doblemente ligada de las interfaces List y Deque, la cual a su vez amplía la interfaz Queue. Incluye todas las operaciones de lista opcionales y permite todos los elementos (incluido null). Además, si se usa como una cola todas las operaciones se realizan como se podría esperar. Por otra parte, las operaciones que indexan la cola permiten recorrerla de inicio a fin o viceversa. Nuevamente, se debe tener en cuenta que esta implementación no está sincronizada, por lo cual, si varios subprocesos acceden a una LinkedList al mismo tiempo y al menos uno de los subprocesos modifica la lista estructuralmente, debe sincronizarse externamente.

### Aspectos a tomar en cuenta para elegir alguna de las clases de Queue

De acuerdo con lo ya mencionado, elegir alguna de las clases PriorityQueue, ArrayDeque o LinkedList para implementar la interfaz Queue depende de nuestras necesidades y de las operaciones que planeamos realizar con la estructura de datos cola. Por consiguiente, conviene tener en cuenta algunos aspectos para determinar la clase que más nos convendrá usar para determinada aplicación.

- PriorityQueue:
  - Se recomienda usar cuando necesitas una cola de prioridad, es decir, cuando quieres que los elementos se almacenen y recuperen según un orden específico definido por su prioridad. Por lo anterior, los elementos se recuperan de la cola en orden ascendente o descendente de acuerdo el criterio que se haya definido.
  - En lo que al rendimiento se refiere, la PriorityQueue es eficiente para insertar y eliminar elementos según su prioridad, pero no es eficiente si necesitas acceder a elementos en posiciones arbitrarias o si la prioridad de los elementos cambia con frecuencia.
- ArrayDeque:

- Cuando quieres una cola doble para insertar y eliminar elementos de manera eficiente tanto al principio como al final, entonces la clase `ArrayDeque` es una excelente opción. Por otra parte, un objeto de esta clase se puede usar como una cola FIFO o una pila LIFO según tus necesidades.
  - El rendimiento de la clase `ArrayDeque` es muy eficiente para operaciones de inserción y eliminación en ambos extremos de la cola, pero no brinda un orden basado en una prioridad como la clase `PriorityQueue`.
- `LinkedList`:
- Un objeto de la clase `LinkedList` puede usarse para implementar una cola FIFO o una pila LIFO. Sin embargo, generalmente es menos eficiente que la clase `ArrayDeque` para estas operaciones. A pesar de lo anterior, puedes considerar usar la clase `LinkedList` si necesitas una cola y también si deseas acceder a elementos en posiciones arbitrarias con frecuencia.
  - Respecto al rendimiento, la clase `LinkedList` es menos eficiente que la clase `ArrayDeque` para operaciones de inserción y eliminación en los extremos de la cola, pero es mejor si necesitas acceder a elementos aleatorios.

## Interfaz Set

Un set es una colección desordenada de objetos en la que no se pueden almacenar elementos duplicados, es decir, un set no posee dos elementos `e1` y `e2` tal que se pueda usar el método `e1.equals(e2)`. Esta interfaz se utiliza cuando deseamos evitar la duplicación de los objetos y deseamos almacenar sólo objetos únicos, con un máximo de un elemento nulo, puesto que, esta colección modela la abstracción matemática de un conjunto. Por ello, esta interfaz se puede implementar mediante la creación de una instancia de alguna clase como `HashSet`, `TreeSet` y `LinkedHashSet`.

**Clase `HashSet`** Esta clase implementa la interfaz `Set`, y a su vez está respaldada por una tabla hash, la cual en realidad es una instancia de la clase `HashMap`. Sin embargo, esta clase no ofrece garantía en cuanto al orden de iteración del conjunto; en particular, no garantiza que el orden se mantendrá constante en el tiempo. Asimismo, esta clase sí permite el elemento `null`, al igual que tiene un rendimiento de tiempo constante para las operaciones de inserción y eliminación promedio. Finalmente, hay que tener en cuenta que esta implementación no está sincronizada, dicho de otra forma, si varios subprocesos acceden a un `HashSet` al mismo tiempo y al menos uno de los subprocesos lo modifica, se debe sincronizar externamente. Por lo general, esto se logra mediante la sincronización de algún objeto que encapsule naturalmente el `HashSet`. Si no existe tal objeto, el `HashSet` debe ajustarse utilizando el método `Collections.synchronizedSet`. Es

mejor hacerlo en el momento de la creación, para evitar el acceso accidental y no sincronizado al HashSet.

**Clase `LinkedHashSet`** Esta clase es la implementación de una tabla hash y una lista ligada de la interfaz Set, por lo cual, tiene un orden de iteración predecible. A diferencia de la clase HashSet, la clase LinkedHashSet mantiene una lista doblemente ligada que recorre todas sus entradas. Por lo cual, esta lista doblemente ligada define el orden de iteración, el cual es el orden en el que se insertaron los elementos en el Set. No hay que perder de vista que el orden de inserción no se ve afectado si un elemento se vuelve a insertar en el LinkedHashSet. En consecuencia, esta implementación evita los pedidos no especificados y generalmente caóticos que se generan con la clase HashSet, sin incurrir en el mayor costo asociado con la clase TreeSet. Por lo anterior, se puede utilizar la clase LinkedHashSet para generar una copia de un Set que tenga el mismo orden que el original, independientemente de la implementación del conjunto original.

### Interfaz `SortedSet`

Esta interfaz es muy similar a la interfaz Set. La única diferencia es que esta interfaz tiene métodos adicionales que mantienen el orden de los elementos. Por consiguiente, la interfaz SortedSet amplía la interfaz Set y se utiliza para manejar los datos que deben ordenarse. La clase que implementa esta interfaz es TreeSet, por lo que se puede crear una instancia con esta clase para usar la interfaz SortedSet.

### Interfaz `NavigableSet`

La interfaz NavigableSet se extiende desde la interfaz SortedSet. Además de los métodos de la misma, la interfaz NavigableSet tiene métodos de navegación que brindan coincidencias más cercanas como floor, ceiling, lower y higher. Un NavigableSet se puede recorrer en orden ascendente y descendente. Aunque permite el uso de elementos nulos, no se recomienda, ya que estos elementos pueden generar resultados ambiguos.

**Clase `TreeSet`** Esta clase implementa la interfaz Set, SortedSet y NavigableSet utilizando una estructura de árbol, normalmente un Red-Black Tree, pues se usa para administrar el orden. Por defecto, se tiene un orden ascendente, pero se puede cambiar usando un Comparator. Además, su complejidad temporal asintótica es de  $O(\log n)$  para sus operaciones de inserción, eliminación y búsqueda. Por otra parte, al igual que la clase HashSet esta clase tampoco es sincronizada, por lo que, si se quiere sincronizar se debe hacer de la misma forma que se hace con un objeto de la clase HashSet. Dada la naturaleza de este Set, se tienen algunos métodos exclusivos a la hora de implementarlo.

- **`ceiling(E element)`** devuelve el elemento más pequeño en el conjunto que es mayor o igual al elemento especificado.

- **floor(E element)** devuelve el elemento más grande en el conjunto que es menor o igual al elemento especificado.
- **higher(E element)** devuelve el elemento más pequeño en el conjunto que es estrictamente mayor que el elemento especificado.
- **lower(E element)** devuelve el elemento más grande en el conjunto que es estrictamente menor que el elemento especificado.
- **pollFirst()** recupera y elimina el primer elemento más pequeño en el conjunto.
- **pollLast()** recupera y elimina el último elemento más grande en el conjunto.
- **descendingSet()** devuelve una vista inversa del conjunto, donde los elementos se almacenan en orden inverso.

### Aspectos a tomar en cuenta para elegir alguna de las clases de Set

Para las clases mencionadas anteriormente, se debe definir cuál queremos usar según el problema que se nos presente y que encaje con las características especiales de cada Set. Por ejemplo, en el siguiente problema (B. Sereja and Suffixes Codeforces Problem 368B): Se nos presenta una situación en la que se nos da un conjunto de números que tiene cualquier orden y contiene números repetidos, y se nos da otro conjunto de números que representan índices del anterior conjunto de números brindados, queremos saber la cantidad de números diferentes que se encuentren después de cada índice que se nos ha brindado, además el problema debe ser solucionado en máximo un segundo para entradas de 105. La situación es clara, necesitamos un Set, pues la respuesta será su tamaño después de recorrer la colección de números desde el índice brindado, pero además de esto necesitamos que el Set sea rápido al momento de realizar sus operaciones para cumplir con el límite de un segundo, y la clase de la interfaz Set que cumple con estas características es el HashSet, dado que este tiene las operaciones más rápidas y a su vez cumple con la función, como el resto de clases en la interfaz, de no admitir números repetidos. Por lo anterior, el código en el lenguaje de programación Java queda de la siguiente manera. Cabe destacar que también se uso programación dinámica para calcular todas las respuestas en una sola pasada de la colección, así el índice que se nos proporciona por la segunda colección será el índice que contenga la respuesta en nuestro arreglo de soluciones, en este caso se le resta uno para adecuarse a los índices de los arreglos que son de 0-n, porque los índices de la segunda colección son desde 1-n. Como ya se dijo, la elección entre la clase *HashSet*, *LinkedHashSet* y *TreeSet* para implementar la interfaz *Set* en **Java** depende de nuestros requerimientos junto con el rendimiento y comportamiento que busquemos. Por ende, nos conviene tener en cuenta ciertos aspectos a la hora de elegir alguna de estas clases para nuestra aplicación.



```

import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int t = 1;
        // t = scanner.nextInt();
        for (int i = 0; i < t; i++) {
            solve(scanner);
            System.out.println();
        }
    }

    public static void solve(Scanner scanner) {
        int n = scanner.nextInt();
        int m = scanner.nextInt();
        int num;
        int[] arr2 = new int[m];
        int[] dp = new int[n];
        Set<Integer> s = new HashSet<>();
        int[] arr = new int[n];

        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }

        for (int i = 0; i < m; i++) {
            arr2[i] = scanner.nextInt();
        }

        for (int i = n-1; i >= 0; i--) {
            num = arr[i];
            s.add(num);
            dp[i] = s.size();
        }

        for (int i = 0; i < m; i++) {
            System.out.println(dp[arr2[i] - 1]);
        }
    }
}

```

Figura 1.3: Hernández, G. D. A. (2023). *Solución a B. Sereja and Suffixes Codeforces Problem 368B* [PNG]. Herramienta de Recortes.

#### ■ HashSet:

- Se suele utilizar cuando se necesita una colección de elementos únicos sin tomar en cuenta el orden de inserción ni tampoco un orden específico para los elementos. Asimismo, es la clase más rápida para la mayoría de las operaciones, como inserción, eliminación y búsqueda.
- En lo que al rendimiento se refiere, un objeto de la clase *HashSet* es altamente eficiente para las operaciones mencionadas anteriormente, ya que utiliza una tabla hash para el almacenamiento. Sin embargo, no garantiza ningún orden específico de los elementos.

#### ■ LinkedHashSet:

- Se recomienda utilizar si quieres mantener el orden de inserción de los elementos, y en consecuencia los elementos también se pueden recuperar en el mismo orden en que se insertaron.
- En cuanto al rendimiento la clase *LinkedHashSet* es un poco más lenta que la clase *HashSet* debido a que mantiene el orden de inserción, pero es más rápida que la clase *TreeSet*.

#### ■ TreeSet:

- Se utiliza usualmente cuando necesitamos que los elementos se almacenen y recuperen de acuerdo con el orden natural o con algún orden personalizado. Por consiguiente, la clase *TreeSet* garantiza que los elementos estén ordenados.
- Con relación al rendimiento de la clase *TreeSet*, dado que esta utiliza un *Red-Black Tree* para el almacenamiento, provoca que las operaciones de inserción, eliminación y búsqueda tengan un rendimiento ligeramente más lento que la clase *HashSet* y *LinkedHashSet*. No obstante, tener un orden nos puede ser muy útil para ciertas aplicaciones.

## Interfaz Map

Un Map es una estructura de datos que admite el par clave-valor para mapear los datos. Esta interfaz no admite claves duplicadas porque la misma clave no puede tener múltiples asignaciones; sin embargo, permite valores duplicados en claves diferentes. Un Map es útil cuando existen datos con los que queremos realizar operaciones en base a una clave. Además, la interfaz Map se puede implementar mediante diversas clases como *HashMap*, *Hashtable* y *LinkedHashMap* con la creación de una instancia de alguna de estas.

**Clase *HashMap*** Esta clase es una implementación de la interfaz Map basada en una tabla hash. Por ende, nos proporciona todas las operaciones opcionales de Map y permite valores null junto con claves null. Por otro lado, la clase *HashMap* es similar a la clase *Hashtable*, con la diferencia de que la clase *HashMap* no está sincronizada y permite valores nulos. Lamentablemente, esta clase no ofrece garantía en cuanto al orden del mapa; en particular, no garantiza que el orden se mantendrá constante en el tiempo. Asimismo, esta implementación nos proporciona un rendimiento en tiempo constante para las operaciones básicas get y put, suponiendo que la función hash disperse los elementos adecuadamente entre los depósitos.

**Clase *Hashtable*** Esta clase implementa una tabla hash, que asigna claves a valores. Cualquier objeto no nulo se puede utilizar como clave o como valor. Para almacenar y recuperar objetos de una tabla hash con éxito, los objetos utilizados como claves deben implementar el método *hashCode* y el método *equals*. Consecuentemente, una instancia de la clase *Hashtable* tiene dos parámetros que afectan su rendimiento: *initial capacity* y *load factor*. El *load factor* es la cantidad de depósitos en la tabla hash y la *initial capacity* es simplemente la capacidad en el momento en que se crea la tabla hash. Hay que tener en cuenta que la tabla hash está abierta: en el caso de una colisión, un único depósito almacena varias entradas, que deben buscarse secuencialmente. El *load factor* es una medida de qué tan llena se permite que esté la tabla hash antes de que su capacidad aumente automáticamente. Por lo anterior, los parámetros de *initial capacity* y *load factor* son meros consejos para la implementación.

**Clase `LinkedHashMap`** La clase implementa una tabla hash junto con una lista ligada de la interfaz `Map`, por lo cual, se cuenta con un orden de iteración predecible. Además, esta clase se diferencia de la clase `HashMap` porque mantiene una lista doblemente ligada que recorre todas sus entradas. Esta lista doblemente ligada define el orden de iteración, que normalmente es el orden en que se insertaron las claves en el mapa. Cabe destacar que el orden de inserción no se ve afectado si se vuelve a insertar una clave en el mapa. Asimismo, la clase evita los pedidos no especificados y generalmente caóticos proporcionados por las clases `HashMap` y `Hashtable`, sin incurrir en el mayor costo asociado con `TreeMap`. Por consiguiente, esta clase se puede utilizar para producir una copia de un mapa que tenga el mismo orden que el original, sin importar la implementación del mapa original.

### Interfaz `SortedMap`

La interfaz `SortedMap` amplía la interfaz `Map` con una estipulación adicional de un orden total de claves. Por ello, las claves se ordenan por orden natural o en base a un `Comparator` especificado en el momento de la construcción, de acuerdo con el constructor utilizado. En consecuencia, todas las claves deben ser comparables.

**Clase `Treemap`** Esta clase usa un *Red-Black Tree* basado en la interfaz *NavigableMap* y *SortedMap*, el cual se ordena en el orden natural de las `Keys` o por un comparador definido por el programador. Esta implementación garantiza un costo de tiempo de  $\log(n)$  para las operaciones de búsqueda, obtención, inserción y eliminación de los elementos. Sin embargo, esta clase no está sincronizada. Por otra parte, dada su naturaleza al igual que el *TreeSet* esta clase posee métodos exclusivos que nos permiten acceder a ciertos datos de manera más rápida.

- **`firstKey()`** devuelve la clave del primer elemento en el mapa, que es la clave mínima en orden natural.
- **`lastKey()`** devuelve la clave del último elemento en el mapa, que es la clave máxima en orden natural.
- **`headMap(K toKey)`** devuelve una vista del mapa que contiene todas las entradas cuyas claves son menores que `toKey`.
- **`tailMap(K fromKey)`** devuelve una vista del mapa que contiene todas las entradas cuyas claves son mayores o iguales a `fromKey`.
- **`subMap(K fromKey, K toKey)`** devuelve una vista del mapa que contiene todas las entradas cuyas claves están en el rango `[fromKey, toKey)`, es decir, desde `fromKey` (inclusivo) hasta `toKey` (exclusivo).
- **`comparator()`** devuelve el `Comparator` utilizado para ordenar las claves en el mapa. Si el mapa utiliza el orden natural, este método devolverá `null`.

### Aspectos a tomar en cuenta para elegir alguna de las clases de Map

La elección entre las clases `HashMap`, `Hashtable`, `LinkedHashMap` o `TreeMap` para implementar la interfaz `Map` en Java depende de los requerimientos que se nos soliciten junto con el rendimiento y comportamiento que se busque. Por lo cual, nos conviene tener presentes ciertos aspectos al momento de elegir alguna de estas clases para nuestra aplicación.

- `HashMap`:
  - Se recomienda utilizarlo cuando se necesite una estructura de datos de mapeo clave-valor con acceso rápido a los elementos. Es la elección predeterminada en la mayoría de los casos, ya que ofrece un buen equilibrio entre rendimiento y funcionalidad.
  - El rendimiento de esta clase es altamente eficiente para la mayoría de las operaciones, como inserción, eliminación y búsqueda, pero no garantiza ningún orden específico de las claves.
- `Hashtable`:
  - Aunque la clase `Hashtable` es similar a la clase `HashMap` en términos de funcionalidad, se utiliza con menos frecuencia en aplicaciones modernas debido a su sincronización, que puede degradar el rendimiento en entornos multihilo. Por ello, si necesitas un mapa sincronizado, thread-safe, puedes optar por la clase `Hashtable`. De lo contrario, la clase `HashMap` es preferible.
  - El rendimiento de la clase `Hashtable` es menor que el de la clase `HashMap` debido a la sincronización. Por ende, si no requieres de la sincronización, la clase `HashMap` será más rápida.
- `LinkedHashMap`:
  - Si deseas mantener el orden de inserción de las claves o el orden en el que se accede a los elementos, se recomienda utilizar la clase `LinkedHashMap`.
  - El rendimiento de la clase `LinkedHashMap` es ligeramente más lento que el de la clase `HashMap`, ya que debe realizar un seguimiento del orden de inserción o acceso. Sin embargo, es más rápido que el de la clase `TreeMap`.
- `TreeMap`:
  - Se recomienda utilizar la clase `TreeMap` cuando requieras que las claves sean almacenadas de acuerdo al orden natural o de acuerdo con un orden personalizado. Por lo anterior, esta clase garantiza un orden ascendente o descendente de las claves.

- El rendimiento de la clase `TreeMap` es menos eficiente que el de las clases `HashMap` y `LinkedHashMap` en la mayoría de las operaciones debido a la estructura de árbol utilizada para mantener el orden. No obstante, ofrece la ventaja de un acceso ordenado a las claves.

## EJERCICIOS DEL PROYECTO

### Búsqueda Lineal y Binaria en listas de objetos

#### 1. Elaboración de Clases

- Elabora las clases necesarias donde definas los atributos y métodos de un objeto para crear el tipo de dato Alumno y el tipo de dato Asignatura. Asegúrate de considerar lo siguiente:
  - Atributos privados.
  - Acceso a los atributos a través de getters y setters.
  - Constructor con parámetros.

#### 2. Clase Búsqueda Lineal

- Elabora una clase llamada BúsquedaLineal similar a la práctica 4, en la cual se implementarán métodos que devuelvan objetos y que reciban los siguientes parámetros:
  - a) Nombre (búsqueda por nombre del alumno).
  - b) Clave (búsqueda por clave de la asignatura).

#### 3. Clase Principal

- En la clase principal del proyecto, crea una lista de asignaturas y una lista de alumnos. Luego, comprueba el funcionamiento de los métodos realizados para las búsquedas.

#### 4. Búsqueda Binaria (Opcional)

- Para obtener puntos extras en el proyecto, puedes realizar la búsqueda binaria. Deberás investigar cómo ordenar la lista de alumnos o la

lista de asignaturas a partir de alguno de sus atributos y posteriormente aplicar el algoritmo de búsqueda binaria.

El algoritmo de búsqueda lineal en su esencia es sencillo, solo consta de recorrer toda la colección en busca de los elementos que coincidan con lo que buscamos, lo difícil fue la parte de implementación en el lenguaje Java pues se requirió de investigar y conocer cómo hacer distintas adaptaciones necesarias para la correcta ejecución del programa. Para este ejercicio se crearon 2 paquetes, el paquete Escuela, el cual tiene las clases de Asignatura y Alumno lo que nos dará la posibilidad de crear los objetos necesarios para ser insertados en una lista y después buscarlos. Y el paquete Búsquedas que poseen la búsqueda lineal y la búsqueda binaria.

### Descripción de clases

**Clase Alumno:** Esta clase implementa el Comparable esto para definir que los objetos creados son comparables y así admitir el método CompareTo. La clase posee 3 atributos un contador estático que determinará cuántos objetos Alumno hay, un String Alumno que contendrá el nombre del alumno y un entero en cuenta que poseerá el número de cuenta del alumno, así diferenciando los objetos que posean el mismo nombre; dos constructores uno que inicializa todos los atributos y otro que solo inicializa el atributo nombre, este se utilizará posteriormente para la búsqueda binaria, posee métodos de acceso, un método de clase que sirve para llenar los datos del alumno, crea el objeto y lo devuelve para ser guardado en la lista; y 3 métodos sobrescritos, el ToString que contendrá los formatos de impresión del objeto, un método equals que comprueba que el atributo nombre sea igual al nombre que buscamos (esto para la búsqueda lineal), y un método compareTo que nos servirá para poder ordenar la lista y determinar si un objeto es mayor o menor que otro según qué atributo que nos servirá para crear la búsqueda binaria.

**Clase Asignatura:** Esta clase implementa el Comparable esto para definir que los objetos creados son comparables y así admitir el método CompareTo. La clase posee tres atributos un entero de clase contador que contendrá la cantidad de objetos creados, un String de instancia que contendrá el nombre de la asignatura y un entero de instancia que contendrá la clave de la asignatura; posee también sus métodos de acceso; dos constructores, uno para inicializar todos los atributos y otro para inicializar solo su clave (lo que nos servirá para la búsqueda binaria); un método de clase que sirve para instanciar el objeto, devolviéndolo ya con todos los datos solicitados; 3 métodos sobrescritos, el método ToString que maneja el formato de impresión del objeto y sus atributos, el método equals que servirá en la búsqueda lineal para comparar las claves y determinar si el objeto comparte la misma clave o no, y el método CompareTo que nos servirá para poder determinar cuando un objeto es mayor que otro según el atributo elegido para comparar, esto nos servirá para la búsqueda binaria.

**Clase Main:** Esta clase es el menú donde se ejecutarán todas las opciones, este menú contiene dos Linked List una de alumnos y otra de asignaturas para tener guardados todos los objetos en los que se buscará, posteriormente se despliega un menú que muestra todas las opciones disponibles:

```

1      do{
2          Utilerias.clearScreen();
3          System.out.println("Bienvenido al sistema de
busqueda de alumnos");
4          System.out.println("Escoge una opcion");
5          System.out.println("1.-Agregar Alumno");
6          System.out.println("2.-Agregar materia");
7          System.out.println("3.-Imprimir Listas");
8          System.out.println("4.-Busqueda lineal");
9          System.out.println("5.-Busqueda Binaria");
10         s=cin.nextInt();
11         switch (s) {
12             case 1: a.add(Alumno.crear());break;
13             case 2: m.add(Asignatura.crear()); break;
14             case 3: System.out.println("1.-Lista Alumnos 2.-
Lista Materias"); s=cin.nextInt(); if(s==1){Imprimir(a);}else{
Imprimir(m);} ; break;
15             case 4: BL.menu(a, m);break;
16             case 5: BB.menu(a, m);break;
17             default:
18                 return;
19         }
20     }while(true);

```

Figura 2.1: Menú de opciones

## Búsqueda Lineal

Para la búsqueda lineal, en su clase se crearon los atributos privados:

```

1      private boolean rb;
2      private LinkedList<Integer> ind;
3      private int c=0;

```

Figura 2.2: Atributos clase búsqueda lineal

Estos atributos nos servirán para guardar los datos que necesitamos, es decir, si el objeto es encontrado en rb se guardará un True, en la LinkedList se guardarán todas las posiciones en las que el objeto es encontrado y en el entero se guardarán la cantidad de veces que el elemento haya sido encontrado.



El método que realiza la búsqueda llamado BLB, devuelve un objeto de tipo BL, esto porque el objeto BL será aquel que contenga los atributos mencionados y los mostrará como salida en el programa.

El método BLB, recibe un Object y una List, es decir, recibe dos tipos genéricos uno de objeto y otro de lista, su propósito es que este mismo método pueda recibir una lista de Asignaturas o una lista de Alumnos y el objeto genérico recibe como objeto el atributo que se va a comparar para hacer la búsqueda, en caso de un Alumno recibirá un String y en caso de Asignatura recibirá un Integer. La parte esencial del código es la siguiente:

```
1 for (Object i : list) {
2     if (i.equals(v)) {
3         b = true;
4         count++;
5         index.add(j);
6     }
7     j++;
8 }
9 BL OBL = new BL(b, index, count);
10 return OBL;
```

Figura 2.3: Algoritmo de búsqueda Lineal

Esta parte del código llena los atributos que posteriormente construirán el objeto mencionado para finalmente devolverlo: Algo a destacar es el método equals, pues nos sirve para comparar el atributo con lo que estemos buscando. Para usarlo fue necesario sobrescribir este método tanto en la clase de Asignatura como en la Clase Alumno quedando de la siguiente manera.

```
1 @Override
2 public boolean equals(Object obj) {
3     return this.getAlumno().equals((String)obj);
4 }
```

Figura 2.4: Método sobrescrito de Alumno: equals

Cómo se puede observar, recibe un objeto genérico, para este caso se sabe que lo que compararemos será el nombre del alumno, es decir, un String, por lo que el objeto genérico se castea para convertirse a un objeto de tipo String, una vez esto, se utiliza el método equals de la clase String para determinar si ambos Strings son iguales; el método sobrescrito solo hace que lo que se evalúe sea el atributo nombre del objeto, para esto el getter getAlumno devuelve el nombre del alumno el cual es una String, este String tiene su propio método equals para determinar si dos Strings son iguales por lo que se usa con el objeto genérico casteado/convertido a una String si son iguales el método devolverá un true y si no un false. Esta misma lógica se aplica para el equals de Asignatura

únicamente que aquí casteamos el objeto a un Integer y lo pasamos a entero para que pueda ser comparado de la manera usual: Una vez esto en la clase de

```
1  @Override
2  public boolean equals(Object obj) {
3      Integer entero = (Integer) obj;
4      return this.clave == entero.intValue();
5  }
```

Figura 2.5: Método sobrescrito de Asignatura: equals

la búsqueda lineal se tiene un método de clase llamado menú, el cual recibirá las dos listas, tanto la de alumnos como la de asignaturas y se crearán los dos genéricos (lista y objeto), una vez esto se le solicita al alumno que escoja que quiere buscar si escoge alumno la lista genérica tomará a la lista de alumnos y el objeto genérico recibe un String, si escoge Asignatura la lista genérica tomara la lista de asignaturas y el objeto recibirá un Integer. Después de esto se ejecuta el método BLB explicado anteriormente y el objeto devuelto será guardado en un objeto de tipo BL(búsqueda lineal) llamado OBL(Objeto Búsqueda Lineal).

Ahora el usuario tendrá la posibilidad de escoger que quiere mostrar

1. Búsqueda Booleana (Encontrado, No encontrado)
2. Índice encontrado (Los índices de la lista en donde se localiza el objeto)
3. Cuántas veces fue encontrado (Mostrará la cantidad de incidencias que tenga el objeto)
4. Imprime todos los datos.(Mostrara el número de veces, los índices y los objetos en cada índice pues estos pueden tener sus demás atributos diferentes)

**Método sistema de impresión:** Este método es interesante pues es un método de instancia, el cuál recibirá la lista original y utilizará los atributos llenados en el objeto para ir imprimiendo todos los índices de los objetos encontrados, recibe una lista genérica por lo que puede usarse para imprimir cualquier objeto tanto Alumno como Asignatura y emplea el método ToString para imprimir todos los atributos del objeto localizado.

## Búsqueda Binaria:

Para la búsqueda binaria no se empleó ningún atributo de instancia o clase, pues todas las respuestas son generadas por los métodos de clase creados, solo tiene un atributo de clase que es el scanner que se usará para el registro de datos.

**Métodos CompareTo():** Estos métodos son esenciales para que la ejecución del algoritmo de búsqueda binaria funcione, pues es lo que determina cuando un objeto es mayor o menor que otro caso, para esto se implementó la interfaz Comparable que habilita el método CompareTo para sobrescribirlo según lo que queramos evaluar. Para la clase Alumno quedó sobrescrito de tal manera que solo evalúa el atributo que contiene el nombre del alumno: Se usa el getter para

```
1  @Override
2  public int compareTo(Alumno A2) {
3      return this.getAlumno().compareTo(A2.getAlumno());
4  }
```

Figura 2.6: Método sobrescrito de Alumno: CompareTo

obtener el String y usar el propio compareTo del string con el string del otro objeto a recibir, así retornado el resultado que puede ser positivo para mayor que, negativo para menor que y cero para valores iguales. Para la clase Asignatura quedo sobrescrito de tal manera que solo use la clave para compararlo:

```
1  @Override
2  public int compareTo(Asignatura A2) {
3      return Integer.compare(this.getClave(), A2.getClave());
4  }
```

Figura 2.7: Método sobrescrito de Asignatura: CompareTo

Utilizando el compare de la clase Integer para comparar los dos enteros, devolviendo así un valor negativo para menor que, un valor positivo para mayor que y cero para valores iguales.

**Método BBI (Búsqueda Binaria index)** Este método devolverá un entero y usa `<T extends Comparable <T>>` para definir un objeto genérico comparable, pues recibirá dos parámetros de ese tipo un objeto genérico comparable `v` y una lista de objetos comparables `list`, esta lista cabe a destacar que debe estar ordenada, para esto se definió a todas las clases cómo comparables (lo anteriormente mencionado) y se reescribieron los métodos de `CompareTo`, en el caso de `Asignatura` se reescribió de tal manera que solo compara las claves de ambos objetos, y para el caso de `Alumno` solo compara el nombre de cada alumno, así al usar `Collections.Sort()` este método se basará en los métodos `CompareTo` sobrescritos para ir ordenando las colecciones. El método inicializa una variable `b` en `-1` pues esta es la variable que registra el índice por lo que si no lo encuentra devolverá un `-1` que indicará que no hay tal objeto buscado, si la lista está vacía retorna `-1`. Después define una variable `mid` el cual es el valor que se encuentra en medio de la lista y empieza el algoritmo, se obtiene el objeto en el `index mid` y a este se le hace un `CompareTo` con el objeto genérico que fue recibido de la siguiente manera: Este método lo que hace es regresar un valor negativo si el

```

1  if(v.compareTo(list.get(mid))==0){
2      b=mid;
3      return b;
4  }else if(v.compareTo(list.get(mid))>0){
5      aux+=BBI(v, list.subList(mid+1, list.size()));
6      if(aux==-1){
7          return -1;
8      }else{
9
10         b=mid+1+aux;
11     }
12 }else if(v.compareTo(list.get(mid))<0){
13     b=0;
14     b+=BBI(v, list.subList(0,mid));
15 }

```

Figura 2.8: Método sobrescrito de `Asignatura`: `CompareTo`

objeto es menor, un valor positivo si es mayor y cero si los valores son iguales, por lo que así define cómo se va partiendo la lista en cada llamada recursiva.

Para cada llamada recursiva se va definiendo nuevamente `b` para ir alineando el `index` que debe llevar según cómo se parta la lista, en caso de que la lista vaya hacia los índices después del valor central se usa una variable auxiliar que evalúa si el valor fue encontrado para que la variable que registre el índice se ajuste y si no, devuelva `-1` sin el ajuste del índice, pues no es necesario ya que no encontró el valor.

Al final devuelve el `index` en el que encontró el elemento, este elemento puede ser el de en medio, la primera incidencia o la última incidencia, por lo que, para encontrarlas todas se usa el método `EncontrarIndicesdetodaslasincidencias`.

**Método Encontrar Índices de todas las incidencias:** Este método devuelve una lista de enteros, recibe un objeto genérico comparable, una lista de objetos genéricos comparables y el índice generado por el método de búsqueda binaria; solo funciona si se encontró por lo menos una incidencia en el método de búsqueda binaria, es decir, si el valor regresado por el método de es mayor o igual que cero. De lo contrario, devolverá la lista vacía. Se crea una LinkedList llamada index. Añade el índice recibido como parámetro a index y define dos enteros lb (límite izquierdo) y rbound (límite derecho) los cuales se mueven desde donde está el índice encontrado, hacia la izquierda en caso de lb y hasta la derecha en caso de rb así hasta que los índices se topen con un valor distinto al que se busca para esto se utilizan los métodos compareto ya mencionados anteriormente, cada que encuentra un elemento igual añade la variable de límite a la lista index pues estas variables representan los índices de la lista, así hasta que ambos límites se hayan encontrado con un valor distinto al buscado o se hayan llegado a los límites de la lista. Retorna la lista index la cual contiene todos los índices localizados.

**Menú:** El método recibe dos listas, la lista de alumnos y la lista de asignaturas, define un objeto genérico v una linkedlist de objetos genéricos comparables, sortea ambas listas con ayuda del collections.sort(). Le solicita al usuario que decidirá buscar, en caso de que busque un alumno por nombre la lista de alumnos se casteará a (LinkedList<T>) y se asignará a la lista genérica, aquí es donde entra el constructor de que solo inicializa el nombre pues, se necesitará que este objeto genérico contenga un objeto del mismo tipo que se va a buscar, pues el método compareto solo admite comparaciones con objetos del mismo tipo, pero solo inicializamos el nombre porque es el único atributo que consideraremos, en caso de que el usuario escoja buscar asignatura por nombre realiza lo mismo pero con el constructor de asignatura. Una vez esto, imprime la lista ordenada y ejecuta el método de ordenamiento por búsqueda binaria, devolviendo el primer índice encontrado, posteriormente se crea una lista de enteros denominada Index que será la lista que devolverá el método que encontrará las demás incidencias. Ahora el usuario tendrá la posibilidad de escoger que quiere mostrar

1. Búsqueda Booleana (Encontrado, No encontrado)
2. Índice encontrado (Los índices y los objetos en cada índice pues estos pueden tener sus demás atributos diferentes)
3. Cuántas veces fue encontrado (Mostrara el numero de veces que fue encontrado)
4. Imprime todos los datos.(Mostrara el numero de veces, los índices y los objetos en cada índice pues estos pueden tener sus demás atributos diferentes)

**Método sistema de impresión:** Este método recibe la lista de índices y una lista de genéricos para así imprimir todos los índices que contienen tal objeto y

el objeto que se encuentra en ese índice utilizando el método toString.

Las búsquedas binaria y lineal requirieron de su saber su funcionamiento teórico para poder realizar la implementación correctamente y dando los datos necesarios para que los algoritmos fueran útiles, la estrategia se basó en consular la información sobre su funcionamiento e investigar y adaptar la implementación, las principales dificultades fueron en la elaboración del Binary Search pues el uso de la recursividad complica las cosas cuando se requiere mantener un orden como es el caso de mantener el índice por cada iteración, para la búsqueda binaria si se implementaba de manera iterativa hubiera sido más sencillo manejarlo también como un objeto y funcionase igual a Búsqueda Lineal al momento de recolectar los datos, pero se decidió por la forma recursiva para implementar una solución distinta. Se concreto el ejercicio en un 100 % sin muchas dificultades y relacionando los conceptos teóricos con la implementación correctamente.

### Ejecución:

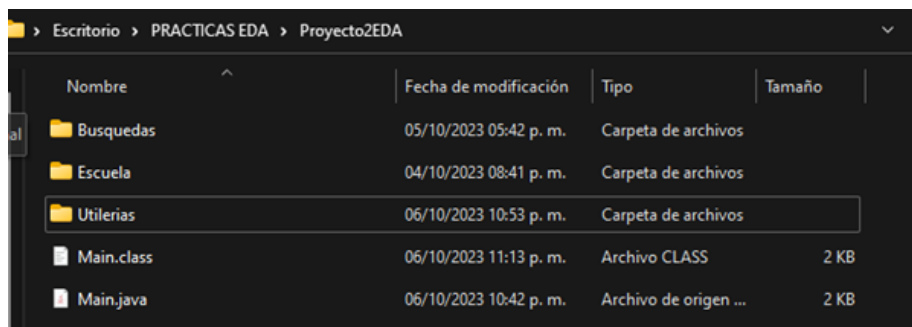


Figura 2.9: Paquetes y Clase Main:

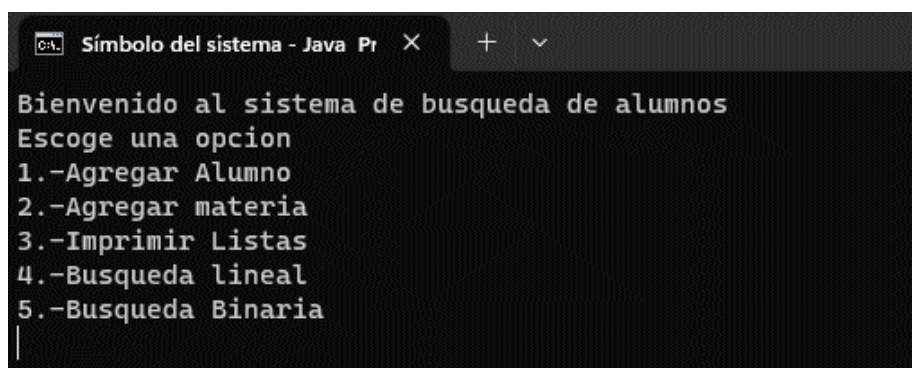


Figura 2.10: Menú

```
Bienvenido al menu de creacion de alumno
Dame el nombre del alumno
Daniel
Dame su numero de cuenta
42311|
```

Figura 2.11: Creación de Alumno Daniel

```
Bienvenido al menu de creacion de alumno
Dame el nombre del alumno
Daniel
Dame su numero de cuenta
1122|
```

Figura 2.12: Creación Alumno Daniel 2

```
Bienvenido al menu de creacion de alumno
Dame el nombre del alumno
Isaac
Dame su numero de cuenta
441122|
```

Figura 2.13: Creación Alumno Isaac

```
Bienvenido al menu de creacion de alumno
Dame el nombre del alumno
Sebastian
Dame su numero de cuenta
44411|
```

Figura 2.14: Creación Alumno Sebastián

```
Bienvenido al menu de creacion de alumno  
Dame el nombre del alumno  
Alonso  
Dame su numero de cuenta  
556611|
```

Figura 2.15: Creación Alumno Alonso



```
1.-Lista Alumnos 2.-Lista Materias
1
Alumno[
alumno= Daniel
ncuenta= 42311
]
Alumno[
alumno= Daniel
ncuenta= 1122
]
Alumno[
alumno= Isaac
ncuenta= 441122
]
Alumno[
alumno= Sebastian
ncuenta= 44411
]
Alumno[
alumno= Alonso
ncuenta= 556611
]
Press Any Key To Continue...
```

Figura 2.16: Impresión

```
Asignatura
Dame el nombre de la asignatura
Calculo
Dame la clave de la asignatura
1111|
```

Figura 2.17: Creación asignatura Cálculo

```
Asignatura
Dame el nombre de la asignatura
Algebra
Dame la clave de la asignatura
1111|
```

Figura 2.18: Creación Asignatura Álgebra

```
Asignatura
Dame el nombre de la asignatura
Mecanica
Dame la clave de la asignatura
2222|
```

Figura 2.19: Creación materia Mecánica

```
Asignatura
Dame el nombre de la asignatura
Quimica
Dame la clave de la asignatura
3333|
```

Figura 2.20: Creación materia Química

```
Asignatura
Dame el nombre de la asignatura
EDA
Dame la clave de la asignatura
4444|
```

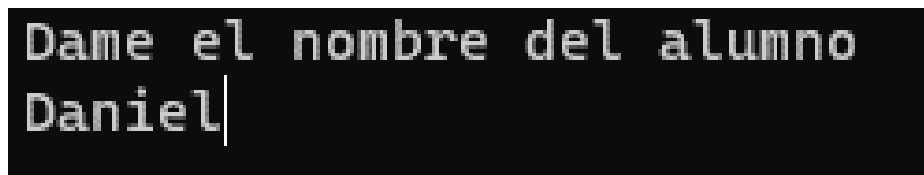
Figura 2.21: Creación materia EDA

```
Asignatura:[Nombre= Calculo  
clave= 1111  
]  
Asignatura:[Nombre= Algebra  
clave= 1111  
]  
Asignatura:[Nombre= Mecanica  
clave= 2222  
]  
Asignatura:[Nombre= Quimica  
clave= 3333  
]  
Asignatura:[Nombre= EDA  
clave= 4444  
]  
Press Any Key To Continue...  
|
```

Figura 2.22: Impresión de las Asignaturas

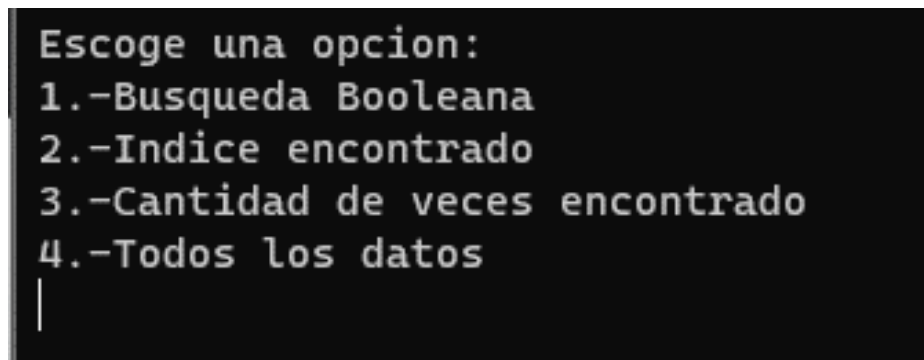
```
Que quieres buscar?  
1.-Alumno por nombre  
2.- Materia por clave  
|
```

Figura 2.23: Menú Búsqueda Lineal



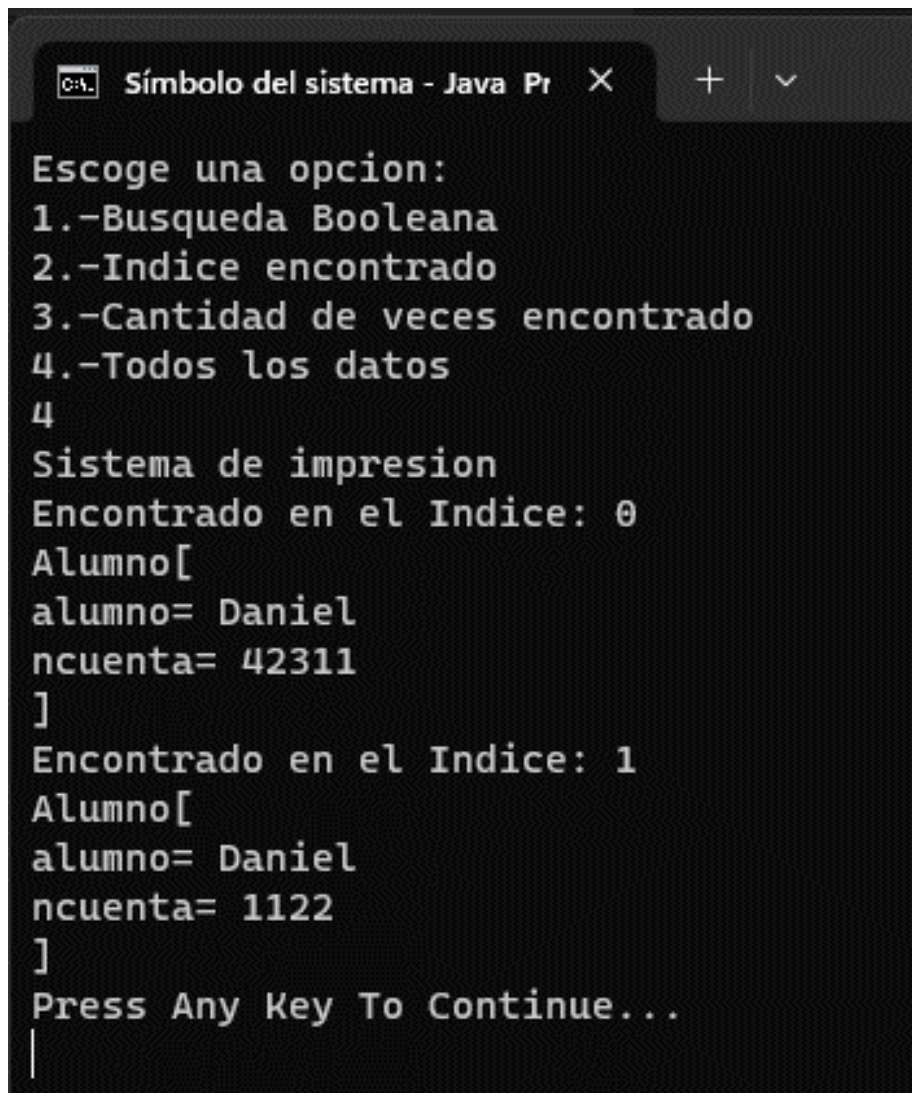
```
Dame el nombre del alumno  
Daniel|
```

Figura 2.24: Solicitud de Búsqueda por nombre a Daniel



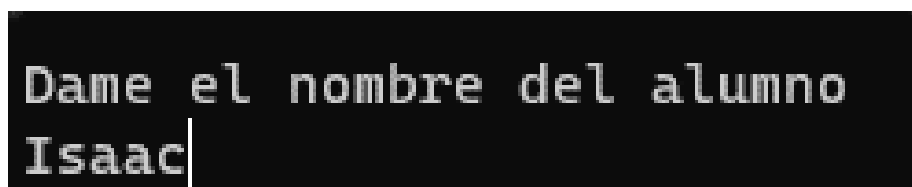
```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
|
```

Figura 2.25: Menú 2 Búsqueda Lineal



```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
Sistema de impresion  
Encontrado en el Indice: 0  
Alumno[  
alumno= Daniel  
ncuenta= 42311  
]  
Encontrado en el Indice: 1  
Alumno[  
alumno= Daniel  
ncuenta= 1122  
]  
Press Any Key To Continue...  
|
```

Figura 2.26: Impresión de todos los datos



```
Dame el nombre del alumno  
Isaac|
```

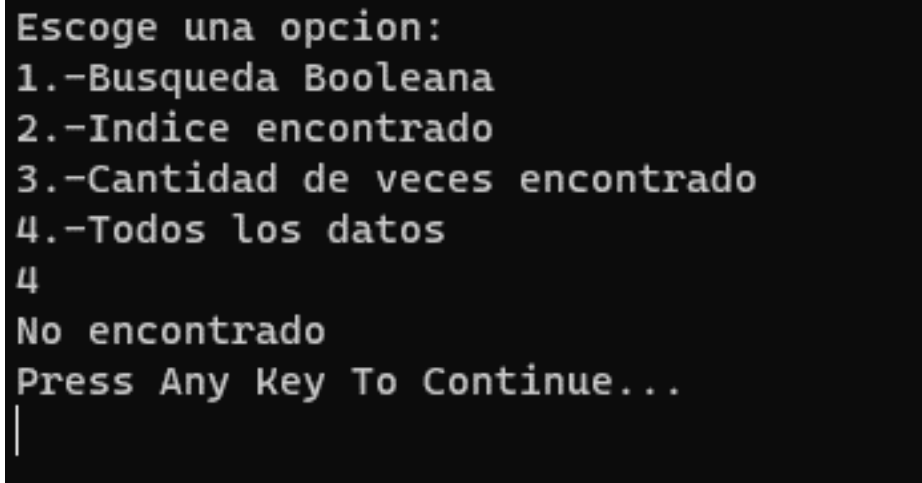
Figura 2.27: Solicitud de búsqueda por nombre de Isaac

```
Sistema de impresion
Encontrado en el Indice: 2
Alumno[
  alumno= Isaac
  ncuenta= 441122
]
Press Any Key To Continue...
```

Figura 2.28: Impresión de todos los datos


```
Dame el nombre del alumno
Inexistente|
```

Figura 2.29: Búsqueda Lineal por nombre de alumno inexistente



```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
No encontrado  
Press Any Key To Continue...  
|
```

Figura 2.30: Impresión de todos los datos



```
Dame la clave de la materia  
1111|
```

Figura 2.31: Búsqueda de materia con Clave 1111



```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
4
Sistema de impresion
Encontrado en el Indice: 0
Asignatura:[Nombre= Calculo
clave= 1111
]
Encontrado en el Indice: 1
Asignatura:[Nombre= Algebra
clave= 1111
]
Press Any Key To Continue...
|
```

Figura 2.32: Impresión de todos los datos

```
Dame la clave de la materia
2222|
```

Figura 2.33: Búsqueda de materia con Clave 2222

```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
Sistema de impresion  
Encontrado en el Indice: 2  
Asignatura:[Nombre= Mecanica  
clave= 2222  
]  
Press Any Key To Continue...  
|
```

Figura 2.34: Impresión de todos los datos

```
Dame la clave de la materia  
0000|
```

Figura 2.35: Búsqueda de una clave inexistente

```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
No encontrado  
Press Any Key To Continue...  
|
```

Figura 2.36: Impresión de todos los datos

**Búsqueda binaria:** Usando los mismos datos realizaremos las mismas búsquedas por el algoritmo de búsqueda binaria.

```
Bienvenido al sistema de busqueda de alumnos
Escoge una opcion
1.-Agregar Alumno
2.-Agregar materia
3.-Imprimir Listas
4.-Busqueda lineal
5.-Busqueda Binaria
5
Que quieres buscar?
1.-Alumno por nombre
2.- Materia por clave
|
```

Figura 2.37: Menú Búsqueda Binaria

```

Símbolo del sistema - Java Pr X + v
Dame el nombre del alumno
Daniel|
```

Figura 2.38: Solicitud del nombre a buscar

```
Lista ordenada para Binary Search
Alumno[
alumno= Alonso
ncuenta= 556611
]
Alumno[
alumno= Daniel
ncuenta= 42311
]
Alumno[
alumno= Daniel
ncuenta= 1122
]
Alumno[
alumno= Isaac
ncuenta= 441122
]
Alumno[
alumno= Sebastian
ncuenta= 44411
]
Press Any Key To Continue...
|
```

Figura 2.39: Impresión de la lista ordenada

```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
```

Figura 2.40: Menú de impresión de datos Binary Search

```
Sistema de impresion
Encontrado en el Indice: 2
Alumno[
alumno= Daniel
ncuenta= 1122
]
Encontrado en el Indice: 1
Alumno[
alumno= Daniel
ncuenta= 42311
]
Press Any Key To Continue...
```

Figura 2.41: Impresión de todos los datos

```
Dame el nombre del alumno
Isaac|
```

Figura 2.42: Búsqueda por nombre de Isaac

```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
4
Sistema de impresion
Encontrado en el Indice: 3
Alumno[
  alumno= Isaac
  ncuenta= 441122
]
Press Any Key To Continue...
|
```

Figura 2.43: Impresión de todos los datos

```
Dame el nombre del alumno
Inexistente|
```

Figura 2.44: Búsqueda Alumno inexistente

```
Escoge una opción:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
No encontrado  
Press Any Key To Continue...  
|
```

Figura 2.45: Impresión de todos los datos

```
Dame la clave de la materia  
1111|
```

Figura 2.46: Búsqueda binaria de materia por clave



```
Lista ordenada para Binary Search
Asignatura:[Nombre= Calculo
clave= 1111
]
Asignatura:[Nombre= Algebra
clave= 1111
]
Asignatura:[Nombre= Mecanica
clave= 2222
]
Asignatura:[Nombre= Quimica
clave= 3333
]
Asignatura:[Nombre= EDA
clave= 4444
]
Press Any Key To Continue...
|
```

Figura 2.47: Lista Ordenada por claves

```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
4
Sistema de impresion
Encontrado en el Indice: 1
Asignatura:[Nombre= Algebra
clave= 1111
]
Encontrado en el Indice: 0
Asignatura:[Nombre= Calculo
clave= 1111
]
Press Any Key To Continue...
|
```

Figura 2.48: Impresión de todas las coincidencias

```
Dame la clave de la materia
4444|
```

Figura 2.49: Búsqueda binaria de la clave 4444

```
Escoge una opcion:  
1.-Busqueda Booleana  
2.-Indice encontrado  
3.-Cantidad de veces encontrado  
4.-Todos los datos  
4  
Sistema de impresion  
Encontrado en el Indice: 4  
Asignatura:[Nombre= EDA  
clave= 4444  
]  
Press Any Key To Continue...  
|
```

Figura 2.50: Impresión de todos los datos

```
Dame la clave de la materia  
0000|
```

Figura 2.51: Búsqueda clave inexistente

```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
4
No encontrado
Press Any Key To Continue...
|
```

Figura 2.52: Impresión de todos los datos

```
Bienvenido al sistema de busqueda de alumnos
Escoge una opcion
1.-Agregar Alumno
2.-Agregar materia
3.-Imprimir Listas
4.-Busqueda lineal
5.-Busqueda Binaria
2
Asignatura
Dame el nombre de la asignatura
POO
Dame la clave de la asignatura
0000|
```

Figura 2.53: Inserción de la materia inexistente para verificación ordenamiento de la lista por atributo clave

```
1.-Lista Alumnos 2.-Lista Materias
2
Asignatura:[Nombre= Calculo
clave= 1111
]
Asignatura:[Nombre= Algebra
clave= 1111
]
Asignatura:[Nombre= Mecanica
clave= 2222
]
Asignatura:[Nombre= Quimica
clave= 3333
]
Asignatura:[Nombre= EDA
clave= 4444
]
Asignatura:[Nombre= P00
clave= 0
]
Press Any Key To Continue...
```

Figura 2.54: Impresión de la lista antes de ordenarla

```
Dame la clave de la materia
0000|
```

Figura 2.55: Búsqueda de la clave recién insertada

```
Lista ordenada para Binary Search
Asignatura:[Nombre= P00
clave= 0
]
Asignatura:[Nombre= Calculo
clave= 1111
]
Asignatura:[Nombre= Algebra
clave= 1111
]
Asignatura:[Nombre= Mecanica
clave= 2222
]
Asignatura:[Nombre= Quimica
clave= 3333
]
Asignatura:[Nombre= EDA
clave= 4444
]
Press Any Key To Continue...
|
```

Figura 2.56: Lista ordenada

```
Escoge una opcion:
1.-Busqueda Booleana
2.-Indice encontrado
3.-Cantidad de veces encontrado
4.-Todos los datos
4
Sistema de impresion
Encontrado en el Indice: 0
Asignatura:[Nombre= P00
clave= 0
]
Press Any Key To Continue...
|
```

Figura 2.57: Impresión de todos los datos

## Simulación de Encadenamiento

Elabora un nuevo programa llamado Encadenamiento, en el cual deberás realizar lo siguiente:

1. Crear una lista de 15 listas de enteros (con índices de 0 a 14).
2. Elabora un menú de usuario para que al ejecutar el programa se pueda elegir:
  - Agregar elemento
  - Mostrar Lista de listas
3. Cada vez que el usuario decida agregar un elemento, una función aleatoria seleccionará un número entre 0 y 14 y en la lista que corresponda a esa posición se insertará el numero indicado por el usuario.

El encadenamiento es una estrategia de solución de colisiones, para cada índice del arreglo se construye una lista ligada para insertar los valores cuyo hash resulte en ese índice, por lo tanto aquellos valores que colisionan se insertan en una misma lista ligada. En este método el proceso de búsqueda consiste de dos pasos, primero se utiliza la función hash para buscar el índice correspondiente y luego se recorre la lista ligada para verificar si el elemento está en la tabla. Se debe utilizar una tabla suficientemente grande y una función hash que distribuya los elementos de manera uniforme para garantizar que el tamaño de las listas ligadas sea el mínimo posible.

La principal desventaja de esta estrategia es que si se ingresa una serie de elementos donde todos resultan en el mismo hash todos quedarán en la misma lista ligada y la complejidad temporal de la búsqueda se ve reducida a la complejidad de búsqueda en listas ligadas, es decir  $O(n)$ .

Para esta actividad se realizó una simulación del proceso de hash y encadenamiento, al insertar un elemento en la tabla se genera un número aleatorio que corresponde a un índice del arreglo donde se insertará el elemento.

En este caso utilizamos como colección de la tabla hash simulada un arreglo de listas ligadas, cuando se inserta un elemento a uno de los índices del arreglo este se añade al final de la lista ligada correspondiente.

Para mostrar el contenido de la colección se utiliza el método `toString()` de la clase `Arrays`.

Para la simulación de encadenamiento se crearon los siguientes métodos:

- `private static void initLists(LinkedList[] t)`: Método estático de tipo void, recibe como argumento un arreglo de listas ligadas. Inicializa cada una de las listas ligadas del arreglo.



- `private static void printMenu()`: Método estático de tipo `void`, no recibe argumentos. Imprime el menú de usuario utilizado en el método `main`.
- `private static int readInt()`: Método estático de tipo `int`, no recibe argumentos. Permite al usuario ingresar un número entero, el método utiliza manejo de excepciones para evitar errores causados por la entrada de tipos de datos incorrectos.

En el programa principal se define un arreglo de listas ligadas que simulará el contenedor de una tabla hash y se utiliza el método `initLists()` para inicializar los elementos del arreglo.

Usando un ciclo `while` en conjunto con una estructura `switch` se creo un menú de usuario que presenta las siguientes opciones:

```
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
```

Figura 2.58: Menú de opciones

Para agregar un elemento a la lista se llama a la función `readInt()` para asegurarse de que la entrada de datos sea válida, después se genera un número aleatorio “i” entre 0 y el último índice del arreglo, por último usando el método `add()` de la clase `LinkedList` se añade el elemento ingresado en la i-ésima lista del arreglo.

Para mostrar el contenido de la lista se utiliza el método `ToString()` para imprimir el contenido del arreglo.

Para probar la funcionalidad del programa insertamos elementos del 1 al 15 e imprimimos el contenido del arreglo.

```

(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
1
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
2
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
3
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
4
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
5

```

(a) Inserción de datos

```

(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
6
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
7
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
8
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
9
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
10

```

(b) Inserción de datos

```

(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
11
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
12
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
13
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
14
***** Opciones: *****
(0) Agregar elemento:
(1) Mostrar lista:
(2) Salir:
Seleccione una opcion:
0
Ingrese el valor del elemento:
15

```

(c) Inserción de datos

Contenido del arreglo:

```
Contenido de la lista:  
[[], [3], [12], [2, 13, 15], [], [], [11], [5, 6], [7], [], [1, 8], [4, 14], [], [10], [9]]
```

Figura 2.60: Impresión de datos

Podemos observar colisiones en los índices 3, 7, 10 y 11.

## Conclusiones

### Hernández Gallardo Daniel Alonso

El objetivo se cumplió en su totalidad pues se realizaron programas que implementan todos los conceptos vistos teóricos, con un agregado de aplicación a la programación orientada a objetos y el uso de colecciones, los ejercicios vistos en este proyecto fueron muy útiles para aprender sobre los conceptos vistos pues se implementaron desde cero para poder crear el programa, además el uso del lenguaje Java en el paradigma orientado a objetos le agrega un extra al aprendizaje y dominio de este lenguaje de programación. La estrategia se basó en investigar los conceptos teóricos, analizarlos y buscar la manera de implementar una solución que satisfaga con los datos que necesitamos, en el caso del encadenamiento, la creación de una lista de listas que permita "solucionar" las colisiones y, para las búsquedas que se nos brinde, si lo encontró, en dónde y cuántas veces, de forma en que se puedan buscar objetos por los atributos de los mismos. La ventaja del uso de la búsqueda lineal es que es infalible pues en todos los casos funcionará y, de la misma manera, la desventaja es su complejidad algorítmica  $O(n)$ . La ventaja de la búsqueda binaria es su complejidad algorítmica  $O(\log n)$  pues, es una forma muy rápida de buscar un elemento; su desventaja es que la lista requiere de un ordenamiento por lo que hace perder su ventaja de complejidad pues haría que por lo menos se vuelva lineal. La ventaja del encadenamiento es su ahorro en memoria al momento de solucionar colisiones pues, la memoria se ocupa conforme van llegando los datos, su desventaja es que si estamos trabajando datos que resulten en una key igual pero con un valor distinto terminaríamos haciendo que se vuelva una búsqueda lineal dado que, se tendría que buscar entre todas las colisiones para encontrar el dato buscado quitándole su utilidad.

## Perez Osorio Luis Eduardo

A partir de la investigación del funcionamiento de las colecciones de java implementamos los diferentes métodos de búsqueda por comparación de llaves vistos en clase, en el proceso de implementar estos algoritmos pudimos observar las ventajas presentes en estos, por ejemplo la relativa facilidad de la búsqueda lineal así como la manera en la que esta garantiza su funcionamiento para cualquier lista de elementos, para la búsqueda binaria vimos el incremento drástico en el rendimiento de la búsqueda pero también observamos como la necesidad de ordenar la colección que utiliza puede reducir el rendimiento del programa hasta el punto de volver la búsqueda binaria poco practica en algunos casos.

A diferencia de los ejemplos vistos en clase no estamos buscando elementos en colecciones de primitivos, sino que estamos buscando valores en colecciones de objetos, para poder realizar la búsqueda de manera efectiva se hizo uso extensivo de la herencia y el polimorfismo para generalizar el funcionamiento de la comparación de objetos en nuestras funciones de búsqueda.

Con respecto al tema de búsqueda por transformación de llaves realizamos una simulación de la estrategia de resolución de colisiones por encadenamiento, en esta podemos apreciar la mejora en la complejidad de memoria comparado al método de arreglos anidados visto en clase, aunque también vimos que un conjunto de datos particular es capaz de reducir el rendimiento de nuestra tabla hash a aquel de una lista ligada, sin embargo, esto puede solucionarse con la selección de una función hash adecuada para nuestro conjunto de datos.

Logramos implementar los conceptos vistos en clase y aquellos que encontramos durante la investigación de este proyecto, es por estas razones que considero que se alcanzaron al 100 % los objetivos establecidos por el proyecto.

## Valle Chavez Anton Yael

El avance de la sociedad contemporánea a raíz de la introducción de los sistemas informáticos a provocado una profunda transformación de los fenómenos sociales, industriales y económicos a lo largo de su existencia, la revolución de la información como subproducto del avance tecnológico ha provocado fuertes cambios arraigados a nuestra existencia. Una de las maneras en la cuál es posible manipular estos nuevos procesos es mediante programas de información, que a su vez, están codificados en lenguajes de programación, en este caso Java. Java es un lenguaje de programación orientado a objetos, es decir, implementa estos junto con clases para crear abstracciones de la realidad derivables de un acercamiento computacional sobre el cuál es posible ejercer cierto control sobre los mismos, asimismo, al conjunto de un grupo de objetos se les llama colecciones. En resumen, la colección `collection` tiene varias interfaces que heredan de ella, entre ellas está `listas`, `set` y `map`, que intrínsecamente, comparten ciertas similitudes pero tienen características distintas, tanto en la forma en cómo admiten objetos, cómo en sus métodos para manipularlos. Al poder entonces utilizar estas colecciones, es posible desarrollar o emplearlas en situaciones de

la vida real, cómo lo es la búsqueda lineal que verifica si el valor es el mismo en cada caso para "buscar" su igual, en búsqueda binaria, se compara entonces por mitades hasta encontrar el valor dado. El encadenamiento a su vez, posee la capacidad de reducir el espacio necesario para resolver las colisiones de los arreglos anidados al emplear listas ligadas como una proposición diferente. Finalmente, la relación que guardan entonces todas las colecciones de Java resulta en un cúmulo de conocimientos necesarios para poder manipular grandes cantidades de información como objetos aplicables a cualquier situación de la vida cotidiana, mejorando entonces el perfil profesiográfico del estudiante.

## Referencias

- GeeksforGeeks. (2021). *How to learn java collections: A complete guide*. Descargado de <https://www.geeksforgeeks.org/how-to-learn-java-collections-a-complete-guide/>
- GeeksforGeeks. (2023). *Collections in java*. Descargado de <https://www.geeksforgeeks.org/collectionsin-java-2/>
- Luaces, H. (2013). *Guía de colecciones en java*. Descargado de <https://www.luaces-novo.es/guia-de-colecciones-en-java/>
- Monteserín, P. (2023). *Sobreescritura del equals - pablo monteserín*. Descargado de <https://pablomonteserin.com/curso/java/sobreescritura-del-equals/>
- Oracle. (2023a, junio). *List (java platform se 8)*. Descargado de <https://docs.oracle.com/javase/8/docs/api/java/util/List.html>
- Oracle. (2023b, julio). *Overview (java se 17 jdk 17)*. Descargado de <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>
- Oracle. (2023c, junio). *Set (java platform se 8)*. Descargado de <https://docs.oracle.com/javase/8/docs/api/java/util/Set.html>
- Oracle. (2023d, junio). *Treemap (java platform se 8)*. Descargado de <https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>
- Oracle. (2023e, junio). *Treeset (java platform se 8)*. Descargado de <https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
- Oracle. (s. f.). *The set interface (the javatm tutorials ¿ collections ¿ interfaces)*. Descargado de <https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html>
- Oracle Corporation. (sin fecha). *Lesson: Generics*. Descargado de <https://docs.oracle.com/javase/tutorial/java/generics/index.html> (The Java Tutorials ¿ Learning the Java Language)
- Oracle Corporation. (sin fecha). *Lesson: Generics*. Descargado de <https://docs.oracle.com/javase/tutorial/extra/generics/index.html> (The Java Tutorials ¿ Bonus)
- Rodriguez, A. (s.f.). *Jerarquías de herencia en java*. Descargado de [https://www.aprenderaprogramar.com/index.php?option=com\\_content&view=article&id=652:jerarquias-de-herencia-en-java-concepto-de-superclases-y-subclases-el-api-java-ejemplos-cu00685b&catid=68&Itemid=188](https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=652:jerarquias-de-herencia-en-java-concepto-de-superclases-y-subclases-el-api-java-ejemplos-cu00685b&catid=68&Itemid=188)
- Rodríguez, A. (sin fecha). *Sobreescibir métodos tostring y equals en java. ejemplos. ejercicios resueltos. comparar objetos. (cu00694b)*. Descargado de [https://www.aprenderaprogramar.com/index.php?option=com\\_content&view=article&id=666:sobreescibir-metodos-tostring-y-equals-en-java-ejemplos-y-ejercicios-resueltos-comparar-objetos-cu00694b&catid=68&Itemid=188](https://www.aprenderaprogramar.com/index.php?option=com_content&view=article&id=666:sobreescibir-metodos-tostring-y-equals-en-java-ejemplos-y-ejercicios-resueltos-comparar-objetos-cu00694b&catid=68&Itemid=188)
- Sedgewick, R., y Wayne, K. D. (2017). *Algorithms*. W. Ross Macdonald School Resource Services Library.
- s. f. (s.f.). *Colecciones de java*. Descargado de <https://www.cosmiclearn.com/lang-es/java-collections.php>

- s.f. (s.f.). *Colecciones en java*. Descargado de <https://www.luaces-novo.es/guia-de-colecciones-en-java/>
- s. f. (s.f.). *Collections de java*. Descargado de <http://www.reloco.com.ar/prog/java/collections.html>
- Vindel, R. (2015). *Introducción a las colecciones en java*. Descargado de <https://www.adictosaltrabajo.com/2015/09/25/introduccion-a-colecciones-en-java/>