PROYECTO FINAL

“Sistema de Gestión Editorial”

Alumna: Jocelyn Flores Gutiérrez

Maestro: Adalberto Emmanuel Rojas Perea

Materia: Estructura de Datos

Fecha: 28/09/2025

TECMILENIO

● Índice de contenidos

Implementar y manipular diferentes estructuras de datos en Java, así como entender las aplicaciones prácticas de pilas, colas, listas enlazadas, tablas Hash, recursividad, etc. en el manejo de información para desarrollar habilidades de programación y resolución de problemas en un contexto empresarial simulado.

● Sistema de Gestión Editorial

Las editoriales son las entidades que se encuentran detrás de la creación, publicación y distribución de las grandes obras que podemos encontrar en el mercado. A menudo sus tareas van desde el diálogo y trabajo conjunto con autores y artistas hasta la organización de eventos de promoción y venta de las obras producidas.

Con este software se pretenden cubrir las necesidades más fundamentales de organización de una editorial, haciendo más eficiente el proceso de agendar, gestionar y consultar las propuestas de obras y los eventos de presentación y publicación de libros.

Se pretende implementar distintas estructuras de datos en el lenguaje de programación orientado a objetos 'JAVA', tales como las listas enlazadas, las pilas, los montículos, los árboles binarios, la recursividad, algunos métodos de búsqueda y ordenamiento, tablas Hash y grafos, entre otros, analizando cuál es la estructura más óptima para cada tarea.

Montículos para la gestión de próximas publicaciones/presentaciones de libros.

Utiliza montículos (colas priorizadas) para gestionar los próximos lanzamientos de obras, siendo una excelente opción porque en el mundo editorial es importante priorizar los eventos con mayor impacto, más dedicación o que resultan más urgentes que otros. Los montículos ofrecen un sistema ideal para adelantar las presentaciones que tienen mayor prioridad, pero al mismo tiempo mantienen un flujo constante de antigüedad para que los eventos se realicen en el orden de llegada (si no hay algún otro que sea más urgente).

Pilas para guardar historial de eventos pasados.

Como se ha mencionado anteriormente, la gran parte de las actividades de una editorial son los eventos de publicación/presentación de las obras, pero además de llevar registro de las actividades venideras, también es importante llevar un historial de las ya acontecidas en un orden de antigüedad.

La estructura de las pilas es ideal para gestionar un historial de reciente/antiguo donde la consulta de los eventos se encuentran en orden cronológico y, por seguridad, no es posible alterar el orden sino que sólamente se puede agregar o eliminar el evento más reciente.

Listas enlazadas para gestionar las propuestas de obras.

Cada editorial suele tener su propio método para trabajar con los autores, y, a menudo, las propuestas de obras tienen que revisarse más de una vez por distintas personas, dejarse de lado para atender otra y luego ser retomada; es por ello que su forma de almacenamiento puede ser adecuada para las listas enlazadas puesto que son versátiles, flexibles y de tamaño variable, lo que permite agregar y eliminar elementos sin un orden específico.

Árboles binarios para gestionar a los empleados.

En una editorial, como en cualquier trabajo u organización que esté integrada por más de una persona, la gestión de personas puede ser muy útil para organizar a los miembros. En este caso, un árbol binario puede ser la estructura adecuada para manejar personas con una cierta jerarquía. Al ser una estructura de datos no lineal y que sigue un conjunto de reglas para organizarse (los nodos se asignan dependiendo de si son mayores o menores que su padre: si son menores se colocan a la izquierda, y si son mayores, a la derecha) puede ser muy eficiente para la búsqueda de sus elementos, algo que es importante a la hora querer hacer una consulta rápida del personal.

Recursividad y algoritmos divide y vencerás para procesos y tareas diversas.

La recursividad y los algoritmos divide y vencerás son herramientas muy útiles que pueden implementarse para ahorrar recursos y resolver un problema de forma eficiente, por lo que, más que asignarles una tarea en específico, realmente están presente en varios de los procesos de las estructuras anteriores, ayudando a hacerlos más sencillos o aportando una forma distinta de resolverlos. En este caso, fue posible utilizar la recursividad como un medio para generar los códigos ISBN de identificación de los libros y almacenarlos en un arreglo, así como de utilizarla para recorrer los árboles binarios y buscar elementos. Mientras que el algoritmo de divide y vencerás, aunque comúnmente utiliza la recursividad, realmente su metodología se encuentra presente en muchas cosas, pues en lo que se basa es en dividir una tarea compleja en tareas más sencillas, es por ello que puede implementarse por ejemplo en el método de búsqueda binaria.

Métodos de ordenamiento y búsqueda para un catálogo de libros.

El registro y almacenamiento de las obras ya registradas (en otras palabras, que cuentan con un código ISBN) es importante para saber cuáles son las obras ya terminadas con las que cuenta la editorial, y un método de ordenamiento puede hacer más eficiente el encontrarlos si están ordenados de menor a mayor (o en orden alfabético, si fueran letras). Los métodos de ordenamiento como el bubble o por selección son dos ejemplos que funcionan y son mejores para casos distintos, siendo más sencillo el bubble, pero también menos eficiente para colecciones grandes; por otro lado, el método de selección es más rápido que el bubble y realiza menos pasos, pero sigue siendo de cierta forma ineficiente.

En el caso de los algoritmos de búsqueda, su finalidad es recorrer la colección buscando el método más eficiente, haciéndolo cada uno a su manera. Por ejemplo, el secuencial, es sencillo y confiable, pudiendo utilizarse en un arreglo que no esté ordenado, pero como recorre elemento por elemento, puede ser tanto rápido como lento dependiendo de la posición del elemento, así que no se recomienda para colecciones muy grandes. En el caso del algoritmo de búsqueda binaria, este es más sencillo y hace menos movimientos al comparar si el elemento buscado es menor o mayor a los elementos, por lo que es muy eficiente, pero necesita de tener los elementos ordenados previamente para utilizarlo.

Tablas Hash y HashMap para el almacenamiento de obras y socios

Así como se lleva un registro de las obras terminadas, eso no significa que no sea importante tener un registro de todas las obras (terminadas y en proceso) que tiene la editorial, así como de sus socios comerciales y contactos estratégicos. Para eso las estructuras como las tablas Hash y los Mapas Hash son particularmente ideales por su estructura de llave/valor que funciona como un diccionario donde las consultas son muy rápidas y eficientes, así como su almacenamiento dinámico.

Grafos

Si una editorial tiene varias sedes, conocer los recorridos más cortos de una sede a otra puede ahorrar mucho tiempo para la paquetería y el envío de archivos o elementos, por lo que los grafos pueden ser de mucha ayuda para mapear los lugares (vértices) y su valor de ruta (aristas), calculando la ruta más corta de un lugar a otro.

• Diseño

Los menús muestran las opciones de permitir ingresar, buscar, eliminar y ver elementos de cada tipo de estructura de datos. Se dividió en uno principal y varios secundarios para las opciones del primero.

MENÚ PRINCIPAL

- 1) Agendar evento (Montículo/cola)
- 2) Historial de eventos (Pila)
- 3) Propuestas de proyecto (Lista enlazada)
- 4) Empleados (Árbol binario)
- 5) Catálogo de obras terminadas (Recursividad/Divide y vencerás y métodos de ordenamiento y búsqueda)
- 6) Catálogo general (Tabla hash y HashMap)
- 7) (Grafos)
- 8) Salir

MENÚS SECUNDARIOS

Montículo: “Agendar evento”

- 1) Agregar evento a la cola (push) - enqueue
- 2) Mostar evento actual (peek) - front
- 3) Completar evento actual (pop) - dequeue
- 4) Mostrar todos los eventos en cola (display)
- 5) Volver

Pila: “Historial de eventos”

- 1) Agregar evento al historial (push)
- 2) Mostar evento más reciente (peek)
- 3) Borrar evento más reciente (pop)
- 4) Mostrar historial de eventos (display)
- 5) Volver

Lista enlazada: “Propuestas de proyecto”

- 1) Agregar propuesta (insert)
- 2) Buscar propuesta (find)
- 3) Borrar propuesta (delete)
- 4) Mostrar propuestas (display)
- 5) Volver

Árbol binario: “Gestión de empleados”

- 1) Agregar empleado (insertar)
- 2) Eliminar empleado (eliminar)
- 3) Buscar empleado (buscar)
- 4) Mostrar empleados (preorden, inorden, posorden)
- 5) Volver

Recursividad, DYV, Ordenamiento y Búsqueda: “Catálogo obras terminadas”

- 1) Generar ISBN (recursividad)
- 2) Mostrar ISBN registrados (mostrar arreglo)
- 3) Ordenar registros (menor a mayor) (selection sort)
- 4) Búsqueda secuencial de ISBN (búsqueda secuencial)
- 5) Búsqueda binaria de ISBN (búsqueda binaria)
- 6) Volver

HashTable y HashMap: “Catálogo general”

- 1) Socios (tabla Hash)
- 2) Obras (HashMap)
- 3) Volver

HashTable: “Socios”

- 1) Agregar socio (put)
- 2) Buscar socio (find)
- 3) Borrar socio (remove)
- 4) Mostrar socios (display)
- 5) Volver

HashMap: “Obras”

- 1) Agregar obra (put)
- 2) Buscar obra (find)
- 3) Borrar obra (remove)
- 4) Mostrar obras (display)
- 5) Volver

• Implementación

Métodos estáticos generales del Main:

Estos métodos pretenden poder usarse globalmente en las clases del paquete, para reutilizar código.

El método **swap** encuentra en el Main y está diseñado para realizar un intercambio de elementos en el arreglo, guardando la referencia del primer objeto en una variable temporal, para luego asignarle al index del primer objeto la referencia del segundo objeto, y finalmente el index del segundo objeto tomará la referencia que se guardó en la variable temporal.

```
public static <E> void swap(E[] arreglo, int a, int b) {    //Intercambiar elementos en el arreglo
    E temp = arreglo[a];   // Guardar la referencia al objeto en una variable temporal
    arreglo[a] = arreglo[b];   //Mover al padre al lugar del objeto
    arreglo[b] = temp;     //Mover al objeto al lugar del padre
}
```

El método **getIndex** pretende encontrar el index de un elemento en el arreglo. Logra esto recorriendo todo el arreglo y haciendo una comparación entre elementos hasta encontrar uno que coincida, entonces regresa el valor que tiene registrado el contador del ciclo.

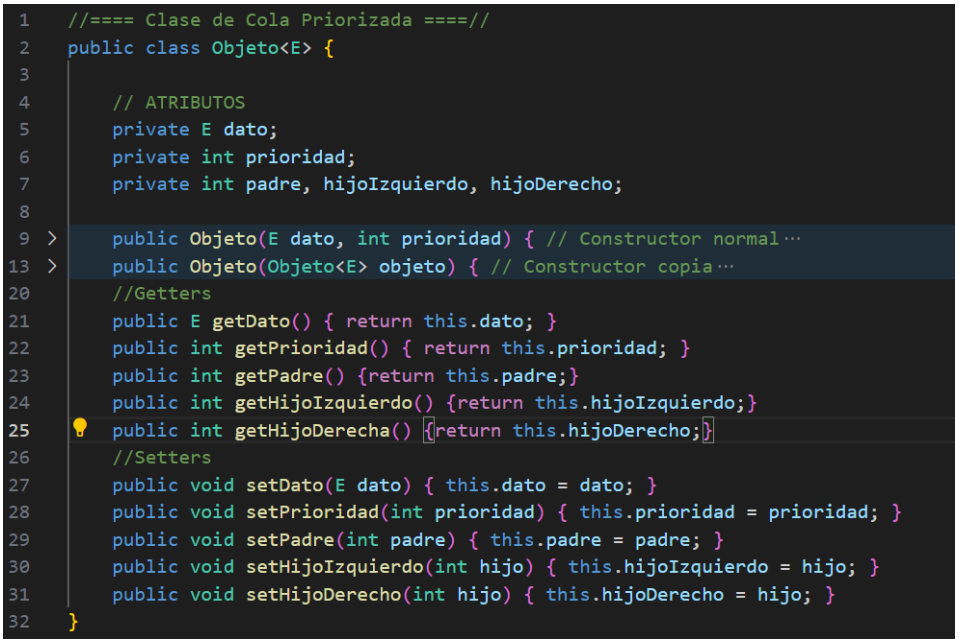
```
public static int getIndex(Objeto objeto, Objeto[] arreglo) { //Obtener un index del arreglo
    for (int i = 1; i < arreglo.length; i++) {
        if (arreglo[i].equals(objeto)) {     //Comparar si es el mismo objeto en memoria con ==, sino reemplazar
            return i; // devuelve el índice
        }
    }
    return -1; //Dará error porque -1 está fuera del rango
}
```

De forma similar al anterior, el método **getSize** regresa el tamaño de los elementos reales que hay en el arreglo, recorriendo todos los elementos y contando únicamente los que no tienen un valor nulo.

```
public static int getSize(Objeto[] arreglo) { //Obtener el num. de elementos del arreglo
    int tamano = 0;
    for (Objeto elemento: arreglo) { // Recorre todos los elementos del arreglo
        if (elemento != null) { //Si no está vacío el espacio lo cuentas
            tamano += 1; // Contador que cuenta los elementos reales
        }
    }
    return tamano;
}
```

Montículo/cola prioritaria:

Se hizo una clase pública con el propósito de crear objetos a partir de ella que contengan los atributos privados de “dato”, “prioridad”, “padre”, “hijoIzquierdo” e “hijoDerecho”, así como métodos getters y setters que permitan acceder a ellos y modificarlos.



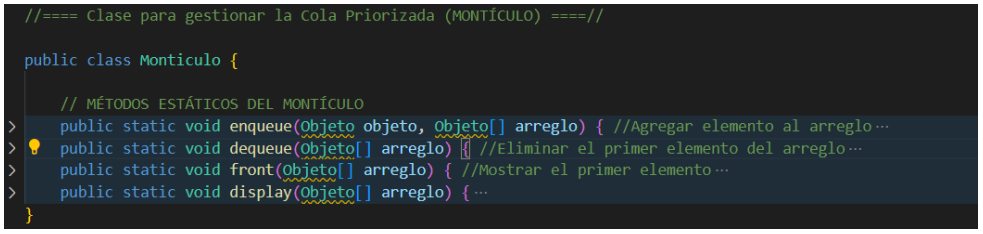
```java
//==== Clase de Cola Priorizada ====//
public class Objeto<E> {

    // ATRIBUTOS
    private E dato;
    private int prioridad;
    private int padre, hijoIzquierdo, hijoDerecho;

    public Objeto(E dato, int prioridad) { // Constructor normal ···
    public Objeto(Objeto<E> objeto) { // Constructor copia ···
    //Getters
    public E getDato() { return this.dato; }
    public int getPrioridad() { return this.prioridad; }
    public int getPadre() {return this.padre;}
    public int getHijoIzquierdo() {return this.hijoIzquierdo;}
    public int getHijoDerecha() {return this.hijoDerecho;}
    //Setters
    public void setDato(E dato) { this.dato = dato; }
    public void setPrioridad(int prioridad) { this.prioridad = prioridad; }
    public void setPadre(int padre) { this.padre = padre; }
    public void setHijoIzquierdo(int hijo) { this.hijoIzquierdo = hijo; }
    public void setHijoDerecho(int hijo) { this.hijoDerecho = hijo; }
}
```

Se pretende que los objetos sean el tipo de dato ingresado en el arreglo que representará al montículo que se declara en el main.

```java
Objeto[] monticulo = new Objeto[100]; //Declarar montículo. Le ponemos tamaño grande
```

La principal desventaja de hacer la cola prioritaria con un arreglo es que se debe indicar un tamaño fijo, el cuál puede establecerse con un valor muy alto para dar la ilusión de ser “infinito”.

Luego se crearon los métodos estáticos en la clase Montículo para poder gestionar el montículo, así como se utilizaron algunos métodos estáticos generales del Main:



```java
//==== Clase para gestionar la Cola Priorizada (MONTÍCULO) ====//

public class Monticulo {

    // MÉTODOS ESTÁTICOS DEL MONTÍCULO
    public static void enqueue(Objeto objeto, Objeto[] arreglo) { //Agregar elemento al arreglo ···
    public static void dequeue(Objeto[] arreglo) { //Eliminar el primer elemento del arreglo ···
    public static void front(Objeto[] arreglo) { //Mostrar el primer elemento ···
    public static void display(Objeto[] arreglo) { ···
}
```

El método **enqueue** es el más complejo, pues es donde se encuentra todo el procedimiento para hacer de una cola ordinaria, un montículo.
Comienza pidiendo como argumentos el objeto que se quiere agregar y el arreglo a donde se quiere agregar (el montículo), siendo lo primero de todo el comprobar si la lista tiene elementos o no. Si no los tiene entonces el nuevo elemento se asignará directamente a la posición 1 y se calculará la posición de sus hijos y su padre (que como no tiene, entonces se puede omitir o ponerle un valor de cero, no hay problema porque no se utilizará).

Hasta ahí llegarán las opciones si es el primer elemento.

```
public static void enqueue(Objeto objeto, Objeto[] arreglo) { //Agregar elemento al arreglo

    if (getSize(arreglo) == 0) {    //Si la lista está vacía agregarlo en la posición 1
        arreglo[1] = objeto;    //Agregarlo al arreglo
        objeto.setHijoIzquierdo(hijo:2); //2 - 2*n
        objeto.setHijoDerecho(hijo:3); //3 - 2*n +1
        objeto.setPadre(padre:0);//No tiene padre aún

    } else {    //No es el primer elemento
```

Si no es el primer elemento, entonces se agregará al próximo espacio que esté disponible y, de igual manera se le asignará el lugar de sus hijos y padre siguiendo las fórmulas siguientes:

Padre: n / 2 *Hijo izquierdo: n*2* *Hijo derecho: n*2 + 1*

```
arreglo[getSize(arreglo) + 1] = objeto; //Agregarlo primeramente
//Asignar a sus hijos
objeto.setHijoIzquierdo(getIndex(objeto, arreglo)*2); //2*n
objeto.setHijoDerecho((getIndex(objeto, arreglo)*2) +1); //2*n +1
objeto.setPadre(getIndex(objeto, arreglo)/2);  //Asignarle padre
```

Ahora se utiliza un ciclo para hacer la condición más importante del método: comprobar si el nuevo objeto tiene una prioridad menor a la de su padre, y si es así entonces se guardará en una variable la posición inicial del objeto nuevo, y luego se procederá a intercambiar el lugar del nuevo nodo con su padre con el método **swap**.

```
while(true) {
    if (objeto.getPrioridad() < arreglo[objeto.getPadre()].getPrioridad()) { //Si es menor la prioridad del nuevo
        int index_inicial_objeto = getIndex(objeto, arreglo);   //Guarda la posición inicial del objeto antes de cambiar
        swap(arreglo, getIndex(objeto, arreglo), objeto.getPadre()); //Cambia lugar con el padre en el arreglo
```

Es importante que una vez realizado el intercambio se actualicen los valores del padre y los hijos de ambos objetos: el padre y el nuevo objeto. Es importante tomar en cuenta que en caso de que el nuevo objeto haya quedado en la posición 1, no se modifica su padre por lo mismo de que la posición 1 no tiene un padre como tal.

```
//Si quedó en la posición 1 del arreglo no se modifica su padre
if (getIndex(objeto, arreglo) == 1) {
} else {    //Si no es el 1, reasignarle padre
    objeto.setPadre(getIndex(objeto, arreglo)/2);
}

//Actualiza a sus hijos
objeto.setHijoIzquierdo(getIndex(objeto, arreglo)*2); //2*n
objeto.setHijoDerecho((getIndex(objeto, arreglo)*2) +1); //2*n +1

//Actualizar datos del padre anterior, que se movió al lugar inicial del objeto
arreglo[index_inicial_objeto].setPadre(index_inicial_objeto/2); //actualizarle el padre al padre
arreglo[index_inicial_objeto].setHijoIzquierdo(index_inicial_objeto*2); //Actualizar hijo izquierdo del padre
arreglo[index_inicial_objeto].setHijoDerecho((index_inicial_objeto*2) +1); //Actualizar hijo derecho del padre
```

Para el método **dequeue** el procedimiento es nuevamente comprobar si está vacío el arreglo, y si no lo está entonces sí puede borrar el primer elemento del arreglo. Primero guarda el valor que se eliminará para poder mostrarlo al final, luego recorre todos los elementos del arreglo un espacio para

que quede el segundo en la posición número uno, y así sucesivamente (esto permite aprovechar bien el espacio del arreglo). Finalmente señala el último elemento que se movió como nulo y se muestra el elemento eliminado.

```
public static void dequeue(Objeto[] arreglo) { //Eliminar el primer elemento del arreglo

    if (getSize(arreglo) == 0) { //El arreglo está vacío
        System.out.println(x:"No hay eventos registrados.");
    } else {
        Objeto eliminado = arreglo[1]; //Guardar en una variable el objeto eliminado
        //Recorrer los elementos un lugar en el arreglo
        for (int i = 1; i < getSize(arreglo); i++) { //De 1 a 'tamaño del arreglo' -1
            arreglo[i] = arreglo[i+1];
        }
        arreglo[getSize(arreglo)] = null; //Se actualiza el último espacio para que sea null
        System.out.println("Evento completado: " + eliminado.getDato());
    }

}
```

El método **front** se encarga de devolver el primer elemento del montículo, comprobando si está vacío y si no lo está entonces busca el elemento en la posición 1 y lo muestra junto con su prioridad.

```
public static void front(Objeto[] arreglo) { //Mostrar el primer elemento

    if (getSize(arreglo)== 0) { //Si el arreglo está vacío
        System.out.println(x:"No hay eventos registrados.");
    } else {
        System.out.println("Evento: " + arreglo[1].getDato() + " Prioridad: " + arreglo[1].getPrioridad());
    }
}
```

El conocido método **display** muestra todos los elementos del montículo repitiendo casi los mismo pasos que ya hemos visto: comprueba si tiene elementos y luego recorre todo el arreglo para mostrar cada elemento con un ciclo for que lo recorre del primero al último.

```
public static void display(Objeto[] arreglo) {
    if (getSize(arreglo) == 0) { //Si el arreglo está vacío
        System.out.println(x:"Aún no hay eventos.");
    } else { //Recorrer cada elemento
        System.out.println(x:" <= Actual | En espera =>");
        for (Objeto elemento: arreglo) {
            if (elemento != null) { //Si no está vacío el espacio lo muestra
                System.out.print(elemento.getDato() + "(" + elemento.getPrioridad() + ") "); //Muestra el dato y su prioridad entre par
            }
        }
        System.out.println();
    }
}
```

Pila:

Se hizo una clase pública con el propósito de crear objetos a partir de ella que contengan el arreglo de String que se utilizará y atributos privados como “top” y “capacity”, así como su respectivo constructor para crear el arreglo e indicar su capacidad. Así como métodos para gestionar la pila que se crea en el Main.

```java
//==== Clase de Pila ====//
public class Pila {
    // ATRIBUTOS
    private String[] data;  // Para guardar el arreglo de Strings
    private int top;        // Indica el lugar donde está el último elemento
    private int capacity;   // Indica la capacidad del arreglo

    // CONSTRUCTOR
    public Pila(int capacity) {
        this.capacity = capacity;
        data = new String[capacity]; // Crear arreglo con la capacidad asignada
        top = -1; // Pila vacía
    }

    //MÉTODOS
    public boolean isEmpty() { // Ver si está vacía la pila···
    public boolean isFull() { // Ver si ya se llenó la pila···
    public void push(String x) { // Agregar elemento a la pila···
    public String pop() { // Eliminar el último elemento···
    public String peek() { // Muestra el último elemento ingresado···
    public void display() { // Muestra todos los elementos de la pila del más reciente al más antiguo···
}
```

```java
Pila pila = new Pila(capacity:100); // Creamos una pila
```

El método **isEmpty** verifica si la pila está vacía comprobando si top no ha sido actualizado, de la misma manera el método **isFull** comprueba si el tope es el tamaño -1 de la capacidad.

```java
public boolean isEmpty() { // Ver si está vacía la pila
    return top == -1; // -1 es el valor para la pila vacía
}
```

```java
public boolean isFull() { // Ver si ya se llenó la pila
    return top == capacity -1; //Si el último elemento es igual a la capacidad -1
}
```

El método **push** se encarga de agregar elementos a la pila, comprobando primero si se encuentra llena y si no es el caso entonces agrega el elemento al próximo espacio disponible.

```java
public void push(String x) { // Agregar elemento a la pila
    if (isFull()) { // Si está lleno no agrega nada porque da error
        System.out.println("Límite alcanzado. No se pueden agregar más elementos. " + x);
        return;
    }
    data[++top] = x; // Lo agrega al próximo espacio disponible
    System.out.println(x:"Registro exitoso!");
}
```

El método **pop** pretende eliminar el último elemento introducido a la pila, comprobando si hay elementos y si los hay entonces reduce el tamaño de los elementos un espacio.

```java
public String pop() { // Eliminar el último elemento
    if (isEmpty()) { // Si está vacío no puede eliminar nada
        return "Historial vacío."; //indicador de error
    }
    return "Eliminado: " + data[top--]; //Muestra el dato eliminado y le quita un espacio a la pila
}
```

El método **peek** hace casi lo mismo que el método pop pero sin eliminar un espacio, sino simplemente mostrando el elemento en el tope.

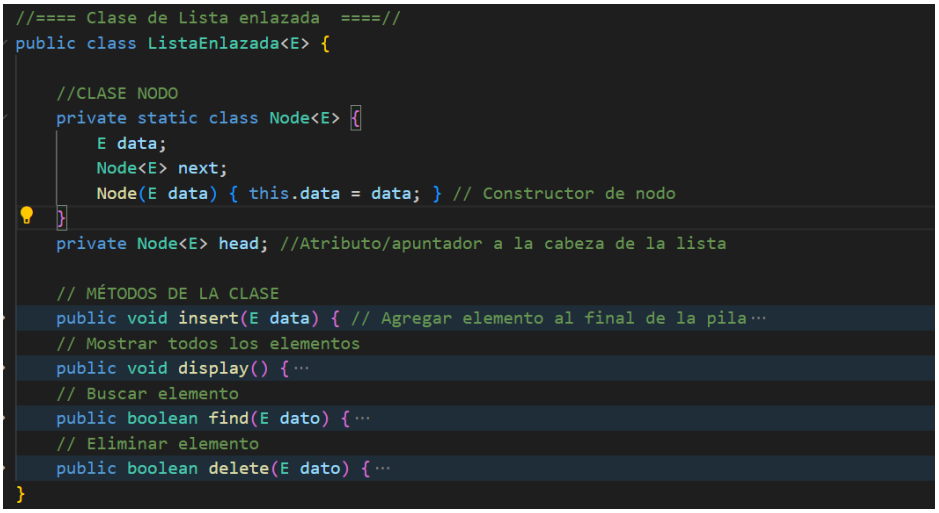
```java
public String peek() { // Muestra el último elemento ingresado
    if (isEmpty()) {
        return "Historial vacío.";
    }
    return data[top]; //Devuelve el último elemento ingresado
}
```

El método **display** muestra todos los elementos de la pila en un orden del más reciente al más antiguo (empieza desde el tope del arreglo hasta el inicio), recorriendo todos los elementos del arreglo y mostrándolos.

```java
public void display() { // Muestra todos los elementos de la pila del más reciente al más antiguo
    if (isEmpty()) { // Si está vacío no puede mostrar nada
        System.out.println(x:"Historial vacío.");
    } else {
        System.out.println(x:"<- Reciente | Antiguo -> "); // Muestra de top a fondo de la pila
        for(int i = top; i >= 0; i--) { // Recorre todos los elementos de la pila
            System.out.print(data[i] + " - "); // Muestra los elementos
        }
        System.out.println();
    }
}
```

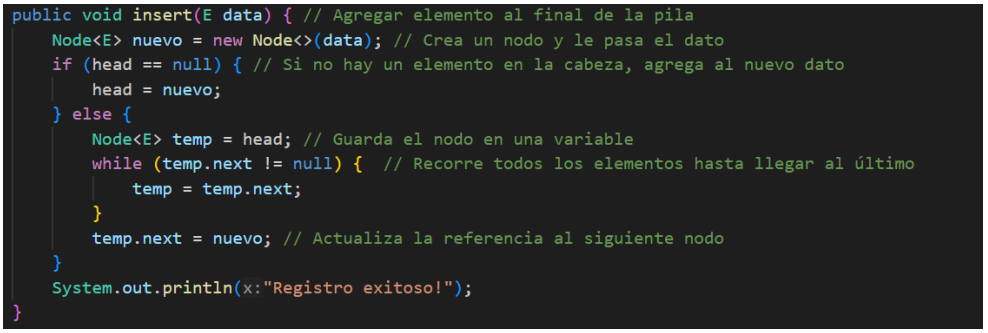
Lista enlazada:

Se hizo una clase pública para la lista enlazada con una clase nodo dentro de ella (el nodo es de tipo de dato genérico y guarda tanto el dato como la referencia al siguiente nodo), se agregó como atributo un apuntador que señale la cabeza de la lista y también se declararon ahí los métodos para manipular la lista.



```
//==== Clase de Lista enlazada  ====//
public class ListaEnlazada<E> {

    //CLASE NODO
    private static class Node<E> {
        E data;
        Node<E> next;
        Node(E data) { this.data = data; } // Constructor de nodo
    }
    private Node<E> head; //Atributo/apuntador a la cabeza de la lista

    // MÉTODOS DE LA CLASE
    public void insert(E data) { // Agregar elemento al final de la pila···
    // Mostrar todos los elementos
    public void display() {···
    // Buscar elemento
    public boolean find(E dato) {···
    // Eliminar elemento
    public boolean delete(E dato) {···
}
```

El método **insert** se encargará de agregar un elemento al final de la lista, creando primero un nodo y pasándole el dato que contendrá. Luego se comprueba si hay elementos o si sería el primero, de ser el caso se asigna el apuntado de la cabeza al nuevo nodo, en caso contrario recorre todos los elementos moviéndose por la referencia y luego actualiza la referencia del último elemento para que apunte al nuevo.



```
public void insert(E data) { // Agregar elemento al final de la pila
    Node<E> nuevo = new Node<>(data); // Crea un nodo y le pasa el dato
    if (head == null) { // Si no hay un elemento en la cabeza, agrega al nuevo dato
        head = nuevo;
    } else {
        Node<E> temp = head; // Guarda el nodo en una variable
        while (temp.next != null) {  // Recorre todos los elementos hasta llegar al último
            temp = temp.next;
        }
        temp.next = nuevo; // Actualiza la referencia al siguiente nodo
    }
    System.out.println(x:"Registro exitoso!");
}
```

Para el método **display** se pretende mostrar todos los elementos guardando primeramente el elemento de la cabeza en una variable temporal y comprobando si hay elementos en la lista, si los hay recorre todas las posiciones mostrando los datos y moviéndose por las referencias.

```java
public void display() {
    Node<E> temp = head; // Guarda la cabeza en una variable
    if (temp == null) { // Mira si no hay elementos
        System.out.println(x:"No se encuentran registros.");
    } else {
        while (temp != null) { // Recorre todos los elementos
            System.out.print(temp.data + " -> "); // Los muestra
            temp = temp.next;
        }
        System.out.println(x:"null"); //Voy a dejar el null para que se vea que es una lista enlazada
    }
}
```

El método **find** pretende encontrar en la lista un cierto dato entregado, lo que consigue con el mismo procedimiento que el método anterior, pero en lugar de imprimir los datos que se encuentre, comparará los elementos con el valor entregado. Cuando se llegue a dar la coincidencia se regresa un valor true.

```java
public boolean find(E dato) {
    Node<E> temp = head;
    while (temp != null) { //Repasa todos los elementos
        if (temp.data.equals(dato)) { // Si encuentra la coincidencia
            return true; // Elemento encontrado
        }
        temp = temp.next; //se mueve con la referencia
    }
    return false; // Elemento no encontrado
}
```

El método **delete** es un poco más complejo porque debe realizar una serie de pasos más diferentes, pues lo que pretende es eliminar un elemento, y para lograrlo es necesario que las referencias apunten al siguiente elemento. Primeramente comprueba si no hay elementos, luego, si el dato está en la primera posición, mueve el puntero que señala la cabeza al siguiente. Si no es el primer dato, recorre los elementos, busca la coincidencia y luego mueve la referencia del nodo anterior al nodo que está después del dato.

```java
public boolean delete(E dato) {
    if (head == null) return false; //Si no hay elementos regresa falso

    if (head.data.equals(dato)) { // Si el dato está en la cabeza
        head = head.next; // Mueve el apuntador al próximo nodo
        return true;
    }

    Node<E> temp = head; // Guarda la cabeza en una variable
    while (temp.next != null) { // recorre los elementos
        if (temp.next.data.equals(dato)) { // revisa la coincidencia
            temp.next = temp.next.next; // elimina el nodo
            return true;
        }
        temp = temp.next; // se desplaza
    }
    return false; // no encontrado
}
```

Árbol binario:

Primero se creó la clase **Nodo** para definir la estructura que tendrá cada elemento del árbol. Para eso le definimos los atributos de *nombre* (que es donde tendrá el ID del empleado), *dato* (nombre del empleado) y los apuntadores a los nodos *hijoIzquierdo* e *hijoDerecho*.
Luego se agrega el constructor usual donde se inicializan los valores y los apuntadores a los hijos se definen como null porque la asignación de las referencias se hace hasta que se invoque el método "insertar()". Por último el método toString() del Nodo nos permite conocer el contenido principal del nodo (acabo de darme cuenta que había olvidado este método y no lo incluí en los próximos métodos, cosa que me hubiera permitido ahorrar un poco de código).

```java
public class Nodo<E> {
    int nombre;    // Nombre del nodo
    E dato;     // Dato del nodo
    Nodo<E> hijoIzquierdo;     // Apuntador al hijo izquierdo
    Nodo<E> hijoDerecho;       // Apuntador al hijo derecho

    public Nodo(int nombre, E dato) { // Constructor
        this.dato = dato;
        this.nombre = nombre;
        this.hijoIzquierdo = null; // Se inicializan como nulos
        this.hijoDerecho = null;
    }
    // Mostrar
    @Override
    public String toString() {
        return nombre + " - su dato es: " + dato; // Muestra el contenido del nodo
    }
}
```

Después se creó la clase **ArbolBinario** con los atributos de *raiz* (para señalar el nodo raiz) y *eliminado* (variable booleana para utilizarse en el método eliminar), así como el respectivo constructor que inicializa la raiz señalando que está vacía por el momento.

```java
class ArbolBinario<E> {
    Nodo<E> raiz;       // Definir un apuntador a la raiz
    boolean eliminado;  // Para ver si se eliminó un elemento en eliminar()

    public ArbolBinario() { // Constructor
        raiz = null; // Sin raiz al principio
    }
```

El método de insertar() es relativamente sencillo pues sólo se centra en seguir las reglas de jerarquía para acomodar el nuevo nodo en su lugar correspondiente según si es menor o mayor que el nodo padre.

Siguiéndolo paso a paso, primero se crea el nuevo nodo y lo primero que se hace es comprobar si hay elementos en el árbol; si no los hay, se asigna al nuevo nodo como la raíz.

```java
// Insertar nodo en el árbol
public void insertar(int nombre, E dato) {

    Nodo<E> nuevo = new Nodo(nombre, dato); // Crear nuevo nodo
    if (raiz == null) { // Si no hay raiz (esta vacio el arbol)
        raiz = nuevo;   // Asigna al nuevo nodo como raiz
        return;
    }
```

Si ya existen elementos, entonces se crean dos variables (auxiliar y padre) cuya función será prácticamente ir recorriendo el árbol en orden (primero la variable auxiliar, y cuando la decisión sea segura, le asigna el valor del nodo en el que está a la variable padre). Si el nuevo nodo es menor al valor del padre, entonces la variable se desplaza por la subrama izquierda hasta que llegue al final y entonces asigna el nuevo nodo como el hijo izquierdo del nodo padre en el que se encuentre la variable. De la misma forma ocurre si el nuevo nodo resulta mayor al valor del padre, únicamente con la diferencia de que se mueve por la subrama derecha y se asigna como el hijo derecho del padre.

```
Nodo<E> auxiliar = raiz; // Declara dos variables temporales para pasarse los valores
Nodo<E> padre; // Inician con el valor de la raiz
while (true) {
    padre = auxiliar; // El nodo que indica al padre toma el valor al que llega el auxiliar
    if (nombre < auxiliar.nombre) { // Si el nodo nuevo es menor al nodo actual que lleva el auxiliar
        auxiliar = auxiliar.hijoIzquierdo;  // Asigna al auxiliar el valor del nodo izq del nodo que lleva actualmente
        if (auxiliar == null) {             // Si es nulo es que llegó al último nodo de la rama
            padre.hijoIzquierdo = nuevo; // Le indica al padre que tome al nuevo nodo como su hijo izq
            return;
        }
    } else {        // Si el nuevo nodo es mayor o igual al nodo actual del aux
        auxiliar = auxiliar.hijoDerecho; // Se mueve al siguiente nodo derecho del nodo actual que lleva
        if (auxiliar == null) {         // Si es nulo es que llegó al final de la rama
            padre.hijoDerecho = nuevo; // Indica al padre que asigne al nuevo como su hijo derecho
            return;
        }
    } // Es como que el auxiliar va tanteando primero el camino y cuando está seguro le pasa la posicion al padre
}
```

El método *buscar()* está dividido en dos partes para poder pasarle un único parámetro (el id) a la llamada del método que se encuentra en el Main.

```
// Buscar nodo en el árbol
public boolean buscar(int id) {
    return buscarPreorden(raiz, id); // Le pasamos la raiz y el id
}
```

Para realizar la búsqueda por el árbol se utilizó el recorrido de preorden (me pareció que era buena idea) con recursividad, con los parámetros de nodo y id para que recorriera nodo por nodo hasta encontrar la coincidencia (funcionando como caso base) que regresaría true y mostraría el contenido del nodo. Si el primer nodo (el padre) no coincide, sigue con el hijo izquierdo y luego con el derecho. Si después de todas las comprobaciones no encontró una coincidencia, entonces regresa false a la otra función, y a su vez, la otra también devuelve el valor de false.

```
// Recorrido PreOrden para buscar elementos
public boolean buscarPreorden(Nodo<E> r, int id) {
    if (r != null) {
        if (r.nombre == id) {  // si coincide
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra el contenido
            return true; // Devuelve true porque lo encontro
        }
        // Si no está en el nodo actual, buscamos en hijos
        if (buscarPreorden(r.hijoIzquierdo, id)) {
            return true;       // Si lo encuentra devuelve true
        }
        if (buscarPreorden(r.hijoDerecho, id)) {
            return true;       // Si lo encuentra devuelve true
        }
    }
    return false; // no se encontró
}
```

Ahora sigue lo más difícil. Si bien insertar y buscar fueron relativamente sencillos, el trabajo más complejo y angustiante está en la eliminación de un elemento. ¿Por qué? Podría preguntarse uno, bueno, en el caso de las listas enlazadas es tan sencillo como mover la referencia del nodo anterior al siguiente del que se quiere eliminar, “ignorando” el elemento.

Pero en los árboles binarios es más complejo que eso, por el hecho de que no es lineal, por lo tanto no se puede simplemente mover la referencia del anterior al siguiente (porque hay dos nodos siguientes, no uno, y cada padre sólo puede tener dos hijos) además de que en todo momento se debe respetar la jerarquía de que si un nodo es menor a su padre va a la izquierda y si es mayor a la derecha. Todas esas cuestiones complican esa parte y la vuelven la más difícil, es por ello que el método se divide de cierta forma en varias partes: el método público que se encarga de recibir únicamente un parámetro y administra lo que debe devolver, el método privado que utiliza recursividad y contiene todo el procedimiento, y el método mínimo() que elige al candidato más pequeño de la subrama izquierda para ser el reemplazo.

```java
// Eliminar nodo en el árbol
public boolean eliminar(int dato) {
    eliminado = false;              // reiniciamos el indicador
    eliminar(raiz, dato);           // eliminamos en la raíz
    return eliminado;               // devolvemos si se eliminó o no
}
```

El método privado primero revisa si el árbol se encuentra vacío y luego viaja buscando entre los nodos la coincidencia con el dato que buscamos, cuando lo encuentre revisa cada caso dependiendo de la situación del nodo: si no tiene hijos (caso base), si tiene uno (caso base) o si tiene los dos (se invoca a la función mínimo para seleccionar el mejor candidato y transferir los datos con la variable temporal “sucesor”).

```java
// Encuentra el valor mínimo (sucesor inorden)
private Nodo<E> minimo(Nodo<E> nodo) {
    while (nodo.hijoIzquierdo != null) { // Selecciona el último elemento de la subrama izquierda
        nodo = nodo.hijoIzquierdo;
    }
    return nodo;
}
```

```java
// Eliminar2.0, que sirve para hacer todo el proceso
private Nodo<E> eliminar(Nodo<E> raiz, int id) {
    if (estaVacio()) { // Está vacío el árbol
        return null; // No se encontró el nodo
    }

    if (id < raiz.nombre) {
        raiz.hijoIzquierdo = eliminar(raiz.hijoIzquierdo, id); // Viaja a través de la subrama izquierda
    } else if (id > raiz.nombre) {
        raiz.hijoDerecho = eliminar(raiz.hijoDerecho, id); // Viaja por la subrama derecha
    } else { // Si coincide sigue el proceso
        eliminado = true; // Se encontró el Nodo

        // Caso 1: sin hijos
        if (raiz.hijoIzquierdo == null && raiz.hijoDerecho == null) {
            return null;
        }
        // Caso 2: un solo hijo
        else if (raiz.hijoIzquierdo == null) {
            return raiz.hijoDerecho;
        } else if (raiz.hijoDerecho == null) {
            return raiz.hijoIzquierdo;
        }
        // Caso 3: dos hijos
        else {
            Nodo<E> sucesor = minimo(raiz.hijoDerecho);
            raiz.dato = sucesor.dato; // Le pasamos los datos del sucesor
            raiz.nombre = sucesor.nombre; // Pasamos los datos del sucesor
            raiz.hijoDerecho = eliminar(raiz.hijoDerecho, sucesor.nombre);
        }
    }
    return raiz;
}
```

Por último, los recorridos son métodos void con recursividad que se encargan de mostrar el valor, pero se declaran como públicos y privados para protegerlos más y pasarles el valor de la raíz de forma interna (abstracción). La recursividad funciona recorriendo los nodos (mientras exista y no sea nulo) y lo imprime y recorre en el orden que corresponda.

```
    // Wrappers públicos
    public void preorden() { preorden(raiz); }
    public void inorden() { inorden(raiz); }
    public void posorden() { posorden(raiz); }

    // Recorrido PreOrden
    private void preorden(Nodo r) {
        if (r != null) {
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
            preorden(r.hijoIzquierdo);
            preorden(r.hijoDerecho);
        }
    }
    // Recorrido InOrden
    private void inorden(Nodo r) {
        if (r != null) {
            inorden(r.hijoIzquierdo);
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
            inorden(r.hijoDerecho);
        }
    }
    // Recorrido PosOrden
    private void posorden(Nodo r) {
        if (r != null) {
            posorden(r.hijoIzquierdo);
            posorden(r.hijoDerecho);
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
        }
    }
}
```

Recursividad y divide y vencerás:

Para la recursividad se creó una clase exclusiva para contener los métodos que utilizará, empezando por **generarISBN()** que regresa el código ISBN de 13 dígitos que generó. Este método utiliza la recursividad como si fuera un sustituto de un ciclo for, llamándose a sí mismo hasta que el contador que se pasó como parámetro incrementa hasta el límite de 13, que servirá como caso base para dejar de llamarse a sí mismo y entregar el valor de ISBN completo como un Long.

```
public class Recursividad {

    public static long generarISBN(int contador, String isbn) { // GENERAR ISBN DE 13 DÍGITOS CON RECURSIVIDAD
        if (contador == 13) {
            return Long.parseLong(isbn); // Regresa el ISBN generado, convertido de nuevo a número
        }
        isbn += (int)(Math.random() * 10); // Agrega un dígito aleatorio al ISBN
        return generarISBN(contador + 1, isbn); // Llama recursivamente incrementando el contador
    }
```

El próximo método es el que permite insertar los códigos creados con el método anterior en el arreglo de ISBNs, recibiendo como parámetros el arreglo de tipo Long y un contador para saber en qué index se añadirá el elemento.

```
public static void insertarISBN(Long[] arreglo, int contISBNs) { // Insertar el ISBN generado en el arreglo
    int cont = 0;                                   // Contador para los 13 dígitos del ISBN
    String isbnAux = "";                            // String para ir guardando el ISBN
    long nuevoISBN = generarISBN(cont, isbnAux);    // Genera un nuevo ISBN
    arreglo[contISBNs] = nuevoISBN;           // Inserta el nuevo ISBN en la posición indicada por el contador
    System.out.println("ISBN generado: " + nuevoISBN + " en la posición " + contISBNs);   // Incrementa el contador para la próxima
}
```

Por otro lado, el algoritmo de divide y vencerás se implementó en la búsqueda binaria de los algoritmos de búsqueda y ordenamiento, entonces vayamos a ellos.

Algoritmos de búsqueda y ordenamiento:

Si bien hay muchos algoritmos de ordenamiento y de búsqueda, para hacer el proyecto se seleccionaron únicamente el algoritmo de ordenamiento por selección y por ordenamiento de burbuja, mientras que en los buscadores se utilizaron la búsqueda secuencial y la búsqueda binaria.

En este caso se decidió asignar el ordenamiento por selección como una opción individual para ordenar el arreglo de ISBNs, pero el método bubble sort se utilizará de forma predeterminada para la búsqueda binaria.

```java
public static void selectionSort(Long[] arreglo, int contISBNs) {   //MÉTODO DE ORDENAMIENTO POR SELECCIÓN

    for (int i = 0; i < contISBNs - 1; i++) { // Recorre todos los elementos del arreglo
        for (int j = i + 1; j < contISBNs; j++) { // Compara el elemento actual con los siguientes
            if (arreglo[i] > arreglo[j]) { // Si el elemento actual es mayor al siguiente, se intercambian
                Long temp = arreglo[i]; // Intercambio
                arreglo[i] = arreglo[j];
                arreglo[j] = temp;
            }
        }
    }
}
```

El método de ordenamiento por selección no devuelve ningún valor porque únicamente modifica los elementos del arreglo, por lo que toma como atributos el arreglo con el que trabajará y en este caso también el contador para conocer cuántos elementos reales tiene almacenados.

Se utilizan dos ciclos for para recorrer el arreglo, el primero va de 0 al límite de elementos, y el segundo va una posición más adelante para comparar ese número con todos los demás que le siguen, intercambiando lugares si el primer número es mayor que los que le siguen. Así ya regresa el arreglo ordenado para cuando termina el primer ciclo de recorrer todos los elementos. (Nota: Sí, se pudo usar el método estático general **swap()** para hacer el intercambio, pero me dí cuenta de que no lo modifiqué cuando ya estaba redactando el documento, por lo que se queda para luego)

El método **bubbleSort()** es bastante parecido al anterior, con la mayor y más notable diferencia de que va comparando dos valores consecutivos y los intercambia poniendo el menor antes. Sin embargo, es un proceso mucho más largo y que tiene muchas iteraciones, porque ese proceso lo hace el segundo ciclo for, pero el primero se encarga de volver a recorrer el arreglo repitiendo los mismos pasos mencionados, hasta llegar al último elemento.

```java
public static void bubbleSort(Long[] arreglo, int contISBNs) {      // MÉTODO DE ORDENAMIENTO BURBUJA

    for(int i = 0; i < (contISBNs -1); i++) { // Recorre todos los elementos del arreglo
        for(int j = 0; j < (contISBNs -1 -i); j++) { // Compara el elemento actual con el siguiente
            if(arreglo[j] > arreglo[j+1]) { // Si el elemento actual es mayor al siguiente, se intercambian
                Long temp = arreglo[j]; // Intercambio
                arreglo[j] = arreglo[j + 1];
                arreglo[j + 1] = temp;
            }
        }
    }
}
```

Pasando a los métodos de búsqueda, el más sencillo de comprender es el secuencial, porque no necesita un arreglo ordenado y simplemente recorre cada elemento haciendo una comparación con los elementos hasta encontrar al indicado (o no hacerlo). No es conveniente para arreglos grandes porque puede tomar mucho tiempo e iteraciones.

```java
public static int busquedaSecuencial(Long[] arreglo, Long valorBuscado, int contISBNs) { // BÚSQUEDA SECUENCIAL
    // Está bien sencillo y bonito este método, pero no es eficiente para arreglos grandes
    for (int i = 0; i < contISBNs; i++) {
        if (arreglo[i] == valorBuscado.longValue()) {
            return i;   // Devuelve el índice donde se encontró el valor
        }
    }
    return -1; // Retorna -1 si no se encontró el valor
}
```

El algoritmo de búsqueda binaria se puede utilizar sin el método de divide y vencerás (por ello hay dos versiones en el código) pero en este caso se utilizará este porque lo que pretende este algoritmo es dividir tareas grandes y complejas en tareas más pequeñas y sencillas.

```java
public static int busquedaBinariaDYV(Long[] arreglo, Long valorBuscado, int inicio, int fin) { // BÚSQUEDA BINARIA DIVIDE Y VENCERÁS
    // Necesitamos que el arreglo esté ordenado, así que usamos un bubble sort antes de usarlo.

    if (inicio > fin) {
        return -1; // Caso base: no se encontró el valor
    }

    int medio = (inicio + fin) / 2; // Señala la mitad del arreglo

    if (arreglo[medio] == valorBuscado.longValue()) {
        return medio; // El valor está en la mitad

    } else if (arreglo[medio] < valorBuscado) {
        return busquedaBinariaDYV(arreglo, valorBuscado, medio + 1, fin); // Buscar en la mitad derecha

    } else {
        return busquedaBinariaDYV(arreglo, valorBuscado, inicio, medio - 1); // Buscar en la mitad izquierda
    }
}
```

El algoritmo emplea recursividad para hacer las comparaciones de partir el arreglo a la mitad y si coincide esa mitad con el elemento buscado entonces se devuelve (caso base), pero si no lo es entonces comparará si la mitad es mayor o menor al elemento buscado y mandará a buscar a la rama correspondiente del árbol llamando al mismo método (que volverá a partir la rama a la mitad hasta que encuentre el valor).

Otro caso base ocurre si ya se recorrió todo el árbol y no se encontró el elemento, entonces regresa -1.

Hash table y HashMap:

Para este apartado, se hizo el código orgánico de la tabla Hash, pero también se hizo la prueba implementándola directamente de las librerías de java. De esta manera, sólo se realizó el código de la Hashtable, pero para el HashMap se utilizó únicamente el importado.

Las tablas hash funcionan en tres partes principalmente: el dato con su llave y valor (en este caso se utilizarán nodos), la función Hash (que obtiene el index donde se almacenará el dato/nodo) y la tabla hash donde se almacenarán los elementos.

Además, como en las tablas Hash y los HashMap es común la colisión de datos por la asignación de la Hash Function, entonces puede utilizarse el método de las listas enlazadas en la tabla Hash para ir guardando los elementos en un mismo index (así se implementó aquí).

```
public class MiHashTable<K, V> { // ALMACENA SOCIOS

    private static class Nodo<K, V> {
        K clave;
        V valor;
        Nodo<K, V> siguiente;
        Nodo(K clave, V valor) {
            this.clave = clave;
            this.valor = valor;
        }
    }

    private Nodo<K, V>[] tablaHash;
    private int capacidad;
    private int tamaño;
```

Se creó la clase MiHashTable con dos tipos de datos genéricos (uno para la llave y otro para el valor).

También se creó el nodo con los mismos tipos de datos genéricos que maneja y sus atributos que almacenarán la llave, el valor y el apuntador al siguiente nodo (porque es para una lista enlazada), así como su constructor obligatorio que inicializa las variables.

Se declaran los atributos de la clase, como la tabla Hash (siendo un arreglo de nodos, que también puede ser un ArrayList para hacerlo más flexible), la capacidad que tendrá la tabla y el número de elementos que se irán añadiendo a la tabla. Además de dos constructores, uno en caso de que se indique el tamaño de la tabla, y otro que no (con un tamaño default de 10):

```
@SuppressWarnings("unchecked")
public MiHashTable(int capacidadInicial) {
    capacidad = capacidadInicial;
    tablaHash = new Nodo[capacidad]; // Inicializa el arreglo con el tamaño dado
    tamaño = 0;
}
public MiHashTable() { // Constructor por defecto
    capacidad = 10; // Capacidad inicial por defecto
    tablaHash = new Nodo[capacidad];
    tamaño = 0;
}
```

El resto de los métodos incluyen por ejemplo a la función Hash, que obtiene el index del nuevo elemento obteniendo el residuo de la división de la llave entre el total de tamaño de la tabla.

```
private int funcionHash(K clave) { // Obtiene el índice para la HashTable
    int hash = clave.hashCode();
    return Math.abs(hash) % capacidad; // Asegura que el índice sea positivo
}
```

En el método **put()** se pide la clave y el valor para crear un nuevo nodo, llamando a la función Hash para asignarle un index en la tabla Hash. Si encuentra que el nodo en esa posición tiene la misma clave/llave, entonces actualiza el valor, pero sino, sigue recorriendo los nodos hasta el final y agrega el nuevo nodo a la lista. Incrementando también el contador de elementos.

```java
public void put(K clave, V valor) { // Agrega o actualiza un par clave-valor
    int indice = funcionHash(clave);
    Nodo<K, V> actual = tablaHash[indice];

    while (actual != null) {
        if (actual.clave.equals(clave)) {
            actual.valor = valor;
            return;
        }
        actual = actual.siguiente;
    }
    Nodo<K, V> nuevo = new Nodo<>(clave, valor);
    nuevo.siguiente = tablaHash[indice];
    tablaHash[indice] = nuevo;
    tamaño++;
}
```

El resto de métodos como **getKey(), getValue()** y **remove()** son muy parecidos porque recorren los elementos de la tabla igual que en put(), pero en lugar de actualizar o agregar un valor, simplemente regresa el valor del nodo que tenga la llave ingresada, y al revés con getValue(); regresando null si no encontró un elemento que coincidiera.

```java
public V getValue(K clave) { // Obtiene el valor asociado a la clave
    int indice = funcionHash(clave);
    Nodo<K, V> actual = tablaHash[indice];

    while (actual != null) {
        if (actual.clave.equals(clave)) {
            return actual.valor;
        }
        actual = actual.siguiente;
    }
    return null; // No encontrado
}
```

Como se mencionó, el procedimiento para **remove()** es similar, con la diferencia principal de actualizar la referencia del nodo anterior para que ya no apunte al elemento y se elimine por el recolector de basura de java. Además de, por supuesto, reducir en uno la variable que lleva la cuenta de los elementos de la tabla.

```java
public void remove(K clave) { // Elimina un elemento por su clave
    int indice = funcionHash(clave);
    Nodo<K, V> actual = tablaHash[indice];
    Nodo<K, V> anterior = null;

    while (actual != null) {
        if (actual.clave.equals(clave)) {
            if (anterior == null) {
                tablaHash[indice] = actual.siguiente;
            } else {
                anterior.siguiente = actual.siguiente;
            }
            tamaño--;
            return;
        }
        anterior = actual;
        actual = actual.siguiente;
    }
}
```

El resto de métodos son principalmente de hacer uso de los métodos anteriores para verificar si la tabla está vacía, obtener el número de elementos de la tabla, entre otros.

• Código JAVA

Nota: El código completo del Main ronda las 568 líneas, por lo que se incluyen únicamente capturas generales.

- *Main.java -*

```java
// ===== Clase principal - Sistema de Gestión Editorial =====//

import java.util.*; // Importamos librerías

public class Main {

    // MENÚS SECUNDARIOS
    public static void menuMonticulo(int seleccion, Objeto[] monticulo) {       // Método para el menú de montículo···
    public static void menuPila(int seleccion, Pila pila) {                     // Método para el menú de pila···
    public static void menuListaEnlazada(int seleccion, ListaEnlazada<String> listaEnlazada) { // Método para el menú de lista enlazada
    public static void menuArbolBinario(int seleccion, ArbolBinario arbolito) { // Método para el menú de árbol binario···
    public static void menuRecursividadOrdenamientoBusqueda(int seleccion, int contISBNs, Long[] isbns) {   // Método para el menú de mé
    public static void menuHashTablesHashMap(int seleccion, MiHashTable<Integer, String> tablaHash, HashMap<Integer, String> hashMap) {

    // MÉTODOS AUXILIARES GENERALES
    public static <E> void mostrarArreglo(E[] arreglo) {        // Método para mostrar arreglos. no imprime null···
    public static <E> void swap(E[] arreglo, int a, int b) {    //Intercambiar elementos en el arreglo···
    public static <E> int getIndex(E objeto, E[] arreglo) {     //Obtener un index del arreglo···
    public static <E> int getSize(E[] arreglo) {                // Obtener el num. de elementos reales del arreglo···
```

```java
    Run main | Debug main | Run | Debug
    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);                     // Creamos el scanner para leer las entradas
        Objeto[] monticulo = new Objeto[100];                       // Declarar montículo. Le ponemos tamaño grande
        Pila pila = new Pila(capacity:100);                         // Creamos una pila
        ListaEnlazada<String> listaEnlazada = new ListaEnlazada<>();    // Creamos una lista enlazada
        ArbolBinario arbolito = new ArbolBinario();                 // Creamos un árbol binario
        //Hashtable<Integer, String> tablaHash = new Hashtable<>();       // Creamos una tabla hash
        MiHashTable<Integer, String> socios = new MiHashTable<>();  // Creamos una tabla hash personalizada
        HashMap<Integer, String> hashMap = new HashMap<>();             // Creamos un HashMap
        Long isbns[] = new Long[50];                                // Arreglo de ISBNs
        boolean ciclo = true;
        int contISBNs = 0; //ContISBNs cuenta los ISBNs almacenados en el arreglo

        // MENÚ PRINCIPAL
        while(ciclo) {
            System.out.println(x:"\n ==== SISTEMA DE GESTIÓN EDITORIAL ====");
            System.out.print("1) Agendar evento                (Montículo)" +                          // Montículo
                             "\n2) Historial de eventos         (Pila)" +                               // Pila
                             "\n3) Propuestas de proyecto       (Lista enlazada)" +                     // Lista enlazada
                             "\n4) Gestión de empleados         (Árbol binario)" +                      // Árbol binario
                             "\n5) Catálogo de obras terminadas (Recursividad, DYV, Ordenamiento/Búsqueda)" + // Recursividad, DYV, Méto
                             "\n6) Catálogo general             (Tablas Hash y HashMap)" +              // Tablas Hash y HashMap
                             "\n7) Ver colaboraciones           (Grafos)" +                             // Grafos
                             "\n8) Salir \n=> ");
            int seleccion = input.nextInt();
            input.nextLine(); // limpiar buffer
```

```java
            switch(seleccion) { //MENÚS SENCUNDARIOS

                case 1: //MONTÍCULO···
                case 2: // PILA···
                case 3: // LISTA ENLAZADA···
                case 4: // ÁRBOL BINARIO···
                case 5: // RECURSIVIDAD, DIVIDE Y VENCERÁS Y MÉTODOS DE ORDENAMIENTO/BÚSQUEDA···
                case 6: // TABLAS HASH Y HASHMAP···
                case 7: // GRAFOS···
                case 8: // SALIR···
                default: // VALIDAR DATO
                    System.out.println(x:"Opción no válida.");
            }
        }
    }
}
```

Con más detalle:

```java
// MENÚS SECUNDARIOS
public static void menuMonticulo(int seleccion, Objeto[] monticulo) {       // Método para el menú de montículo
    Scanner input = new Scanner(System.in);
    boolean ciclo = true;
    while (ciclo) {
        System.out.println(x:"\n === 'AGENDAR EVENTO' ===");
        System.out.print("1) Agregar evento a la cola " +           // enqueue
                         "\n2) Mostrar evento actual " +            // front
                         "\n3) Completar evento actual" +           // dequeue
                         "\n4) Mostrar todos los eventos en cola" + // display
                         "\n5) Volver \n=> ");
        seleccion = input.nextInt();
        input.nextLine(); // limpiar buffer

        switch(seleccion) {
            case 1: // AGREGAR EVENTO (ENQUEUE)···
            case 2: // MOSTRAR EVENTO ACTUAL (FRONT)···
            case 3: // COMPLETAR EVENTO ACTUAL (DEQUEUE)···
            case 4: // MOSTRAR TODOS LOS EVENTOS EN COLA (DISPLAY)···
            case 5: // VOLVER···
            default: // VALIDACIÓN DE DATO···
        }
    }
}
```

```java
        switch(seleccion) {
            case 1: // AGREGAR EVENTO (ENQUEUE)
                System.out.println(x:"Nombre evento: ");
                String dato = input.nextLine();
                System.out.println(x:"Prioridad del evento (1 en delante): ");
                int prioridad = input.nextInt();
                input.nextLine(); // Limpiar buffer
                Monticulo.enqueue(new Objeto(dato, prioridad), monticulo); //Crea el objeto, añade los datos y lo agrega al arreglo
                break;
            case 2: // MOSTRAR EVENTO ACTUAL (FRONT)
                Monticulo.front(monticulo);
                break;
            case 3: // COMPLETAR EVENTO ACTUAL (DEQUEUE)
                Monticulo.dequeue(monticulo);
                break;
            case 4: // MOSTRAR TODOS LOS EVENTOS EN COLA (DISPLAY)
                Monticulo.display(monticulo);
                break;
            case 5: // VOLVER
                ciclo = false;
                break;
            default: // VALIDACIÓN DE DATO···
        }
    }
}
```

```java
                    case 1: //MONTÍCULO
                        menuMonticulo(seleccion, monticulo);
                        break;
                    case 2: // PILA
                        menuPila(seleccion, pila);
                        break;
                    case 3: // LISTA ENLAZADA
                        menuListaEnlazada(seleccion, listaEnlazada);
                        break;
                    case 4: // ÁRBOL BINARIO
                        menuArbolBinario(seleccion, arbolito);
                        break;
                    case 5: // RECURSIVIDAD, DIVIDE Y VENCERÁS Y MÉTODOS DE ORDENAMIENTO/BÚSQUEDA
                        menuRecursividadOrdenamientoBusqueda(seleccion, contISBNs, isbns);
                        break;
                    case 6: // TABLAS HASH Y HASHMAP
                        menuHashTablesHashMap(seleccion, socios, hashMap);
                        break;
                    case 7: // GRAFOS···
                    case 8: // SALIR···
                    default: // VALIDAR DATO
                        System.out.println(x:"Opción no válida.");
                }
            }
        }
    }
```

- *Recursividad.java -*

```
public class Recursividad {

    public static long generarISBN(int contador, String isbn) { // GENERAR ISBN DE 13 DÍGITOS CON
RECURSIVIDAD
        if (contador == 13) {
            return Long.parseLong(isbn); // Regresa el ISBN generado, convertido de nuevo a número
        }
        isbn += (int)(Math.random() * 10); // Agrega un dígito aleatorio al ISBN
        return generarISBN(contador + 1, isbn); // Llama recursivamente incrementando el contador
    }
    public static void insertarISBN(Long[] arreglo, int contISBNs) { // Insertar el ISBN generado en el
arreglo
        int cont = 0;                                   // Contador para los 13 dígitos del ISBN
        String isbnAux = "";                            // String para ir guardando el ISBN
        long nuevoISBN = generarISBN(cont, isbnAux);    // Genera un nuevo ISBN
        arreglo[contISBNs] = nuevoISBN;           // Inserta el nuevo ISBN en la posición indicada por el
contador
        System.out.println("ISBN generado: " + nuevoISBN + " en la posición " + contISBNs);
// Incrementa el contador para la próxima inserción
    }
}
```

- *OrdenamientoBusqueda.java -*

```
public class OrdenamientoBusqueda {

    // METODOS DE ORDENAMIENTO
    public static void selectionSort(Long[] arreglo, int contISBNs) {   //MÉTODO DE ORDENAMIENTO POR
SELECCIÓN

        for (int i = 0; i < contISBNs - 1; i++) {
            for (int j = i + 1; j < contISBNs; j++) {
                if (arreglo[i] > arreglo[j]) {
                    Long temp = arreglo[i];
                    arreglo[i] = arreglo[j];
                    arreglo[j] = temp;
                }
            }
        }
    }
    public static void bubbleSort(Long[] arreglo, int contISBNs) {      // MÉTODO DE ORDENAMIENTO BURBUJA

        for(int i = 0; i < (contISBNs -1); i++) { // Recorre todos los elementos del arreglo
            for(int j = 0; j < (contISBNs -1 -i); j++) {
                if(arreglo[j] > arreglo[j+1]) { // Si el elemento actual es mayor al siguiente, se
intercambian
                    Long temp = arreglo[j]; // Intercambio
                    arreglo[j] = arreglo[j + 1];
                    arreglo[j + 1] = temp;
                }
            }
        }
    }
    /*********************************************
        Nota: Para usar la búsqueda binaria es necesari tener el arreglo ordenado.
        Por eso se invocan los métodos de ordenamiento antes de llamar un método de búsqueda.
    *********************************************/

    // MÉTODOS DE BÚSQUEDA
```

```
    public static int busquedaSecuencial(Long[] arreglo, Long valorBuscado, int contISBNs) { // BÚSQUEDA
SECUENCIAL
        // Está bien sencillo y bonito este método, pero no es eficiente para arreglos grandes
        for (int i = 0; i < contISBNs; i++) {
            if (arreglo[i] == valorBuscado.longValue()) {
                return i;   // Devuelve el índice donde se encontró el valor
            }
        }
        return -1; // Retorna -1 si no se encontró el valor
    }
    public static int busquedaBinaria(int[] arreglo, int valorBuscado) { // BÚSQUEDA BINARIA
        // Necesitamos que el arreglo esté ordenado, así que usamos un método de ordenamiento antes de
usarlo.
        // Arrays.sort(arreglo); // Se puede usar el método Arrays para ordenar el arreglo, pero ese no es
el chiste del ejercicio.

        int inicio = 0;                 // Indica el inicio
        int fin = arreglo.length - 1;   // Indica el final

        while (inicio <= fin) {             // Mientras no se llegue al final
            int medio = (inicio + fin) / 2; // Señala la mitad del arreglo

            if (arreglo[medio] == valorBuscado) { // El valor está en la mitad
                return medio;

            } else if (arreglo[medio] < valorBuscado) {
                inicio = medio + 1; // Buscar en la mitad derecha

            } else {
                fin = medio - 1; // Buscar en la mitad izquierda
            }
        }
        return -1; // Retorna -1 si no se encontró el valor
    }
    public static int busquedaBinariaDYV(Long[] arreglo, Long valorBuscado, int inicio, int fin) { //
BÚSQUEDA BINARIA DIVIDE Y VENCERÁS
        // Necesitamos que el arreglo esté ordenado, así que usamos un bubble sort antes de usarlo.

        if (inicio > fin) {
            return -1; // Caso base: no se encontró el valor
        }
        int medio = (inicio + fin) / 2; // Señala la mitad del arreglo
        if (arreglo[medio] == valorBuscado.longValue()) {
            return medio; // El valor está en la mitad
        } else if (arreglo[medio] < valorBuscado) {
            return busquedaBinariaDYV(arreglo, valorBuscado, medio + 1, fin); // Buscar en la mitad derecha
        } else {
            return busquedaBinariaDYV(arreglo, valorBuscado, inicio, medio - 1); // Buscar en la mitad
izquierda
        }
    }
}
```

- *ArbolBinario.java -*

```
//==== Clase para gestionar el Árbol Binario de Búsqueda ====//

public class ArbolBinario<E> {
    Nodo<E> raiz;       // Definir un apuntador a la raiz
    boolean eliminado;  // Para ver si se eliminó un elemento en eliminar()

    public ArbolBinario() { // Constructor
```

```
        raiz = null; // Sin raiz al principio
    }

    // Insertar nodo en el árbol
    public void insertar(int nombre, E dato) {

        Nodo<E> nuevo = new Nodo<E>(nombre, dato); // Crear nuevo nodo
        if (raiz == null) { // Si no hay raiz (esta vacio el arbol)
            raiz = nuevo;   // Asigna al nuevo nodo como raiz
            return;
        }

        Nodo<E> auxiliar = raiz; // Declara dos variables temporales para pasarse los valores
        Nodo<E> padre; // Inician con el valor de la raiz
        while (true) {
            padre = auxiliar; // El nodo que indica al padre toma el valor al que llega el auxiliar
            if (nombre < auxiliar.nombre) { // Si el nodo nuevo es menor al nodo actual que lleva el
auxiliar
                auxiliar = auxiliar.hijoIzquierdo;  // Asigna al auxiliar el valor del nodo izq del nodo
que lleva actualmente
                if (auxiliar == null) {             // Si es nulo es que llegó al último nodo de la rama
                    padre.hijoIzquierdo = nuevo; // Le indica al padre que tome al nuevo nodo como su hijo
izq
                    return;
                }
            } else {        // Si el nuevo nodo es mayor o igual al nodo actual del aux
                auxiliar = auxiliar.hijoDerecho; // Se mueve al siguiente nodo derecho del nodo actual que
lleva
                if (auxiliar == null) {         // Si es nulo es que llegó al final de la rama
                    padre.hijoDerecho = nuevo; // Indica al padre que asigne al nuevo como su hijo derecho
                    return;
                }
            } // Es como que el auxiliar va tanteando primero el camino y cuando está seguro le pasa la
posicion al padre
        }
    }
    // Buscar nodo en el árbol
    public boolean buscar(int id) {
        return buscarPreorden(raiz, id); // Le pasamos la raiz y el id
    }
    // Recorrido PreOrden para buscar elementos
    public boolean buscarPreorden(Nodo<E> r, int id) {
        if (r != null) {
            if (r.nombre == id) {  // si coincide
                System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra el contenido
                return true; // Devuelve true porque lo encontro
            }
            // Si no está en el nodo actual, buscamos en hijos
            if (buscarPreorden(r.hijoIzquierdo, id)) {
                return true;       // Si lo encuentra devuelve true
            }
            if (buscarPreorden(r.hijoDerecho, id)) {
                return true;       // Si lo encuentra devuelve true
            }
        }
        return false; // no se encontró
    }
    // Eliminar nodo en el árbol
    public boolean eliminar(int dato) {
        eliminado = false;              // reiniciamos el indicador
        eliminar(raiz, dato);           // eliminamos en la raiz
        return eliminado;               // devolvemos si se eliminó o no
    }
    // Eliminar2.0, que sirve para hacer todo el proceso
    private Nodo<E> eliminar(Nodo<E> raiz, int id) {
```

```
        if (estaVacio()) { // Está vacío el árbol
            return null; // No se encontró el nodo
        }

        if (id < raiz.nombre) {
            raiz.hijoIzquierdo = eliminar(raiz.hijoIzquierdo, id); // Viaja a través de la subrama
izquierda
        } else if (id > raiz.nombre) {
            raiz.hijoDerecho = eliminar(raiz.hijoDerecho, id); // Viaja por la subrama derecha
        } else { // Si coincide sigue el proceso
            eliminado = true; // Se encontró el Nodo

            // Caso 1: sin hijos
            if (raiz.hijoIzquierdo == null && raiz.hijoDerecho == null) {
                return null;
            }
            // Caso 2: un solo hijo
            else if (raiz.hijoIzquierdo == null) {
                return raiz.hijoDerecho;
            } else if (raiz.hijoDerecho == null) {
                return raiz.hijoIzquierdo;
            }
            // Caso 3: dos hijos
            else {
                Nodo<E> sucesor = minimo(raiz.hijoDerecho);
                raiz.dato = sucesor.dato; // Le pasamos los datos del sucesor
                raiz.nombre = sucesor.nombre; // Pasamos los datos del sucesor
                raiz.hijoDerecho = eliminar(raiz.hijoDerecho, sucesor.nombre);
            }
        }
        return raiz;
    }
    // Encuentra el valor mínimo (sucesor inorden)
    private Nodo<E> minimo(Nodo<E> nodo) {
        while (nodo.hijoIzquierdo != null) { // Selecciona el último elemento de la subrama izquierda
            nodo = nodo.hijoIzquierdo;
        }
        return nodo;
    }

    // Comprueba si la raiz está vacía
    public boolean estaVacio() { return raiz == null; }

    // Wrappers públicos
    public void preorden() { preorden(raiz); }
    public void inorden() { inorden(raiz); }
    public void posorden() { posorden(raiz); }

    // Recorrido PreOrden
    private void preorden(Nodo<E> r) {
        if (r != null) {
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
            preorden(r.hijoIzquierdo);
            preorden(r.hijoDerecho);
        }
    }
    // Recorrido InOrden
    private void inorden(Nodo<E> r) {
        if (r != null) {
            inorden(r.hijoIzquierdo);
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
            inorden(r.hijoDerecho);
        }
    }
    // Recorrido PosOrden
```

```
    private void posorden(Nodo<E> r) {
        if (r != null) {
            posorden(r.hijoIzquierdo);
            posorden(r.hijoDerecho);
            System.out.println("ID: " + r.nombre + " - " + r.dato); // Muestra "ID: 2 - Rosa"
        }
    }
}
```

- *Monticulo.java -*

```
//==== Clase para gestionar la Cola Priorizada (MONTÍCULO) ====//

public class Monticulo {

    // MÉTODOS ESTÁTICOS DEL MONTÍCULO
    public static void enqueue(Objeto objeto, Objeto[] arreglo) { //Agregar elemento al arreglo

        if (Main.getSize(arreglo) == 0) {    //Si la lista está vacía agregarlo en la posición 1
            arreglo[1] = objeto;    //Agregarlo al arreglo
            objeto.setHijoIzquierdo(2); //2 - 2*n
            objeto.setHijoDerecho(3); //3 - 2*n +1
            objeto.setPadre(0);//No tiene padre aún

        } else {    //No es el primer elemento

            arreglo[Main.getSize(arreglo) + 1] = objeto; //Agregarlo primeramente
            //Asignar a sus hijos
            objeto.setHijoIzquierdo(Main.getIndex(objeto, arreglo)*2); //2*n
            objeto.setHijoDerecho((Main.getIndex(objeto, arreglo)*2) +1); //2*n +1
            objeto.setPadre(Main.getIndex(objeto, arreglo)/2);  //Asignarle padre

            while(true) {
                if (objeto.getPrioridad() < arreglo[objeto.getPadre()].getPrioridad()) { //Si es menor la
prioridad del nuevo
                    int index_inicial_objeto = Main.getIndex(objeto, arreglo);   //Guarda la posición inicial
del objeto antes de cambiar
                    Main.swap(arreglo, Main.getIndex(objeto, arreglo), objeto.getPadre()); //Cambia lugar con
el padre en el arreglo

                    //Si quedó en la posición 1 del arreglo no se modifica su padre
                    if (Main.getIndex(objeto, arreglo) == 1) {
                    } else {    //Si no es el 1, reasignarle padre
                        objeto.setPadre(Main.getIndex(objeto, arreglo)/2);
                    }

                    //Actualiza a sus hijos
                    objeto.setHijoIzquierdo(Main.getIndex(objeto, arreglo)*2); //2*n
                    objeto.setHijoDerecho((Main.getIndex(objeto, arreglo)*2) +1); //2*n +1

                    //Actualizar datos del padre anterior, que se movió al lugar inicial del objeto
                    arreglo[index_inicial_objeto].setPadre(index_inicial_objeto/2); //actualizarle el padre al
padre
                    arreglo[index_inicial_objeto].setHijoIzquierdo(index_inicial_objeto*2); //Actualizar hijo
izquierdo del padre
                    arreglo[index_inicial_objeto].setHijoDerecho((index_inicial_objeto*2) +1); //Actualizar
hijo derecho del padre

                } else { //Si el objeto no es menor que su padre no se mueve
                    break;
                }
```

```
            }
        }
        System.out.println("Registro exitoso!");
    }
    public static void dequeue(Objeto[] arreglo) { //Eliminar el primer elemento del arreglo

        if (Main.getSize(arreglo) == 0) { //El arreglo está vacío
            System.out.println("No hay eventos registrados.");
        } else {
            Objeto eliminado = arreglo[1]; //Guardar en una variable el objeto eliminado
            //Recorrer los elementos un lugar en el arreglo
            for (int i = 1; i < Main.getSize(arreglo); i++) { //De 1 a 'tamaño del arreglo' -1
                arreglo[i] = arreglo[i+1];
            }
            arreglo[Main.getSize(arreglo)] = null; //Se actualiza el último espacio para que sea null
            System.out.println("Evento completado: " + eliminado.getDato());
        }

    }
    public static void front(Objeto[] arreglo) { //Mostrar el primer elemento

        if (Main.getSize(arreglo)== 0) { //Si el arreglo está vacío
            System.out.println("No hay eventos registrados.");
        } else {
            System.out.println("Evento: " + arreglo[1].getDato() + " Prioridad: " + arreglo[1].getPrioridad());
        }
    }
    public static void display(Objeto[] arreglo) {
        if (Main.getSize(arreglo) == 0) { //Si el arreglo está vacío
            System.out.println("Aún no hay eventos.");
        } else { //Recorrer cada elemento
            System.out.println(" <= Actual | En espera =>");
            for (Objeto elemento: arreglo) {
                if (elemento != null) { //Si no está vacío el espacio lo muestra
                    System.out.print(elemento.getDato() + "(" + elemento.getPrioridad() + ") "); //Muestra el
dato y su prioridad entre paréntesis
                }
            }
            System.out.println();
        }
    }
}
```

- *Objeto.java -*

```
//==== Clase de Cola Priorizada ====//

public class Objeto<E> {

    // ATRIBUTOS
    private E dato;
    private int prioridad;
    private int padre, hijoIzquierdo, hijoDerecho;

    public Objeto(E dato, int prioridad) { // Constructor normal
        this.dato = dato;
        this.prioridad = prioridad;
    }
    public Objeto(Objeto<E> objeto) { // Constructor copia
        this.dato = objeto.getDato();
        this.prioridad = objeto.getPrioridad();
        this.padre = objeto.getPadre();
```

```
        this.hijoIzquierdo = objeto.getHijoIzquierdo();
        this.hijoDerecho = objeto.getHijoDerecha();
    }
    //Getters
    public E getDato() { return this.dato; }
    public int getPrioridad() { return this.prioridad; }
    public int getPadre() {return this.padre;}
    public int getHijoIzquierdo() {return this.hijoIzquierdo;}
    public int getHijoDerecha() {return this.hijoDerecho;}

    //Setters
    public void setDato(E dato) { this.dato = dato; }
    public void setPrioridad(int prioridad) { this.prioridad = prioridad; }
    public void setPadre(int padre) { this.padre = padre; }
    public void setHijoIzquierdo(int hijo) { this.hijoIzquierdo = hijo; }
    public void setHijoDerecho(int hijo) { this.hijoDerecho = hijo; }
}
```

Nota: El constructor copia planeaba utilizarse para un procedimiento del montículo, pero al final no se necesitó, entonces puede removerse o dejarse para un uso futuro sin ningún problema.

Su propósito es permitir crear otro objeto con los mismos datos que el que se le pase.

- *Pila.java -*

```
//==== Clase de Pila====//

public class Pila {
    // ATRIBUTOS
    private String[] data;  // Para guardar el arreglo de Strings
    private int top;        // Indica el lugar donde está el último elemento
    private int capacity;   // Indica la capacidad del arreglo

    // CONSTRUCTOR
    public Pila(int capacity) {
        this.capacity = capacity;
        data = new String[capacity]; // Crear arreglo con la capacidad asignada
        top = -1; // Pila vacía
    }

    //MÉTODOS
    public boolean isEmpty() { // Ver si está vacía la pila
        return top == -1; // -1 es el valor para la pila vacía
    }
    public boolean isFull() { // Ver si ya se llenó la pila
        return top == capacity -1; //Si el último elemento es igual a la capacidad -1
    }
    public void push(String x) { // Agregar elemento a la pila
        if (isFull()) { // Si está lleno no agrega nada porque da error
            System.out.println("Límite alcanzado. No se pueden agregar más elementos. " + x);
            return;
        }
        data[++top] = x; // Lo agrega al próximo espacio disponible
        System.out.println("Registro exitoso!");
    }
    public String pop() { // Eliminar el último elemento
        if (isEmpty()) { // Si está vacío no puede eliminar nada
            return "Historial vacío."; //indicador de error
        }
        return "Eliminado: " + data[top--]; //Muestra el dato eliminado y le quita un espacio a la pila
```

```
    }
    public String peek() { // Muestra el último elemento ingresado
        if (isEmpty()) {
            return "Historial vacío.";
        }
        return data[top]; //Devuelve el último elemento ingresado
    }
    public void display() { // Muestra todos los elementos de la pila del más reciente al más antiguo
        if (isEmpty()) { // Si está vacío no puede mostrar nada
            System.out.println("Historial vacío.");
        } else {
            System.out.println("<- Reciente | Antiguo -> "); // Muestra de top a fondo de la pila
            for(int i = top; i >= 0; i--) { // Recorre todos los elementos de la pila
                System.out.print(data[i] + " - "); // Muestra los elementos
            }
            System.out.println();
        }
    }
}
```

- *ListaEnlazada.java -*

```
//==== Clase de Lista enlazada  ====//

public class ListaEnlazada<E> {
    //CLASE NODO
    private static class Node<E> {
        E data;
        Node<E> next;
        Node(E data) { this.data = data; } // Constructor de nodo
    }
    private Node<E> head; //Atributo/apuntador a la cabeza de la lista
    // MÉTODOS DE LA CLASE
    public void insert(E data) { // Agregar elemento al final de la pila
        Node<E> nuevo = new Node<>(data); // Crea un nodo y le pasa el dato
        if (head == null) { // Si no hay un elemento en la cabeza, agrega al nuevo dato
            head = nuevo;
        } else {
            Node<E> temp = head; // Guarda el nodo en una variable
            while (temp.next != null) {  // Recorre todos los elementos hasta llegar al último
                temp = temp.next;
            }
            temp.next = nuevo; // Actualiza la referencia al siguiente nodo
        }
        System.out.println("Registro exitoso!");
    }
    // Mostrar todos los elementos
    public void display() {
        Node<E> temp = head; // Guarda la cabeza en una variable
        if (temp == null) { // Mira si no hay elementos
            System.out.println("No se encuentran registros.");
        } else {
            while (temp != null) { // Recorre todos los elementos
                System.out.print(temp.data + " -> "); // Los muestra
                temp = temp.next;
            }
            System.out.println("null"); //Voy a dejar el null para que se vea que es una lista enlazada
        }
    }
    // Buscar elemento
    public boolean find(E dato) {
        Node<E> temp = head;
```

```
        while (temp != null) { //Repasa todos los elementos
            if (temp.data.equals(dato)) { // Si encuentra la coincidencia
                return true; // Elemento encontrado
            }
            temp = temp.next; //se mueve con la referencia
        }
        return false; // Elemento no encontrado
    }
    // Eliminar elemento
    public boolean delete(E dato) {
        if (head == null) return false; //Si no hay elementos regresa falso
        if (head.data.equals(dato)) { // Si el dato está en la cabeza
            head = head.next; // Mueve el apuntador al próximo nodo
            return true;
        }
        Node<E> temp = head; // Guarda la cabeza en una variable
        while (temp.next != null) { // recorre los elementos
            if (temp.next.data.equals(dato)) { // revisa la coincidencia
                temp.next = temp.next.next; // elimina el nodo
                return true;
            }
            temp = temp.next; // se desplaza
        }
        return false; // no encontrado
    }
}
```

- *MiHashTable.java -*

```
public class MiHashTable<K, V> { // ALMACENA SOCIOS

    private static class Nodo<K, V> {
        K clave;
        V valor;
        Nodo<K, V> siguiente;
        Nodo(K clave, V valor) {
            this.clave = clave;
            this.valor = valor;
        }
    }

    private Nodo<K, V>[] tablaHash;
    private int capacidad;
    private int tamaño;

    // CONSTRUCTOR
    @SuppressWarnings("unchecked")
    public MiHashTable(int capacidadInicial) {
        capacidad = capacidadInicial;
        tablaHash = new Nodo[capacidad]; // Inicializa el arreglo con el tamaño dado
        tamaño = 0;
    }
    public MiHashTable() { // Constructor por defecto
        capacidad = 10; // Capacidad inicial por defecto
        tablaHash = new Nodo[capacidad];
        tamaño = 0;
    }

    private int funcionHash(K clave) { // Obtiene el índice para la HashTable
        int hash = clave.hashCode();
        return Math.abs(hash) % capacidad; // Asegura que el índice sea positivo
    }
    public void put(K clave, V valor) { // Agrega o actualiza un par clave-valor
        int indice = funcionHash(clave);
```

```
        Nodo<K, V> actual = tablaHash[indice];

        while (actual != null) {
            if (actual.clave.equals(clave)) {
                actual.valor = valor;
                return;
            }
            actual = actual.siguiente;
        }
        Nodo<K, V> nuevo = new Nodo<>(clave, valor);
        nuevo.siguiente = tablaHash[indice];
        tablaHash[indice] = nuevo;
        tamaño++;
    }
    public V getValue(K clave) { // Obtiene el valor asociado a la clave
        int indice = funcionHash(clave);
        Nodo<K, V> actual = tablaHash[indice];

        while (actual != null) {
            if (actual.clave.equals(clave)) {
                return actual.valor;
            }
            actual = actual.siguiente;
        }
        return null; // No encontrado
    }
    public K getKey(V valor) { // Obtiene la clave asociada a un valor
        for (int i = 0; i < 10; i++) { // Recorre toda la tabla Hash
            Nodo<K, V> actual = tablaHash[i];
            while (actual != null) {
                if (actual.valor.equals(valor)) { // Verifica si el primer elemento de la lista coincide
                    return actual.clave;
                }
                actual = actual.siguiente; // Recorre la lista
            }
        }
        return null; // No encontrado
    }
    public void remove(K clave) { // Elimina un elemento por su clave
        int indice = funcionHash(clave);
        Nodo<K, V> actual = tablaHash[indice];
        Nodo<K, V> anterior = null;

        while (actual != null) {
            if (actual.clave.equals(clave)) {
                if (anterior == null) {
                    tablaHash[indice] = actual.siguiente;
                } else {
                    anterior.siguiente = actual.siguiente;
                }
                tamaño--;
                return;
            }
            anterior = actual;
            actual = actual.siguiente;
        }
    }
    public boolean containsKey(K clave) {       // Verifica si una clave existe en la tabla
        return getValue(clave) != null;
    }
    public boolean containsValue(V valor) {     // Verifica si un valor existe en la tabla
        return getKey(valor) != null;
    }
    public int size() { return tamaño; }               // Devuelve el número de elementos en la tabla
    public boolean isEmpty() { return tamaño == 0; }   // Verifica si la tabla está vacía
```

```
    public void display() { // Muestra el contenido de la tabla Hash
        for (int i = 0; i < capacidad; i++) {
            Nodo<K, V> actual = tablaHash[i];
            if (actual != null) {
                System.out.print("Índice " + i + ": ");
                while (actual != null) {
                    System.out.print("[" + actual.clave + ": " + actual.valor + "] -> ");
                    actual = actual.siguiente;
                }
                System.out.println("null");
            }
        }
    }
    public Integer[] keySet() { // Devuelve un arreglo con todas las claves (asumiendo que K es Integer)
        Integer[] claves = new Integer[tamaño];
        int index = 0;
        for (int i = 0; i < capacidad; i++) {
            Nodo<K, V> actual = tablaHash[i];
            while (actual != null) {
                claves[index++] = (Integer) actual.clave; // Cast a Integer
                actual = actual.siguiente;
            }
        }
        return claves;
    }
}
```

• Pruebas de uso

- *Menú principal:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento                (Montículo)
2) Historial de eventos          (Pila)
3) Propuestas de proyecto        (Lista enlazada)
4) Gestión de empleados          (Árbol binario)
5) Catálogo de obras terminadas  (Recursividad, DYV, Ordenamiento/Búsqueda)
6) Buscar libros                 (Tablas Hash y HashMap)
7) Ver colaboraciones            (Grafos)
8) Salir
=>
```

- *Salir:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento                (Montículo)
2) Historial de eventos          (Pila)
3) Propuestas de proyecto        (Lista enlazada)
4) Gestión de empleados          (Árbol binario)
5) Catálogo de obras terminadas  (Recursividad, DYV, Ordenamiento/Búsqueda)
6) Buscar libros                 (Tablas Hash y HashMap)
7) Ver colaboraciones            (Grafos)
8) Salir
=> 8
Saliendo... Hasta pronto!
PS C:\Users\LENOVO\OneDrive\Documentos\06 Escuela\02 TECMILENIO\04 Materias\07 Estructura de datos\Proyecto Final>
```

- *Validación de dato:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento                (Montículo)
2) Historial de eventos          (Pila)
3) Propuestas de proyecto        (Lista enlazada)
4) Gestión de empleados          (Árbol binario)
5) Catálogo de obras terminadas  (Recursividad, DYV, Ordenamiento/Búsqueda)
6) Buscar libros                 (Tablas Hash y HashMap)
7) Ver colaboraciones            (Grafos)
8) Salir
=> 0
Opción no válida.
```

- **MONTÍCULO**

- *Menú:*

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=>
```

- *Agregar evento a la cola (queue):*

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 1
Nombre evento:
Domingo feliz
Prioridad del evento (1 en delante):
5
Registro exitoso!
```

- *Mostrar evento actual (front):*

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 2
Evento: Feria del libro Prioridad: 2
```

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 2
No hay eventos registrados.
```

- *Completar evento actual (dequeue):*

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 3
Evento completado: Feria del libro
```

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 3
No hay eventos registrados.
```

- *Mostrar todos los eventos en cola (display):*

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 4
 <= Actual | En espera =>
Feria del libro(2) Domingo feliz(5)
```

```
 === 'AGENDAR EVENTO' ===
1) Agregar evento a la cola
2) Mostrar evento actual
3) Completar evento actual
4) Mostrar todos los eventos en cola
5) Volver
=> 4
Aún no hay eventos.
```

- **PILA**

- *Menú:*

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=>
```

- *Agregar evento al historial (push):*

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 1
Ingrese el nombre del evento => Domingo feliz
Registro exitoso!
```

- *Mostrar evento más reciente (peek):*

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 2
Evento más reciente: Domingo feliz
```

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 2
Evento más reciente: Historial vacío.
```

- *Borrar evento más reciente (pop):*

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 3
Eliminado: Domingo feliz
```

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 3
Historial vacío.
```

- *Mostrar historial de eventos (display):*

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 4
Historial de eventos:
<- Reciente | Antiguo ->
Feria del libro - Presentacion Antologia - Domingo feliz -
```

```
 === 'HISTORIAL DE EVENTOS' ===
1) Agregar evento al historial
2) Mostrar evento más reciente
3) Borrar evento más reciente
4) Mostrar historial de eventos
5) Volver
=> 4
Historial de eventos:
Historial vacío.
```

- **LISTA ENLAZADA**

- *Menú:*

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=>
```

- *Agregar propuesta (insert):*

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 1
Nombre de la propuesta => Domingo feliz
Registro exitoso!
```

- *Buscar propuesta (find):*

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 2
Nombre de la propuesta a buscar: Domingo feliz
Propuesta existente.
```

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 2
Nombre de la propuesta a buscar: Domingo feliz
No se encuentra la propuesta.
```

- *Borrar propuesta (delete):*

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 3
Propuesta a eliminar: Domingo feliz
Propuesta eliminada.
```

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 3
Propuesta a eliminar: Domingo feliz
Propuesta no encontrada.
```

- *Mostrar propuestas (display):*

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=> 4
Propuestas:
Domingo feliz -> Antologia Verano-invierno -> Cuento corto -> null
```

```
 === 'PROPUESTAS DE PROYECTO' ===
1) Agregar propuesta
2) Buscar propuesta
3) Borrar propuesta
4) Mostrar propuestas
5) Volver
=>
4
Propuestas:
No se encuentran registros.
```

- **Árbol binario**

- *Menú:*

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=>
```

- *Agregar empleado (insertar):*

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 1
Ingrese ID del empleado => 3
Ingrese el nombre del empleado:
Julia
```

- *Eliminar empleado (eliminar):*

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 2
Ingrese ID del empleado a eliminar => 9
Registro dado de baja exitosamente!
```

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 2
No hay registros existentes.
```

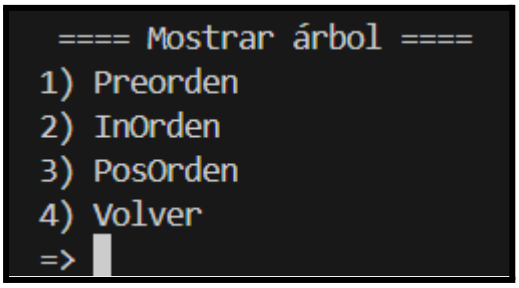
- *Buscar empleado (buscar):*

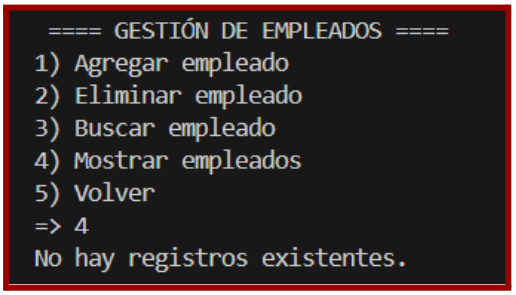
```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 3
Ingrese ID del empleado a buscar => 3
ID: 3 - Julia
Registro existente.
```

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 3
Ingrese ID del empleado a buscar => 1
Registro no encontrado.
```

```
 ==== GESTIÓN DE EMPLEADOS ====
1) Agregar empleado
2) Eliminar empleado
3) Buscar empleado
4) Mostrar empleados
5) Volver
=> 3
No hay registros existentes.
```

- *Mostrar empleados (mostrar):*





- **Recursividad, Divide y vencerás, Métodos de Ordenamiento y Búsqueda**

- *Menú:*

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 6
```

- *Generar ISBN (recursividad):*

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 1

 === 'GENERAR ISBN' ===
ISBN generado: 162368807643 en la posición 1
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 1

 === 'GENERAR ISBN' ===
No es posible generar más ISBNs.
```

- *Mostrar ISBN registrados (mostrar arreglo):*

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 2

 === 'ISBN REGISTRADOS' ===
9160689530596 162368807643
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 2

 === 'ISBN REGISTRADOS' ===
No hay registros para mostrar.
```

- *Ordenar registros (menor a mayor) (ordenamiento por selección):*

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 3

 === 'ORDENAR REGISTROS (MENOR A MAYOR)' ===
Registros ordenados:
162368807643 9160689530596
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 3

 === 'ORDENAR REGISTROS (MENOR A MAYOR)' ===
No hay registros para ordenar.
```

- *Búsqueda secuencial de ISBN (búsqueda secuencial):*

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 4

 === 'BÚSQUEDA SECUENCIAL' ===
Ingrese el ISBN a buscar => 9160689530596
ISBN encontrado en la posición: 1
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 4

 === 'BÚSQUEDA SECUENCIAL' ===
No hay registros para buscar.
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 4

 === 'BÚSQUEDA SECUENCIAL' ===
Ingrese el ISBN a buscar => 712738312
ISBN no encontrado.
```

- Búsqueda binaria de ISBN (búsqueda binaria con bubble sort interno):

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 5

 === 'BÚSQUEDA BINARIA' ===
Ingrese el ISBN a buscar => 8068497171142
ISBN encontrado en la posición: 1
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 5

 === 'BÚSQUEDA BINARIA' ===
No hay registros para buscar.
```

```
 === 'CATÁLOGO OBRAS' ===
1) Generar ISBN
2) Mostrar ISBN registrados
3) Ordenar registros (menor a mayor)
4) Búsqueda secuencial de ISBN
5) Búsqueda binaria de ISBN
6) Volver
=> 5

 === 'BÚSQUEDA BINARIA' ===
Ingrese el ISBN a buscar => 123213245
ISBN no encontrado.
```

- **HashTable y HashMap**

- *Menú:*

```
=== 'CATÁLOGO GENERAL' ===
1) Socios
2) Obras
3) Volver
=>
```

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=>
```

```
=== 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=>
```

```
¿Es una obra terminada o en proceso?
(1)Terminada
(2)En proceso
=>
```

- *Agregar socio (put):*

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 1
ID del socio => 34
Nombre del socio => Juan Perez
Socio agregado.
```

- *Agregar obra (put):*

```
¿Es una obra terminada o en proceso?
(1)Terminada
(2)En proceso
=> 1
ISBN de la obra => 12394342
Nombre de la obra => Cuentos estacionales
Obra agregada.
```

```
¿Es una obra terminada o en proceso?
(1)Terminada
(2)En proceso
=> 2
Nombre de la obra => Vulcanos Shop
Obra agregada.
```

- *Buscar socio (find):*

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 2
ID del socio a buscar => 34
Socio existente: Juan Perez
```

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 2
ID del socio a buscar => 2
No se encuentra el socio.
```

- *Borrar socio (remove):*

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 3
ID del socio a eliminar => 34
Socio eliminado.
```

```
=== 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 3
ID del socio a eliminar => 2
No se encuentra el socio.
```

- *Mostrar socios (display):*

```
 === 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 4
Socios registrados:
ID: 21 - Nombre: Ran BaGian
ID: 51 - Nombre: Qin ZiShen
ID: 34 - Nombre: Juan Perez
```

```
 === 'SOCIOS' ===
1) Agregar socio
2) Buscar socio
3) Borrar socio
4) Mostrar socios
5) Volver
=> 4
No hay registros existentes.
```

- *Buscar obra (find):*

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 2
Nombre de la obra a buscar:
Vulcanos Shop
Obra existente: Vulcanos Shop
```

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 2
Nombre de la obra a buscar:
Coleccion de cuentos estacionales
No se encuentra la obra.
```

- *Borrar obra (remove):*

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 3
Nombre de la obra a eliminar:
Cuentos estacionales
Obra eliminada.
```

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 3
No hay registros existentes.
```

- *Mostrar obras (display):*

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 4
Obras registradas:
Nombre: Cuentos estacionales
Nombre: Vulcanos Shop
```

```
 === 'OBRAS' ===
1) Agregar obra
2) Buscar obra
3) Borrar obra
4) Mostrar obras
5) Volver
=> 4
No hay registros existentes.
```

● Puntos de mejora

Si bien el código es funcional y cumple con lo fundamental, hay puntos que podrían mejorarse o implementarse en versiones futuras:

- Validación de datos y/o excepciones en todas las ocasiones que se introduce un dato.
- Mayor abstracción en el código al encapsular los menús y los elementos más repetitivos.
- Añadirle una interfaz gráfica.
- Permitir añadir más detalles a los distintos procesos (ejemplo: en las propuestas de proyecto se pueden pedir varios detalles como autor, título y detalles de la propuesta).
- Al momento de borrar o eliminar elementos, mostrar los elementos que puede eliminar (para reducir el riesgo de errores y ahorrar tiempo).
- Mantener actualizado el código. A la larga pueden surgir opciones más eficientes que las actuales.
- Modificar la pila y el montículo para que funcione con lista enlazada en lugar de con arreglo para que no tenga la limitación del espacio.
- Hacer de forma manual la implementación del HashMap y los grafos.

● Conclusión

Las estructuras de datos y la programación son herramientas muy poderosas que tienen el objetivo de resolver problemas de la manera más eficiente posible, y es trabajo del programador elegir la mejor opción para cada caso, considerando los pros y contras de cada estructura.

En este proyecto, se procuró elegir una estructura, método o algoritmo que vaya acorde o se pueda usar para resolver las tareas asignadas que se pueden encontrar en una editorial; aunque es importante señalar que en el mundo real esas opciones pueden ser ineficientes y obsoletas, pero como el propósito de este proyecto es conseguir implementar varios ejemplos de estos procesos, entonces esa es la razón por la cual se utilizan.

Mis experiencias con este proyecto y con la materia es que las estructuras de datos son muy geniales. Realmente me pareció divertido que la explicación (teoría) de los algoritmos sea relativamente fácil de entender, pero el problema viene a la hora de implementarlo (es hasta cierto punto irónico). Aunque hubo muchas ocasiones en las que fue complicado y me quería lavar el cerebro porque lo sentía quemado, fue una materia que se fue a mi top de favoritas porque lo sentí semejante a resolver un sudoku: es desafiante pero divertido y muy interesante.

Mis algoritmos favoritos (si tuviera que elegir algunos) serían los montículos, los árboles binarios (más en la teoría que en la práctica, porque borrar elementos me pareció tremendamente complicado) con sus recorridos, los métodos de ordenamiento y búsqueda y las HashTable y HashMap (son muy eficientes y prácticas, pero también son difíciles de implementar).

Un detalle aparte para añadir y que me parece excelente es que java ya tiene librerías que permiten utilizar muchos de los algoritmos y métodos, simplemente importandoles. Por supuesto el propósito de la actividad es que nosotros hagamos el planteamiento por nuestra cuenta y comprendamos cómo funciona, pero luego de hacerlo, podemos en un futuro utilizar las herramientas ya listas que tiene java para ahorrar tiempo y esfuerzo. Un detalle magnífico de la comunidad programadora, es que si alguien logró hacer algo, busca compartirlo para facilitar las cosas a los demás. Así se crea una red de apoyo muy grande donde todos aportan algo que puede ayudar a los demás.