AVANCE DE PROYECTO

“Sistema de Gestión Editorial”

Alumna: Jocelyn Flores Gutiérrez

Maestro: Adalberto Emmanuel Rojas Perea

Materia: Estructura de Datos

Fecha: 30/08/2025

TECMILENIO

● Índice de contenidos

Implementar y manipular diferentes estructuras de datos en Java, así como entender las aplicaciones prácticas de pilas, colas y listas 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 y los montículos, 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.

● Diseño

CLASES

Main.java — Clase principal donde se encuentran los menús de interacción con el usuario y los métodos estáticos para manipular el **montículo.** Ahí se crea la lista enlazada y la pila.

Objeto.java — Clase diseñada para la creación de los objetos que serán utilizados en el **montículo.** El objeto es capaz de guardar el dato, la prioridad y el index del padre y los hijos del objeto. Cuenta con setters y getters para manipular los atributos privados del objeto.

ListaEnlazada.java — Clase encargada de crear y gestionar la lista enlazada que se crea en el main. Incluye los métodos para interactuar con la lista y una clase nodo para almacenar el dato y la referencia al siguiente nodo.

Pila.java — Clase principal especializada en la configuración de la pila. Cuenta con varios métodos en su interior para manipular la pila, tales como 'display', 'push', 'pop' y 'peek', etc. Se distingue porque sus elementos se acomodan como una pila en la cual sólo puede eliminarse el último elemento ingresado y se agregan elementos al 'tope'.

Pilas: Utiliza pilas para guardar el historial de eventos pasados y mostrarlos en el orden de antiguo/reciente utilizando un arreglo de Strings. (Push, pop, peek, display)

- Historial de presentaciones pasadas.

Colas: Administra los eventos venideros con una cola de prioridad/**montículo** en un arreglo de objetos. (Enqueue, dequeue, front y display)

- Gestionar presentaciones/publicaciones de libros.

Listas: Almacena las propuestas de obras que requieren acceso aleatorio y tamaño variable en nodos. (Insert, delete, find y display)

- Propuestas de proyecto para aprobar.

Como se pretendió dar un ejemplo de las estructuras de datos y sus diferentes modos de implementación, el montículo pretende enseñar lo que es una 'cola' y cómo la modificación de sus reglas para añadir elementos puede convertirla en un 'montículo'; además de que tanto la cola como la pila utilizadas fueron implementadas con arreglos, para dejar el ejemplo del funcionamiento de las listas enlazadas a una lista enlazada simple.

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) Salir

MENÚ SECUNDARIO

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

• Implementación

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 Main para poder gestionar el montículo:

```java
// MÉTODOS ESTÁTICOS DEL MONTÍCULO
public static <E> void swap(E[] arreglo, int a, int b) { //Intercambiar elementos en el arreglo···
public static int getIndex(Objeto objeto, Objeto[] arreglo) { //Obtener un index del arreglo···
public static int getSize(Objeto[] arreglo) { //Obtener el num. de elementos del arreglo···
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 **swap** 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
}
```

Esto pretende ahorrar código en los procesos venideros (particularmente el push).

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;
}
```

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.

```
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.

```
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.

```
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
}
```

● Código JAVA

Nota: Se trabajó para que los códigos incluyeran comentarios en cada línea que expliquen el proceso paso a paso de las acciones que se tomaron.

- *Main.java -*

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

import java.util.*; // Importamos librerías para usar el scanner

public class Main {

    // MÉTODOS ESTÁTICOS DEL MONTÍCULO
    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
    }
    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 con .equals()
                return i; // devuelve el índice
            }
        }
        return -1; //Dará error porque -1 está fuera del rango
    }
    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;
    }
    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(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[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

            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

                    //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

                } 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 (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 < 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());
        }

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

```
        if (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 (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();
        }
    }

    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(100); // Creamos una pila
        ListaEnlazada<String> listaEnlazada = new ListaEnlazada<>(); // Creamos una lista
enlazada
        boolean ciclo = true;

        // MENÚ PRINCIPAL
        while(ciclo) {
            System.out.println("\n ==== SISTEMA DE GESTIÓN EDITORIAL ====");
            System.out.print("1) Agendar evento " +         // Montículo
                             "\n2) Historial de eventos " +  // Pila
                             "\n3) Propuestas de proyecto" + // Lista enlazada
                             "\n4) Salir \n=> ");
            int seleccion = input.nextInt();
            input.nextLine(); // limpiar buffer

            switch(seleccion) { //MENÚS SENCUNDARIOS

                case 1: //MONTÍCULO
                    System.out.println("\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)
```

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

            case 2: // PILA
                System.out.println("\n === 'HISTORIAL DE EVENTOS' ===");
                System.out.print("1) Agregar evento al historial " +        // push
                                 "\n2) Mostrar evento más reciente " +       // peek
                                 "\n3) Borrar evento más reciente" +         // pop
                                 "\n4) Mostrar historial de eventos" +       // display
                                 "\n5) Volver \n=> ");
                seleccion = input.nextInt();
                input.nextLine(); // limpiar buffer

                switch(seleccion) {
                    case 1: // AREGAR EVENTO AL HISTORIAL (PUSH)
                        System.out.print("Ingrese el nombre del evento => ");
                        String dato = input.nextLine();
                        pila.push(dato);
                        break;
                    case 2: // MOSTRAR EVENTO MÁS RECIENTE (PEEK)
                        System.out.println("Evento más reciente: " + pila.peek());
                        break;
                    case 3: // BORRAR EVENTO MÁS RECIENTE (POP)
                        System.out.println(pila.pop());
                        break;
                    case 4: // MOSTRAR HISTORIAL DE EVENTOS (DISPLAY)
                        System.out.println("Historial de eventos:");
                        pila.display();
                        System.out.println();
                        break;
                    case 5: // VOLVER
                        break;
                    default: // VALIDAR DATO
                        System.out.println("Opción inválida.");
                }
                break;
```

```
                case 3: // LISTA ENLAZADA
                    System.out.println("\n === 'PROPUESTAS DE PROYECTO' ===");
                    System.out.print("1) Agregar propuesta " +                  // insert
                                     "\n2) Buscar propuesta " +                  // find
                                     "\n3) Borrar propuesta" +                   // delete
                                     "\n4) Mostrar propuestas" +                 // display
                                     "\n5) Volver \n=> ");
                    seleccion = input.nextInt();
                    input.nextLine(); // limpiar buffer

                    switch(seleccion) {
                        case 1: // AGREGAR PROPUESTA (insert)
                            System.out.print("Nombre de la propuesta => ");
                            listaEnlazada.insert(input.nextLine());
                            break;
                        case 2: // BUSCAR PROPUESTA (find)
                            System.out.print("Nombre de la propuesta a buscar: ");
                            String dato = input.nextLine();

                            if (listaEnlazada.find(dato)) { // Si está el dato es true
                                System.out.println("Propuesta existente. ");
                            } else {
                                System.out.println("No se encuentra la propuesta. ");
                            }
                            break;
                        case 3: // BORRAR PROPUESTA (delete)
                            System.out.print("Propuesta a eliminar: ");
                            String datoEliminar = input.nextLine();

                            if(listaEnlazada.delete(datoEliminar)) { // Sí encontró el dato
y lo elimina
                                System.out.println("Propuesta eliminada.");
                            } else {
                                System.out.println("Propuesta no encontrada."); // No
encontró el dato
                            }
                            break;
                        case 4: // MOSTRAR PROPUESTAS (DISPLAY)
                            System.out.println("Propuestas:");
                            listaEnlazada.display();
                            break;
                        case 5: // VOLVER
                            break;
                        default: // VALIDACIÓN DE DATO
                            System.out.println("Opción inválida.");
                    }
                    break;
                case 4: // SALIR
                    System.out.println("Saliendo... Hasta pronto!");
                    ciclo = false;
                    break;
                default: // VALIDAR DATO
                    System.out.println("Opción no válida.");
            }
        }
    }
```

```
}
```

- *Objeto.java -*

```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
    }
}
```

● Pruebas de uso

- *Menú principal:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento
2) Historial de eventos
3) Propuestas de proyecto
4) Salir
=>
```

- *Salir:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento
2) Historial de eventos
3) Propuestas de proyecto
4) Salir
=> 4
Saliendo... Hasta pronto!
PS C:\Users\LENOVO\OneDrive\Documentos\06 Escuela\02 TECMILENIO\04 Materias\07 Estructura de datos\Avance proyecto>
```

- *Validación de dato:*

```
 ==== SISTEMA DE GESTIÓN EDITORIAL ====
1) Agendar evento
2) Historial de eventos
3) Propuestas de proyecto
4) 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.
```

● 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.