

**UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE SEDE SANTO DOMINGO**

**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS**

#### CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

**PERIODO** : 2023 noviembre – marzo 2024

**ASIGNATURA** : Estructura de datos

**TEMA**  : Guía

**NOMBRES** : Saldarriaga Salazar Gissela Elisa

: Torres Jenniffer

: Gualpa Mathias

**NIVEL-PARALELO**  : Tercero NRC: 16137

**DOCENTE** : Ing. Javier Cevallos

#### FECHA DE ENTREGA : 25/12/2023

**SANTO DOMINGO - ECUADOR**

**2023**

#### Índice

[Introducción 5](#_Toc158148406)

[Sistemas de objetivos 6](#_Toc158148407)

[Objetivo general 6](#_Toc158148408)

[Objetivos específicos 6](#_Toc158148409)

[Desarrollo 7](#_Toc158148410)

[Arreglos 7](#_Toc158148411)

[Ejemplo 1 Arreglos: 8](#_Toc158148412)

[Tamaño del array: .length 9](#_Toc158148413)

[Matrices o Arrays de varios subindices 9](#_Toc158148414)

[Incialización de Arrays 11](#_Toc158148415)

[Ejemplo 2 Arreglos 11](#_Toc158148416)

[Pilas 12](#_Toc158148417)

[Ejemplo 1 pila: 14](#_Toc158148418)

[Colas 19](#_Toc158148419)

[Listas 28](#_Toc158148420)

[- Arboles 31](#_Toc158148421)

[- Característica de los Arboles 31](#_Toc158148422)

[- Terminología de los Arboles 31](#_Toc158148423)

[Tipos de Arboles 32](#_Toc158148424)

[- Árbol binario 32](#_Toc158148425)

[- Árbol de búsqueda binaria 32](#_Toc158148426)

[- Árbol binario completo 33](#_Toc158148427)

[Recorrido de los arboles 33](#_Toc158148428)

[- Recorrido in-orden 33](#_Toc158148429)

[- Recorrido pre-orden 34](#_Toc158148430)

[- Recorrido post-orden 34](#_Toc158148431)

[Tipos de árboles binarios 34](#_Toc158148432)

[- Arboles binarios semejantes 34](#_Toc158148433)

[- Arboles binarios equivalentes 35](#_Toc158148434)

[- Arboles binarios completos 35](#_Toc158148435)

[- Arboles binarios degenerados 35](#_Toc158148436)

[- Ejemplo de codificación de Arboles (Copiado codigo) 36](#_Toc158148437)

[- Ejemplo de codificación de Arboles (En Capturas) 40](#_Toc158148438)

[- Ejecución del Código 42](#_Toc158148439)

[- Que es un grafo 44](#_Toc158148440)

[- Característica de un grafo 44](#_Toc158148441)

[- Ventajas y desventajas del uso de grafos 45](#_Toc158148442)

[Tipos de Grafos 45](#_Toc158148443)

[- Grafos dirigidos 45](#_Toc158148444)

[- Grafos no dirigidos 46](#_Toc158148445)

[- Matriz adyacencia 46](#_Toc158148446)

[- Lista adyacencia 47](#_Toc158148447)

[- Ejemplo de codificación de grafo no dirigido (Copiado codigo) 47](#_Toc158148448)

[Conclusiones 50](#_Toc158148449)

[Recomendaciones 50](#_Toc158148450)

[Bibliografía 50](#_Toc158148451)

[**Ilustración 1** Ejemplo de un Árbol 30](#_Toc158148452)

[**Ilustración 2** Ejemplo de árbol binario 31](#_Toc158148453)

[**Ilustración 3** Estructura de un árbol binario 31](#_Toc158148454)

[**Ilustración 4** Dos árboles de búsqueda binaria con mismos elementos 31](#_Toc158148455)

[**Ilustración 5** Ejemplo de árbol binario completo 32](#_Toc158148456)

[**Ilustración 7** Ejemplo de árboles binarios semejantes 33](#_Toc158148457)

[**Ilustración 8** Ejemplo de árbol binario equivalentes 34](#_Toc158148458)

[**Ilustración 9** Ejemplo de árbol binarios completos 34](#_Toc158148459)

[**Ilustración 10** Ejemplo de árbol binarios degenerados 34](#_Toc158148460)

[**Ilustración 3** Ejemplo de grafo no dirigido 45](#_Toc158148461)

[**Ilustración 5** Ejemplo de lista adyacencia de un grafo 46](#_Toc158148462)

[**Ilustración 6** Grafo no dirigido 46](#_Toc158148463)

# Introducción

En la siguiente guía se detalla información sobre diversas estructuras de datos, fundamentales en el desarrollo de sistemas informáticos, especialmente en aplicaciones donde la gestión eficiente de la información es crucial, como en sistemas de la policía. Se abordarán los conceptos de arreglos, pilas, colas, listas, árboles y grafos, junto con ejemplos prácticos de su aplicación en proyectos reales. Comenzando con los arreglos, estos son una estructura de datos que nos permite almacenar una secuencia de elementos del mismo tipo, facilitando el acceso y la manipulación de estos datos de manera eficiente. En el contexto de un sistema de la policía, los arreglos podrían utilizarse para almacenar información sobre criminales, casos abiertos o registros de incidentes. Las pilas y las colas son estructuras de datos fundamentales que operan sobre un único extremo. En una pila, los elementos se insertan y se extraen siempre desde la cima, siguiendo el principio de LIFO (Last In, First Out), lo que podría ser útil para mantener un registro de las acciones realizadas por los agentes de policía en un sistema de seguimiento de actividades. (Vazquez, 2021).

Por otro lado, una cola sigue el principio FIFO (First In, First Out), donde los elementos se insertan al final de la cola y se extraen desde el frente. Esta estructura sería valiosa en un sistema de gestión de solicitudes de servicio, donde las peticiones de ayuda se atienden en el orden en que fueron recibidas. Las listas son colecciones ordenadas de elementos que permiten la inserción y eliminación de datos de manera dinámica. En el contexto de un sistema policial, las listas podrían emplearse para llevar un registro de contactos de emergencia, testigos o evidencia recopilada durante una investigación. Además, se explorarán estructuras más complejas como los árboles y los grafos, que ofrecen una manera organizada de representar relaciones jerárquicas y conexiones entre diferentes elementos de datos. Estas estructuras podrían ser útiles en un sistema de análisis de redes delictivas o en la planificación de rutas para operativos policiales (Blogspot, 2021).

# Sistemas de objetivos

## Objetivo general

• Diseñar e implementar un sistema informático para la policía que utilice eficientemente diversas estructuras de datos, como arreglos, pilas, colas, listas, árboles y grafos, con el fin de gestionar de manera efectiva la información y mejorar la respuesta ante situaciones críticas.

## Objetivos específicos

* Desarrollar módulos de software que empleen arreglos para el almacenamiento y acceso rápido a datos cruciales, como registros de criminales y casos abiertos.
* Implementar funcionalidades de seguimiento y registro de actividades utilizando pilas y colas, para mantener un historial ordenado de acciones realizadas por los agentes de policía.
* Diseñar sistemas de gestión de información basados en listas dinámicas, para organizar contactos de emergencia, testigos y evidencia recopilada durante investigaciones criminales.

# Desarrollo

## Arreglos

Un array [Java](http://www.manualweb.net/tutorial-java/) es una estructura de datos que nos permite almacenar una ristra de datos de un mismo tipo. El tamaño de los arrays se declara en un primer momento y no puede cambiar en tiempo de ejecución como puede producirse en otros lenguajes. La declaración de un array en Java y su inicialización se realiza de la siguiente manera:

tipo\_dato nombre\_array[]; *//Declarar array* nombre\_array = new tipo\_dato[tamanio]; *//Crear array*

Por ejemplo, podríamos declarar un array de caracteres e inicializarlo de la siguiente manera:

char arrayCaracteres[]; *//Declarar array* arrayCaracteres = new char[10]; *//Crear array*

Los arrays [Java](http://www.manualweb.net/tutorial-java/) se numeran desde el elemento cero, que sería el primer elemento, hasta el tamaño-1 que sería el último elemento. Es decir, si tenemos un array de diez elementos, el primer elemento sería el cero y el último elemento sería el nueve. Para acceder a un elemento especifico utilizaremos los corchetes de la siguiente forma. Entendemos por acceso, tanto el intentar leer el elemento, como asignarle un valor.

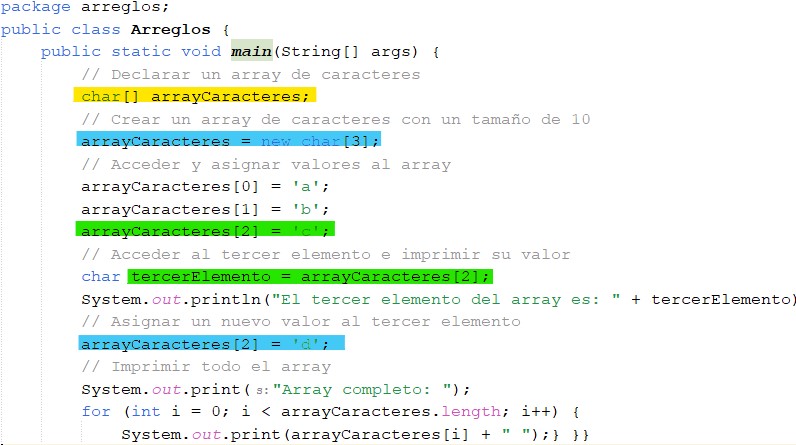
arrayCaracteres[numero\_elemento];

Por ejemplo, para acceder al tercer elemento lo haríamos de la siguiente forma:

char x = arrayCaracteres[2]; *// Lectura de su valor.*

arrayCaracteres[2] = 'b'; *// Asignación de un valor. Como se puede comprobar se pone el número dos, que coincide con el tercer elemento. Ya que como dijimos anteriormente el primer elemento es el cero.*

### Ejemplo 1 Arreglos:

Creamo un array de carácter aplicando lo recién vsito

Crear array de carácter, asignarle datos y mostrar uno es específico modificándolo al mostrar el array.

**Codigo**

**y Ejecucion**

public class Arreglos {

public static void main(String[] args) {

char[] arrayCaracteres;

*/*

*/*

*Declarar un array de caracteres*

arrayCaracteres = new char[3];

*// Crear un array de caracteres con un tamaño de*

*3*

arrayCaracteres[0] = 'a'

*; // Acceder y asignar valores al array*

arrayCaracteres[1] = 'b';

*// Acceder y asignar valores al array*

arrayCaracteres[2] = 'c'

*; // Acceder y asignar valores al array*

char tercerElemento = arrayCaracteres[2];

*// Acceder al tercer elemento*

*,*

*imprimir su valor*

System.out.println("El tercer elemento del array es: " + tercerElemento);

arrayCaracteres[2] = 'd';

*// Asignar un nuevo valor al tercer elemento*

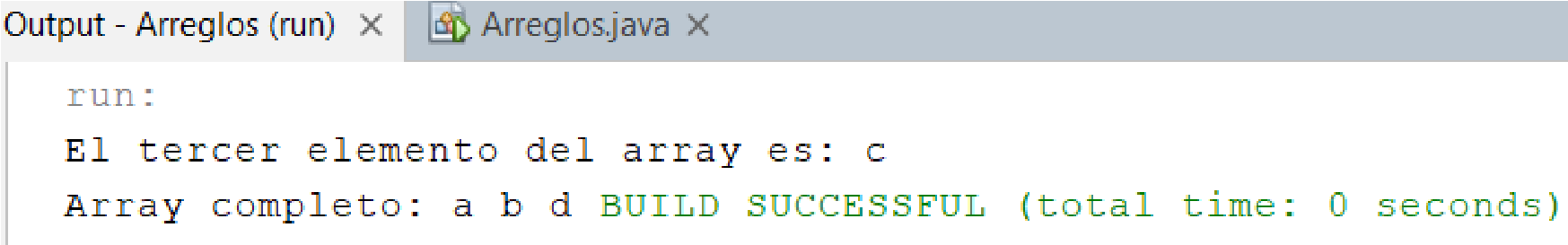
System.out.print("Array completo:

*"); // Imprimir todo el array*

for (int i = 0; i < arrayCaracteres.length; i++) {

System.out.print(arrayCaracteres[i] + " ");} }}

El objeto array, aunque podríamos decir que no existe como tal, posee una variable, la



cual podremos utilizar para facilitar su manejo.

## Tamaño del array: .length

Este atributo nos devuelve el número de elementos que posee el array. Hay que tener en cuenta que es una variable de solo lectura, es por ello que no podremos realizar una asignación a dicha variable. Por ejemplo esto nos serviría a la hora de mostrar el contenido de los elementos de un array:

char array[];

*// Declarar un array de caracteres*

array = new char[10];

*// Crear un array de caracteres con un tamaño de 10*

for (int x=0;x<array.length;x++)

*// Imprimir todo el array*

System.out.printnln(array[x]);

## Matrices o Arrays de varios subindices

Podremos declarar arrays [Java](http://www.manualweb.net/tutorial-java/) de varios subíndices, pudiendo tener arrays [Java](http://www.manualweb.net/tutorial-java/) de dos niveles, que serían similares a las matrices, arrays [Java](http://www.manualweb.net/tutorial-java/) de tres niveles, que serían como cubos y así sucesivamente, si bien a partir del tercer nivel se pierde la perspectiva geométrica. Para declarar e inicializar un array de varios subíndices lo haremos de la siguiente manera:

tipo\_dato nombre\_array[][]; *// Declarar un array subindices*

nombre\_array = new tipo\_dato[tamanio][tamanio]; *// Crear un array subindices*

De esta forma podemos declarar una matriz [Java](http://www.manualweb.net/tutorial-java/) de 2x2 de la siguiente forma:

int matriz[][]; *// Declarar un array subindices*

matriz = new int[2][2]; *// Crear un array subíndices dando valores*

El acceso se realiza de la misma forma que antes:

int x = matriz[1][1]; *// Para leer el contenido de un elemento* matriz[1][1] = x; *// Para asignar un valor*

|  |
| --- |
| .length |

Hay que tener en cuenta que para mostrar su contenido tendremos que utilizar dos bucles. Para saber el número de columnas lo haremos igual que antes mediante la variable , pero para saber el numero de filas que contiene cada columna lo tendremos que realizar de la siguiente manera:

matriz[numero\_elemento].lenght; *// Declarar una matriz*

Nuestra lectura de los elementos de una matriz quedaría de la siguiente forma:

|  |
| --- |
| int matriz[][]; *// Declarar una matriz* matriz = new int[4][4]; *// Crear una matriz*  for (int x=0; x < matrix.length; x++) { *// Imprimir array 1* for (int y=0; y < matriz[x].length; y++) { *// Imprimir array 2*  System.out.println (matriz[x][y]); }} |

## Incialización de Arrays

Existe una forma de inicializar un array en [Java](http://www.manualweb.net/tutorial-java/) con el contenido, amoldándose su tamaño al número de elementos a los que le inicialicemos. Para inicializar un array [Java](http://www.manualweb.net/tutorial-java/) utilizaremos las llaves de la siguiente forma:

tipo\_dato array[] = {elemento1,elemento2,...,elementoN}; *// inicializar un array*

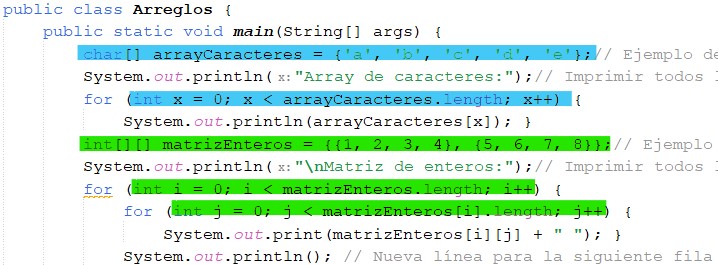
Así, por ejemplo, podríamos inicializar un array [Java](http://www.manualweb.net/tutorial-java/) o una matriz [Java:](http://www.manualweb.net/tutorial-java/)

char array[] = {'a','b','c','d','e'}; *// Tenemos un array de 5 elementos.*

int array[][] = { {1,2,3,4}, {5,6,7,8}}; *// Tenemos un array de 4x4 elementos*

### Ejemplo 2 Arreglos

Este ejemplo aplica lo recién visto, creando un array y una matriz asignando datos.



*Ilustración 2Ejemplo 2Arreglo*

#### Codigo y Ejecucion

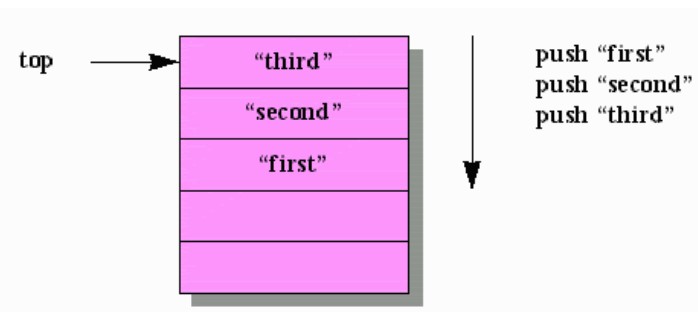
|  |
| --- |
| public class Arreglos {  public static void main(String[] args) { char[] arrayCaracteres = {'a', 'b', 'c', 'd', 'e'}; *// Ejemplo de array de caracteres*  System.out.println("Array de caracteres:"); *// Imprimir todos los elementos del array* for (int x = 0; x < arrayCaracteres.length; x++) { System.out.println(arrayCaracteres[x]); }  int[][] matrizEnteros = {{1, 2, 3, 4}, {5, 6, 7, 8}}; *// Ejemplo de matriz de enteros* System.out.println("\nMatriz de enteros:"); *// Imprimir todos los elementos de la matriz* for (int i = 0; i < matrizEnteros.length; i++) { for (int j = 0; j < matrizEnteros[i].length; j++) {  System.out.print(matrizEnteros[i][j] + " "); }  System.out.println(); *// Nueva línea para la siguiente fila* } }} |
|  |

## Pilas

En programación, una pila es una estructura de datos lineal que sigue el principio de LIFO (Last In, First Out), lo que significa que el último elemento que se añade es el primero en ser retirado. Las operaciones básicas en una pila son la inserción (push) y la eliminación

(pop) de elementos. Además, se puede acceder al elemento superior sin retirarlo (top).

Las pilas son comúnmente utilizadas en la implementación de algoritmos como el recorrido de árboles y la inversión de cadenas. (Sites.upiicsa, 2022).



*Ilustración 20 Ejemplo1*

las pilas se construyen en memoria. Por cada dato insertado, el ítem superior anterior y todos los datos inferiores se mueven hacia abajo. Cuando llega el momento de sacar un ítem de la pila, se recupera y se borra de la pila el ítem superior (que en la figura anterior se revela como "third").

En Java, puedes implementar una pila utilizando un array. Para ello, primero, declaras un array que servirá como el contenedor de la pila. Luego, inicializas la pila estableciendo un límite en el tamaño del array.

char[] pilaArray; *// Declarar un array para la pila*

pilaArray = new char[5]; *// Inicializar la pila con un límite de tamaño, por ejemplo, 5*

Aquí hay un ejemplo específico utilizando una pila de caracteres. Se declara la pila llamada miPila y se inicializa con un límite de tamaño de 5.

char[] pilaArray; *// Declarar un array para la pila de caracteres*

pilaArray = new char[5]; *// Inicializar la pila con un límite de tamaño, por ejemplo, 5*

**Operaciones Básicas en una Pila:**

1. **Push (Agregar un Elemento):** La operación push se utiliza para añadir un nuevo elemento a la pila. En el contexto de una pila basada en un array, esto implica colocar un elemento en la parte superior de la pila. Antes de agregar un elemento, es necesario verificar si la pila está llena para evitar desbordamientos.

public void push(char elemento) { *// Método push para agregar un elemento a la pila* if (top < pilaArray.length - 1) { *// Verifica si el índice top es menor que el tamaño de la pila*  pilaArray[++top] = elemento; *// Incrementa el índice top asignando el elemento*

} else { *//Osino imprime un mensaje si la pila está llena,*

System.out.println("La pila está llena. No se puede agregar más elementos.");

}

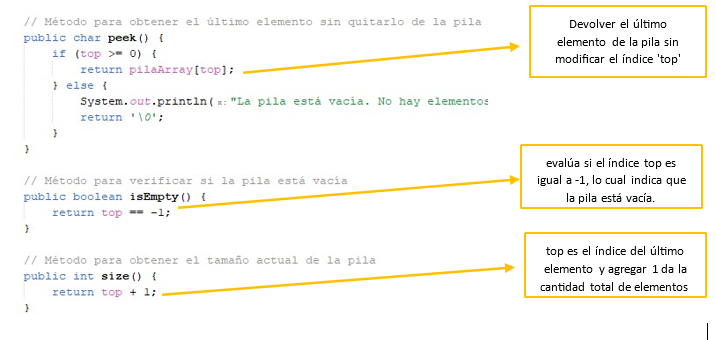
}

1. **Pop (Quitar y Obtener el Último Elemento):** La operación pop se encarga de retirar y devolver el último elemento añadido a la pila. Antes de intentar retirar un elemento, se verifica si la pila está vacía para evitar errores.

|  |
| --- |
| public char pop() *{// Método pop para quitar y obtener el último elemento de la pila*  if (top >= 0) {*//verifica si el índice top es mayor o igual a cero, indicando que hay elementos*  return pilaArray[top--];*// Devolver el último elemento y luego decrementar el índice 'top'*  } else { *// Osino Imprime un mensaje si la pila está vacía*  System.out.println("La pila está vacía. No se puede extraer más elementos."); return '\0'; *// Carácter nulo indicando que la pila está vacía*  }  } |

1. **Peek (Observar el Último Elemento sin Quitarlo):** La operación peek permite observar el elemento superior de la pila sin retirarlo. Esto es útil para conocer el estado actual de la pila sin modificarla.

|  |  |  |
| --- | --- | --- |
| public char peek() { *// Método peek para obtener el último elemento sin quitarlo de la pila*  if (top >= 0) {/*/ verifica si el índice top es mayor o igual a cero, indicando si hay elementos* return pilaArray[top];/*/ Devuelve el último elemento de la pila sin modificar el índice top*.  } else {*// Osino Imprime un mensaje si la pila está vacía*  System.out.println("La pila está vacía. No hay elementos para ver."); return '\0'; *// Carácter nulo indicando que la pila está vacía*  }  } | | |
| Ejemplo 1 pila: |  |  |

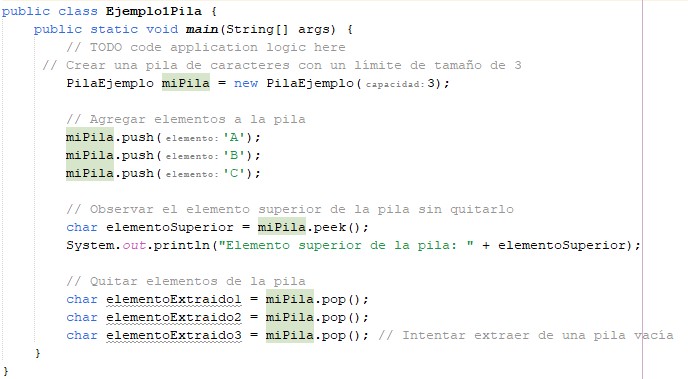


*Ilustración 1 Ejemplo 1 Pila*

#### Main

Aquí en el main utiliza la clase PilaEjemplo para simular el comportamiento de una pila. Crea una pila con un límite de tamaño de 3, agrega tres elementos (A, B, C), observa el elemento superior sin quitarlo y luego intenta extraer elementos de la pila (desapilando).

*Ilustración 2 Ejemplo 1 Pila*



#### Código y Ejecución

|  |
| --- |
| package ejemplo1pila; public class PilaEjemplo { public char[] pilaArray; public int top; *// Índice que sigue al último elemento añadido*  *// Constructor*  public PilaEjemplo(int capacidad) { pilaArray = new char[capacidad];  top = -1; *// Inicializar la pila como vacía*  }  *// Método para agregar un elemento a la pila* public void push(char elemento) { if (top < pilaArray.length - 1) { pilaArray[++top] = elemento;  System.out.println("Elemento '" + elemento + "' agregado a la pila.");  } else {  System.out.println("La pila está llena. No se puede agregar más elementos.");  }  }  *// Método para quitar y obtener el último elemento de la pila*  public char pop() { if (top >= 0) {  char elemento = pilaArray[top--];  System.out.println("Elemento '" + elemento + "' retirado de la pila."); return elemento;  } else {  System.out.println("La pila está vacía. No se puede extraer más elementos."); return '\0';  }  }  *// Método para obtener el último elemento sin quitarlo de la pila*  public char peek() { if (top >= 0) {  return pilaArray[top];  } else {  System.out.println("La pila está vacía. No hay elementos para ver."); return '\0';  }  }  *// Método para verificar si la pila está vacía*  public boolean isEmpty() { return top == -1;  }  *// Método para obtener el tamaño actual de la pila*  public int size() { return top + 1;  }  } |
| package ejemplo1pila; public class Ejemplo1Pila { |
| public static void main(String[] args) {  *// TODO code application logic here*  *// Crear una pila de caracteres con un límite de tamaño de 3*  PilaEjemplo miPila = new PilaEjemplo(3);  *// Agregar elementos a la pila*  miPila.push('A'); miPila.push('B'); miPila.push('C');  *// Observar el elemento superior de la pila sin quitarlo*  char elementoSuperior = miPila.peek();  System.out.println("Elemento superior de la pila: " + elementoSuperior);  *// Quitar elementos de la pila*  char elementoExtraido1 = miPila.pop(); char elementoExtraido2 = miPila.pop(); char elementoExtraido3 = miPila.pop(); *// Intentar extraer de una pila vacía*  }  } |
| **Ejecucion** |

Detalle:

Este programa implementa una pila en Java para simular un sistema de gestión de elementos. Puedes agregar nuevos elementos, observar el elemento superior, retirar elementos y verificar el estado de la pila. Ten en cuenta que este es un ejemplo educativo y que, en aplicaciones más complejas, podrías optar por estructuras de datos más avanzadas según los requisitos específicos.

Detalle:

Este programa implementa una pila en Java para simular un sistema de gestión de elementos. Puedes agregar nuevos elementos, observar el elemento superior, retirar elementos y verificar el estado de la pila. Ten en

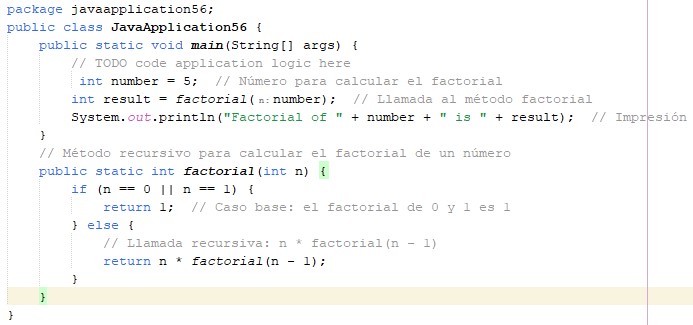
cuenta que este es un ejemplo educativo y que, en aplicaciones más complejas, podrías optar por estructuras de datos más avanzadas según los requisitos específicos.

**Las pilas son muy útiles en varios escenarios de programación. Dos de los más comunes son:**

**Pilas que contienen direcciones de retorno:**

Cuando el código llama a un método, la dirección de la primera instrucción que sigue a la llamada se inserta en la parte superior de la pila de llamadas de métodos del thread actual. Cuando el método llamado ejecuta la instrucción return, se saca la dirección de la parte superior de la pila y la ejecución continúa en esa dirección. Si un método llama a otro método, el comportamiento LIFO de la pila asegura que la instrucción return del segundo método transfiere la ejecución al primer método, y la del primer método transfiere la ejecución al código que sigue al código que llamó al primer método. Como resultado una pila "recuerda" las direcciones de retorno de los métodos llamados.

*Ilustración 24 Ejemplo de Pila con retorno*

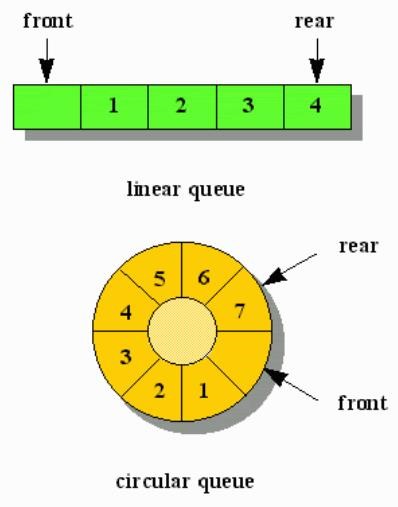


**Pilas que contienen todos los parámetros del método llamado y las variables locales:** Cuando se llama a un método, la JVM reserva memoria cerca de la dirección de retorno y almacena todos los parámetros del método llamado y las variables locales de ese método. Si el método es un método de ejemplar, uno de los parámetros que almacena en la pila es la referencia this del objeto actual. (Sites.upiicsa, 2022).

## Colas

La Cola es una estructura de datos donde la inserción de ítem se hace en un final (el fin de la cola) y la recuperación/borrado de elementos se hace en el otro final (el inicio de la cola). Como el primer elemento insertado es el primero en ser recuperado, los desarrolladores se refieren a estas colas como estructuras FIFO (*first-in, first-out*). (Blogspot, 2021).

los desarrolladores trabajan con dos tipos de colas: lineal y circular. En ambas colas, la inserción de datos se realiza en el fin de la cola, se mueven hacia adelante y se recuperan/borran del inicio de la cola. La siguiente figura ilustra las colas lineal y circular:



*Ilustración 25 Ejemplo1*

La cola lineal de la figura anterior almacena cuatro enteros, con el entero 1 en primer lugar. Esa cola está llena y no puede almacenar más datos adicionales porque rear identifica la parte final de la cola. La razón de la posición vacía, que identifica front, implica el comportamiento lineal de la cola. Inicialmente, front y rear identifican la posición más a la izquierda, lo que indica que la cola está vacía. Para almacenar el entero 1, rear avanza una posición hacia la derecha y almacena 1 en esa posición. Para recuperar/borrar el entero 1, front avanza una posición hacia la derecha. (Blogspot, 2021).

**Declaración e Inicialización de la Cola:**

En Java, se declara un array que actuará como el contenedor de la cola. A continuación, se muestra un ejemplo de cómo declarar y luego inicializar una cola con un límite de tamaño predefinido:

|  |
| --- |
| javaCopy code  *// Declarar un array para la cola* private char[] colaArray;  *// Inicializar la cola con un límite de tamaño, por ejemplo, 5* colaArray = new char[5]; |

**Operaciones Básicas en una Cola:**

1. **Enqueue (Agregar un Elemento):**

La operación enqueue se utiliza para añadir un nuevo elemento al final de la cola. Antes de agregar un elemento, es necesario verificar si la cola está llena para evitar desbordamientos. El siguiente código muestra la implementación de esta operación:

|  |
| --- |
| *// Operación enqueue para agregar un elemento al final de la cola* public void enqueue(char elemento) {  if (isEmpty()) { *// Verificar si la cola está vacía*  frente = 0; *// Establecer el frente en 0 si la cola está vacía*  }  *// Incrementar la posición de la parte trasera y asignar el elemento en esa posición* colaArray[++trasera] = elemento;  *// Incrementar el tamaño actual de la cola*  tamañoActual++;  *// Imprimir un mensaje indicando que el elemento se agregó correctamente*  System.out.println("Elemento '" + elemento + "' agregado a la cola.");  } |

1. **Dequeue (Quitar y Obtener el Último Elemento):**

La operación dequeue se encarga de retirar y devolver el último elemento añadido a la cola. Se verifica si la cola está vacía antes de intentar retirar un elemento:

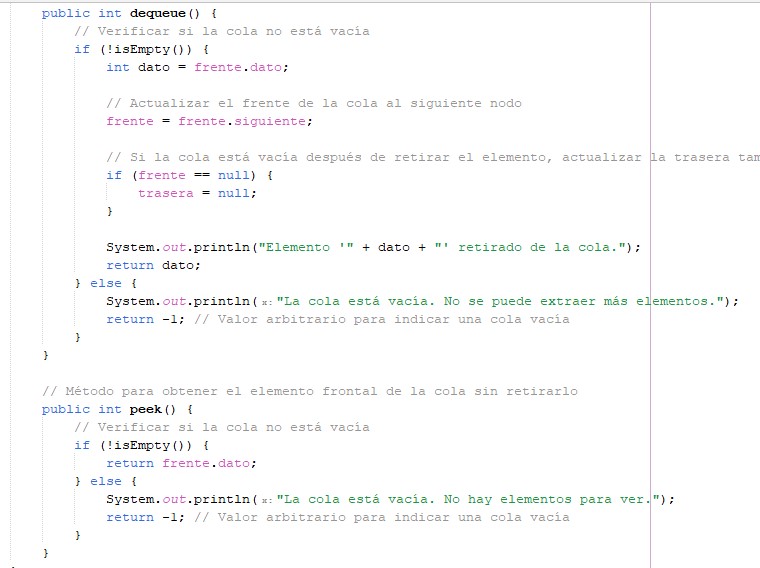
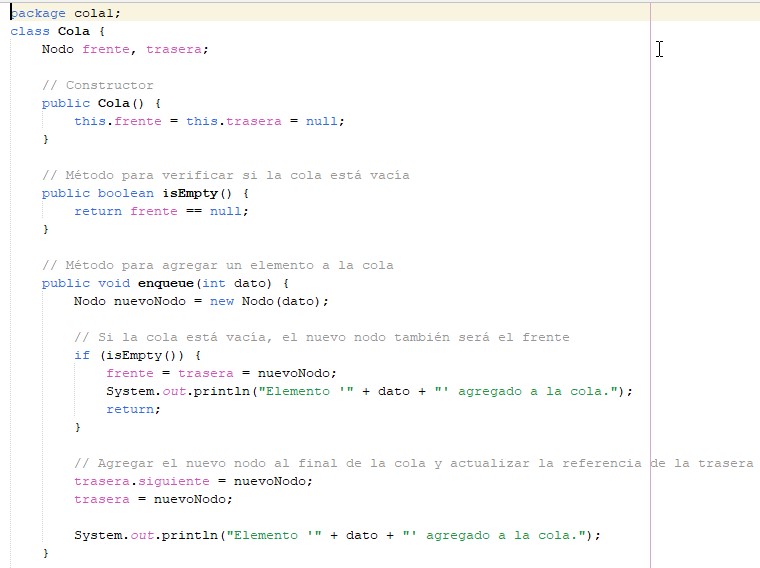
|  |
| --- |
| *// Operación dequeue para retirar y obtener el elemento frontal de la cola* public char dequeue() {  if (!isEmpty()) { *// Verificar si la cola no está vacía*  *// Obtener el elemento frontal de la cola y luego incrementar el índice de frente* char elemento = colaArray[frente++];  *// Decrementar el tamaño actual de la cola*  tamañoActual--;  *// Verificar si la cola está vacía después de retirar el elemento*  if (isEmpty()) {  *// Restaurar los índices de frente y trasera a sus valores iniciales si la cola está vacía*  frente = trasera = -1;  } |
| *// Imprimir un mensaje indicando que el elemento se retiró correctamente*  System.out.println("Elemento '" + elemento + "' retirado de la cola."); return elemento; *// Devolver el elemento retirado*  } else {  *// Imprimir un mensaje si la cola está vacía y no se pueden extraer más elementos*  System.out.println("La cola está vacía. No se puede extraer más elementos.");  *// Devolver '\0' como carácter nulo indicando que la cola está vacía*  return '\0';  }  } |

1. **Peek (Observar el Último Elemento sin Quitarlo):**

La operación peek permite observar el elemento frontal de la cola sin retirarlo. Esto es útil para conocer el estado actual de la cola sin modificarla:

|  |
| --- |
| *// Operación peek para obtener el elemento frontal de la cola sin retirarlo* public char peek() {  *// Verificar si la cola no está vacía*  if (!isEmpty()) {  *// Devolver el elemento frontal de la cola sin modificar los índices*  return colaArray[frente];  } else {  *// Imprimir un mensaje si la cola está vacía y no hay elementos para ver*  System.out.println("La cola está vacía. No hay elementos para ver.");  *// Devolver '\0' como carácter nulo indicando que la cola está vacía*  return '\0';  }  } |

• **Ejemplo 1 Cola:** Clase Cola



Declara

los atributos de

la cola

Constructor de la clase

cola

Método para ver si

la cola esta vacía

Método para agregar

un elemento a la cola

Método para retirar y

obtener el elemento frontal

de la

cola

Método para obtener el

elemento frontal de la cola

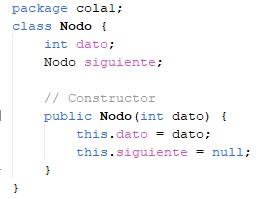
sin retirarlo

*Ilustración 26 Ejemplo 1 Cola*

|  |
| --- |
| Constructor de la clase Nodo |

#### Clase Nodo

*Ilustración 27 Ejemplo 1 Cola*

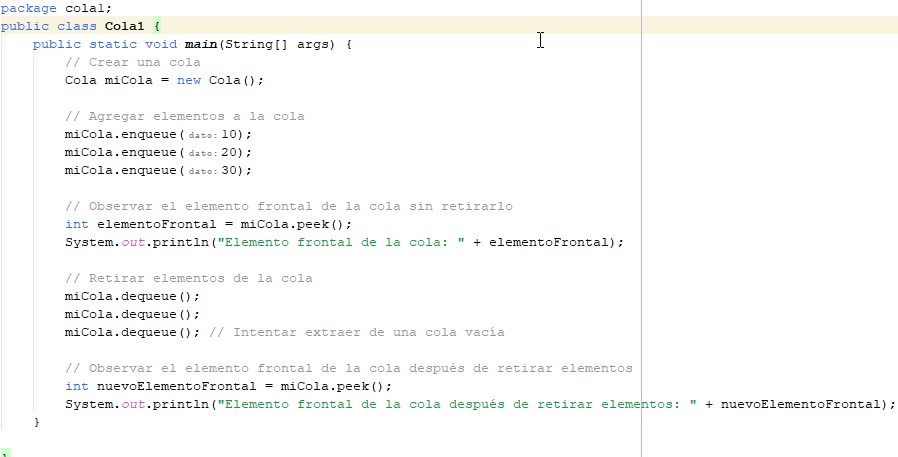


Inicializa los atributos de la

clase nodo

#### Main

En el método main de este código Java, se utiliza la clase Cola1 para simular el comportamiento de una cola. Se crea una instancia de la cola, se agregan tres elementos, se observa el elemento frontal sin retirarlo y se intenta extraer elementos de la cola. Este fragmento ilustra de manera concisa las operaciones fundamentales de una cola en el contexto del programa principal.



*Ilustración 28 Ejemplo 1 Cola*

#### Código y Ejecución

|  |
| --- |
| **CLASE COLA** |
| package cola1; class Cola {  Nodo frente, trasera;  *// Constructor*  public Cola() {  this.frente = this.trasera = null;  }  *// Método para verificar si la cola está vacía*  public boolean isEmpty() { return frente == null;  }  *// Método para agregar un elemento a la cola*  public void enqueue(int dato) {  Nodo nuevoNodo = new Nodo(dato);  *// Si la cola está vacía, el nuevo nodo también será el frente*  if (isEmpty()) {  frente = trasera = nuevoNodo;  System.out.println("Elemento '" + dato + "' agregado a la cola."); return;  }  *// Agregar el nuevo nodo al final de la cola y actualizar la referencia de la trasera* trasera.siguiente = nuevoNodo; trasera = nuevoNodo;  System.out.println("Elemento '" + dato + "' agregado a la cola.");  }  *// Método para retirar y obtener el elemento frontal de la cola*  public int dequeue() {  // Verificar si la cola no está vacía if (!isEmpty()) { int dato = frente.dato;  *// Actualizar el frente de la cola al siguiente nodo* |

|  |
| --- |
| frente = frente.siguiente;  *// Si la cola está vacía después de retirar el elemento, actualizar la trasera también*  if (frente == null) { trasera = null;  }  System.out.println("Elemento '" + dato + "' retirado de la cola."); return dato;  } else {  System.out.println("La cola está vacía. No se puede extraer más elementos."); return -1; *// Valor arbitrario para indicar una cola vacía*  }  }  *// Método para obtener el elemento frontal de la cola sin retirarlo*  public int peek() {  *// Verificar si la cola no está vacía*  if (!isEmpty()) { return frente.dato;  } else {  System.out.println("La cola está vacía. No hay elementos para ver."); return -1; /*/ Valor arbitrario para indicar una cola vacía*  } } } |
| **CLASE NODO** |
| package cola1; class Nodo {  int dato;  Nodo siguiente;  *// Constructor*  public Nodo(int dato) { this.dato = dato; this.siguiente = null; } } |
| **MAIN** |
| package cola1; public class Cola1 { public static void main(String[] args) {  Cola miCola = new Cola();*// Crear una cola*  *// Agregar elementos a la cola*  miCola.enqueue(10); miCola.enqueue(20); miCola.enqueue(30);  *// Observar el elemento frontal de la cola sin retirarlo*  int elementoFrontal = miCola.peek();  System.out.println("Elemento frontal de la cola: " + elementoFrontal);  *// Retirar elementos de la cola*  miCola.dequeue(); miCola.dequeue(); miCola.dequeue(); *// Intentar extraer de una cola vacía*  *// Observar el elemento frontal de la cola después de retirar elementos*  int nuevoElementoFrontal = miCola.peek();  System.out.println("Elemento frontal de la cola después de retirar elementos: " + nuevoElementoFrontal);  } } |
| **EJECUCIÓN** |
|  |

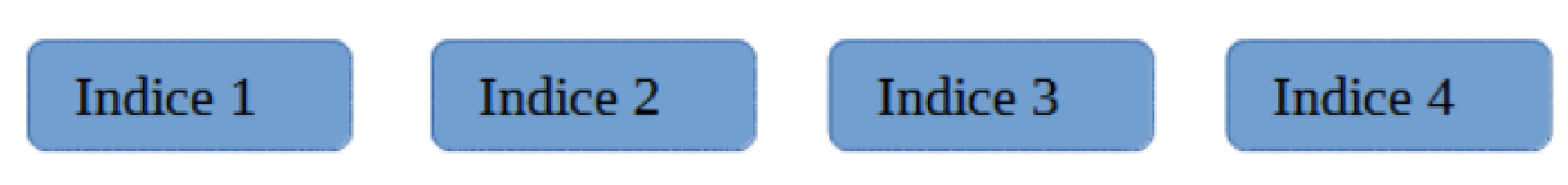
Detalle:

Este programa en Java implementa una cola para gestionar elementos. Permite agregar, observar y retirar elementos en una secuencia ordenada. A través de las clases Nodo y Cola, se establece la estructura básica.

El código incluye mensajes informativos y destaca su naturaleza educativa. En contextos más complejos, se sugiere considerar estructuras de datos más avanzadas según los requisitos de la aplicación.

## Listas

Las listas son un tipo de colección que hereda de la interface collection, son una estructura de datos que respeta el orden en el cual fueron agregados los elementos, también permiten registros repetidos.



Esta interface o los objetos de esta, representan una colección ordenada de elementos, en la cual se tiene un control absoluto y preciso del lugar en el que se quiere insertar. Por medio de la Lista nosotros tenemos la posibilidad de decir en que posición (al inicio, al final o cualquier otro lugar) puede ser insertado o eliminado un elemento.

También es posible acceder a sus elementos a través de su índice; el cual representa la posición del elemento en la lista y así se podría buscar un elemento en dicha lista. La interface List se encuentra en el paquete java.util, algunas de las clases que implementa esta interface son ArrayList, LinkedList, Vector y Stack. Puedes declarar las distintas instancias de la lista de la siguiente manera:

|  |
| --- |
| List linkedList = new LinkedList(); *// LinkedList*  List arrayList = new ArrayList(); *// ArrayList*  List vecList = new Vector(); *// Vector* List stackList = new Stack(); *//Stack* |

Métodos más importantes utilizados en relación con las listas Java:

1. int size(): *determina la cantidad de elementos en una lista.*
2. void add(int index, E element): *agrega un elemento en una posición específica.*
3. boolean isEmpty(): *comprueba si la lista está vacía.*
4. void clear(): *elimina todos los elementos de la lista.*
5. boolean contains(Object o): *devuelve el valor true si el objeto o está en la lista*.
6. boolean add(E e): *agrega el elemento especificado al final de la lista.*
7. boolean remove(Object o): *elimina la primera aparición de un elemento específico.*
8. E get(int index): *devuelve el elemento en el índice indicado.*
9. E set(int index, E element): *reemplaza o inserta un elemento en el índice especificado.*
10. Object[] toArray(): *devuelve un array con los elementos de la lista.*
11. List<E>subList(int fromIndex, int toIndex): *elementos dentro del intervalo definido.*
12. default void replaceAll(UnaryOperator<E> operator): *aplica operadores de Java a cada elemento y sustituye el resultado por dicho elemento.*

Crear un ArrayList

List<String> lista = new ArrayList<>(); *// Declarar una arraylist*

Creación mediante la clase Arrays

String [] nombres = {"Marco","Marisol","Roberto"}; *// Creamos el Array nombres*

List<String>listaNombres=Arrays.asList(nombres) *//Mediante Arrays creamos una nueva lista*

Adición de elementos individuales

List<String> listaAutos = new ArrayList<>(); *// Creamos el ArrayList* listaAutos.add("Mercedes Benz"); *// Agregamos un elemento a la lista*

Adición de elementos de otra colección

List <String> elementos = new ArrayList<>(listaAutos); *// Creamos una lista a la cual le pasamos una lista de autos // A la lista elementos ahora le agregamos la listaNombres* elementos.addAll(listaNombres); *// De esta forma estamos combinando dos listas*

Recorrer una lista

for (String str : elementos) { *// Acontiniacion recorremos la lista elementos con un foreach* System.out.println(str); }

Recorrer lista con ListIterator

ListIterator es un objeto que se apropia de los elementos de una colección y tiene funcionalidades que permiten recorrer dicha colección o lista. La ventaja de recorrer con iteradores en vez de realizarlo con ciclos ,es que, los iteradores nos permiten realizar acciones cuando estemos recorriendo la lista.

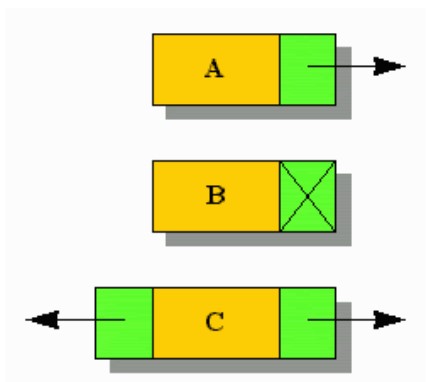
ListIterator<String> iterador = listaNombres.listIterator(); *// Creamos el iterador* while (iterador.hasNext()){ *// Mediante el ciclo while lo recorremos* System.out.println(iterador.next());}

Eliminar elementos de una lista

listaNombres.remove(0); *// El parametro 0 representa el indice que queremos eliminar* listaNombres.remove("Roberto") *// Eliminar elemento mediante su nombre* listaNombres.clear(); *// Para eliminar todos los elmentos de la lista usamos clear*

Estos son los usos más comunes que le podemos dar a las listas.

**Lista enlazada** *es una secuencia de nodos que se interconectan mediante sus campos de enlace.* En ciencia de la computación se utiliza una notación especial para ilustrar las listas enlazadas. En la siguiente imagen aparece una variante de esta notación que utilizaré a lo largo de esta sección:



*Ilustración 21Ejemplo3*

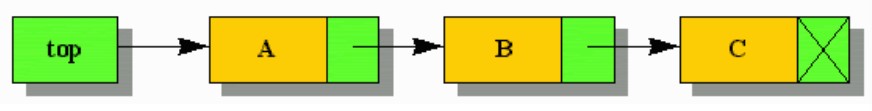
La figura anterior presenta tres nodos: A, B y C. Cada nodo se divide en áreas de contenido (en naranja) y una o más áreas de enlace (en verde). Las áreas de contenido representan todos los campos que no son enlaces, y cada área de enlace representa un campo de enlace. Las áreas de enlace de A y C tienen unas flechas para indicar que referencian a otro nodo del mismo tipo (o subtipo). La única área de enlace de B incorpora una X para indicar una referencia nula. En otras palabras, B no está conectado a ningún otro nodo. (Vazquez, 2021).

Aunque se pueden crear muchos tipos de listas enlazadas, las tres variantes más populares son la lista de enlace simple, la lista doblemente enlazada y la lista enlazada circular.

Exploremos esas variantes, empezando con la lista enlazada.

Lista de Enlace Simple

Una **lista de enlace simple** es una lista enlazada de nodos, donde cada nodo tiene un único campo de enlace. Una variable de referencia contiene una referencia al primer nodo, cada nodo (excepto el último) enlaza con el nodo siguiente, y el enlace del último nodo contiene null para indicar el final de la lista. Aunque normalmente a la variable de referencia se la suele llamar top, usted puede elegir el nombre que quiera. La siguiente figura presenta una lista de enlace simple de tres nodos, donde top referencia al nodo A, A conecta con B y B conecta con C y C es el nodo final:



*Ilustración 22Ejemplo4*

Un algoritmo común de las listas de enlace simple es la inserción de nodos. Este algoritmo está implicado de alguna forma porque tiene mucho que ver con cuatro casos: cuando el nodo se debe insertar antes del primer nodo; cuando el nodo se debe insertar después del último nodo; cuando el nodo se debe insertar entre dos nodos; y cuando la lista de enlace simple no existe. (Vazquez, 2021).

# Arboles

**Ilustración 1** Ejemplo de un Árbol

Un árbol se puede definir como una estructura jerárquica y en forma no lineal, aplicada sobre una colección de elementos u objetos llamados nodos. (Meneses, 2021)

En si los árboles son considerados las estructuras de datos no lineales y dinámicas de datos muy importantes del área de computación y también son muy utilizados en informática como un método eficiente para búsquedas grandes y complejas. Casi todos los sistemas operativos almacenan sus archivos en árboles o estructuras similares a árboles

Los árboles se consideran o se las denomina estructuras dinámicas ya que las mismas pueden cambiar tanto de forma como de tamaño durante la ejecución del programa y también son estructuras no lineales porque cada elemento del árbol puede tener más de un sucesor

# Característica de los Arboles

|  |  |  |  |
| --- | --- | --- | --- |
| Son una estructura idealmente no lineal | Se compone de un nodo raíz y n sub arboles | Se puede definir de forma recursiva | Tienen una jerarquía |

# Terminología de los Arboles

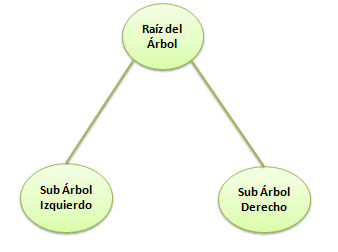
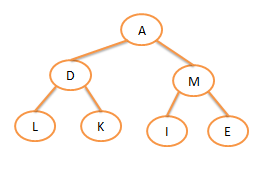
|  |  |
| --- | --- |
| **Raíz** | Es el nodo superior de un árbol |
| **Hijo** | El nodo siguiente de la raíz |
| **Nodo interno** | Nodo que tiene al menos un hijo |
| **Nodo externo** | Nodo que no tiene hijos |
| **Brazo o rama** | Conexión de un nodo a otro |
| **Orden de un árbol** | Número máximo de hijos que un nodo puede tener |
| **Grado de un nodo** | Número de hijos que tiene un determinado nodo |
| **Nivel** | Orden de la jerarquía |
| **Altura** | Distancia de la raíz a la hoja de nivel más bajo |
| **Peso** | Número total de nodos de un árbol |
| **Bosque** | Es un conjunto de árboles distintos |

# Tipos de Arboles

# Árbol binario

Los árboles binarios son estructuras de datos muy similares a las listas doblemente enlazadas, en el sentido que tienen dos punteros que apuntan a otros elementos, pero no tienen una estructura lógica de tipo lineal o secuencial como aquellas, sino ramificada.  Tienen aspecto de árbol, de ahí su nombre (wordpress, 2020)

En si se puede decir que es una estructura de datos utilizada en la ciencia de la computación. Esta estructura inicia con una raíz que luego se extiende en dos ramificaciones hasta que finalmente terminan en una hoja.



**Ilustración 2** Ejemplo de árbol binario

**Ilustración 3** Estructura de un árbol binario

# Árbol de búsqueda binaria

**Ilustración 4** Dos árboles de búsqueda binaria con mismos elementos

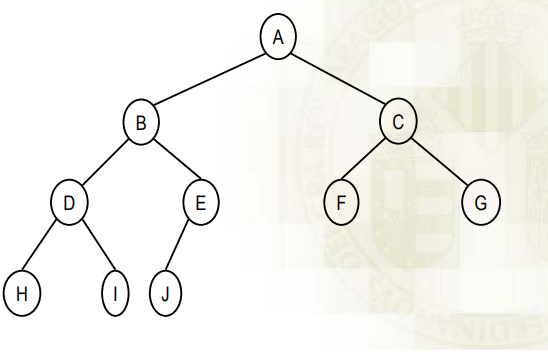
Un árbol binario de búsqueda (ABB) es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo x son menores que el elemento almacenado en x, y todos los elementos almacenados en el subárbol derecho de x son mayores que el elemento almacenado en x. (Lopez, 2019)

En si un árbol binario de búsqueda son una estructura de datos fundamental en informática y ciencias de la computación. Su característica distintiva es la propiedad de orden que dicta la relación entre los elementos almacenados en el árbol

# Árbol binario completo

Se dice que un árbol binario de altura k está completo si está lleno hasta altura k-1 y el último nivel está ocupado de izquierda a derecha (pascual, 2005)

Es decir un árbol binario completo es una estructura eficiente y organizada que se llena de manera ordenada hasta la altura K-1 y completa su ocupación en el último nivel de izquierda a derecha. Esta definición proporciona un marco claro para el diseño y análisis de árboles binarios en diversos contextos informáticos.



**Ilustración 5** Ejemplo de árbol binario completo

# Recorrido de los arboles

# Recorrido in-orden

Este recorrido se realiza así: primero recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho. En síntesis:

hijo izquierdo — raíz — hijo derecho

# Recorrido pre-orden

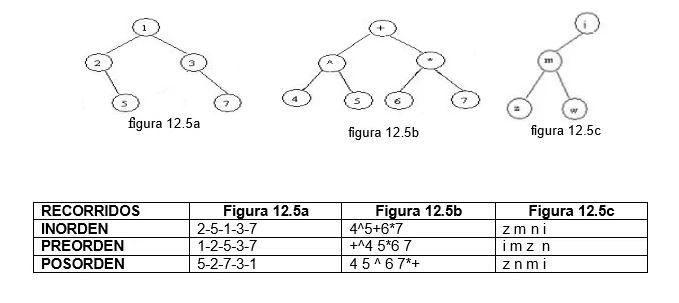
Este recorrido se realiza así: primero visita la raíz; segundo recorre el subárbol izquierdo y por último va a subárbol derecho. En síntesis:

raíz — hijo izquierdo — hijo derecho

# Recorrido post-orden

Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y por último, visita la raíz. En síntesis:

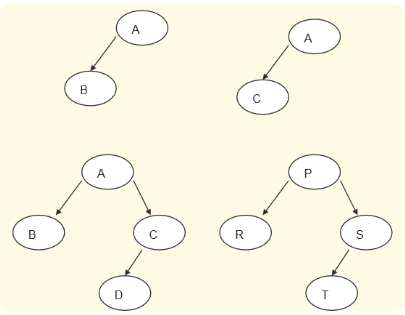
hijo izquierdo– hijo derecho — raíz



# Tipos de árboles binarios

# Arboles binarios semejantes

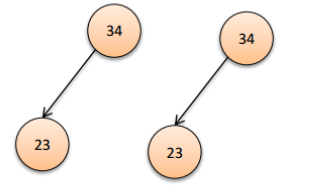
Dos árboles binarios son semejantes cuando sus estructuras son idénticas, pero la información que contienen sus nodos es diferente.



**Ilustración 7** Ejemplo de árboles binarios semejantes

# Arboles binarios equivalentes

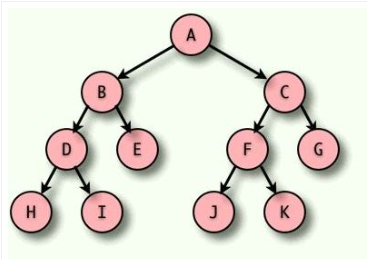
Se definen como aquellos que son similares y además los nodos contienen la misma información



**Ilustración 8** Ejemplo de árbol binario equivalentes

# Arboles binarios completos

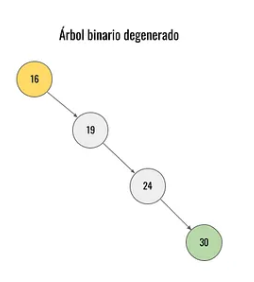
Son aquellos árboles en los que todos sus nodos excepto los del último nivel, tiene dos hijos; el subárbol izquierdo y el subárbol derecho



**Ilustración 9** Ejemplo de árbol binarios completos

# Arboles binarios degenerados

Cuando un árbol contiene 1 sólo hijo por nodo. Los árboles degenerados tienen la profundidad máxima posible dado un número de elementos. Este tipo de árboles se comportan como listas.



**Ilustración 10** Ejemplo de árbol binarios degenerados

# Ejemplo de codificación de Arboles (Copiado codigo)

En la clase principal hay un menú, el cual incluye las siguientes opciones:

* Recorrido Inorden
* Recorrido Preorden
* Recorrido Postorden
* Salir

|  |
| --- |
| **Clase Principal** |
| package tarea3u2;  import java.util.Scanner;  public class Tarea3U2 {  public static void main(String[] args) {  ArbolBinario arbol = new ArbolBinario();  Scanner scanner = new Scanner(System.in);    arbol.insertar(50);  arbol.insertar(30);  arbol.insertar(70);  arbol.insertar(20);  arbol.insertar(40);  arbol.insertar(60);  arbol.insertar(80);  int opcion;  do {  System.out.println("========================");  System.out.println("DATOS DEL ARBOL");  System.out.println("\n 50-30-70-20-40-60-80");  System.out.println("\nRECORRIDOS DE ARBOLES");  System.out.println("1. Recorrido Inorden");  System.out.println("2. Recorrido Preorden");  System.out.println("3. Recorrido Postorden");  System.out.println("4. Salir");  System.out.print("Seleccione el recorrido: ");  opcion = scanner.nextInt();  switch (opcion) {  case 1:  System.out.println("Recorrido Inorden:");  arbol.inorden();  System.out.println("");  break;  case 2:  System.out.println("Recorrido Preorden:");  arbol.preorden();  System.out.println("");  break;  case 3:  System.out.println("Recorrido Postorden:");  arbol.postorden();  System.out.println("");  break;  case 4:  System.out.println("Salio con exito.");  break;  default:  System.out.println("Opcion incprrecta. Intente de nuevo.");  }  } while (opcion != 4);  }} |

La clase nodo sirve como una estructura básica para construir un árbol binario. Cada objeto nodo tiene un valor y enlaces a los nodos hijos izquierdo y derecho. La inicialización de los enlaces a null indica que inicialmente no hay nodos hijos. Esta clase es utilizada en conjunto con la clase ArbolBinario para construir y manipular un árbol binario.

|  |
| --- |
| **Clase nodo** |
| package tarea3u2;  public class nodo {  int valor;  nodo izquierdo, derecho;  public nodo(int valor) {  this.valor = valor;  this.izquierdo = this.derecho = null;  }  } |

La clase ArbolBinario encapsula la lógica para construir y manipular un árbol binario, incluyendo operaciones de inserción y recorridos en diferentes órdenes:

|  |
| --- |
| **Clase ArbolBinario** |
| package tarea3u2;  public class ArbolBinario {  nodo raiz;  public ArbolBinario() {  raiz = null; }  public void insertar(int valor) {  raiz = insertarRec(raiz, valor); }  private nodo insertarRec(nodo nodo, int valor) {  if (nodo == null) {  nodo = new nodo(valor);  return nodo; }  if (valor < nodo.valor) {  nodo.izquierdo = insertarRec(nodo.izquierdo, valor);  } else if (valor > nodo.valor) {  nodo.derecho = insertarRec(nodo.derecho, valor);  }return nodo; }  public void inorden() {  inordenRec(raiz); }  private void inordenRec(nodo nodo) {  if (nodo != null) {  inordenRec(nodo.izquierdo);  System.out.print(nodo.valor + " ");  inordenRec(nodo.derecho); }  public void preorden() {  preordenRec(raiz); }  private void preordenRec(nodo nodo) {  if (nodo != null) {  System.out.print(nodo.valor + " ");  preordenRec(nodo.izquierdo);  preordenRec(nodo.derecho); }}  public void postorden() {  postordenRec(raiz); }  private void postordenRec(nodo nodo) {  if (nodo != null) {  postordenRec(nodo.izquierdo);  postordenRec(nodo.derecho);  System.out.print(nodo.valor + " ");}}} |

# Ejemplo de codificación de Arboles (En Capturas)

|  |
| --- |
|  |

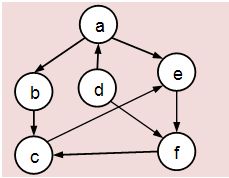
# Ejecución del Código

|  |
| --- |
|  |

# Que es un grafo

Un grafo es una estructura matemática que consta de un conjunto de vértices (nodos) y un conjunto de aristas (conexiones) que conectan pares de vértices. Los grafos son una poderosa herramienta para representar relaciones y conexiones entre objetos en diversas disciplinas, como ciencias de la computación, matemáticas, ingeniería, redes, biología, entre otras.

Los grafos son una estructura de datos no lineal parecida a la de los árboles, pero este sería un árbol sin las leyes de acomodo que rigen a un árbol normal. (Hernández, 2020)



Los grafos están conformados de nodos o también nombrados vértices los cuales son registros con datos y al menos un apuntador a otro nodo, y de aristas, las cuales son las conexiones que existen de un nodo a otro.

# Característica de un grafo

|  |  |
| --- | --- |
| **Vértices y aristas** | Los vértices representan entidades individuales, mientras que las aristas representan relaciones o conexiones entre pares de vértices |
| **Dirección**: | Los grafos pueden ser dirigidos (las aristas tienen una dirección) o no dirigidos (las aristas son bidireccionales). |
| **Peso** | Las aristas pueden tener un peso asociado, que representa una medida o costo entre los vértices conectados. |
| **Ciclos** | Los ciclos son secuencias de aristas que permiten regresar al mismo vértice partiendo y terminando en él mismo. |
| **Conectividad** | Los grafos pueden ser conexos (hay un camino entre cualquier par de vértices) o no conexos. |

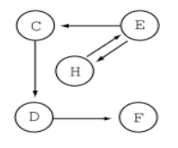
# Ventajas y desventajas del uso de grafos

|  |  |
| --- | --- |
| **Ventajas** | **Desventajas** |
| Representación clara y visual | Consumo de memoria elevado |
| Acceso rápido a conexiones directas | Ineficiente para grafos dispersos o grandes |
| Fácil implementación de algoritmos | Dificultad para agregar o eliminar nodos/aristas |
| Ideal para grafos densos y pequeños | Mayor complejidad en operaciones de modificación |
| Eficiente en verificación de aristas | Espacio cuadrático para grafos completos |

# Tipos de Grafos

# Grafos dirigidos

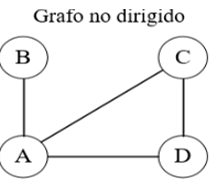
Un grafo dirigido conocido también como dígrafo consta de un conjunto de vértices y aristas donde cada arista se asocia de forma unidireccional a través de una flecha con otro. Las aristas dependiendo de su salida o ingreso reciben la calificación de entrante o saliente, la condición común, es que siempre tienen un destino hacia un nodo.



Un grafo dirigido (o digrafo) G consiste de un conjunto de vértices y un conjunto de arcos E. A los vértices se les llama también nodos o puntos y a los arcos aristas dirigidas o líneas dirigidas. Un arco es un par ordenado de vértices (v, w) donde v es la cola y w la cabeza del arco. Un arco (v, w) se expresa también como v w y se dibuja como (Pedro, 2008)

# Grafos no dirigidos

Los grafos no dirigidos son aquellos que constan un conjunto de vértices que están conectados a un conjunto de aristas de forma no direccional. Esto significa que una arista puede indistintamente recorrerse desde cualquiera de sus puntos y en cualquier dirección

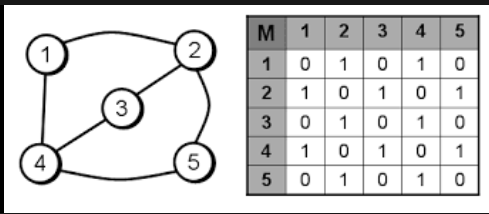


**Ilustración 3** Ejemplo de grafo no dirigido

Un grafo en el cual todas las aristas son no dirigidas se denominará "grafo no dirigido". El grafo no dirigido es aquel que no tiene sentido su arista. Un grafo no dirigido G representa elementos, y una arista (v, w) representa una incompatibilidad entre los elementos v y w. (Cortes., 2020)

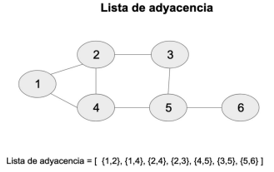
# Matriz adyacencia

La matriz de adyacencia es una matriz de n × n donde n es la cantidad de nodos del grafo, que en la posición (i, j) tiene un 1 (o true) si hay una arista entre los nodos i y j y 0 (o false) si no Esta es una de las representaciones más utilizadas. Si bien el ejemplo es para un grafo no dirigido, también se puede utilizar la misma estructura para grafos dirigidos y grafos con pesos.



# Lista adyacencia

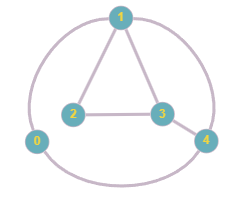
La lista de adyacencia es una representación de todas las aristas de un grafo mediante una lista. Aquí tenemos un ejemplo de un grafo no dirigido y aquí tenemos la lista de adyacencia



**Ilustración 5** Ejemplo de lista adyacencia de un grafo

# Ejemplo de codificación de grafo no dirigido (Copiado codigo)

A continuación, tenemos el código representando el siguiente grafo no dirigido:



**Ilustración 6** Grafo no dirigido

|  |
| --- |
| **Clase GrafoNoDirigido** |
| package tarea4u2;  import java.util.HashMap;  import java.util.LinkedList;  import java.util.List;  import java.util.Map;  public class GrafoNoDirigido {  private int vertices;  private Map<Integer, List<Integer>> listaAdyacencia;  public GrafoNoDirigido(int vertices) {  this.vertices = vertices;  listaAdyacencia = new HashMap<>();  for (int i = 0; i < vertices; i++) {  listaAdyacencia.put(i, new LinkedList<>());  }}  public void agregarArista(int origen, int destino) {  listaAdyacencia.get(origen).add(destino);  listaAdyacencia.get(destino).add(origen); }  public void imprimirGrafo() {  for (int i = 0; i < vertices; i++) {  List<Integer> vecinos = listaAdyacencia.get(i);  System.out.print("Vertice " + i + " se conecta a: ");  for (Integer vecino : vecinos) {  System.out.print(vecino + " "); }  System.out.println();  }}} |

# Conclusiones

* La comprensión y aplicación adecuada de diversas estructuras de datos, como arreglos, pilas, colas, listas, árboles y grafos, son fundamentales para el diseño eficiente de sistemas informáticos en el ámbito policial.
* La utilización de estas estructuras de datos permite una gestión más efectiva de la información, lo que se traduce en una mejora en la capacidad de respuesta ante situaciones críticas y en una toma de decisiones más informada.
* La implementación de un sistema informático para la policía que integre estas estructuras de datos ofrece una solución completa y robusta para la gestión de datos y la optimización de los procesos operativos.

# Recomendaciones

* Antes Capacitar al personal policial en el uso y manejo de sistemas informáticos basados en estas estructuras de datos, con el fin de aprovechar al máximo su potencial y garantizar una correcta operación del sistema.
* Mantener actualizados los sistemas informáticos con las últimas tecnologías y prácticas de desarrollo de software, para asegurar su eficiencia y seguridad en todo momento.
* Realizar periódicamente auditorías y pruebas de seguridad en el sistema informático, con el objetivo de detectar posibles vulnerabilidades y asegurar la integridad y confidencialidad de la información almacenada.

# Bibliografía

Blog.hubspot. (2022). *Herencia y polimorfismo.* Recuperado el 2023, de https://blog.hubspot.es/website/polimorfismo-

java#:~:text=Podr%C3%ADamos%20decir%20que%20la%20herencia,el%20uso%20de% 20diferentes%20funciones.

Blog.hubspot. (2023). *Herencia.* Obtenido de https://blog.hubspot.es/website/que-esherencia-

java#:~:text=%C2%BFQu%C3%A9%20es%20la%20herencia%20en,herencia%20entre% 20clases%20es%20extends.

Blogspot. (2021). *Colas en java.* Obtenido de https://programacionfacilestructuradedatos.blogspot.com/p/colas-en-java.html

Sites.upiicsa. (2022). *Pilas en java.* Obtenido de https://www.sites.upiicsa.ipn.mx/estudiantes/academia\_de\_informatica/estructura\_y\_ rd/docs/u2/Pilas%20en%20Java.pdf

Vazquez, M. (2021). *Listas en java.* Obtenido de https://blog.codmind.com/listas-en-java/