# PORTADA

UNIVERSIDAD TÉCNICA DE AMBATO

Facultad de Ingeniería en Sistemas, Electrónica e Industrial

Título: “Pilas y colas”

Carrera: Software

Nivel y Paralelo: 3RO “A”

Alumnos participantes: Barros Lozada Leonel

Jijón Viscaino Diego Patricio

López Garcés Elkin Joao

Paredes Lozada Ariel Mateo

Asignatura: Estructura de datos

Docente: Ing. Félix Fernandez M.Sc.

# INFORME DEL PROYECTO

## Título

Pilas y colas

## **Objetivos**

**Objetivo General**

Conocer el funcionamiento de pilas y colas.

**Objetivos Específicos**

* Describir las características y diferencias entre pilas y colas en Java, enfocándose en sus principios LIFO y FIFO.
* Implementar una aplicación en Java que inserte y extraiga datos de una pila y una cola, documentando el proceso con código y capturas.
* Analizar el comportamiento de pilas y colas mediante la aplicación, destacando sus aplicaciones prácticas.

## Palabras clave: ()

## Introducción

En el ámbito de la programación, es fundamental entender cómo operan las diferentes estructuras de datos para seleccionar la más adecuada según las necesidades del problema a resolver. Las pilas (stacks) y las colas (queues) son dos de las estructuras de datos más utilizadas, cada una con características únicas que las hacen idóneas para ciertos tipos de operaciones y escenarios. Este informe se centra en explorar en profundidad estas dos estructuras de datos dentro del entorno de programación Java.

El objetivo principal de este estudio es conocer a fondo el funcionamiento de las pilas y las colas, identificando sus propiedades, diferencias y casos de uso específicos. Para lograr esto, se describirán sus características operativas básicas, incluyendo los principios de "último en entrar, primero en salir" (LIFO) para las pilas, y "primero en entrar, primero en salir" (FIFO) para las colas. Además, se implementará una aplicación en Java que permitirá insertar y extraer datos en ambas estructuras, proporcionando una plataforma para observar y comparar su comportamiento en tiempo real.

A través de la implementación práctica y el análisis comparativo, este informe no solo busca enriquecer la comprensión teórica de estas estructuras de datos, sino también demostrar su aplicabilidad y eficiencia en situaciones prácticas de programación.

## Resultados y Discusión (Desarrollo)

## Tipo de datos abstractos y Análisis de algoritmos

Para tener una mejor comprensión acerca de las características, ventajas y desventajas de las pilas y colas, primero es necesario conocer a qué hace referencia cuando se habla de un tipo abstracto de datos (ADT, por sus siglas en inglés), así como un breve repaso acerca del análisis de algoritmos.

Primeramente, un tipo abstracto de datos se refiere a una abstracción en la cual se definen las operaciones que puede tener un objeto, pero no su implementación [1]. Los tipos de dato abstractos, sólo hacen referencia a qué accione van a ser posible, no cómo se las hace. En Java, se puede entender a los tipos de dato abstractos como las interfaces. Un tipo especial de tipos de datos abstractos son aquellos que funcionen como colecciones, que son objetos que recopilan y organizan otros objetos. Al conjunto de estructuras que se usan para implementar colecciones se las llama estructuras de datos [2].

Lo que permiten los tipos de dato abstractos es definir una estructura básica para los tipos de dato concretos, en los cuales sí se definen la implementación de los métodos. Por ejemplo, se define una interfaz “Lista”, que tiene las operaciones de añadir, remover y buscar. Ahora, se puede usar esta Lista de modo que otras listas la implementen. Estas listas concretas pueden usar arrays, estructuras enlazadas o más, pero todas tienen en común las operaciones definidas por la interfaz.

El análisis de algoritmos sirve para ver cuánta memoria usa un algoritmo (complejidad espacial) o cuánto tiempo tarda (complejidad temporal). Para expresar esto, se utiliza una función que muestra la relación entre el tamaño del problema y la complejidad espacial o temporal del problema.

Sin embargo, en vez de expresar esta relación como una función común y corriente, se utiliza la notación O grande (“Big O notation”) [1]. La definición formal de la notación O grande es la siguiente:

Si existen constantes y tales que cuando

Esto, a grandes rasgos, dice que la función , que expresa la complejidad temporal o espacial del algoritmo, crece a una velocidad no mayor que , siendo el tamaño del problema o tamaño de la entrada.

Para el análisis de algoritmos no se suele usar esta definición formal, sino que se estima basado en un conocimiento general de cómo crece la función. En la notación O grande, lo único que se anota es el término dominante del crecimiento de la función de complejidad. De esa manera, se ignoran constantes, términos de menor grado y únicamente se enfoca en el término que crece más rápido que los demás [2].

El término que crece más rápido es .

El término que crece más rápido es

Cuando sólo hay constantes, es

Mediante la notación O grande, es posible comparar la eficiencia de los algoritmos y decir si uno es mejor que otro con un sustento matemático.

Otra ventaja de la notación O grande es que simplifica la manera en que se analizan los algoritmos. Comprar dos algoritmos, únicamente basándose en las funciones de complejidad, implicaría analizar todo el código, obtener la función de complejidad, comparar su crecimiento con la otra función y determinar cuál crece de mejor manera. Con la notación O grande, sólo es necesario observar las partes que más problemas darán, y analizarlas para determinar cómo se comportan [2], no es necesario analizar todo el algoritmo.

Para obtener la complejidad con notación O grande, se suelen seguir las pautas:

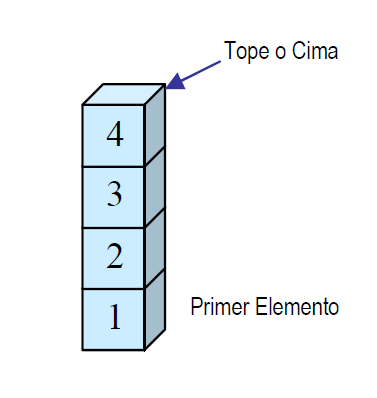
1. Una asignación, cálculo u operación simple tiene una complejidad de
2. Los bucles multiplican la complejidad del cuerpo del bucle por el número de veces que se repite. Así, un bucle con un cuerpo de complejidad que se repita veces, tiene una complejidad de
3. Si se tiene un bucle anidado, se analiza de adentro hacia afuera, multiplicando las complejidades de los bucles.
4. En un condicional, se suma la complejidad de la condición más la complejidad de la rama con mayor complejidad
5. Si se llama a funciones, se suma la complejidad de estas

El análisis de algoritmos permite determinar y comparar la eficiencia de distintos algoritmos. Mediante esta herramienta, es posible definir qué algoritmo se puede usar en qué situación y por qué.

## Pilas (Stacks)

Una pila es una colección de elementos organizada según el principio de "último en entrar, primero en salir" (LIFO, Last In First Out). Esto significa que el último elemento añadido a la pila será el primero en ser eliminado.

Otra forma de entender a una pila es como una estructura de datos en la cual sólo es posible acceder al último elemento, llamado cima. Todas las operaciones que se hacen en una pila están limitadas a hacerlo en la cima, no es posible acceder a los elementos que están por debajo. Una analogía para esto sería, justamente, una pila de libros o una torre de cubos, en donde es posible sacar un elemento desde arriba, desde la cima, pero no desde abajo o del medio porque se cae la pila [2].



La ventaja de esta estructura de datos es que las condiciones impuestas permiten manejar los datos de forma eficiente y rápida. Las operaciones dentro de una pila tienen una complejidad temporal , por lo que el tiempo que se tarda en hacer cualquier cosa es constante. Además, son las estructuras de datos usadas por preferencia si se intenta usar un principio LIFO.

**Operaciones:**

Las dos operaciones primarias en una pila son **push** (agregar un elemento a la pila) y **pop** (eliminar el elemento superior de la pila).

**push():** Agrega un elemento a la pila, siempre por la cima

**pop():** Elimina el elemento superior de la pila. Es común que este elemento también devuelva el elemento eliminado.

**isEmpty():** Verifica si la pila está vacía. Devuelve true si la pila no contiene elementos.

**size():** Devuelve el número de elementos en la pila. Esto es útil para controlar la cantidad de datos en la estructura de datos en cualquier momento.

**peek():** Permite ver el elemento en la cima de la pila sin eliminarlo. Este método es crucial para operaciones que requieren acceso al último elemento ingresado sin modificar la pila.



**Utilidad:** Ideal para algoritmos de navegación, como los utilizados en la función de deshacer (undo) en editores de texto, o para manejar llamadas recursivas en programación.

**Implementación:** En Java, las pilas se pueden implementar usando Stack<E> de la biblioteca Java Utilities (java.util), aunque también es común utilizar otras estructuras de datos, como listas enlazadas, para manejar dinámicamente los datos.

Una debilidad de la clase Stack<E> es que hereda de la clase Vector<E>, la cual tiene un método para iterar a través de los elementos de la colección, un método para insertar en una posición dentro de la pila y otros cuantos que rompen con la idea de qué es y cómo se debe comportar una pila. La clase Stack<E>, para almacenar los datos, utiliza un array.

## Colas (Queue)

Una cola es una colección de elementos organizada según el principio de "primero en entrar, primero en salir" (FIFO, First In First Out). Esto significa que el primer elemento añadido será el primero en ser eliminado.

Se puede entender a una cola como una estructura de datos en al cual sólo se puede acceder por dos lados, la parte de atrás, denominada trasero, por donde ingresan los datos, y una parte inicial, denominada frente, por donde salen los datos. Sólo es posible acceder a estas dos partes de la cola, no se puede acceder a un elemento de en medio, ya que rompería la estructura.

Una analogía para esto sería la fila que se forma para pagar los productos de una tienda. Los clientes entran por el extremo trasero de la cola, y se acumulan uno detrás del otro. Luego, al ser atendidos, se atiende al que está al frente, este sale y se atiende al que está atrás suyo, y así sucesivamente hasta que todas las personas salgan. Es muy importante notar que primero se atiende al primero que se formó en la fila, y que no es posible que una persona se “cole” dentro de la fila [2].



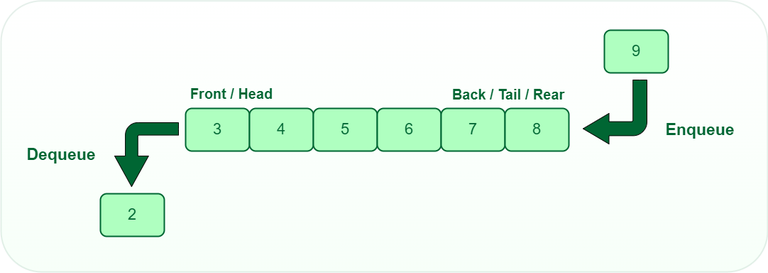
**Operaciones:  
isEmpty():** Similar a las pilas, este método verifica si la cola está vacía, devolviendo true si no hay elementos.

**size():** Retorna el número de elementos presentes en la cola, permitiendo el seguimiento del volumen de la cola.

**peek():** Permite acceder al primer elemento de la cola sin eliminarlo, proporcionando una forma de ver el próximo elemento a procesar sin modificar la cola.

**enqueue() / add():** Agrega un elemento al final de la cola. En Java, este método puede tener nombres como add(e) o offer(e), dependiendo de si la implementación debe devolver un valor booleano o lanzar una excepción si no puede agregar el elemento.

**dequeue() / remove():** Elimina el elemento al frente de la cola y lo devuelve. Similar al método add(), puede existir como remove() que lanza una excepción si la cola está vacía, o poll(), que devuelve null si la cola no tiene elementos.



**Utilidad:** Las colas son útiles en la programación de eventos, gestión de tareas y algoritmos de buffering, donde el orden de los elementos debe ser preservado. También se usa una implementación con colas para algoritmos como la ordenación raíz o el algoritmo de Dijkstra.

**Implementación:** En Java, las colas se implementan comúnmente usando Queue<E> de la biblioteca Java Utilities (java.util), y se utilizan clases como LinkedList<E> o PriorityQueue<E> para diferentes comportamientos de ordenación y acceso. La implementación mediante estructuras enlazadas tiene la ventaja de que la mayoría de operaciones son de complejidad .

Al igual que el caso de Stack<E> de Java, la PriorityQueue<E> de Java tiene algunos métodos, como remove(), que violan la idea de cómo funciona una cola, ya que elimina un elemento independientemente de su posición dentro de la cola, cuando sólo debería ser posible interactuar por uno de los extremos.

En la comparación entre pilas (stacks) y colas (queues), ambos tipos de estructuras de datos manejan elementos con estrategias opuestas que influencian directamente su aplicación en escenarios de programación específicos. Las pilas, que operan bajo el principio de "último en entrar, primero en salir" (LIFO), son ideales para tareas que requieren un acceso inverso. Por otro lado, las colas, adheridas al principio de "primero en entrar, primero en salir" (FIFO), son cruciales en procesos donde se debe preservar el orden de llegada, tal como en la planificación de tareas y el manejo de colas de espera. Las pilas suelen implementarse con listas enlazadas que permiten expansión y contracción dinámica, facilitando operaciones rápidas de inserción y eliminación sin preocupaciones sobre el tamaño predefinido, mientras que las colas pueden usar tanto listas enlazadas como arreglos circulares, Esta distinción conceptual no solo afecta la elección técnica entre una estructura u otra, sino también la eficacia con la que se manejan los datos en diferentes contextos de programación, haciendo esencial una comprensión detallada de sus características y aplicaciones prácticas.

## Conclusiones

Las pilas y colas son estructuras de datos muy útiles al tratar problemas que impliquen mantener un orden de llegada en los datos. Son una de las estructuras de datos más básicas que existen, pero tienen múltiples aplicaciones en varios ámbitos, y son muy prácticas a la hora de tratar con problemas que requieran mantener un orden de llegada o establecer un algoritmo de salida, como puede ser el LIFO o FIFO.

Es muy importante conocer sus características e implementación para poder saber usarlas cuando la situación lo requiera, así como modificarlas según las necesidades del programador.

## Referencias bibliográficas

[1]

M. A. Weiss, *Estructuras de Datos Y Algoritmos*. Buenos Aires: Addison-Wesley Iberoamericana, 1995.

[2]

J. Lewis and J. Chase, *Estructuras de datos con Java: Diseño de estructuras y algoritmos*. Boston: Pearson-Addison Wesley, 2006.