ESTRUCTURA DE DATOS LINEALES EN JAVA: UN ENFOQUE PRÁCTICO

Fredy Gavilanes-Sagnay
Luis Alberto Castillo Salinas
Luis Armando Ortiz Delgado
Luis Manuel Chica Moncayo

CASA EDITORA DEL POLO

Estructura de Datos Lineales en Java: Un Enfoque práctico

Fredy Gavilanes-Sagnay
Facultad de Ingeniería, Universidad Nacional del Chimborazo, Ecuador
UNACH-Universidad Nacional de Chimborazo

Luis Alberto Castillo Salinas
Departamento de Ciencias de la Computación, Universidad de las Fuerzas Armadas-ESPE sede Santo Domingo de los Tsáchilas Santo Domingo
ESPE- Universidad de las Fuerzas Armadas -ESPE

Luis Armando Ortiz Delgado
Departamento de Ciencias de la Computación, Universidad de las Fuerzas Armadas-ESPE sede Santo Domingo de los Tsáchilas- Santo Domingo
ESPE- Universidad de las Fuerzas Armadas -ESPE

Luis Manuel Chica Moncayo
Departamento de Ciencias de la Computación, Universidad de las Fuerzas Armadas-ESPE sede Santo Domingo de los Tsáchilas- Santo Domingo
ESPE- Universidad de las Fuerzas Armadas -ESPE

CASA EDITORA DEL POLO

Casa Editora del Polo - CASEDELPO CIA. LTDA.
Departamento de Edición

Editado y distribuido por:

Editorial: Casa Editora del Polo
Sello Editorial: 978-9942-816
Manta, Manabí, Ecuador. 2019
Teléfono: (05) 6051775 / 0991871420
Web: www.casedelpo.com
ISBN: 978-9942-621-52-8

Revisión, Ortografía y Redacción:
Lic. Jessica Mero Vélez

Diseño de Portada:
Michael Josué Suárez-Espinar

Diagramación:
Ing. Edwin Alejandro Delgado-Veliz

Director Editorial:
Dra. Tibisay Milene Lamus-García

Constancia de Arbitraje

La Casa Editora del Polo, hace constar que este libro proviene de una investigación realizada por los autores, siendo sometido a un arbitraje bajo el sistema de doble ciego (peer review), de contenido y forma por jurados especialistas. Además, se realizó una revisión del enfoque, paradigma y método investigativo; desde la matriz epistémica asumida por los autores, aplicándose las normas APA, Sexta Edición, proceso de anti plagio en línea Plagiarisma, garantizándose así la cientificidad de la obra.

Comité Editorial

CONTENIDO

CAPÍTULO I

INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

Las estructuras de datos lineales son fundamentales en la ciencia de la computación y desempeñan un papel crucial en el almacenamiento y manipulación de datos de manera organizada. Estas estructuras permiten representar conjuntos de elementos de forma secuencial, donde el acceso a los datos se realiza de manera secuencial siguiendo un orden específico. Algunos ejemplos comunes de estructuras de datos lineales son las listas, pilas y colas.

Las listas son estructuras de datos lineales que permiten almacenar una colección de elementos en la que cada elemento se enlaza con el siguiente a través de referencias. Según Goodrich, Tamassia y Goldwasser (2014), "las listas proporcionan flexibilidad en cuanto a la inserción y eliminación de elementos en cualquier posición de la lista" (p. 78). Las listas pueden ser implementadas como listas enlazadas, donde cada elemento tiene un puntero que apunta al siguiente elemento, o como listas basadas en arrays, donde los elementos se almacenan en un array y se accede a ellos mediante índices.

En las listas enlazadas, cada elemento, conocido como nodo, contiene un dato y un puntero que apunta al siguiente nodo de la lista. Esto permite un acceso eficiente a los elementos y facilita la inserción y eliminación de elementos en cualquier posición de la lista. Por otro lado, en las listas basadas en arrays, los elementos se almacenan en un array contiguo de memoria, lo que

permite un acceso aleatorio a los elementos a través de sus índices. Aunque las operaciones de inserción y eliminación pueden ser más costosas en términos de tiempo en comparación con las listas enlazadas, las listas basadas en arrays ofrecen un acceso directo a los elementos.

Las pilas son otra estructura de datos lineal común que sigue el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés). Según Zhang, Luo y Li (2020), "las pilas son ampliamente utilizadas en problemas que requieren un manejo eficiente de tareas o elementos en un orden específico" (p. 156). Las pilas pueden implementarse utilizando una lista enlazada o un array. En una lista enlazada, la inserción y eliminación de elementos se realizan en el extremo superior de la pila, conocido como la cima. En un array, la cima de la pila se representa mediante un índice y las operaciones de inserción y eliminación se realizan moviendo dicho índice.

Las colas son estructuras de datos lineales que siguen el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés). Según Oracle (2021), "las colas son adecuadas para situaciones en las que es importante mantener un orden estricto de los elementos" (sección "Java LinkedList Class"). Al igual que las pilas, las colas se pueden implementar utilizando una lista enlazada o un array. En una lista enlazada, los elementos se insertan en la parte trasera de la cola y se eliminan

desde el frente. En un array, se utilizan dos índices para representar la parte delantera y trasera de la cola, y las operaciones de inserción y eliminación se realizan desplazando estos índices.

En conclusión, las estructuras de datos lineales son elementos fundamentales en la ciencia de la computación y permiten organizar y manipular datos de manera secuencial. Las listas, pilas y colas son ejemplos comunes de estructuras de datos lineales. Las listas proporcionan flexibilidad en la inserción y eliminación de elementos en cualquier posición. Las pilas siguen el principio LIFO y son útiles en situaciones que requieren un manejo eficiente de elementos en un orden específico. Las colas siguen el principio FIFO y son adecuadas para mantener un orden estricto de elementos. Estas estructuras se pueden implementar utilizando listas enlazadas o arrays, cada uno con sus ventajas y desventajas en términos de acceso y eficiencia.

1.1. Definición y características de las estructuras de datos lineales

Las estructuras de datos lineales son elementos fundamentales en la ciencia de la computación que permiten organizar y manipular datos de forma secuencial, donde el acceso se realiza siguiendo un orden específico. Estas estructuras ofrecen flexibilidad y eficiencia para el almacenamiento y procesamiento de datos en aplicaciones informáticas.

En su libro “Data Structures and Algorithms in Java”, Goodrich, Tamassia y Goldwasser (2014) definen las estructuras de datos lineales como “colecciones de elementos donde cada elemento tiene un sucesor bien definido y, excepto por el primero y el último, cada elemento tiene un predecesor bien definido” (p. 78). Esto implica que los elementos se organizan en una secuencia lineal, donde cada elemento se conecta con su elemento adyacente.

Una característica importante de las estructuras de datos lineales es que permiten la inserción y eliminación de elementos en cualquier posición de la secuencia. Por ejemplo, en una lista enlazada, cada elemento tiene un puntero que apunta al siguiente elemento, lo que permite la inserción y eliminación eficiente en cualquier parte de la lista. De acuerdo con Goodrich et al. (2014), las listas enlazadas proporcionan flexibilidad en cuanto a la inserción y eliminación de elementos en cualquier posición de la lista.

Además, las estructuras de datos lineales se pueden implementar de diferentes maneras. Una opción común es la utilización de una lista enlazada, donde cada elemento contiene un enlace o puntero que apunta al siguiente elemento de la secuencia. Esta implementación permite una inserción y eliminación eficiente de elementos en cualquier parte de la lista, ya que solo se requieren modificaciones en los enlaces de los elementos adyacentes.

Otra opción es utilizar una lista basada en arrays, donde los elementos se almacenan en un arreglo contiguo de memoria. En este caso, cada elemento se accede mediante un índice numérico. Aunque las operaciones de inserción y eliminación pueden ser más costosas en términos de tiempo en comparación con las listas enlazadas, las listas basadas en arrays ofrecen un acceso aleatorio más eficiente a los elementos, ya que no es necesario recorrer la secuencia para acceder a un elemento en una posición específica.

Además de las listas, existen otras estructuras de datos lineales ampliamente utilizadas, como las pilas y las colas. Una pila es una estructura de datos lineal que sigue el principio de “último en entrar, primero en salir” (LIFO, por sus siglas en inglés). Esto significa que el último elemento que se inserta en la pila es el primero en ser eliminado. Las pilas son utilizadas en numerosas aplicaciones, como la evaluación de expresiones matemáticas, el manejo de llamadas de funciones y la implementación de algoritmos recursivos.

Según Zhang, Luo y Li (2020), las pilas son ampliamente utilizadas en problemas que requieren un manejo eficiente de tareas o elementos en un orden específico. La estructura de datos de pila puede implementarse utilizando tanto una lista enlazada como un array.

Por otro lado, una cola es una estructura de datos

lineal que sigue el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés). En una cola, los elementos se insertan en la parte posterior y se eliminan desde el frente. Las colas son adecuadas para situaciones en las que es importante mantener un orden estricto de los elementos, como la planificación de tareas y la gestión de procesos en sistemas operativos.

En resumen, las estructuras de datos lineales son fundamentales en la ciencia de la computación y se caracterizan por organizar y manipular datos de forma secuencial. Estas estructuras permiten la inserción y eliminación de elementos en cualquier posición de la secuencia, lo que proporciona flexibilidad en el manejo de los datos. Las listas enlazadas y las listas basadas en arrays son implementaciones comunes de las estructuras de datos lineales. Además, las pilas y las colas son estructuras lineales especializadas que siguen diferentes principios de ordenamiento y son ampliamente utilizadas en diversas aplicaciones y algoritmos.

1.2. Importancia y aplicaciones en la programación

La programación es una disciplina clave en el campo de la ciencia de la computación, y las estructuras de datos lineales desempeñan un papel fundamental en el desarrollo de algoritmos eficientes y la optimización de la gestión de datos. En este tema, exploraremos la importancia de las estructuras de datos lineales y su amplio espectro de aplicaciones en la programación moderna.

1.2.1. Importancia de las estructuras de datos lineales

Las estructuras de datos lineales son fundamentales en la programación debido a su capacidad para organizar y manipular datos de manera secuencial. Proporcionan una forma eficiente de almacenar y acceder a los datos en diferentes escenarios. Al utilizar estructuras de datos lineales adecuadas, los programadores pueden optimizar el rendimiento de sus algoritmos y mejorar la eficiencia de sus programas.

Una de las principales razones por las que las estructuras de datos lineales son importantes en la programación es su capacidad para manejar grandes volúmenes de datos de manera eficiente. Cuando se trabaja con una gran cantidad de información, es crucial utilizar estructuras de datos que permitan un acceso y manipulación eficientes de los datos. Las estructuras de datos lineales, como las listas, pilas y colas, ofrecen métodos de inserción, eliminación y búsqueda que se pueden implementar de manera eficiente para garantizar un rendimiento óptimo.

Además, las estructuras de datos lineales proporcionan una mayor flexibilidad en la gestión de datos en comparación con otras estructuras más simples, como los arreglos. Por ejemplo, en una lista enlazada, es posible insertar y eliminar elementos en cualquier posición, lo que permite una adaptabilidad dinámica según las necesidades del programa. Esta capacidad

de adaptación es especialmente valiosa en situaciones donde los datos cambian con el tiempo o cuando se requiere un manejo dinámico de los mismos.

Otra razón importante para utilizar estructuras de datos lineales en la programación es la capacidad de compartir y reutilizar código. Al utilizar estructuras de datos lineales comunes, los programadores pueden aprovechar bibliotecas y módulos existentes que ya han sido probados y optimizados. Esto acelera el proceso de desarrollo y reduce la posibilidad de errores.

1.2.2. Aplicaciones de las estructuras de datos lineales en la programación

Las estructuras de datos lineales tienen una amplia gama de aplicaciones en la programación. A continuación, se presentan algunas de las aplicaciones más comunes:

Procesamiento de texto: Las estructuras de datos lineales son esenciales para el procesamiento de texto, donde las cadenas de caracteres se almacenan y manipulan de manera secuencial. Por ejemplo, se pueden utilizar listas enlazadas para implementar un editor de texto que permita la inserción y eliminación de caracteres en cualquier posición.

Gestión de archivos: En muchas aplicaciones, es necesario manejar y organizar grandes volúmenes de archivos. Las estructuras de datos lineales, como las listas y colas, se utilizan para mantener el orden y

realizar operaciones eficientes de búsqueda, inserción y eliminación de archivos.

Recorrido de árboles y grafos: Las estructuras de datos lineales también son fundamentales en el recorrido de estructuras más complejas, como árboles y grafos. Por ejemplo, se pueden utilizar pilas para implementar el recorrido en profundidad (DFS) en un árbol o un grafo.

Implementación de algoritmos: Las estructuras de datos lineales son utilizadas en la implementación de algoritmos para resolver una variedad de problemas. Por ejemplo, las pilas se utilizan en la evaluación de expresiones matemáticas y en el algoritmo de retroceso (backtracking), mientras que las colas se utilizan en el algoritmo de búsqueda en amplitud (BFS) y en la planificación de tareas.

Estructuras de datos avanzadas: Las estructuras de datos lineales también se utilizan como componentes clave en la implementación de estructuras de datos más avanzadas. Por ejemplo, las listas enlazadas se utilizan para implementar listas circulares, listas doblemente enlazadas y otras estructuras más complejas.

En resumen, las estructuras de datos lineales son de vital importancia en la programación. Proporcionan una forma eficiente de organizar y manipular datos de manera secuencial, permitiendo un acceso y gestión eficientes de la información. Las aplicaciones de las estructuras de datos lineales en la programación son

numerosas, abarcando desde el procesamiento de texto hasta la implementación de algoritmos y estructuras de datos más complejas. Al comprender la importancia y las aplicaciones de las estructuras de datos lineales, los programadores pueden mejorar la eficiencia y la calidad de sus programas.

CAPÍTULO II

CONCEPTOS BÁSICOS DE PROGRAMACIÓN ORIENTADA A OBJETOS EN JAVA

La programación orientada a objetos (POO) es un paradigma de programación ampliamente utilizado en el desarrollo de software. En este tema, exploraremos los conceptos básicos de la programación orientada a objetos en el contexto de Java, uno de los lenguajes de programación más populares. Comprender estos conceptos es fundamental para construir aplicaciones robustas y modulares en Java.

2.1. Encapsulación

La encapsulación es un principio clave en la programación orientada a objetos que implica agrupar datos y los métodos que operan sobre esos datos en una sola entidad llamada "objeto". Esto permite ocultar los detalles internos y proteger los datos dentro del objeto. En Java, se utilizan modificadores de acceso, como public, private y protected, para controlar el acceso a los datos y métodos de un objeto. Por ejemplo, un atributo privado solo es accesible dentro de la clase en la que se declara. Como señala Horstmann (2018), "el principio de encapsulamiento protege la integridad de los datos y proporciona una forma controlada de interactuar con los objetos" (p. 143).

2.1.1. Definición y Principios de Encapsulación

La encapsulación se define como el mecanismo de ocultar los detalles internos de un objeto y proporcionar una interfaz controlada para acceder a esos detalles. Según Liang (2019), la encapsulación "combina datos y

comportamientos en un solo componente llamado objeto y oculta los detalles internos de cómo se implementa el objeto" (p. 232).

Uno de los principios clave de la encapsulación es el principio de ocultamiento de información, que se refiere a la idea de que los detalles internos de un objeto no deben ser accesibles directamente desde fuera del objeto. En su lugar, se debe proporcionar una interfaz controlada para interactuar con el objeto. Como señalan Deitel y Deitel (2017), la encapsulación "permite que los objetos se comuniquen entre sí mediante mensajes, protegiendo sus detalles internos de accesos no autorizados" (p. 144).

2.1.2. Modificadores de Acceso

En la programación orientada a objetos, los modificadores de acceso se utilizan para controlar el nivel de acceso a los miembros (atributos y métodos) de una clase. Java, como lenguaje orientado a objetos, proporciona cuatro modificadores de acceso: public, private, protected y default.

El modificador de acceso public permite que los miembros sean accesibles desde cualquier clase o paquete. Por otro lado, el modificador de acceso private restringe el acceso a los miembros solo dentro de la misma clase. El modificador de acceso protected permite el acceso a los miembros dentro de la misma clase, el mismo paquete y las clases derivadas (subclases). El modificador de acceso default, que no se especifica

explícitamente, permite el acceso a los miembros dentro del mismo paquete.

La utilización de estos modificadores de acceso es esencial para implementar la encapsulación. De acuerdo con Sierra y Bates (2020), "la clave para lograr la encapsulación es que los atributos de un objeto deben ser declarados como privados, y luego se deben proporcionar métodos públicos para acceder y modificar esos atributos" (p. 193).

2.1.3. Beneficios de la Encapsulación

La encapsulación ofrece una serie de beneficios en el desarrollo de software orientado a objetos:

- Ocultamiento de detalles internos: La encapsulación permite ocultar los detalles internos de un objeto, lo que proporciona una interfaz más clara y facilita el mantenimiento y la evolución del código. Los cambios internos de implementación de un objeto no afectan a las clases que lo utilizan, siempre y cuando la interfaz pública se mantenga inalterada.

- Protección de datos: Al declarar los atributos como privados y proporcionar métodos de acceso controlados, la encapsulación protege los datos de un objeto de accesos no autorizados o modificaciones incorrectas. Solo los métodos definidos en el objeto pueden acceder y manipular sus datos internos.

• Modularidad: La encapsulación fomenta la creación de módulos de software independientes y cohesivos. Los objetos encapsulados se convierten en unidades autónomas y se pueden desarrollar y probar de forma aislada. Esto promueve la reutilización de código y facilita la colaboración en equipos de desarrollo.

• Facilidad de uso: La encapsulación proporciona una interfaz clara y coherente para interactuar con los objetos. Los usuarios del objeto solo necesitan conocer los métodos de acceso públicos disponibles, sin necesidad de preocuparse por los detalles internos. Esto simplifica el uso del objeto y reduce la posibilidad de errores de programación.

2.1.4. Ejemplo de Encapsulación en Java

Para ilustrar la encapsulación en Java, consideremos un ejemplo de una clase "Cuenta Bancaria" que encapsula información sobre una cuenta bancaria y proporciona métodos para acceder y modificar esa información. Los atributos privados de la clase podrían incluir el número de cuenta, el saldo y el titular de la cuenta.

Clase Cuenta Bancaria

```
public class CuentaBancaria {
    private int numeroCuenta;
    private double saldo;
    private String titular;
    public void depositar(double monto) {
        // Lógica para realizar el depósito
    }
    public void retirar(double monto) {
        // Lógica para realizar el retiro
    }
    public double obtenerSaldo() {
        return saldo;
    }
}
```

En este ejemplo, los atributos de la clase "CuentaBancaria" se declaran como privados para encapsular los datos. Los métodos públicos, como "depositar", "retirar" y "obtenerSaldo", proporcionan una interfaz controlada para interactuar con la cuenta bancaria. Esto garantiza que cualquier operación realizada en la cuenta se realice a través de estos métodos, lo que permite la validación y el control adecuados.

2.2. Herencia

La herencia es un mecanismo en la programación orientada a objetos que permite crear nuevas clases basadas en clases existentes. En Java, una clase puede heredar atributos y métodos de otra clase utilizando

la palabra clave “extends”. La clase base se denomina “superclase” o “clase padre”, y la clase derivada se denomina “subclase” o “clase hija”. La herencia permite la reutilización de código y facilita la creación de jerarquías de clases. Según Liang (2019), “la herencia permite modelar relaciones entre objetos y promueve la reutilización de código” (p. 230).

2.2.1. Definición y Principios de Herencia

La herencia se define como el mecanismo mediante el cual una clase adquiere las propiedades (atributos y métodos) de otra clase. Según Deitel y Deitel (2017), “la herencia nos permite definir una clase en función de otra clase existente” (p. 328). La clase original se denomina clase base o superclase, y la nueva clase creada se conoce como clase derivada o subclase.

Uno de los principios clave de la herencia es el principio de sustitución, que establece que los objetos de una clase derivada pueden usarse en lugar de objetos de la clase base sin afectar el comportamiento del programa. Esto significa que los objetos de la subclase heredan todas las características de la superclase y también pueden tener sus propias características adicionales.

2.2.2. Relación entre Superclase y Subclase

En la herencia, la superclase y la subclase están estrechamente relacionadas. La subclase hereda todos los miembros (atributos y métodos) de la superclase y puede agregar nuevos miembros o modificar los

miembros heredados. La relación entre la superclase y la subclase se denota utilizando el término “es-un”. Por ejemplo, si tenemos una clase “Vehículo” como superclase y una clase “Automóvil” como subclase, podemos decir que “Automóvil es un Vehículo”.

2.2.3. Beneficios de la Herencia

La herencia ofrece una serie de beneficios en el desarrollo de software orientado a objetos:

- Reutilización de código: La herencia permite la reutilización de código al aprovechar los miembros de la superclase en la subclase. Los atributos y métodos heredados no necesitan ser reescritos en la subclase, lo que ahorra tiempo y esfuerzo en el desarrollo de software.

- Extensibilidad: La herencia permite extender y especializar las clases existentes. La subclase puede agregar nuevos atributos y métodos específicos que son relevantes para su contexto, mientras hereda el comportamiento general de la superclase. Esto permite la adaptación y personalización del código base.

- Mantenibilidad: La herencia facilita la estructura y el mantenimiento del código. Al agrupar características comunes en una superclase y definir comportamientos específicos en las subclases, se crea una jerarquía de clases clara y organizada. Esto mejora la legibilidad, facilita la depuración y reduce la duplicación de código.

- Polimorfismo: La herencia es esencial para el concepto de polimorfismo, que permite que los objetos de diferentes clases se comporten de manera similar. Al usar referencias de la superclase para objetos de la subclase, se puede acceder a los métodos y atributos comunes a través de la herencia, lo que simplifica la implementación de algoritmos genéricos.

2.2.4.Ejemplo de Herencia en Java

Para ilustrar la herencia en Java, consideremos un ejemplo de una superclase "Vehículo" y una subclase "Automóvil":

Clase Vehiculo

```
public class Vehiculo {
    private String marca;
    private int año;
    public void acelerar() {
        // Lógica para acelerar
    }
    public void frenar() {
        // Lógica para frenar
    }
}
public class Automóvil extends Vehículo {
    private int cilindrada;
  public void encender() {
        // Lógica para encender el automóvil
    }
    public void apagar() {
        // Lógica para apagar el automóvil
   }
}
```

En este ejemplo, la clase "Vehículo" actúa como la superclase y contiene los atributos y métodos comunes a todos los vehículos. La clase "Automóvil" hereda de "Vehículo" utilizando la palabra clave "extends" y agrega su propio atributo "cilindrada" y métodos específicos.

2.3. Polimorfismo

El polimorfismo es otro concepto fundamental en la programación orientada a objetos en Java. Se refiere a la capacidad de un objeto de tomar diferentes formas y comportarse de manera diferente según el contexto. El polimorfismo se logra mediante la herencia y la implementación de métodos con el mismo nombre en diferentes clases. El polimorfismo permite escribir código más genérico y flexible, lo que facilita el mantenimiento y la extensión del software. Según Deitel y Deitel (2017), "el polimorfismo proporciona una forma poderosa de programación orientada a objetos al permitir que los objetos de diferentes clases respondan a las mismas operaciones" (p. 269).

2.3.1. Definición y Principios de Polimorfismo

El polimorfismo se define como la capacidad de un objeto de tomar muchas formas. Según Liang (2019), "el polimorfismo permite que un mismo método se comporte de diferentes maneras en diferentes clases" (p. 234). En otras palabras, objetos de diferentes clases pueden responder de manera diferente a la misma llamada de método.

El polimorfismo se basa en dos principios clave: el enlace dinámico y la herencia. El enlace dinámico se refiere al proceso en el cual el método que se invoca se determina en tiempo de ejecución en función del tipo real del objeto, en lugar del tipo declarado de la

referencia. La herencia es la base del polimorfismo, ya que las subclases pueden sobrescribir los métodos heredados de las superclases para proporcionar su propia implementación.

2.3.2. Tipos de Polimorfismo

Existen dos tipos principales de polimorfismo en la programación orientada a objetos: polimorfismo de tiempo de compilación y polimorfismo de tiempo de ejecución.

Polimorfismo de tiempo de compilación: También conocido como enlace estático o polimorfismo estático, se refiere a la selección de un método en tiempo de compilación en función del tipo declarado de la referencia. Este tipo de polimorfismo se logra mediante la sobrecarga de métodos, donde diferentes métodos tienen el mismo nombre pero diferentes parámetros.

Polimorfismo de tiempo de ejecución: También conocido como enlace dinámico o polimorfismo dinámico, se refiere a la selección de un método en tiempo de ejecución en función del tipo real del objeto. Este tipo de polimorfismo se logra mediante la sobrescritura de métodos, donde una subclase proporciona una implementación específica de un método heredado de su superclase.

2.3.3. Beneficios del Polimorfismo

El polimorfismo ofrece una serie de beneficios en el

desarrollo de software orientado a objetos:

•Flexibilidad y extensibilidad: El polimorfismo permite escribir código genérico y flexible que puede manejar objetos de diferentes clases. Esto facilita la extensibilidad del código, ya que se pueden agregar nuevas clases sin modificar el código existente.

•Reutilización de código: Al utilizar referencias de tipo superclase para objetos de diferentes subclases, se puede reutilizar el código existente. Esto promueve la modularidad y evita la duplicación de código.

•Mantenimiento simplificado: El polimorfismo facilita el mantenimiento del código al permitir que se realicen cambios en una clase sin afectar el código que utiliza objetos de esa clase. Esto ayuda a reducir el acoplamiento y a mantener una base de código más limpia y organizada.

•Polimorfismo en colecciones: El polimorfismo es especialmente útil en el manejo de colecciones de objetos. Al utilizar referencias de tipo superclase en lugar de tipos específicos, se pueden crear colecciones genéricas que pueden contener objetos de diferentes subclases.

2.3.4. Ejemplo de Polimorfismo en Java

Para ilustrar el polimorfismo en Java, consideremos un ejemplo de una superclase “Figura” y dos subclases “Rectángulo” y “Círculo”. Ambas subclases implementan el método “calcularÁrea()”, pero cada una proporciona

su propia implementación:

Clase Figura

```
public abstract class Figura {
    public abstract double calcularÁrea();
}
public class Rectángulo extends Figura {
    private double base;
    private double altura;
    public double calcularÁrea() {
        return base * altura;
    }
}
public class Círculo extends Figura {
    private double radio;
    public double calcularÁrea() {
        return Math.PI * Math.pow(radio, 2);
    }
}
```

En este ejemplo, la superclase "Figura" declara el método "calcularÁrea()" como abstracto, lo que significa que las subclases deben proporcionar una implementación concreta. Las subclases "Rectángulo" y "Círculo" sobrescriben el método "calcularÁrea()" y proporcionan su propia lógica de cálculo del área.

El polimorfismo es un concepto esencial en la programación orientada a objetos que permite que objetos de diferentes clases se comporten de manera similar. Al utilizar el enlace dinámico y la herencia, el polimorfismo ofrece flexibilidad, reutilización de código y mantenibilidad en el desarrollo de software. Los beneficios del polimorfismo incluyen la capacidad

de escribir código genérico, la extensibilidad, la modularidad y el manejo eficiente de colecciones de objetos. Al comprender y aplicar el polimorfismo, los programadores pueden escribir código más flexible y escalable.

2.4. Abstracción

La abstracción es un concepto esencial en la programación orientada a objetos que implica enfocarse en los aspectos esenciales de un objeto y ocultar los detalles de implementación. En Java, se pueden definir clases abstractas e interfaces para establecer contratos y definir comportamientos comunes. Una clase abstracta es una clase que no se puede instanciar y puede contener métodos abstractos, mientras que una interfaz es una colección de métodos abstractos que se pueden implementar en diferentes clases. La abstracción permite modelar objetos y acciones de manera más general y concisa. Como menciona Sierra y Bates (2020), "la abstracción nos permite enfocarnos en el 'qué' en lugar del 'cómo'" (p. 61).

2.4.1. Definición de Abstracción

La abstracción se define como el proceso de identificar las características y comportamientos esenciales de un objeto y representarlos en un modelo conceptual. Según Deitel y Deitel (2017), "la abstracción simplifica un objeto real o un concepto en un modelo de programación que incluye las características y comportamientos más

relevantes" (p. 134). En otras palabras, la abstracción nos permite centrarnos en los aspectos importantes de un objeto y omitir los detalles irrelevantes.

2.4.2. Importancia de la Abstracción

La abstracción juega un papel crucial en el desarrollo de software orientado a objetos por varias razones:

• Simplificación del modelado: La abstracción nos permite representar objetos complejos del mundo real de manera simplificada en el software. Nos permite identificar las características y comportamientos esenciales de un objeto y modelarlos de manera eficiente.

• Ocultamiento de la complejidad: La abstracción nos permite ocultar los detalles internos de un objeto y proporcionar una interfaz clara y consistente para interactuar con él. Esto promueve la encapsulación y evita la exposición de detalles innecesarios, lo que facilita el mantenimiento y la evolución del software.

•Reutilización de código: Al abstraer las características y comportamientos comunes de varios objetos en una clase base, podemos reutilizar código a través de la herencia y la creación de jerarquías de clases. Esto promueve la modularidad y evita la duplicación de código.

•Facilitación del diseño y desarrollo: La abstracción nos permite separar el diseño conceptual del diseño de implementación. Podemos crear modelos conceptuales

que representen las entidades y relaciones del dominio, lo que facilita la comunicación entre los miembros del equipo de desarrollo y permite un diseño más estructurado y coherente.

2.4.3. Aplicación de la Abstracción en POO

En la programación orientada a objetos, la abstracción se aplica mediante el uso de clases e interfaces. Una clase define una estructura que representa un objeto y encapsula sus datos y comportamientos. Proporciona una abstracción de un tipo de objeto específico y define cómo interactuar con él.

Una interfaz, por otro lado, define un conjunto de métodos que una clase concreta debe implementar. Proporciona una abstracción de un comportamiento común compartido por varias clases. Las interfaces permiten la creación de código genérico y promueven la reutilización y la extensibilidad.

2.4.4. Ejemplo de Abstracción en Java

Para ilustrar la abstracción en Java, consideremos un ejemplo de una clase "Vehículo" y sus subclases "Automóvil" y "Motocicleta". La clase "Vehículo" podría tener métodos como "acelerar()", "frenar()" y "obtenerVelocidad()", mientras que las subclases proporcionarían implementaciones específicas de estos métodos.

Clase Vehículo

```
public abstract class Vehiculo {
    private int velocidad;
    public void acelerar() {
        // Lógica para aumentar la velocidad
    }
    public void frenar() {
        // Lógica para disminuir la velocidad
    }
    public int obtenerVelocidad() {
        return velocidad;
    }
}
public class Automóvil extends Vehículo {
    // Implementación específica para Automóvil
}
public class Motocicleta extends Vehículo {
    // Implementación específica para Motocicleta
}
```

En este ejemplo, la clase abstracta “Vehículo” define la abstracción común para todos los vehículos. Proporciona métodos genéricos que son aplicables a cualquier vehículo, como “acelerar()” y “frenar()”. Las subclases “Automóvil” y “Motocicleta” heredan de la clase “Vehículo” y proporcionan sus propias implementaciones específicas de los métodos.

La abstracción es un concepto clave en la programación orientada a objetos que permite simplificar y representar objetos complejos en el software. Proporciona un nivel de abstracción adecuado para comprender y manipular

objetos sin conocer todos los detalles internos de su implementación. La abstracción es importante para simplificar el modelado, ocultar la complejidad, reutilizar código y facilitar el diseño y desarrollo de software. En la programación orientada a objetos, la abstracción se logra a través del uso de clases e interfaces. Mediante la abstracción, podemos crear modelos conceptuales claros y estructurados que representen entidades del mundo real y su interacción en el software.

2.5. Instanciación y uso de objetos

En la programación orientada a objetos, los objetos se crean a partir de una clase utilizando el proceso de instanciación. En Java, se utiliza el operador “new” para crear una instancia de una clase. Una vez que se crea un objeto, se puede acceder a sus atributos y métodos utilizando el operador de acceso punto. Por ejemplo, para acceder a un atributo “nombre” de un objeto “persona”, se utiliza la expresión “persona.nombre”. Según Liang (2019), “la instanciación de objetos nos permite crear múltiples instancias de una clase y trabajar con ellos en tiempo de ejecución” (p. 83).

2.5.1. Definición de instanciación y objeto

La instanciación es el proceso de crear un objeto específico a partir de una clase. Según Gaddis (2019), “una clase es una plantilla para crear objetos y define los datos y métodos que los objetos contendrán” (p. 258). Por otro lado, un objeto es una instancia única

de una clase que encapsula datos y comportamientos relacionados. Cada objeto tiene su propio estado (datos) y comportamiento (métodos) definido por su clase.

2.5.2. Proceso de instanciación

La instanciación de un objeto implica tres pasos básicos:

•Declaración: En este paso, se declara una variable del tipo de la clase a la que pertenece el objeto. La declaración especifica el nombre de la variable y su tipo. Por ejemplo, si tenemos una clase llamada “Persona”, podemos declarar una variable llamada “persona” de la siguiente manera en Java:

Clase Persona
Persona persona;

•Creación: Una vez que se ha declarado la variable, se utiliza el operador “new” para crear una instancia del objeto. El operador “new” reserva memoria para el objeto y llama al constructor de la clase para inicializarlo. Por ejemplo, para crear una instancia de la clase “Persona”, se utiliza el siguiente código:

Clase Persona
persona = new Persona();

Inicialización: Después de crear el objeto, se pueden asignar valores a sus atributos utilizando los métodos establecedores (setters) o accediendo directamente a los atributos si son públicos. Esto permite definir el estado

inicial del objeto. Por ejemplo, si la clase “Persona” tiene un atributo “nombre”, se puede asignar un valor utilizando el método establecedor de la siguiente manera:

Clase Persona
persona.setNombre(“Juan”);

Uso de objetos

Una vez que se ha creado un objeto, se puede acceder a sus atributos y métodos para realizar diferentes operaciones. Para acceder a un atributo de un objeto, se utiliza la notación de punto (“.”) seguida del nombre del atributo. Por ejemplo, si tenemos una clase “Persona” con un atributo “nombre”, podemos acceder a él de la siguiente manera:

Clase Persona
String nombre = persona.getNombre();

En este caso, estamos utilizando el método getter (“getNombre()”) para obtener el valor del atributo “nombre” del objeto “persona”.

Del mismo modo, para llamar a un método de un objeto, se utiliza la notación de punto seguida del nombre del método y, si corresponde, se pasan los argumentos necesarios entre paréntesis. Por ejemplo, si tenemos una clase “Persona” con un método “saludar()”, podemos llamar a ese método de la siguiente manera:

Clase Persona
persona.saludar();

En este caso, estamos llamando al método “saludar()” del

objeto “persona”, que imprimirá un saludo en la consola.

Además de acceder a los atributos y métodos de un objeto, se pueden utilizar operaciones adicionales, como la comparación de objetos utilizando el operador “==”, la llamada a métodos encadenados o la asignación de un objeto a otro. Estas operaciones permiten realizar tareas más complejas y manipular objetos de manera eficiente.

2.5.4. Importancia de la instanciación y uso de objetos

La instanciación y el uso de objetos son fundamentales en la POO y brindan varias ventajas:

•Modularidad: La instanciación y el uso de objetos permiten dividir un programa en módulos independientes y reutilizables. Cada objeto encapsula datos y comportamientos relacionados, lo que facilita la comprensión y el mantenimiento del código.

•Abstracción: Los objetos nos permiten representar entidades del mundo real en el software y modelar su comportamiento de manera abstracta. Esto nos ayuda a comprender y trabajar con conceptos complejos de manera más simple y clara.

•Reutilización de código: La instanciación y el uso de objetos promueven la reutilización de código a través de la creación de clases y la herencia. Una vez que se ha definido una clase, se pueden crear múltiples objetos basados en ella, lo que evita la duplicación de código y

facilita la evolución y el mantenimiento del software.

•Flexibilidad: La instanciación y el uso de objetos permiten crear estructuras de datos dinámicas y flexibles. Los objetos pueden ser creados, modificados y eliminados durante la ejecución del programa, lo que brinda una mayor capacidad de adaptación y respuesta a los cambios en los requerimientos del software.

En resumen, la instanciación y el uso de objetos son conceptos fundamentales en la POO. La instanciación implica la creación de objetos específicos a partir de una clase, mientras que el uso de objetos implica acceder a sus atributos y métodos para realizar diferentes operaciones. Estos conceptos proporcionan modularidad, abstracción, reutilización de código y flexibilidad en el desarrollo de software orientado a objetos.

2.6. Aplicaciones en Java

2.6.1. Aplicación 1:

Se tiene la implementación de Polimorfismo por Sobrecarga de Métodos

Programa Principal

```
package app_polimorfismo2;
import javax.swing.JOptionPane;
/**
 *
 * @author Marcelo
 */
public class App_Polimorfismo2 {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // TODO code application logic here
      Programador prog1=new Programador();
      AdministradorBD adminbd=new AdministradorBD();
      Analista analista1=new Analista();
      prog1.datos("Juan Fernando","Cáceres Limaico");
      prog1.sueldo(40,12);
      prog1.labores("Programar en Java");
      /////
      String nombre1, labores1;
      float sueldo1;
      nombre1=JOptionPane.showInputDialog(null, "Ingrese los nombres del adminBD
",
            "Ingresando Datos", JOptionPane.INFORMATION_MESSAGE);
      sueldo1=Float.parseFloat(JOptionPane.showInputDialog(null,"Ingrese el sueldo",
            "Ingresando el sueldo",  JOptionPane.INFORMATION_MESSAGE));
     labores1=JOptionPane.showInputDialog(null, "Ingrese las funciones del adminBD
",
            "Ingresando Datos", JOptionPane.INFORMATION_MESSAGE);
      adminbd.datos(nombre1);
      adminbd.sueldo(sueldo1);
      adminbd.labores(labores1);

      ///////
      analista1.datos("Sandra Valentina", "Suarez Fernandez", "Luz de América");
      analista1.sueldo(20, 15, 35);
      analista1.labores("Analizar los Requerimientos de SW", 20);
   }

}
```


Este programa se apoya de las siguientes clases

Clase Padre: Informáticos

Clase Informáticos

```
package app_polimorfismo2;


/**

 *

 * @author Marcelo

 */

public class Informaticos {


   public Informaticos() {

   }


   public void sueldo(){


   }

   public void datos(){


   }

   public void labores(){


   }

}
```

Clases Hija: Clase AdministradorBD, Clase Analista, y Clase Programador

Clase AdministradorBD

```
package app_polimorfismo2;
import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class AdministradorBD extends Informaticos{
   public void sueldo(double pago){
      JOptionPane.showMessageDialog(null,"El pago es: "+pago);
   }
   public void datos(String nombres){
      JOptionPane.showMessageDialog(null, "Los nombres son: "+nombres);
   }
   public void labores(String labores){
        JOptionPane.showMessageDialog(null, "Soy Administrador BD y me dedico a:
"+labores);
   }
}
```

Clase Analista

```
package app_polimorfismo2;
import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class Analista extends Informaticos{
   public void sueldo(int horas, double pagoHoras, double bono){
     JOptionPane.showMessageDialog(null, "El pago es: "+((horas*pagoHoras)+bono));
   }
   public void datos(String nom, String ape, String direccion){
      JOptionPane.showMessageDialog(null, "Los nombres son: "+nom
      + "\n Los apellidos son: " + ape + "\n La Dirección es: "+direccion);
   }
   public void labores(String labores, int horas){
      JOptionPane.showMessageDialog(null, "Soy Analista y mis funciones son: "
      + labores + " y trabajo por " + horas + " horas.");
   }

}
```


Clase Programador

```
package app_polimorfismo2;
import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class Programador extends Informaticos{
   public void sueldo(int horas, double pagoHoras){
      JOptionPane.showMessageDialog(null,"El pago es: "+(horas*pagoHoras));
   }
   public void datos(String nombres, String apellidos){
      JOptionPane.showMessageDialog(null, "Los nombres son: "+nombres
      + "\n Los Apellidos son: "+apellidos);
   }
   public void labores(String labores){
      JOptionPane.showMessageDialog(null, "Soy Programador y mis funciones son: "
            + labores);
   }
}
```

2.6.2. Aplicación 2:

Se tiene la mismas clases de la aplicación 1, pero la diferencia, es que aquí se implementa el Polimorfismo mediante la sobreescritura de métodos

Programa Principal

```
package app_polimorfismo2;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class mainOverride {
   public static void main(String[] args) {
      ProgramadorOverride prog2=new ProgramadorOverride();
      AnalistaOverride ana2=new AnalistaOverride();
      AdministradorBDOverride admin2=new AdministradorBDOverride();
      JOptionPane.showMessageDialog(null, "Polimorfismo Override");
      prog2.datos("Pedro");
      prog2.sueldo(450);
      prog2.labores("Programar en Java");
      ////////////
      ana2.datos("Fernanda");
      ana2.sueldo(650);
      ana2.labores("Analizar Requerimientos SW");
      //////
      admin2.datos("Sandra");
      admin2.sueldo(500);
      admin2.labores("Administrar BD en SQL Server");
    }
}
```

Clase Padre Informáticos

Programa Informáticos Override

```
package app_polimorfismo2;

/**
 *
 * @author Marcelo
 */
public abstract class InformaticosOverride {
   abstract public void sueldo(double pago);

   abstract public void datos(String nom);

   abstract public void labores(String lab);

}
```

Clases Hijas Heredadas de la Clase Informáticos Override:

AdministradorBDOverride, AnalistaOverride, y Programador Override

Clase AdministradorBD Override

```
package app_polimorfismo2;
import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class AdministradorBDOverride extends InformaticosOverride{
   @Override
   public void sueldo(double pago){
      JOptionPane.showMessageDialog(null, "El pago del Admin BD es: "+pago);
   }
   @Override
   public void datos(String nombre){
      JOptionPane.showMessageDialog(null, "El nombre del Admin BD es: "+nombre);
   }
   @Override
   public void labores(String labor){
      JOptionPane.showMessageDialog(null, "Las labores del Admin BD es: "+labor);
   }
}
```

Clase Analista Override

```
package app_polimorfismo2;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class AnalistaOverride extends InformaticosOverride{
   @Override
   public void sueldo(double pago){
      JOptionPane.showMessageDialog(null,"El pago del ANALISTA es: "+pago);
   }

   @Override
   public void datos(String nombre){
      JOptionPane.showMessageDialog(null,"Los nombres del ANLISTA son: "+nombre);
   }

   @Override
   public void labores(String labor){
      JOptionPane.showMessageDialog(null,"La labor del ANALISTA es: "+labor);
   }
}
```

Clase Programador Override

```
package app_polimorfismo2;
import javax.swing.JOptionPane;
/**
 *
 * @author Marcelo
 */
public class ProgramadorOverride extends InformaticosOverride{
   @Override
   public void sueldo(double pago){
      JOptionPane.showMessageDialog(null,"El pago del PROGRAMADOR es: "+pago);
   }
   @Override
   public void datos(String nombre){
      JOptionPane.showMessageDialog(null,"Los nombres del PROGRAMADOR son: "+nombre);
   }
   @Override
   public void labores(String labor){
      JOptionPane.showMessageDialog(null,"Las funciones del PROGRAMADOR son: "+labor);
   }
}
```

2.7. Resumen del Capítulo

La programación orientada a objetos en Java se basa en conceptos clave como la encapsulación, la herencia, el polimorfismo y la abstracción. Estos conceptos proporcionan una forma estructurada y modular de diseñar y desarrollar software. La encapsulación protege los datos y oculta los detalles internos, mientras que la herencia permite la reutilización de código y la creación de jerarquías de clases. El polimorfismo permite que los objetos de diferentes clases respondan a las mismas operaciones, y la abstracción nos permite enfocarnos en los aspectos esenciales de los objetos. La instanciación

y el uso de objetos nos permiten trabajar con múltiples instancias de una clase en tiempo de ejecución. Al comprender y aplicar estos conceptos, los programadores pueden construir aplicaciones más eficientes, flexibles y mantenibles en Java.

CAPÍTULO III

PREPARACIÓN DEL IDE ENTORNO DE DESARROLLO INTEGRADO PARA JAVA

3.1. Introducción

En el mundo de la programación o desarrollo de software, se debe tener conciencia que un factor preponderante, corresponde a los aspectos relacionados con los tipos de datos o lo que se conoce como la Estructura de Datos, en este caso, el enfoque que se realizará va a ser el práctico, debido a que se puede tomar como una base en el caso de la utilización de los siguientes ejercicios prácticos.

Para el presente libro, se determina como lenguaje de programación Java, a través de cualquier entorno de Programación que nos permita su interpretación como Netbeans o Apache Netbeans.

3.2. Instalación JDK para Java

Lo primero que se requiere es la instalación del Entorno Java Development Kit (JDK), el cual consiste en un conjunto de herramientas para brindar el soporte necesario para el desarrollo y la ejecución de un entorno estable de la codificación en Java YYY, para esto se debe descargar la versión disponible, del entorno de desarrollo de programación Netbeans.

Para la instalación del JDK, se debe tener los requerimientos mínimos descritos en la tabla 1, para un sistema operativo de 64 bits (*Windows System*

Requirements for JDK and

Tabla 1

Requerimientos Recomendados Java Development Kit

Item	Requerimientos Recomendados
Sistema Operativo	Windows 7, Windows 8, Windows 10, Windows 11

JRE, n.d.)

Para la correcta instalación, lo primero que se debe realizar es dirigirse a la página de descarga https://www.oracle.com/java/technologies/downloads/, y seleccionar la correcta plataforma del sistema operativo, en este caso se selecciona, para un Sistema Operativo Windows de 64 bits, tal como se muestra en la Figura 1.

En la Figura 2., se puede determinar el inicio de la instalación del programa relacionado al Java Development Kit, para la versión del sistema operativo Windows de 64 bits, para ello se debe ir determinando las imágenes respectivas con las opciones a seleccionar.

Figura 2

Página de Descarga Java Development Kit

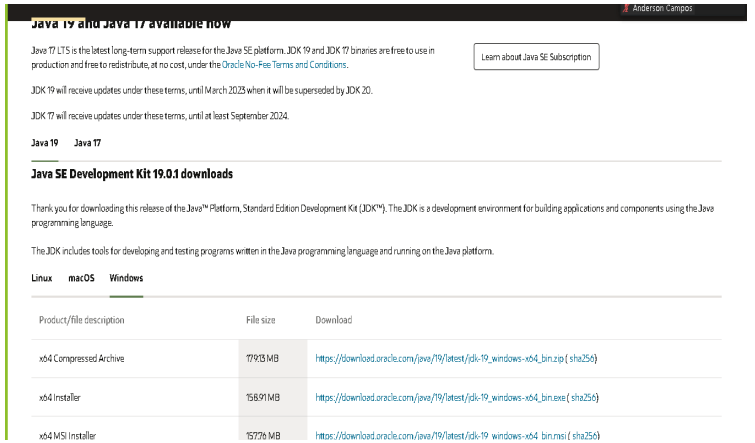


Figura 1

Página de Descarga Java Development Kit

3.3. Instalación del Entorno de Desarrollo Integrado Apache Netbeans

En cambio para la instalación del Entorno de

Desarrollo Netbeans (*Netbeans 8 (Windows) System Requirements, Minimum Requirements, Recommended*

Tabla 2

Requerimientos IDE Netbeans

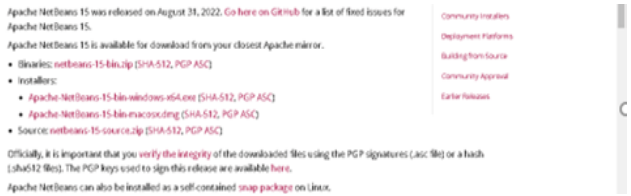
Item	Requerimientos Recomendados
Sistema Operativo	Windows 7, Windows 8, Windows 10, Windows 11
Memoria RAM disponible	512 MB

Requirements - PcRequirements.Net, n.d.), se requiere los requerimientos descritos en la Tabla 2.

En la Figura 3., se puede determinar las opciones principales de descarga, en la siguiente dirección url: https://netbeans.apache.org/download/index.html, especialmente para los sistemas operativos Windows y Linux, tanto de arquitectura

Figura 3

Página de Descarga Apache Netbeans

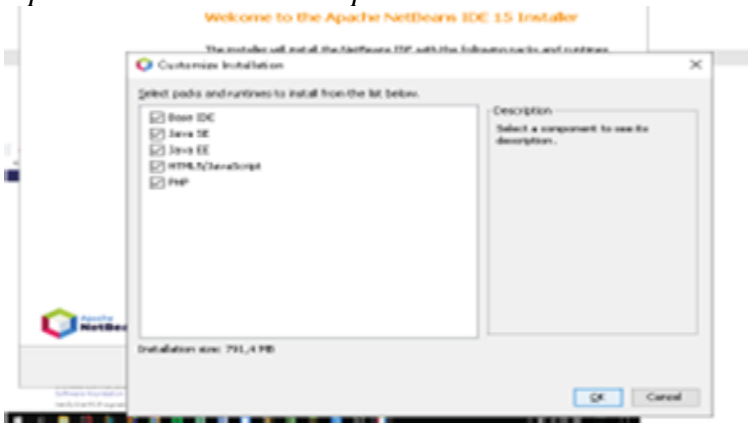


de 32 y 64 bits, para el desarrollo de los ejercicios, se va a utilizar el sistema operativo de 64 bits en la Versión de Windows 10.

Para seleccionar con el o los lenguajes de programación que se va a trabajar, se presentan las opciones en la Figura 4., relacionadas a los Lenguajes Java, HTML y Php, para la ejecución de los ejercicios se recomienda el lenguaje Java, debido al enfoque práctico que se está utilizando.

Figura 4

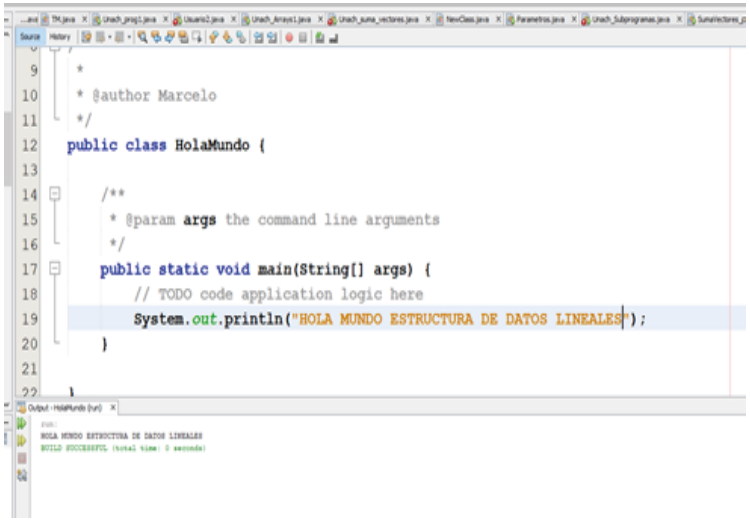
Opciones de Instalación Apache Netbeans



Para determinar si el Entorno de Desarrollo Netbeans, se encuentra funcionando de manera correcta, se debe realizar una prueba, para comprobar si se encuentra trabajando en conjunto a Java Development Kit, para ello se ejecuta el programa “Hola Mundo”, como se muestra en la figura

Figura 5

Ejecución Programa Hola Mundo



CAPÍTULO IV

LISTAS SIMPLES

En el campo de la programación y las estructuras de datos, las listas simples son una estructura fundamental y ampliamente utilizada. Una lista simple es una colección lineal de elementos donde cada elemento está conectado al siguiente mediante enlaces. En este tema, exploraremos en detalle las listas simples, su funcionamiento y sus aplicaciones en la programación.

4.1. Definición y propiedades de las listas simples

Las listas simples son una estructura de datos fundamental en el campo de la programación y las ciencias de la computación. También se conocen como listas enlazadas simples y son ampliamente utilizadas debido a sus propiedades y eficiencia en diversas aplicaciones. En este tema, exploraremos la definición y las propiedades clave de las listas simples, junto con su importancia en el ámbito de la programación.

4.1.1. Definición de las Listas Simples

Una lista simple es una estructura de datos en la que cada elemento, conocido como nodo, está compuesto por dos partes: un campo que almacena el valor o la información del nodo y un enlace o puntero que apunta al siguiente nodo en la secuencia. El primer nodo de la lista se llama nodo cabeza y el último nodo se conoce como nodo cola. La lista finaliza cuando el puntero del último nodo apunta a un valor nulo, indicando el final de la secuencia.

Según Cormen et al. (2009), “las listas enlazadas

simples son una forma de estructurar datos en la que cada elemento contiene un valor y un puntero que apunta al siguiente elemento de la secuencia" (p. 202). Esta definición resalta la característica fundamental de las listas simples, donde los nodos están interconectados mediante enlaces para formar una secuencia lineal.

4.1.2. Propiedades de las Listas Simples

Las listas simples presentan varias propiedades que las hacen útiles y versátiles en el ámbito de la programación. A continuación, se describen algunas de las propiedades más importantes:

•Inserción Eficiente: Una de las principales ventajas de las listas simples es la eficiencia en la inserción de nuevos elementos. En comparación con otras estructuras de datos, como los arreglos, las listas simples permiten la inserción eficiente de elementos en cualquier posición. Esto se debe a que solo es necesario modificar los enlaces de los nodos adyacentes, sin requerir cambios masivos en la estructura subyacente de la lista.

•Eliminación Eficiente: Al igual que la inserción, la eliminación de elementos en las listas simples es eficiente. Para eliminar un nodo de la lista, solo es necesario actualizar los enlaces de los nodos adyacentes para omitir el nodo a eliminar. Esto permite una eliminación rápida sin tener que reorganizar los demás elementos de la lista.

- Flexibilidad en el Tamaño: A diferencia de las

estructuras de datos estáticas, como los arreglos, las listas simples no tienen una capacidad fija y pueden crecer o reducirse dinámicamente según sea necesario. Esto brinda flexibilidad en el manejo de datos y permite una gestión eficiente de la memoria.

- Acceso Secuencial: Las listas simples permiten un acceso secuencial a los elementos. Dado que los nodos están enlazados secuencialmente, es posible recorrer la lista de manera ordenada, accediendo a cada elemento en su posición correspondiente. Sin embargo, el acceso aleatorio a elementos específicos es menos eficiente y requiere recorrer la lista desde el inicio.

- Fácil Implementación de Operaciones: Las listas simples ofrecen una base sólida para implementar diversas operaciones, como la búsqueda de un elemento, la inversión de la lista o la concatenación de listas. Estas operaciones se pueden realizar mediante la manipulación adecuada de los enlaces entre los nodos.

- Uso Eficiente de la Memoria: A diferencia de las estructuras de datos estáticas, las listas simples utilizan memoria de forma eficiente. Los nodos se pueden asignar dinámicamente en la memoria a medida que se necesiten, lo que permite una asignación flexible y un uso óptimo de los recursos disponibles.

4.1.3. Importancia de las Listas Simples en la Programación

Las listas simples desempeñan un papel fundamental

en la programación debido a su flexibilidad y eficiencia en el manejo de datos. Son ampliamente utilizadas en diversas aplicaciones, como:

• Implementación de Pilas y Colas: Las pilas y las colas son estructuras de datos esenciales en la programación. Las listas simples se utilizan para implementar estas estructuras, donde la cabeza de la lista representa el tope de la pila o el frente de la cola. La inserción y eliminación de elementos se realizan de manera eficiente al modificar los enlaces de los nodos adyacentes.

•Implementación de Iteradores: Los iteradores son utilizados para recorrer y acceder a los elementos de una estructura de datos de manera secuencial. Las listas simples proporcionan una base para implementar iteradores, permitiendo un acceso ordenado y controlado a los elementos de la lista.

•Almacenamiento de Datos enlazados: En aplicaciones donde los datos deben estar enlazados de manera secuencial, las listas simples son una opción conveniente. Por ejemplo, en editores de texto, se pueden utilizar listas simples para almacenar las líneas del texto, donde cada nodo contiene una línea y un enlace al siguiente nodo.

•Gestión de Memoria Dinámica: En algunos lenguajes de programación, como C o C++, las listas simples se utilizan para gestionar la memoria dinámica. Cada nodo de la lista representa un bloque de memoria asignado

dinámicamente, y la lista mantiene un registro de los bloques libres y ocupados.

• Implementación de Algoritmos de Ordenamiento: Algunos algoritmos de ordenamiento, como el ordenamiento por inserción (insertion sort) o el ordenamiento por selección (selection sort), se pueden implementar de manera eficiente utilizando listas simples. Los elementos se insertan y eliminan de la lista durante el proceso de ordenamiento, lo que permite una implementación más sencilla y comprensible.

En resumen, las listas simples son una estructura de datos versátil y eficiente en la programación. Su flexibilidad en la inserción, eliminación y gestión de datos las hace adecuadas para una amplia gama de aplicaciones. Además, su implementación y uso son fundamentales en la programación orientada a objetos, donde se utilizan como base para construir estructuras de datos más complejas. Entender las propiedades y características de las listas simples es esencial para cualquier programador que desee utilizar eficientemente estas estructuras en sus proyectos.

4.2. Implementación de listas simples en Java

Las listas simples, también conocidas como listas enlazadas simples, son una estructura de datos fundamental en la programación. En Java, podemos implementar estas listas utilizando clases y referencias. En este tema, exploraremos cómo implementar listas

simples en Java y cómo aprovechar sus funcionalidades en la manipulación de datos.

4.2.1. Definición de Listas Simples

Antes de adentrarnos en la implementación de listas simples en Java, es importante comprender su definición. Según Cormen et al. (2009), una lista simple es una estructura de datos en la que cada elemento, llamado nodo, contiene un valor y una referencia al siguiente nodo en la secuencia (p. 202). Esto significa que los nodos están enlazados entre sí, creando una secuencia lineal.

4.2.2. Implementación de Nodos

En Java, podemos implementar un nodo para una lista simple utilizando una clase. Cada nodo debe tener al menos dos atributos: un campo para almacenar el valor y una referencia al siguiente nodo. A continuación, se muestra un ejemplo de implementación de la clase Nodo:

Clase Nodo

```
public class Nodo {
    private int valor;
    private Nodo siguiente;

    public Nodo(int valor) {
        this.valor = valor;
        this.siguiente = null;
    }

    // Métodos getters y setters
    // ...
}
```


En este ejemplo, la clase Nodo tiene un atributo valor para almacenar el valor del nodo y un atributo siguiente que es una referencia al siguiente nodo en la secuencia. El constructor se encarga de inicializar los valores y establecer el siguiente nodo como null.

4.2.3. Implementación de la Lista Simple

Una vez que hemos definido la clase Nodo, podemos proceder a implementar la lista simple en sí. La lista simple constará de una serie de nodos enlazados, comenzando por un nodo cabeza. También incluirá métodos para insertar, eliminar y buscar elementos en la lista.

Clase Lista Simple

```
public class ListaSimple {
    private Nodo cabeza;

    public ListaSimple() {
        this.cabeza = null;
    }

    // Métodos de la lista
    // ...
}
```

En este ejemplo, la clase ListaSimple tiene un atributo cabeza que representa el primer nodo de la lista. Al igual que en el nodo, el constructor inicializa la cabeza como null, indicando una lista vacía.

4.2.4. Métodos de la Lista Simple

La implementación de los métodos en la lista simple nos permitirá realizar diversas operaciones, como insertar

elementos, eliminar elementos y buscar elementos en la lista. A continuación, se muestran algunos ejemplos de métodos comunes:

Clase Lista Simple (Métodos)

```
public void insertar(int valor) {
    Nodo nuevoNodo = new Nodo(valor);
    if (cabeza == null) {
        cabeza = nuevoNodo;
    } else {
        Nodo nodoActual = cabeza;
        while (nodoActual.siguiente != null) {
            nodoActual = nodoActual.siguiente;
        }
        nodoActual.siguiente = nuevoNodo;
    }
}

public void eliminar(int valor) {
    if (cabeza == null) {
        return;
    }
    if (cabeza.valor == valor) {
        cabeza = cabeza.siguiente;
    } else {
        Nodo nodoActual = cabeza;
        while (nodoActual.siguiente != null) {
            if (nodoActual.siguiente.valor == valor) {
                nodoActual.siguiente = nodoActual.siguiente.siguiente;
                return;
            }
            nodoActual = nodoActual.siguiente;
        }
    }
}

public boolean buscar(int valor) {
    Nodo nodoActual = cabeza;
    while (nodoActual != null) {
        if (nodoActual.valor == valor) {
            return true;
        }
        nodoActual = nodoActual.siguiente;
    }
    return false;
}
```


En el método insertar, se crea un nuevo nodo con el valor proporcionado y se busca el último nodo de la lista para enlazar el nuevo nodo al final. En el método eliminar, se busca el nodo con el valor especificado y se ajustan las referencias para eliminarlo de la lista. En el método buscar, se recorre la lista buscando un nodo con el valor dado.

4.2.5. Uso de la Lista Simple en Java

La implementación de la lista simple en Java nos brinda una herramienta versátil para almacenar y manipular datos. Podemos utilizarla en una variedad de aplicaciones, como:

•Registro de Datos Secuenciales: La lista simple es útil cuando necesitamos almacenar datos en una secuencia específica. Por ejemplo, podemos utilizarla para mantener un registro de eventos en orden cronológico.

•Implementación de Pilas: Las listas simples se pueden utilizar como base para implementar pilas. Al insertar y eliminar elementos desde la cabeza de la lista, logramos el comportamiento de una pila, donde el último elemento en entrar es el primero en salir (LIFO).

•Implementación de Colas: También podemos utilizar listas simples para implementar colas. Al insertar elementos al final de la lista y eliminarlos desde la cabeza, obtenemos el comportamiento de una cola, donde el primer elemento en entrar es el primero en salir (FIFO).

•Manipulación de Datos Dinámicos: La lista simple

nos permite manejar conjuntos de datos que pueden cambiar en tiempo de ejecución. Podemos agregar y eliminar elementos de manera eficiente sin tener que redimensionar un arreglo.

La implementación de listas simples en Java nos brinda una estructura de datos flexible y eficiente para el manejo de datos en secuencia. A través de la clase Nodo y la clase Lista Simple, podemos crear listas enlazadas y realizar operaciones como inserción, eliminación y búsqueda de elementos. Estas listas son útiles en una variedad de aplicaciones, como registros de datos, implementación de pilas y colas, y manipulación de datos dinámicos. Al dominar la implementación de listas simples, los programadores pueden aprovechar sus características para resolver problemas complejos de manera efectiva.

4.3. Ejemplos de aplicaciones de listas simples

4.3.1. Aplicación 1

A continuación se implementa las clases necesarias para un programa en Java con listas simples

Clase Nodo

```
package ufa_tda5_lista_simple;

/**
 *
 * @author Marcelo
 */
public class Nodo {
   public int dato;//INformación
   public Nodo siguiente;//Puntero

   //Constructor para insertar al final de lista un elemento
   public Nodo(int ndato){
      this.dato=ndato;
      this.siguiente=null;
   }

   //////COnstructor para insertar un nodo al inicio de la lista
   public Nodo(int d, Nodo n){
      dato=d;
      this.siguiente=n;
   }
}
```

Clase Lista Simple

```
package ufa_tda5_lista_simple;

/**
 *
 * @author Marcelo
 */
public class Lista {

   protected Nodo inicio, fin;//Punteros de mi lista, tanto del primer elemento como
el último

   //COnstructor
   public Lista() {
      inicio = null; //cabecera
      fin = null;  //cola
   }

Método para agrega un nodo INICIO de la lista
   public void agregarInicio(int elemento) {
      inicio = new Nodo(elemento, inicio);
      if (fin == null) { //cuando no tengo ningún elemento en mi lista
         fin = inicio;
      }
   }

   //Mostar los elementos de mi lista
   public void mostrarLista() {
     Nodo recorrer = inicio;
```

```
        System.out.println();
    while (recorrer != null) {
        System.out.print(" |" + recorrer.dato + "|--->");
        recorrer = recorrer.siguiente;
    }
    System.out.println("NULL");
}

//Método para verificar si la lista está VACIA
public boolean estaVacia() {
    if (inicio == null) {//lista vacía
        return true;
    } else {
        return false;
    }
}
 //Método para insertar un elemento al FINAL de la lista
//caso I: tengo elementos; caso II: no hay elementos en la lista
public void agregarFinal(int nelemento) {
    if (!estaVacia()) {
        //CASO I:tengo elementos
        this.fin.siguiente = new Nodo(nelemento);
        this.fin = fin.siguiente;
    } else {
        //CASO II:no hay elementos en la lista
        inicio = fin = new Nodo(nelemento);
```



```
        }
    }

    //METODO PARA ELIMINAR UN NODO AL INICIO DE LA LISTA
    //CASO I: NO TENGO ELEMENTOS, SOLO 1 ELEMENTO
    //CASO II: TENGO 2 O MAS ELEMENTOS
    public int borrarInicio() {
        int elemento;

        if (inicio == null) {
            inicio = fin = null;
            return -1;
        } else {
            //CASO I
            elemento = this.inicio.dato;
            if (inicio == fin) {
                inicio = null;
                fin = null;
                return 0;
            } else {
                //caso II
                this.inicio = this.inicio.siguiente;
            }
        }
        return elemento;
//mÉTODO PARA ELELIMINAR AL FINAL DE LA LISTA
    public int borrarFinal() {
        int elemento;
        if (inicio == null) {
            inicio = fin = null;
            return -1;
        } else {
            //CASO I
            elemento = this.inicio.dato;
            if (inicio == fin) {
```

```
 inicio = null;
              fin = null;
             return 0;
          } else {
             //caso II
             Nodo temporal=inicio;
             while (temporal.siguiente!=fin){
                temporal=temporal.siguiente;
             }
             fin=temporal;
             fin.siguiente=null;
          }
       }
       return elemento;
    }

}
```

Programa Principal

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package ufa_tda5_lista_simple;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
*/
```



```
public class UFA_TDA5_Lista_Simple {


   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // TODO code application logic here
      int opcion = 0, e1;
      Lista olista = new Lista();
      do {//menú principal
         try {
            opcion = Integer.parseInt(JOptionPane.showInputDialog(null,
                  "1. Agregar un nuevo elemento al INICIO de la lista"
                  + "\n 2. Agregar un nuevo elemento al FINAL de la lista "
                  + "\n 3. Mostrar los datos de la Lista"
                  + "\n 4. Eliminar un elemento del INICIO de la lista"
                  + "\n 5. Eliminar un elemento del FINAL de la lista"
                  + "\n 6. Salir"));
            switch (opcion) {
               case 1:
                  try {
                     e1 = Integer.parseInt(JOptionPane.showInputDialog(null, "Ingrese el
elemento",
                           "Insertando el elemento al INICIO de la lista", 3));
                     olista.agregarInicio(e1);
                  } catch (NumberFormatException n) {
                     JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
                  }
```

```
                    break;
                case 2:
                    try{
                        e1=Integer.parseInt(JOptionPane.showInputDialog(null,"Ingrese el
elemento",
                                "Insertando el elemento al FINAL de la lista", 3));
                        //Agregar el nodo
                        olista.agregarFinal(e1);
                    }catch(NumberFormatException n){
                        JOptionPane.showMessageDialog(null,"Error"+n.getMessage());
                    }
                    break;
                case 3:
                    olista.mostrarLista();
                    break;
                case 4:
                    try{
                    e1=olista.borrarInicio();
                        if (e1==-1){
                        System.out.println("LISTA VACIA");
                    }else{
                    JOptionPane.showMessageDialog(null,"El elemento eliminado es: "+
                            e1, "Eliminando Nodo del INICIO de la lista",
                            JOptionPane.INFORMATION_MESSAGE);
                    }
                    }catch(NumberFormatException n){
                        JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
                    }
                    break;
                case 5://ELMINAR FINAL
                    try{
                    e1=olista.borrarFinal();
```



```
                        if (e1==-1){
                        System.out.println("LISTA VACIA");
                    }else{
                    JOptionPane.showMessageDialog(null,"El elemento eliminado es: "+
                            e1, "Eliminando Nodo del FINAL de la lista",
                            JOptionPane.INFORMATION_MESSAGE);
                    }
                    }catch(NumberFormatException n){
                        JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
                    }
                    break;
                    case 6:
                    JOptionPane.showMessageDialog(null, " Programa FINALIZADO ");
                    break;
                default:
                    JOptionPane.showMessageDialog(null, " Opción Incorrecta ");
                }
            } catch (NumberFormatException e) {
                JOptionPane.showMessageDialog(null, "Error" + e.getMessage());
            }

        } while (opcion != 6);
    }

}
```

4.3.2. Aplicación 2

A continuación se implementa las clases necesarias para un programa en Java con listas simples de un Tipo de Dato Abstracto de Estudiante

Clase Estudiante

```
/**
 *
 * @author Marcelo
 */

public class Estudiante {
    private String nombre;
    private int edad;

    public Estudiante(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Métodos getters y setters
    // ...
}
```

La clase Estudiante representa a cada estudiante en la lista. Tiene dos atributos: nombre para almacenar el nombre del estudiante y edad para almacenar su edad. El constructor se utiliza para inicializar los valores de los atributos.

Clase Nodo

```
/**
 *
 * @author Marcelo
 */

public class Nodo {
    private Estudiante estudiante;
    private Nodo siguiente;

    public Nodo(Estudiante estudiante) {
        this.estudiante = estudiante;
        this.siguiente = null;
    }

    // Métodos getters y setters
    // ...
}
```


La clase Nodo representa cada nodo en la lista. Tiene un atributo estudiante que almacena un objeto de la clase Estudiante y un atributo siguiente que es una referencia al siguiente nodo en la secuencia. El constructor se utiliza para crear un nodo y establecer el siguiente nodo como null.

Clase Lista Simple

```
/**
 *
 * @author Marcelo
 */

public class ListaSimple {
    private Nodo cabeza;

    public ListaSimple() {
        this.cabeza = null;
    }

    public void insertar(Estudiante estudiante) {
        Nodo nuevoNodo = new Nodo(estudiante);
        if (cabeza == null) {
            cabeza = nuevoNodo;
        } else {
            Nodo nodoActual = cabeza;
            while (nodoActual.siguiente != null) {
                nodoActual = nodoActual.siguiente;
            }
            nodoActual.siguiente = nuevoNodo;
        }
    }

    public void eliminar(Estudiante estudiante) {
```

```
        if (cabeza == null) {
            return;
        }
        if (cabeza.estudiante.equals(estudiante)) {
            cabeza = cabeza.siguiente;
        } else {
            Nodo nodoActual = cabeza;
            while (nodoActual.siguiente != null) {
                if (nodoActual.siguiente.estudiante.equals(estudiante)) {
                    nodoActual.siguiente = nodoActual.siguiente.siguiente;
                    return;
                }
                nodoActual = nodoActual.siguiente;
            }
        }
    }

    public boolean buscar(Estudiante estudiante) {
        Nodo nodoActual = cabeza;
        while (nodoActual != null) {
            if (nodoActual.estudiante.equals(estudiante)) {
                return true;
}
            nodoActual = nodoActual.siguiente;
        }
        return false;
    }
}
```

La clase Lista Simple representa la lista simple en sí. Tiene un atributo cabeza que es un nodo que apunta al primer elemento de la lista. Los métodos insertar, eliminar y buscar permiten realizar las operaciones correspondientes en la lista.

En el método insertar, se crea un nuevo nodo con el estudiante proporcionado y se busca el último nodo de

la lista para enlazar el nuevo nodo al final. En el método eliminar, se busca el estudiante en la lista y se ajustan las referencias para eliminarlo de la lista. En el método buscar, se recorre la lista buscando un nodo con el estudiante dado.

Con estas clases, puedes crear una lista simple de estudiantes y realizar operaciones como insertar, eliminar y buscarestudiantes en la lista de manera eficiente.

Programa Principal

```
/**
 *
 * @author Marcelo
 */

public class Main {
    public static void main(String[] args) {
        ListaSimple listaEstudiantes = new ListaSimple();

        Estudiante estudiante1 = new Estudiante("John Smith", 20);
        Estudiante estudiante2 = new Estudiante("Jane Doe", 18);
        Estudiante estudiante3 = new Estudiante("David Johnson", 22);

        listaEstudiantes.insertar(estudiante1);
        listaEstudiantes.insertar(estudiante2);
        listaEstudiantes.insertar(estudiante3);

        System.out.println("Lista de estudiantes:");
        Nodo nodoActual = listaEstudiantes.getCabeza();
        while (nodoActual != null) {
            Estudiante estudiante = nodoActual.getEstudiante();
```

```
            System.out.println("Nombre: " + estudiante.getNombre() + ", Edad: " +
estudiante.getEdad());
            nodoActual = nodoActual.getSiguiente();
        }

        Estudiante estudianteBuscar = new Estudiante("Jane Doe", 18);
        boolean encontrado = listaEstudiantes.buscar(estudianteBuscar);
        System.out.println("¿Estudiante encontrado?: " + encontrado);

        Estudiante estudianteEliminar = new Estudiante("John Smith", 20);
        listaEstudiantes.eliminar(estudianteEliminar);

        System.out.println("Lista de estudiantes después de eliminar:");
        nodoActual = listaEstudiantes.getCabeza();
        while (nodoActual != null) {
            Estudiante estudiante = nodoActual.getEstudiante();
            System.out.println("Nombre: " + estudiante.getNombre() + ", Edad: " + estu-
diante.getEdad());
            nodoActual = nodoActual.getSiguiente();
        }
    }
}
```


CAPÍTULO V

PILAS EN JAVA

Las pilas son una estructura de datos fundamental en la programación que sigue el principio de "último en entrar, primero en salir" (Last-In, First-Out, LIFO). En Java, podemos implementar pilas utilizando las estructuras de datos disponibles en el lenguaje. En este tema, exploraremos la introducción a las pilas en Java, su funcionamiento y cómo implementarlas.

5.1. Definición y propiedades de las pilas

Las pilas son una estructura de datos fundamental en la programación que sigue el principio LIFO (Last-In, First-Out), lo que significa que el último elemento que se inserta en la pila es el primero en ser eliminado. En una pila, las operaciones de inserción y eliminación se realizan en un extremo llamado "tope" (top). Las pilas se utilizan para almacenar elementos en un orden específico y tienen propiedades y características únicas.

Según Cormen et al. (2009), una pila es una colección de elementos en la que las operaciones de inserción y eliminación se realizan en el tope (p. 236). La inserción de elementos se conoce como "push" y la eliminación se conoce como "pop". Al realizar un "pop", el elemento eliminado es siempre el último elemento que se insertó en la pila.

Las pilas tienen varias propiedades y características importantes:

- Principio LIFO: La propiedad fundamental de las pilas es que siguen el principio LIFO. Esto significa que

el último elemento que se inserta en la pila es el primero en ser eliminado. Los elementos se eliminan en orden inverso al que se insertaron.

• Acceso limitado: En una pila, solo se puede acceder al elemento en el tope de la pila. Los elementos que están por debajo del tope no son accesibles directamente. Para acceder a los elementos en posiciones inferiores de la pila, es necesario realizar operaciones de "pop" en los elementos superiores.

• Operaciones básicas: Las operaciones básicas en una pila son la inserción ("push") y la eliminación ("pop"). La operación de inserción agrega un elemento al tope de la pila, mientras que la operación de eliminación retira el elemento del tope de la pila.

• Propiedad de tamaño limitado: Las pilas pueden tener un tamaño limitado o ser implementadas con un tamaño dinámico. En el caso de una pila con tamaño limitado, una vez que se alcanza el límite de capacidad, no se pueden agregar más elementos.

• Eficiencia: Las operaciones de inserción y eliminación en una pila tienen una complejidad de tiempo constante, es decir, O(1). Esto las hace eficientes para aplicaciones que requieren una estructura de datos LIFO.

• Estructura de datos abstracta: La pila es una estructura de datos abstracta (EDA) que se puede implementar utilizando arreglos, listas enlazadas u otras

estructuras de datos. La EDA define las operaciones y propiedades básicas de la pila sin especificar la implementación subyacente.

Las pilas tienen diversas aplicaciones en programación y resolución de problemas. Algunas de las aplicaciones comunes de las pilas incluyen:

• Evaluación de expresiones: Las pilas se utilizan en la evaluación de expresiones matemáticas para determinar el orden de las operaciones. Al descomponer una expresión en componentes (operandos y operadores), se puede utilizar una pila para realizar la evaluación correctamente.

• Recursión: Las pilas son ampliamente utilizadas en la programación recursiva. Cuando se realiza una llamada recursiva, se almacena la información de la llamada actual en la pila. Esto permite que las llamadas recursivas se realicen de manera ordenada y se pueda regresar a los estados anteriores una vez que las llamadas recursivas se completen.

• Navegación en profundidad (Depth-First Search, DFS): El algoritmo DFS se utiliza para recorrer grafos. En este algoritmo, se utiliza una pila para almacenar los nodos visitados y explorar los vecinos de un nodo en profundidad antes de retroceder.

• Implementación de algoritmos: Las pilas se utilizan en numerosos algoritmos y estructuras de datos. Por

ejemplo, se utilizan en el algoritmo de inversión de palabras o frases, en la solución de problemas de paréntesis balanceados y en la implementación de otras estructuras de datos, como las colas y los árboles.

Las pilas son una estructura de datos fundamental en la programación que sigue el principio LIFO. Tienen propiedades y características únicas, como el acceso limitado al tope, operaciones básicas de inserción y eliminación, y eficiencia en tiempo constante. Las pilas se utilizan en diversas aplicaciones, como la evaluación de expresiones, la programación recursiva, la navegación en grafos y la implementación de algoritmos. Comprender la definición y propiedades de las pilas es esencial para utilizar esta estructura de datos de manera efectiva en el desarrollo de programas.

5.2. Implementación de pilas en Java

Las pilas son una estructura de datos fundamental en la programación que sigue el principio LIFO (Last-In, First-Out). En Java, podemos implementar pilas utilizando las clases proporcionadas por el lenguaje, como LinkedList o Stack. Estas clases ofrecen métodos y funcionalidades para trabajar con pilas de manera eficiente.

5.1.1. Implementación de pilas utilizando LinkedList

La clase LinkedList de Java proporciona métodos y operaciones que son útiles para implementar pilas. Para

crear una pila utilizando LinkedList, podemos utilizar los métodos push y pop. El método push se utiliza para insertar un elemento en el tope de la pila, mientras que el método pop se utiliza para eliminar y devolver el elemento en el tope de la pila.

Clase Pila utilizando la clase LinkedList

```
import java.util.LinkedList;

public class Pila {
    private LinkedList<Integer> pila;

    public Pila() {
        pila = new LinkedList<>();
    }

    public void push(int elemento) {
        pila.push(elemento);
    }

    public int pop() {
        return pila.pop();
    }

    public int peek() {
        return pila.peek();
    }

    public boolean isEmpty() {
        return pila.isEmpty();
    }

    public int size() {
        return pila.size();
    }
}
```

En este ejemplo, creamos una clase Pila que utiliza

la clase LinkedList de Java para implementar una pila de enteros. La clase LinkedList proporciona métodos como push, pop, peek, isEmpty y size que nos permiten realizar operaciones comunes en pilas.

5.2.1. Implementación de pilas utilizando Stack

Otra opción para implementar pilas en Java es utilizando la clase Stack del paquete java.util. La clase Stack extiende la clase Vector y proporciona métodos y operaciones específicas para trabajar con pilas.

Clase Pila utilizando la clase Stack

```
import java.util.Stack;

public class Pila {
    private Stack<Integer> pila;

    public Pila() {
        pila = new Stack<>();
    }

    public void push(int elemento) {
        pila.push(elemento);
    }

    public int pop() {
        return pila.pop();
    }

    public int peek() {
        return pila.peek();
    }

    public boolean isEmpty() {
        return pila.isEmpty();
    }

    public int size() {
        return pila.size();
    }
}
```

En este ejemplo, creamos una clase Pila que utiliza la clase Stack de Java para implementar una pila de enteros. La clase Stack proporciona métodos como push,

pop, peek, isEmpty y size que nos permiten realizar operaciones comunes en pilas.

5.1.3.Comparación entre LinkedList y Stack

Tanto LinkedList como Stack pueden ser utilizados para implementar pilas en Java. Sin embargo, hay algunas diferencias importantes a tener en cuenta. Mientras que LinkedList es una clase más general que puede ser utilizada como una lista enlazada, Stack se especializa específicamente en la implementación de pilas.

La clase LinkedList de Java es una listaenlazada doblemente en la que se puede acceder tanto al inicio como al final de la lista. Esto la convierte en una opción más versátil que puede ser utilizada para implementar otras estructuras de datos además de pilas. Sin embargo, debido a su mayor flexibilidad, LinkedList puede tener un rendimiento ligeramente inferior en comparación con Stack en aplicaciones que requieren un enfoque específico en la implementación de pilas.

Por otro lado, la clase Stack es una implementación más específica para pilas y hereda las propiedades de la clase Vector. Esto hace que Stack sea más eficiente en términos de rendimiento en aplicaciones que se centran en la implementación de pilas. Sin embargo, es importante tener en cuenta que la clase Stack también hereda algunas de las limitaciones del Vector, como su capacidad fija y la sincronización en operaciones de

subprocesos, lo que puede afectar el rendimiento en ciertos casos.

En general, tanto LinkedList como Stack pueden ser utilizados para implementar pilas en Java, y la elección entre ellos depende de los requisitos específicos de tu aplicación. Si necesitas una estructura de datos más versátil que pueda adaptarse a diferentes escenarios, LinkedList puede ser la opción adecuada. Por otro lado, si te centras en la implementación eficiente de pilas y no necesitas las funcionalidades adicionales de LinkedList, Stack puede ser una elección más adecuada.

La implementación de pilas en Java es esencial para el desarrollo de aplicaciones que requieren un enfoque LIFO en el manejo de datos. Tanto LinkedList como Stack son opciones viables para implementar pilas en Java, y cada una tiene sus propias características y ventajas. Comprender cómo utilizar estas clases y sus métodos asociados es fundamental para trabajar de manera eficiente con pilas en Java.

Al implementar pilas en Java, es importante considerar los requisitos específicos de tu aplicación y seleccionar la clase que mejor se adapte a tus necesidades. Tanto LinkedList como Stack proporcionan métodos y funcionalidades para realizar operaciones comunes en pilas, como la inserción, eliminación y acceso al elemento en el tope de la pila.

5.3. Operaciones comunes en pilas: push, pop y peek

Las pilas son una estructura de datos fundamental en la programación que sigue el principio LIFO (Last-In, First-Out). En Java, las pilas se pueden implementar utilizando las clases LinkedList o Stack. Estas clases proporcionan métodos y operaciones para realizar las operaciones comunes en pilas, como “push”, “pop” y “peek”.

5.3.1. Operación “push”

La operación “push” se utiliza para insertar un elemento en el tope de la pila. En Java, tanto LinkedList como Stack proporcionan un método con el mismo nombre para realizar esta operación.

Clase Pila Operación push

```
import java.util.LinkedList;
import java.util.Stack;

public class Pila {
    private LinkedList<Integer> pila1;
    private Stack<Integer> pila2;

    public Pila() {
        pila1 = new LinkedList<>();
        pila2 = new Stack<>();
    }

    public void push(int elemento) {
        pila1.push(elemento);
        pila2.push(elemento);
    }
}
```

En este ejemplo, creamos una clase Pila que utiliza tanto LinkedList como Stack para implementar una pila de enteros. El método push se utiliza para insertar un elemento en ambas pilas.

5.3.2. Operación “pop”

La operación “pop” se utiliza para eliminar y devolver el elemento en el tope de la pila. En Java, tanto LinkedList como Stack proporcionan un método con el mismo nombre para realizar esta operación.

Clase Pila Operación Pop

```
import java.util.LinkedList;
import java.util.Stack;

public class Pila {
    private LinkedList<Integer> pila1;
    private Stack<Integer> pila2;

    public Pila() {
        pila1 = new LinkedList<>();
        pila2 = new Stack<>();
    }

    public int pop() {
        pila1.pop();
        return pila2.pop();
    }
}
```

En este ejemplo, agregamos el método pop a la clase Pila para eliminar y devolver el elemento en el tope de ambas pilas.

5.3.3. Operación “peek”

La operación “peek” se utiliza para acceder al elemento en el tope de la pila sin eliminarlo. En Java, tanto LinkedList como Stack proporcionan un método con el mismo nombre para realizar esta operación.

Clase Lista Operación Peek

```
import java.util.LinkedList;
import java.util.Stack;

public class Pila {
    private LinkedList<Integer> pila1;
    private Stack<Integer> pila2;

    public Pila() {
        pila1 = new LinkedList<>();
        pila2 = new Stack<>();
    }

    public int peek() {
        return pila1.peek();
    }
}
```

En este ejemplo, implementamos el método peek en la clase Pila utilizando LinkedList para acceder al elemento en el tope de la pila sin eliminarlo.

Estas operaciones comunes en pilas son esenciales para administrar y manipular los elementos almacenados en la pila. La operación “push” se utiliza para agregar elementos al tope de la pila, la operación “pop” se utiliza paraeliminar y devolver el elemento en el tope de la pila, y la operación “peek” se utiliza para acceder al elemento en el tope sin eliminarlo. Estas operaciones permiten gestionar los elementos de la pila de acuerdo con el principio LIFO.

Es importante tener en cuenta que las implementaciones de pilas en Java, como LinkedList y Stack, ofrecen métodos adicionales y funcionalidades para trabajar con pilas. Por ejemplo, LinkedList proporciona métodos para obtener el tamaño de la pila (size) y verificar si está vacía (isEmpty), mientras que

Stack hereda estos métodos de la clase Vector. Estas funcionalidades adicionales pueden ser útiles para realizar operaciones y validaciones específicas en las pilas.

Las operaciones comunes en pilas, como “push”, “pop” y “peek”, son esenciales para trabajar con pilas en Java. Estas operaciones nos permiten insertar, eliminar y acceder a los elementos en el tope de la pila de manera eficiente. Tanto LinkedList como Stack son opciones válidas para implementar pilas en Java, y la elección entre ellas depende de los requisitos específicos de tu aplicación.

5.4. Ejemplos de aplicaciones de pilas

5.4.1. Aplicación 1: Pila Estática

Se presenta a continuación las clases necesarias en Java para determinar un programa que no puede ser dinamizada en tiempo de ejecución

Clase Pila Estatica

```
package ufa_tda6_pila_estatica;

/**
 *
 * @author Marcelo
 */
public class Pila {
    int vectorPila[];
    int cima;

    //Constructor
    public Pila(int tamanio){
        vectorPila=new int[tamanio];
        cima=-1;
    }

    //Push
    public void push(int dato){
        cima++;
        vectorPila[cima]=dato;
    }

    //Pop
    public int pop(){
        int sale=vectorPila[cima];
        cima--;
        return sale;
    }

    //Metodo para saber el elementos de la cima
    public int cimaPila(){
        return vectorPila[cima];
    }

    //Método para saber el tamaño
    public int tamanioPila(){
        return vectorPila.length;
    }

    //Método para verificar si la PILA está Vacía
    public boolean estaVacia(){
        return cima == -1;
    }

    //Metodo para verficar si la PILA está Llena
    public boolean estaLlena(){
        return vectorPila.length-1 == cima; //verdadero , esta llena
    }


}
```

Programa Principal

```
package ufa_tda6_pila_estatica;


import javax.swing.JOptionPane;
```

```
*/
public class UFA_TDA6_Pila_Estatica {

   /**
    * @param args the command line arguments
    */
   public static void main(String[] args) {
      // TODO code application logic here
      int opcion = 0, elemento, tamanio;

      try {
         //tamaño de la pila por parte del usuario
         tamanio =
Integer.parseInt(JOptionPane.showInputDialog(null,
               "Cuál es el tamaño que desea la PILA?: ", "Solicitando el tamaño",
               JOptionPane.INFORMATION_MESSAGE));
         Pila opila = new Pila(tamanio);

         do {
            //menu
            opcion = Integer.parseInt(JOptionPane.showInputDialog(null,
                  "1. Push de la Pila(1 elemento) \n"
                  + "2. Pop de la Pila (1 elemento)\n"
                  + "3. ¿La Pila está Vacía?\n"
                  + "4. ¿La Pila está Llena?\n"
                  + "5. Cúal es el elemento de la CIMA \n"
                  + "6. Cuál es el tamaño de la Pila \n"
                  + "7. Salir \n"
                  + "Qué opción desea????", "MENU DE OPCIONES",
                  JOptionPane.INFORMATION_MESSAGE));
            //switch
            switch (opcion) {
               case 1:
                  el elemento de la CIMA \n"  elemento =
Integer.parseInt(JOptionPane.showInputDialog(null,
                                                "Ingrese el elemento para ejecutar
```



```
el push", "Apilando DATO",
                        JOptionPane.INFORMATION_MESSAGE));
                  if (!opila.estaLlena()){
                     opila.push(elemento);
                  }else{
                     JOptionPane.showMessageDialog(null, "La Pila está Llena",
                           "Pila Llena...", JOptionPane.INFORMATION_MESSAGE);
                  }
                     break;
               case 2:
                  if (!opila.estaVacia()){
                     JOptionPane.showMessageDialog(null,"El elemento obtenido es: "+
                           opila.pop(), "Obteniendo el dato de la Pila",
JOptionPane.INFORMATION_MESSAGE);
                  }else{
                     JOptionPane.showMessageDialog(null, "La Pila está Vacía",
                           "Pila Vacía", JOptionPane.INFORMATION_MESSAGE);
                  }
                  break;
               case 3:
                  if (opila.estaVacia()){
                     JOptionPane.showMessageDialog(null, "La Pila está Vacía", "Pila
Vacía",
                           JOptionPane.INFORMATION_MESSAGE);
                  }else{
                     JOptionPane.showMessageDialog(null, "La Pila NO está Vacía ",
"Pila COntiene datos",
                           JOptionPane.INFORMATION_MESSAGE);
                  }
                  break;
               case 4:
                  if (opila.estaLlena()){
                     JOptionPane.showMessageDialog(null, "La Pila está Llena", "Pila
Llena",
 JOptionPane.INFORMATION_MESSAGE);
                                          }else{
```

```
                                JOptionPane.showMessageDialog(null, "La
Pila NO está Llena",
                        "Pila contiene espacio todavía",
                        JOptionPane.INFORMATION_MESSAGE);
                }
                break;
            case 5:
                if (!opila.estaVacia()){
                    JOptionPane.showMessageDialog(null, "El elemento de la cima es: "
                        + opila.cimaPila(), "Cima de la Pila",
                        JOptionPane.INFORMATION_MESSAGE);
                }else{
                    JOptionPane.showMessageDialog(null,"La Pila está vacía",
                        "Pila Vacía", JOptionPane.INFORMATION_MESSAGE);
                }
                break;
            case 6:
                JOptionPane.showMessageDialog(null, "El tamaño de la pila es: "
                    +opila.tamanioPila(), "Tamaño de la Pila",
                    JOptionPane.INFORMATION_MESSAGE);
                break;
            case 7:
                JOptionPane.showMessageDialog(null, "Programa Finalizado","Fin",
                    JOptionPane.INFORMATION_MESSAGE);
                break;
            default:
                JOptionPane.showMessageDialog(null, "Opción Incorrecta",
                    "Error", JOptionPane.INFORMATION_MESSAGE);
        }

        } while (opcion != 7);

    } catch (NumberFormatException n) {
        JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
    }
  }

}
```


5.3.5. Aplicación 2: Pila Dinámica

Se presenta a continuación las clases necesarias en Java para determinar un programa dinamizada en tiempo de ejecución

Clase Nodo Pila

```
package ufa_tda7_pila_dinamica;

/**
 *
 * @author Marcelo
 */
public class NodoPila {
    int dato;
    NodoPila siguiente;

    public NodoPila(int ndato){
        this.dato=ndato;
        siguiente=null;
    }

}
```

Clase Pila Dinámica

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package ufa_tda7_pila_dinamica;

/**
 *
 * @author Marcelo
 */
public class Pila {
    private NodoPila cima;
    int tamanio;

    public Pila(){
        this.cima=null;
        this.tamanio=0;
    }

    //Método para determinar si la pila esta vacía
    public boolean estaVacia(){
        return cima==null;
    }

    //Método PUSH
    public void push(int nelemento){
        NodoPila nuevo=new NodoPila(nelemento);
        nuevo.siguiente=cima;
        cima=nuevo;
        tamanio++;
    }

    //Método POP
    public int pop(){
        int aux=cima.dato;
        cima=cima.siguiente;
        tamanio--;
        return aux;
    }

    //Método para saber el elemento de la CIMA
    public int cima(){
        return cima.dato;
    }

    //Método para saber el tamaño de la PILA
    public int tamanioPila(){
        return this.tamanio;
    }

    //Método para limpiar la PILA(vaciar)
    public void limpiarPila(){
        while(!estaVacia()){
            pop();
        }
    }
}
```


Programa Principal

```
package ufa_tda7_pila_dinamica;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class UFA_TDA7_Pila_Dinamica {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int opcion = 0, elemento = 0;
        Pila opila = new Pila();

        do {
            try {
                opcion = Integer.parseInt(JOptionPane.showInputDialog(null,
                        "1. Empujar un elemento de la Pila \n"
                        + "2. Sacar un elemento de la Pila \n"
                        + "3. La Pila está vacía? \n"
                        + "4. Cuál es el elemento CIMA de la Pila? \n"
                        + "5. Cuál es el tamaño de la Pila? \n"
                        + "6. Desea vaciar la Pila? \n"
                        + "7. Salir\n"
                        + "Qué desea realizar?", "Menú de Opciones",
                        JOptionPane.INFORMATION_MESSAGE));

                switch (opcion) {
                    case 1:
                        elemento=Integer.parseInt(JOptionPane.showInputDialog(null,
                                "Ingrese el elemento para APILARLO", "Apilando Datos",
                                JOptionPane.INFORMATION_MESSAGE));
                        opila.push(elemento);
```

```
                break;
            case 2:
                if (!opila.estaVacia()){
                    JOptionPane.showMessageDialog(null, "El elemento que va a salir
de la Pila es: "
                            +opila.pop(), "Obteniendo el elemento de la Pila",
                            JOptionPane.INFORMATION_MESSAGE);
                }else{
                    JOptionPane.showMessageDialog(null, "La Pila Vacía ",
                            "Pila Vacía", JOptionPane.INFORMATION_MESSAGE);
                }
                break;
            case 3:
                if (opila.estaVacia()){
                    JOptionPane.showMessageDialog(null, "La Pila Vacía ",
                            "Pila Vacía", JOptionPane.INFORMATION_MESSAGE);
                }else{
                    JOptionPane.showMessageDialog(null, "La Pila NO está Vacía ",
                            "Pila Contiene Datos", JOptionPane.INFORMATION_MESSA-
GE);
                }
                break;
            case 4:
                if (!opila.estaVacia()){
                    JOptionPane.showMessageDialog(null, "El elemento que se encuen-
tra en la cima es: "
                    +opila.cima(), "Cima de la Pila", JOptionPane.INFORMATION_MES-
SAGE);
                }else{
                    JOptionPane.showMessageDialog(null, "La Pila está Vacía ",
                            "Pila Vacía", JOptionPane.INFORMATION_MESSAGE);
                }
                break;
            case 5:

                JOptionPane.showMessageDialog(null, "El tamaño de la PILA es: "
```



```
                        +opila.tamanioPila(), "Tamaño de la Pila",
                        JOptionPane.INFORMATION_MESSAGE);
                break;
            case 6:
                if (!opila.estaVacia()){
                    opila.limpiarPila();
                    JOptionPane.showMessageDialog(null, "La PILA se ha vaciado",
                            "Vaciando Pila", JOptionPane.INFORMATION_MESSAGE);
                }else{
                    JOptionPane.showMessageDialog(null, "La Pila está vacía, "
                            + "No hay elementos para vaciarlos", "Pila Vacía ",
                                JOptionPane.INFORMATION_MESSAGE );
                }
                break;
            case 7:
                JOptionPane.showMessageDialog(null, "Programa Finalizado", "Fin",
                        JOptionPane.INFORMATION_MESSAGE);
                break;
            default:
                JOptionPane.showMessageDialog(null, "Opción Incorrecta",
                        "Error", JOptionPane.INFORMATION_MESSAGE);
            }

        } catch (NumberFormatException n) {
            JOptionPane.showMessageDialog(null, "Error" + n.getMessage());
        }

    } while (opcion != 7);
  }

}
```

CAPÍTULO VI

COLAS EN JAVA

Las colas son una estructura de datos lineal que sigue el principio FIFO (First-In, First-Out), lo que significa que el primer elemento en ingresar es el primero en salir. En Java, las colas se pueden implementar utilizando la interfaz Queue y sus clases concretas, como LinkedList o ArrayDeque. Estas clases proporcionan métodos para realizar operaciones comunes en colas, como “enqueue” (encolar) y “dequeue” (desencolar).

6.1. Definición y propiedades de las colas

Según Cormen et al. (2009), una cola es una estructura de datos que admite dos operaciones fundamentales: “enqueue”, que agrega un elemento al final de la cola, y “dequeue”, que elimina y devuelve el elemento del frente de la cola. Además, las colas suelen tener una operación adicional llamada “peek”, que devuelve el elemento del frente de la cola sin eliminarlo.

Las colas en Java tienen varias propiedades que las hacen útiles en la programación. En primer lugar, garantizan el ordenamiento adecuado de los elementos según el principio FIFO. Esto es especialmente útil en situaciones donde el orden de procesamiento de los elementos es crítico, como en simulaciones o algoritmos de planificación. La propiedad FIFO asegura que los elementos se procesen en el mismo orden en que fueron agregados a la cola.

En segundo lugar, las implementaciones de colas en Java ofrecen métodos eficientes para agregar y

eliminar elementos. Por ejemplo, la operación “enqueue” (encolar) tiene una complejidad temporal constante O(1) en la mayoría de las implementaciones de colas, lo que significa que el tiempo requerido para agregar un elemento no depende del tamaño de la cola. De manera similar, la operación “dequeue” (desencolar) también tiene una complejidad temporal constante O(1) en la mayoría de los casos.

Otra propiedad importante de las colas en Java es su capacidad para almacenar elementos de cualquier tipo de objeto. Esto se logra mediante el uso de genéricos en la interfaz Queue y sus implementaciones concretas. Por ejemplo, se puede crear una cola de enteros utilizando Queue<Integer>, una cola de cadenas de texto utilizando Queue<String>, o incluso una cola de objetos personalizados utilizando Queue<MiObjeto>.

Además de las operaciones básicas de encolar (enqueue) y desencolar (dequeue), las implementaciones de colas en Java también proporcionan otras operaciones útiles. Por ejemplo, la operación “peek” permite acceder al elemento del frente de la cola sin eliminarlo. Esto es útil cuando se desea consultar el próximo elemento a ser procesado sin alterar el estado de la cola.

En la programación, las colas se utilizan en una variedad de escenarios y aplicaciones. Por ejemplo, son ampliamente utilizadas en algoritmos de búsqueda en amplitud (BFS, por sus siglas en inglés) para recorrer

árboles o grafos. También se utilizan en sistemas de procesamiento de tareas, donde los elementos se encolan para ser procesados por diferentes

6.2. Implementación de colas en Java

En la programación, las colas son una estructura de datos esencial que sigue el principio FIFO (First-In, First-Out), lo que significa que el primer elemento en ingresar es el primero en salir. En Java, las colas se pueden implementar utilizando la interfaz Queue y sus clases concretas, como LinkedList y ArrayDeque.

Según Cormen et al. (2009), una cola es una estructura de datos que admite dos operaciones fundamentales: “enqueue”, que agrega un elemento al final de la cola, y “dequeue”, que elimina y devuelve el elemento del frente de la cola. Además, las colas suelen tener una operación adicional llamada “peek”, que devuelve el elemento del frente de la cola sin eliminarlo.

En Java, la interfaz Queue define los métodos básicos para trabajar con colas, incluyendo add, remove y element. La clase LinkedList es una implementación común de la interfaz Queue y proporciona métodos adicionales para realizar operaciones en colas.

La implementación de colas más simple en Java es utilizando la clase LinkedList. Por ejemplo, podemos crear una cola de cadenas de texto de la siguiente manera:

Clase Cola utilizando LinkedList

```
import java.util.LinkedList;
import java.util.Queue;

public class ImplementacionCola {
    public static void main(String[] args) {
        Queue<String> cola = new LinkedList<>();

        // Agregar elementos a la cola
        cola.add("Elemento 1");
        cola.add("Elemento 2");
        cola.add("Elemento 3");

        // Eliminar y mostrar el elemento del frente de la cola
        System.out.println(cola.remove()); // Resultado: Elemento 1

        // Mostrar el elemento del frente de la cola sin eliminarlo
        System.out.println(cola.element()); // Resultado: Elemento 2
    }
}
```

En este ejemplo, creamos una cola utilizando la clase LinkedList y la interfaz Queue. Luego, utilizamos el método add para encolar elementos en la cola y el método remove para desencolar y devolver el elemento del frente de la cola. También utilizamos el método element para acceder al elemento del frente sin eliminarlo.

Otra implementación de colas en Java es utilizando la clase ArrayDeque, que proporciona una cola de doble final (double-ended queue). Esto significa que se pueden agregar y eliminar elementos tanto al frente como al final de la cola. La clase ArrayDeque implementa la interfaz Queue y ofrece métodos como offer, poll y peek

para realizar operaciones en la cola.

Clase Cola utilizando ArrayDeque

```
import java.util.ArrayDeque;
import java.util.Queue;

public class ImplementacionCola {
    public static void main(String[] args) {
        Queue<String> cola = new ArrayDeque<>();

        // Agregar elementos al frente de la cola
        cola.offer("Elemento 1");
        cola.offer("Elemento 2");

        // Agregar elementos al final de la cola
        cola.offer("Elemento 3");

        // Eliminar y mostrar el elemento del frente de la cola
        System.out.println(cola.poll()); // Resultado:Elemento 1

        // Mostrar el elemento del frente de la cola sin eliminarlo
        System.out.println(cola.peek()); // Resultado: Elemento 2
    }
}
```

En este ejemplo, utilizamos la clase ArrayDeque para implementar una cola. Utilizamos el método offer para agregar elementos tanto al frente como al final de la cola, y el método poll para eliminar y devolver el elemento del frente. El método peek se utiliza para acceder al elemento del frente sin eliminarlo.

Es importante destacar que tanto la implementación de LinkedList como la de ArrayDeque ofrecen eficiencia en términos de tiempo de ejecución y consumo de memoria para operaciones de encolar, desencolar y acceder a los elementos de la cola. La elección de la implementación adecuada dependerá de los requisitos específicos de tu aplicación.

Las colas son una estructura de datos importante en programación que sigue el principio FIFO. En Java, se pueden implementar utilizando la interfaz Queue y las clases LinkedList y ArrayDeque. Estas implementaciones

ofrecen métodos eficientes para realizar operaciones de encolar, desencolar y acceder a los elementos de la cola. La elección de la implementación adecuada dependerá de las necesidades específicas de tu aplicación.

6.3. Operaciones comunes en colas: enqueue, dequeue y peek

Las colas son una estructura de datos esencial en programación que sigue el principio FIFO (First-In, First-Out). En Java, existen diferentes formas de implementar colas y realizar operaciones comunes como "enqueue", "dequeue" y "peek". Estas operaciones permiten agregar elementos a la cola, eliminar elementos del frente de la cola y acceder al elemento del frente sin eliminarlo. A continuación, exploraremos cómo realizar estas operaciones utilizando las clases y métodos proporcionados por Java.

Para comenzar, es importante comprender que en Java, la interfaz Queue y sus implementaciones, como LinkedList y ArrayDeque, son comúnmente utilizadas para representar colas. Estas implementaciones ofrecen métodos eficientes para realizar operaciones comunes en colas.

La operación de "enqueue" consiste en agregar un elemento al final de la cola. En Java, podemos realizar esta operación utilizando el método offer de la interfaz Queue. Por ejemplo:

Clase Colo Operación Queue

```
Queue<Integer> cola = new LinkedList<>();

cola.offer(10);

cola.offer(20);
```

En este ejemplo, hemos creado una cola utilizando la implementación LinkedList. Luego, utilizamos el método

offer para agregar dos elementos, 10 y 20, a la cola. Los elementos se agregan al final de la cola en el orden en que se insertan.

La operación de "dequeue" implica eliminar y devolver el elemento del frente de la cola. En Java, podemos realizar esta operación utilizando el método poll de la interfaz Queue. Por ejemplo:

Clase Cola Método Poll

```
int elementoEliminado = cola.poll();

System.out.println("Elemento eliminado: " + elementoEliminado);
```

En este caso, utilizamos el método poll para eliminar y obtener el elemento del frente de la cola. El elemento eliminado se asigna a la variable elementoEliminado y se imprime en la consola. Es importante tener en cuenta que si la cola está vacía, el método poll devolverá null.

Finalmente, la operación de "peek" permite acceder al elemento del frente de la cola sin eliminarlo. En Java, podemos realizar esta operación utilizando el método peek de la interfaz Queue. Por ejemplo:

Clase Cola Operación Peek

```
int elementoFrente = cola.peek();

System.out.println("Elemento del frente: " + elementoFrente);
```

En este ejemplo, utilizamos el método peek para obtener el elemento del frente de la cola sin eliminarlo. El elemento del frente se asigna a la variable elementoFrente y se muestra en la consola.

Es importante tener en cuenta que todas estas operaciones, "enqueue", "dequeue" y "peek", se pueden realizar en tiempo constante (O(1)) en la implementación LinkedList y ArrayDeque, lo que las hace eficientes para manejar colas en Java.

Java proporciona clases e interfaces útiles para trabajar con colas y realizar operaciones comunes como “enqueue”, “dequeue” y “peek”. La interfaz Queue y sus implementaciones, como LinkedListy ArrayDeque, ofrecen métodos eficientes para agregar, eliminar y acceder a elementos en colas en Java.

Al utilizar el método offer, podemos agregar elementos al final de la cola de manera segura. Por ejemplo, según Duggal (2020), el método offer en la clase LinkedList devuelve true si el elemento se agrega correctamente y false si no se puede agregar debido a restricciones de capacidad.

Por otro lado, para eliminar y obtener el elemento del frente de la cola, utilizamos el método poll. Según Bloch (2018), el método poll en la clase LinkedList devuelve el elemento del frente de la cola o null si la cola está vacía.

Además, para acceder al elemento del frente sin eliminarlo, podemos utilizar el método peek. Según Oracle (2021), el método peek en la clase LinkedList devuelve el elemento del frente de la cola o null si la cola está vacía.

Es importante mencionar que la elección de la implementación de la cola depende de los requisitos específicos de tu programa. Si necesitas una cola con funcionalidad adicional, como la capacidad de redimensionamiento automático, la clase ArrayDeque puede ser una opción más adecuada. Por otro lado, si necesitas una implementación más simple y no te preocupas por el rendimiento en escenarios de tamaño variable, la clase LinkedList puede ser suficiente.

Las operaciones comunes en colas, como “enqueue”, “dequeue” y “peek”, se pueden realizar en Java utilizando la interfaz Queue y sus implementaciones, como LinkedList y ArrayDeque. Estas clases ofrecen métodos eficientes para agregar, eliminar y acceder a elementos en colas. Al utilizar estos métodos, puedes implementar colas de manera efectiva en tus programas y beneficiarte

de las propiedades y características que ofrecen las estructuras de datos de cola.

6.4. Ejemplos de aplicaciones de colas

6.4.1. Aplicación 1:

Implementación de una Estructura de Datos, utilizando por separado el Tipo de Dato Abstracto Nodo, Cola y el programa principal

Clase NodoCola

```
package ufa_tda8_cola;

/**
 *
 * @author Marcelo
 */
public class NodoCola {
    int dato;
    NodoCola siguiente;

    public NodoCola(int ndato){
        this.dato=ndato;
        this.siguiente=null;
    }

    public void setDato(int dato) {
        this.dato = dato;
    }

    public void setSiguiente(NodoCola siguiente) {
        this.siguiente = siguiente;
    }

    public int getDato() {
        return dato;
    }

    public NodoCola getSiguiente() {
        return siguiente;
    }


}
```

Clase Cola

```
package ufa_tda8_cola;


/**
 *
 * @author Marcelo
 */
//TDA COLA ESTRUCTURA QUE GESTIONA DE FORMAR DINÁMICA LA MEMO-
RIA FIFO
public class Cola {
    NodoCola inicio, cola;
```

```
 int tamanio;
//Constructor
public Cola(){
    this.inicio=this.cola=null;
    this.tamanio=0;
}

//Esta Vacía
//verdadero si no tengo elementos
public boolean estaVacia(){
    return this.inicio==null;
}
//Encolar
public void encolar(int ele){
    NodoCola nuevo=new NodoCola(ele);
    if (estaVacia()){
        this.inicio=nuevo;
    }else{
        this.cola.siguiente=nuevo;
    }
    this.cola=nuevo;
    this.tamanio++;
}

//Dequeue
public int desencolar(){
    int aux=this.inicio.dato;
   inicio=inicio.siguiente;
    tamanio--;
    return aux;
}
//Conocer el elemento del frente
public int inicioCola(){
    return this.inicio.dato;
}
```



```
    //Conocer el tamaño de la cola
    public int tamanioCola(){
        return this.tamanio;
    }

    public int getTamanio() {
        return tamanio;
    }




}
```

Clase Programa Principal

```
package ufa_tda8_cola;

import javax.swing.JOptionPane;

/**
 *
 * @author Marcelo
 */
public class UFA_TDA8_Cola {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        int opcion = 0, elemento = 0;
        Cola ocola = new Cola();
        do {
            try {
                opcion
```

```
Integer.parseInt(JOptionPane.showInputDialog(null,
                    "1. Para ENCOLAR un elemento\n"
                  + "2. Para DESENCOLAR un elemento\n"
                  + "3. La COLA está Vacía\n"
                  + "4. Cuál es el elemento del Inicio/Frente \n"
                  + "5. Cuál es el tamaño de la COLA?\n"
                  + "6. SALIR\n", "Menú de Opciones TDA COLA", JOption-
Pane.QUESTION_MESSAGE));
            switch (opcion) {
                case 1:
                  elemento=Integer.parseInt(JOptionPane.showInputDialo-
g(null,
                        "Ingresa el elemento a ENCOLAR", "ENCOLADO",
                        JOptionPane.QUESTION_MESSAGE));
                  ocola.encolar(elemento);
                  break;
                case 2:
                  if(!ocola.estaVacia()){//tengo elementos
                    JOptionPane.showMessageDialog(null, "El elemento ya
fue atendido: "+
                          ocola.desencolar(),"DESENCOLANDO elementos del
TDA COLA",
                           JOptionPane.INFORMATION_MESSAGE);
                  }else{
                    JOptionPane.showMessageDialog(null, "No se puede
DESENCOLAR, EL TDA COLA esta vacío....",
                          "TDA COLA VACÍO", JOptionPane.INFORMATION_
MESSAGE);
                  }
                  break;
                case 3:
                  if(ocola.estaVacia()){
                    JOptionPane.showMessageDialog(null, "EL TDA COLA
esta vacío....",
                          "TDA COLA VACÍO",
JOptionPane.INFORMATION_MESSAGE);
```



```
                  }else{
                    JOptionPane.showMessageDialog(null, "EL TDA COLA
NO esta vacío, tiene elementos....",
                          "TDA COLA CON ELEMENTOS", JOptionPane.
INFORMATION_MESSAGE);
                  }
                  break;
                case 4:
                  if (!ocola.estaVacia()){
                    JOptionPane.showMessageDialog(null, "El elemento del
FRENTE del TDA COLA es: "+
                          ocola.inicioCola(), "TDA COLA con elementos",
                          JOptionPane.INFORMATION_MESSAGE);
                  }else{
                    JOptionPane.showMessageDialog(null, "EL TDA COLA
esta vacío....",
                          "TDA COLA VACÍO", JOptionPane.INFORMATION_
MESSAGE);
                  }
                  break;
                case 5:
                  JOptionPane.showMessageDialog(null, "El tamaño del TDA
COLA es: "
                      +ocola.getTamanio(),"Elementos COLA",
                      JOptionPane.INFORMATION_MESSAGE );
                  break;
                case 6:
                  JOptionPane.showMessageDialog(null, "Programa Finaliza-
do", "FIN",
                      JOptionPane.INFORMATION_MESSAGE);
                  break;
                default:
                  JOptionPane.showMessageDialog(null, "Opción Incorrecta",
                          "Seleccione una opción correcta", JOp-
tionPane.INFORMATION_MESSAGE);
```

```
                break;
            }

        } catch (NumberFormatException n) {
            JOptionPane.showMessageDialog(null, "Error " + n.getMessa-
ge());
        }

    } while (opcion != 6);

  }

}
```

6.4.2. Aplicación 2:

Implementación de una Tipo de Dato Abstracto en Java denominada Cola con un TDA Automóviles

Clase Cola Automoviles

```
import java.util.LinkedList;
import java.util.Queue;

public class ColaDeAutomoviles {
   public static void main(String[] args) {
      // Crear una cola de automóviles
      Queue<String> colaDeAutomoviles = new LinkedList<>();

      // Agregar automóviles a la cola
      colaDeAutomoviles.offer("Automóvil 1");
      colaDeAutomoviles.offer("Automóvil 2");
      colaDeAutomoviles.offer("Automóvil 3");
      colaDeAutomoviles.offer("Automóvil 4");
      colaDeAutomoviles.offer("Automóvil 5");
```



```
      // Imprimir la cola de automóviles
      System.out.println("Cola de automóviles: " + colaDeAutomoviles);

      // Obtener y eliminar el automóvil del frente de la cola
      String automovilFrente = colaDeAutomoviles.poll();
      System.out.println("Automóvil del frente: " + automovilFrente);

      // Imprimir la cola de automóviles después de eliminar el automóvil
del frente
      System.out.println("Cola de automóviles después de eliminar el auto-
móvil del frente: " + colaDeAutomoviles);

      // Obtener el automóvil del frente sin eliminarlo
      String automovilFrenteSinEliminar = colaDeAutomoviles.peek();
      System.out.println("Automóvil del frente sin eliminarlo: " + automovil-
FrenteSinEliminar);

      // Imprimir la cola de automóviles nuevamente
      System.out.println("Cola de automóviles actualizada: " + colaDeAu-
tomoviles);
   }
}
```

En este programa, se utiliza la clase LinkedList de Java como la implementación de la cola. Primero, se crea una instancia de LinkedList llamada colaDeAutomoviles, que representa la cola de automóviles.

Luego, se agregan algunos automóviles a la cola utilizando el método offer, que agrega el elemento al final de la cola.

Después de eso, se imprime la cola de automóviles

utilizando el método toString() de la lista enlazada.

A continuación, se utiliza el método poll para obtener y eliminar el automóvil del frente de la cola. El automóvil eliminado se asigna a la variable automovilFrente.

Después de eliminar el automóvil del frente, se imprime la cola de automóviles actualizada.

Luego, se utiliza el método peek para obtener el automóvil del frente sin eliminarlo. El automóvil del frente se asigna a la variable automovilFrenteSinEliminar.

Finalmente, se imprime la cola de automóviles actualizada una vez más.

Referencias

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Deitel, P., & Deitel, H. (2017). Java: How to program (Early objects) (11th ed.). Pearson.

Gaddis, T. (2019). Starting Out with Java: Early Objects (7th ed.). Pearson.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java. Hoboken, NJ: John Wiley & Sons.

Gosling, J., Joy, B., Steele, G., & Bracha, G. (2018). The Java® Language Specification (Java SE 14th ed.). Oracle Corporation.

Horstmann, C. S. (2018). Core Java SE 9 for the Impatient (2nd ed.). Addison-Wesley.

Lafore, R. (2002). Data Structures and Algorithms in Java (2nd ed.). Sams Publishing.

Liang, Y. D. (2019). Introduction to Java programming: Comprehensive version (12th ed.). Pearson.

LinkedList (Java Platform SE 8). (n.d.). Retrieved July 5, 2023, from https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Marín Ardila, L. F. (2007). La noción de paradigma. Signo y Pensamiento, 50, 34–45. http://www.scielo.

org.co/scielo.php?script=sci_arttext&pid=S0120-48232007000100004&lng=en&nrm=iso&tlng=es

Oracle. (2021). Java LinkedList Class. Recuperado de https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/LinkedList.html

Netbeans 8 (Windows) System Requirements, Minimum Requirements, Recommended Requirements - PcRequirements.net. (n.d.). Retrieved December 12, 2022, from https://www.pcrequirements.net/en/softwares/netbeans-8-windows-system-requirements/

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.

Sierra, K., & Bates, B. (2020). Head first Java (3rd ed.). O'Reilly Media.

Wigglesworth, M., & McMillan, R. (2019). Java in easy steps (6th ed.). In Easy Steps Limited.

Windows System Requirements for JDK and JRE. (n.d.). Retrieved December 12, 2022, from https://docs.oracle.com/javase/7/docs/webnotes/install/windows/windows-system-requirements.html#bit-64

Zhang, X., Luo, Y., & Li, C. (2020). A Comparative Study on Data Structure in Computer Science Education. Journal of Educational Technology Development and Exchange, 13(2), 153-168.