Estructura de Datos

Apuntes hechos por Brianda García, Lorena Ballesteros, Lía de Miguel y Marina Sancho

Sesión 24/01/2025: Introducción a los tipos abstractos de datos💻

1. Concepto de abstracción

La abstracción es un mecanismo de la mente humana que resulta fundamental para la comprensión de fenómenos o situaciones que incluyen gran cantidad de detalles.
El proceso de abstracción comprende dos aspectos que son complementarios:

1. Destacar los detalles relevantes del objeto, sistema,... en estudio.
2. Ignorar los detalles irrelevantes, por supuesto en este nivel de abstracción.

La abstracción permite tratar la complejidad, entendida como exceso en el número de detalles. Generalmente se trabaja en sentido descendente, de menor a mayor nivel de detalle.

2. Abstracción en el diseño de programas

• Ensamblador: mnemotécnicos y macros.
• Lenguajes de alto nivel: las instrucciones y los programas.
• Programación estructurada: abstracciones de control y abstracción funcional o procedimental.
• Programación modular: división en parte pública y privada.
• Programación funcional y lógica: abstracción de la secuencia de instrucciones que sigue la máquina al ejecutar el programa.
• Programación orientada a objetos: proceso de abstracción entre lo que
puede hacer un objeto y cómo lo hace.ç

3. Abstracción en los datos

Los tipos de datos abstractos (TAD's) permiten el uso de los datos y operaciones (encapsulación) sin conocer su representación ni la implementación de las operaciones (ocultación).

4. Problemas con tipos de datos estructurados

Podemos considerar la implementación siguiente:

```
tipo fecha = Registro
        dia: 1..31
        mes: 1..12
        año: 1..9999
finRegistro
```

Además se puede crear una función auxiliar festivo: fecha → boolean,
pero no hay nada que impida al programador hacer cosas como:

f1.mes ← f2.dia*f2.año (aunque no tenga sentido)

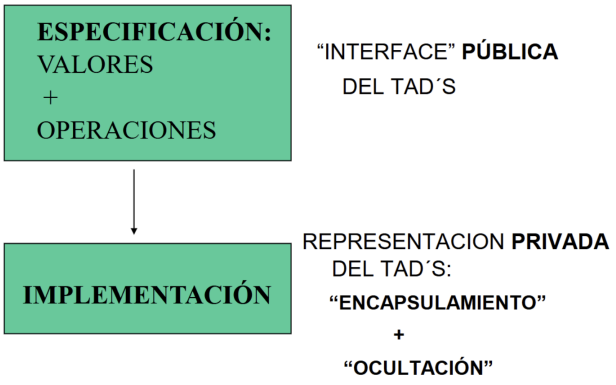
5. Tipos abstractos de datos

El programador debe establecer la interface del Tipo Abstracto de Datos, es decir especificar los valores y las operaciones que van a formar parte del mismo. La interface debe ser pública.

Posteriormente, el programador deberá elegir y realizar la implementación o representación de los datos y operaciones del tipo especificado.

La implementación debe hacerse con ámbito de declaraciones que no sea accesible desde fuera (ámbito privado). Así cualquier modificación o actualización que se realice en dicha implementación no afectará a los programas que la utilicen.

La implementación de las operaciones se "encapsula" junto con la representación del tipo de datos. De esta forma, se facilita la localización para posteriores modificaciones o actualizaciones.



6. Especificación formal de tipos abstractos de datos

Ejemplo: Los números naturales

```
espec NATURALES
        géneros natural
        operaciones
                0:  natural
                suc: natural  natural
                _ + _: natural natural  natural
var x,y: natural
ecuaciones
        x + 0 = x
        x + suc(y) = suc(x + y)
fespec
```

7. Notación asintótica

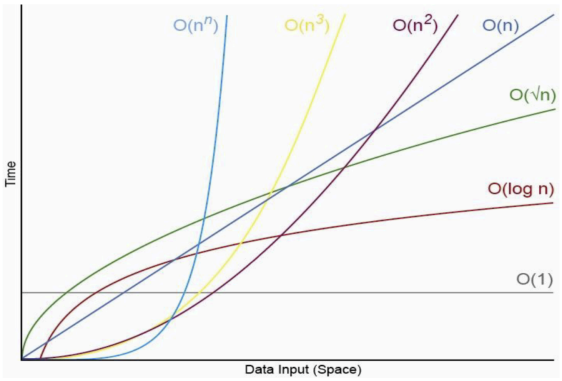
Es una función del tamaño de la entrada de datos (n) utilizada para hacer referencia al tiempo máximo de ejecución T(n) de un programa para una entrada de tamaño n.

– Ejemplo: El tiempo de ejecución de un programa es O(n2): existen dos valores n0 y c de forma que para tamaños de entrada n> n0, se cumple que T(n)<cn2

Suponemos que es posible evaluar programas comparando sus funciones de tiempo de ejecución.

No podemos olvidar la importancia de la velocidad de crecimiento de la función y, por tanto, la influencia del tamaño de las entradas en el tiempo de ejecución.

La elección del algoritmo puede basarse en el comportamiento de la función.



Sesión 31/1/25: Las listas👩‍💻

Listas

Las listas son un **conjunto de elementos relacionados entre sí** que tienen un **determinado orden** (no están ordenadas, pero si tienen orden). Pueden ser de **dos tipos**: **dato primitivo** (entero,string,boolean,etc) o tipo de **datos de objetos y de clases**.

Se pueden dividir las listas de **tres formas**: **listas básicas**, **listas simplemente enlazadas** y **listas doblemente enlazadas**.

Si tenemos una lista podemos: **añadir**(ADD), **borrar**(Delete), **recuperar**(GetIndex o GetValue), **insertar**(Set(Posición,Valor)), **mover o desplazar**(INSERT(Posición,Valor)), **contar** el número de elementos que tiene una determinada lista (GetCount()), **agregar un elemento al tope de la pila** (PUSH) o **eliminar y devolver un elemento en la cima de la pila** (POP).

Las listas básicas tienen **dos implicaciones**: la primera es que el **rendimiento varía** según la implementación, en ArrayList, el acceso es rápido porque usa un array interno. La segunda es que debe de tener un **tamaño máximo** (T_MAX), en otras palabras, tiene un tamaño limitado por la memoria, en Arraylist puede crecer dinámicamente, pero en la práctica está limitado por la memoria disponible del sistema.

Es considerada la mejor forma para tener nuestros datos de forma estructurada. Su tamaño es dinámico (puede cambiar en el tiempo). Además pueden tener dos **tipos de orden**:

- Orden **FIFO** (First In First Out): significa que el primero en entrar es el primero en salir, es decir, en el orden de registros que vamos a ir agregando, el primero que entra, va a ser el primero que va a ser leído después
- Orden **LIFO** (Last In First Out): significa que el último que entró va a ser el primero en salir

En Java existen diferentes **tipos de listas**:

- Arraylists
- Linked Lists
- Stack

Los **Arraylists** son una clase que se representan como una matriz dinámica que permite **almacenar elementos**. **Heredan** de la clase AbstractList, la cual implementa la interfaz List, permite colecciones o **elementos duplicados**, el orden de los registros es el **orden en el que fueron insertados**, permiten un **acceso aleatorio** (tiene índice) y es de **manipulación lenta** (hay que recorrer todo el Arraylist para hacer un cambio).

Para decir de qué va a ser nuestra lista, lo ponemos entre **<type>**.

Para **agregar** elementos a mi lista utilizo: **lista.add()**. Si ya tengo un objeto o persona creado simplemente dentro del paréntesis pongo el nombre. Si no es así utilizo: **lista.add(new Persona())**, dentro de esos dos parámetros pongo los parámetros que están ya declarados.

Hay dos **formas para recorrer** los elementos de mi lista: **for**(utilizando un índice) o **for each**

- **Recorrido foreach**: for(Persona perso:lista){
  System.out.println("prueba: "+perso.getNombre());
  }
- **Recorrer por índice**: for (int i=0(nuestro índice); i<lista.size(); i++(va recorriendo de uno en uno)) {
  System.out.println("prueba: "+ lista.get(i).getNombre());
  }

Para **implementar** o importar la interfaz list hacemos lo siguiente: **import java.util.List**
Para **inicializar** nuestra lista hacemos: **List<Persona> list= new ArrayList<Persona> ()**; [ponemos esos dos paréntesis para hacer referencia a que es una construcción de un nuevo ArrayList].

Dentro de una lista en Java, puedes **almacenar cualquier tipo de datos**, siempre que sean del mismo tipo o compatibles con la declaración de la lista. Ejemplo: números, texto, imágenes o archivos (fotos), o incluso listas dentro de listas.

A lo que metemos en una posición de la lista lo llamamos **e** (no tiene mucha importancia, debido a que no operamos con él), y se usa como abreviatura estándar para referirse a un **elemento genérico**.

En una **lista básica**, podemos dividir su estructura en **dos áreas** principales: el **área de datos**, que en programación orientada a objetos son los **atributos**(variables), se encarga de **guardar los elementos** dentro de la lista, **representan la información** que tiene un objeto y se declaran como **variables de instancia** dentro de la clase. Por ejemplo: ArrayDatos, N_elementos, T_Max, etc. El otro **área es el de funciones**, que en programación orientada a objetos son los **métodos**, **define** las **operaciones/funciones** que se pueden hacer con la lista (como agregar, eliminar o buscar elementos), pueden **modificar o acceder** a los atributos y se declaran como **métodos** dentro de la clase. Ejemplo: Add, Del, GetIndex,GetValue, Insert, Set, Push y Pop.

Se usa **public class** cuando queremos que la clase sea accesible desde cualquier parte del programa. Es obligatorio cuando la clase principal (main) está en un archivo con el mismo nombre.

Sesión 14/02/25: Listas Dinámicas: Listas Simplemente Enlazadas ⛓️

```
class lista <tipoDato> {
    tipoDato datos[ ] = new tipoDatos[ ][7]
```

```
lista <Integrar> a = new lista <Integrar>()
```

1	7	3	/	9	8	7	2	¿4?

Introducción

Una **lista dinámica** es una estructura de datos que permite la gestión eficiente de elementos sin un tamaño fijo predefinido. Se diferencia de los arreglos en que su tamaño puede crecer o reducirse dinámicamente sin necesidad de reservar memoria estática.

Las listas dinámicas pueden clasificarse en:

1. **Listas Simplemente Enlazadas**
2. **Listas Doblemente Enlazadas**

Concepto de Lista Simplemente Enlazada

Una **lista simplemente enlazada** es una estructura de datos dinámica compuesta por nodos. Cada nodo contiene dos partes:

- **Dato**: Almacena la información que queremos guardar en la lista.
- **Referencia al siguiente nodo**: Un puntero o referencia al siguiente nodo en la lista.

Este tipo de lista solo permite recorrer los elementos en una dirección (de la cabeza al final de la lista).

Uso de Templates en Listas

Para hacer la lista genérica y permitir almacenar cualquier tipo de dato, utilizamos **templates**. Esto permite definir una lista sin especificar el tipo de dato que contendrá hasta el momento de su instanciación (es el proceso de crear un objeto a partir de una clase en programación orientada a objetos).

Ejemplo de la definición de una lista genérica:

```
class ListaSe<T> {
   private ElementoSe<T> primerElemento;
}
```

Aquí, T representa un tipo genérico que puede ser reemplazado por cualquier tipo de dato al momento de la creación de la lista.

Implementación de un Nodo en la Lista

Cada nodo (elemento) de la lista almacena un dato y una referencia al siguiente nodo:

```
class ElementoSe<T> {
   T dato;
   ElementoSe<T> siguiente;

   public ElementoSe(T dato) {
      this.dato = dato;
      this.siguiente = null;
   }
}
```

Creación e Inicialización de una Lista en Java

Para crear una lista de enteros:

ListaSe<Integer> miLista = new ListaSe<Integer>();

Ejemplo de uso dentro de un método:

```
void miPrueba() {
   Integer a = new Integer(5);
   miLista.add(a);
}
```

Métodos Fundamentales de una Lista Simplemente Enlazada

Método para Insertar un Elemento en la Lista

La inserción de un elemento sigue el esquema de una pila (LIFO - Last In, First Out):

```
public void add(T dato) {
   ElementoSe<T> temporal = new ElementoSe<T>(dato);
   temporal.siguiente = this.primerElemento;
   this.primerElemento = temporal;
}
```

Método para Buscar un Elemento en la Lista

```
public boolean contains(T dato) {
   ElementoSe<T> actual = primerElemento;
   while (actual != null) {
      if (actual.dato.equals(dato)) {
```

```
            return true;
        }
        actual = actual.siguiente;
    }
    return false;
}
```

Método para Eliminar un Elemento de la Lista

```
public void remove(T dato) {
    if (primerElemento == null) return;

    if (primerElemento.dato.equals(dato)) {
        primerElemento = primerElemento.siguiente;
        return;
    }

    ElementoSe<T> actual = primerElemento;
    while (actual.siguiente != null && !actual.siguiente.dato.equals(dato)) {
        actual = actual.siguiente;
    }

    if (actual.siguiente != null) {
        actual.siguiente = actual.siguiente.siguiente;
    }
}
```

Consideraciones sobre el uso de Object

En Java, todos los objetos derivan de la clase Object. Sin embargo, usar Object en lugar de un tipo genérico puede causar problemas si se requiere una conversión de tipos estricta.

Casos Límite en el Manejo de Listas

- **Lista vacía:** Se debe comprobar que la lista no está vacía antes de realizar operaciones como eliminación o acceso.
- **Elemento no encontrado:** En métodos de eliminación o búsqueda, hay que considerar el caso en que el elemento no esté en la lista.
- **Eliminación del primer elemento:** Se requiere un tratamiento especial si el elemento a eliminar es el primero de la lista.

Comparación de Objetos en la Lista

Cuando se trabaja con objetos, la comparación de elementos debe hacerse correctamente:

```
if (actual.dato.equals(dato)) {
    // Comparación correcta
```

```
}
```

Usar == en lugar de equals() podría llevar a errores, ya que == compara referencias de memoria en lugar de contenido.

Sesión 19/02/25: Listas

¿Cuántas veces recorro yo una lista?

Dependerá de la operación que se realice sobre la lista. En algunos algoritmos, como el de la burbuja, se recorrerá varias veces hasta que los elementos estén ordenados.

Si se trata de una búsqueda secuencial, la lista se recorrerá hasta encontrar el elemento o llegar al final.

Cuando se usa un iterador, este controla el recorrido y avisa cuando no hay más elementos disponibles.

¿Tiene la lista límites?

Por otro lado, las listas pueden ser **limitadas o ilimitadas** en función de su implementación.

- **ArrayList o listas basadas en arreglos**: tienen un tamaño máximo predefinido o dinámico (pueden crecer, pero con costo).
- **Listas enlazadas**: pueden crecer de manera dinámica sin una restricción fija, pero cada nodo tiene un costo en memoria adicional.

¿Cómo recorrer una lista?

Existen varias formas de recorrer una lista:

1. **Mediante índices (si es una lista basada en arrays).**
2. **Con búcles (for, while, foreach).**
3. **Usando un iterador**, que encapsula la lógica del recorrido.

Comportamiento

El número de elementos n de la lista puede aumentar (mediante inserciones de elementos en la lista) o disminuir (mediante eliminaciones de elementos), según se requiera en t. de ejecución.

Se pueden insertar y eliminar elementos en cualquier posición de la lista (por el principio, por el final o en cualquier otra posición).

Puede haber listas de elementos de clase String, listas de reales, etc… y de objetos de cualquier clase.

Operaciones sobre listas

Un **TAD** lista, además de definir los datos que puede contener una lista, tiene definidas las operaciones que pueden realizarse con la lista.

Un TAD (Tipo Abstracto de Datos) Lista es una estructura de datos abstracta que representa una secuencia ordenada de elementos, donde cada elemento tiene una posición definida y pueden realizarse operaciones como inserción, eliminación y recorrido.

Características:

- Ordenada: los elementos se almacenan en un orden específico.
- Puede contener duplicados: A diferencia de los conjuntos, las listas permiten repetir elementos.
- Acceso secuencial o por posición: Dependiendo de la implementación (listas enlazadas o basadas en arreglos).
- Tamaño variable o fijo: Puede ser estática (arrays) o dinámica (listas enlazadas).

Operaciones básicas de un TAD LISTA

Operación	Descripción
insertar(pos, elem)	Agrega un elemento en una posición específica
eliminar(pos)	Elimina el elemento en la posición indicada.
buscar(elem)	Devuelve la posición de un elemento si existe
obtener(pos)	Devuelve el elemento en la posición dada
tamaño()	Retorna la cantidad de elementos en la lista
vacía()	Indica si la lista esta vacia
recorrer()	Permite iterar sobre los elementos de la lista

Implementaciones comunes

1. Lista basada en arreglos

Se implementa con un array dinámico, lo que permite acceso rápido a los elementos mediante un índice, pero la inserción/eliminación en posiciones intermedias es costosa.

Ventajas

- Acceso por índice en **O(1)**.
- Menos consumo de memoria que una lista enlazada.

Desventajas

- La inserción y eliminación en el medio son costosas **O(n)**.
- Tamaño limitado si no se usa una estructura dinámica (ej. ArrayList).

```
class ListaArray<T> {
    private T[ ] datos;
    private int tamaño;

    public ListaArray(int capacidad) {
            datos = (T[ ]) new Object[capacidad];
            tamaño = 0;}

     public void insertar(int pos, T elem) {
            if (pos < 0 || pos > tamaño) throw new IndexOutOfBoundsException();
            for (int i = tamaño; i > pos; i--) {
                  datos[i] = datos[i - 1];    }

            datos[pos] = elem;
            tamaño++;    }

    public T obtener(int pos) {
             if (pos < 0 || pos >= tamaño) throw new IndexOutOfBoundsException();
                return datos[pos];
                  }
}
```

Las operaciones son necesarias para dotar a las listas del comportamiento descrito

¿Qué operaciones definimos para el TAD Lista?

Como mÌnimo un conjunto de operaciones capaces de generar cualquier lista de este TAD:

- Crear una lista vacía (en Java lo hará el constructor),
- Insertar un elemento (o al principio o en una posición intermedia o al final de la lista).
- Eliminar un elemento (o el primero o de una posición intermedia o el del final de la lista).

Además de las operaciones suficientes para otorgar al TAD Lista el comportamiento especificado, pueden añadirse otras que sean útiles para obtener propiedades sobre la lista y para acceder a los datos:

- Decir si una lista está vacía o no.
- Decir la longitud de la lista.
- Decir qué elemento hay en cierta posición de la lista
- Decir si cierto elemento está o no en la lista.
- Recorrer una lista para obtener cada elemento de esta desde el primero hasta el último en el orden en que se encuentran en la lista.
- Modificar el valor de un elemento de la lista

Iteradores

Un **iterador** es una clase de utilidad que permite recorrer una colección elemento a elemento.

Interfaz Iterator en Java:

```
public interface Iterator<E> {
   boolean hasNext(); // ¿Hay más elementos?
   E next(); // Devuelve el siguiente elemento.
   void remove(); // Elimina el último elemento retornado.
}
```

Implementación básica de un iterador en Java:

```
class Iterador<T> {
   private Lista<T> lista;
   private int indice;
   public Iterador(Lista<T> lista) {
      this.lista = lista;
      this.indice = 0;
   }
   public boolean haySiguiente() {
      return indice < lista.tamaño();
   }
   public T getDatos() {
      return lista.obtener(indice++);
   }
}
```

Métodos clave relacionados con el iterador:

- getActual() : Devuelve el elemento actual.
- getIterador() : Devuelve un iterador de la lista.
- find(K clave) : Busca un elemento por clave.

Diferencias entre listas simples y listas doblemente enlazadas

Características	**Listas Simplemente Enlazadas**	**Listas Doblemente Enlazadas**
Dirección de enlaces	Solo hacia adelante	Hacia adelante y hacia atrás
Complejidad de inserción	O(1) si se tiene el nodo	O(1) sis e tiene el nodo
Acceso a elementos intermedios	O(n)	O(n)
Consumo de memoria	Menor (un puntero por nodo)	Mayor (dos punteros por nodo)

Consideraciones finales

Antes de implementar un TAD mediante una clase en Java, es conveniente diseñarlo. El diseño de un TAD se llama **especificación.** Llamemos **E** al tipo o a la clase de los elementos de una lista.

Es muy importante hacer una especificación detallada de:

- La descripción de los elementos o valores que puede contener la lista
- Las operaciones, que pueden ser las siguientes:

Por ejemplo, una lista en Java podría implementar estas operaciones básicas:

- boolean add(E elem);
- void add(int index, E elem);
- int size();
- boolean isEmpty();
- E get(int index);

Estos métodos permiten manipular la lista y recorrer sus elementos de forma eficiente según la implementación elegida.

Sesión 21/02/25

Listas Doblemente Enlazadas

Definición

Una lista doblemente enlazada es una estructura de datos en la que cada nodo contiene un puntero al nodo anterior y otro al nodo siguiente. Esto permite recorrer la lista en ambas direcciones, facilitando la inserción y eliminación de elementos en cualquier posición.

Método add(TD dato)

- Se utiliza para añadir un elemento al final de la lista.
- Se crea un nuevo nodo con el dato proporcionado.
- Si la lista está vacía (primero es `null`), el nuevo nodo se convierte en el primero y el último.
- Si la lista ya contiene elementos:
  1. Se enlaza el nuevo nodo con el último nodo actual.
  2. Se actualiza el puntero `siguiente` del último nodo actual para apuntar al nuevo nodo.
  3. Se actualiza `último` para que sea el nuevo nodo.

Método insert(ElementoDE<TD> elemento, TD dato)

- Permite insertar un nuevo nodo antes de un nodo dado.
- Si el nodo dado es `null`, se usa `add(dato)` para añadir al final.
- Si se inserta antes del primer nodo:
  1. Se enlaza el nuevo nodo como el primero.
  2. Se ajustan los punteros correspondientes.
- Si se inserta en otra posición:
  1. Se enlaza el nuevo nodo entre `elemento.anterior` y `elemento`.
  2. Se actualizan los punteros de los nodos adyacentes para mantener la estructura de la lista.

Método delete(ElementoDE<TD> elemento)

- Permite eliminar un nodo de la lista si no es `null`.
- Se verifican tres casos:
  1. Si el nodo a eliminar es el primero, se actualiza `primero`.
  2. Si el nodo a eliminar es el último, se actualiza `ultimo`.
  3. Si el nodo está en medio, se ajustan los punteros de los nodos anterior y siguiente.

Estructura del nodo `ElementoDE<TD>

- Cada nodo almacena:
  - Un dato genérico `TD`.
  - Un puntero `anterior` al nodo previo.
  - Un puntero `siguiente` al nodo siguiente.
- Se inicializan `anterior` y `siguiente` en `null` al crear un nuevo nodo.

Diferencias entre ‘add’ e ‘insert’

- `add(dato)`: Agrega un nuevo nodo al final de la lista.
- `insert(elemento, dato)`: Inserta un nuevo nodo antes de un nodo existente en la lista.

Ventajas de la Lista Doblemente Enlazada

- Facilita la eliminación de nodos en comparación con listas simplemente enlazadas.
- Permite recorrer la lista en ambas direcciones.
- Inserción y eliminación eficientes en cualquier posición de la lista.

Desventajas de la Lista Doblemente Enlazada

- Mayor uso de memoria debido a los punteros adicionales (`anterior` y `siguiente`).
- Mayor complejidad en la implementación en comparación con una lista simplemente enlazada.

Estos métodos permiten gestionar eficientemente una **Lista Doblemente Enlazada**, manteniendo referencias a los nodos anteriores y siguientes para facilitar la navegación y modificación de la estructura.

Pilas

Una pila es una estructura de datos en la que los elementos se añaden y eliminan siguiendo el principio LIFO (Last In, First Out), es decir, el último elemento agregado es el primero en ser eliminado.

Características de una pila

- Se puede implementar con arrays o listas enlazadas.
- Solo permite acceso a un único extremo: el tope de la pila.
- Útil en situaciones como deshacer operaciones, evaluación de expresiones, gestión de llamadas a funciones (pila de ejecución).

Operaciones en una pila

- push(T dato): Añade un elemento a la pila.
- pop(): Elimina y devuelve el último elemento insertado.

- peek(): Devuelve el último elemento sin eliminarlo.
- isEmpty(): Devuelve `true` si la pila está vacía, `false` en caso contrario.

Implementación de una pila con lista doblemente enlazada

```
class PilaLIFO<T> {
   ListaDE<T> pila = new ListaDE<>();

   void push(T dato) {
      pila.add(dato);
   }

   T pop() {
      if (pila.ultimo == null) {
         throw new RuntimeException("La pila está vacía");
      }
      T temporal = pila.ultimo.getDato();
      pila.delete(pila.ultimo);
      return temporal;
   }
}
```

Ventajas y desventajas

✅ Simplicidad en la implementación.
✅ Uso eficiente de memoria si se implementa con listas enlazadas.
❌ Acceso limitado a elementos internos.

Colas

Una cola es una estructura de datos en la que los elementos se añaden y eliminan siguiendo el principio FIFO (First In, First Out), es decir, el primer elemento añadido es el primero en salir.

Características de una cola

- Tiene un extremo de entrada (rear) y otro de salida (front).
- Se usa en gestión de procesos en sistemas operativos, impresión de documentos, tráfico en redes, entre otros.

Operaciones en una cola

- enqueue(T dato): Añade un elemento al final de la cola.
- dequeue(): Elimina y devuelve el primer elemento de la cola.
- peek(): Devuelve el primer elemento sin eliminarlo.
- isEmpty(): Devuelve `true` si la cola está vacía, `false` en caso contrario.

Variaciones de colas

- Cola con prioridad: Los elementos se insertan en la cola, pero pueden adelantarse según su prioridad.
- Cola circular: Se maneja de manera circular, permitiendo reutilizar posiciones anteriores.
- Cola de doble extremo (deque): Permite inserciones y eliminaciones por ambos extremos.

Ejemplo de implementación de una cola

```
class ColaFIFO<T> {
   ListaDE<T> cola = new ListaDE<>();

   void enqueue(T dato) {
      cola.add(dato);
   }

   T dequeue() {
      if (cola.primero == null) {
         throw new RuntimeException("La cola está vacía");
      }
      T temporal = cola.primero.getDato();
      cola.delete(cola.primero);
      return temporal;
   }
}
```

Ventajas y desventajas

✅ Gestión ordenada de elementos.
✅ Útil en múltiples aplicaciones prácticas.
❌ Puede haber desperdicio de memoria si no se usa una implementación circular.

Ejercicio de examen: Orden en un semáforo

La Situación: Dado el siguiente orden de llegada de vehículos:

1. Coche
2. Coche
3. Coche
4. Moto
5. Ambulancia

Reglas

- Los vehículos respetan el orden FIFO (los primeros en llegar son los primeros en avanzar).
- Las ambulancias tienen prioridad y no se detienen en el semáforo.

Orden de paso por el semáforo

1. Moto
2. Coche
3. Coche
4. Coche

¿Y la ambulancia?

- La ambulancia no se detiene en el semáforo, avanza directamente sin esperar turno.

Conjuntos

Un conjunto es una estructura de datos que almacena elementos únicos, es decir, no permite duplicados.

Operaciones en un conjunto

- insert(T dato): Agrega un elemento si no está en el conjunto.
- delete(T dato): Elimina un elemento del conjunto.
- contains(T dato): Devuelve `true` si el elemento está en el conjunto, `false` en caso contrario.
- union(Conjunto otro): Devuelve un nuevo conjunto con la unión de ambos conjuntos.
- intersección(Conjunto otro): Devuelve un nuevo conjunto con los elementos comunes.

Ejemplo de implementación de un conjunto

```
import java.util.HashSet;

class Conjunto<T> {
    private HashSet<T> elementos = new HashSet<>();

    void insert(T dato) {
        elementos.add(dato);
    }

    void delete(T dato) {
        elementos.remove(dato);
    }

    boolean contains(T dato) {
        return elementos.contains(dato);
    }
}
```

Ventajas y desventajas

✅ Operaciones rápidas de búsqueda y eliminación.
✅ Evita elementos duplicados.
❌ No mantiene orden de inserción en algunas implementaciones.

Sesión 25/02/25: Listas Doblemente Enlazadas

1. Diccionarios

Los diccionarios son estructuras de datos que permiten almacenar información en pares clave-valor (<K, V>). Son estructuras que permiten una búsqueda rápida de los datos a través de sus claves.

- Características:

- Cada clave es única y no se puede repetir.
- Los valores pueden ser cualquier tipo de dato.
- Se pueden considerar como listas doblemente enlazadas, donde cada elemento tiene un nodo previo y otro siguiente, utilizándose hasta llegar al final.
- En algunos casos, los diccionarios pueden mantener el orden de inserción, lo que facilita algunas operaciones.
- Permiten acceso rápido a los elementos mediante sus claves.

- Tipos de Diccionarios

1. **Diccionario no ordenado**: No garantiza el orden de los elementos.
2. **Diccionario ordenado**: Mantiene un orden predefinido de los elementos.

- Estructura de un diccionario

- **Diccionario <K, V>:** Representa la colección de pares clave-valor.
- **Elemento D <K, V>**: Cada par clave-valor dentro del diccionario.
- **Iterador D <K, V>**: Permite recorrer los elementos del diccionario.
- **Elemento D <K, V> Primero**: Primer elemento del diccionario.
- **Elemento D <K, V> Último**: Último elemento del diccionario.
- **Elemento D <K, V> Anterior**: Elemento anterior en la estructura.
- **Elemento D <K, V> Siguiente**: Elemento siguiente en la estructura.
- **K (Índice):** Clave única de cada elemento.
- **V (Dato):** Valor asociado a la clave.

- Implementación

Los diccionarios pueden implementarse como clases, aunque también pueden representarse mediante interfaces. La forma en que se manejan los enlaces entre los elementos varía dependiendo de la

implementación específica. En estructuras ordenadas, los enlaces "anterior" y "siguiente" pueden basarse en relaciones de menor o mayor valor.

- **Métodos comunes en un diccionario**

```
Diccionario<K, V> miDiccionario;
ElementoD<K, V> actual;

boolean add(K, V); // Agrega un nuevo elemento
boolean insert(K, V); // Inserta un nuevo elemento
boolean delete(K, V); // Elimina un elemento específico

Lista<K> getKeys(); // Obtiene todas las claves
Lista<V> getValues(); // Obtiene todos los valores
boolean exists(K key); // Verifica si una clave existe en el diccionario
V getValue(K key); // Obtiene el valor asociado a una clave

boolean updateValue(K key, V value); // Actualiza el valor de una clave específica

boolean hasNext(); // Verifica si hay un siguiente elemento
V next(); // Devuelve el siguiente valor
K getKey(); // Obtiene la clave del elemento actual (sin avanzar)
V getValue(); // Obtiene el valor del elemento actual (sin avanzar)
```

- **Diferencias entre Diccionarios y Listas Enlazadas**

Características	Diccionarios	Listas Enlazadas
Acceso	Rápido por clave	Secuencial
Orden	Que puede mantener el orden de inserción	Mantiene orden implícito por enlace
Búsqueda	Basada en clave	Requiere recorrer la lista
Elemento único	Clases únicas	Elementos repetibles
Enlace	Puede tener enlaces anteriores y siguientes	Generalmente solo “siguiente”en listas simples

2. Listas

- Definición

Son estructuras de datos en las que cada nodo tiene dos referencias: una al nodo anterior y otra al nodo siguiente.

- No existe un orden implícito.
- No se mantiene un orden predefinido.

- Características Principales

- Permiten recorridos bidireccionales (hacia adelante y hacia atrás).
- Mayor flexibilidad que las listas simplemente enlazadas para insertar y eliminar elementos.
- El acceso a un elemento intermedio es más costoso en términos de tiempo en comparación con arreglos.
-

- Operaciones Comunes

- **Insertar**: Agregar un nodo en una posición dada (inicio, final o intermedio).
- **Eliminar**: Quitar un nodo de una posición específica.
- **Buscar**: Localizar un nodo con un valor determinado.
- **Recorrer**: Desplazarse por la lista hacia adelante o hacia atrás.

- Interfaz Comparable

- Permite comparar objetos para definir un orden.
- Es una interfaz en Java que obliga a implementar el método **compareTo()**.

Ejemplo de comparación:

- Si son cadenas de caracteres, "Antonio" es mayor que "Maria" (orden lexicográfico).
- Si son fechas de nacimiento, el criterio de orden será cronológico.

- Propiedades de los Diccionarios

- **Clave única**: No puede haber dos elementos con la misma clave.
- **Acceso rápido**: El acceso a un valor dado su clave suele ser eficiente (O(1) en diccionarios implementados con tablas hash).
- **Mutabilidad**: Puede insertar, actualizar y eliminar elementos.

- Métodos del Diccionario

```
boolean delete(K clave);
K getKey();
V getValue();
boolean updateValue(K clave, V valor);
```

- delete(): Elimina un elemento del diccionario.
- getKey(): Devuelve la clave de un elemento.
- getValue(): Devuelve el valor asociado a una clave.
- updateValue(): Actualiza el valor asociado a una clave.
- **Nota**: Estos métodos no avanzan el iterador, solo devuelven los datos actuales.

3. Iteradores

Un iterador es un objeto que permite recorrer una colección de elementos uno a uno.

- No incluyen un método específico de búsqueda.
- Pueden existir distintos tipos de iteradores.
- Sirven para desplazarse por la estructura y extraer elementos.

- Métodos del Iterador

```
boolean hasNext(); // Verifica si hay un siguiente elemento.
V next();          // Devuelve el siguiente elemento y avanza el iterador.
```

- **Método para Extraer Claves**

```
Lista<K> getKeys() {
        Iterador<K,V> it = this.getIterador();
        Lista<K> claves = new Lista<K>();

        while (it.hasNext()) {
            it.next();
            claves.add(it.getKey());
        }
        return claves;
}
```

getValues(): Funciona de la misma forma que getKeys(), pero extrayendo los valores.

- **Verificar Existencia de una Clave**

```
boolean exists(K clave) {
        Iterador<K,V> it = this.getIterator();
        while (it.hasNext()) {
            it.next();
            if (it.getKey().equals(clave)) {
        return true;
    }
  }
  return false;
}
```

- **Obtener un Valor por su Clave**

```
V getValue(K clave) {
        Iterador<K,V> it = new IteradorD<>();
        while (it.hasNext()) {
                it.next();
                if (it.getKey().equals(clave)) {
                        return it.getValue();} }
  return null;
}
```

Nuevos Métodos del Diccionario

```
V getValue(K clave) {
        Iterador<K,V> it = new IteradorD<>();
        while (it.hasNext()) {
                it.next();
                if (it.getKey().equals(clave)) {
                        return it.getValue();} }
  return null;
}
```

Método find() con Iterador

```
IteradorD<K,V> find(Iterador<K,V> it, K clave) {
   while (it.hasNext()) {
      it.next();
      if (it.getKey().equals(clave)) {
         return it;
      }
   }
   return null;
}
```

Método find() por Clave

```
IteradorD<K,V> find(K clave) {
   IteradorD<K,V> it = new IteradorD<>();
   return this.find(it, clave);
}
```

Método exists() Mejorado

```
boolean exists(K clave) {
   return this.find(clave) != null;
}
```

Obtener Valor por Clave con find()

```
V getValue(K clave) {
   IteradorD<K,V> it = this.find(clave);
   return (it != null) ? it.getValue() : null;
}
```

Insertar o Actualizar un Elemento

```
boolean insert(K clave, V dato) {
   IteradorD<K,V> it = this.find(clave);
```

```
    if (it != null && it.getActual() != null) {
        it.getActual().setValue(dato);
    } else {
        this.add(clave, dato);
    }
}
```

Árboles Binarios de Búsqueda (ABB)

Un árbol binario de búsqueda es una estructura de datos en forma de árbol donde cada nodo tiene como máximo dos hijos y se organiza según las siguientes reglas:

- No permite elementos repetidos.
- Los elementos menores van a la izquierda del nodo.
- Los elementos mayores van a la derecha del nodo.

Problemas con los Árboles

- Mantener el equilibrio del árbol (para optimizar las operaciones de búsqueda, inserción y eliminación).
- Identificar los puntos de intersección para mantener el balance.

Desplazamiento con el Iterador

- Permite recorrer la lista y pasar la información a otras funciones.
- Se utiliza para localizar y operar sobre elementos específicos.