Financiado por la Unión Europea NextGenerationEU

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Plan de Recuperación, Transformación y Resiliencia

GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

GVANEXT
Fondos Next Generation en la Comunitat Valenciana

IES ANTONIO SEQUEROS ALMORADÍ

TEMA 09 – TIPOS LINEALES

PROGRAMACIÓN
CFGS DAM

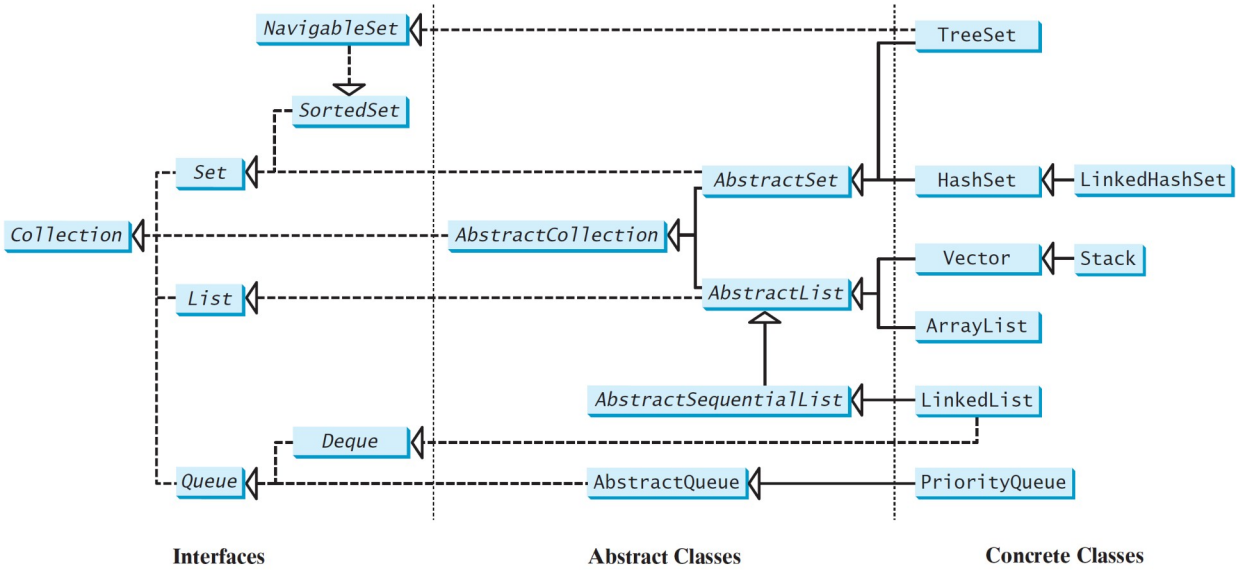
2023/2024

Versión: 240205.1038

ÍNDICE DE CONTENIDO

UD09 – TIPOS LINEALES

1. COLECCIONES



2. TIPOS LINEALES

Aquellos cuyos elementos están formados por secuencias, en los que generalmente se pueden realizar las operaciones de inserción, eliminación, consulta o búsqueda de elementos en una posición determinada.

Tres tipos de datos lineales son: Lista, Pila y Cola

2.1 Listas enlazadas

El objetivo para colecciones como pilas y colas es garantizar que la cantidad de memoria utilizada sea proporcional al número de elementos de la colección.

El uso de una matriz de longitud fija para implementar una pila va en contra de este objetivo, si se crea una pila con una capacidad determinada se está desperdiciando una cantidad potencialmente enorme de memoria cuando la pila está vacía o casi vacía.

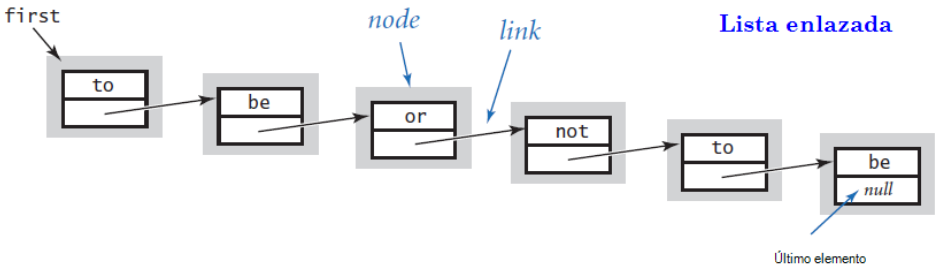
Consideremos ahora el uso de una estructura de datos fundamental conocida como lista enlazada, que puede proporcionar implementaciones de colecciones (y, en particular, pilas y colas) que alcanzan el objetivo citado.

Una lista enlazada consta de una secuencia de nodos, cada uno de los cuales contiene una referencia (o enlace) a su sucesor. Por convención, el enlace del último nodo es nulo, para indicar que termina la lista.

Un nodo es una entidad abstracta que puede contener cualquier tipo de dato, además del enlace que caracteriza su papel en la construcción de listas enlazadas.

Usaremos una representación visual en la que:

- Dibujamos un rectángulo para representar cada nodo de la lista enlazada.
- Colocamos el elemento y el enlace dentro del rectángulo.
- Utilizamos flechas que apuntan a los objetos referenciados para representar las referencias.



Esta representación visual captura la característica esencial de las listas enlazadas y se centra en los enlaces. Por ejemplo, el diagrama ilustra una lista enlazada simple que contiene la secuencia de elementos to, be, or, not, to, y be.

Definimos una clase para la abstracción de nodos que es recursiva por naturaleza. Al igual que con las funciones el concepto de estructuras de datos recursivas puede ser un poco alucinante al principio. Un objeto Nodo tiene dos variables de instancia: una String y un Nodo.

```
class Node
{
   String item;
   Node next;
}
```

La variable de instancia String son los datos del Nodo. La variable de instancia Nodo caracteriza la naturaleza enlazada de la estructura de datos: almacena una referencia al Nodo sucesor en la lista enlazada (o null para indicar que no existe tal nodo).

Utilizando esta definición recursiva, podemos representar una lista enlazada con una variable de tipo Nodo asegurándonos de que su valor es nulo o una referencia a un Nodo cuyo campo siguiente sea una referencia a una lista enlazada.

Para enfatizar que sólo estamos utilizando la clase Node para estructurar los datos, no definimos ningún método de instancia. Como con cualquier clase, podemos crear un objeto de tipo Nodo

invocando al constructor (sin argumentos) con new Nodo(). El resultado es una referencia a un nuevo objeto Node cuyas variables de instancia se inicializan con el valor por defecto null. Por ejemplo, para construir una lista enlazada que contiene la secuencia de elementos *to*, *be* y *or*, creamos un Nodo para cada elemento:

```
Nodo first = new Nodo();
Nodo second = new Nodo();
Nodo third = new Nodo();
```

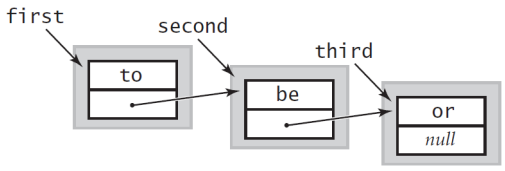
Y asignamos la variable de instancia item en cada uno de los nodos al valor deseado:

```
first.item = "to";
second.item = "be";
third.item = "or";
```

y establece las siguientes variables de instancia para construir la lista enlazada:

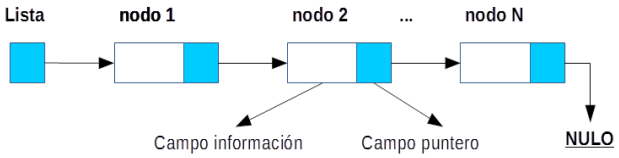
```
first.next = second;
second. next = third;
```

Como resultado, first es una referencia al primer nodo nodo de una lista enlazada de tres nodos, second es una referencia al segundo nodo, y third es una referencia al último nodo.

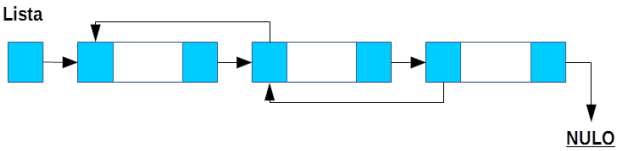


2.1.1 Tipos de Listas

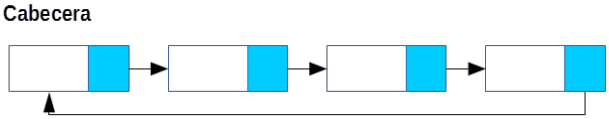
Lista simplemente enlazada



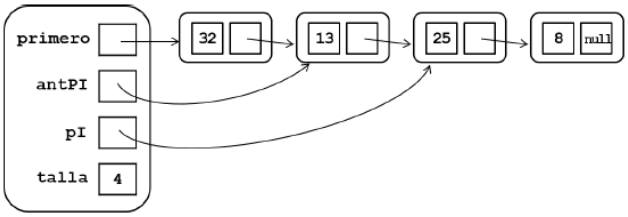
Lista doblemente enlazada



Lista circular



Listas con Cursor



2.2 Tipos genéricos de Java.

Si necesitáramos que el dato fuera Integer o float debemos crear una clase para cada tipo distinto?

Un mecanismo específico en Java conocido como tipos genéricos resuelve el problema.

Con los genéricos, podemos construir colecciones de objetos de un tipo a especificar por el código cliente. El principal beneficio de hacer esto es la capacidad de descubrir errores de desajuste de tipos en tiempo de compilación (cuando se está desarrollando el software) en lugar de en tiempo de ejecución (cuando el software está siendo utilizado por un cliente). Conceptualmente, los genéricos son un poco confusos al principio (su impacto en el lenguaje de programación es tan profundo que no se incluyeron en las primeras versiones de Java), pero su uso en el presente contexto implica sólo un poco más de sintaxis de Java y es fácil de entender.

Llamamos a la clase genérica Pila y elegimos el genérico llamado Item para el tipo de los objetos de la pila (puede utilizar cualquier nombre). El código de Pila es idéntico al código Lista excepto que reemplazamos cada ocurrencia de de String por Item y declaramos la clase con la siguiente primera línea de código:

```
public class Pila<Item>
```

El nombre Item es un parámetro de tipo, un marcador de posición simbólico para algún tipo real a especificar por el cliente. Puedes leer Pila<Item> como pila de elementos, que es precisamente lo que queremos. Al implementar Pila, no conocemos el tipo real de Item, pero un cliente puede usar nuestra pila para cualquier tipo de dato, incluso uno definido mucho tiempo

después de que desarrollemos nuestra implementación. El código del cliente especifica el argumento de tipo Manzana cuando se crea la pila:

```
Pila<Manzana> pila = new Pila<Manzana>();
Manzana a = new Manzana();
…
pila.push(a);
```

Si intentas añadir un objeto del tipo incorrecto en la pila, como esto

```
Pila<Manzana> pila = new Pila<Manzana>();
Manzana a = new Manzana();
Naranja b = new Naranja();
pila.push(a);
pila.push(b); // Error de compilación.
```

obtendrás un error de compilación:

```
push(Manzana) en Pila<Manzana> no se puede aplicar a (Naranja)
```

Además, en nuestra implementación de Pila (Stack), Java puede utilizar el parámetro de tipo Item para comprobar si hay errores de coincidencia de tipo: aunque no se conozca el tipo real, a las variables de tipo Item deben tener asignados valores de tipo Item, etc.

2.3 Pilas

Una pila (stack en inglés) es una secuencia en la que el acceso al primer elemento se realiza siguiendo un criterio LIFO (Last In First Out). Los elementos de una pila siempre se eliminan de ella en orden inverso al que fueron colocados, de modo que el último en entrar es el primero en salir y viceversa, el primero en entrar es el último en salir. Un ejemplo típico es la secuencia de registros de activación que coexisten en memoria, que se gestiona como una pila y de ahí el nombre que recibe la zona de memoria en la que se ubican estos registros.

El tipo de datos Pila presenta la siguiente interfaz de operaciones disponibles: crear una pila, apilar un nuevo elemento sobre la pila, desapilar el elemento que se encuentra en la cima de la pila, consultar (sin desapilar) el valor del dato que se encuentra en la cima de la pila, preguntar si una pila está vacía y, por último, obtener el número de elementos de la pila.

```
package lineales;

import java.util.NoSuchElementException;

/**
 * Clase Pilas: Pila de int. Implementación enlazada.
 *
 * @author
 * @version
```

```
 */
public class Pilas {

    private NodoInt cima;
    private int talla;

    /**
     * Crea una pila vacía.
     */
    public Pilas() {
        cima = null;
        talla = 0;
    }

    /**
     * Apila x en la cima de la pila.
     *
     * @param int x, el valor a apilar.
     */
    public void apilar(int x) {
        cima = new NodoInt(x, cima);
        talla++;
    }

    /**
     * Desapila y devuelve el elemento de la cima de la pila.
     *
     * @return int, la cima de la pila antes de desapilar.
     * @throws NoSuchElementException si la pila está vacía.
     */
    public int desapilar() {
        if (cima == null) {
            throw new NoSuchElementException("Pila vacía");
        }
        int x = cima.dato;
        cima = cima.siguiente;
        talla--;
        return x;
    }

    /**
     * Devuelve el elemento en la cima de la pila.
     *
     * @return int, la cima de la pila.
     * @throws NoSuchElementException si la pila está vacía.
     */
    public int cima() {
        if (cima == null) {
            throw new NoSuchElementException("Pila vacía");
        }
        return cima.dato;
    }

    /**
     * Comprueba si la pila está vacía.
     *
```

```
     * @return boolean, true si la pila está vacía y false en caso contrario.
     */
    public boolean esVacia() {
        return (cima == null);
    }

    /**
     * Devuelve la talla de la pila.
     *
     * @return int, la talla.
     */
    public int talla() {
        return talla;
    }
}
```

2.4 Colas

Una cola (queue en inglés) es una colección de datos del mismo tipo en la que el acceso se realiza siguiendo un criterio FIFO (First In First Out), es decir, el primer elemento que llega (que entra en la cola) es el primero en ser atendido (en ser eliminado de la cola).

En la vida real, las colas se utilizan muy a menudo como política de gestión de un modelo de negocio cliente-servidor. Por ejemplo, los usuarios de un comercio suelen hacer cola frente a las cajas a la hora de comprar un producto. Los ordenadores, a su vez, también gestionan muchos procesos mediante colas como, por ejemplo, la impresión de documentos.

El tipo de datos Cola presenta la siguiente interfaz de operaciones disponibles: crear una cola, añadir un nuevo elemento al final de la cola, eliminar el elemento que se encuentra a la cabeza de la cola, consultar (sin eliminar) el valor del dato que está el primero de la cola, preguntar si una cola está vacía y, por último, obtener el número de elementos de la cola. Siguiendo una aproximación similar a la de la clase Pila, este interfaz se implementa en Java por medio de:

- Un método constructor de objetos Cola.
- Sendos métodos modificadores para los procesos de encolar y desencolar.
- Varios métodos consultores que no alteran la estructura de una Cola.

```
package lineales;

import java.util.NoSuchElementException;

/**
 * Clase Colas: Cola de int. Implementación enlazada.
 *
 * @author
 * @version
```

```
 */
public class Colas {

    private NodoInt primero, ultimo;
    private int talla;

    /**
     * Crea una cola vacía.
     */
    public Colas() {
        primero = null;
        ultimo = null;
        talla = 0;
    }

    /**
     * Encola un nuevo elemento en la cola.
     *
     * @param int x, el elemento a encolar.
     */
    public void encolar(int x) {
        NodoInt nuevo = new NodoInt(x);
        if (ultimo != null) {
            ultimo.siguiente = nuevo;
        } else {
            primero = nuevo;
        }
        ultimo = nuevo;
        talla++;
    }

    /**
     * Desencola y devuelve el primer elemento de la cola.
     *
     * @return int, la cabeza de la cola antes de desencolar.
     * @throws NoSuchElementException si la cola está vacía.
     */
    public int desencolar() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        int x = primero.dato;
        primero = primero.siguiente;
        if (primero == null) {
            ultimo = null;
        }
        talla--;
        return x;
    }

    /**
     * Devuelve el primer elemento de la cola.
     *
     * @return int, la cabeza de la cola.
     * @throws NoSuchElementException si la cola está vacía.
     */
```

```
    public int primero() {
        if (talla == 0) {
            throw new NoSuchElementException("Cola vacía");
        }
        return primero.dato;

}
```