Programación II
Práctica 02-Parte a: Tipos Abstractos de Datos (TAD) Básicos

IMPORTANTE: Para todos los ejercicios se debe escribir el ***invariante de representación (IREP)*** antes de comenzar la implementación.

Como ejemplo un IREP de los Números naturales podría ser:

> *Las instancias validadas del TAD Nat, que representa a los números naturales son: Los números enteros positivos.*

Ejercicio1: Números naturales (TAD Nat)

En algunos casos se necesita modificar el comportamiento de TADs que ya existen.
Realizaremos una implementación de los números naturales(N) basándonos en Integer como el tipo soporte.

> Como Nat se define alrededor de Integer y semánticamente son similares, se dice que Nat ***envuelve*** (redefine) a Integer.

Esto se realiza principalmente, para modificar el comportamiento del tipo base, sin modificar al tipo base en sí mismo. En este caso, no queremos números negativos.

Especificación

```
Nat(Integer n){}        // Constructor. n≥0
sumar(Nat n){}
```

Se pide

a) **Implementar Nat**
b) **Escribir el invariante de representación**

Notas: Ocultamiento de información

Implementar también tostring() de manera de poder mostrar los resultados. Cualquier función o variable que utilice la clase salvo las pedidas en la implementación, deben ser privadas.

Ejercicio2: TAD Tupla

Muchas veces necesitamos una estructura de datos que relacione dos TADs. Para ello utilizaremos un par (o una tupla) de elementos dentro del **TAD Tupla**. Estos elementos serán de tipo T1 y T2.

Especificación (Tupla de T1, T2)

```
Tupla(T1 x, T2 y)   //Constructor
T1 obtenerX()  // devuelve X
T2 obtenerY(){}    // devuelve Y
Void establecerX(T1 x)   //da valor a X
Void establecerY(T2 y){}   //da valor a Y
```

Respetando esta especificación se pide:

a) Implementar el **TAD Tupla**.

b) Implementar una lista de coordenadas, que se representara mediante un

ArrayList de `Tupla<Integer,Integer>`

Ej: *ArrayList<`Tupla<Integer,Integer>`> coordenadas*;

c) Escribir el invariante de representación

Ejercicio 3 : GENERICS : TAD Conjunto<T>

a) Definir el **TAD Conjunto**, que se comporta como el conjunto de la teoría de conjuntos. No se puede utilizar **Set de Java** ni ninguna de sus subclases para implementarlo.
Es decir, no queremos envolver Set, queremos definirlo basado en otros tipos básicos.

Sugerimos utilizar la clase *Array(class Arrays)* para el nuevo Tad.

Especificación

```
Conjunto<T>(){}                    // Constructor1
Integer tamaño(){}
boolean agregar(T n) {}        // agrega si no existe
T dameUno( ){}                  // devuelve un elemento – lo quita – Si no hay
elementos devuelve null

Void union1(Conjunto<T> c){}        // union1: Destructiva

Conjunto<T> union2(Conjunto<T> c){} // union2: No debe  tener Aliasing!

Void interseccion1(Conjunto<T> c){} // interseccion1: Destructiva
```

```
Conjunto<T> interseccion2(Conjunto<T> c){} //interseccion2: No debe tener Aliasing!
```

Ejemplo:

```
Integer tamaño(){
      return conj.size();
}
```

- Para evitar Aliasing, "*union2*" e "*interseccion2*" deben devolver un nuevo conjunto, que no referencie al conjunto de la clase (this).

Nota1: **Encapsulamiento**

Siempre que sea posible, se deben utilizar las funciones de la clase en lugar de preguntar por sus variables internas o privadas (this). En este caso, utilizar *tamaño(),* en lugar de *this.vector.size()* siempre que sea posible.

Nota2: **Reutilización**: implementar unión/intersección intentando utilizar dameUno/agregar.

Siempre que sea posible, se deben reutilizar los métodos de la propia clase para implementar nuevos métodos.

b) Calcular la **complejidad** de

```
1. Void union(Conjunto<T> c){}            // union1: Destructiva
2. Conjunto<T> union2(Conjunto<T> c){}    // union2: No debe tener
   Aliasing!
3. Void interseccion(Conjunto<T> c){}     // interseccion 1: Destructiva
4. Conjunto<T> interseccion2(Conjunto<T> c){}
```

Asumiendo que:

El peor caso de agregar está en **O(n)**, donde **n** es el tamaño del conjunto más grande.

Para ello utilizar las siguientes variables:

- **n1 es el tamaño de *this***
- **n2 es el tamaño de *c***

Ejemplo: Unión

```
public void union(ConjInt<T> c) {
      for (int i=0; i<c.tamano(); i++){ //(1)
            agregar(c.dameUno); //(2)
      }
}
```

(1) O(n1) el ciclo se ejecuta n1 veces

(2) *dameUno* es O(1) y luego se ejecuta el *agregar* teniendo en cuenta que cada vez que se ejecuta deja un elemento más en ***this***, siempre considerando el peor caso ***(¿cuál es?)***

(3) ¿Cómo es la complejidad de ***union*** cuando n1 == n2, la que llamaremos n? Usar álgebra de órdenes para demostrarlo.

Ejercicio 4: GENERICS : TAD Diccionario<C, V>

Diccionario de entradas compuestas por clave y valor (o también significado) (C y V)
(MUY IMPORTANTE RESOLVERLO)

Un Diccionario es una generalización del concepto de conjunto, en la cual cada elemento quepertenece al conjunto (denominado clave) tiene asociado un valor:

- Los elementos del Diccionario son pares (o tuplas) (clave, valor).
- No pueden existir claves repetidas.
- Sin embargo, sí pueden existir significados repetidos.
- Los elementos se localizan mediante su clave.
- Dada una entrada, un valor puede no estar definido, una clave siempre debe estar definida

Especificación

Valores:
Colección de pares de elementos asociados como **Tipo C (Clave)** y **Tipo V (Valor)**, tal que los elementos Tipo C no son repetidos y no tienen ningún orden.

Interfaz:

```
Diccionario<C, V>(){}
// Constructor

Boolean agregar (C c, V v){}
// Agrega la entrada (c, v) al diccionario si la clave c no existe. O cambia
el valor por v si ya existe la clave (la pisa).

V obtener(C c){}
//Devuelve el valor que se corresponde con la clave c.

Boolean Pertenece(C n){}
// Devuelve verdadero si la clave c existe en alguna entrada del
diccionario.

Boolean eliminar (C c)
// Devuelve verdadero si pudo eliminar la entrada con clave c y Falso si no
existe.

V definicion( C c)
// Devuelve el valor de la clave c. Requiere que c pertenezca al
diccionario.

Integer tamaño()
// Devuelve la cantidad de claves o entradas del diccionario.

Boolean estaVacio()
// Dice si tiene o no entradas
```

Implementar el TAD Diccionario<C, V> sin utilizar la clase Map de Java

Ejercicio 5: Uso de las clases concretas de MAP - Class HashMap<C, V>

Se quiere conocer la frecuencia (cantidad de veces que aparecen) de cada palabra que componen un arreglo de Strings. También se quiere mostrar cuáles son y la posición en que aparece cada uno en el arreglo. Como se muestra en el ejemplo a continuación:

0	1	2	3	4	5	6	7	8	9
casa	hola	dia	hola	casa	hola	taza	dia	hola	casa

1)

casa	3
hola	4
dia	2
taza	1

3)

casa	**0, 4, 9**
hola	**1, 3, 5, 8**
dia	**2, 7**
taza	**6**

2)

casa	hola
dia	taza

Implementar la solución utilizando la clase HashMap de Java.

Algunas referencias de consulta:

https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/

http://www.lcc.uma.es/~jlleivao/algoritmos/t5.pdf