Programación II

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

Versión del 23/03/2022

IMPORTANTE: Para todos los ejercicios se debe escribir el ***invariante de representación (IREP)*** antes decomenzar 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, sinmodificar al tipo base en sí mismo. En este caso, no queremos números negativos.

Especificación

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

a) Implementar Nat

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*;

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** 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.

Especificacíon

```
Conjunto<T>(){}                    // Constructor1
Integer tamaño(){}
void Agregar(T n) {}

T iesimo(Interger indice){}   // indice< tamaño()

Void union(Conjunto<T> c){}         // union1: Destructiva
Conjunto<T> union2(Conjunto<T> c){}
                              // union2: No debe tener Aliasing!
```

```
Void interseccion(Conjunto<T> c){} // interseccion 1: Destructiva
Conjunto<T> interseccion2(Conjunto<T> c){}
```

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.

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 union/interseccion intentando utilizar iesimo/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; Union

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

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

(2) O(n1 + n2) porque iesimo está en O(n2) y luego se ejecuta el agregar en O(n1)

Total: O(n1) * O(n1 + n2) = O(n1 * (n1 + n2))

c) ¿Como queda la complejidad cuando n1 == n2?

Ejercicio 5: Diccionario de clave y significado (C y S) (MUY IMPORTANTE RESOLVERLO)

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

- Los elementos del Diccionario son pares (clave, significado).
- No pueden existir claves repetidas.
- Sin embargo, sí pueden existir significados repetidos.
- Los elementos se localizan mediante su clave.

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>(){}            // Constructor1
Boolean agregar (C, V v){}       //  Agrega la entrada (c, v ) al
                                    diccionario si la clave no existe. O
                                   cambia el valor s si ya existe.
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 no tiene entradas
```

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

Algunas referencias de consulta:

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

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