Programación II

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

Versión del 01/08/2015

Introducción

En las clases teóricas se estudiaron las ventajas que tienen los TADs. A continuación las repasamos:

Abstracción:

- Destacar los aspectos relevantes del objeto.
- Ignorar los aspectos irrelevantes del mismo

Abstracción funcional:

Crear procedimientos y funciones e invocarlos mediante un nombre donde se destaca qué hace la función y se ignora cómo lo hace. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar el resto de los detalles (el cómo).

Ocultamiento de información:

Ocultar *decisiones de diseño* en un programa susceptible de cambios con la idea de proteger a otras partes del código si éstos se producen.

Proteger una decisión de diseño supone proporcionar una interfaz estable que proteja el resto del programa de la implementación (susceptible de cambios).

Encapsulamiento:

Mantener todas las características, habilidades y responsabilidades de un objeto por separado.

Especificar comportamientos inesperados:

Una buena práctica de diseño es ser declarativo con los nombres (hacer lo que se dice) y ser coherente respecto de los efectos secundarios de nuestros.

Por ejemplo, avisar si tendremos aliasing. O si el Tad queda alterado de forma no deseable luego de cierta operación.

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 positivos

Ejercicio1: Números naturales (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 si mismo.

En este caso, no queremos números negativos.

Especificacíon

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

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 (obligatorio): Tupla

Muchas veces necesitamos relacionar dos TAD en una estructura.

Por las propiedades de los TAD, es muy diferente tener una estructura que dos variables independientes.

Para ellos utilizaremos un par (o tupla) de elementos dentro del TAD Tupla.

Especificacíon (Tupla de T1,T2)

```
TuplaIntegerInteger (T1 x, T2 y){}
T1 getX(){}
T2 getY(){}
Void setX(T1 x){}
Void setY(T2 y){}
```

a) Implementar el TAD Tupla
b) Implementar una lista de coordenadas, que se representara mediante un ArrayList de `Tupla<Integer,Integer>`
    a. Ej: ArrayList<`Tupla<Integer,Integer>`> coordenadas

Ejercicio3 (obligatorio): Conjunto de 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 i) {}
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

```
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){}
```

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 esta 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?

Ejercicio4 (obligatorio*): Pila de T

***La entrega de los ejercicios a) y b) es obligatoria. La parte c) es opcional.**

a)Implementar la Pila de enteros, sin envolver el tipo Stack.

Especificación (operaciones básicas)

```
PilaInt<T>(){}                // Constructor1
Boolean vacia()
Void agregar (T i){}    // Agrega en la cima.
T quitar(){}            // Quita de la cima.
T cima(){}              // Devuelve la cima sin quitarla.
```

b) Implementar los siguiente métodos utilizando solamente las operaciones básicas

```
T minimo(){}      // obtiene y quita el mínimo de la pila*

void ordenar(){}
void MezclarOrdenar(PilaInt pila2){}
Pila<T> MezclarOrdenar(Pila<T> pila2){}
Pila<T> OrdenarMezclar(Pila<T> pila2){}

*Asumir que T tiene un método Integer compareTo(T e2)
Que devuelve -1 cuando e2 < this

Ej
      T t1 = new T("casa")
      T t2 = new T("perro")

      t1.compareTo(t2) devuelve -1


ordenar()
Deja la pila ordenada.
Ayuda: Recorrer la pila utilizando mínimo y una pila auxiliar donde dejar
el mínimo en cada operación.
```

MezclarOrdenar()
Mezcla pila2 con this y luego ordena this

PilaInt MezclarOrdenar(PilaExtendida pila2)
Hace lo mismo, pero devuelve una referencia a otra pila (no genera aliasing).

```
Pila<T>  OrdenarMezclar (Pila<T>  pila2)
Ordena this y pila2. Luego hace un merge de las dos pilas de manera
inteligente, recorriendo una sola vez cada pila.

Ayuda:
Como ambas pilas están ordenadas se puede recorrer las dos pilas con dos
punteros, tomando los elementos según correspondan.
```

c)Calcular el Orden de complejidad de los métodos
odenar()
MezclarOrdenar()
OrdenarMezclar()

¿Cuál tarda menos, MezclarOrdenar() o OrdenarMezclar() ?

Ejercicio5 (obligatorio): Diccionario de clave y significado (C y S)

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.

Especificacíon

```
Dicc<C,S>(){}                  // Constructor1
Void agregar (C i, S s){}      //
S obtener(C i){}          //
Boolean Pertenece(C i){}             //
```

Implementar el TAD Dicc<C,S> sin utilizar Map.

Bibliografía:

[Cormen1990]:http://www.amazon.com/exec/obidos/ISBN=0262031418/none01A/

[Cormen2001]: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Introduction to Algorithms. Sección 31.8: "Primality testing", pp.887–896. MIT Press & McGraw-Hill, 2a edición,2001. (ISBN 0-262-03293-7.)