Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI2692 – Laboratorio de Algoritmos y Estructuras II
Enero – Marzo 2010

Proyecto: Variantes de un Consultor Ortográfico

(Semana 6 → Semana 12)

El objetivo de este proyecto de laboratorio es el construir un par de variantes de implementación de un mismo Tipo Abstracto de Datos (TAD). Además de las variantes del TAD, se requiere construir un programa cliente que permita manipular ambas variantes; una de las opciones del cliente permitirá hacer una pequeña comparación de rendimiento entre ellas. Toda la especificación e implementación será realizada en el lenguaje de programación Java/JML.

0. El tipo abstracto de datos (TAD) Consultor Ortográfico

El TAD a considerar en este proyecto permite realizar consultas de ortografía usando prefijos de palabras. Un prefijo será definido como las *k* primeras letras de una palabra de longitud *n*, para cualquier $0 \leq k \leq n$. Por ejemplo, *algo* y *escribir* son prefijos de las palabras *algoritmo* y *escribir*, respectivamente.

Definiremos a nuestro consultor ortográfico como un programa que permite identificar una o más palabras que tenga (n) como prefijo una cadena de caracteres dada. Se tiene que el consultor ortográfico posee un conjunto de palabras de consulta que llamaremos *vocabulario*.

A continuación presentaremos una especificación para el TAD Consultor Ortográfico, al cual llamaremos *ConsultOrt,* cuyo vocabulario es modelado como un conjunto de "*Strings*" (el lenguaje matemático que utilizaremos para manejar el modelo abstracto de representación será el presentado en el capítulo 9 de [Morgan 94].

Veamos entonces nuestra especificación con modelo abstracto:

Especificación A **de TAD** ConsultOrt

Modelo de Representación
var vocabulario : Set String

Invariante de Representación
$(\forall x : x \in vocabulario : bf(x))$

Operaciones
.
.
.

Fin TAD

Note que usamos un predicado llamado *bf* que nos permite decidir si un *String x* está o no bien formado. Decimos que un *String x* está bien formado si *x* no es vacío y si cada caracter que forma parte de *x* es una letra minúscula del alfabeto *[a..z]*, exceptuando el caracter ñ.

Formalmente, *bf : String* → *boolean*, tal que

$$bf(p) = p \neq \text{“”} \wedge (\forall i : 0 \leq i < p.\text{length} : p(i) \in \text{Alfa})$$

Donde:

- p(i) : es el i-ésimo caracter del *String p*

- $\text{Alfa} = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$

Veamos ahora las operaciones de nuestro TAD:

Función crearVacio: Constructor para crear un nuevo consultor ortográfico sin información alguna. Esta función devuelve el consultor ortográfico vacío.

fun crearVacio() → ConsultOrt
{**pre**: true}
{**post**: crearVacio.vocabulario = $\emptyset$ }

Método agregar: Permite registrar en el parámetro implícito *this* una nueva palabra al vocabulario del consultor ortográfico. Recibe la palabra *p* a registrar.

meth agregar (**in** p : String)
{**pre**: $bf(p) \wedge p \notin \text{this.vocabulario}$ }
{**post**: $\text{this.vocabulario} = \text{this}_0.\text{vocabulario} \cup \{p\}$ }
{**exc**: $\neg bf(p) \sim > \text{ExcepcionPalabraNoBienFormada}(p)$
$p \in \text{this.vocabulario} \sim > \text{ExcepcionPalabraYaRegistrada}(p)$ }

Note que hemos agregado el componente e**xc** a la especificación, el cual nos permite señalar comportamientos “excepcionales” de la operación; entendiendo por comportamiento “no-excepcional” aquel señalado en el par **pre**/**post** convencional. El contenido de esta cláusula indica que si en el estado inicial se cumple la “precondición excepcional” $\neg bf(p)$ o $p \in \text{this.vocbulario}$, la operación debe terminar lanzando una excepción de tipo *ExcepcionPalabraNoBienFormada* o *ExcepcionPalabraYaRegistrada*, respectivamente; ambas con valor *p*. Note que la precondición convencional y la precondición excepcional son disjuntas, y que entre ambas cubren todos los casos

posibles.

Esta posibilidad de terminación excepcional podemos manejarla en este proyecto gracias a que trabajaremos con el lenguaje Java, el cual permite el uso de excepciones.

Método funcional consultarPorPrefijo: Devuelve un arreglo de "*Strings*" con todas las palabras del vocabulario que tienen en común el prefijo *pr*. No modifica el registro *this*.

fmeth consultarPorPrefijo (**in** pr : String) → array[] of String
{**pre**: bf(pr)}
{**post**: $this = this_0$
$\wedge$
$(\forall i, j : 0 \leq i, j < \text{consultarPorPrefijo.length} \wedge i \neq j : \text{consultarPorPrefijo}[i] \neq \text{consultarPorPrefijo}[j])$
$\wedge$
$(\forall i : 0 \leq i < \text{consultarPorPrefijo.length} : \text{consultarPorPrefijo}[i] \in \text{this.vocabulario})$
$\wedge$
$(\forall i : 0 \leq i < \text{consultarPorPrefijo.length} : \text{ipf}(pr, \text{consultarPorPrefijo}[i]))$
}
{**exc:** $\neg \text{bf}(p) \sim> \text{ExcepcionPalabraNoBienFormada}(p)$ }

Note que hemos usado un nuevo predicado llamado *ipf(p,q),* el cual permite decidir si un *String p* es prefijo de otro *String q*. Donde *p* y *q* son palabras bien formadas.

Formalmente, *ipf : String x String → boolean*, tal que

$$\text{ipf}(p, q) = p.\text{length} \leq q.\text{length} \wedge (\forall i : 0 \leq i < p.\text{length} : q(i) = p(i))$$

Donde:
- p(i) y q(i) son es el i-esimo caracter del *String p* y el *String q,* respectivamente.

Método funcional prefijoMasLargo: Devuelve un "*String"* que contiene el prefijo más largo que puede extraerse de *pl*, para el cual existe al menos una palabra en *this.vocabulario* con ese prefijo. No modifica el registro *this*.

fmeth prefijoMasLargo (**in** pl : String) → array[] of String
{**pre**: $\text{bf}(pl)$ }
{**post**: $this = this_0 \wedge \text{ipf}(\text{prefijoMasLargo}, pl) \wedge (\forall s : s \in \text{this.vocabulario} : \text{ipf}(s, pl) \Rightarrow \text{ipf}(s, \text{prefijoMasLargo}))$ }
{**exc:** $\neg \text{bf}(p) \sim> \text{ExcepcionPalabraNoBienFormada}(p)$ }

Método funcional iterador: Devuelve un iterador que permite recorrer todas las palabras registradas en el vocabulario del consultor ortográfico *this*. No modifica *this*. En una sección posterior se explicará

qué es un iterador; en este momento, basta con pensar en el iterador de nuestro interés como una secuencia de palabras del vocabulario del consultor ortográfico.

fmeth iterador() → Iterador(String)
{**pre**: true }
{**post**: $this = this_0 \wedge$ iterador contiene la secuencia ordenada y sin repeticiones, formada por todas las palabras del conjunto $\{s | s \in this.vocabulario\}$ }

Debemos explicar qué es un método funcional. Así como un método, a secas, es un procedimiento con parámetro implícito de entrada-salida *this*, un método funcional es una función con parámetro implícito *this*. Esto es, se mantiene la noción del parámetro extra implícito, pero a diferencia de los métodos a secas, por ser función se devuelve un valor resultado. En el caso de Gacela, no se permiten métodos funcionales, pues todos los parámetros de una función en Gacela deben ser de entrada y por tanto, no modificables. Java si permite método funcionales y, del hecho de que el parámetro implícito sea potencialmente modificable, surge la necesidad de señalar en la postcondición de nuestro método funcional iterador que al final se cumplirá $this = this_0$.

El tipo resultado de nuestro método funcional, *Iterador(String)*, está construido a partir del tipo *String* y el tipo genérico *Iterator(T)*. Este último será definido en la siguiente sección. Hemos utilizado el término "iterator" en inglés, tanto para el nombre del tipo como para el nombre del método, a efectos de seguir convenciones del lenguaje Java.

1. El tipo abstracto de datos (TAD) Iterador

Un iterador, en términos abstractos, simplemente provee operaciones para recorrer una secuencia de principio a fin. En sus posibles implementaciones concretas, la historia es diferente, pero de esto hablaremos en otro momento.

Por ahora, simplemente veamos una especificación con modelo abstracto para el TAD de iteradores:

Especificación A **de TAD** Iterator(T)

Modelo de Representación
var s : seq T

Invariante de Representación
true

Operaciones
fmeth hasNext() → boolean
{**pre**: true }
{**post**: $hasNext \equiv (this.s \neq <>) \wedge this = this_0$ }
fmeth next() → T
{**pre**: $this.s \neq <>$ }
{**post**: $next = hd\ this_0.s \wedge this.s = tl\ this_0.s$ }
{**exc**: $this.s = <> \sim >$ NoSuchElementException }

Fin TAD

Note que el modelo abstracto es una secuencia. La notación y operaciones utilizadas para secuencias para modelos abstractos están en [Morgan 94].

El modelo *s* representa la secuencia de elementos por recorrer, esto es, la secuencia de elementos sobre los cuales "se iterará". El método funcional *hasNext* sirve para indicar si aún existen elementos por recorrer y no altera el iterador, de allí que el resultado devuelto sea el valor booleano de la desigualdad $this.s \neq <>$. Por su parte, el método funcional *next* sirve para tomar el primer elemento del iterador y avanzar hacia el siguiente, de allí que el resultado devuelto sea el primer elemento o cabeza de la secuencia (el nombre de la operación "*hd*" se refiere a "head") y la secuencia sea alterada quitándole la cabeza y quedándose con el resto o cola (el nombre de la operación "*tl*" se refiere a "tail"). Otra formulación equivalente de la postcondición del método funcional *next* habría sido $<next> ++ this.s = this_0.s$.

Así como escogimos el nombre "Iterator" para este TAD por convenciones del lenguaje Java, los nombre de los métodos y el nombre de la excepción utilizados también corresponden a Java.

Note que no hemos especificado función constructora para el tipo. Esto se debe a que se permite que cada implementación con modelo concreto determine qué tipo de constructor le conviene utilizar, en lugar de prefijarlo desde la especificación original con modelo abstracto.

La noción de iterador es tratada en el capítulo 6 de [Liskov & Guttag 01] de manera muy completa y clara. Se explica lo que son iteradores, cómo usarlos, cómo implementarlos en Java, e incluso se definen los invariantes de representación y relaciones de acoplamiento (funciones de abstracción, en terminología Liskov) de los modelos concretos presentados como ejemplos contra un modelo abstracto como el dado por nosotros acá. Se recomienda fuertemente la lectura del referido capítulo; sin esa lectura es poco probable aprender a manejar iteradores adecuadamente.

2. Las variantes de implementación

Como fue indicado anteriormente, se desea trabajar un par de variantes de implementación del TAD *ConsultOrt*. En esta sección se describen las dos variantes deseadas, especificando parcialmente los modelos concretos correspondientes.

3.0. Primer modelo concreto: arreglos

En nuestra primera variante de implementación, trabajaremos con el modelo concreto arreglo.

Especificación B **de TAD** ConsultOrt, **refinamiento de** A

Modelo de Representación
var tam : int;
var va : array[] of String;

...
Fin TAD

El arreglo *va*, por "v"ocabulario "a"rreglo, tiene una posición para cada una de las palabras que conforman el vocabulario. Note que el arreglo es declarado sin un tamano preestablecido. La variable *tam* guarda la cantidad de palabras que tiene actualmente el vocabulario.

En el invariante de representación se deberá exigir que todas las cadenas de *va* sean diferentes y que estén ordenadas lexicográficamente:

Invariante de Representación

$$(\forall x, y : 0 \leq x, y < tam \wedge x \neq y : va[x] \neq va[y])$$
$$\wedge$$
$$(\forall x : 0 \leq x < tam - 1 : va[x] < va[x+1])$$
$$\wedge$$
$$0 \leq tam \leq va.length$$

Es importante senalar que la comparación de *Strings* se hace lexicográficamente. Esta comparación está basada en el valor "*Unicode*" (o la codificación correspondiente) de cada caracter en los *Strings* a comparar. Así, si s_1 y s_2 son *Strings*, entonces $s_1 \neq s_2$ si s_1 es lexicográficamente menor o mayor que s_2 . En Java, la clase *String* implementa un método llamado *compareTo,* el cual recibe un tipo *String* como argumento y devuelve un tipo *int*, para comparar dos *Strings* de acuerdo al orden lexicográfico usando codificación "*Unicode*" . Por ejemplo, si *s1* y *s2* son *Strings*, entonces *s1.compareTo(s2)* devuelve: 0 si *s1* es igual a *s2 (*equivalente a la instrucción *s1.equals(s2))*; un valor entero menor que cero si *s1* es lexicográficamente menor que *s2*; y un valor entero mayor que 0 si *s1* es lexicográficamente mayor que *s2*.

Por último, como parte anexa a este enunciado, le proporcionaremos (vía Wiki de Gacela, página oficial del curso) la interfaz Java llamada *ConsultOrt*, correspondiente el modelo abstracto del TAD *ConsultOrt,* y una implementación parcial de la ésta primera variante llamada *ConsultOrtArreglos*, que incluye lo siguiente:

- Modelo de representación
- Invariante de representación
- Constructor de la clase (correspondiente a la función crear de la Especificación A)
- Método agregar

La relación de acoplamiento, las implementaciones del resto de los métodos y los invariantes y variantes de iteración deben ser escritos por Ud. para completar ésta primera variante del TAD *ConsultOrt*.

Como podrá observar en el archivo que implementa la primera variante de nuestro TAD, el constructor de la clase *ConsultOrtArreglos* exige que, inicialmente, la longitud de *this.va* sea igual a uno; *tam* igual a cero nos dice que el vocabulario del consultor ortográfico no tiene elementos (o está vacío). Note además, que para este modelo concreto, el invariante de representación exige que todos los elementos de *this.va* estén ordenados; en nuestro caso, por tratarse de *Strings*, el orden es lexicográfico.

Un aspecto importante que debe exigirse al método agregar es que duplique la longitud del arreglo *this.va* cada vez que la variable *this.tam* sea igual a *this.va.length*. Cuando se cumple esta condición, creamos un arreglo auxiliar *vtemp* de longitud *2 * this.va.length*, al que le copiamos los elementos de *this.va* en el mismo orden. Luego a la referencia *this.va* le asignamos la referencia de *vtemp* para asegurarnos que puedan agregarse nuevos elementos a *this.va*. De esta manera podemos lidiar un poco con las limitaciones que tienen los arreglos en cuanto a su necesidad de definir sus tamanos estáticamente.

En cuanto a la exigencia del invariante de representación de que los elementos de *this.va* estén ordenados lexicográficamente, exige que al agregar un elemento a *this.va* se haga de manera ordenada. Para ello, utilizamos una versión limitada de búsqueda binaria, para arreglos sin repeticiones, que devuelve la posición *posi* donde el nuevo elemento *p* debe almacenarse en *this.va*. Esto último requiere que todos los elementos a la derecha de *this.va[posi]* sean desplazados una posición en ese sentido. Luego el nuevo elemento *p* es insertado de manera ordenada en el arreglo *this.va*.

Finalmente, note que no le hemos proporcionado el código de la función *bb,* que devuelve la posición donde quedará insertado el nuevo elemento *p*. Tampoco le hemos proporcionado la función booleana *bf* que decide si una palabra está bien formada o no de acuerdo a la definición del predicado *bf* especificado arriba, y tampoco la función booleana *ipf* que decide si una palabra dada es prefijo de otra. Es importante que *bb, bf* e *ipf* sean declarados métodos puros.

3.1. Segundo modelo concreto: tries

En la segunda variante de implementación, utilizaremos un tipo especial de árboles llamados *tries*. Un *trie* es un árbol de prefijos, construido a partir de las letras minúsculas de un alfabeto. Específicamente, un *trie* es una estructura de árbol en la que:

- La raíz del árbol representa la cadena vacía.
- Un nodo puede tener tantos hijos como letras minúsculas existen. Cada rama en el árbol está etiquetada con una de esas letras.
- La sucesión de etiquetas desde la raíz hasta un nodo cualquiera representa un prefijo de palabra, que puede ser o no una palabra completa.

Con nuestro segundo modelo concreto, la especificación luciría como sigue:

Especificación ℂ **de TAD** ConsultOrt, **refinamiento de** 𝔸

Modelo de Representación
var vt : Trie;

.
.
.

Fin TAD

siendo *Trie* un tipo concreto correspondiente al siguiente tipo algebraico-libre:

tipo libre Trie = tvac | nodo(boolean, array[0..26] of Trie)

Así, en el árbol *vt*, por "v"ocabulario "t"rie, cada nodo es vacío (tvac) o contiene alguna información representada por un par <b,a>, tal que b es un booleano y a es un arreglo de 26 elementos de tipo *Trie*.

En el invariante de representación se deberá exigir que todos los caminos en *vt*, desde la raíz a las hojas (nodos en los que todos los elementos del arreglo son tries vacíos, tvac), conduzcan a prefijos de palabras completas.

Invariante de Representación
ok(vt)

donde *ok* es una función que se define inductivamente sobre la estructura Trie:

ok : Trie → boolean
con cláusulas
ok(tvac) = true
ok(b, a) = $((\forall i : 0 \le i < 26 : a[i] = \text{tvac} \Rightarrow b) \wedge (\forall i : 0 \le i < 26 : \text{ok}(a[i])))$

Para la relación de acoplamiento, es necesario definir una función inductiva sobre la estructura que permita recorrerla adecuadamente de manera que puedan obtenerse todas las palabras representadas en el árbol de prefijos:

Relación de acoplamiento
vocabulario = extr(vt)

extr : Trie → Set String
con cláusulas
extr(tvac) = $\emptyset$
extr(nodo(b,a)) = $\{i, x : 0 \le i < 26 \wedge x \in \text{extr}(a[i]) : \text{char}(i) \circ x\} \cup B$

donde

$$B = \begin{cases} \emptyset \text{ si } \neg b \\ \{\lambda\} \text{ si } b \end{cases}$$

λ : Es la cadena (String) vacía

$\circ$: Define la operación de concatenación de cadenas (Strings)

char : int → Alfa, tal que
char(0) = 'a'
char(1) = 'b'
...
char(25) = 'z'

En cuanto a la construcción de este modelo concreto en Java, al igual que se usan las clases para construir TADs, son también las clases el mecanismo de Java que puede ser utilizados para construir registros ("record"s). Para evitar confusión en cuanto a lo que se desea, y las posibles maneras de lograrlo en Java, indicaremos explícitamente a continuación parte de la clase de Java que deberá ser construida:

```
class ConsultOrtTriesArreglos implements ConsultOrt {

        final static int TAMINIC = 26;
        private /*@ spec_public @*/ NodoTries trie;

        private static class NodoTries{

                public NodoTries lista[];
                public boolean esPalabra;

                NodoTries(){
                        this.lista = new Nodo[TAMINIC];
                        this.esPalabra = false;
                }
        }
}
```

Note que la clase con atributos públicos utilizada para construir el registro es la clase interna privada de la clase *ConsultOrtTriesArreglos*, pues estos registros son sólo para ser manejados dentro de esta clase. *ConsultOrt* es la interfaz correspondiente a la especificación del TAD, la cual debe ser implementada por esta clase.

Igual que para la primera variante, le proporcionamos (vía Wiki de Gacela, página oficial del curso) una implementación parcial de la ésta segunda variante llamada *ConsultOrtTriesArreglos*. Al igual que para esa primera parte, Ud. debe especificar e implementar las funciones *bf* e *ipf*; esta vez de acuerdo a las sugerencias descritas en el Wiki. Además, note que tampoco le hemos proporcionado las funciones recursivas *ok* y *extr*, las cuales son usadas, respectivamente, para establecer el invariante de representación y la relación de acoplamiento. Ud. debe especificar e implementar estas funciones como un método puro-modelo, las cuales formaran parte de las especificaciones en modelo concreto de nuestro TAD.

3.2. Los iteradores concretos

Para cada uno de los dos modelos concretos descritos, Ud. debe proveer una implementación del TAD *Iterator* , según lo requerido por el TAD *ConsultOrt*. Estos iteradores concretos deben ser construidos, tal como lo indica el capítulo 6 de [Liskov & Guttag 01], como clases privadas internas de las clases *ConsultOrtArreglos* y *ConsultOrtTries*. Cada una de esas implementaciones concretas proveerá el constructor que Ud. considere conveniente.

4. El programa cliente

Contando con las dos implementaciones del TAD *ConsultOrt*, que en Java serán las dos clases antes descritas, cada una de las cuales implementa la interfaz *ConsultOrt*, Ud. desarrollará un programa cliente de ellas para el usuario final.

El programa inicia en una iteración de menú con las siguientes opciones: las 4 primeras correspondientes a las operaciones básicas sobre el TAD (crear un consultor ortográfico, agregar por teclado palabras al vocabulario, consultar por prefijos y generar el prefijo más largo válido); una 5ta opción para cargar un vocabulario completo desde un archivo; una 6ta opción para listar todas las palabras del vocabulario del consultor ortográfico, en la cual Ud. hará uso de sus propios iteradotes; y salir del programa que sería la 7ma y última opción.

En la 1ra opción, para creación de un consultor ortográfico, se debe preguntar al usuario con cuál variante de implementación desea que manipular el consultor ortográfico: arreglos o tries. La variante de implementación con que haya nacido el consultor ortográfico a consultar determinará la variante de implementación en uso. Por lo tanto, en las opciones de consulta (no de creación), no tendrá sentido preguntar al usuario con cuál implementación se debe trabajar. De hecho, el procedimiento cliente que Ud. cree para implementar cada una de estas opciones deberá ser independiente de la variante de implementación en uso, aprovechando el "polimorfismo" logrado mediante el uso de la interfaz *ConsultOrt*. Esto quiere decir lo siguiente: si Ud. implementa estas opciones adecuadamente haciendo uso sólo de la interfaz *ConsultOrt* y sus operaciones, sin utilizar explícitamente, para nada, las clases *ConsultOrtArreglos* y *ConsultOrtTries*, el procesador del lenguaje Java se encargará de direccionar la ejecución hacia la variante de implementación en uso, sin intervención de Ud. A esto se refiere el "polimorfismo" ofrecido por Java para Ud.

En la 2da opción, para la agregación de palabras al vocabulario del consultor ortográfico, se le debe pedir al usuario cuántas palabras desea agregar en forma "manual". Luego, el programa debe pedir al usuario cada una de las palabras e insertarlas en la estructura de datos correspondiente (primera o segunda variante del consultor ortográfico).

En la 5ta opción, Ud. le preguntará al usuario el nombre del archivo de texto que contiene las palabras que se registrarán en el vocabulario del consultor ortográfico. Este archivo contendrá una lista de *Strings* organizadas en una columna (cada línea es un *String*). Se considera válido a aquel *String* que esté bien formado (ver definición *bf* arriba). Los *Strings* que no estén bien formados serán ignorados.

Ud. deberá medir la cantidad de tiempo que toma cargar el vocabulario desde un archivo, e indicar al usuario el resultado de esta medida. Esto le permitará hacer una pequena comparación de rendimiento, esto es, de eficiencia entre las dos implementaciones del TAD ConsultOrt constridas.

La 6ta opción debe ser implementada por Ud. Con un procedimiento cliente, valiéndose del método *iterator* del TAD *ConsultOrt* y de las operaciones del TAD *Iterator*.

En relación con las excepciones que pueden ser lanzadas desde las operaciones de los TADs utilizados, no se deben permitir que éstas alcancen al usuario. Esto es, su programa cliente debe atrapar todas estas excepciones para presentar los errores amigablemente al usuario, sin permitir que el programa cliente aborte como consecuencia de éstas excepciones.

4. Entrega de avances y entrega final

Durante el desarrollo de este proyecto, Ud. realizará tres entregas: dos primeros avances parciales y una entrega final con todo.

Avance I - Lunes de semana 8

En este primer avance, Ud. trabajará sólo en parte la primera variante de implementación del TAD *ConsulOrt* con arreglos. Esto le permitirá familiarizarse con el problema y con la implementación de TADs en Java.

Específicamente, en este avance I, Ud. deberá entregar:

- Una versión parcial de la primera variante de implementación del TAD *ConsulOrt*, con las operaciones *crearVacio* (que en Java sería un constructor), *agregar* (solo para agregar por teclado), y *consultarPorPrefijo.*

- Una versión parcial del programa cliente que tenga activas la 1ra opción sólo para la variante de implementación con arreglos, y las opciones 2da y 3ra del menú.

Avance II - Lunes de semana 10

En este segundo avance, Ud. deberá completar la primera variante de implementación del TAD *ConsulOrt* tomando en cuenta las consideraciones de desarrollo y eficiencia descritas en el Wiki. También debe iniciar el desarrollo de la segunda.

Específicamente, en este avance II, Ud. deberá entregar:

- Una versión de la primera variante del TAD *ConsulOrt* que debe estar ya completa, salvo posiblemente por el método *iterator.*

- Una versión de la segunda variante del TAD *ConsulOrt* construida sólo parcialmente, de manera similar a la construcción parcial de la primera variante en el avance I. Esto es: la implementación de las operaciones *crearVacio* (que en Java sería un constructor), *agregarPorTeclado y consultarPorPrefijo*.

- Una versión completa del programa cliente, teniendo activas todas las opciones del menú, salvo posiblemente, para el uso de alguno de los iteratores.

- Una implementación del método *iterator* en alguna de las dos variantes del TAD *ConsulOrt*. Ud. puede escoger si prefiere empezar implementando este método en la primera o en la segunda variante. Su escogencia determinará con cual de las dos variantes funcionará la opción 6ta del menú del cliente en este avance.

Entrega Final – Lunes de semana 12

Esta última entrega incluye la versión final del programa cliente, con todas las opciones activas, lo cual requiere por supuesto que todas las variantes de implementación de todos los TADs hayan sido completadas adecuadamente. El código de todas las clases e interfaces utilizadas en su programa debe ser entregado tanto en un CD como en papel, y también debe ser entregado un informe que describa el proceso de desarrollo y las preguntas adicionales que fueron dejadas en este enunciado.

El CD que Ud. entregará debe contener una versión operativa de todas las clases que conforman el programa, y deberá estar debidamente identicado con el nombre de los integrantes del equipo. También se requiere un listado sobre papel de todas las clases e interfaces utilizadas en el programa.

En relación con el informe, éste deberá seguir un formato similar al que usualmente se da como guía en el laboratorio de Algoritmos y Estructuras I. Este deberá contener las siguientes secciones:

- Portada, la cual debe contener:
    - Arriba a la izquierda: el nombre de la universidad, la carrera y la asignatura.
    - Centrado: el nombre del proyecto.
    - Abajo a la izquierda: nombre del profesor del laboratorio.
    - Abajo a la derecha: nombre y carné de los integrantes del equipo.

- Introducción:
    - Breve descripción del problema atacado en el proyecto.
    - Descripción del contenido del resto del informe.

- Diseño:

    - Especificación completa de las variantes de implementación del TAD *ConsultOrt*, exceptuando la especificación de las operaciones. Esto es: modelo de representación, invariante de representación, y relación de acoplamiento. Note que esto corresponde a las especificaciones de diseño de las estructuras de datos utilizadas para el TAD *ConsultOrt*, las cuales fueron dadas sólo parcialmente en este enunciado.

    - Especificación de las dos variantes de implementación del TAD *Iterator* , asociadas a las dos variantes de *ConsultOrt*, exceptuando al igual que antes la especificación de las operaciones. Se debe incluir ambos modelos concretos de representación, con sus invariantes de representación y relaciones de acoplamiento con el modelo abstracto de la especificación A dada en este enunciado. Note que estos modelos concretos de *Iterator* están asociados, al igual que en la implementación, con los modelos concretos de *ConsultOrt*.

    - Especificación de cualquier otra estructura de datos que Ud. haya utilizado en el proyecto.

    - Decisiones de diseño tomadas por Ud. en relación con cualquier aspecto del proyecto, que Ud. considere conveniente explicar en el informe.

- Detalles de implementación:
    - Peculiaridades relacionadas con el paso de parámetros en métodos de Java, que pudiesen diferir de lo diseñado o deseado por Ud.
    - Cualquier otro comentario referente a peculiaridades de Java.

- Estado actual:
    - Operatividad del programa: señalar si funciona perfectamente o no; en caso negativo, describir las anomalías.
    - Manual de operación: nombre de los archivos correspondientes a todas las clases e interfaces que conforman el programa, nombre del archivo a ejecutar (esto es: la clase con el programa principal), y modo de operar el programa.

- Conclusiones:
    - Experiencias adquiridas.
    - Dificultades encontradas durante el desarrollo.
    - Recomendaciones.

- Bibliografía: libros, revistas o cualquier otro recurso consultado (que puede incluso ser conversaciones consultas con amigos o expertos). Una referencia bibliográfica debe incluir la siguiente información:
    - Libro: autores, nombre, editorial y año.
    - Artículo: autores, nombre, revista, volumen, número y año.
    - Página web: autores, nombre, URL, fecha de la última visita.
    - Conversaciones personales o consultas: interlocutor y fecha.

Esta entrega final será realizada el día lunes de la semana 12 a las 3:00pm, en la secretaría del Departamento de Computación y Tecnología de la Información (MYS216). En la fecha y la hora referidas, Ud. entregará el CD, el listado del programa y el informe dentro de un sobre debidamente identificado (nombre y carné de los integrantes del equipo, y nombre del proyecto). En la clase de laboratorio del bloque B (jueves) de la semana 12, tanto para la sección del bloque A como el bloque B, se le indicará si se requiere un proceso de revisión y prueba del proyecto por parte de su profesor de laboratorio con el equipo programador presente.

Es importante que el equipo trabaje de manera integrada. Es posible que durante la revisión del funcionamiento del programa se realice un breve interrogatorio individual a cada miembro del equipo.

5. Comentarios Finales

- Cualquier error que a posteriori sea hallado en este enunciado, así como cualquier otro tipo de observación adicional sobre el desarrollo del proyecto, serán publicados como fe de erratas en el wiki del curso.

- Nota importante: El proyecto es un trabajo en equipo de dos personas. Si bien lleva más de dos semanas, se considera similar a un examen en cuanto a que no puede haber intercambios contrarios a la ética con otros equipos.