Proyecto final: Estructura de Datos

Luis Alfredo Rodriguez
Santiago Collantes
Gabriella Troncoso

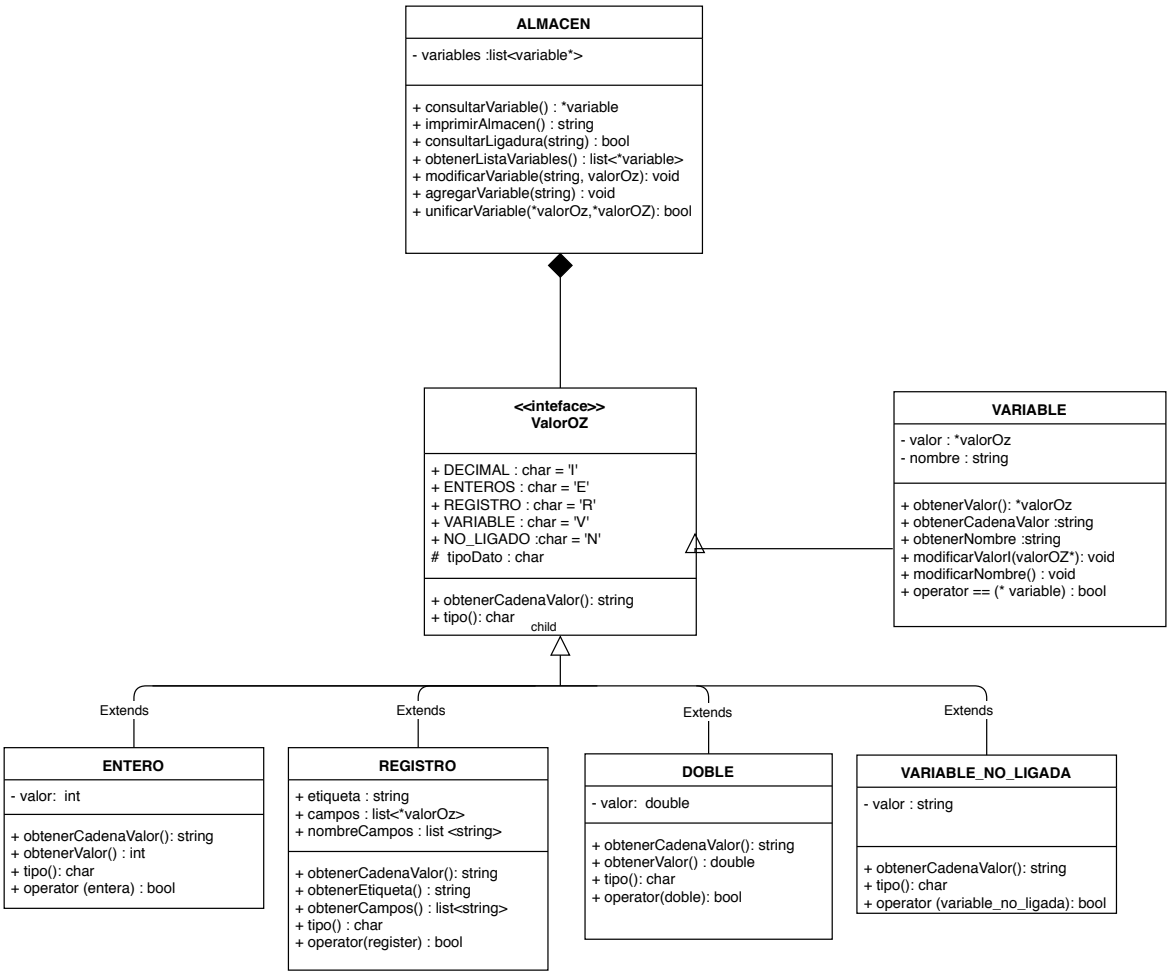
29 de Mayo 2019

1. Introducción

En el siguiente documento se hará un resumen explicativo del proyecto final para el curso de Estructura de Datos. El proyecto se basa en el lenguage ***Oz***, el cual utiliza una serie de complejas estructuras de datos para ser interpretado e implementado. ***Oz*** posee muchos atributos que lo caracterizan y que lo vuelven un lenguage interesante, sin embargo, para propositos del curso, en este proyecto solo se tendrán en cuenta los tipos basicos de datos de *Oz*(numeros y registros), asi mismo como el operador =, el cual permite asociar a una variable con un valor en un proceso denominado *unificación*. Otra caracteristica importante de ***Oz*** es que sus variables solo pueden ser asignadas una vez, para lo cual se utiliza una estructura especial llamada *almacen de asignación unica*.

El objetivo del proyecto final es implementar algunas de las estructuras utilizadas por un interpretador, mas especificamente las mencionadas anteriormente. Por lo tanto se busca poder crear la estructura de un *almacen de asignación unica* en el cual se puedan almacenar y ligar variables con numeros o registros, asi mismo como poder definir e implementar la *unificación* de variables. Para dichos propositos, en este proyecto se tomó el paradigma de Promagramación Orientada a Objetos (POO)con lo cual se realizaron diferentes clases que, relacionadas entre s, constituyen la implementación y definición del *almacen de asignación unica*.
Entre las clases mencionadas anteriormente cabe destacar la creación de la clase *ValorOz*, la cual representa los tipos de dato basico presentes en la implementación de este proyecto: numeros(enteros y decimales) y registros. Para que la clase *ValorOz* fuera implementada de manera correcta fue necesario crear otras clases que heredaran de esta para cada uno de sus tipos, por lo tanto se creó la clase *Integer* y *Doble*, que abarcan los tipos de numeros permitidos en el proyecto, además de la clase *Registro*, la cual comprende los datos de tipo registro.

Otras clases que se deben resaltar son las de *Variable* y *variableNoLigada* las cuales representan a las variables las cuales tendrn un valor ligado o estarán sin ligar, pero que al ser creadas harán parte del *almacen de asignación unica*. La creación de estas clases posibilitan la implementaciíon de la clase *Almacén* que en la cual se hallan algunas de las operaciones más importantes del proyecto, como unificación. El UML adjunto permite visualizar más claramente la relación entre clases que se esta implementando.


ALMACEN
- variables :list<variable*>
+ consultarVariable() : *variable
+ imprimirAlmacen() : string
+ consultarLigadura(string) : bool
+ obtenerListaVariables() : list<*variable>
+ modificarVariable(string, valorOz): void
+ agregarVariable(string) : void
+ unificarVariable(*valorOz,*valorOZ): bool
<<inteface>>
ValorOZ
+ DECIMAL : char = 'I'
+ ENTEROS : char = 'E'
+ REGISTRO : char = 'R'
+ VARIABLE : char = 'V'
+ NO_LIGADO :char = 'N'
# tipoDato : char
+ obtenerCadenaValor(): string
+ tipo(): char
child
VARIABLE
- valor : *valorOz
- nombre : string
+ obtenerValor(): *valorOz
+ obtenerCadenaValor :string
+ obtenerNombre :string
+ modificarValorI(valorOZ*): void
+ modificarNombre() : void
+ operator == (* variable) : bool
Extends
Extends
Extends
Extends
ENTERO
- valor: int
+ obtenerCadenaValor(): string
+ obtenerValor() : int
+ tipo(): char
+ operator (entera) : bool
REGISTRO
+ etiqueta : string
+ campos : list<*valorOz>
+ nombreCampos : list <string>
+ obtenerCadenaValor(): string
+ obtenerEtiqueta() : string
+ obtenerCampos() : list<string>
+ tipo() : char
+ operator(register) : bool
DOBLE
- valor: double
+ obtenerCadenaValor(): string
+ obtenerValor() : double
+ tipo(): char
+ operator(doble): bool
VARIABLE_NO_LIGADA
- valor : string
+ obtenerCadenaValor(): string
+ tipo(): char
+ operator (variable_no_ligada): bool

2. TAD Almacén

TAD Almacén

$$Almacen: [var_1 \rightarrow oz_1, var_2 \rightarrow oz_2 \ldots var_n \rightarrow oz_n]$$

{ inv1: $\forall i \cdot Oz_i \in$ ValorOz, $0 < i \leq n$ }

{ inv1: $\forall i \cdot var_i \in$ Variables, $0 < i \leq n$ }

Operaciones Primitivas:

• Almacén		→ Almacén
• Almacén	ListaValorOz	→ Almacén
• Imprimir Almacén		→ Cadena de caracteres
• Consultar Variable	Nombre Variable	→ ValorOz
• Consultar Ligadura	Nombre Variable	→ Booleano
• Modificar Variable	Variable x ValorOz	→ Booleano
• Agregar Variable	Nombre Variable	→ ValorOz
• Obtener Variables		→ ListaValorOz
• Unificar Variable	ValorOz x ValorOz	→ Booleano

Figura 2: TAD Almacn

2.1. Almacen()

Almacén ()

* Esta operación permite crear una instancia del TAD Almacen.*

{pre : `True`}
{post: almacén = [ ]}

Figura 3: Constructor Almacn

2.2. Almacen(list)

Almacén (lst)

* Esta operación permite crear una instancia del TAD Almacén inicializado con variables provistos de una lista de elemento de la clase ValorOz.*

{pre : $lst = [\boldsymbol{var_1} \rightarrow "\_", \boldsymbol{var_2} \rightarrow "\_" \ldots \boldsymbol{var_n} \rightarrow "\_"] \wedge \forall \boldsymbol{var} \in \boldsymbol{Variable}$}
{post: almacén = $[\boldsymbol{var_1} \rightarrow "\_", \boldsymbol{var_2} \rightarrow "\_" \ldots \boldsymbol{var_n} \rightarrow "\_"]$}

Figura 4: Constructor Almacn(con parametros)

2.3. imprimirAlmacen()

imprimirAlmacén ()

Esta operación imprime el contenido del almacén en formato de cadena.

{pre: $almacen = [[var_1 \rightarrow oz_1, var_2 \rightarrow oz_2 \ldots var_n \rightarrow oz_n]] \wedge n \geq 1$}
{post: $[var_1 \rightarrow oz_1, var_2 \rightarrow oz_2 \ldots var_n \rightarrow oz_n]$}

Figura 5: TAD imprimirAlmacen

2.4. consultarVariable()

consultarVariable (var)

*Permite consultar el valor con el que está ligado una variable en el almacén. *

{pre: $oz \in ValorOZ \wedge var \in Variable \wedge var \rightarrow oz$ }
{post: oz}

Figura 6: TAD consultarVariable

2.5. consultarLigadura()

consultarLigadura(var)

*Esta operación permite consultar si una variable está ligada (asociada a un valor) o no en el almacén retornando un booleano en cualquiera de los casos. *

{pre: $var \in Variable \ \wedge oz \ \in \ ValorOz$}
{post: `True` si $var \rightarrow oz$
`False` si $var \nrightarrow oz$ }

Figura 7: TAD consultarLigadura

2.6. modificarVariable()

modificarVariable (var, oz)

Esta operación permite crear una ligadura para una variable.

{pre:$var \in Variable \ \wedge \ oz \in ValorOz \ \wedge \ var \ \rightarrow$ "_" }
{post: $var \rightarrow oz$}

Figura 8: TAD modificarVariable

2.7. agregarVariable()

agregarVariable (var)

Esta operación permite agregar una variable nueva y sin ligar al almacén.

{pre:$var \in Variable \ \wedge var \notin Almacen$ }
{post: $(var \rightarrow "\_") \in Almacen$}

Figura 9: TAD agregarVariable

2.8. obtenerListaVariables()

obtenerListaVariables()

Esta operación permite obtener la lista de variables en el almacén. Tenga en cuenta que el tipo de retorno de esta función debe ser una lista.

{pre: $almacen = [var_1 \rightarrow oz_1, var_2 \rightarrow oz_2 \ldots var_n \rightarrow oz_n] \wedge n \geq 1$}
{post:$lista = [var_1, var_2 \ldots var_n$}

Figura 10: TAD obtenerListaVariables

2.9. unificarVariables()

unificarVariables(oz_1, oz_2)

Esta operación permite unificar dos variables en el almacén.

{pre:$oz \in ValorOz \land oz \in Almacen$}
{post:$oz_1 \rightarrow oz_2$ si $oz_1 \rightarrow$ "_"
$oz_2 \rightarrow oz_1$ si $oz_2 \rightarrow$ "_" }

Figura 11: TAD unificarVariables

2.10. Detalles de Implementación

En la clase Almacén la implementación que más se destaca es la de unificar variables, ya que esta utiliza a gran parte de las funciones de de la clase como modificar variable, consultar variable, y consultar ligadura. Para propsitos de esta clase se implemento punteros a valores Oz con el propósito de poder hacer un *cast* a cualquier extensión de ValorOz , en esta funcin se corroboraba que las guardas mas importantes de la unificación se cumplieran , pero esto solo sería posible si la variable a la que se buscaba ligar estuviera dentro de la lista del almacn, de esta forma observamos que la clase Almacén solo trabajara sobre almacenes con valores ya establecidos pues esta solo tiene como funcionalidad modificarla.

2.11. Complejidad

La complejidad del almacen se puede describir brevemente entre gran parte de sus operaciones constantes y lineales , un caso claro de esto es la funcion de unificar variable donde se encuentra en uno de sus mejores casos como O(1) y como peor caso O(n).

3. TAD ValorOz

TAD ValorOz

ValorOz: {Etiqueta < caracter1, …, carácter(n)>, Val <ValorOz> }

$\{inv1: Etiqueta \in caractes \wedge Val \in \mathbb{Z} \Rightarrow Etiqueta \in ValorOz\}$

$\{inv2: Etiqueta \in caracteres \wedge Val \in caractes \Rightarrow Etiqueta \in ValorOz\}$

$\{inv3: Etiqueta \in caracteres \wedge Val = "\_" \Rightarrow Etiqueta \in ValorOz\}$

Operaciones Primitivas:

▪ Crear Valor	Cadena	→ ValorOz
▪ Obtener Cadena Valor		→ Cadena
▪ Obtener Etiqueta	ValorOz	→ Cadena
▪ Obtener Lista Campos	ValorOz	→ Lista Cadena

Figura 12: TAD TADValorOz

3.1. crearValor()

crearValor(*c*)

*Crea una instancia de TAD ValorOz a partir de una cadena ***c.*** *

{pre: $c \in Cadena$ }

{post: $c \mid c \in ValorOz$ }

Figura 13: TAD crearValor()

3.2. obtenerCadenaValor()

obtenerCadenaValor()

*Permite obtener la representación de una instancia del TAD ValorOz como cadena. *

{pre: $c \in Registro \vee c \in Integer \vee c \in Doble \vee c \in Variable$ }

{post: c }

Figura 14: TAD obtenerCadenaValor()

3.3. obtenerEtiqueta()

obtenerEtiqueta()

Permite obtener la etiqueta de una instancia del TAD ValorOz que esté asociada a un registro.

{pre:$Registro = etiqueta(c_1: oz_1 \ c_2: oz_2 \ldots \ c_n: oz_n) \wedge n \geq 1 \wedge oz \in ValorOz \wedge etiqueta \in cadena$}

{post: *etiqueta*}

Figura 15: TAD obtenerEtiqueta()

3.4. obtenerListaCampos()

obtenerListaCampos ()

Permite obtener una lista con los campos de una instancia del TAD ValorOz que esté asociada a un registro.

{pre:$Registro = etiqueta(n_1: oz_1 \ n_2: oz_2 \ldots \ n_n: oz_n) \ \wedge \ n \geq 1 \ \wedge oz \in ValorOz \wedge etiqueta \in cadena$}
{post: $oz_1, oz_2 \ldots \ oz_n$}

Figura 16: TAD obtenerListaCampos()

3.5. Detalles de Implementación

Para la clase Valor0z, las caracterisiticas más destacables es el uso de clases para definir cada uno de los tipos de datos que puede abarcar ValorOz. El más intresante de todos es el de *Regístro*, ya que implementa operaciones extensas para poder recuperar los datos que sean declarados bajo su tipo, como la sobrecarga del operador == para realizar comparaciones dntro de la clase. Mas allá de esto el resto de operacion presentan operaciones más comunes, como por ejemplo el de obtener y modificar datos retornando cada uno de sus atributos.

3.6. Complejidad

La complejidad la clase ValorOz varia desde sus diferentes funciones, gran parte de las clases que heredan de ValorOz comparten la cualidad de que sus operaciones son constantes ya que la mayora obtiene informacin que solo retorna y cambia el valor de sus atributos, por otro lado esta la clase registro que tiene mtodos hasta de recorridos de complejidad cubica esto por el hecho de que sus elementos necesitan ser comparados uno a uno, para reconocer si son igual nmero, valor y etiqueta.

4. Conclusiones

En nuestro progreso como equipo, nos enfrentamos a varios procesos para llegar a la implementacin mas asertiva, debatiendo cual era la idea mas apropiada y la que mas nos permitiera hacer uso las estructuras impartidas en el curso, por otra parte, se hizo uso de una gran variedad de librerias de C++ con el propsito de optimizar el cdigo. en este proyecto hubo varios factores decisivos como el apoyo en equipo en las distintas soluciones planteadas, los integrantes se repartan distintas labores para ir facilitando algunas tareas. En el proceso

sobre las practicas de programacin, se hizo uso de tcnicas que hicieran legible el cdigo, lo que permitera depurar y mejorar los aspectos que no fueran a perjudicar a largo plazo el proyecto. Para la planeacin; el TAD y el grafico UML permitieron tener más presente el plan a tratar, concordando al plan de desarrollo a la hora de programar. El uso de repositorios,para nuestro proyecto fue esencial manejar el uso de versiones para que cada integrante del grupo tuviera acceso, hacer y verificar cambios en el cdigo , para esto se hizo uso de un manejador de versiones llamado GitHub, permiti tener un seguimiento de las lneas incluidas y descartadas. Como conclusión, el presente proyecto nos ha dejado un mejor entendimiento de los aspectos y aplicaciones de la POO incluyendo las implicaciones que se requieren para tener un buen codigo.