Proyecto de Estructuras de Datos

Jeremías Villalobos Tenorio
Santiago Caicedo Rojas
Carlos Andrés Ángel Calderón

Estructuras de Datos
Carlos Alberto Ramírez Restrepo, Ph.D

29 de Mayo del 2019

Índice

1. Introducción

En el presente documento se evidenciará la implemetación del ***TAD Almacen*** y del ***TAD ValorOz*** por medio del paradigma de la **POO**, de una manera eficiente y adecuada.

Esta implementación se llevo a cabo en el lenguaje de programación ***C++*** mediante dos clases, la clase **Almacen** y la clase **ValorOz** con sus respectivas subclases. Para esto utilizamos algunos de los contenedores de ***STL*** como lo son las listas y los mapas, por otro lado, la implementación de los **registros** se logró a través del uso de la recursión.

A continuación observaremos de manera detallada cada uno de los ***TAD'S*** con sus respectivas operaciones, al igual que se explicará el uso de las funciones axuliares que se utilizaron para nuestra implementación.

2. TAD Almacen

A continuación se mostrará la impletación detallada de cada una de las operaciones del ***TAD Almacen***.

TAD Almacen
Almacen = { Almacen = <almacen> }
{ inv: TRUE }
- CrearAlmacenLleno: map<string, ValorOz*> -> Almacen - ImprimirAlmacen: Almacen -> Vacio - ConsultarVariable: Texto -> ValorOz* - ConsultarLigaduraVariable: Texto -> Bool - ModificarVarible: Texto **x** ValorOz* -> Almacen - AgregarVariable: Texto -> Almacen - ObtenerListaVariables: Almacen -> List <string> - UnificarVaribles: Texto -> Almacen

2.0.1. CrearAlmacenLleno

CrearAlmacenLleno(map<string, ValorOz*>**)**

"Crea el almacen con respecto a un mapa pasado por parametro"

{ pre: map<string, ValorOz*> ^ n ∈ texto ^ m ∈ ValorOz }

{ post: almacen = map<string, ValorOz*> }

Esta operación recibe como parámetro un mapa con llaves **strings** y claves de punteros a **ValorOz**, después de esto se accede al atributo almacén y este se iguala al parámetro ingresado.

2.0.2. ImprimirAlmacen

ImprimirAlmacen()

"Muestra la variable y el valor asociado para cada elemento del almacen"

{ pre: Existe al menos un elemento en el Almacen }

{ post: TRUE }

En esta operación se crear un iterador de almacén, para así poder recorrer el almacén mediante un "for". Durante este ciclo se realiza un casting dependiendo del tipo de dato que este en la clave relacionado con el nombre de la variable **ValorOz**.

2.0.3. ConsultarVariable

ConsultarVariable(string**)**

"Consulta un valor con el que esta ligado una variable en el almacen"

{ pre: Existe la variable a consultar en el almacen }

{ post: ValorOz asociado a la variable }

Para esta operación se sobrescribió el método **ConsultarVariable** para que reciba dos parámetros: el nombre de la variable y el tipo de la variable; para así, poder retornar el respectivo **ValorOz**.

2.0.4. ConsultarLigaduraVariable

ConsultarLigaduraVariable(string**)**

"Consulta si una variable está ligada o no en el almacen"

{ pre: El string a consultar tiene que ser una varible }

{ post: TRUE v FALSE }

Para la esta operación la entrada es un **"string"**, el cual es un valor a consultar, para encontrar este valor se crea un iterador para recorrer el almacén y verificar si el elemento **NO** está, en este caso el ciclo se acaba y retorna **"false"**, pero si el elemento está, se verifica que el tipo no sea **"underscore"**, si el tipo es diferente a "underscore" la operación retorna **"true"**, de lo contrario retorna **"false"**.

2.0.5. ModificarVariable

ModificarVariable(string, ValorOz***)**

"Crea una ligadura para una variable"

{ pre: La variable debe de existir y debe de ser underscore }

{ post: variable modificada en el almacen }

En esta operación se reciben dos parámetros, el primero es la variable a modificar y el segundo es el valor que se le va a asignar a la variable que se paso en el primer parámetro, esto solo es posible si el tipo de la variable ingresado es **"underscore"**, de ser así, se asignara como llave la variable y como clave el valor ingresado, de no ser así se presenta un error de modificación.

2.0.6. AgregarVariable

AgregarVarible(string**)**

"Crea una nueva variable sin ligar al almacen"

{ pre: La variable no debe de existir en el almacen; si existe, debe de ser la misma variable }

{ post: Una nueva variable en el almacen }

Para esta operación se pasa un parámetro de tipo **"string"**; dentro de esta operacioón se crea un puntero a **"ValorOzUnderscore"**, en el cual creamos una nueva instancia de **"ValorOzUnderscore"**, en donde la llave es el **"string"** pasado como parÃąmetro y la clave es el **"ValorOzUndercore"** creado.

2.0.7. ObtenerListaVariables

ObtenerListaVariables()

"Se obtiene una lista con las variables del almacen"

{ pre: El almacen no debe de estar vacío }

{ post: Lista con las variables del almacen }

Para la operación **ObtenerListaVariables** creamos la lista resultante de **"strings"**, creamos un iterador que recorra el almacén y guardamos en la lista la llave de cada posición del mapa, para así finalmente retornar la lista.

2.0.8. UnificarVaribles

UnificarVariables(string, string**)**
"Permite unificar dos variables en el almacen"
{ pre: Las variables deben de existir en el almacen }
{ post: Las dos variables unificadas en el almacen }

Esta operación recibe dos **"strings"** como parámetro, los cueles se representan con variables y entra a verificar si existen en el almacén como llaves del almacén, en caso de ser posible las unifica y en caso contrario, se produce un error.

3. TAD ValorOz

A continuación se mostrará la impletación detallada de cada una de las operaciones del ***TAD ValorOz***.

TAD ValorOz
ValorOz = { ValorOz = <Entero, Flotante, Registro, Underscore> }
{ inv: TRUE }
- CrearValor: ValorOz **x** string **->** ValorOz - ObtenerCadenaValor: ValorOz **->** ValorOz - ObtenerEtiqueta: ValorOz **x** Registro **->** Etiqueta - ObtenerListaCampos: ValorOz **x** Registro **->** ListaStrings

3.0.1. CrearValor

CrearValor(string**)**

"Permite construir una instancia de ValorOz de cualquier tipo"

{ pre: Para todo **x,** tal que, **x** ∈ entero **v** flotante **v** registro **v** variable **v** underscore }

{ post: ValorOz = <entero **v** flotante **v** registro **v** variable **v** underscore> }

En esta operación se lee el tipo de dato que pasa por parámetro, para así construir el tipo de **"ValorOz"** del dato recibido.

3.0.2. ObtenerCadenaValor

ObtenerCadenaValor(ValorOz**)**

"Permite Obtener la representación de una instancia de ValorOz"

{ pre: Para todo **x,** tal que, **x** ∈ entero **v** flotante **v** registro **v** variable **v** underscore }

{ post: ValorOz = <string> }

Es un método de las subclases de ***ValorOz*** que permite retornar su valor asociado como **string**.

3.0.3. ObtenerEtiqueta

ObtenerEtiqueta(Registro**)**

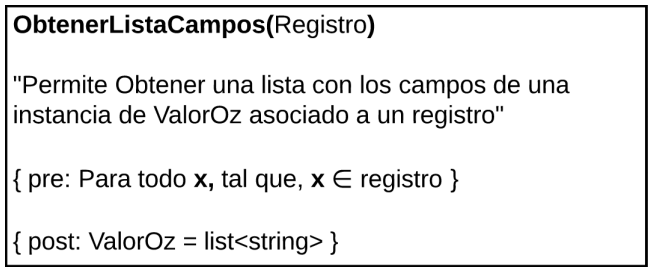
"Permite Obtener la etiqueta de una instancia de ValorOz asociado a un registro"

{ pre: Para todo **x,** tal que, **x** ∈ Etiqueta }

{ post: ValorOz = <Etiqueta> }

En esta operación se recibe como parámetro el nombre de la variable **"string"**, se realiza un casting para acceder a la subclase registro y se retorna el atributo de etiqueta.

3.0.4. ObtenerListaCampos



Esta operación se creo un iterador para recorrer el mapa de registros, donde los campos son la llave y el valor la clave.

4. Funciones Auxiliares

Para poder lograr la buena implementación de las diferentes operaciones del ***TAD Almacén*** y el ***TAD ValorOz***, se crearon unas funciones auxiliares que facilitan la organización de los datos y su manipulación.

4.0.1. Funciones Auxiliares del TAD Almacén

- ***recursion***: La función sirve para imprimir cuando **ValorOz** es una variable.
- ***imprimirRegistro***: Esta función recibe un registro e imprime sus elementos, y en caso de que uno de los elementos sea un registro se llama a si misma recursivamente.

4.0.2. Funciones Auxiliares del TAD ValorOz

- ***indentificarTipos***: Esta función determina de que tipo es el campo de un registro, su valor y su nombre.
- ***separarRegistros***: Esta función separa los campos de un registro en formato **"string"** y retorna una lista de campos de tipo **"string"**.

5. Análisis complejidad

Para las complejidades del código, no se tuvieron en cuenta la complejidad de las operaciones de bibliotecas o archivos diferentes a los desarrollados por nosotros, y en los llamados recursivos se hicieron suposiciones porque ese tema no fue abordado en el curso.

5.0.1. TAD Almacén

- **CrearAlmacen (Constructor):** $O(1)$**.** No contiene ciclos.
- **CrearAlmacenLleno:** $O(n)$**.** Esta operación recorre un mapa de manera lineal (cada elemento).
- **ImprimirAlmacen:** $O(n^m)$**.** Contiene ciclos para cada elemento del almacén y recursión para algunos elementos.
- **ConsultarVariable:** $O(1)$**.** Esta operación no contiene ciclos.
- **ConsultarLigaduraVariable:** $O(n)$**.** Contiene un ciclo para buscar un elemento de manera lineal en un mapa.
- **ModificarVariable:** $O(1)$**.** Esta operación no contiene ciclos.
- **AgregarVariable:** $O(n)$**.** Contiene un ciclo para iterar sobre las llaves de un mapa linealmente.
- **ObtenerListaVariables:** $O(n)$**.** Contiene un ciclo para iterar sobre las llaves de un mapa linealmente.
- **UnificarVaribles:** $O(n)$**.** Es lineal para recorrer un mapa, pero hace llamados de otras funciones que pueden ser recursivas.

5.0.2. TAD ValorOz

- **ValorOz:**
  - **CrearValor (Constructor):** $O(1)$**.** Esta operación no contiene ciclos.

- **ValorOzInt:**
  - **CrearValor (Constructor):** $O(n)$**.** Esta operación contiene un ciclo que recorre el tamaño de la entrada (string).
  - **ObtenerCadenaValor:** $O(1)$**.** Esta operación no contiene ciclos.
- **ValorOzFloat:**
  - **CrearValor (Constructor):** $O(1)$**.** Esta operación contiene varios ciclos independientes y lineales sobre el tamaño de la entrada(string).
  - **ObtenerCadenaValor:** $O(1)$**.** Esta operación no contiene ciclos.
- **ValorOzRegistro:**
  - **CrearValor (Constructor):** $O(n^m)$**.** Esta operación contiene varios ciclos independientes y lineales sobre el tamaño de la entrada(string) y hace llamado a la función **separarRegistros** e **identificarTipos**, además es recursiva por lo tanto 'més la cantidad de registros dentro de registros.
  - **ObtenerCadenaValor:** $O(1)$**.** Esta operación no contiene ciclos.
  - **identificarTipos:** $O(n)$**.** Esta operación recorre un string linealmente.
  - **separarRegistros:** $O(n)$**.** Esta operación recorre un string linealmente.
- **ValorOzUnderscore:**
  - **CrearValor (Constructor):** $O(1)$**.** Esta operación no contiene ciclos.
  - **ObtenerCadenaValor:** $O(1)$**.** Esta operación no contiene ciclos.
- **ValorOzVariable:**
  - **CrearValor (Constructor):** $O(1)$**.** Esta operación no contiene ciclos.
  - **ObtenerCadenaValor:** $O(1)$**.** Esta operación no contiene ciclos.

6. Conclusiones y Aspectos a mejorar

- En este proyecto fue indispensable comprender el manejo de punteros e iteradores en *C++*, pues la mayor parte de la implementación dependen de ello.

- Se presentaron muchas dificultades en la comprensión de algunas operaciones y su implementación, y esto probablemente se dio por falta de comunicación con el profesor, es decir, debimos empezar a consultar mucho antes con el profesor.

- El analizador (parser) para registros requirió crear funciones auxiliares para separar los datos en un formato más fácil de trabajar, es decir la técnica de ingeniería conocida como dividir y conquistar. Además requirió de la comprensión de cierto nivel de recursión para poder hacer registros multinivel.

- Un aspecto a mejorar es la planeación y organización en cuanto al trabajo del proyecto, pues a veces trabajábamos en alguna parte del código y la dejábamos incompleta y luego se nos olvidaba qué hacía, para despueés perder tiempo por eso. Además en algunas ocasiones agregamos funcionalidades sin pensar en sus dependencias de futuras funcionalidades.