TIPOS DE DATOS ABSTRACTOS

Contenido:

Tipos de Datos Abstractos (TDA)

1. Abstracción

La abstracción es la acción y efecto de abstraer, es decir, separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción (RAE, 2010).

También se puede decir que la abstracción es una representación que incluye los atributos más significativos de una entidad (Sebesta, 2005).

Este proceso permite clasificar a las entidades en grupos con características similares, lo cual permite enfocarse en las características que hacen a cada entidad en un grupo una entidad única, lo cual es beneficioso al aplicarse a un problema ya que permite simplificarlo, por lo tanto, se puede decir que la abstracción es uno de los mejores aliados de la programación.

Durante el proceso de abstracción se identifican las características y funcionalidades de una entidad para considerarlas de forma aislada, y así tomar en cuenta aquellas que sean esenciales o relevantes.

Se puede clasificar la abstracción en dos tipos fundamentales:

- Abstracción de Datos: La abstracción de datos es un proceso que permite tomar las características esenciales que pueden describir a un objeto.
- Abstracción Funcional: La abstracción funcional está orientada al funcionamiento del objeto. Separa el "qué" hace del "como" lo hace.

2. Idea Intuitiva

Se requiere desarrollar un programa que permita crear y manipular triángulos en un espacio bidimensional. Para lograrlo se deben definir los tipos de datos Punto y Triangulo:

```
tipo

      Punto = registro
            arreglo [1..2] de real: coord
      fregistro

      Triangulo = registro
            arreglo [1..3] de Punto: puntos
      fregistro
```

Luego se define una operación llamada construirTriangulo, que permite generar un triángulo definido por tres puntos, *a*, *b* y *c*.

```
proc construirTriangulo(ref Triangulo: t; Punto: a, b, c)
inicio
      t.puntos[1].coord[1] = a.coord[1]
      t.puntos[1].coord[2] = a.coord[2]
      t.puntos[2].coord[1] = b.coord[1]
      t.puntos[2].coord[2] = b.coord[2]
      t.puntos[3].coord[1] = c.coord[1]
      t.puntos[3].coord[2] = c.coord[2]
fproc
```

Esta operación puede ser utilizada luego dentro del algoritmo principal:

```
algoritmo Principal
tipo
      // Tipos Punto y Triángulo
inicio
      var
            Punto: p1, p2, p3
            Triangulo: t1
      ...
      construirTriangulo(t1, p1, p2, p3);
      escribir("Triangulo:")
      escribir("Punto 1: (", t1.puntos[1].coord[1], ", ", t1.puntos[1].coord[2], ")")
      escribir("Punto 2: (", t1.puntos[2].coord[1], ", ", t1.puntos[2].coord[2], ")")
      escribir("Punto 3: (", t1.puntos[3].coord[1], ", ", t1.puntos[3].coord[2], ")")
fin
```

¿Qué sucedería si se cambiara la estructura de datos utilizada para almacenar un punto?

```
Punto = registro
      real: x, y
fregistro
```

En este caso habría que cambiar todas las operaciones de Triangulo y aquellas partes del programa principal que hagan uso de Punto.

El problema en este caso es que los algoritmos que hacen uso de la estructura Punto fueron creados específicamente para la misma sin tomar en cuenta que en el tiempo ésta podría cambiar. Para evitar este problema, es necesario separar la definición de la estructura de datos de aquellos algoritmos que hacen uso de la misma, es decir, la estructura de datos pasa a ser un ente abstracto, del cual no se conoce su estructura interna. De esta forma nace un tipo de dato abstracto.

Ahora bien, si un algoritmo que utiliza un tipo de dato abstracto no puede conocer la estructura interna del mismo, ¿Cómo puede utilizarlo?

Para que un tipo de dato abstracto pueda ser utilizado por un algoritmo, debe brindar una serie de operaciones que permitan su manipulación. Esto permite que al realizar algún cambio en la estructura de datos, solamente baste con modificar sus operaciones. Las operaciones de un tipo de dato abstracto permiten conocer que hace el mismo sin tomar en cuenta como lo hace.

¿Cómo se podría reestructurar el algoritmo anterior convirtiendo Punto y Triangulo en tipos de datos abstractos?

```
// Tipos

tipo

      Punto = registro
            real: x, y
      fregistro

      Triangulo = registro
            arreglo [1..3] de Punto: puntos
      fregistro

// Operaciones de Punto

proc construirPunto(ref Punto: p; real: x, y)
inicio
      p.x ← x
      p.y ← y
fproc

proc copiarPunto(ref Punto: p1; Punto: p2)
inicio
      p1.x ← p2.x
      p1.y ← p2.y
fproc

func obtenerX(Punto: p): real
inicio
      retornar(p.x)
ffunc

func obtenerY(Punto: p): real
inicio
      retornar(p.y)
ffunc

proc modificarX(ref Punto: p; real: x)
inicio
      p.x ← x
fproc

proc modificarY(ref Punto: p; real: y)
inicio
      p.y ← y
fproc

proc imprimirPunto(Punto: p)
inicio
      escribir("(", p.x, ", ", p.y, ")")
fproc
```

```
// Operaciones de Triangulo

proc construirTriangulo(ref Triangulo: t; Punto: a, b, c)
inicio
      copiarPunto(t.puntos[1], a)
      copiarPunto(t.puntos[2], b)
      copiarPunto(t.puntos[3], c)
fproc

proc copiarTriangulo(ref Triangulo: t1; Triangulo: t2)
inicio
      copiarPunto(t1.puntos[1], t2.puntos[1])
      copiarPunto(t1.puntos[2], t2.puntos[2])
      copiarPunto(t1.puntos[3], t2.puntos[3])
fproc

func obtenerA(Triangulo: t): Punto
inicio
      retornar(t.puntos[1])
ffunc

func obtenerB(Triangulo: t): Punto
inicio
      retornar(t.puntos[2])
ffunc

func obtenerC(Triangulo: t): Punto
inicio
      retornar(t.puntos[3])
ffunc

proc modificarA(ref Triangulo: t; Punto: a)
inicio
      copiarPunto(t.puntos[1], a)
fproc

proc modificarB(ref Triangulo: t; Punto: b)
inicio
      copiarPunto(t.puntos[2], b)
fproc

proc modificarC(ref Triangulo: t; Punto: c)
inicio
      copiarPunto(t.puntos[3], c)
fproc

proc imprimirTriangulo(Triangulo: t)
inicio
      escribir("Punto 1: ”)
      imprimirPunto(t.puntos[1])
      escribir("Punto 2: ”)
      imprimirPunto(t.puntos[2])
      escribir("Punto 3: ”)
      imprimirPunto(t.puntos[3])
fproc
```

El algoritmo principal quedaría como sigue:

```
algoritmo Principal
tipo
      // Tipos Punto y Triángulo
inicio
      var
            Punto: p1, p2, p3
            Triangulo: t1
      ...
      construirTriangulo(t1, p1, p2, p3);
      escribir("Triangulo:")
      imprimirTriangulo(t1)
fin
```

Nótese que en las operaciones del TDA Triangulo no se accede directamente a la estructura interna del TDA Punto, sino que se utilizan las operaciones del TDA Punto para manipular la estructura. Igualmente en el algoritmo principal no se accede directamente a las estructuras de los TDAs Punto y Triangulo, sino que se utilizan sus operaciones.

Ejercicio: Cambie la estructura del TDA Triangulo para que, en vez de tener un vector de puntos, tenga tres variables de tipo punto llamadas *a*, *b* y *c*. Realice los cambios necesarios en las operaciones del TDA Triangulo para que funcionen con la nueva estructura.

3. Tipo de Dato Abstracto

Un tipo de dato abstracto (TDA) es una combinación de la representación de datos de un tipo de dato específico y los subprogramas que proveen las operaciones para el mismo (Sebesta, 2005). Un TDA es resultado del proceso de abstracción de datos y funcionalidades de una entidad y está formado por:

a) Una representación o estructura de datos.
b) Un conjunto de operaciones sobre dichos datos.

En el diseño de un TDA se aplican dos conceptos importantes:

- Encapsulamiento: Consiste en agrupar la estructura de datos y las operaciones producto del proceso de abstracción.
- Ocultación de Información: Consiste en ocultar los aspectos relacionados al diseño de la estructura de datos o la implementación de sus operaciones con la finalidad de hacerlos inaccesibles del resto del programa en caso tal que haya cambios en el diseño.

4. Ventajas de la Utilización de TDAs

- Permiten modelar objetos complejos del mundo real de una forma simple.
- Contienen los datos y funcionalidades más relevantes de un objeto para un problema determinado.
- Separa la especificación de la implementación.

- Permiten crear sistemas modulares.
- Pueden ser reutilizados para otros sistemas.

5. Ciclo de Vida de un TDA

En el proceso de desarrollo de un TDA podemos distinguir una serie de etapas las cuales conforman el ciclo de vida de un TDA. Estas etapas son: Diseño, implementación y uso. En la salida de cada etapa se obtienen uno o varios documentos que sirven de entrada para la siguiente etapa.

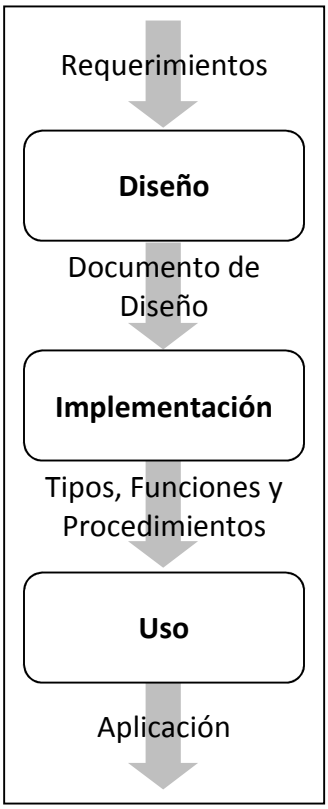

Figura 1. Ciclo de Vida

5.1. Diseño o Especificación

Esta etapa responde a la pregunta: ¿Cómo es el TDA?

Esto es posible gracias a la definición de un modelo matemático para el tipo de dato abstracto, el cual describe su comportamiento independientemente de cualquier representación de datos o implementación de las operaciones.

Para que una especificación sea útil debe ser precisa (sólo tiene que decir aquello realmente imprescindible), general (debe ser adaptable a diferentes contextos), legible (que sirva como instrumento de comunicación entre el especificador y los usuarios por un lado; y entre el especificador y el implementador por el otro) y no ambigua (que evite posteriores problemas de interpretación) (Franch, 1993).

La especificación de un TDA se puede definir formalmente como una terna E = (D, F, A) donde (Dale, 1996):

- D (Data Types) es el conjunto de dominios y rangos en el contexto del TDA.
- F (Formal Syntax) es el conjunto de operaciones válidas sobre el TDA.
- A (Axiomatic Semantics) es el conjunto de axiomas que definen el comportamiento del TDA.

La especificación del conjunto de operaciones del TDA se conoce como especificación sintáctica o sintaxis y el conjunto de axiomas se conoce como especificación semántica o sólo semántica.

De acuerdo al tipo de semántica que se utilice para especificar al TDA, se obtienen dos posibles tipos de especificación: La especificación algebraica y la especificación operacional.

Especificación Algebraica

La especificación algebraica nos da una visión global del tipo de datos y define el comportamiento de las operaciones para todos los valores del TDA (Franch, 1993).

1. Sintaxis: La sintaxis o signatura de un TDA define su nombre y los nombres de sus operaciones, incluyendo una definición de sus parámetros y valores devueltos (Louden, 2004), es decir, el dominio y el rango de sus operaciones.

   Para lograr la independencia del lenguaje, las operaciones son representadas mediante la notación funcional.

   Ejemplo: Si se quiere especificar el TDA Conjunto, su sintaxis sería la siguiente:

   TDA Conjunto[Elemento]

vacio :		$\rightarrow$	Conjunto
es_vacio :	Conjunto	$\rightarrow$	Booleano
pertenece :	Conjunto x Elemento	$\rightarrow$	Booleano
insertar :	Conjunto x Elemento	$\rightarrow$	Conjunto
eliminar :	Conjunto x Elemento	$\rightarrow$	Conjunto

2. Semántica: La semántica de la especificación algebraica es una semántica axiomática ya que aplica la lógica matemática a la descripción del comportamiento del TDA mediante un conjunto de proposiciones lógicas llamadas axiomas.

   Ejemplo: Para el TDA Conjunto:

   $\forall c \in$ Conjunto; e, e_1, $e_2 \in$ Elemento; $e \neq e_1$; $e_1 \neq e_2$;

   Ax1: es_vacio(vacio) = Verdadero
   Ax2: es_vacio(insertar(c, e)) = Falso
   Ax3: es_vacio(eliminar(insertar(vacio, e),e)) = Verdadero
   Ax4: pertenece(vacio, e) = Falso
   Ax5: pertenece(insertar(c, e), e) = Verdadero
   Ax6: pertenece(insertar(c, e), e_1) = pertenece (c, e_1)
   Ax7: pertenece(eliminar(insertar(vacio, e), e_1), e_2) = (e = e_2)
   Ax8: $\neg$es_vacio(c) $\rightarrow$ pertenece(eliminar(insertar(c, e), e_1), e_2) = pertenece(insertar(eliminar(c, e_1), e), e_2) (Conmutatividad)
   Ax9: $\neg$es_vacio(c) $\rightarrow$ pertenece(eliminar(c, e), e) = Falso (Unicidad)

Especificación Operacional

La especificación operacional, describe individualmente el comportamiento de cada operación en términos de la representación escogida para el tipo, y es el punto de partida de una verificación posterior del código de la operación (Franch, 1993). La especificación operacional está formada por dos partes, la sintaxis y la semántica.

1. Sintaxis: La sintaxis es la misma utilizada en la especificación algebraica.

2. Semántica: La semántica de la especificación operacional de un TDA está formada por dos elementos:

   a. Precondiciones: Son predicados lógicos que deben satisfacerse antes de comenzar la ejecución de una operación.
   b. Postcondiciones: Son predicados lógicos que deben satisfacerse al acabar la ejecución de una operación.

   Por esta razón también se le conoce como especificación Pre/Post. Una especificación Pre / Post para un programa o fragmento de programa *P* se escribe de la siguiente manera:

   { Pre : A_1 }
   P
   { Post : A_2 }

   Esta especificación denota que si el programa *P* comienza a funcionar en un estado que satisface el predicado A_1 entonces *P* termina en un tiempo finito y en un estado que satisface el predicado A_2.

   En este sentido, la especificación Pre / Post describe el comportamiento esperado del programa *P*.

   Ejemplo: Especificación Pre/Post del TDA Conjunto[Elemento]

   { Pre: }
   func vacio ( ) : Conjunto
   { Post: vacio ← { } }

   { Pre: }
   func es_vacio (Conjunto: C) : **lógico**
   { Post: es_vacio ← (C = { }) }

   { Pre: }
   func pertenece (Conjunto: C; Elemento: e) : **lógico**
   { Post: pertenece ← (e ∈ C) }

   { Pre: }
   proc insertar (**ref** Conjunto: C; Elemento: e)
   { Post: C ← C ∪ {e} }

   { Pre: }
   proc eliminar (**ref** Conjunto: C; Elemento: e)
   { Post: C ← C – {e} }

5.2. Implementación

La implementación de un TDA consiste en darle una representación concreta en algún lenguaje (pseudoformal o de programación).

Implementación vs. Implementación de un TDA: En ciencias de la computación una implementación es la realización de una especificación técnica o algoritmo en forma de programa, componente de software u otro sistema de cómputo, mediante la utilización de algún lenguaje de programación. Es una de las etapas del desarrollo de *software* (análisis, diseño, implementación, pruebas). Por otro lado, la implementación de un TDA consiste en obtener una representación concreta del tipo de dato abstracto. Si se realiza esta etapa en lenguaje pseudoformal, forma parte de la etapa de diseño del *software* y puede ser llamada etapa de desarrollo del TDA, en cambio si se realiza esta etapa directamente en un lenguaje de programación, forma parte de la etapa de implementación del *software*.

Se puede pensar en la implementación de un TDA como:

- Una estructura de datos que representa el TDA.
- La implementación de las operaciones del TDA.
- Convenciones sobre como las implementaciones utilizaran la estructura de datos elegida. Estas convenciones son definidas mediante el invariante de la representación y la función de abstracción.

De acuerdo a lo antes descrito, la implementación de un TDA se puede dividir en dos fases: (1) Representación del TDA y (2) Desarrollo de las Operaciones del TDA.

Fase 1: Representación del TDA

Esta fase consiste en la representación de los elementos que constituyen el TDA mediante una estructura de datos en algún lenguaje de programación.

En primer lugar se debe escoger una representación interna adecuada para representar el TDA, es decir, una forma de estructurar la información de manera que podamos representar todos los objetos de nuestro tipo de dato abstracto de una manera eficaz. Para lograr esto, se debe seleccionar una estructura de datos adecuada para la implementación y sobre la cual se implementarán las operaciones. A éste tipo, se le denomina tipo *Rep* (tipo concreto).

Se debe establecer la relación entre el tipo *Rep* y el tipo abstracto que se está construyendo, esto se logra mediante la definición de una función, llamada función de abstracción. Esta función relaciona cada objeto que se puede representar en el tipo *Rep* con un objeto abstracto.

Igualmente se debe establecer una condición sobre el conjunto de valores del tipo *rep* que nos indique si corresponden a un objeto válido. Esta condición es una proposición lógica que se denomina invariante de la representación.

Ejemplo:

Representación en Lenguaje Pseudoformal:

```
const N ← 10000
tipo
   Elemento = ?
   Elementos = arreglo[1..N] de Elemento
   Conjunto = registro
         Elementos e
         entero cardinalidad
   fregistro
```

Función de Abstracción:

f_{abs} : *Rep* $\rightarrow$ Conjunto

r $\rightarrow$ { *r*.e[i] / 1 ≤ i ≤ r.cardinalidad }

Invariante de la Representación:

(0 ≤ *r*.cardinalidad ≤ N) ∧

(*r*.cardinalidad > 0 $\rightarrow$ ∀i∀j: ((i, j∈ [1, r.cardinalidad]) ∧ (i ≠ j)) $\rightarrow$ (r.e[i] ≠ r.e[j]))

Fase 2: Desarrollo de las Operaciones del TDA.

Consiste en desarrollar las operaciones definidas en la especificación del TDA mediante funciones o procedimientos.

Ejemplo:

```
func vacio(): Conjunto
var Conjunto: C
inicio
   C.cardinalidad ← 0
   retornar(C)
ffunc

func es_vacio(Conjunto: C): lógico
inicio
   retornar(C.cardinalidad = 0)
ffunc

func pertenece(Conjunto: C; Elemento: e): lógico
var
   entero: i
   lógico: encontrado
inicio
   i ← 1
   encontrado ← falso
   mientras (i ≤ C.cardinalidad ∧ ¬encontrado) hacer
         si (C.e[i] = e) entonces
               encontrado ←  verdadero
         fsi
```

```
            i ← i + 1
      fmientras
      retornar(encontrado)
   ffunc

   proc insertar(ref Conjunto: C; Elemento: e)
   inicio
      si (¬pertenece(C, e) ∧ C.cardinalidad < N) entonces
            C.cardinalidad ← C.cardinalidad + 1
            C.e[C.cardinalidad] ← e
      fsi
   fproc

   proc eliminar(ref Conjunto: C; Elemento: e)
   var
      entero: i
      lógico: encontrado
   inicio
      i ← 1
      encontrado ← falso
      mientras (i ≤ C.cardinalidad ∧ ¬encontrado) hacer
            si (C.e[i] = e) entonces
                  encontrado ← verdadero
            sino
                  i ← i + 1
            fsi
      fmientras
      si (encontrado) entonces
            C.e[i] ← C.e[C.cardinalidad]
            C.cardinalidad ← C.cardinalidad – 1
      fsi
   fproc
```

5.3. Uso

En esta etapa se utilizan las operaciones del TDA para resolver un problema en una aplicación. El uso del TDA se limita a realizar llamadas a las operaciones sobre la estructura que se requiera cuidando siempre cumplir con las reglas de cada operación, ya especificadas en las etapas anteriores (Martínez y Quiroga, 2002).

6. Tipos de Operaciones de un TDA

Las operaciones de un Tipo de Dato Abstracto (TDA) se clasifican según su funcionalidad sobre el objeto abstracto como:

- Constructoras: Son operaciones encargadas de crear elementos del TDA. En el caso típico, una operación constructora genera el objeto abstracto más simple. Un TDA puede tener varias constructoras.

- Modificadoras: Son operaciones que pueden alterar el estado de un elemento del TDA.

- Observadoras: Son operaciones que consultan el estado de un elemento del TDA y retornan algún tipo de información. Son operaciones que no alteran el estado del objeto.

- Destructoras: Son operaciones que se encargan de retornar el espacio de memoria ocupado por un objeto abstracto. Después de su ejecución el objeto abstracto deja de existir y cualquier operación que se aplique sobre él va a generar un error.

- Persistencia: Son operaciones que permiten salvar / leer el estado de un objeto abstracto de algún medio de almacenamiento en memoria secundaria.

7. Organización de los Datos

Los tipos de datos cuentan con cierta estructura lógica de organización dependiente de la naturaleza de los datos a almacenar. Existen 3 tipos posibles de organizaciones:

- Lineal: La relación entre los elementos es de uno a uno. Ejemplo: Listas, Pilas y Colas.
- Jerárquica: La relación entre los elementos es de uno a muchos. Ejemplo: Árboles.
- Multienlazada: La relación entre los elementos es de muchos a muchos. Ejemplo: Grafos y matrices.

Nos enfocaremos solamente en las estructuras lineales.

8. Estructuras Lineales

Las estructuras lineales son aquellas en las cuales la relación existente entre los elementos de la estructura es de uno a uno, es decir un elemento solamente puede estar relacionado únicamente con otro elemento.

8.1. Lista (List)

Una lista es una secuencia de cero o más elementos de un tipo determinado (Aho, 1983). Una posible notación para representar una lista es la siguiente:

$$L = < e_1, e_2, e_3, \ldots, e_n >$$

Donde:

- $n \geq 0$ es la cantidad de elementos de la lista.
- e_i $(0 \leq i \leq n)$ es un elemento de la lista.

A la cantidad n de elementos de la lista se le denomina longitud. Si $n = 0$ se dice que la lista es vacía y se denota $L = <>$. Si $n > 0$, entonces se dice que e_1 es el primer elemento y e_n es el último elemento. La posición de un elemento e_i dentro de la lista es el lugar ocupado por dicho elemento dentro de la secuencia, es decir, i. Una propiedad importante de una lista es que sus elementos pueden ser ordenados linealmente de acuerdo a la posición que ocupen en la lista; se puede decir que e_i es el predecesor de e_{i+1} para todo $i = 1, 2, 3, \ldots, n - 1$, y e_i es el sucesor de e_{i-1} para todo $i = 2, 3, 4, \ldots, n$.

Especificación del TDA Lista

TDA Lista[Elemento]

Sintaxis:

vacia:		→	Lista
es_vacia:	Lista	→	Logico
insertar:	Lista × Natural × Elemento	→	Lista
eliminar:	Lista × Natural	→	Lista
consultar:	Lista × Natural	→	Elemento
esta:	Lista × Elemento	→	Logico
longitud:	Lista	→	Natural

Semántica:

a) Algebraica

$\forall L \in$ Lista; $p, p_1 \in$ N; $e, e_1 \in$ Elemento, tenemos:

Ax1: es_vacia(vacia) = verdadero
Ax2: $1 \le p \le$ longitud(L)+1 → es_vacia(insertar(L, p, e)) = falso
Ax3: $1 \le p \le$ longitud(L)+1 → consultar(insertar(L, p, e), p) = e
Ax4: $1 \le p_1 < p \le$ longitud(L)+1 → consultar(insertar(L, p, e), p_1) = consultar(L, p_1)
Ax5: $1 \le p < p_1 \le$ longitud(L)+1 → consultar(insertar(L, p, e), p_1) = consultar(L, p_1–1)
Ax6: $1 \le p_1 < p \le$ longitud(L) → consultar(eliminar(L, p, e), p_1) = consultar(L, p_1)
Ax7: $1 \le p < p_1 \le$ longitud(L)–1 → consultar(eliminar(L, p, e), p_1) = consultar(L, p_1+1)
Ax8: $1 \le p \le$ longitud(L)+1 → consultar(eliminar(insertar(L, p, e), p), p_1) = consultar(L, p_1)
Ax9: esta(vacia, e) = falso
Ax10: $1 \le p \le$ longitud(L)+1 → esta(insertar(L, p, e), e) = verdadero
Ax11: $e \ne e_1 \wedge 1 \le p \le$ longitud(L)+1 → esta(insertar(L, p, e), e_1) = esta(L, e_1)
Ax12: $1 \le p \le$ longitud(L)+1 → esta(eliminar(insertar(L, p, e), p), e_1) = esta(L, e_1)
Ax13: longitud(vacia) = 0
Ax14: $1 \le p \le$ longitud(L)+1 → longitud(insertar(L, p, e)) = longitud(L) +1
Ax15: $1 \le p \le$ longitud(L)+1 → longitud(eliminar(insertar(L, p, e), p)) = longitud(L)

b) Operacional

{Pre: }
func vacia(): Lista
{Post: vacia ← <>}

{Pre: }
func es_vacia(Lista: L): **lógico**
{Post: es_vacia ← (L = <>) }

{Pre: $1 \le p \le longitud(L)+1$ }
proc insertar(**ref** Lista: L; **entero**: p; Elemento: e)
{Post: Si $L = <e_1, e_2, \ldots, e_{p-1}, e_p, e_{p+1} \ldots, e_n>$ entonces luego de insertar
$L = <e_1, e_2, \ldots, e_{p-1}, e, e_p, e_{p+1}, \ldots, e_n>$ }

{Pre: $1 \le p \le longitud(L)$ }
proc eliminar(**ref** Lista: L; **entero**: p)
{Post: Si $L = <e_1, e_2, \ldots, e_{p-1}, e_p, e_{p+1} \ldots, e_n>$ entonces luego de eliminar
$L = <e_1, e_2, \ldots, e_{p-1}, e_{p+1}, \ldots, e_n>$ }

{Pre: $1 \le p \le longitud(L)$ }
func consultar(Lista: L; **entero**: p): Elemento
{Post: Si $L = <e_1, e_2, \ldots, e_{p-1}, e_p, e_{p+1} \ldots, e_n>$ entonces consultar $\leftarrow e_p$ }

{Pre: }
func esta(Lista: L; Elemento: e): **lógico**
{Post: esta $\leftarrow \exists i \in [1, longitud(L)]: e_i = e$ }

{Pre: }
func longitud(Lista: L): **lógico**
{Post: Si $L = <>$ entonces longitud $\leftarrow 0$
Si $L = <e_1, e_2, \ldots, e_n>$ entonces longitud $\leftarrow n$ }

Implementación del TDA Lista

Representación:

```
const
   N ← 10000
tipo
   Elemento = ?
   Lista = registro
         arreglo[1..N] de Elemento: e
         entero: longitud
   fregistro
```

Función de Abstracción:

f_{abs} : *Rep* $\rightarrow$ Lista
r $\rightarrow$ < *r*.e[1], *r*.e[2], …, *r*.e[i], …, *r*.e[*r*.longitud] >

Invariante de la Representación:

$0 \le r.longitud \le N$

Desarrollo de las Operaciones:

```
func vacia(): Lista
var
   Lista: L
inicio
   L.longitud ← 0
   retornar(L)
ffunc

func es_vacia(Lista: L): lógico
inicio
   retornar(L.longitud = 0)
ffunc

proc insertar(ref Lista: L; entero: p; Elemento: e)
var
   entero: i
inicio
   si(L.longitud < N) entonces
         para i ← L.longitud hasta p hacer
               L.e[i+1] ← L.e[i]
         fpara
         L.e[p] ← e
         L.longitud ← L.longitud + 1
   fsi
fproc

proc eliminar(ref Lista: L; entero: p)
var
   entero: i
inicio
   para i ← p hasta L.longitud – 1 hacer
         L.e[i] ← L.e[i+1]
   fpara
   L.longitud ← L.longitud – 1
fproc

func consultar(Lista: L; entero: p): Elemento
inicio
   retornar(L.e[p])
ffunc

func esta(Lista: L; Elemento: e): lógico
var
   entero: i
   lógico: encontrado
inicio
   encontrado ← falso
   i ← 1
   mientras (¬encontrado ∧ i≤L.longitud) entonces
         si(L.e[i] = e) entonces
               encontrado ← verdadero
```

```
        sino
              i ← i + 1
        fsi
   fmientras
   retornar(encontrado)
ffunc

func longitud(Lista: L; Elemento: e): lógico
inicio
   retornar(L.longitud)
ffunc
```

8.2. Pila (Stack)

Es un tipo especial de lista en la cual todas las inserciones y eliminaciones se hacen en uno de los finales de la lista, llamado tope (Aho, 1983). En esta estructura el último elemento que llega es el primer elemento que sale, por lo cual también se llama estructura LIFO (Last In, First Out).

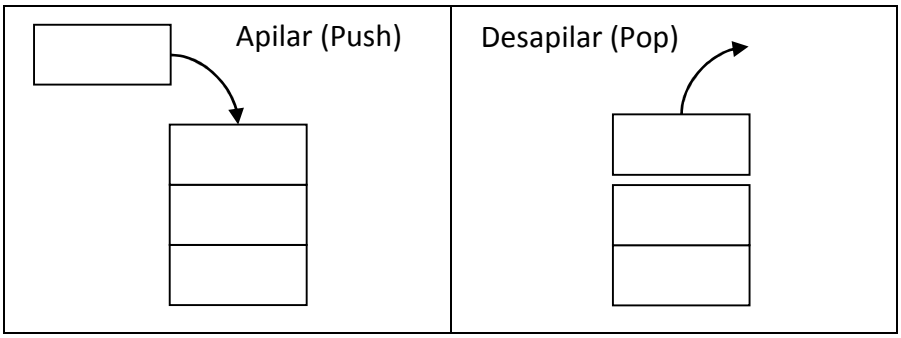

Figura 2. Movimientos válidos de una Pila

En todo momento el único elemento visible es el último elemento que se colocó, y el punto en el cual se encuentra se denomina tope. El lugar en el cual se encuentra el primer elemento que se insertó a la pila se denomina fondo. Si una pila no tiene elementos se dice que se encuentra vacía y no tiene sentido referirse ni a su tope ni a su fondo.

Las pilas son muy utilizadas en programación para realizar tareas tales como:

- Evaluación de expresiones.
- Reconocimiento de lenguajes.
- Recorrido de árboles.
- Simulación de procesos recursivos.

Especificación del TDA Pila

TDA Pila[Elemento]

Sintaxis:

vacia:		→	Pila
es_vacia:	Pila	→	lógico
apilar:	Pila × Elemento	→	Pila
desapilar:	Pila	→	Pila
tope:	Pila	→	Elemento

Semántica:

a) Algebraica

$\forall P \in$ Pila; $\forall e \in$ Elemento, tenemos:

Ax1: es_vacia(vacia) = verdadero
Ax2: es_vacia(apilar(*P*, *e*)) = falso
Ax3: desapilar(apilar(*P*, *e*)) = *P*
Ax4: tope(apilar(*P*, *e*)) = *e*

b) Operacional

{Pre: }
func vacia(): Pila
{Post: vacia ← <>}

{Pre: }
func es_vacia(Pila: P): **lógico**
{Post: es_vacia ← (P = <>) }

{Pre: }
proc apilar(**ref** Pila: P; Elemento: e)
{Post: Si P = <e_1, e_2, …, e_{n-1}, e_n> entonces luego de apilar
P = <e, e_1, e_2, …, e_{n-1}, e_n> }

{Pre: ¬es_vacia(P) }
proc desapilar(**ref** Pila: P)
{Post: Si P = <e_1, e_2, …, e_{n-1}, e_n> entonces luego de desapilar
P = <e_2, …, e_{n-1}, e_n> }

{Pre: ¬es_vacia(P) }
func tope(Pila: P): Elemento
{Post: Si P = <e_1, e_2, …, e_{n-1}, e_n> entonces tope ← e_1 }

Implementación del TDA Pila

Representación:

```
const N ← 10000
tipo
   Elemento = ?
   Pila = registro
         arreglo[1..N] de Elemento: e
         entero: longitud
   fregistro
```

Función de Abstracción:

f_{abs} : *Rep* → Pila

r → < *r*.e[*r*.longitud], …, *r*.e[2], *r*.e[1] >

Invariante de la Representación:

0 ≤ *r*.longitud ≤ N

Desarrollo de las Operaciones:

```
func vacia(): Pila
var
   Pila: P
inicio
   P.longitud ← 0
   retornar(P)
ffunc

func es_vacia(Pila: P): lógico
inicio
   retornar(P.longitud = 0)
ffunc

proc apilar(ref Pila: P; Elemento: e)
inicio
   si(P.longitud < N) entonces
         P.longitud ← P.longitud +1
         P.e[P.longitud] ← e
   fsi
fproc

proc desapilar(ref Pila: P)
inicio
   P.longitud ← P.longitud – 1
fproc

func tope(Pila: P): Elemento
inicio
   retornar(P.e[P.longitud])
ffunc
```

8.3. Cola (Queue)

Es un tipo especial de lista en la cual los elementos se insertan por un extremo y se eliminan por el otro (Aho, 1983). Debido a esta propiedad las colas también son llamadas estructuras FIFO (First In, First Out).

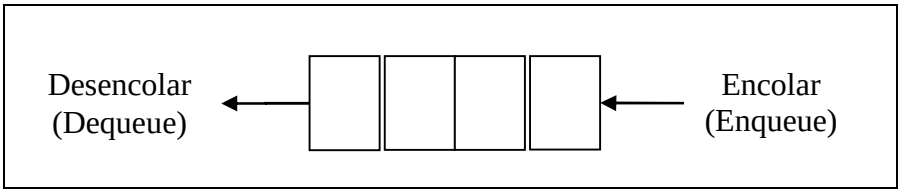

Figura 3. Movimientos válidos de una cola.

En todo momento, el único elemento visible de la estructura es el primero que se colocó, denominado frente, y mientras éste no haya salido no es posible tener acceso al siguiente elemento. Si una cola no contiene elementos, se dice que se encuentra vacía, y no tiene sentido referirse a su frente.

Las colas son muy utilizadas en programación para realizar tareas tales como:

- Algoritmos de recorrido de árboles y grafos.
- Procesos de simulación en los cuales se quiere determinar el comportamiento de un sistema que presta un servicio a un conjunto de usuarios que esperan mientras les toca el turno de ser atendidos. Ejemplos de estos sistemas son los bancos, los aeropuertos y los procesos dentro de un computador.

Especificación del TDA Cola

TDA Cola[Elemento]

Sintaxis:

vacia:		→	Cola
es_vacia:	Cola	→	lógico
encolar:	Cola × Elemento	→	Cola
desencolar:	Cola	→	Cola
frente:	Cola	→	Elemento

Semántica:

a) Algebraica

$\forall C \in$ Cola; $\forall e, e_1 \in$ Elemento, tenemos:

Ax1: es_vacia(vacia) = verdadero
Ax2: es_vacia(encolar(*C*, *e*)) = falso
Ax3: frente(encolar(vacia, *e*)) = *e*
Ax4: frente(encolar(*C*, *e*)) = frente(*C*)
Ax5: es_vacia(desencolar(encolar(vacia, *e*))) = verdadero
Ax6: desencolar(encolar(encolar(*C*, *e*), e_1)) = encolar(desencolar(encolar(*C*, *e*)), e_1)

b) Operacional

{Pre: }
func vacia(): Cola
{Post: vacia ← <>}

{Pre: }
func es_vacia(Cola: C): **lógico**
{Post: es_vacia ← (C = <>) }

{Pre: }
proc encolar(**ref** Cola: C; Elemento: e)
{Post: Si C = <e_1, e_2, …, e_{n-1}, e_n> entonces luego de encolar
C = <e_1, e_2, …, e_{n-1}, e_n, e> }

{Pre: ¬es_vacia(C) }
proc desencolar(**ref** Cola: C)
{Post: Si C = <e_1, e_2, …, e_{n-1}, e_n> entonces luego de desencolar
C = <e_2, …, e_{n-1}, e_n> }
{Pre: ¬es_vacia(C) }
func frente(Cola: C): Elemento
{Post: Si C = <e_1, e_2, …, e_{n-1}, e_n> entonces frente ← e_1 }

Implementación del TDA Cola

Representación:

Vectores circulares: Se colocan dos campos en la estructura que indican las posiciones en las cuales se encuentran el primer y el último elemento de la cola, de manera tal que sólo el espacio comprendido entre estas dos marcas se encuentre ocupado por los elementos de la cola.

Se debe tener en cuenta que la posición siguiente a una posición i de un arreglo de dimensión N está dada por (i mod N) + 1, para todo i entre 1 y N. Para ello se define la función sig que suma 1 a la posición i en el sentido circular, retornando la posición siguiente a la posición i.

```
const
   N ← 10000
```

```
tipo
   Elemento = ?
   Cola = registro
          arreglo[1..N] de Elemento: e
          entero: longitud, primera, ultima
   fregistro

func sig(entero: i): entero
inicio
   retornar(i mod N + 1)
ffunc
```

Función de Abstracción:

f_{abs} : *Rep* → Cola

r → < *r*.e[*r*.primera], …, *r*.e[i], …, *r*.e[*r*.ultima] >

Invariante de la Representación:

$(0 \le r.\text{longitud} \le N) \wedge (1 \le r.\text{primera} \le N) \wedge (1 \le r.\text{ultima} \le N)$

Desarrollo de las Operaciones:

```
func vacia(): Cola
var
   Cola: C
inicio
   C.longitud ← 0
   C.primera ← 1
   C.ultima ← N
   retornar(C)
ffunc

func es_vacia(Cola: C): lógico
inicio
   retornar(C.longitud = 0)
ffunc

proc encolar(ref Cola: C; Elemento: e)
inicio
   si(C.longitud < N) entonces
          C.ultima ← sig(C.ultima)
          C.e[C.ultima] ← e
          C.longitud ← C.longitud + 1
   fsi
fproc

proc desencolar(ref Cola: C)
inicio
   C.primera ← sig(C.primera)
   C.longitud ← C.longitud – 1
fproc
```

```
func frente(Cola: C): Elemento
inicio
   retornar(C.e[C.primera])
ffunc
```

9. Estructuras Lineales Dinámicas

Las estructuras lineales dinámicas, a diferencia de las estáticas cuyos elementos se almacenan en espacios de memoria contiguos, están formadas por elementos que pueden almacenarse de manera dispersa en la memoria, por esta razón, si se requiere generar un nuevo elemento, este puede ser almacenado en una posición de memoria que se reserva en tiempo de ejecución.

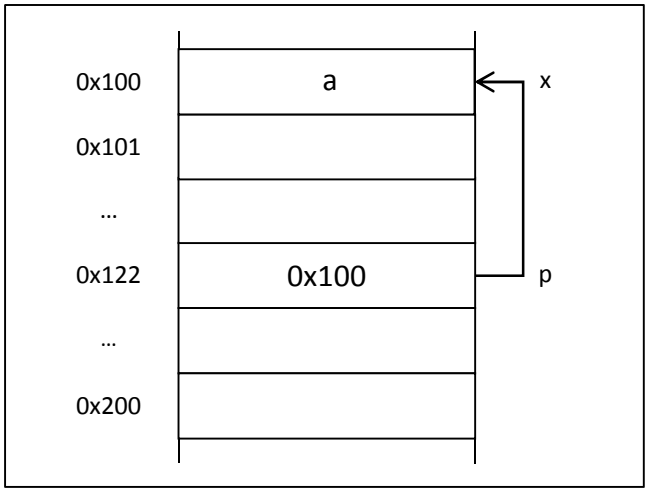

Figura 4. Un apuntador en la memoria.

9.1. Repaso de Apuntadores

Un apuntador es una variable que contiene una dirección de memoria. Normalmente, esa dirección es la posición en memoria de otra variable. Si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda. También se dice que un apuntador hace referencia a otra variable.

Gráficamente un apuntador se puede representar tal como se muestra en la Figura 5.

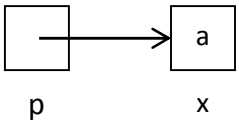

Figura 5. Representación gráfica de un apuntador.

Declaración

Sintaxis

```
apuntador a Tipo_de_Dato: p
```

Inicialización

a) Creación de un apuntador nulo:

Figura 6. Representación gráfica del apuntador nulo.

Sintaxis
```
p ← nulo
```

b) Reservación de un espacio de memoria del tamaño del tipo de dato al cual hace referencia el apuntador:

Sintaxis
```
crear(p)
```

c) Asignación del valor de otro apuntador.

Sintaxis
```
apuntador a Tipo_de_Dato: p, q
crear(p)
q ← p
```

d) Asignación de la dirección de memoria de una variable existente del tipo de dato al cual hace referencia el apuntador.

Sintaxis
```
Tipo_de_Dato: x
apuntador a Tipo_de_Dato: p
p ← βx
```

Operaciones

a) Referenciación (β): La operación de referenciación consiste en obtener la dirección de memoria correspondiente a una variable. Esta operación se representa con el operador β (beta).

b) Desreferenciación (↑): La operación de desreferenciación o indirección consiste en obtener el valor de la variable a la cual hace referencia un apuntador. Esta operación se representa con el operador ↑.

Sintaxis
```
Tipo_de_Dato: x
apuntador a Tipo_de_Dato: p
crear(p)
p↑ ← x
```

Creación y Eliminación Dinámica de Objetos

Reservar Memoria: `crear(p)`
Liberar Memoria: `liberar(p)`

9.2. Lista

La representación dinámica de lista se denomina lista enlazada (*Linked List*), la cual es una colección o secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente por medio de un enlace o apuntador. Estos elementos son llamados nodos y se componen de dos campos:

a) Un campo para la Información, el cual es de tipo genérico.
b) Un apuntador al siguiente nodo en la lista.

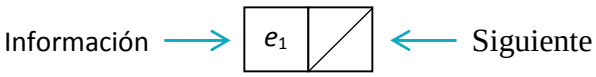

Figura 7. Nodo.

Al enlazar varios nodos, cada uno con el siguiente, se obtiene la lista.

Gráficamente los enlaces se representan mediante flechas, pero hay que tomar en cuenta que son apuntadores y cada uno tiene la dirección de memoria de donde se encuentra el siguiente nodo en la lista.

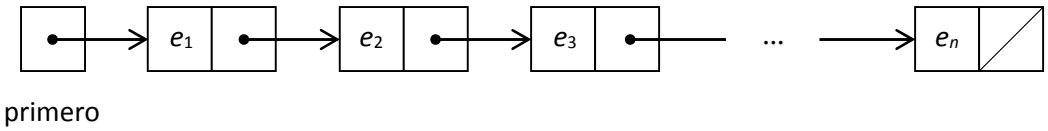

Figura 8. Representación general de lista simplemente enlazada.

Implementación del TDA Lista con una Estructura de Datos Dinámica

Representación:

```
tipo
   Elemento = ?
   Nodo = registro
         Elemento: info
         apuntador a Nodo: siguiente
   fregistro

   Lista = registro
         apuntador a Nodo: primero
         entero: longitud
   fregistro
```

Función de Abstracción:

f_{abs} : *Rep* → Lista

r → < *r*.primero↑.info, r.primero↑.siguiente↑.info, *r*.primero↑.siguiente↑.siguiente↑.info, …>

Invariante de la Representación:

(*r*.longitud ≥ 0) ∧ (*r*.longitud = 0 → *r*.primero = **nulo**)

Desarrollo de las Operaciones:

```
proc vacia(ref Lista: l)
inicio
   l.primero ← nulo
   l.longitud ← 0
fproc

func esVacia(Lista: l): lógico
inicio
   retornar(l.primero = nulo)
ffunc

proc insertar(ref Lista: l, entero: e, pos)
var
   entero: i
   apuntador a Nodo: nuevo, ant, sig
inicio
   si(pos < 1 || pos > l.longitud + 1) entonces
         escribir("Error: Se intentó insertar en una posición inválida.")
   sino
         crear(nuevo)
         nuevo↑.info ← e
         si(pos = 1) entonces
               nuevo↑.siguiente ← l.primero
               l.primero ← nuevo
         sino
               ant ← l.primero
               sig ← ant↑.siguiente
               para i ← 2 hasta pos – 1 hacer
                     ant ← sig
                     sig ← sig↑.siguiente
               fpara
               ant↑.siguiente ← nuevo
               nuevo↑.siguiente ← sig
         fsi
         l.longitud ← l.longitud + 1
   fsi
fproc

proc eliminar(ref Lista: l; entero: pos)
var
   apuntador a Nodo: ant, act, sig
   entero: i
inicio
```

```
      si(pos < 1 || pos > l.longitud) entonces
            escribir("Error: Se intentó eliminar una posición inexistente.")
      sino
            si(pos = 1) entonces
                  act ← l.primero
                  l.primero ← act↑.siguiente
                  liberar(act)
            sino
                  ant ← l.primero
                  act ← ant↑.siguiente
                  sig ← act↑.siguiente
                  i ← 2
                  mientras(i < pos) hacer
                        ant ← act
                        act ← sig
                        sig ← sig↑.siguiente
                        i ← i + 1
                  fmientras
                  ant↑.siguiente ← sig
                  liberar(act)
            fsi
            l.longitud ← l.longitud – 1
      fsi
fproc

func consultar(ref Lista: l; entero: pos): elemento
var
      apuntador a Nodo: act
      entero: i
inicio
      si(pos < 1 || pos > l.longitud) entonces
            escribir("Error: Se intentó consultar una posición inexistente.")
      sino
            act ← l.primero
            i ← 1
            mientras(i < pos) hacer
                  act ← act->siguiente
                  i ← i + 1
            fmientras
            retornar(act↑.info)
      fsi
fproc

func longitud(Lista: l): entero
inicio
      retornar(l.longitud)
ffunc

proc destruir(ref Lista: l)
var
      apuntador a Nodo: act
inicio
      act ← l.primero
      mientras(act ≠ nulo) hacer
```

```
            l.primero ← l.primero↑.siguiente
            liberar(act)
            act ← l.primero
        fmientras
    fproc
```

10. Referencias

Aho, A., Hopcroft, J. y Ullman, J. (1983). Data Structures and Algorithms. Massachusetts, USA: Addison-Wesley.

Dale, N. y Walker, H. (1996). Abstract Data Types: Specifications, Implementations and Applications. Massachusetts, USA: Heath and Company.

Franch, X. (1999). Estructuras de datos. Especificación, Diseño e Implementación (Tercera Edición). Barcelona, España: Edicions UPC.

Louden, K. (2004). Lenguajes de Programación, Principios y Práctica (Segunda edición). México D.F., México: International Thomson Editores S.A.

Martínez, A., Rosquete, D. (2009). NASPI: Una Notación Estándar para Programación Imperativa. Télématique.

Martínez, R., Quiroga, E. (2002). Estructuras de Datos: Referencia Práctica con Orientación a Objetos. México: Thomson Learning.

Sebesta, R. (2005). Concepts of Programming Languages (Séptima Edición). Colorado, USA: Pearson Education Inc.