Programación 2

Tipos Abstractos de Datos (Introducción)

Abstracción

1. Ignorar detalles
2. Posponer la consideración de detalles

En ciencias experimentales

Se observan fenómenos y se crean modelos abstractos que explican aspectos de esos fenómenos.

- Ejemplo: Leyes de Kepler sobre el movimiento planetario.
    - Los planetas y el sol se reducen a puntos
    - Se consideran sólo propiedades del movimiento de los planetas (modelo cinético)
    - Se ignoran masa, composición química, etc.

Abstracción (cont.)

En programación

- Tenemos un lenguaje para expresar programas.
- Si deseamos que los programas sean ejecutados por computadoras, entonces deberán finalmente ser expresados en lenguaje de máquina.
- En estos lenguajes, las primitivas son elementales y por ello los programas deben ser especificados a un nivel de extremo detalle, lo cual los hace muy complejos.
- Tratamos entonces de construir programas por refinamientos sucesivos. Los programas se expresan en términos de componentes que no están expresados directamente en lenguaje de máquina. Luego se aplica el mismo procedimiento a cada componente, refinándolo, hasta alcanzar el nivel de detalle requerido.

Lenguajes de programación

- Un lenguaje de programación de "alto" nivel de abstracción es aquel cuyas primitivas pueden ser refinadas de modo mecánico a lenguaje de máquina. El programa que efectúa ese refinamiento es el compilador del lenguaje.
- Ejemplo visto: programa que lee coordenadas de vértices de un triángulo y calcula el área de éste.
  -- Podemos expresarlo en un lenguaje de "alto nivel" como C++ de la siguiente forma:

```
< importaciones >
< declaraciones >
main ()  // Area Triang.
< declaraciones >
  {
    < instrucciones >
  } // Area Triang.
```

Ejemplo: Area de Triángulo

- **Podemos comenzar introduciendo el tipo de datos que representará el concepto de punto (en coordenadas cartesianas):**
  - `< declaraciones >`

```
struct Punto {
    float x;
    float y; };
```

  - `<···>`
- **y luego las variables del programa:**
  - `< declaraciones ...>`

```
Punto p1, p2, p3;
```

  - `<···>`

Ejemplo: Area de Triángulo (cont.)

- **Luego refinamos la parte de instrucciones:**
  - `< instrucciones >`

    `< Leer (p1, p2, p3) >;`

    `< Desplegar (AreaTri (p1, p2, p3)) >;`
  - `<···>`
- **Donde estamos haciendo uso de "instrucciones abstractas"**
- **<u>Abstractas</u> porque no están (todavía) dadas al nivel concreto de detalle del lenguaje de programación. Pero podemos especificar su efecto con precisión.**
- **Otra forma de decir lo mismo:**
  - **son instrucciones (se puede especificar su efecto) pero no son instrucciones concretas del lenguaje.**

Ejemplo: Area de Triángulo (cont.)

- El problema original (construir un programa) se resuelve por medio de una estructura cuyos componentes son o bien instrucciones concretas o bien otros programas a ser refinados separadamente.

  – **El método permite acotar el nivel de detalle a ser considerado cada vez.**

- Otra forma de decir lo mismo: se usan abstracciones para particionar el problema dado. Luego se refinan las abstracciones siguiendo el mismo método.

Ejemplo: Area de Triángulo (cont.)

- En el ejemplo introducimos una función AreaTri que hay que refinar:
  - **< declaraciones ...>**

```
float  AreaTri (Punto p1, Punto p2, Punto p3)
{ < declaraciones AreaTri ...>
        < Calcular base = distancia (p1, p2) >;
        < Calcular altura = distancia p3 a p1p2 >;
        return base * altura / 2.0;
};  // end AreaTri.
```

Abstracción Procedural

- El uso de instrucciones abstractas se llama:
  - **ABSTRACCION de PROCEDIMIENTO**
  - **SUBPROGRAMAS**
  - **PROCEDIMENTAL**
  - **(Inglés: PROCEDURAL ABSTRACTION)**

  y es la forma más elemental de abstracción.

- Algunas instrucciones abstractas se refinan en subprogramas (procedimientos, funciones) en C++.
- Otras simplemente en instrucciones que sustituyen textualmente a la instrucción abstracta.

Abstracción de Datos

- Nos interesa extender la idea de abstracción a los TIPOS de DATOS.
- Es decir, usar TIPOS ABSTRACTOS de DATOS además de instrucciones abstractas, para diseñar programas.
- ¿Qué es un tipo abstracto de datos?

  Es un tipo de datos, es decir, puede ser especificado como tal con precisión, pero no está dado como un tipo de datos concreto del lenguaje. Esto es: no está dado en términos de los constructores de tipo del lenguaje.

- En C++:
  - **elementales:** int, float, char, T * (punteros)
  - **estructurales:** [ ] (arreglos) , struct, tipos de estructuras dinámicas.

Abstracción de Datos (cont.)

- ¿Cómo se introduce un Tipo Abstracto de Datos?

  Básicamente: dándole un nombre y asociando a él un número de operaciones aplicables a los elementos del tipo. En C++, estas operaciones son subprogramas (procedimientos, funciones, métodos).

- ¿Cómo se refina un Tipo Abstracto de Datos?

  Definiéndolo como un tipo concreto del lenguaje y refinando en forma acorde los subprogramas asociados.

- Usualmente se habla de
  - ESPECIFICACION del T.A.D.
    - DEFINICION: correspondiendo a su introducción
  - IMPLEMENTACION del T.A.D.: correspondiendo a su refinamiento.

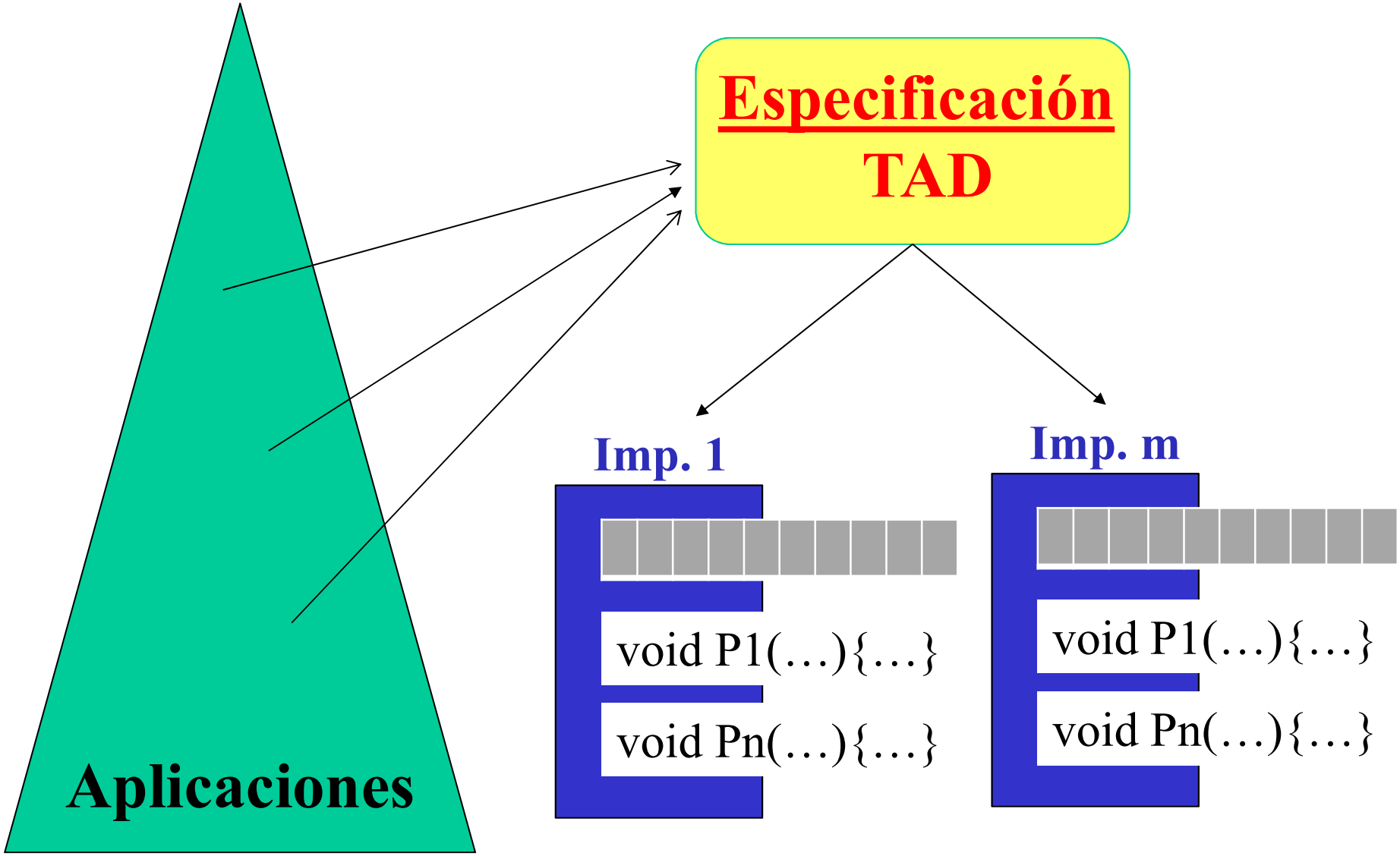
Abstracción de Datos (cont.)

- En este contexto, el tipo concreto provisto para definir el T.A.D. en una implementación dada se llama una REPRESENTACION del T.A.D.
- Hay una analogía con las estructuras algebraicas:

  Ej: monoide denomina una clase de estructuras, formadas por un conjunto, un elemento x del conjunto y una operación asociativa que tiene a x como neutro. **Representación**: (N,+,0)

- Podemos decir: un conjunto y unas operaciones que cumplen ciertas leyes.
- En general, tipo abstracto corresponde a "clase de estructura algebraica", mientras implementación del tipo abstracto se corresponde con una "estructura concreta que satisface la especificación de la clase".

Sobre TADs


Especificación
TAD
Aplicaciones
Imp. 1
void P1(…){…}
void Pn(…){…}
Imp. m
void P1(…){…}
void Pn(…){…}

Uso de TADs

Veamos un caso de diseño de programa usando un tipo abstracto de datos.

PROBLEMA: Escribir un programa que simule una calculadora con la que se pueda operar con números racionales, dados en la forma m/n con m, n enteros (n<>0).

- Pensamos en el modelo de una calculadora común, que consta de un visor donde se muestra el último resultado computado.
- Con ese número se puede operar, sumando, restando, multiplicando o dividiendo el mismo con otro número, lo cual da lugar a un nuevo resultado que reemplaza al anterior.
- También es posible borrar el último resultado o reemplazarlo por otro que se ingresa directamente.

⇒ El programa a construir aceptará comandos, posiblemente con argumentos correspondiendo a cada una de las operaciones enumeradas.

⇒ Contestará a cada uno de esos comandos con el resultado, simulando de esta forma el visor de la calculadora.

Uso de TADs (cont.)

He aquí un diseño del programa:

```
< importaciones main >
< declaraciones >
void main()   // el void es optativo
{  //  Calc. Racionales
        < declaraciones main >
        < instrucciones main >
 }  //  Calc. Racionales
```

Comenzamos por introducir las variables (i.e. la estructura de datos del programa).

- **Necesitamos una variable para almacenar el último resultado computado (es decir, la variable que simula el visor de la calculadora).**

Uso de TADs (cont.)

- 1. ¿Qué tipo tendrá esa variable?

  - **Deberá almacenar números racionales.**
  - **Este tipo no es un tipo de datos primitivo del lenguaje. Ahora bien, queremos continuar el diseño del programa sin detenernos a pensar en este momento cómo representar números racionales.**
  - **El método será: usar el tipo Racional como un tipo abstracto, a ser refinado posteriormente.**
  - **Las operaciones asociadas a este tipo abstracto serán aquellas que vayamos detectando como necesarias en el transcurso del diseño del programa.**

Definición de TADs

2. **¿Cómo se especificará un T.A.D. en C++?**

   – **Se escribe un módulo de definición, donde se introduce:**

      • **la definición del tipo**

      • **la definición de sus operaciones**

   pero no su implementación

   O sea: en el módulo de definición sólo especificamos el tipo, no lo implementamos.

- **Ventajas de este mecanismo**:

   1) **Nos permite trabajar a un mayor nivel de abstracción. Esto es: posponer el problema de implementar este tipo.**

   – **En general existirán varias implementaciones posibles y habrá que estudiarlas en detalle para optar por una de ellas. No queremos involucrarnos en ese proceso ahora.**

Definición de TADs (cont.)

2) El programa principal, como se verá, no dependerá de la implementación que oportunamente se escoja, sino sólo de la especificación del tipo.

- **Por lo tanto, modificar la implementación del tipo no implica tener que modificar el programa principal.**

3) Ahora refinamos el programa principal, importando el módulo de definición recién introducido.

< importaciones main >

```
Incluir módulo de definición del tipo Racional;
```

<…>

<declaraciones de main>

Declaración de variables

4) Ahora podemos declarar variables de tipo Racional en el programa principal. De hecho necesitamos dos, a saber:
   - **una para el "visor"**
   - **otra para almacenar cada operando a ser combinado con el "visor" por medio de las operaciones aritméticas**

```
< declaraciones de main >
    < tipos, procedimientos, constantes de main >
    Racional visor, opnd;
<...>
```

Comandos

5) Ahora necesitamos considerar los comandos aceptados por el programa. Introducimos el tipo de los comandos como una enumeración:

< tipos, procedimientos, constantes de main >

```
enum Comando { suma, resta, producto,
               división, borrar, ingreso, fin };
```

< … >

y declaramos una variable de este tipo para almacenar cada comando ingresado por el usuario.

< declaraciones main >

Comando c;

< … >

Refinando main

6) Vamos ahora a refinar la parte de instrucciones del programa principal:

< instrucciones main >

< inicialización >

< bloque principal >

< terminación >

< -- >

Refinando el bloque principal

```
Repetir
   LeerComando (c);
   En el caso que c sea
        suma: < proceso suma >
   |    resta: < proceso resta >
   |    producto: < proceso producto >
   |    división: < proceso división >
   |    borrar: < proceso borrado >
   |    ingreso: < proceso ingreso >
   < mostrar visor si corresponde >
Mientras (c!=fin)
```

Ojo!, estamos usando un pseudo C++

Refinando procesos

7) Al refinar los procesos correspondientes a los comandos, obtendremos las operaciones requeridas para el tipo Racional.

```
< proceso suma >
  LeerRacional (opnd);
  SumaRacional (opnd, visor);
< -- >
```

- Aquí **`LeerRacional`** lee un número racional en la variable dada como parámetro y **`SumaRacional`** suma el primer parámetro al segundo, modificando este último.
- La resta, producto y división funcionan de manera similar. Veamos los otros casos:

Refinando procesos (cont.)

```
< proceso borrado >
  BorrarRacional(visor);
< -- >

< proceso ingreso >
  LeerRacional(visor);
< -- >
```

- Finalmente, mostramos el visor luego de procesado cada comando, excepto el de finalización:

```
< mostrar visor si corresponde >
  if (c!=fin) EscribirRacional(visor);
< -- >
```

Operaciones del TAD Racional

8) La inicialización no hace otra cosa que borrar el visor y desplegar mensajes apropiados.

- Esto último es lo único que hace por su parte la terminación.
- Es decir, que ya hemos detectado todas las operaciones que el programa principal necesita usar en relación a los racionales. Estas operaciones deberán ser importadas del módulo Racionales:

```
< operaciones necesarias >
   SumaRacional, RestaRacional,
   ProductoRacional, DivisiónRacional,
   BorrarRacional, LeerRacional, EscribirRacional
< -- >
```

Especificación de operaciones

9) Todos estos procedimientos deben ser declarados en el módulo de especificación Racionales, y sus efectos deben ser especificados con precisión:

< declaraciones de operaciones sobre Racionales >

< operaciones básicas >

```
Racional SumaRacional (Racional q, Racional r);
/* precondición: q es un racional válido Q
        r es un racional válido R o está borrado
   postcondición: r = Q + R, o r borrado si lo
   estaba originalmente
*/
```

y similarmente se especifica el resto de las ops.

Operaciones básicas de Racionales

10) Las operaciones básicas mencionadas antes son:

- las que permiten formar racionales a partir de un numerador y un denominador enteros. (CONSTRUCTORAS)
- las que permiten obtener el numerador y el denominador de un racional dado. (SELECTORAS / DESTRUCTORAS)
- las que permiten detectar si una variable racional está o no "borrada" (definida o vacía). (PREDICADOS)

- Estas no son directamente referenciadas por nuestro programa principal, pero:
  - Serán útiles en general para otros programas que manipulen racionales.
  - Serán aquellas cuya implementación manipule directamente la representación que se elija para el tipo abstracto.

Operaciones básicas de Racionales (cont)

- En particular, las operaciones ya declaradas arriba pueden definirse en términos de estas operaciones básicas.
- Declaramos, por lo tanto:

```
< operaciones básicas >
Racional CrearRacional (int m, int n);
/*  precondición: n != 0
    retorna: m / n                    */
int Numerador (Racional q);
/*  precondición: q es un racional válido
    retorna: el numerador de q        */
int Denominador (Racional q);
boolean EsVálido (Racional q);
< -- >
```

Implementando el TAD Racional (cont)

11) En este punto, el programa principal y el módulo Racionales pueden desarrollarse en forma separada. La comunicación entre ambos ha quedado precisamente establecida. Si se compila el módulo de especificación de racionales, entonces el programa principal puede ser compilado también pues sólo depende del módulo de especificación.

- En forma independiente puede desarrollarse y compilarse el módulo de implementación de los racionales. Esto favorece el trabajo en grupo que puede operar en paralelo una vez definidos los módulos de especificación necesarios.

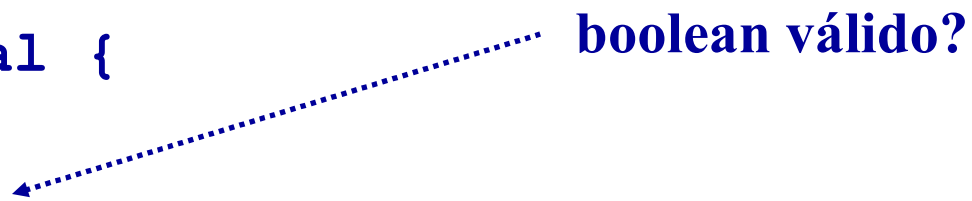
12) Por el lado del programa principal, resta solamente implementar el procedimiento de lectura de comandos y terminar de refinar el bloque principal.

Implementando el TAD Racional (cont)

Veamos ahora cómo se puede refinar el TAD Racional.

- En nuestro caso podemos tener:

```
< representación elegida >
struct Racional {
    int num;
    int denom; };
< -- >
```




```
¿Tiene sentido la siguiente representación?:
* Racional
```

Implementando el TAD Racional (cont)

- Un ejemplo de implementación de la suma de racionales es el siguiente:

```
Racional SumaRacional (Racional q, Racional r)
{ Racional sum;
  sum.num = q.num * r.denom + r.num * q.denom;
  sum.denom = q.denom * r.denom;
  sum = Normalizar(sum);
  return sum;
} // consideramos sólo racionales válidos, extenderlo:
```

Numerador(q) "usar oper. básicas"

sum.válido? = r.válido?

SumaRacional cumple su especificación ?

(No!, SumaRacional debería dejar el resultado en el segundo argumento, sin retornar nada, según la especificación inicial. Ver otras alternativas -como función-)

- El procedimiento Normalizar calcula la forma simplificada del racional dado como parámetro.
- Puede ser local al módulo de implementación.

Conclusiones

- La metodología permite especificar tipos abstractos durante el proceso de diseño de programas.
- Provee un método conveniente de abstracción y refinamiento.
- !!! Los programas que usan tipos abstractos permanecen independientes de la implementación de éstos. Las representaciones son opacas para los programas, lo cual garantiza:
  - 1) que los programas manipulen los objetos del tipo abstracto SOLO A TRAVES DE LAS OPERACIONES DEFINIDAS (consistencia en el uso del tipo).
  - 2) que los programas no deben modificarse si la implementación del tipo abstracto se modifica.

Conclusiones

Algunas ventajas del uso de TADs

- Modularidad
- Adecuados para sistemas no triviales
- Separación entre especificación e implementación. Esto hace al sistema:
  - más legible
  - más fácil de mantener
  - más fácil de verificar y probar que es correcto. Robustez.
  - más fácil de reusar
  - más extensible
  - lo independiza en cierta manera de las distintas implementaciones (complejidad tiempo-espacio).

Conclusiones

- La genericidad es una característica que buscaremos desarrollar y explotar para fomentar el reuso.
- El uso de TADs permite una rápida prototipación de sistemas, dejando de lado detalles de eficiencia para un etapa posterior.
- …..

Observaciones

Los ejemplos desarrollados en estas transparencias hicieron uso de un pseudo C (pseudo-código).

Veremos más adelante en el curso la especificación e implementación de TAD's en C.

Introducción a TADs Lineales
Lista – Pila – Cola

TAD LISTA

Observación

Es importante remarcar que NO existe un único TAD Lista (una única especificaión de Lista).

Para Listas, como para muchos TADs, existe más de una especificación, con operaciones diferentes.

Por ejemplo, si consideramos la inserción de elementos en una lista podríamos incluir operaciones que insertan:

- al comienzo de una lista o al final (lugares fijos)
- en una posición dada (listas indizadas o de posiciones explícitas),
- luego de una posición corriente (lista de posiciones implícitas),
- de manera ordenada (listas ordenadas).

Ejemplo de una especificación de Lista

Considere el TAD **Lista Indizada no acotada** de elementos de un tipo genérico, con las siguientes operaciones:

Constructoras:

Lista_Vacia: construye la lista vacía.

Insertar: dados una lista, un entero n y un elemento e, inserta e en la lista en la posición n. Si la lista tiene longitud m, con m menor a n-1, lo inserta en la posición m+1. Si la lista tiene longitud m, con m mayor o igual a n-1, inserta e en la posición n y desplaza en una posición los elementos que estuvieran en las posiciones siguientes.

Predicados:

Esta_Vacia: retorna true si, y solamente si, la lista es Vacía.

Esta_Definido: dados una lista y un entero n, retorna true si, y solamente si, la lista está definida en la posición n.

Ejemplo de una especificación de Lista (cont.)

Selectoras:

Elemento: dados una lista y un entero n, retorna el elemento en la posición n. Si la lista tiene longitud menor a n, la operación está indefinida.

Borrar: dados una lista y un entero n, elimina de la lista el elemento en la posición n. Si la posición no está definida, la operación no hace nada. Si la posición está definida, elimina el elemento en dicha posición y desplaza en una posición los elementos que estuvieran en las posiciones siguientes (contrae la lista).

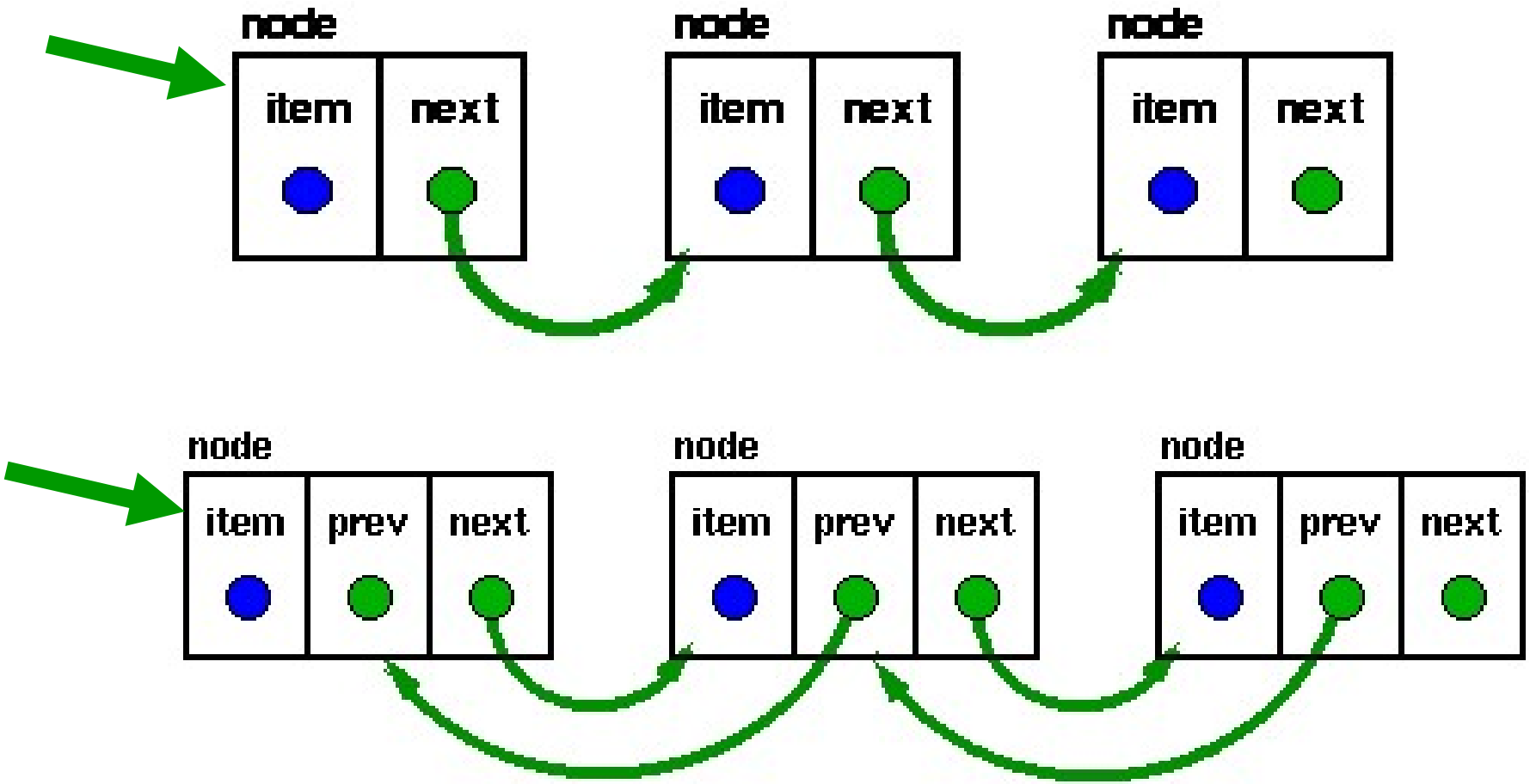
Destructora:

Destruir: destruye una lista, liberando la memoria que ésta ocupa.

Implementación

- Una implementación de un TAD consiste en un tipo de estructura de datos concreto que se elige como representación del TAD y las correspondientes implementaciones de los procedimientos.
- En el caso de las listas especificadas arriba, una implementación natural se basa en la representación por medio de listas encadenadas. Veremos esto más adelante en el curso.

Algunas implementaciones de Listas



TAD PILA (STACK)

Definición

Un **stack (o pila)** es una clase especial de lista en la que todas las inserciones y supresiones de elementos se efectúan sobre uno de sus extremos, llamado el ***tope*** del stack.

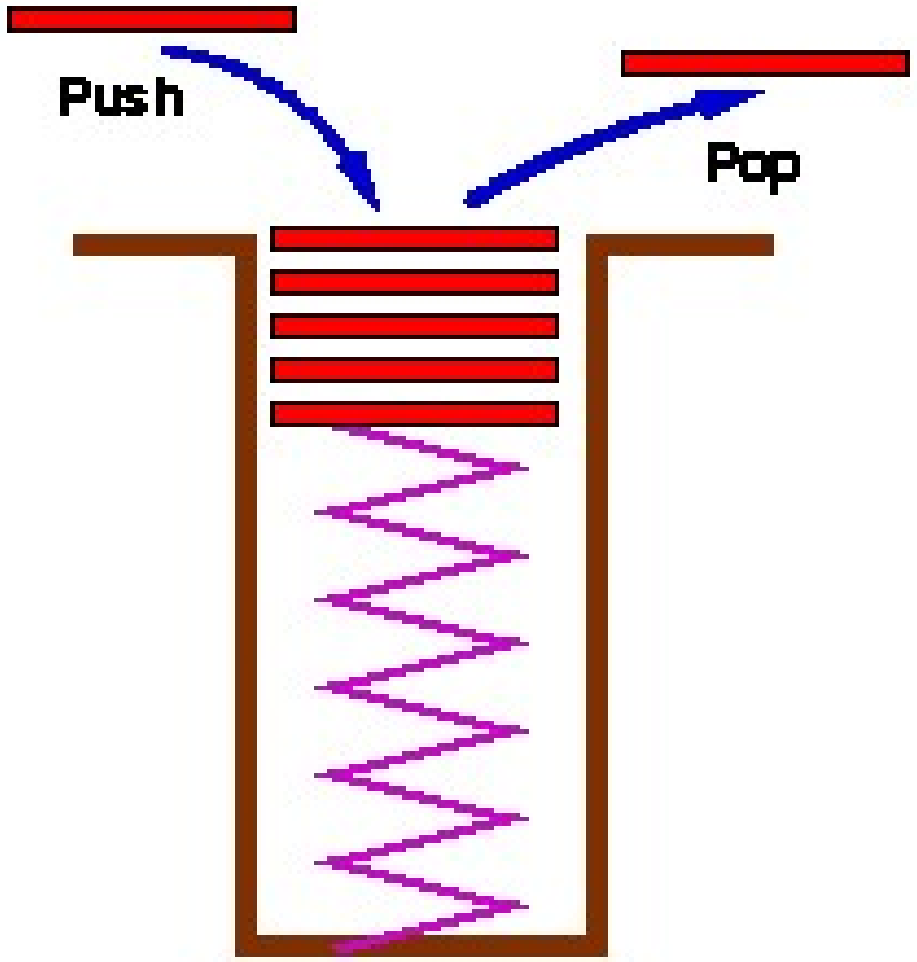
Ejemplos de stacks son:

- pila de fichas de poker en una mesa
- pila de platos para lavar, pila de libros, ...
- stack de ejecuciones

donde es claramente conveniente quitar el elemento que está en el tope de la pila o agregar un elemento nuevo sobre el tope de la misma.

Definición

Otro nombre que se le da a este tipo de estructura es el de lista ***LIFO*** (last-in-first-out).



Operaciones

El TAD Stack (Pila) incluye básicamente las siguientes operaciones:

- la operación que construye un stack vacío, **`Empty`** (**`crearPila`**);
- **`Push`** (**`apilar`**), que inserta un elemento en el tope del stack;
- **`Top`** (**`cima`**), que retorna el elemento que se encuentra en el tope del stack;
- **`Pop`** (**`desapilar`**), remueve el elemento que se encuentra al tope (en la cima) del stack;
- **`IsEmpty`** (**`esVaciaPila`**), que testea si el stack es vacío o no;
- Si se trata de un stack acotado se incluye un predicado adicional, **`IsFull`** (**`esLlenaPila`**), que testea si el stack está lleno. Notar que en este caso la operación **`Push`** tendría precondición;
- Finalmente puede incluirse una operación para destruir un stack (**`destruirPila`**), liberando la memoria que éste ocupa.

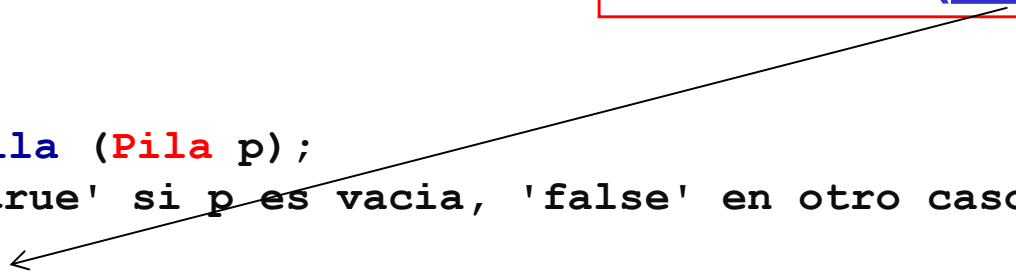
En C/C++

Especificación del TAD Pila (Acotada)

```
#ifndef _PILA_H
#define _PILA_H

struct RepresentacionPila;
typedef RepresentacionPila * Pila;

// CONSTRUCTORAS
void crearPila (int cota, Pila &p);
/* Devuelve en p la pila vacía, que podrá contener hasta cota
   elementos. */
// crearPila podría ser una función que retorne la Pila creada.

void apilar (int i, Pila &p);
/* Si !esLlenaPila(p) inserta i en la cima de p,
   en otro caso no hace nada. */

// SELECTORAS
int cima (Pila p);
/* Devuelve la cima de p.
   Precondicion: ! esVaciaPila(p). */

void desapilar (Pila &p);
/* Remueve la cima de p.
   Precondicion: ! esVaciaPila(p). */
```

En C/C++

Especificación del TAD Pila (Acotada)



```
// PREDICADOS
bool esVaciaPila (Pila p);
/* Devuelve 'true' si p es vacia, 'false' en otro caso. */

bool esLlenaPila (Pila p);
/* Devuelve 'true' si p tiene cota elementos, donde cota es el
   valor del
   parametro pasado en crearPila, 'false' en otro caso. */

// DESTRUCTORA
void destruirPila (Pila &p);
/* Libera toda la memoria ocupada por p. */


#endif /* _PILA_H */
```

Aplicación: Balanceo de paréntesis

Un compilador para un lenguaje de programación, entre otras cosas, verifica la corrección sintáctica de programas escritos en ese lenguaje.

Frecuentemente la ausencia de una “{”, por ejemplo, puede hacer generar a un compilador una larga lista de diagnósticos, no identificando, sin embargo, el error real.

Una herramienta muy útil en esta situación sería un procedimiento que verifique si el programa a compilar es sintácticamente balanceado.

Aplicación (cont.)

Obviamente no es muy conveniente escribir un programa sólo para resolver este problema, pero veremos que es muy fácil, haciendo uso de un stack, resolver problemas de este tipo.

Problema:

Dada una lista de caracteres que sólo puede contener los elementos **(**, **)**, **[**, **]**, **{** y **}**, deseamos construir un procedimiento que verifique que la expresión es balanceada, donde, por ejemplo, las secuencias **[()]** y **(){}** son correctas, pero **[(])** y **{(** no lo son.

Especificación informal

Construya un stack vacío.

Si la lista no es vacía, obtenga el primer elemento de la misma, y,

- si es un símbolo de apertura: (, [, {, agréguelo al stack.
- Si es un símbolo de clausura, entonces,
  - si el stack es vacío la expresión no es balanceada.
  - Sino, si el tope del stack no es el correspondiente símbolo de apertura tampoco lo es.
  - En el caso contrario, borre el tope del stack y siga inspeccionando la lista.

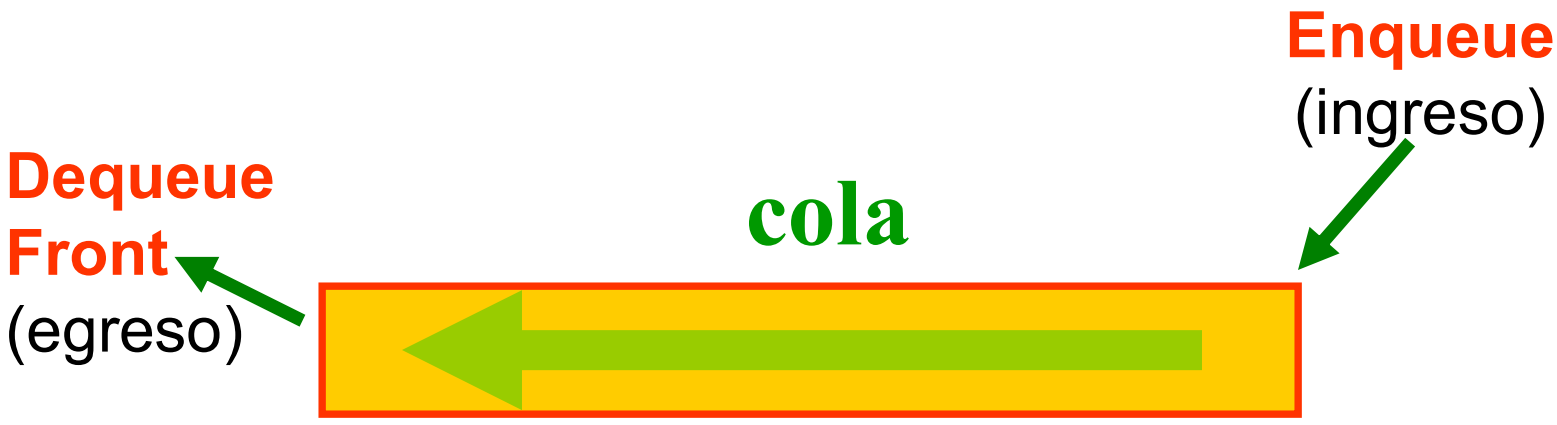
Si la lista es vacía y el stack también lo es, entonces es una expresión correcta. Si el stack no es vacío la expresión no es balanceada.

TAD COLA (QUEUE)

Definición

Una ***Queue*** (cola) es otra clase especial de lista, donde los elementos son insertados en un extremo (el final de la cola) y borrados en el otro extremo (el frente de la cola).

Otro nombre usado para este tipo abstracto es el de lista FIFO (first-in-first-out).



Definición

Las operaciones para una cola son análogas a las de stack, la diferencia sustancial es que las inserciones son efectuadas al final de la lista.

La terminología tradicional para stacks y colas es también diferente.

Operaciones

El TAD Queue incluye las siguientes operaciones:

- la operación que construye una cola vacía, **`Empty`**.
- **`Enqueue`**, que inserta un elemento al final de la cola.
- **`Front`**, que retorna el elemento que se encuentra en el comienzo de la cola.
- **`Dequeue`**, que borra el primer elemento de la cola.
- **`IsEmpty`**, que testea si la cola es vacía.
- **`IsFull`**, que testea si la cola está llena (si es una cola acotada).
- Una operación destructora, para eliminar una cola y liberar la memoria que ésta ocupa.

NOTAS:

- *Enqueue* podría no especificar donde se hacen las inserciones y en este caso las operaciones selectoras deberían referirse al primer elemento ingresado (el más antiguo).
- Si se trata de una cola acotada, *Enqueue* tendría precondición.

Algunas Aplicaciones

- Prácticamente, toda fila real es (supuestamente) una cola: “se atiende al primero que llega”.
- Las listas de espera son en general colas (por ejemplo, de llamadas telefónicas en una central) .
- Los trabajos enviados a una impresora se manejan generalmente siguiendo una política de cola (suponiendo que los trabajos no son cancelados).
- Dado un servidor de archivos en una red de computadoras, los usuarios pueden obtener acceso a los archivos sobre la base de que el primero en llegar es el primero en ser atendido, así que la estructura es una cola (no siempre se usa este esquema).
- Existe toda una rama de las matemáticas, denominada *teoría de colas*, que se ocupa de hacer cálculos probabilistas de cuánto tiempo deben esperar los usuarios en una fila, cuán larga es la fila.....

Conjuntos y Diccionarios

El TAD SET (Conjunto)

- En el diseño de algoritmos, la noción de conjunto es usada como base para la formulación de tipos de datos abstractos muy importantes.

- Un conjunto es una colección de elementos (o miembros), los que a su vez pueden ellos mismos ser conjuntos o si no elementos primitivos, llamados también átomos.
- Todos los elementos de un conjunto son distintos, lo que implica que un conjunto no puede contener dos copias del mismo elemento.

El TAD SET

- Una notación usual para exhibir un conjunto es listar sus elementos de la siguiente forma: {1, 4}, que denota al conjunto cuyos elementos son los naturales 1 y 4. Es importante destacar que los conjuntos no son listas: el orden en que los elementos de un conjunto son listados (orden posicional) no es relevante ({4, 1} = {1,4}) y en los conjuntos no hay elementos repetidos.
- **Lists** (orden y repetición posible de elementos)
- **MultiSets** (no hay orden pero si se permite repet.)
- **Sets** (no hay orden ni repetición de elementos)

El TAD SET

- La relación fundamental en teoría de conjuntos es la de pertenencia, la que usualmente se denota con el símbolo $\in$. Es decir, a $\in$ A significa que a es un elemento del conjunto A. El elemento a puede ser un elemento atómico u otro conjunto, pero A tiene que ser un conjunto.
- Un conjunto particular es el conjunto vacío, usualmente denotado $\varnothing$, que no contiene elementos.
- Las operaciones básicas sobre conjuntos son **unión, intersección y diferencia**.

El TAD SET

- **Vacio** c: construye el conjunto c vacío;
- **Insertar x c: agrega x a c, si no estaba en el conjunto;**
- **EsVacio** c: retorna true si y sólo si el conjunto c está vacío;
- **Pertenece** x c: retorna true si y sólo si x está en c;
- **Borrar** x c: elimina a x del conjunto c, si estaba;
- **Destruir** c: destruye el conjunto c, liberando su memoria;
- **Operciones para la unión, intersección y diferencia de conjuntos, entre otras.**

Especificación del TAD SET

- Otras operaciones adicionales que suelen considerarse para conjuntos son, entre otras, las siguientes:
- **Borrar**: dado un elemento lo elimina del conjunto, si es éste pertenece al mismo. También **Borrar_min** y **Borrar_max** (borran el mínimo y el máximo elem).
- **Min (Max)**: devuelve el elemento menor (mayor) del conjunto. Pre-condición: el conjunto es no vacío. Esta operación requiere un orden lineal sobre los elementos (o sobre sus claves).
- **Igual**: dado un conjunto, devuelve true si y sólo si el conjunto parámetro es igual al cual se le aplica el método. La igualdad es una operación útil y necesaria para la mayoría de los TADs.
- **Inclusiones, Cardinalidad, …**

Diccionarios y Colas de Prioridad

- Hay dos TADs especialmente utilizados, basados en el modelo de conjuntos:

Los **diccionarios** y las **colas de prioridad**

Diccionarios y Colas de Prioridad

- A menudo lo que se necesita es simplemente manipular un conjunto de objetos al que periódicamente se le agregan o quitan elementos. También es usual que uno desee verificar si un determinado elemento forma parte o no del conjunto.
- Un Tad Set con las operaciones Vacio, Insertar, EsVacio, Borrar y Pertenece recibe el nombre de ***diccionario***.
- (Una cola de prioridad es esencialente un Tad Set con las operaciones Vacio, Insertar, EsVacio, Borrar_Min y Min)

El TAD Diccionario

- **Vacio** d: construye el diccionario (conjunto) vacío
- **Insertar x d: agrega x a d, si no estaba en el diccionario**
- **EsVacio** d: retorna true si y sólo si el diccionario d está vacío
- **Pertenece** x d: retorna true si y sólo si x está en d
- **Borrar** x d: elimina a x del diccionario d, si estaba
- **Destruir** d: destruye el diccionario d, liberando su memoria.

Sobre TADs, recordar: