# TEMA 1
# Introducción a los TADs.
# Los tipos lineales

PROGRAMACIÓN Y ESTRUCTURAS DE DATOS



## Introducción. Los tipos lineales

- 1. Introducción a los TADs
- 2. Vectores
- 3. Listas
- 4. Pilas
- 5. Colas

Lista

$\boxed{A} \rightarrow \boxed{B} \rightarrow \boxed{C}$

SINTAXIS { CREAR LISTA: () → LISTA
Insertar lista (LISTA, ITEM) → LISTA.

SEMÁNTICA
(Como funciona)



# 1. Introducción a los TADs

- TAD: Tipo Abstracto de Datos
- *Tipo de datos*:
  - Clasifica los objetos de los programas (variables, parámetros, constantes) y determina los valores que pueden tomar
  - También determina las operaciones que se aplican
    - Entero: operaciones aritméticas enteras (suma, resta, …)
    - Booleano: operaciones lógicas (y, o, …)
- *Abstracto*:
  - La manipulación de los datos sólo dependen del comportamiento descrito en su especificación (*qué hace*) y es independiente de su implementación (*cómo se hace*)
  - Una especificación → Múltiples implementaciones





# 1. Introducción a los TADs

- Especificación de un TAD:
  - Consiste en establecer las propiedades que lo definen
  - Para que sea útil debe ser:
    - Precisa: sólo produzca lo imprescindible
    - General: sea adaptable a diferentes contextos
    - Legible: sea un comunicador entre especificador e implementador
    - No ambigua: evite problemas de interpretación
  - Definición informal (lenguaje natural) o formal (algebraica)

# 1. Introducción a los TADs

- Implementación de un TAD:
  - Consiste en determinar una representación para los valores del tipo y en codificar sus operaciones a partir de esta representación
  - Para que sea útil debe ser:
    - Estructurada: facilita su desarrollo
    - Eficiente: optimiza el uso de recursos → Evaluación de distintas soluciones mediante la complejidad (espacial y temporal)
    - Legible: facilita su modificación y mantenimiento





# 1. Introducción a los TADs

ESPECIFICACIÓN ALGEBRAICA (I)

- Especificación algebraica (ecuacional): establece las propiedades de un TAD mediante ecuaciones con variables cuantificadas universalmente, de manera que las propiedades dadas se cumplen para cualquier valor que tomen las variables
- Pasos:
  - Identificación de los objetos del TAD y sus operaciones (declaración del TAD, módulos que usa, parámetros)
  - Definición de la signatura (sintaxis) de un TAD (nombre del TAD y perfil de las operaciones)
  - Definición de la semántica (significado de las operaciones)
- Operación: es una función que toma como parámetros (entrada) cero o más valores de diversos tipos, y produce como resultado un solo valor de otro tipo. El caso de cero parámetros representa una constante del tipo de resultado

# ESPECIFICACIÓN ALGEBRAICA PILA.

SINTAXIS

PILA: CREARPILA : $() \rightarrow$ PILA

APILAR : PILA, ITEM $\rightarrow$ PILA

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (II)

MODULO …
    USA …
    PARAMETRO TIPO …
        OPERACIONES
            …
            …
    FPARAMETRO
    TIPO (GÉNERO) …
    OPERACIONES
        …
        …
FMODULO





# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (III)

MODULO NATURAL1
    TIPO natural
    OPERACIONES
        cero : → natural
        suc : natural → natural
FMODULO

¡EJEMPLO!

Mediante aplicación sucesiva de cero y suc se obtienen los distintos valores del tipo:

cero, suc(cero), suc(suc(cero)), …

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (IV)

MODULO NATURAL2
    TIPO natural
    OPERACIONES
        cero : → natural
        suc : natural → natural
        suma : natural natural → natural
FMODULO

¡NO SABEMOS SUMA LO QUE HACE!

¿suma(cero, suc(cero)) y suc(cero) denotan valores distintos?

¿ suma(cero, suc(cero)) y suma(suc(cero), cero) denotan el mismo valor?





# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (V)

- Solución:
  - Utilización de ecuaciones de la forma $t_1 = t_2$, donde $t_1$ y $t_2$ son términos sintácticamente correctos del mismo tipo
  - Semánticamente, expresa que el valor construido mediante el término $t_1$ es el mismo que el valor construido mediante el término $t_2$
  - Para no tener que escribir infinitas ecuaciones, se admite que los términos que aparecen en una ecuación tengan variables

# 1. Introducción a los TADs

ESPECIFICACIÓN ALGEBRAICA (VI)

MODULO NATURAL3
  TIPO natural
  OPERACIONES
    cero : → natural
    suc : natural → natural
    suma : natural natural → natural
  VAR
    x, y: natural
  ECUACIONES
    suma(x, cero) = x
    suma(cero, x) = x
    suma(x, suc(y)) = suc(suma(x, y))
FMODULO



Es recursivo, la propia expresión se auto ejecuta hasta llegar al caso base. Por tanto hay que reducir la expresión a los casos base.



# 1. Introducción a los TADs

EJERCICIOS

- Sea el conjunto de los números *naturales* con las operaciones *cero* y *suc*. Define la sintaxis y la semántica de las operaciones "==" y "<=" que permiten realizar una ordenación de los elementos del conjunto

# EJERCICIO == Y <=

IMPORTANTE!

## SINTAXIS

==: NATURAL NATURAL → BOOLEANO

<=: NATURAL NATURAL → BOOLEANO

## VARIABLES

X, Y: NATURAL

## OPERACIONES

==(CERO, CERO) → TRUE

==(SUC(X), CERO) → FALSE

==(CERO, SUC(Y)) → FALSE

==(SUC(X), SUC(Y) → ==(X, Y)

<=(CERO, CERO) → TRUE

<=(SUC(X), CERO) → FALSE

<=(CERO, SUC(Y)) → TRUE

<=(SUC(X), SUC(Y) → <=(X, Y)

# 1. Introducción a los TADs

## EJERCICIOS

- Completa en esta misma hoja las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de: *resta* en el conjunto de los números Naturales en el que sólo existen las operaciones *cero:* $\rightarrow$ *natural* y la operación *suc: natural* $\rightarrow$ *natural* (devuelve el sucesor de un número Natural). Se asume que el primer operando de la *resta* es siempre mayor o igual que el segundo.

*resta(natural,natural)* $\rightarrow$ *natural*

*resta(    ,    ) = ................................................*

*resta(    ,    ) = ................................................*

Resta(X,0)=X

Resta(suc(X),suc(Y))=Resta(x,y)





# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (VII)

- ¿Cómo podemos estar seguros de que no son necesarias más ecuaciones?
- Propiedades importantes: *consistencia* y *completitud*
  - Si se ponen ecuaciones de más, se pueden igualar términos que están en clases de equivalencia diferentes, mientras que si se ponen de menos, se puede generar un número indeterminado de términos incongruentes con los representantes de las clases existentes

# 1. Introducción a los TADs

ESPECIFICACIÓN ALGEBRAICA (VIII)

- Clasificación de las operaciones:
  - Constructoras: devuelven un valor del tipo
    - Generadoras: permiten generar, por aplicaciones sucesivas, todos los valores del TAD a especificar
    - Modificadoras: el resto
  - Consultoras: devuelven un valor de un tipo diferente
- En general, las operaciones modificadoras y consultoras se especifican en términos de las generadoras. En ocasiones, una operación modificadora puede especificarse en términos de otras modificadoras o consultoras. Diremos que se trata de una operación *derivada*





# 1. Introducción a los TADs

ESPECIFICACIÓN ALGEBRAICA (IX)

- Ecuación condicional: es equivalente a un conjunto finito de ecuaciones no condicionales

```
si (n1 <> n2) entonces
    saca(añade(s, n1), n2) = añade(saca(s, n2), n1)
sino
    saca(añade(s, n1), n2) = saca(s, n2)
fsi
```

# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (X)

- Operaciones auxiliares: se introducen en una especificación para facilitar su escritura y legibilidad. Son invisibles para los usuarios del TAD (también se les llama ocultas o privadas)





# 1. Introducción a los TADs

## ESPECIFICACIÓN ALGEBRAICA (XI)

- Tratamiento de errores: puede ocurrir que alguna operación sea una función parcial (no se puede aplicar sobre ciertos valores del dominio de los datos)

```
MODULO NATURAL4
   TIPO natural
   OPERACIONES
         cero :  → natural
         suc, pred : natural → natural
         suma, mult : natural  natural → natural
   VAR x, y: natural;
   ECUACIONES
         suma(cero, x) = x
         suma(x, cero) = x
         suma(x, suc(y)) = suc(suma(x, y))
         mult(cero, x) = cero
         mult(x, cero) = cero
         mult(suc(y), x) = suma(mult(y, x), x)
         pred(suc(x)) = x
FMODULO
```

¿Cuánto vale pred(cero)?

# 1. Introducción a los TADs

ESPECIFICACIÓN ALGEBRAICA (XII)

- Tratamiento de errores:
  - Se añade una constate a la signatura que modeliza un valor de error: $error_{nat} \rightarrow natural$
  - Se añade una ecuación que completa la especificación de pred: *pred(cero)* $= error_{nat}$
  - Se supondrá que los valores sobre los que se aplica una operación en una ecuación normal están libres de error





# 1. Introducción a los TADs

IMPLEMENTACIÓN (I)

- Dada una especificación de un tipo, se pueden construir diversas implementaciones
- Cada implementación se define en un módulo diferente, llamado módulo de implementación
- La construcción de estos módulos consta de dos fases:
  - Elección de una representación para los diferentes tipos definidos en la especificación
  - Codificación de las operaciones en términos de la representación elegida

# 1. Introducción a los TADs

IMPLEMENTACIÓN (II)

- Mecanismos de abstracción en los lenguajes de programación:
  - Encapsulamiento de la representación del TAD
  - Ocultación de la información, para limitar las operaciones posibles sobre el TAD
  - Genericidad, para lograr implementaciones genéricas válidas para distintos tipos
  - Herencia, para reutilizar implementaciones
- Los lenguajes de programación tradicionales (Fortran, Basic, Pascal, C) resultan ineficientes para utilizar los mecanismos de abstracción
- Es necesario emplear lenguajes modernos (ADA, C++, Java, C#)



Constructoras = mismo tipo
Consultoras = tipo distinto

Puede salir en el examen, tipo test.



# 1. Introducción a los TADs

Preguntas de tipo test: Verdadero vs. Falso

- EsVacia: PILA → BOOLEAN. Si P y Q son pilas: Q = EsVacia ( P ), es un uso sintácticamente correcto de la operación F
- En la especificación de un TAD, una operación consultora devuelve un valor del tipo definido F
- Sea el siguiente TAD:

  *MÓDULO NATURALEXAMEN*
  *TIPO natural*
  *OPERACIONES*
  *uno: → natural; siguiente: natural → natural*
  *sumar: natural natural → natural*
  *FMÓDULO*

  Si *N* es un *natural*: *N = sumar(uno,siguiente(uno))* es un uso sintácticamente incorrecto de la operación *sumar* F

# 2. Vectores

- Un vector es un conjunto ordenado de pares <índice, valor>. Para cada índice definido dentro de un rango finito existe asociado un valor. En términos matemáticos, es una correspondencia entre los elementos de un conjunto de índices y los de un conjunto de valores





# 2. Vectores

## ESPECIFICACIÓN ALGEBRAICA

```
MODULO VECTOR USA BOOL, ENTERO
//en todas las ecuaciones, c ≤ i, j ≤ f
    PARAMETRO TIPO item
        OPERACIONES
            c, f: → int    //límites inf. y sup.
            error( ) → item
    FPARAMETRO

    TIPO vector

    OPERACIONES
      crear( ) → vector
      asig( vector, int, item ) → vector
      recu( vector, int ) → item
      esvaciapos( vector, int ) → bool
```

```
VAR v: vector; i, j: int; x, y: item;
ECUACIONES
    si ( i <> j ) entonces
        asig( asig( v, i, x ), j, y ) = asig( asig( v, j, y ), i, x )
    si no asig( asig( v, i, x ), j, y ) = asig( v, i, y ) fsi

    recu( crear( ), i ) = error( )
    recu( asig(v, i, x ), j )    NOS ASEGURA QUE EL VECTOR
        si ( i == j ) entonces x    NO ES VACÍO.
        si no  recu( v, j ) fsi

    esvaciapos( crear( ), i ) = CIERTO
    esvaciapos( asig( v, i, x ), j )
        si ( i == j ) entonces FALSO
        si no esvaciapos( v, j ) fsi

FMODULO
```

# 2. Vectores

## REPRESENTACIÓN DE VECTORES

```
//Vector de item

const int kTam = 10;
class TVector {
      friend ostream& operator << ( ostream&, TVector&);

public:
TVector( );
TVector( const TVector &v );
˜TVector( );
TVector& operator =( TVector &v );

TItem & Recu( int i );
void Asig( int i, TItem c );
bool Esvaciapos( int i );

private:
TItem  fv[ kTam ];      //tamaño fijo
// TItem *fv;      tamaño dinámico
int fLong;
};
```

SOBRECARGA DE LA ASIGNACIÓN.

Cambio de Asig y Recu por la sobrecarga del operador corchete:
TItem & operator [] ( int i );

```
TItem&
TVector::operator[] (int indice){
  if (indice>=1 && indice<=fLong)
    return (fv[indice-1]);
  else
    return (error); }
```

SÓLO FUNCIONA PARA VECTOR NO CONSTANTE





# 2. Vectores

## EJERCICIOS *eliminar*

- Sea un vector de números naturales. Utilizando exclusivamente las operaciones *asignar* y *crear*, define la sintaxis y la semántica de la operación *eliminar* que borra las posiciones pares del vector marcándolas con "0" (para calcular el resto de una división, se puede utilizar la operación MOD)

ELIMINAR: VECTOR → VECTOR
VAR v: VECTOR; i: ENTERO; x: ENTERO
ELIMINAR(CREAR()) = CREAR()
ELIMINAR(ASIGNAR(v,i,x)) =
SI (i MOD 2) == 0
    ASIG(ELIMINAR(v),i,0)
SI NO
    ASIG(ELIMINAR(v),i,x)
FSI

LA CLAVE ES HACER UN EJEMPLO Y APLICARLE LA OPERACIÓN.

Sea un vector de números naturales. Utilizando exclusivamente las operaciones *asignar* y *crear*, define la sintaxis y la semántica de la operación *eliminar* que borra las posiciones pares del vector marcándolas con "0" (para calcular el resto de una división, se puede utilizar la operación MOD)

ELIMINAR (vector) → vector

Sintaxis

Var v: vector; i: natural; x: item

ELIMINAR (crear ()) = crear ()

ELIMINAR (asignar (v, i, x)) =

si ((i MOD 2) == 0)

ASIGNAR (ELIMINAR (v), i, 0)

sino

ASIGNAR (ELIMINAR (v), i, x)

FSi.

→ Semántica

Utilizando las operaciones definidas en clase para la definición del tipo vector definir la sintaxis y la semántica de la operación *palindromo* que indica si un vector de naturales con 100 elementos es palíndromo. Por ejemplo, el vector 1,25,12,3,3,12,25,1 es palíndromo (se ha simplificado el ejemplo con un vector de 8 elementos). IMPORTANTE: se asume que el vector está creado con tamaño 100, está lleno y el rango de las posiciones es de 1 a 100. Se premiará la solución temporalmente más eficiente.

Palíndromo (vector) → Booleano

Sintaxis

Var v: vector; i, x: natural;

Palíndromo (crear ()) = V

Palíndromo (asignar (v, i, x)) =

si i <= 50

si (recu (v, 101 - i) == x)

Palíndromo (v, i)

sino

falso

sino

Verdadero.

→ Semántica

ESTE ES EL EJEMPLO SOBRE EL QUE HAY QUE APLICAR LA OPERACIÓN.



## 2. Vectores

EJERCICIOS *operación M*

- Dada la sintaxis y la semántica de la operación M que actúa sobre un vector:

M (vector) → vector

Var v: vector; i: int; x: item;

M(crear()) = crear()

si i == 1 entonces

M(asig(v, i, x)) = M(v)

si no M(asig(v, i, x)) = asig(M(v), i –1, x)

a) Aplicar la operación M al siguiente vector:

asig(asig(asig(crear(), 3, a), 1, b), 2, c)

b) Explicar en un párrafo qué es lo que hace la operación M

LO QUE SE HACE ES APLICAR LA M POR PASOS, MOVIÉNDOLA

M(ASIG(ASIG(ASIG((CREAR(),3,a),1,b),2,c)) =

ASIG(M(ASIG(ASIG((CREAR(),3,a),1,b),1,c)) =

ASIG(M(ASIG((CREAR(),3,a),1,c)) =

ASIG(ASIG(M((CREAR(),2,a),1,c) =

ASIG(ASIG((CREAR(),2,a),1,c).



| B | C | A |
|---|---|---|

inicio

| C | A |
|---|---|

M(V)



## 2. Vectores

EJERCICIOS *palíndromo*

- Utilizando las operaciones definidas en clase para la definición del tipo vector definir la sintaxis y la semántica de la operación *palindromo* que indica si un vector de naturales con 100 elementos es palíndromo. Por ejemplo, el vector 1,25,12,3,3,12,25,1 es palíndromo (se ha simplificado el ejemplo con un vector de 8 elementos). IMPORTANTE: se asume que el vector está creado con tamaño 100, está lleno y el rango de las posiciones es de 1 a 100. Se premiará la solución temporalmente más eficiente.

# 2. Vectores

Preguntas de tipo test: Verdadero vs. Falso

- El tipo de datos vector se define como un conjunto en el que sus componentes ocupan posiciones consecutivas de memoria
- Un vector es un conjunto ordenado de pares <índice, valor>





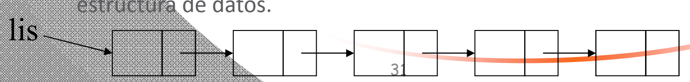
# 3. Listas

- Una lista es una secuencia de cero o más elementos de un mismo tipo de la forma $e_1, e_2, \ldots, e_n \quad \forall n \geq 0$
- De forma más general: $e_p, e_{sig(p)}, \ldots, e_{sig(sig \ldots n) \ldots (p))}$
  Al valor *n* se le llama *longitud de la lista*. Si $n = 0$ tenemos una *lista vacía*. A $e_1$ se le llama primer elemento, y a $e_n$ último elemento
- Propiedades:
  - Se establece un orden secuencial estricto sobre sus elementos por la *posición* que ocupan dentro de la misma. De esta forma $e_i$ precede a $e_{sig(i)}$ para i = 1, 2, ... , n-1 y $e_{sig(i)}$ sucede a $e_i$ para i = 1, 2, ... , n-1. Por último, el elemento $e_i$ ocupa la posición i
  - La lista nos permite conocer cualquier elemento de la misma *accediendo a su posición*, algo que no podremos hacer con las pilas y con las colas. Utilizaremos el concepto generalizado de posición, con una ordenación definida sobre la misma, por lo tanto no tiene por qué corresponderse exactamente con los números enteros, como clásicamente se ha interpretado este concepto
- Una *lista ordenada* es un tipo especial de lista en el que se establece una relación de orden definida entre los items de la lista

# 3. Listas

## REPRESENTACIÓN DE LISTAS

- El TAD *lista* se utiliza para almacenar listas de un número **variable** de objetos.
- Representación secuencial (internamente un *array*)
  - **A partir de tipos base ("arrays")**
  - **A partir de tipos definidos por el usuario ("tvector" –herencia o layering –)**
- Representación enlazada (internamente *punteros a nodo*)
  - **A partir de tipos base ("punteros a nodo")**
  - Cada objeto se almacena en un nodo, que se enlaza con el siguiente.
  - La lista es un puntero al primer nodo, o NULL si está vacía.
  - Un nodo es un contenedor para almacenar información, y tiene dos partes:
    - La información del objeto que se desea guardar.
    - Uno o más punteros para enlazar el nodo con otros nodos y construir la estructura de datos.



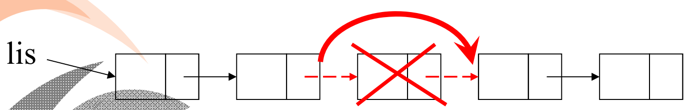




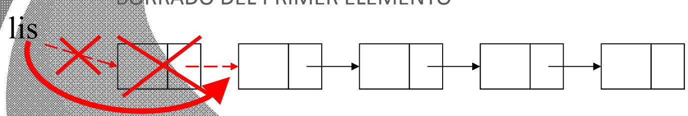

# 3. Listas

## REPRESENTACIÓN ENLAZADA DE LISTAS (I)

- Operaciones Básicas sobre las listas:
  - Averiguar si la lista está vacía
  - Búsqueda de un elemento
  - Inserción y Borrado de un elemento: hay que distinguir si es al principio, en una posición intermedia o al final de la lista
    - BORRADO DE UN ELEMENTO INTERMEDIO O FINAL



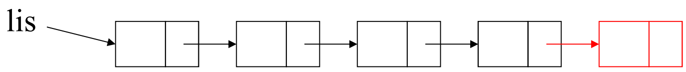

    - BORRADO DEL PRIMER ELEMENTO



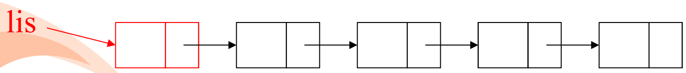
# 3. Listas

## REPRESENTACIÓN ENLAZADA DE LISTAS (II)

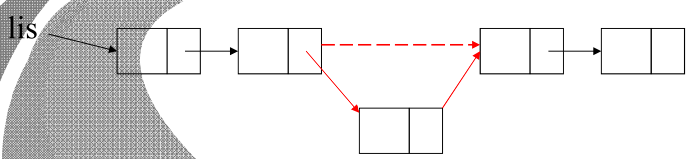
- INSERCIÓN DE UN ELEMENTO AL FINAL




- INSERCIÓN DE UN ELEMENTO AL PRINCIPIO




- INSERCIÓN EN UNA POSICIÓN INTERMEDIA








# 3. Listas

## LISTAS ORDENADAS

- Una lista ordenada es una lista que en todo momento se mantiene ordenada.
- Se consigue insertando siempre los nodos en la posición que les corresponda según el orden definido (los borrados no desordenan la lista).
- Ventajas frente a la lista simple:
  - El tiempo medio de búsqueda se reduce (no es necesario recorrer toda la lista para comprobar que un nodo no está).
  - No es necesario ordenar la lista.

# 3. Listas

## ESPECIFICACIÓN ALGEBRAICA (I)

MODULO LISTA USA BOOL, NATURAL
  PARAMETRO TIPO item, posicion
    OPERACIONES
      == ( posicion, posicion ) → bool
      error_item( ) → item
      error_posicion( ) → posicion
  FPARAMETRO
TIPO lista
OPERACIONES
    crear( ) → lista
    inscabeza( lista, item ) → lista
    esvacia( lista ) → bool
    concatenar( lista, lista ) → lista
    longitud( lista ) → natural
    primera, ultima( lista ) → posicion
    anterior, siguiente( lista, posicion ) → posicion
    insertar( lista, posicion, item ) → lista
    borrar( lista, posicion ) → lista
    obtener( lista, posicion ) → item





# 3. Listas

## ESPECIFICACIÓN ALGEBRAICA (II)

**VAR** $L_1$, $L_2$: lista; x: item; p: posicion;

**ECUACIONES**

esvacia( crear( ) ) = CIERTO
esvacia( inscabeza( $L_1$, x ) ) = FALSO

concatenar( crear( ), $L_1$ ) = $L_1$
concatenar( $L_1$, crear( ) ) = $L_1$
concatenar( inscabeza( $L_1$, x ), $L_2$ ) = inscabeza( concatenar( $L_1$, $L_2$ ), x )

longitud( crear( ) ) = 0
longitud( inscabeza( $L_1$, x ) ) = 1 + longitud( $L_1$ )

primera( crear( ) ) = error_posicion( );   ultima( crear( ) ) = error_posicion( )
**si** esvacia( $L_1$ ) **entonces**
    ultima( inscabeza( $L_1$, x ) = primera( inscabeza( $L_1$, x ) )
**si no** ultima( inscabeza( $L_1$, x ) = ultima( $L_1$ )

anterior( $L_1$, primera( $L_1$ ) ) = error_posicion( );   siguiente( $L_1$, ultima( $L_1$ ) ) = error_posicion( )
**si** p != ultima( $L_1$ ) **entonces** anterior( $L_1$, siguiente( $L_1$, p ) ) = p
anterior( inscabeza( $L_1$, x ), primera( $L_1$ ) ) = primera( inscabeza( $L_1$, x ) )

# 3. Listas

## ESPECIFICACIÓN ALGEBRAICA (III)

**si** p != primera( $L_1$ ) **entonces** siguiente( $L_1$, anterior( $L_1$, p ) ) = p
siguiente( inscabeza( $L_1$, x ), primera( inscabeza( $L_1$, x ) ) = primera( $L_1$ )

insertar( crear( ), p, x ) = crear( )
**si** p == primera( inscabeza( $L_1$, x ) ) **entonces**
insertar( inscabeza( $L_1$, x ), p, y ) = inscabeza( inscabeza( $L_1$, y ), x )
**si no** insertar( inscabeza( $L_1$, x ), p, y ) = inscabeza( insertar( $L_1$, p, y ), x )

borrar( crear( ), p ) = crear( )
**si** p == primera( inscabeza( $L_1$, x ) ) **entonces**
borrar( inscabeza( $L_1$, x ), p ) = $L_1$
**si no** borrar( inscabeza( $L_1$, x ), p ) = inscabeza( borrar( $L_1$, p ), x )

obtener( crear( ), p ) = error_item( )
**si** p == primera( inscabeza( $L_1$, x ) ) **entonces**
obtener( inscabeza( $L_1$, x ), p ) = x
**si no** obtener( inscabeza( $L_1$, x ), p ) = obtener( $L_1$, p )

**FMODULO**





# 3. Listas

## ENRIQUECIMIENTO DE LAS LISTAS

**OPERACIONES**
sublista( lista, posicion, natural ) → lista
inversa (lista ) → lista
**VAR** L: lista; x, y: item; n: natural; p: posicion;
**ECUACIONES**
sublista( L, p, 0 ) = crear( )
sublista( crear( ), p, n ) = crear( )
**si** p == primera( inscabeza( L, x ) ) **entonces**
sublista( inscabeza( L, x ), p, n) = inscabeza( sublista( L, primera( L ), n - 1 ), x )
**si no** sublista( inscabeza( L, x ), p, n) = sublista( L, p, n )

inversa( crear( ) ) = crear ( )
inversa( inscabeza( crear( ), x ) ) = inscabeza( crear( ), x )
inversa( inscabeza( L, x ) ) = insertar( inversa( L ), ultima( inversa( L ) ), x )

QUITA_PARES DEVUELVE LA LISTA ORIGINAL DONDE LOS ELEMENTOS PARES SE HAN SUPRIMIDO.

QUITA_PARES (LISTA) → LISTA
VAR x: item , L: LISTA

QUITA_PARES (CREAR()) = CREAR()
QUITA_PARES (INSCABEZA (CREAR(), x)) = INSCABEZA (CREAR (), x))
QUITA_PARES (IC (IC (L, x) y)) = IC (QP(L1, y))

# 3. Listas

## REPRESENTACIÓN DE LISTAS (I)

```
class TLista {
 friend ostream&
      operator<<(ostream&, TLista&);
 friend class TPosicion;
 public:
  TLista();
  ~TLista();
  void InsCabeza(int);
  TPosicion Primera();
  int& Obtener(TPosicion&);
  void Borrar(TPosicion&);
 private:
  TNodo *lis;
};
```

```
class TNodo {
 friend class TLista; friend class TPosicion;
 public:
  TNodo(); ~TNodo();
 private:
  int dato;   TNodo *sig; };

class TPosicion {
 friend class TLista;
 public:
  TPosicion(); ~TPosicion();
  bool EsVacia();
  TPosicion Siguiente();
  TPosicion& operator=(TPosicion&);
 private:
  TNodo* pos; };
```





# 3. Listas

## REPRESENTACIÓN DE LISTAS (II)

```
TLista::TLista() {  lis = NULL; }

TLista::~TLista() {
 TPosicion p, q;
 q = Primera();
 while(!q.EsVacia()) {
  p = q;
  q = q.Siguiente();
  delete p.pos;
 }
 lis = NULL;
}
```

```
void
TLista::InsCabeza(int i) {
 TNodo* aux = new TNodo;
 aux→dato = i;
 if(lis == NULL) {
  aux→sig = NULL;
  lis = aux;
 }
 else {
  aux→sig = lis;
  lis = aux;
 }
}
```

# 3. Listas

## REPRESENTACIÓN DE LISTAS (III)

```
TPosicion
TLista::Primera() {
  TPosicion p;  p.pos = lis;
  return p;}
int&
TLista::Obtener(TPosicion& p) {
  return p.pos→dato;}
ostream&
operator<<(ostream& os, TLista& l) {
  TPosicion p;
  p = l.Primera();
  while(!p.EsVacia()) {
    os << l.Obtener(p) << ' ';
    p = p.Siguiente();}
  return os; }
```

```
TNodo::TNodo() {
  dato = 0;  sig = NULL; }

TNodo::~TNodo() {
  dato = 0;  sig = NULL; }

TPosicion::TPosicion() {  pos = NULL; }

TPosicion::~TPosicion() {
  pos = NULL; }

bool
TPosicion::EsVacia() {
  return pos == NULL; }
```



hay que comprobar que sea distinto de null.



# 3. Listas

## REPRESENTACIÓN DE LISTAS (IV)

```
TPosicion
TPosicion::Siguiente() {
  TPosicion p;
  p.pos = pos→sig;
  return p;
// ¿si pos es NULL?
}

TPosicion&
TPosicion::operator=(TPosicion& p) {
  pos = p.pos;
  return *this;
}
```

```
int
main(void)
{
  TLista l;
  l.InsCabeza(1); l.InsCabeza(3);
  l.InsCabeza(5); l.InsCabeza(7);
  cout << l << endl;
  TPosicion p;
  p = l.Primera();
  cout << "Primer elemento: "
     << l.Obtener(p) << endl;
  p = p.Siguiente();
  cout << "Segundo elemento: "
     << l.Obtener(p) << endl;
}
```

# Vectores y Listas

Aplicaciones reales

- Clasificación de los equipos de la liga de fútbol.
- Gestión de la lista de espera de un hospital para intervenciones quirúrgicas.

Clasificación 1ª división

| | EQUIPO | PTOS | PJ | PG | PE | PP | GF | GC |
|---|---|---|---|---|---|---|---|---|
| 1 | Barcelona | 50 | 19 | 16 | 2 | 1 | 53 | 12 |
| 2 | At. Madrid | 50 | 19 | 16 | 2 | 1 | 47 | 11 |
| 3 | Real Madrid | 47 | 19 | 15 | 2 | 2 | 53 | 21 |
| 4 | Ath.Bilbao | 36 | 19 | 11 | 3 | 5 | 32 | 24 |
| 5 | Villarreal | 34 | 19 | 10 | 4 | 5 | 37 | 21 |
| 6 | Real Sociedad | 32 | 19 | 9 | 5 | 5 | 36 | 28 |
| 7 | Sevilla FC | 30 | 19 | 8 | 6 | 5 | 36 | 30 |
| 8 | Valencia | 23 | 19 | 7 | 2 | 10 | 26 | 31 |
| 9 | Granada | 23 | 19 | 7 | 2 | 10 | 19 | 25 |
| 10 | Levante | 23 | 19 | 6 | 5 | 8 | 18 | 27 |
| 11 | Getafe | 23 | 19 | 7 | 2 | 10 | 20 | 31 |
| 12 | Espanyol | 22 | 19 | 6 | 4 | 9 | 22 | 25 |
| 13 | Osasuna | 21 | 19 | 6 | 3 | 10 | 17 | 29 |
| 14 | Málaga | 20 | 19 | 5 | 5 | 9 | 19 | 24 |
| 15 | Celta de Vigo | 19 | 19 | 5 | 4 | 10 | 23 | 31 |
| 16 | Almería | 19 | 19 | 5 | 4 | 10 | 21 | 38 |
| 17 | Elche | 18 | 19 | 4 | 6 | 9 | 17 | 28 |
| 18 | Valladolid | 16 | 19 | 3 | 7 | 9 | 21 | 33 |
| 19 | Rayo | 16 | 19 | 5 | 1 | 13 | 19 | 45 |
| 20 | Real Betis | 11 | 19 | 2 | 5 | 12 | 16 | 38 |





# 3. Listas

EJERCICIOS *borraultimo*

- Completa las ecuaciones que aparecen a continuación y que expresan el comportamiento de las operaciones de *borraultimo* (borra el último elemento de la lista) en una lista de acceso por posición:

La sintaxis de la operación es la siguiente:

borraultimo(lista) → lista

borraultimo(crear()) =.......................................................................................

**si** esvacia(l1) **entonces** borraultimo(inscabeza(l1,x)) =.................................

**si no** borraultimo(inscabeza(l1,x)) = ............................................................

Donde: x ∈ elemento                    l1 ∈ lista

# 3. Listas

EJERCICIOS *quita_pares*

- Definir la sintaxis y la semántica de la operación *quita_pares* que actúa sobre una lista y devuelve la lista original en la que se han eliminado los elementos que ocupan las posiciones pares





# 3. Listas

EJERCICIOS *operación X*

- Explicar qué hace la operación X, cuya sintaxis y semántica aparecen a continuación:

  X ( lista ) --> lista

  X ( crear ( ) ) --> crear ( )

  X ( inscabeza ( l, i ) ) <==>

  **si** ( longitud ( l ) == 0 ) **entonces** crear ( )

  **si no** inscabeza ( X ( l ), i )

  Donde: l ∈ lista, i ∈ item

- Simplificar la siguiente expresión: ( IC = inscabeza )

  X ( IC ( IC ( IC ( IC ( crear ( ), a ), b) ,c) ,d) )

BORRARÍA SÓLO 1.



# 3. Listas

Preguntas de tipo test: Verdadero vs. Falso

- La operación BorrarItem tiene la siguiente sintaxis y semántica:

  BorrarItem: LISTA, ITEM -> LISTA

  BorrarItem( Crear, i) = Crear

  BorrarItem( IC(L1,j), i) = si ( i == j ) entonces L1

  sino IC ( BorrarItem (L1, i ), j )

  Esta operación borra ~~todas~~ las ocurrencias del item que se encuentra en la lista

  F

- La complejidad temporal de obtener un elemento en un vector ordenado mediante búsqueda binaria o en una lista ordenada es la misma

  F

Lista secuencial.

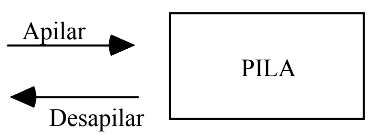


Vector aleatorio



# 4. Pilas

- Una pila es una lista en la que todas las inserciones y borrados se realizan en un único extremo, llamado *tope* o *cima*. Sabemos por tanto que el último elemento insertado en la pila será el primero en ser borrado de la misma, de ahí que también se les llame listas “LIFO” (Last In, First Out). También podemos conocer cuál es el elemento que se encuentra en la *cima*



# 4. Pilas

## ESPECIFICACIÓN ALGEBRAICA

MODULO PILA USA BOOL
PARAMETRO
TIPO item
OPERACIONES
error( ) → item
FPARAMETRO

TIPO pila

OPERACIONES
crear( ) → pila
apilar( pila, item ) → pila
desapilar( pila ) → pila
cima( pila ) → item
esvacia( pila ) → bool

**VAR** p: pila, e: item;
**ECUACIONES**
desapilar( crear( ) ) = crear( )
desapilar( apilar( p, e ) ) = p
cima( crear( ) ) = error( )
cima( apilar( p, e ) ) = e
esvacia( crear( ) ) = CIERTO
esvacia( apilar( p, e ) ) = FALSO

**FMODULO**





# 4. Pilas

## REPRESENTACIÓN SECUENCIAL DE PILAS (I)

- Representación secuencial (internamente un *array*)
  - **A partir de tipos base ("arrays")**
  - **A partir de tipos definidos por el usuario ("tvector" –herencia o layering –)**
- Tipos de algoritmos
  - **Realizando las inserciones por la primera componente. Ineficiente**
  - **Utilizando un cursor que indique la posición actual del primer elemento de la pila**
- Ventajas y desventajas
  - **Desventaja: tamaño máximo de la pila**
  - **Ventaja: sencillez de implementación**

# 4. Pilas

## REPRESENTACIÓN SECUENCIAL DE PILAS (II)

```
const kMax = 10;
class TPila {
    public:
        TPila( ); TPila( TPila & ); ~TPila( ); TPila& operator=( TPila &);
        TItem& Cima( );
        void Apilar( TItem& );
        . . .
    private:
        TItem fpila[ kMax ];   //tamaño fijo
        // TItem *fpila;   tamaño dinámico
        int ftope;
    };
TPila::TPila( ) {
    fpila = new TItem[ 10 ]; //sólo si el vector es dinámico
    ftope = 0;   }
void
TPila::Desapilar( ) {
        ftope --;   }
```





# 4. Pilas

## REPRESENTACIÓN ENLAZADA DE PILAS (I)

- Representación enlazada (internamente *punteros a nodo*)
  - **A partir de tipos base ("punteros a nodo")**
  - **A partir de tipos definidos por el usuario ("tlista" –herencia o layering–)**
- Ventajas
  - **Ventaja: no hay definido un tamaño para la pila**

# 4. Pilas

## REPRESENTACIÓN ENLAZADA DE PILAS (II)

```
class TPila {
     public:
          TPila( ); TPila( TPila & ); ~TPila( ); TPila& operator=( TPila &);
          TItem& Cima( );
          void Apilar( TItem& );
          . . .
     private:
          struct TNodo {
                    TItem dato;
                    TNodo *sig;   };
          TNodo *fp;
     };
TPila::TPila( ) {   fp = NULL;   }
void
TPila::Desapilar( ) {
          TNodo *aux;
          aux = fp;
          fp = fp -> sig;
          delete aux;   }
```





# 4. Pilas

## REPRESENTACIÓN ENLAZADA DE PILAS (III)

```
//HERENCIA PRIVADA
class TPila: private TLista {
  public:
    TPila( ); TPila( TPila & ); ~TPila( );
    void Apilar( TItem& );
    void Desapilar( );
         . . .
};

TPila::TPila( ): TLista( ) {   };

TPila::TPila( TPila &p ): TLista( p ) {   };

~TPila( ) {   }
void
TPila::Apilar( TItem &a ) { InsCabeza( a ); }

void
TPila::Desapilar( ) { Borrar( Primera( ) ); }
```

```
//LAYERING O COMPOSICIÓN
class TPila {
  public:
    TPila( ); TPila( TPila & ); ~TPila( );
    void Apilar( TItem& ); void Desapilar( );
         . . .
    private: TLista L;
};

TPila::TPila( ): L( ) {   };

TPila::TPila( TPila &p ): L( p.L ) {   };

~TPila( ) {   }
void
TPila::Apilar( TItem &a ) { L.InsCabeza( a ); }

void
TPila::Desapilar( ) { L.Borrar( L.Primera( ) ); }
```

SINTAXIS Y SEMÁNTICA OPERACIÓN BASE(PILA)

BASE(PILA) → ITEM.

BASE(CREAR()) → ERROR.

BASE(APILAR(CREAR(), e)) → e

BASE(APILAR(P, e)) → BASE(P)

# 4. Pilas

EJERCICIOS

- Dar la sintaxis y la semántica de la operación **base**, que actúa sobre una pila y devuelve la base de la pila (el primer elemento que se ha apilado)

BASE(PILA) → ITEM

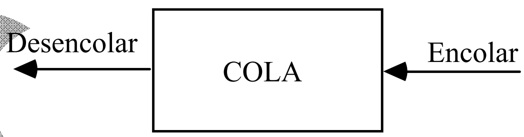
VAR P:PILA ; e:ITEM
BASE(CREAR())= ERROR()
BASE(APILAR(CREAR(),e)) = e
BASE(APILAR(P,e)) = BASE(P)





# 5. Colas

- Una cola es otro tipo especial de lista en el cual los elementos se insertan por un extremo (*fondo*) y se suprimen por el otro (*tope*). Las colas se conocen también como listas "FIFO" (First Int First Out). Las operaciones definidas sobre una cola son similares a las definidas para las pilas con la salvedad del modo en el cual se extraen los elementos



# 5. Colas

## ESPECIFICACIÓN ALGEBRAICA

MODULO COLA USA BOOL
    PARAMETRO
        TIPO item
        OPERACIONES
            error( ) → item
    FPARAMETRO

TIPO cola

OPERACIONES
    crear( ) → cola
    encolar( cola, item ) → cola
    desencolar( cola ) → cola
    cabeza( cola ) → item
    esvacia( cola ) → bool

**VAR** c: cola, x: item;

**ECUACIONES**
    desencolar( crear( ) ) = crear( )
    **si** esvacia( c ) **entonces**
        desencolar( encolar( c, x ) ) = crear ( )
    **si no** desencolar( encolar( c, x ) ) =
        encolar( desencolar ( c ), x )
    cabeza( crear( ) ) = error( )
    **si** esvacia( c ) **entonces**
        cabeza( encolar( c, x ) ) = x
    **si no** cabeza( encolar( c, x ) ) = cabeza( c )
    esvacia( crear( ) ) = CIERTO
    esvacia( encolar( c, x ) ) = FALSO

**FMODULO**



ENC(ENC(ENC(CREAR(),A),B),C) =

Tema 1. Introducción. Los tipos lineales

# 5. Colas

## ENRIQUECIMIENTO DE LAS COLAS

TRAZA

**OPERACIONES**
    concatena( cola, cola ) → cola

CONCATENA(C, ENCOLAR(CREAR(),X))

**VAR** c, q: cola;  x: item;

ENCOLAR(CONCATENA(C, CREAR()),X)
ENCOLAR(C,X)

**ECUACIONES**
    concatena( c, crear( ) ) = c
    concatena( crear( ), c ) = c
    concatena( c, encolar( q, x ) ) = encolar( concatena( c, q ), x )

$COPC(S, Q) \quad Q = E(CREAR(), x) \quad S = E(E(E(CREAR(), A), b), C)$

# 5. Colas

## REPRESENTACIÓN SECUENCIAL DE COLAS (I)

- Representación secuencial (internamente un *array*)
  - **A partir de tipos base ("arrays")**
  - **A partir de tipos definidos por el usuario ("tvector" –herencia o layering–)**
- Tipos de algoritmos
  - **Utilizando un array (*fv*) para almacenar los elementos y dos enteros (*tope* y *fondo*) para indicar la posición de ambos extremos**
    - **Inicializar: tope = 0; fondo = -1;**
    - **Condición de cola vacía: fondo < tope**
    - **Inserción: fondo++; fv[fondo] = x;**
    - **Borrado: tope ++;**

| | | | | |
|---|---|---|---|---|
| a | | | | |

| | | | | |
|---|---|---|---|---|
| a | b | | | |

| | | | | |
|---|---|---|---|---|
| a | b | c | | |

| | | | | |
|---|---|---|---|---|
| | b | c | | |

| | | | | |
|---|---|---|---|---|
| | | c | | |

| | | | | |
|---|---|---|---|---|
| | | c | e | |

| | | | | |
|---|---|---|---|---|
| | | c | e | f |





# 5. Colas

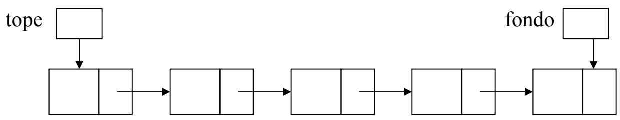
## REPRESENTACIÓN SECUENCIAL DE COLAS (II)

- Problema:
  - Hay huecos pero no puedo insertar
- Soluciones:
  - Cada vez que se borra un elemento, el resto se desplaza una posición a la izquierda para que *tope* siempre esté en la primera posición. ¿Qué problemas presenta esta solución? aumentamos la complejidad de la operación desencolar
  - Colas circulares. *Array* como un círculo en el que la primera posición sigue a la última. Condición de cola vacía *tope == fondo*
- Ventajas y desventajas
  - Desventaja: tamaño máximo de la cola
  - Ventaja: sencillez de implementación

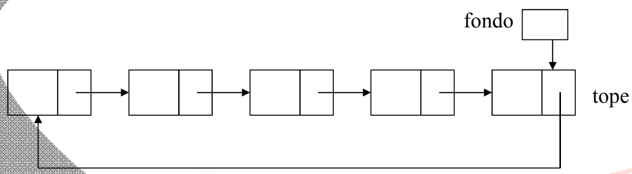
# 5. Colas

## REPRESENTACIÓN ENLAZADA DE COLAS (I)

- Representación enlazada (internamente *punteros a nodo*)
  - **A partir de tipos base (“punteros a nodo”)**




- **Colas circulares enlazadas, en las que sólo se necesita un puntero. El siguiente elemento apuntado por *fondo* es el primero a desencolar**








# 5. Colas

## REPRESENTACIÓN ENLAZADA DE COLAS (II)

- **A partir de tipos definidos por el usuario (“tlista” –herencia o layering–)**

- Ventajas
  - **Ventaja: no hay definido un tamaño para la cola**

# Pilas y Colas

## Aplicaciones reales

- Los editores de texto proporcionan normalmente un botón deshacer que cancela las operaciones de edición recientes y restablece el estado anterior del documento.

- Los Navegadores en Internet almacenan en una pila las direcciones de los sitios más recientemente visitados.






## EJERCICIOS

- Dada la clase *TVector* que almacena un vector dinámico de enteros y un entero que contiene la dimensión del vector, definid en C++:
  - La clase *TVector*
  - Constructor por defecto (dimensión 10 y componentes a -1)

EJERCICIOS

- Dada la clase *TPila* definid en C++ el método *Apilar*





# Pilas y colas

Preguntas de tipo test: Verdadero vs. Falso

- La semántica de la operación *cima* del tipo *pila* vista en clase es la siguiente:

  **VAR** p: pila, e: item;

  cima( crear( ) ) = error( )

  cima( apilar( p, e ) ) = cima( p )

  F

- Es posible obtener una representación enlazada de una cola utilizando un único puntero que apuntará al fondo de la cola

  V