

1º MATEMÁTICAS Y COMPUTACIÓN

# ESTRUCTURA DE DATOS

---

**PAULA GÓMEZ Y RUBÉN MARTINEZ**

## LECTURA 1

Ayuda en la prg. de software - soft contained subroutine

Roles de datos abstractos → Desarrollo del software  
flexible  
variable eficiente

Proceso de abstracción

A través de subrutinas uno puede describir una jerarquía de abstractividades sin embargo los subrutinos son limitados (no están preparados para la descripción de objetos abstractos)

separa el código similarmente, al nivel en que está implementado, es necesario complicar el código a través de la consideración del por qué

Típos de atributos → Descubren la representación de objetos y la implementación de las operaciones asociadas con ellos en términos de otros objetos y operaciones

→ Aquellos que especifican los nombres y definen el significado abstracto de las operaciones asociadas con un objeto (tipos tritik)

PROGRAM NAME		INPUT		OUTPUT	
		CONTROL		CONTROL	
				DESTINATION	
				SEQUENCE OPERATION	

CONDITION	CALCULATION	PROGRAM NAME	INPUT	OUTPUT
ELSE				
IF				
ELSE IF				
SCHEDULE				
SEQUENCE	ACTION	SEQUENCE	OPERATION	OBJECT CONDITION

INFORMATION	SEQUENCE	SEND	INVOCATION
OPERATION	OPERATION	OPERATION	CONTROL
CREATE DESTROY	GO TO EXIT	DESTINATION	M
INITIALIZE INSERT	ENTER	XXX	E
DELETE UPDATE	LEAVE		D R

XXX - can be any program name.

Término Datos de tipo abstracto, para referirnos a una clase de objetos definidos por la especificación de representación independiente

Cuando trabajamos con tipos desconocidos para nosotros es imposible. Por ello, necesitamos mecanismos para especificar la semántica de las operaciones de ese tipo, la mayoría pueden ser especificadas en dos tipos, operacional o de definición. En primer lugar, elabora un esquema de cómo realizarlo (en cuya de describir los tipos de propiedades). Así así tiene ventajas, la fácil construcción por parte de programadores (pero se hacen muy largas a medida que el programa se complica). El mayor problema es que casi siempre fuerzan a uno a sobreespecificar la abstracción y pueden llegar a eliminar la solución del problema.

El segundo, evita el problema a través del uso del álgebra. Las especificaciones algebraicas consisten en una especificación sintáctica y un conjunto de relaciones. La primera aporta información normalmente requerida (nombre, dominio y rango de operaciones). El segundo define los significados de las operaciones a través de sus relaciones con cada uno.

Ejemplo:

Será la especificación sintáctica:

NEW:	→ Queue
ADD:	Queue × Item → Queue
FRONT:	Queue → Item
REMOVE:	Queue → Queue
IS_EMPTY?:	Queue → Boolean

De manera axiomática:

- (1) IS\_EMPTY? (NEW) = true
- (2) IS\_EMPTY? (ADD(q,i)) = false
- (3) FRONT(NEW) = error
- (4) FRONT (ADD(q,i)) = if IS\_EMPTY? (q)  
then i  
else FRONT(q)
- (5) REMOVE(NEW) = error
- (6) REMOVE (ADD(q,i)) = if IS\_EMPTY? (q)  
then NEW  
else ADD(REMOVE(q),i)

Una vez construido es fácil determinar si es consistente y completa.

\* Para ser inconsistente → contradicción

\* Para ser incompleto → No se sabe todo la información

\* Queue : Almacenamiento SI/NO

# → Ejemplo más desarrollado

## ① Operaciones básicas

INIT:	Allocate and initialize the symbol table.
ENTERBLOCK:	Prepare a new local naming scope.
LEAVEBLOCK:	Discard entries from the most recent scope entered, and reestablish the next outer scope.
IS_INBLOCK?:	Has a specified identifier already been declared in this scope? (Used to avoid duplicate declarations.)
ADD:	Add an identifier and its attributes to the symbol table.
RETRIEVE:	Return the attributes associated (in the most local scope in which it occurs) with a specified identifier.

← Problemas → Mala definición formal

Solución a través de significados semánticos

Type:	Symboltable
Operations:	
INIT:	→ Symboltable
ENTERBLOCK:	Symboltable → Symboltable
LEAVEBLOCK:	Symboltable → Symboltable
ADD:	Symboltable × Identifier × Attributelist → Symboltable
IS_INBLOCK?:	Symboltable × Identifier → Boolean
RETRIEVE:	Symboltable × Identifier → Attributelist
Axioms:	
(1)	LEAVEBLOCK(INIT) = error
(2)	LEAVEBLOCK(ENTERBLOCK(symtab)) = symtab
(3)	LEAVEBLOCK(ADD(symtab, id, attrs)) = LEAVEBLOCK(symtab)
(4)	IS_INBLOCK?(INIT, id) = false
(5)	IS_INBLOCK?(ENTERBLOCK(symtab), id) = false

(6)	IS_INBLOCK?(ADD(symtab, id, attrs), idl) = if IS_SAME?(id, idl) then true else IS_INBLOCK?(symtab, id)
(7)	RETRIEVE(INIT, id) = error
(8)	RETRIEVE(ENTERBLOCK(symtab), id) = RETRIEVE(symtab, id)
(9)	RETRIEVE(ADD(symtab, id, attrs), idl) = if IS_SAME?(id, idl) then attrs else RETRIEVE(symtab, idl)

## ② Mejorar type symboltable

Mejorar implementaciones de las operaciones de los tipos

La representación de un tipo T consiste en → 1. Implementación de las operaciones del tipo que es modelo para las axiomas de especificación T.

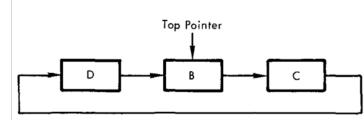
→ 2. Una función (?) que mapea los términos del modelo en sus representantes dominio abstracto

Si no tiene inversa

Así, el programar:

```
x := EMPTY.Q
x := ADD.Q(x, A)
x := ADD.Q(x, B)
x := ADD.Q(x, C)
x := REMOVE.Q(x)
x := ADD.Q(x, D)
```

Representación



La tabla de símbolos se realiza utilizando zando dos tipos abstractos auxiliares

Type: Stack
Operations:
NEWSTACK: → Stack
PUSH: Stack × Array → Stack
POP: Stack → Stack
TOP: Stack → Array
IS_NEWSTACK?: Stack → Boolean
REPLACE: Stack × Array → Stack
Axioms:
(10) IS_NEWSTACK?(NEWSTACK) = true
(11) IS_NEWSTACK?(PUSH(stk, arr)) = false
(12) POP(NEWSTACK) = error
(13) POP(PUSH(stk, arr)) = stk
(14) TOP(NEWSTACK) = error
(15) TOP(PUSH(stk, arr)) = arr
(16) REPLACE(stk, arr) = if IS_NEWSTACK?(stk) then error else PUSH(POP(stk), arr)

Type: Array
Operations:
EMPTY: → Array
ASSIGN: Array × Identifier × Attributelist → Array
READ: Array × Identifier → Attributelist
IS_UNDEFINED?: Array × Identifier → Boolean
Axioms:
(17) IS_UNDEFINED?(EMPTY, id) = true
(18) IS_UNDEFINED?(ASSIGN(arr, id, attrs), idl) = if IS_SAME?(id, idl) then true else IS_UNDEFINED?(arr, idl)
(19) READ(EMPTY, id) = error
(20) READ(ASSIGN(arr, id, attrs), idl) = if IS_SAME?(id, idl) then attrs else READ(arr, idl)

Así, el código para cada una de estas funciones sería:

```
INT' :: PUSH(NEWSTACK, EMPTY)
ENTERBLOCK'(stk, id) :: ENTERBLOCK(stk, EMPTY)
LEAVEBLOCK'(stk) :: if IS_NEWSTACK?(POP(stk))
then error
else POP(stk)
ADD'(stk, id, attrs) :: REPLACE(stk, ASSIGN(TOP(stk), id, attrs))
IS_INBLOCK?'(stk, id) :: if IS_NEWSTACK?(stk)
then false
else ~IS_UNDEFINED?(TOP(stk), id)
RETRIEVE'(stk, id) :: if IS_NEWSTACK?(stk)
then error
else ~IS_UNDEFINED?(TOP(stk), id)
then RETRIEVE(POP(stk), id)
else READ(TOP(stk), id)
```

The interpretation function  $\Phi$  is defined by:

```
(a)  $\Phi(\text{error}) = \text{error}$ 
(b)  $\Phi(\text{NEWSTACK}) = \text{error}$ 
(c)  $\Phi(\text{PUSH}(stk, EMPTY)) = \text{if IS_NEWSTACK?}(stk)$   
then INIT
else ENTERBLOCK( $\Phi(stk)$ )
(d)  $\Phi(\text{PUSH}(stk, ASSIGN(arr, id, attrs))) = \text{ADD}(\Phi(\text{PUSH}(stk,$   
 $arr)), id, attrs))$ 
```

No variación de inherencia  
Esto código carga con los caracteres ↗ No variación de representación

Beneficios de las especificaciones algebraicas

- El tamaño y la complejidad de una especificación es independiente del tamaño y complejidad del sistema
- Hay una separación clara entre el diseño y la implementación
- Pueden comprobar programas que usan datos abstractos, los especifican reglas de inferencia
- Fomenta la modularidad y diseño así como la reducción de errores por la reducción de dependencia

## LECTURA 2

### → Introducción

Los tipos de datos y las clases abstractas juegan importantes roles

Modularidad  
Confianza  
Verificación

¿Qué son los data types?

Como idea general son objetos y operaciones.

Un conjunto de objetos con una serie limitada de operaciones computables

(i) Aplicación  $\varphi: S \rightarrow N$   
(ii)  $\varphi(S)$  es un subconjunto recursivo de  $N$   
(iii)  $\exists$  algoritmo de  $\varphi^{-1}$   
(iv) los elementos de  $S$  son reconocibles

Datos abstractos

No tiene una definición exacta. Por un lado, se puede definir como un tipos de datos definidos por el usuario o para módulos que incorporan tipos de datos similares. Por otro lado, otros piensan que la característica más importante de un data type es que la información sobre la implementación no está asociada con el tipo de dato.

## TIPOS ABSTRACTOS DE DATOS

Def. que agrupa en un conjunto que permite expresar información como fracciones. Dependen del contexto  
Permite extender el tipo de datos

Concepto abstracción: comprender de fenómenos que incluyen gran cantidad de detalles

Comprenden dos conceptos complementarios  Destacar:  Ignorar:

### Abstracción en el diseño de programas

Dependiendo del lenguaje y paradigma la utilización varía

El tipo de dato indica la actuación (ej. Pedir)

Way (forma) de programación al modificar el tipo de dato

Ejemplo: En python las fechas se crean con def. y en java con objetos (los objetos se definen por clases)

★ Programación orientada a objetos: Paradigma más elevado (muy alto nivel de abstracción)

Proceso de abstracción entre lo que puede hacer un objeto y cómo lo hace

Encapsulador: mnemotécnicas y macros

Lenguaje de alto nivel: instrucciones y programas

Programación estructurada: Abstracción de control y funcional o procedimental

Programación modular: Diferencia en parte pública y privada

Programación funcional y lógica: Abstracción de la secuencia de instrucciones que sigue la máquina

Programación orientada a objetos: Abstracción entre lo que puede hacer un objeto y cómo lo hace

## ABSTRACCIÓN EN LOS DATOS

- Tipos de datos en los lenguajes de alto nivel: los datos dejan de tratarse como secuencia de bits manipulados por el programador y pasan a ser independientes de la máquina.
- Con los tipos definidos por el programador se logra un mayor nivel de abstracción. Se definen los valores concretos del tipo y se utilizan mediante operaciones ya predefinidas para su manipulación.
- Tipos de datos abstractados: el lenguaje proporciona constructores genéricos de tipos (el programador debe "concretar" el parámetro formal y operaciones predefinidas para manipular los valores del tipo).
- Tipos de datos abstractos: permiten el uso de los datos y operaciones (encapsulación) sin conocer su representación ni la implementación de los operaciones (extensión).
- Clase de objetos: Soportan adecuado herencia y polimorfismo.

## TIPOS ABSTRACTOS DE DATOS

- Fue introducido por John Guttag (1974), se define como: "colección de valores y operaciones que se definen mediante una especificación que es independiente de cualquier representación". También: "Es un modelo matemático en una operaciones definidas sobre él".
- El programador debe establecer la interfaz del Tipo Abstracto de Datos, es decir, especificar los valores y las operaciones que van a tener parte del mismo. La interfaz debe ser pública.
- Posteriormente, el programador deberá elegir y realizar la implementación o representación de los datos y operaciones (definidas por separado) del tipo especificado.
- La implementación debe hacerse con alcance de declaraciones que no sea accesible desde fuera (alcance privado). Así, cualquier modificación o actualización que se realice no afectará a los programas que la utilizan.
- La implementación de los operaciones se "encapsula" junto con la representación del tipo de datos. De esta forma se facilita la localización para posteriores modificaciones o actualizaciones.

## → CARACTERÍSTICAS

- Compatibilidad con los tipos de datos de los lenguajes de programación: privacidad de la representación (en términos de los datos y las operaciones), y protección: los valores del tipo solo se tratan con las operaciones previstas en la especificación.

- Compartidos con las subprogramas de la programación estructurada: generalización de los tipos de datos (los procedimientos) complementan las operaciones) y encapsulación de la información (la representación de los datos se declara junto con la implementación de las operaciones).

Ventajas de esa interfaz:

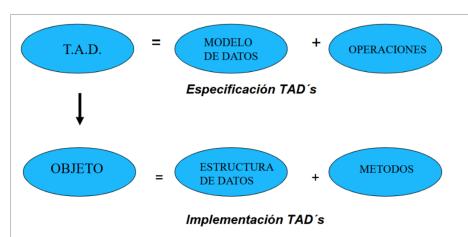
- Independiente de la representación interna de fecha.
- Privacidad: Desconoce los detalles de la representación.
- Protección: solo se pueden usar los op. previstos por la especificación.

```

fun crear (día, mes, año: natural) dev (f: fecha)
fun incrementar (fini: fecha; num_días: entero) dev (ffin: fecha)
fun reducir (fini: fecha; num_días: entero) dev (ffin: fecha)
fun distancia (fini, ffin: fecha) dev (numdías: entero)
fun dia_semana (f: fecha) dev (dia: 1..7)

```

- Destacan la importancia tanto de los datos como de sus operaciones.
- Los valores de un TAD se generan a partir de las operaciones definidas para este.
- Es necesario crear dos documentos diferentes:
  - a) Especificación del TAD: Gesta de sintaxis (nombre operaciones) y semántica (descripción de lo que hacen estos operaciones).
  - b) La implementación de los datos del TAD's y de las operaciones.
- Un TAD representa una abstracción:
  - a) Se destacan los detalles del comportamiento observable (especificación)
  - b) Se ocultan los detalles de implementación



- Programación con TAD's
  1. Descripción en lenguaje informal
  2. Especificación del TAD:
    - Formal (lenguaje algebraico)
    - Pseudocódigo
  3. Implementación:
    - Definir los tipos de datos necesarios para representar los datos en memoria de una manera concreta.
    - Especificar en lenguaje algorítmico los op. sobre los datos (pseudocódigo)

#### - IMPORTANCIA

- La especificación formal proporciona la interfaciabilidad necesaria para la posterior implementación.
- Permite razones sobre la corrección de dicha implementación.
- Permite una interpretación uniforme de las diferentes versiones.
- Es posible deducir de forma automática las propiedades que debe satisfacer cualquier representación.
- Se analiza la eficiencia de las diferentes posibles implementaciones.

## → ESPECIFICACIÓN FORMAL

consta de sintaxis (descripción de operaciones, tipos de parámetros y resultados) y semántica (descripción de "lo que hacen" o significado de cada operación).

## → ESPECIFICACIÓN ALGEBRAICA

Es una técnica formal que tiene el objetivo de definir, de forma no ambigua e independiente de cualquier implementación, un tipo de datos: valores del mismo y efecto de cada op.

Al realizar la definición de estos especificaciones se sigue una técnica modular, desde abajo hacia arriba, comenzando por los tipos más básicos para continuar con los más complejos.

### SINTAXIS DE UNA ESPECIFICACIÓN ALGEBRAICA:

espec NOMBRE\_ESPECIFICACIÓN

{Nombre o identificador de la especificación, en mayúsculas}

usa TIPO1, TIPO2,...

{Nombre de otros TAD's ya especificados, en mayúsculas}

parámetro formal

{Especificación algebraica del parámetro formal, de existir}

parámetro

géneros nombre\_tipo

{Nombre de los valores del TAD que se define, en minúsculas}

### SINTAXIS DE UNA ESPECIFICACIÓN ALGEBRAICA:

#### operaciones

operación: parámetro1 parámetro2 ... → resultado

{Nombre de cada operación en minúsculas, seguido de los tipos de los parámetros y del resultado}

{Las operaciones se declaran prefijas, pero puede indicarse la ubicación de los parámetros usando guiones bajos "\_".}

- Las operaciones son funciones con cualquier cantidad de parámetros y un único resultado.

#### ecuaciones

$t_i = t_j$

{ $t_i$  y  $t_j$  son valores del TAD obtenidos usando operaciones}

- Realmente cada ecuación es una fórmula lógica.

#### fespec

### Semántica:

1. Cada término sintácticamente correcto es un valor del tipo
2. Solo pertenecen al tipo de datos especificando los valores generados por términos sintact. correctos.
3. Cada ecuación expresa la igualdad entre dos términos sintact. diferentes. El orden es irrelevante.

### Aspectos a tener en cuenta

- a) En la definición de las operaciones y ecuaciones debe evitarse destruir las especificaciones realizadas para los tipos de datos
- b) Es difícil determinar el número de ecuaciones necesarias para la especificación correcta del tipo.

### Operaciones de la especificación algebraica

- a) Operaciones constructivas: Su resultado es del tipo especificado. Se pueden dividir en: operaciones generadoras, que son el mínimo conjunto de las operaciones a partir de las cuales se pueden obtener todos los valores del tipo de datos, y operaciones modificadoras, que son el resto de las operaciones constructivas que no forman parte del conjunto de operaciones generadoras.

b) Operaciones generadoras: al menos uno de los parámetros de la operación es del tipo de interés y el resultado es de otro tipo.

Conjunto libre: cuando cada término obtenido a partir de las operaciones generadoras es un valor diferente al tipo especificado se dice que el conjunto de operaciones generadoras es un conjunto libre. En ese caso, para cada valor del tipo de dato existe un único término canónico.

### → Ecuaciones de la especificación algebraica.

- a) Ecuaciones entre generadoras: Son necesarias si es posible obtener, a partir de las operaciones generadoras, términos sintácticamente distintos que representan el mismo valor del tipo,
- b) Para cada operación modificadora se escriben las ecuaciones necesarias para garantizar que cada nuevo término obtenido puede expresarse usando las operaciones generadoras.
- c) Para cada operación clasificadora se escriben las ecuaciones necesarias para garantizar que los términos del tipo de datos de su resultado coincide con alguno de los posibles valores del TAD que se especifica.

#### EJEMPLO: BOOLEANOS

```

espec BOOLEANOS_2
  géneros bool
  operaciones
    T: → bool
    F: → bool
    ¬: bool → bool
    _ ∧ _: bool bool → bool
    _ ∨ _: bool bool → bool
  var x: bool
  ecuaciones  {usando como generadoras T y ¬}
    ¬(¬(x)) = x      F = ¬(T)
  {and}   T ∧ x = x      F ∧ x = F
  {or}    T ∨ x = T      F ∨ x = x
fespec

```

#### EJEMPLO: BOOLEANOS

```

espec BOOLEANOS
  géneros bool
  operaciones
    T: → bool
    F: → bool
    ¬: bool → bool
    _ ∧ _: bool bool → bool
    _ ∨ _: bool bool → bool
  var x: bool
  ecuaciones  {usando como generadoras T y F}
    {not}      ¬(T) = F          ¬(F) = T
    {and}      T ∧ x = x        F ∧ x = F
    {or}       T ∨ x = T        F ∨ x = x
fespec

```

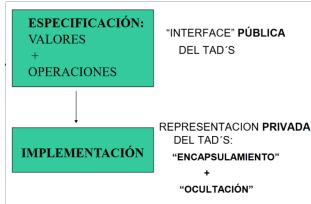
El formalismo debe valer para todo  
se establece unívoco (repredefinible en el resto de paradigmas)

### Alfred Aho

Desde el punto de vista de programación,  
funcionan desde contratos: Los programadores donde el contrato indica la funcionalidad del  
código. El tipo de arbitraje busca abstraer la complejidad, no es necesario conocer cómo  
lo ha hecho (la respuesta es el contrato). El contrato define una intención, a la que se  
adhirieron los programadores.

En Java, tanto datos como funciones pertenecen a la misma clase.  
Los datos tienen un ámbito público, privado, o doble, protegido ...  
Para realizar inserción se ejecutan (privada)  
Para realizar función insert (pública)

Especificaciones  
nos contrarrestan en la implementación



Ejemplo números naturales

espec. Naturales

gentes natural

operaciones

$O: \rightarrow \text{natural}$

$\text{suc} : \text{natural} \rightarrow \text{natural}$

$- + - : \text{natural natural} \rightarrow \text{natural}$

var  $x, y : \text{natural}$

expresión

$x + O = x$

$x + \text{suc}(y) = \text{suc}(x+y)$

leopac

## NOTACION ASINDOTICA

- Función del tamaño de las entradas de datos ( $n$ ) utilizada para hacer referencia al tiempo máximo de ejecución  $T(n)$  de un programa para una entrada de tamaño  $n$
- Suponemos que es posible evaluar programas comparando sus funciones de tiempo de ejecución

→ Son soluciones aproximadas (dependen del observador)

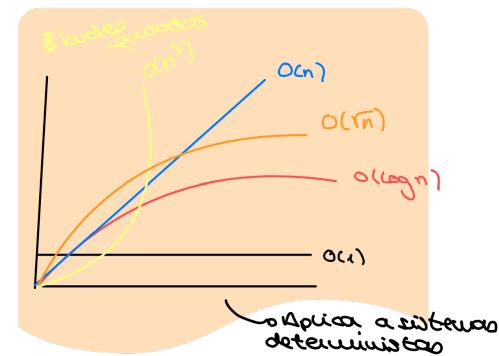
Algoritmos de IA: Buscan soluciones suficientemente buenas condistintos agentes

Técnicas de optimización: Aportan solución óptima

Computación cuántica: Utilizan la física cuántica para operar. Es no determinista

Reglas para el cálculo de tiempo de ejecución

1. Es una sentencia o instrucción simple →  $O(1)$
2. El tiempo de ejecución de una composición de sentencias o instrucciones es la suma de los tiempos de dichas sentencias.
3. De un condicional es el tiempo de ejecución mínimo entre ambas ramas o cada una de las sentencias de las cuales sea más el tiempo de evaluación de la condición



4. El bucle bucle es la suma sobre todos los iteraciones del tiempo de ejecución del bucle más el tiempo de evaluar la condición de fin de bucle

## → Implementación

- No es único: debe elegirse la implementación más adecuada teniendo en cuenta que operaciones son las más frecuentes. (Muy que buscar la eficiencia de los algoritmos usados)

### • Simplificar:

- Representar los datos en memoria de una manera concreta (definir los tipos de datos necesarios)
- Especificar un código algebraico las operaciones sobre los datos (pseudocódigo)

## → Realizar el cálculo de notación asintótica.

- $O(n^k)$  donde  $k$  es el número de bucles anidados en la parte ramificada del programa
- $O(kgn)$  si el programa divide el tamaño de la entrada por un número

```
fun OrdIns(v:vector) dev vector
    i ← 2
    mientras (i < n) hacer
        x ← v[i]
        j ← i - 1
        mientras ((j > 0) y (v[j] > x)) hacer
            v[j+1] ← v[j]
            j ← j - 1
        fmientras
        v[j+1] ← x
        i ← i + 1
    fmientras
    Devolver v
Efun
```

$O(n)$

$O(n^2)$

	VENTAJAS	DESVENTAJAS
Vector de Booleanos	<ul style="list-style-type: none"> <li>- Las operaciones básicas (insertar, eliminar, ... ) se realizan en tiempo constante.</li> <li>- Las operaciones es-vacio? y conjunto = vacío se realizan en tiempo proporcional al tamaño del dominio</li> </ul>	<ul style="list-style-type: none"> <li>- El espacio de almacenamiento es constante proporcional al tamaño del dominio</li> <li>- Debido a las restricciones, solo es posible utilizarla cuando el dominio de los elementos del conjunto son subconjunto finito de los naturales o es posible establecer correspondencia.</li> </ul>
Lista enlazada	<ul style="list-style-type: none"> <li>- Muy general, no impone restricciones respecto al dominio.</li> <li>- El espacio es proporcional al cardinal del conjunto representado, no del dominio</li> </ul>	<ul style="list-style-type: none"> <li>- Debido a que se realizan búsquedas lineales en la estructura, el tiempo utilizado para las operaciones básicas es proporcional al cardinal del conjunto que se almacena.</li> </ul>

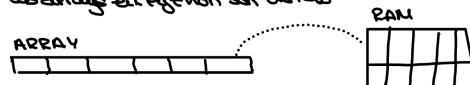
## LISTAS

Def:

Las listas tienen orden, pero no están ordenadas

Tipos de listas → Básicas: Estructuras de datos hiperseñaladas que se pueden combinar con otras más complejas.

losarreng en Python en visto

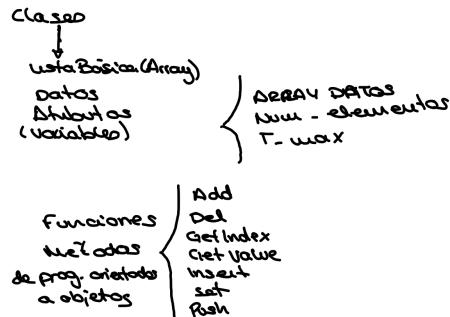


Tiene un tamaño máxino, elegido por el programador en el momento de creación

Métodos de utilidad: se acercan a la interfaz para que no quite mucho espacio. la migración

Puedo mantenerlos en orden de inserción de elementos pero no puedo ordenar los elementos.

Definición de interfaz de clase: las clases son una lista basada en donde se necesita un conjunto de datos para poder trabajar con ella.



→ Encazadas: son de tipo "Dihar". Tienen una gran flexibilidad

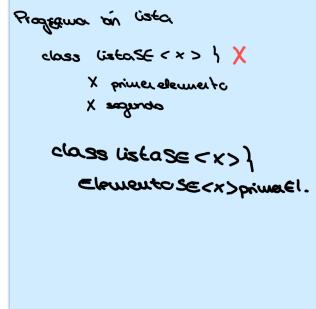
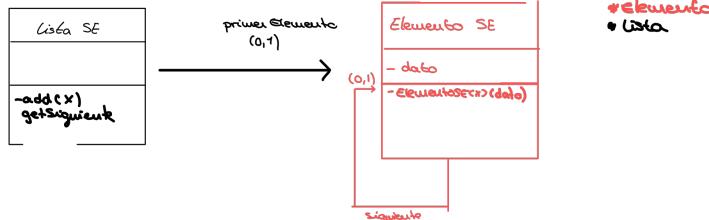
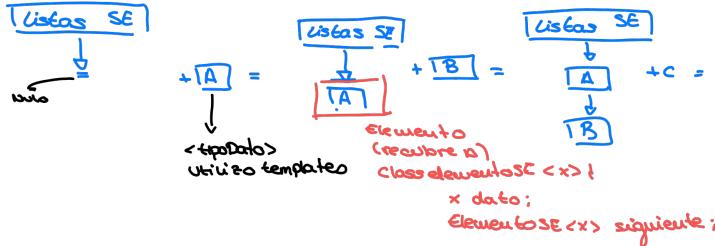
→ Dúbitas e elaborações:

Interfaz: Indice concreto a actores bájicos de datos

Deberías poder: | Add Add  
Básico Delete Delete  
Get GetIndex = Y  
Get Value

Eucaristia tempos where pass. donde grande elementos

→ Listas dinámicas



ListaSE< Integer > milista = new ListaSE< Integer >();

```
void miPrueba() {
    Integer a = new Integer(5);
    milista.add(a);
```

```
// Creo un nuevo elem ← ElementoSE<x> temporal = new ElementoSE<x>(dato)
// El siguiente del elemento ← temporal.setSiguienento(this.primerElemento);
// se colo a la lista
// La lista ahora apunta al temporal
// setPrimerElemento(temporal);
// inElementos();
```

```
class ElementoSE<x> {
    x dato
    ElementoSE<x> siguiente;
    ElementoSE<x> (x undato) {
        class dato = undato;
    }
    private ElementoSE();
}
```

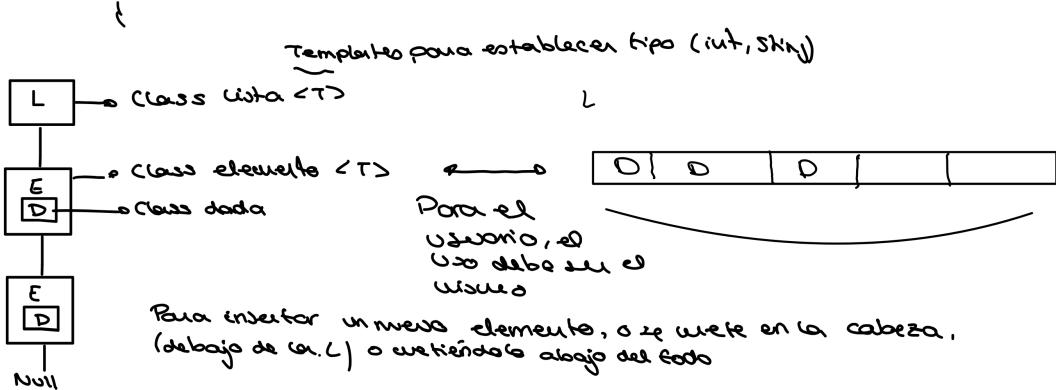
#### \* CASO ESPECIAL

```
int add (x dato) {
    ElementoSE<x> temporal = new ElementoSE<x>(dato);
```

Leyendo  
if (getPrimerElemento() == null) {  
Fergo el otro  
setPrimerElemento(temporal);  
en la lista}

Recorro  
la lista  
else:  
ElementoSE<x> fijoante = getPrimerElemento();  
while (fijoante.getsiguiente() != null) {  
fijoante = fijoante.getsiguiente();

flotante. siguiente (temporal);



¿Cómo recuperarlos los elementos de la lista?

- Permitirnos al usuario que recorra la lista con libertad, pero no tenemos la capacidad de darle un elemento que nos pida
- Esto nos presenta un problema: ¿cómo recorrer la lista?  
Nuestro conjunto de datos, 2 recorridos distintos: Recorre una lista que se llama iterador, que guarda el estado de un recorrido de una lista.
- Las listas no guardan el estado del recorrido
- Se pueden tener varios iteradores
- El iterador, que es una clase, contiene un constructor, que permite saber si hay siguiente elemento retorna dicho dato
- Los iteradores tienen una interfaz que define los métodos: Hay siguiente, get dato
- En la clase lista se llama al iterador.

Lista < T > lista;

Elemento < T > actual;

Son los dos atributos (datos) de la clase iteradora que va a quererse saber dónde está tomando en la lista.  
Estos datos son los que llevamos estado

```
class Iterador {  
    Iterador (Lista < T > l) {  
        hay_siguiente();  
    }  
    boolean hay_siguiente() {  
        return true;  
    }  
    T get_dato() {  
        return dato;  
    }  
}
```

Constructor iterador:

```
Iterador (Lista < T > l) {  
    this.lista = l;  
    this.actual = l.primer_elemento;  
}  
boolean hay_siguiente() {  
    return this.actual != null;  
}
```

```
T get Date() {  
    T temporal = this.actual.getDate();  
    this.actual = this.actual.getSignature();  
    return temporal;  
}
```

Debería haber un `!t` (Mayúscula) = true para evaluar si se pide  
nueve un `get Data()` del siguiente  
elemento

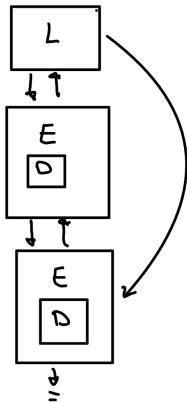
```

Static void main() {
    lista<string> l = new lista<string>();
    l.add("Hola")
    l.add("Mundo")
    Iterator<string> i = l.iterator;
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}

```

```
class List<T> {
    Iterator<T> iterator() {
        return new Iterator<T>(list);
    }
}
```

→ LISTAS PARCIALMENTE ENCAZADAS



class Lista DE{  
 ElementoDE primero;  
 ElementoDE ultimo;  
 {  
 dos Elementos;  
 Tdato;  
 ElementoDE anterior;  
 ElementoDE siguiente;  
 }  
}

Nos permite recorrer la  
línea de artiba abajo  
o viceversa, para lo que  
habrá que establecer  
un Mayón Luer( ) / getAnterior()

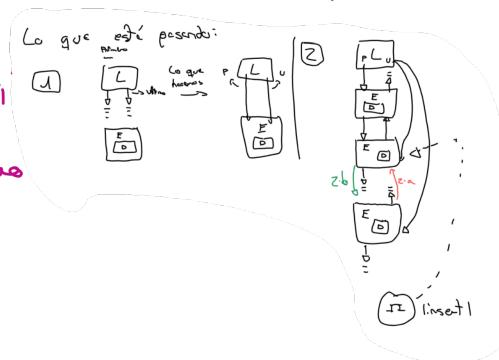
## → OPERACIONES BÁSICAS

1000 void add (T0 data){  
Element0DE< T0> element = new Element0DE (data);

Wta varia: ⑥ if (this->primero == null) {  
    this->primero = elemento;  
    this->ultimo = elemento;}

٢٦

elements. anterior - tris. ultimo  
tris. ultimo. signe = elements  
ultimo = elements



```
2. INSERT void insert(TD data){  
    this.lta.insert(this.actual,data);  
}
```

void insert\_element(DE elements, TD data){

```
utavaara ← ④ if (elements == null) {  
    this.add(data);  
}
```

محله (ع)

ElementsDE < TD > new elements = new elementsDE (date)

```
if (elemento == this.primero) {
```

elemento.anterior = nuevoElemento;  
nuevoElemento.siguiente = elemento;  
elListado.primerElemento = nuevoElemento;

Si el elemento  
está entre z

else h

`novoElemento.siguiente = elementos`

nuevos Elementos.anterior = elementos.anterior

novo Elemento anterior - sujeito = novo Elemento  
novo Elemento - sujeito - anterior = novo Elemento

### 3. DELETE

```
void delete (ElectroDE electro) {
```

Sistano ←  
Jata (si es  
vacía no quita  
nada)

i (electro ! = null) {

boolean primero = false, ultimo = null;

```

if (elemento.anterior == null) {
    elemento.primer = elemento.siguiente;
    primero = true;
}

```

```
if elements.size() == null {
```

this ultimo = eleventh  
ultimo = true

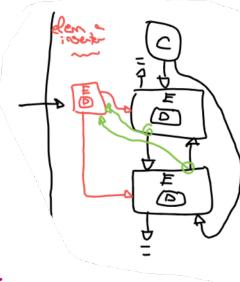
not prwero

```
if (!priero)
```

elements.on this signe = elements.signe;

it (!.ultime)

elements. signifi. anterior = elements. anterior



## Tabla de verdad de (esta con elementos)

<u>Primero</u>	<u>Último</u>
F	F
F	T
T	F
T	T

## → PILAS Y COLAS

Son estructuras de datos menos generalizada

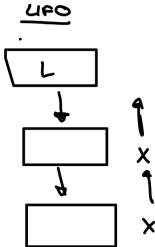
→ **PILAS:** Apila elementos de arriba a abajo  
y saca de abajo a arriba



→ **COLAS:** Apila o de principio y quita desde principio



→ EN LISTAS:



FIFO

Para insertar: enqueve (en cola)  
L.insert (lsta . primero)

Para recuperar: desqueve (descola)

Para insertar: Usamos L.push (push)

Para recuperar: (pop)