

# PROGRAMACIÓN II

Profesor de teoría : EDUARDO SERRANO B-305  
eduardo.serrano@uam.es

SEMINARIO - 12:00 - 14:00  
DE : MARTES - 18:00 - 20:00  
C  
MIÉRCOLES - 9:00 - 11:00

# TIPOS ABSTRACTOS DE DATOS (TAD)

→ ABSTRACCIÓN  $\begin{cases} \text{de datos} \\ \text{funcional} \end{cases}$

→ TAD / EDD (Implementación)

- ① Definir el conjunto de Datos
- ② Definir la interfaz

Tipos Primitivos de Datos en C: int, char, double, float

En C no hay tipos abstractos de Datos, sino que son tipos primitivos.

Los tipos abstractos de datos amplían el conjunto primitivo de datos.

Ejemplo:

TAD COMPLEJO  $\sim \mathbb{Z}$   $\begin{matrix} \text{p. real} & \text{p. imaginaria} \end{matrix}$

COMPLEJO crear\_complejo(float a, float b); → creación complejo (TAD)

complejo z;

z = crear\_complejo(3, 4);

void liberar\_complejo(complejo z); → liberamos los recursos utilizados por el tipo abstr. de datos (en este caso, COMPLEJO)

# TAD (DEFINICIÓN DEL IAD) .h

1. DEFINIR EL CONJUNTO DE DATOS

2. DEFINIR LA INTERFAZ DEL TAD ( $\equiv$  conjunto de funciones que permiten interactuar con el TAD).


(PSC de las primitivas/funciones)

- Que hacen
- Parámetros de entrada (y si modifican alguno)
- Que devuelven

ej: COMPLEJO    complejo-crear(float re, float im).  
void    complejo-liberar(COMPLEJO z)  
float    complejo-getReal(COMPLEJO z)  
float    complejo-getImaginaria(COMPLEJO z)  
status    complejo-setRe(COMPLEJO z, float re)  
status    complejo-setIm(COMPLEJO z, float im)

## ESTRUCTURA DE DATOS del TAD (IMPLEMENTACIÓN DEL TAD) .c

1. ESPECIFICAR la EDD que utiliza PARA ALMACENAR los datos.

array    
array[0] = parteReal  
array[1] = parteImg

   
parteReal    parteImg

2. IMPLEMENTAR los ALGORITMOS para las FUNCIONES de la INTERFAZ.

```
complejo.h  
(#include "status.h")  
typedef struct _COMPLEJO COMPLEJO;
```

```
COMPLEJO * complejo-crear(float re, float im);
```

```
void complejo-liberar(COMPLEJO * z);
```

```
COMPLEJO * complejo-actualizar(COMPLEJO * z, float re, float im);  
(status)
```

complejo.c

```
#include "complejo.h"
```

```
struct _COMPLEJO {
```

```
    float re;
```

```
    float im;
```

```
}
```

```
COMPLEJO * crear-complejo(float re, float im) {
```

```
    COMPLEJO * z;
```

```
    * z = (COMPLEJO *) malloc(sizeof(COMPLEJO));
```

```
    z->re = re;
```

```
    z->im = im;
```

```
    return z;
```

```
}
```

```
* if (z == NULL) {  
    return NULL;  
}
```

## PROBLEMA 11

haz un programa en C que cree dos n<sup>os</sup> complejos con los valores que tú quieras y que imprima por pantalla la suma de ellos.

CUIDADO! utilizamos la metodología que hemos visto, abstracción de datos y abstracción funcional.  
→ conocemos complejo.h

main. C

```
#include "complejo.h";
```

```
COMPLEJO *suma(const COMPLEJO *z1, const COMPLEJO *z2);
```

```
main() {
```

```
    COMPLEJO *z1, *z2, *zsuma;
```

```
    z1 = complejo-crear(1, 2);
```

```
    z2 = complejo-crear(10, 20);
```

```
    zsuma = suma(z1, z2);
```

```
    printf("%f %f", complejo-getRe(zsuma), complejo-getIm(zsuma));
```

```
    complejo-liberar(z1);
```

```
    complejo-liberar(z2);
```

```
    complejo-liberar(zsuma);
```

```
}
```

```
COMPLEJO *suma(const COMPLEJO *z1, const COMPLEJO *z2) {
```

```
    float ReSuma, ImSuma;
```

```
    COMPLEJO *suma;
```

```
    ReSuma = complejo-getReal(z1) + complejo-getReal(z2);
```

```
    ImSuma = complejo-getIm(z1) + complejo-getIm(z2);
```

```
    suma = complejo-crear(ReSuma, ImSuma);
```

```
    return suma;
```

```
}
```

TAD: conjuntos

¿Qué podemos hacer con el TAD?

da- no, a imitiva {  
- crear conjunto  
- liberar conjunto  
- insertar elemento <sup>en</sup> ~~con~~ conjunto

Definición del TAD

.h

→ contenedor de datos

→ PSC (Implementación) de las primitivas de la Interfaz

Implementación

.c

CONJUNTO.h

"pseudocódigo"

CONJUNTO conjunto-crear( )

void conjunto-liberar( conjunto cj)

status/CONJUNTO conjunto-insertar( Elemento ele, conjunto cj)

Boolean conjunto-pertenecer( Elemento ele, conjunto cj)

Elemento conjunto-extraer( Elemento ele, conjunto cj)

{ status conjunto-intersección( conjunto A, conjunto B, conjunto resul)

[ CONJUNTO conjunto-intersección( conjunto A, conjunto B)

CONJUNTO conjunto-union( conjunto A, conjunto B)

;

CONJUNTO.h / En C

```
#include "Elemento.h"
```

```
#include "tipos.h"
```

```
typedef struct - conjunto conjunto;
```

```
conjunto * conjunto_crear();
```

```
void conjunto-liberar(conjunto *cj);
```

```
conjunto * conjunto-insertar(const Elemento *ele, conjunto *cj);
```

```
Boolean conjunto-pertenece(const Elemento *ele, const conjunto *cj);
```

```
:
```

CONJUNTO.C

→ DETERMINAMOS CONTENEDOR DE DATOS: ARRAY

```
#include "Conjunto.h"
```

```
#include "tipos.h"
```

```
#define Ele-max 100
```

```
struct -conjunto {  
    Elemento *a[Ele-max];  
    int cardinal;
```

```
};
```

```
conjunto * conjunto_crear() {
```

```
    conjunto *cj;
```

```
    cj = (conjunto *) malloc(sizeof(conjunto));
```

```
    if (cj == NULL) {
```

```
        return NULL;
```

```
    }
```

```
    cj->cardinal = 0;
```

```
    return cj;
```

```
}
```

```
r = conjunto_crear(); (*)
```

```
int i;
```

```
for(i=0; i < a->cardinal; i++) {
```

```
    if (conjunto_insertar(r, a[i]) == ERROR) {
```

```
        conjunto_liberar(r);
```

```
        return ERROR;
```

```
    }
```

```
}
```

```
for(i=0; i < b->cardinal; i++) {
```

```
    if (conjunto_pertenece(a, b[i]) == FALSE) {
```

```
        if (conjunto_insertar(r, b[i]) == ERROR) {
```

```
            conjunto_liberar(r);
```

```
            return ERROR;
```

```
        }
```

```
    }
```

```
}
```

```
return OK;
```

```
}
```

```
void conjunto_liberar(conjunto *c) {
```

```
    int i;
```

```
    for(i=0; i < c->cardinal; i++) {
```

```
        elemento_liberar(c->a[i]);
```

```
    }
```

```
    free(c);
```

```
}
```

```
if(r == NULL) {  
    return NULL;  
}
```

```
status conjunto_insertar(conjunto *cj, elemento *ele)  
{  
    if(cj->cardinal == ELEM_MAX || cj == NULL ||  
       ele == NULL || conjunto_pertenece(cj, ele) ==  
       TRUE) {  
        return ERROR;  
    }
```

```
    Elemento *copia;
```

```
    copia = elemento_copiar(ele);
```

```
    cj->a[cj->cardinal] = copia;
```

```
    cj->cardinal++;
```

```
}
```



# TAD: PILA (stack)

ila pila-crear()  
id pila-liberar(PILA p) → "pila-insertar"  
tus push(PILA p, Elemento ele) → "pila-extraer"  
ento pop(PILA p)  
ento pila-tope(PILA p)  
ean pila-llena(PILA p)  
ean pila-vacia(PILA p)

## pseudocódigo

PILA.h → #include "Elemento.h"  
#include "types.h"

typedef struct \_pila Pila;

PILA \* pila-crear();

void pila-liberar(PILA \*p);

status push(PILA \*p, const Elemento \*ele);

Elemento \* pop(PILA \*p);

Elemento \* pila-tope(const PILA \*p);

Boolean pila-llena(const PILA \*p);

Boolean pila-vacia(const PILA \*p);

```

struct pila {
    Elemento *dat [MAX-ELE];
    int tope;
};

```

```

PILA * pila-crear() {
    Pila *p;
    p = (*Pila) malloc(sizeof(Pila));
    if (p == NULL) {
        return NULL;
    }

```

```

(*) for (i=0; i < MAX-ELE; i++) {
    p->dat[i] = NULL;
} // no es necesario, pero recomend

```

```

    (*)
    p->tope = -1; // también podía ser p->tope = 0;
    return p;
}

```

```

Boolean pila-vacia (const *PILA *p) {

```

```

    if (p == NULL)
        return TRUE;

```

```

    if (p->tope == -1)
        return TRUE;

```

```

    return FALSE;
}

```

```

void pila-liberar (PILA *p) {

```

```

    int i;
    for (i=0; i <= p->tope; i++) {
        elemento-liberar(p->dat[i]);
    }

```

```

    free(p);
}

```

```

(*) if (p != NULL) {
    [ ]
}

```

```
status push (PILA *p, const Elemento *ele) {  
    Elemento *ele2;  
    if (p == NULL || ele == NULL || pila_llena(p) == TRUE)  
        return ERROR;
```

```
    p->tope ++;
```

```
    ele2 = elemto_copiar(ele);
```

```
    if (ele2 == NULL)  
        return ERROR;
```

```
    p->dat[p->tope] = ele2;
```

```
    return OK;
```

```
}  
  
Elemento * pop (PILA *p) {  
    Elemento *ele = NULL;  
    if (p == NULL || pila_vacia(p) == TRUE) {  
        return NULL;
```

```
    }
```

```
    ele = p->dat[p->tope];
```

```
    p->dat[p->tope] = NULL;
```

```
    p->tope --;
```

```
    return ele;
```

```
}
```

```
Boolean pila_llena(const PILA *p) {
```

```
    if (p == NULL) {  
        return FALSE;
```

```
    }
```

```
    if (p->tope == (MAX-ELE-1)) {  
        return TRUE;
```

```
    }
```

```
    return FALSE;
```

```
}
```

$$75 * + 29 + 8 * -$$

$$\boxed{3 + 7 * 5} - \boxed{(2 + 9) * 8};$$

$$\boxed{3 \quad 75 * +} \quad \boxed{29 + 8 * -}$$

## APLICACIONES TAD PILA

- Balanceo parentesis (símbolos apertura/cierre)  
(parseo de expresiones)
- Expresiones aritméticas
  - { Infijo
  - { Sufijo
  - { Prefijo

### PSEUDOCÓDIGO BALANCEO DE PARENTESIS

```
Boolean evaluarExpresion (cc s) | - char leerCaracter (cc s)
                                | - Boolean esSimboloApertura (char s)
                                | - Boolean esSimboloCierre (char s)
```

```
Boolean evaluarExpresion (cc s) {
    pila_ini(p);
    if (pila_ini(p) == ERROR)
        return ERROR;
```

```
    while (c = leerCaracter(s) != '\0') {
```

```
        if (esSimboloApertura(c) == TRUE)
```

```
            push(p, c);
```

```
        else if (esSimboloCierre(c) == TRUE)
```

```
            q = pop(p);
```

```
            if (q == NULL)
                return ERROR;
```

```
    }
```

```
    if (pila_vacia(p) == TRUE) {
```

```
        pila_destroy(p);
        return TRUE;
```

```
    }
```

```
    pila_destroy(p);
    return FALSE;
```

```
}
```

```
(*) if (push(c,p) == ERROR)
    pila_destroy(p);
    return ERROR;
```

leemos caracter a caracter, si es un operando lo meto en la pila (push) si es un operando sacas 2 ~~de~~ operandos de la pila (2 pop) y los operas. El resultado lo metes (push) en la pila

5  
7 7 35 2 88  
- 3 3 3 3 38 38 38 -50 -  
Pseudocódigo de sufijo a resultado

int evalua Expr Sufijo (string s)

p = pila-ini() ①

while (c = leerCaracter(s) ≠ EoS)

if (esOperando(c) == TRUE)

push(c, p)

②

else

op2 = pop(p) ③

op1 = pop(p) ④

r = evalua (op1, op2, c)

push(r, p)

r = pop(p) ⑤

if (pila-vacia(p) == TRUE)

return r

return NaN

① if (p == NULL)  
return NaN

② if (push(c, p) == ERROR)  
pila-destroy(p)  
return NaN

③ if (op2 == NULL)  
pila-destroy(p)  
return NaN

④ = ③

⑤ if (r == NULL)  
pila-destroy  
return NaN

pasar de infijo a sufijo

$$a * b + c \longrightarrow ab * c +$$

<u>Lee</u>	<u>Exp Salida</u>	<u>Pila</u>
a	a	-
*	a	*
b	ab	*
+	ab *	+
c	ab * c	+
;	ab * c +	-

$$a * b + c - d + f \wedge e \longrightarrow ab * c + d - f e \wedge +$$

<u>Lee</u>	<u>Exp Salida</u>	<u>Pila</u>
a	a	-
*	a	*
b	ab	*
+	ab *	+
c	ab * c	+
-	ab * c +	-
d	ab * c + d	-
+	ab * c + d -	+
f	ab * c + d - f	+
^	ab * c + d - f	+ ^
e	ab * c + d - f e ^ +	-

```

string traducir(string s)
char leerCaracter(string s)
printf(string s', char c)
int prec(char op1, char op2)
Boolean esOperador(char c)

```

```

string traducir(s) {
    pila = ini(p)
    while (c = leerCaracter(s) != Eos) {
        if (esOperador(c) == FALSE)
            printf(s', c)
        else {
            while (preferencia(pila-top(p), c) >= 0)
            {
                c' = pop(p)
                string = concatenar(s', c')
                push(p, c')
            }
        }
    }
}

```



## Balanceo de paréntesis simple (pseudocódigo)

```
Bool balanceoDeParentesisSimple (CCTAD string) {  
    if (string == NULL) return FALSE;  
    Stack p;  
    p = pila-crear();  
    if (p == NULL) return FALSE;  
    while (it = leerSimbolo(string) != EoS) {  
        if (esSimboloApertura(it)) {  
            if (pila-push(p, it) == NULL) {  
                return FALSE;  
            }  
        }  
        else if (esSimboloCierre(it)) {  
            if (ele = pila-pop(p) == NULL) {  
                return FALSE; → pila-destroy(p);  
            }  
            element-destroy(ele);  
        }  
    }  
    if (pila-vacia(p) == FALSE) {  
        pila-destroy(p);  
        return FALSE;  
    }  
    pila-destroy(p);  
    return TRUE;  
}
```

~~Balance~~

Bool balanceOfParenthesisMultiple (cc string) {

if (string == null) return FALSE;

Stack p;

p = pila-crear();

if (p == null) return FALSE;

while (it = leerSimbolo (string)  $\neq$  EoS) {

if (esSimboloApertura (it) {

if (pila-push (p, it) == null) {

return FALSE;

} else if (esSimboloCierre (it) {

if (ele = pila-pop (p) == null || sonPareja (ele, it) == FALSE) {

pila-destroy (p);

return FALSE;

}

element-destroy (ele);

}

}

if (pila-vacia (p) == FALSE)

pila-destroy (p);

return FALSE;

}

pila-destroy (p);

return TRUE;

}

## Evaluación de Expresión Posfijo (pseudocódigo)

status evaluarPosfijo (cc string, int r) {

stack p;

if (string == NULL || r == NULL) return ERROR;

p = pila-crear();

if (p == NULL) return ERROR;

while (it = leerItem(string) != EOS) {

if (esOperando(it)) {

if (pila-push(p, it) == NULL) {

pila-destroy(p);

return ERROR;

} else if (esOperador(it)) {

if (ele1 = pila-pop(p) == NULL || ele2 = pila-pop(p) == NULL) {

pila-destroy(p);

return ERROR;

ele3 = evaluar(ele1, ele2, it);

if (pila-push(p, ele3) == NULL) {

pila-destroy(p);

return ERROR;

} if (r = pila-pop(p) == NULL) {

pila-destroy(p);

return ERROR;

if (pila-vacia(p) == FALSE) {

pila-destroy(p);

return ERROR;

} return OK;

}

```

status int;
if (S1==NULL || S2==NULL) return ERROR;
p = pila - crear();
if (p == NULL) return ERROR;
while (it = leerItem(S1) != EOS) {
    if (esOperando(it)) {
        if (printf(S2, it) == ERROR ERROR) {
            pila - destroy(p); → if (p != NULL)
            return ERROR;
        }
    }
    else if (esOperador(it)) {
        while (pila - vacia(p) == FASE && prec(it) ≤ prec(top(p))) {
            if (ele = pila - pop(p) == NULL) {
                pila - destroy(p);
                return ERROR;
            }
            if (printf(S2, ele) == ERROR) {
                pila - destroy(p);
                return ERROR;
            }
        }
        if (pila - push(p, it) == NULL) {
            pila - destroy(p);
            return ERROR;
        }
    }
}
while (pila - vacia(p) == FALSE) {
    if (it = pila - pop(p) == NULL) {
        pila - destroy(p);
        return ERROR;
    }
    if (printf(S2, it) == ERROR) {
        pila - destroy(p);
        return ERROR;
    }
}

```

```

(*) pila - destroy(p);
return OK;
} // fin función

```

$(b \vee (f * (p - (q + v))))$	!
$(f * (p - (q + v))) \vee$	g
$(p - (q + v)) * \vee$	f
$(q + v) - * \vee$	p
$\vee * - + a$	b
$\vee * - +$	a
$\vee * -$	+
$\vee *$	-
$\vee$	*
<u>Pila</u>	<u>lee</u>
$\vee * - + a b d f g : \rightarrow ((b \vee (f * (p - (q + v))))$	regio a infio

$abc \vee d +$	d	
$abc \vee$	$\vee$	
$abc \vee$	)	
abc	c	
ab	\	
ab	b	
a	(	
a	+	
a	a	
<u>exp. Salida</u>	<u>lee</u>	
$abc \vee d + :$		

```
string decimal2base (int num, int base) {
```

```
    p = pila::ini();
```

```
    s = string::ini();
```

```
    while (num/base  $\geq$  base) {
```

```
        resto = num % base;
```

```
        push(p, resto);
```

```
        num = num / base;
```

```
    }  
    push(p, num/base);
```

```
    push(p, num % base);
```

```
    for (i=0; i < top < i++) {
```

```
        c = pila::pop(p);
```

```
        printf(s, c);
```

```
    }
```

# TAD: LISTA

DEFINICIÓN: Colección de objetos donde:

- todos menos uno tiene un objeto "siguiente".
- todos menos uno tienen un objeto "anterior".

Permite la representación secuencial y ordenada de objetos de cualquier tipo.

PRIMITIVAS:

lista lista-crear()

void lista-eliminar(lista l)

status lista-insertarLprincipio(Lista l, elemento ele)

status lista-insertarLfinal(Lista l, elemento ele)

status lista-insertarLposicion(Lista l, Elemento ele, int posicion)

IMPLEMENTACIÓN EN C:

```
C struct _Nodo {  
    Elemento *info;  
    struct _Nodo *next;  
};
```

```
typedef struct _Nodo Nodo;
```

```
Nodo *nodo-crear() {  
    Nodo *n = NULL;  
    n = (Nodo *) malloc(sizeof(Nodo));  
    if (n == NULL) return NULL;  
    n->info = NULL;  
    n->next = NULL;  
    return n;  
}
```

```
void nodo-eliminar(Nodo *pn) {  
    if (pn == NULL) return  
    *↑  
    elemento-liberar(pn->info);  
    free(pn);  
}
```

```
* if (pn->info != NULL) ↓  
    elemento-liberar(pn->info);
```

```
typedef struct -Nodo {
    elemento *info;
    struct -Nodo *next;
} Nodo;
```

```
struct -Lista {
    Nodo *first;
}
```

```
typedef struct -lista Lista;
```

```
lista * lista-crear();
void lista-eliminar(lista *pl);
status lista-insertarLprincipio(lista *p
    elemento *el
```

```
lista * lista-crear() {
    Lista *l;
    l = (Lista *) malloc(sizeof(Lista));
    if (l == NULL) return NULL;
    l->first = NULL;
    return l;
}
```

```
status lista-insertarLprincipio(lista *pl, elemento *ele);
    if (pl == NULL || ele == NULL) return ERROR;
    n = nodo-crear();
    if (n == NULL) return ERROR;
    n->info = elemento-copiar(ele);
    n->next = pl->first;
    pl->first = n;
    return OK;
```

⊗ if (n->info == NULL) {  
 nodo-eliminar(n);  
 return ERROR;  
 }



```
Elemento * lista_extraerLprincipio (Lista * pl) {
```

```
    Elemento * copy;
```

```
    Nodo * pn;
```

```
    if (pl == NULL) return NULL;
```

```
    * copy = elemento_copiar(pl->first->info);    * if (copy == NULL) return NULL;
```

```
    pn = pl->first;
```

```
    pl->first = pl->first->next;
```

```
    nodo_eliminar(pn);
```

```
    return ele;
```

```
}
```

```
status lista_insertarFin (lista * pl, Elemento * ele) {
```

```
    Nodo * n;    Nodo * i;
```

```
    if (pl == NULL || ele == NULL) return ERROR;
```

```
    n = nodo_crear();
```

```
    if (n == NULL) return ERROR;
```

```
    n->info = elemento_copiar(ele);
```

```
    if (n->info == NULL) {  
        nodo_eliminar(n);  
        return ERROR;  
    }
```

```
    if (lista_vacia(pl) == TRUE) {  
        pl->first = n;  
        return OK;  
    }
```

```
    * for (i = pl->first ; i->next != NULL ; i = i->next);
```

```
    i->next = n;
```

```
    return OK;
```

```
}
```

```

    Nodo *pn;
    Elemento *aux;
    (A) if (pl == NULL)
        return NULL;

```

```

    (B) (C)
    for (pn = pl->first; (pn->next) != NULL; pn = pn->next);

```

```

    aux = element-copy(pn->next->info);
    (D) → if (aux == NULL) return NULL;

```

```

    nodo_eliminar(pn->next);

```

```

    pn->next = NULL;

```

```

    return aux;

```

```

    (E) if (pl->first->next == NULL) {
        aux = element-copy(pl->first->info);
        nodo_eliminar(pl->first);
        pl->first = NULL;
        return aux;
    }

```

```

void lista-liberar (lista *pl) {

```

```

    Elemento *ele;

```

```

    if (pl == NULL) return;

```

```

    while (lista-vacia(pl) != TRUE) {

```

```

        ele = lista-extraerIni(pl);

```

```

        elemento-liberar(ele);
    }

```

```

    Nodo *pn;

```

```

    while (listavacia(pl) != TRUE) {

```

```

        pn = pl->first;

```

```

        pl->first = pn->next;

```

```

        node-destroy(pn);
    }

```

③ Forma recursiva de lista-liberar.

# PILAS UTILIZANDO UNA LISTA COMO EDD

Pila.c

```
struct _PILA {  
    Lista *pl;
```

}

```
Elemento * pop (PILA *p) {
```

```
    Elemento *ele;
```

```
    if (p == NULL || pila-vacia(p) == TRUE)  
        return NULL;
```

```
    ele = lista-extraerIni(p->pl);
```

```
    return ele;
```

```
Pila * pila-ini() {
```

```
    PILA *p;
```

```
    p = (PILA *) malloc(sizeof(PILA));
```

```
    if (p == NULL) return NULL;
```

```
    lista-ini(p->pl);
```

```
    if (p->pl == NULL) {
```

```
        free(p);
```

```
        return NULL;
```

```
}
```

```
return p;
```

```

struct ListaEC {
    Nodo *last;
}

```

```

ListaEC * listaEC_ini() {
    ListaEC *pl;
    pl = (ListaEC *) malloc(sizeof(ListaEC));
    if (pl == NULL) return NULL;
    pl->last = NULL;
    return pl;
}

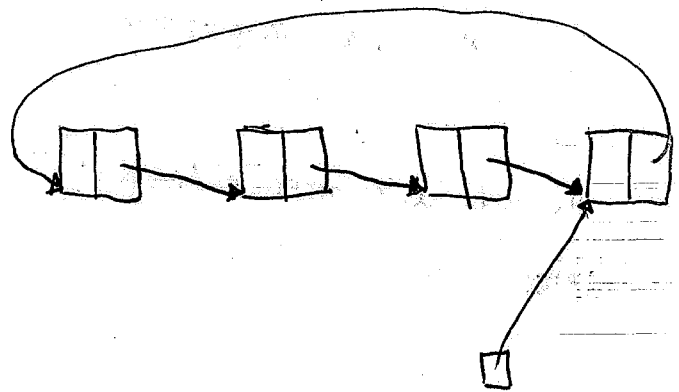
```

```

status listaEC_insertarEnd(ListaEC *pl, Elemento *ele) {
    Nodo *pn;

    pn = nodo_ini();
    pn->info = elemento_copiar(ele);
    pn->next = pl->last->next;
    pl->last->next = pn;
    pl->last = pn;
    return OK;
}

```



```

Elemento * listaEC_extraerIni(ListaEC *pl) {
    Elemento *aux;    Nodo *naux;
    aux = elemento_copiar(pl->last->next->info);
    naux = pl->last->next;
    pl->last->next = pl->last->next->next;
    nodo_eliminar(naux);
    return aux;
}

```

## EXERCÍCIOS LISTAS

### 1) ELIMINAR ÚLTIMO ELEMENTO

```
status lista_elimUlt(Lista *pl){
```

```
    Nodo *pn = NULL;
```

```
    if (pl->first == NULL) { // si lista vacia
```

```
        if (lista_vacia(pl) == TRUE)
```

```
            return OK;
```

```
    }
```

```
    if (pl->first->next == NULL) { // si solo 1 elemento
```

```
        nodo-liberar(pl->first);
```

```
        pl->first = NULL;
```

```
        return OK;
```

```
    }
```

```
    for (pn = pl->first; pn->next->next != NULL; pn = pn->next);
```

```
    nodo-liberar(pn->next);
```

```
    pn->next = NULL;
```

```
    return OK;
```

```
}
```

status lista\_insertarOrden (Lista \*pl, Elemento \*pe) {

Nodo \*pn = NULL;  
Nodo \*pnaux = NULL;

pn = nodo\_crear();

pn->info = elemento\_copiar(pe);

if (pl->first == NULL) // l. vacía

pl->first = pn;

return OK;

if (elemento\_comparar(pn->info, pl->first->info) < 0) // nuevo < 1er elemento lista

pn->next = pl->first;

pl->first = pn;

return OK;

for (pnaux = pl->first; pnaux->next != NULL &&

&& elemento\_comparar(pnaux->next->info, pn->info) < 0; pnaux = pnaux->next;

pn->next = pnaux->next;

pnaux->next = pn;

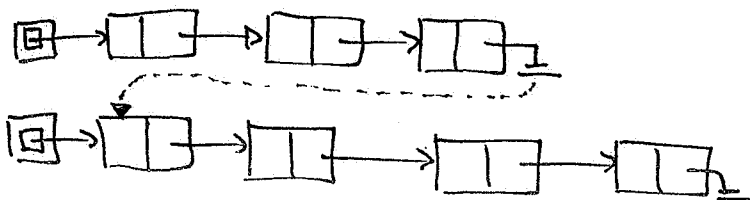
return OK;

Ⓐ if (pl == NULL || pe == NULL) {  
return ERROR;

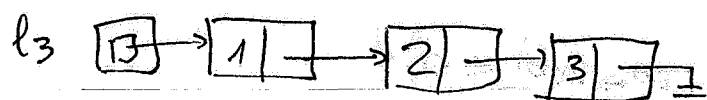
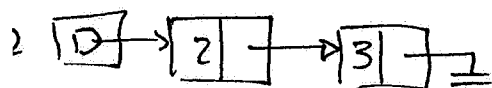
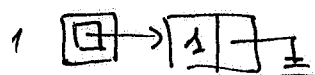
Ⓑ if (pn == NULL) {  
return ERROR;

Ⓒ if (pn->info == NULL) {  
nodo\_liberar(pn);  
return ERROR;

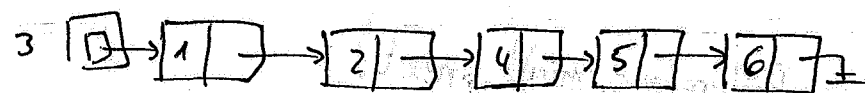
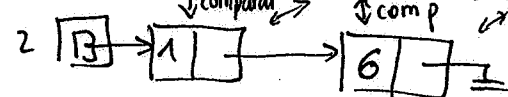
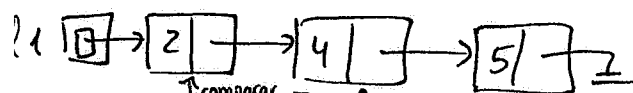
3) CONCATENAR DOS LISTAS EN LA 1ª. (planteamiento)



1) CONCATENAR DOS LISTAS EN UNA 3ª (dejando 1 y 2 como estaban)



2) CONCATENAR ELEMENTOS DE 2 LISTAS ORDENADAS EN UNA 3ª ord.



Función auxiliar para facilitar la vida:

```

puntero al nuevo nodo ← Nodo * lista_insertarTrasNodo (Nodo *pn, Elemento *pe) {
    Nodo *pn2 = NULL;    if (!pe || !pn)
                        return NULL;

    pn2 = nodo_crear();
    if (!pn2)
        return NULL;

    pn2->info = elemento_copiar(pe);
    if (!pn2->info)
        nodo_liberar(pn2);
    return pn2;
}

pn2->next = pn->next;
pn->next = pn2;
    
```

Nodo \*pa = NULL, \*pb = NULL, \*\*pc = NULL;

Lista \*lc = NULL;

Elemento min;

lc = lista-crear();

if (lc == NULL)

return NULL;

pa = lc->first;

pb = lc->first;

pc = &(lc->first);

while (pa != NULL && pb != NULL) {

if (elemento-comparar(pa->info, pb->info) < 0) {

min = pa->info;

pa = pa->next;

}

else {

min = pb->info;

pb = pb->next;

}

pc = &((listaInsertarTrasPuntero(ppc, min))>next);

}

if (pa != NULL) {

pb = pa; // En pb el que no es NULL;

}

while (pb != NULL) {

pc = &((listaInsertarTrasPuntero(ppc, pb->info))>next);

pb = pb->next;

}

return lc;

}



# IMPLEMENTACIÓN EN C DE ÁRBOLES BINARIOS

## ARBOL.C

```
typedef struct _NodoAB {  
    void *info;  
    struct _NodoAB *izq;  
    struct _NodoAB *der;  
} NodoAB;
```

```
typedef struct _ARBOL ARBOL;
```

```
struct _ARBOL {  
    NodoAB *root;  
    Destroy_ele_funct_type destr_ele_f;  
    Copy_ele_funct_type copy_ele_f;  
    Print_ele_funct_type print_ele_f;
```

```
NodoAB *nodo_crear() {  
    NodoAB *nodo;  
    nodo = (NodoAB *) malloc(sizeof(NodoAB));  
    if (nodo == NULL)  
        return NULL;  
    nodo->info = NULL;  
    nodo->izq = NULL;  
    nodo->der = NULL;  
    return nodo;  
}
```

```
void nodo_eliminar(NodoAB *n, ARBOL *ab) {  
    if (n == NULL)  
        return;  
    ab->destroy_element_function(n->info);  
    free(n);  
}
```

```
ARBOL *arbol_crear() {  
    ARBOL *ab;  
    ab = (ARBOL *) malloc(sizeof(ARBOL));  
    if (ab == NULL)  
        return NULL;  
    ab->root = NULL;  
    return ab;  
}
```

```
void arbol_eliminar(ARBOL *ab) {  
    if (ab == NULL)  
        return;  
    if (arbol_vacio(ab) == TRUE) {  
        free(ab);  
    }  
    arbol_eliminar_rec(ab->root);  
    free(ab);  
}
```

```
void arbol_eliminar_rec(NodoAB *n) {  
    if (n == NULL)  
        return;  
    arbol_eliminar_rec(n->der);  
    arbol_eliminar_rec(n->izq);  
    nodo_eliminar(n);  
}
```

Boolean abdb\_buscar (const ARBOL \*ab, const Elemento \*ele)

```
{  
    if (ab == NULL || ele == NULL) {
```

```
        return FALSE;
```

```
    }
```

```
    return abdb_buscar_rec(ab, ele);
```

```
}
```

Boolean abdb\_buscar\_rec (const NodoAB \*n, const Elemento \*ele)

```
{  
    int cmp;
```

```
    if (n == NULL) {
```

```
        return FALSE;
```

```
    }
```

```
    cmp = elemento_comparar(ele, n->info);
```

```
    if (cmp < 0) {
```

```
        return abdb_buscar_rec(n->izq, ele);
```

```
    }
```

```
    if (cmp > 0) {
```

```
        return abdb_buscar_rec(n->der, ele);
```

```
    }
```

```
    return TRUE;
```

```
}
```

CONTAR EL N° DE NODOS QUE HAY EN UN ÁRBOL

```
int abdb_num_nodos (const ARBOL *ab) {
```

```
    if (ab == NULL) return -1;
```

```
    return abdb_num_nodos_rec(ab->root);
```

```
}
```

```
int abdb_num_nodos_rec (NodoAB *n) {
```

```
    if (n == NULL) return 0;
```

```
    return (1 + abdb_num_nodos_rec(n->izq) + abdb_num_nodos_rec(n->der));
```

```
}
```

```
Status abdb-insertar(ARBOL *ab, const Elemento *ele){
```

```
    if(ab == NULL || ele == NULL){
        return ERROR;
```

```
    }
    return abdb-insertar-rec(&ab->root, ele);
```

```
Status abdb-insertar-rec(NodoAB **n, const Elemento *ele){
```

```
    int cmp;
```

```
    if(*n == NULL){
```

INSERTAR NODO

```
        *n = nodoAB-crear();
```

```
        if(*n == NULL) return ERROR;
```

```
        elemento-copiar(*n, ele);
```

\*n->info

```
    }
    cmp = elemento-comparar(ele, (*n)->info);
```

```
    if(cmp < 0){
```

```
        return abdb-insertar-rec(&(*n->izq), ele);
```

```
    }
    if(cmp > 0){
```

```
        return abdb-insertar-rec(&(*n->der), ele);
```

```
    }
    return OK;
```

CONTAR EN N° DE HOJAS QUE HAY EN UN ÁRBOL

```
int abdb-num-hojas(const ARBOL *ab){
```

```
    if(ab == NULL) return -1;
```

```
    return abdb-num-hojas-rec(ab->root);
```

```
int abdb-num-hojas-rec(NodoAB *n){
```

```
    if(n == NULL) return 0;
```

```
    if(n->izq == NULL && n->der == NULL) return 1;
```

```
    return(abdb-num-hojas-rec(n->izq) + abdb-num-hojas-rec(n->der));
```

## BUSCAR MAXIMO ARBOL

```
NodoAB* abdb_buscar_maximo(ARBOL *ab) {  
    if (ab == NULL) return NULL;  
    return abdb_buscar_maximo_rec(ab->root);  
}  
  
NodoAB* abdb_buscar_maximo_rec(NodoAB *n) {  
    if (n == NULL) return NULL;  
    if (n->der == NULL) return n;  
    return abdb_buscar_maximo_rec(n->der);  
}
```

análogo para  
buscar el

mínimo,  
cambiando

n->der por

n->izq

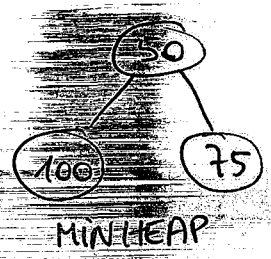
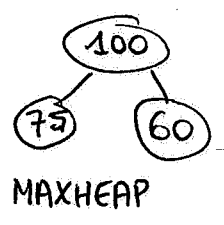
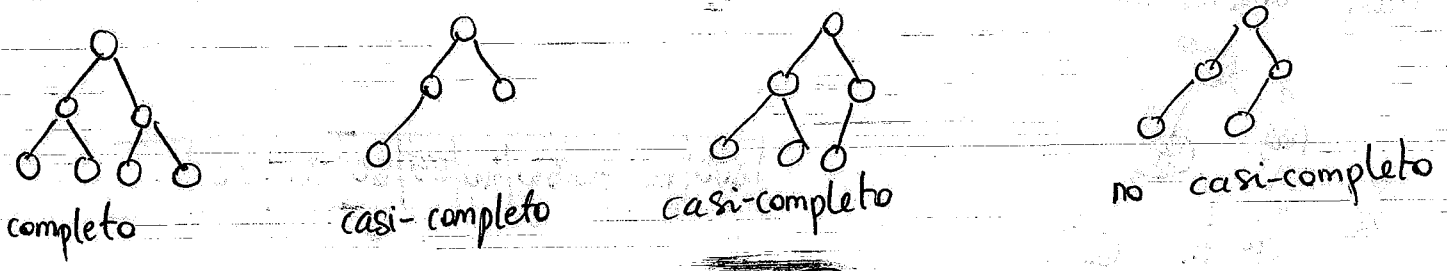
# COLAS DE PRIORIDAD

$\left. \begin{array}{l} \text{PILAS (LIFO)} \\ \text{COLAS (FIFO)} \end{array} \right\} \rightarrow \text{generalización} \rightarrow \text{COLAS DE PRIORIDAD}$

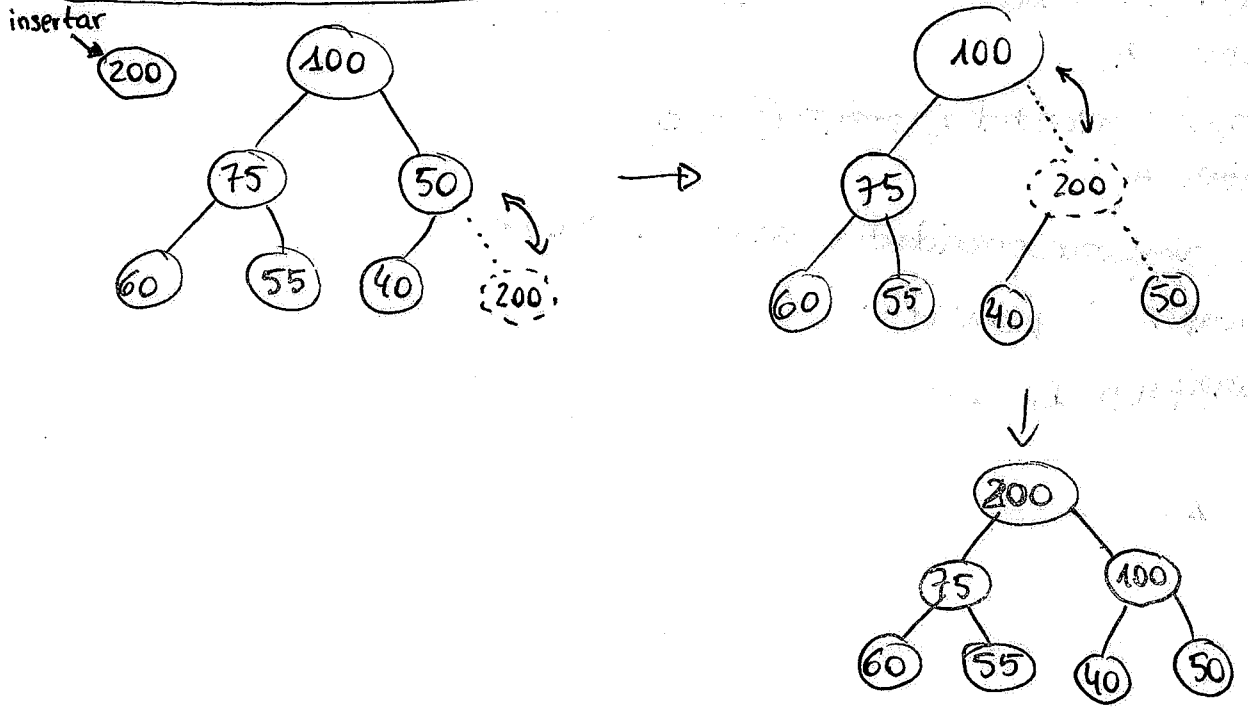
Como la EDD con arrays y listas es normalmente (o en extraer o en insertar) de coste  $O(n)$ , la EDD serán los HEAPS

HEAPS (=TAD) es un tipo de árbol binario

ÁRBOL BINARIO ESTRÍCTAMENTE CASI-COMPLETO: PRIORIDAD DE HEAD



## • ALGORITMO = HEAPIFYUP



left(i)  $\leadsto (2i+1)$

right(i)  $\leadsto (2i+2)$

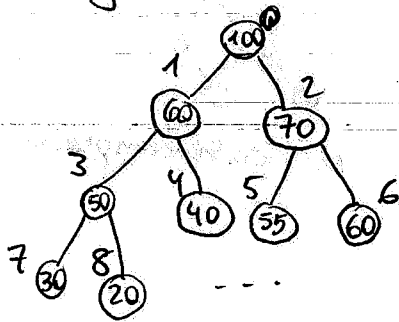
parent(i)  $\leadsto \lfloor (i-1)/2 \rfloor \equiv \text{floor}(\frac{i-1}{2})$

heap-size(A)  $\leadsto$  número de nodos - 1  $\equiv$  posición en el array del último elemento

swap(A, i, j)  $\leadsto$  cambia el info de índice i por el info de j.

EdD HEAP

Array ordenado de tal forma:



100	60	70	50	40	55	60	30	20	...
0	1	2	3	4	5	6	7	8	...

### • Pseudocódigo del algoritmo HeapifyUp

```
Heap heapifyUp (Heap A, int i) {
    if (parent(i) == NULL)
        return A;
    if (comparar-prioridad(i, parent(i)) < 0)
        return A;
    else if (comparar-prioridad(i, parent(i)) > 0) {
        swap(A, i, parent(i));
        heapifyUp(A, i);
    }
    return A;
}
```

```
Status abdb-insertar(ARBOL *ab, const Elemento *ele){
```

```
    if(ab == NULL || ele == NULL){
```

```
        return ERROR;
```

```
    }
```

```
    return abdb-insertar-rec(← direcciónab->root, ele);
```

```
}
```

```
Status abdb-insertar-rec(NodoAB **n, const Elemento *ele){
```

```
    int cmp;
```

```
    if(*n == NULL){
```

```
        *
```

INSERTAR NODO

```
        *n = nodoAB-crear();
```

```
        if(*n == NULL) return ERROR;
```

```
        elemento-copiar(*n, ele);
```

(\*)->info

```
    cmp = elemento-comparar(ele, (*n)->info);
```

```
    if(cmp < 0){
```

```
        return abdb-insertar-rec(&(*n->izq), ele);
```

```
    }
```

```
    if((*n)->info == NULL){
        nodo-liberar(*n);
        *n = NULL;
        return ERROR;
    }
```

```
    return OK;
```

```
    if(cmp > 0){
```

```
        return abdb-insertar-rec(&(*n->der), ele);
```

```
    }
```

```
    return OK;
```

CONTAR EN Nº DE HOJAS QUE HAY EN UN ÁRBOL

```
int abdb-num-hojas(const ARBOL *ab){
```

```
    if(ab == NULL) return -1;
```

```
    return abdb-num-hojas-rec(ab->root);
```

```
}
```

```
int abdb-num-hojas-rec(NodoAB *n){
```

```
    if(n == NULL) return 0;
```

```
    if(n->izq == NULL && n->der == NULL) return 1;
```

```
    return (abdb-num-hojas-rec(n->izq) + abdb-num-hojas-rec(n->der));
```

## BUSCAR MÁXIMO ÁRBOL

```
NodoAB* abdb_buscar_maximo(ARBOL *ab){
```

```
    if (ab == NULL) return NULL;
```

```
    return abdb_buscar_maximo_rec(ab->root);
```

```
}
```

```
NodoAB* abdb_buscar_maximo_rec(NodoAB *n){
```

```
    if (n == NULL) return NULL;
```

```
    if (n->der == NULL) return n;
```

```
    return abdb_buscar_maximo_rec(n->der);
```

```
}
```

análogo para  
buscar el  
mínimo,  
cambiando  
n->der por  
n->izq



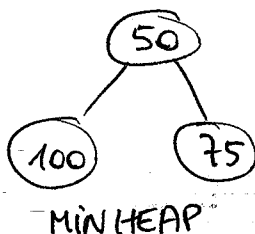
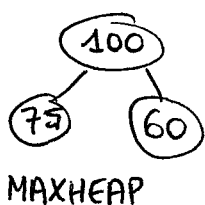
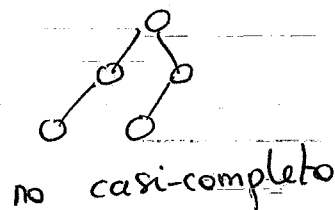
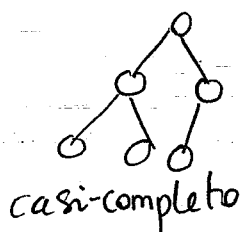
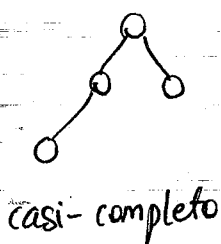
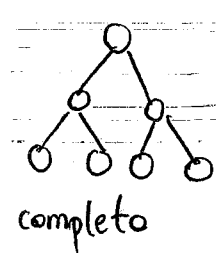
# COLAS DE PRIORIDAD

$\left. \begin{array}{l} \text{PILAS (LIFO)} \\ \text{COLAS (FIFO)} \end{array} \right\} \rightarrow \text{generalización} \rightarrow \text{COLAS DE PRIORIDAD}$

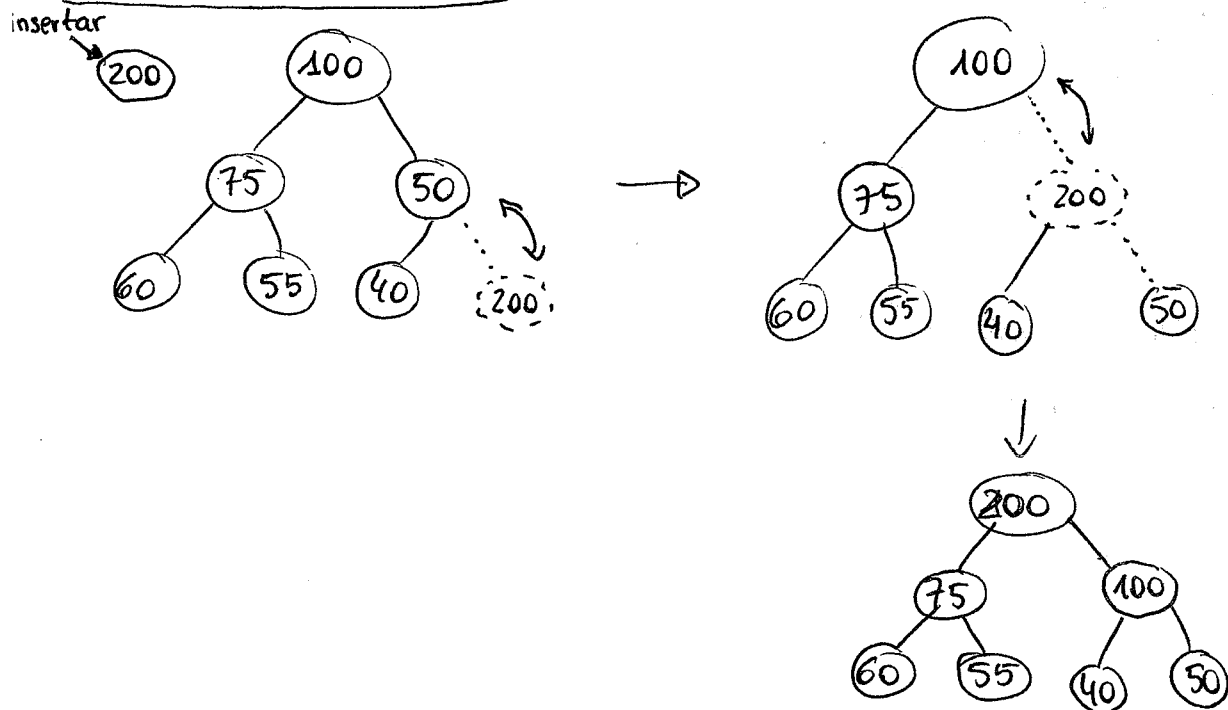
Como la EdD con arrays y listas es normalmente (o en extraer o en insertar) de coste  $O(n)$ , la EdD serán los HEAPS

HEAPS (=TAD) es un tipo de árbol binario

ÁRBOL BINARIO ESTRÍCTAMENTE CASI-COMPLETO: PRIORIDAD DE HEAP



## • ALGORITMO - HEAPIFYUP



$\text{left}(i) \leadsto (2i+1)$

$\text{right}(i) \leadsto (2i+2)$

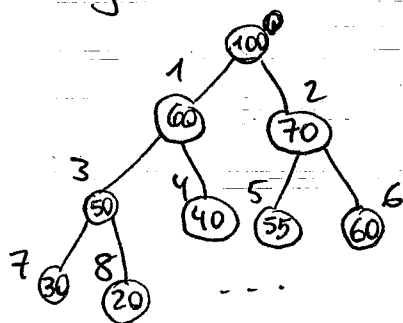
$\text{parent}(i) \leadsto \lfloor (i-1)/2 \rfloor \equiv \text{floor}(\frac{i-1}{2})$

$\text{heap-size}(A) \leadsto \text{número de nodos} - 1 \equiv \text{posición en el array del último elemento}$

$\text{swap}(A, i, j) \leadsto \text{cambia el info de índice } i \text{ por el info de } j.$

### EdD HEAP

Array ordenado de tal forma:



100	60	70	50	40	55	60	30	20	...
0	1	2	3	4	5	6	7	8	...

### Pseudocódigo del algoritmo HeapifyUp

```
Heap heapifyUp (Heap A, int i) {  
    if (parent(i) == NULL)  
        return A;  
    if (comparar-prioridad(i, parent(i)) < 0)  
        return A;  
    else if (comparar-prioridad(i, parent(i)) > 0) {  
        swap(A, i, parent(i));  
        heapifyUp(A, i);  
    }  
    return A;  
}
```

## Pseudocódigo del algoritmo HeapifyDown

1 HeapifyDown(Heap A, int i) {

    max = i;

    l = left(i);

    r = right(i);

    if (l < heap-size AND  $A[l] > A[i]$ ) {

        max = l;

    if (r < heap-size AND  $A[r] > A[max]$ ) {

        max = r;

    if (max  $\neq$  i) {

        swap(A, i, max);

        heapifyDown(A, max);

    }

return;

