# PROGRAMACION II

## MATERIAL para TEORIA

## Lenguaje C

**Código Asignatura 6A4**

**Año 2018**

**Primer Cuatrimestre**

# Introducción al Lenguaje C

## RESEÑA HISTÓRICA

El lenguaje C fue desarrollado por Dennis Ritchie en 1972 en los laboratorios de Bell mejorando el lenguaje B de Thompson. Pronto, empezaron a surgir muchas implementaciones del lenguaje C, a raíz de su popularidad. Por este motivo, se hizo necesario definir un estándar que está representado hoy por el ANSI C.

## CARACTERÍSTICAS DE C

- El lenguaje de programación C está caracterizado por ser de uso general, con una sintaxis sumamente corta y un juego de operadores muy potente.
- C es un lenguaje de "medio-bajo nivel". Esto significa que explota los recursos Hardware así como el manejo de las direcciones y los bits.
- No posee operaciones de entrada-salida, ni métodos de archivos, y tampoco maneja cadenas de caracteres. Estas operaciones se hacen por medio de funciones contenidas en librerías externas al lenguaje, lo que le confiere un alto grado de portabilidad.
- C es un lenguaje semi-estructurado, por la razón de que no se puede declarar funciones dentro de otras. La estructura básica de un programa C se hace alrededor de la función; ésta puede tener sus propias variables locales, acceder a las variables globales del programa, y finalmente devolver uno o más valores o ninguno.
- C es un lenguaje que permite una compilación separada. Se puede partir un archivo en varios archivos más pequeños y que cada uno sea compilado por separado. Luego se enlazan entre sí junto con las rutinas de biblioteca para formar el ejecutable completo. Su ventaja reside en que cambios de código en uno de los archivos no necesita una nueva compilación del programa entero.
- Un programa en C durante su ejecución crea y usa 4 regiones de memoria lógicas diferentes: la primera región contiene el código del programa, la segunda contiene las variables globales, la tercera, es la pila (Stack) que almacena temporalmente las direcciones de las funciones, el estado del procesador, etc., y la cuarta llamada es el Heap, que es la región libre, que sirven para reservación dinámica de memoria.

La descripción del lenguaje se realiza siguiendo las normas del ANSI C, por lo tanto, todo lo expresado será utilizable con cualquier compilador que se adopte.

## PROGRAMAS EN C

Los programas se construyen con:

- ✓ Comentarios.
- ✓ Ordenes para el preprocesador
- ✓ Definiciones de funciones
- ✓ Expresiones formadas con constantes, variables, funciones y operadores.
- ✓ Sentencias

☞ **Ejemplo**

```
#include <stdio.h>            /* inclusión de un archivo */
main()
{
   printf("Bienvenido");      /* una sentencia */
   return 0;                  /*termina el programa */
}
```

## COMENTARIOS

Para el compilador, los comentarios son inexistentes, por lo que no generan líneas de código, permitiendo abundar en ellos tanto como se desee. En C se toma como comentario todo carácter interno a los símbolos: /* */ y pueden ocupar uno o más renglones

```
/* este es un comentario */
/* Este es otro comentario más
largo que el anterior */
```

## FUNCIÓN MAIN()

main() es la función principal, indicando donde empieza el programa su cuerpo es un conjunto de sentencias delimitadas por llaves. Cada sentencia termina con el punto y coma ";".
Es la función invocada desde el sistema operativo cuando comienza la ejecución del programa. La sentencia "return 0" termina el programa y devuelve un valor al Sistema.

```
void main () {
…….
}

int main () {
…….
return 0;
}
```

## ENCABEZADO - SENTENCIAS PARA EL PREPROCESADOR

Son órdenes que el preprocesador interpreta antes de que el código fuente sea compilado, comienzan con # y se encuentran antes del *main()*.
El preprocesador produce un programa escrito en C que es lo que se compila después.
La directiva **#include** es una orden para leer un archivo de texto especificado en el nombre que sigue a la misma **<stdio.h>**. El preprocesador reemplaza la directiva por el contenido de dicho archivo, que contiene las declaraciones de las funciones invocadas por el programa.
La directiva **#include** no es una sentencia de programa, por lo que no es necesario terminarla con el punto y coma '**;**'.
El compilador tiene varias formas de buscar un archivo.

```
#include <stdio.h>             /* Carpeta de instalación */
#include "stdio.h"             /* Carpeta activa en el momento */
#include "g:\milibreria.h"     /* Carpeta indicada */
```

La directiva **#define** se utiliza para declarar constantes que pueden ser identificadores y también expresiones. Lo que *#define* permite en definitiva, es operar con macros (expansión de código).
**#define** *nombre texto_reemplazo* luego cada vez que se hace mención a *nombre* se reemplaza por el *texto_reemplazo*

☞ **Ejemplo de utilización de define**

```
#define PI 3.1416              /* Definición de constante simbólica */
#define X PI+2                 /* Definición de X en base a PI */
#define prod(a,b)(a*b)         /* Macros con parámetros, que si luego se
            escribe x=prod(2,(4+3)) se reemplazará por x=(2*(4+3)); */
```

## IDENTIFICADORES

En C, los nombres de las variables, funciones y otros objetos definidos por el usuario se conocen como **identificadores**. La longitud de un identificador puede variar de uno a varios caracteres, siendo el primero una letra o un guión bajo y los siguientes una combinación de letras, números y guiones bajos.
Ningún identificador debe coincidir con las palabras reservadas del lenguaje, ni con el nombre de alguna función de librería.
En C, las minúsculas y las mayúsculas se tratan como distintas (es *case sensitive*)

```
cont, prueba10, balance_total  /* Son válidos */
1cont, hola!, balance..total   /* Son inválidos */
```

## TIPOS DE DATOS

Existen cinco tipos de datos básicos en C; **char** (carácter), **int** (entero), **float** (real coma flotante), **double** (real de doble precisión) y **void** (sin valor).
El tipo **void** se usa para declarar funciones que no devuelven ningún valor o para declarar algún puntero genérico.

**Modificadores de tipos**

Los modificadores de tipos se usan para alterar el tipo base: **short**, **long**, **unsigned, signed**. Los modificadores se pueden aplicar a los tipos base int y char; **long** también se puede aplicar a double.
Por ejemplo, el tipo *int* permite valores en el rango *-32768 a 32767* mientras que *unsigned int* permite valores en el rango *0 a 65535*.

## VARIABLES

Todas las variables deben declararse antes de ser usadas; esto es, indicar el tipo de datos al que pertenecen. Esto determina:

- Su representación en memoria
- El conjunto de valores posibles
- Las operaciones permitidas y la forma de realizarlas

☞ **Ejemplo de declaración de variables**

```
int i,j,k, v[20], m[10][10];   /* 3 variables enteras, un vector y una
                                  matriz de enteros  */
float beneficio, perdida;      /* 2 variables reales */
```

Existen tres sitios donde se pueden declarar variables: dentro de las funciones "**variables locales"**, en la definición de parámetros de funciones "**parámetros formales"** y fuera de todas las funciones "**variables globales**".

**Dónde se declaran las variables**

- **Variables locales**

Llamadas también variables automáticas y se declaran dentro de las funciones y son referenciadas sólo por sentencias que estén dentro del bloque donde están declaradas.
Recordar que un bloque está encerrado entre dos llaves. Las variables locales existen sólo cuando se ejecuta el bloque y se destruye al salir de él.

- **Parámetros formales**

Los parámetros formales son los argumentos de una función. Su declaración se hace entre los paréntesis de la función. Su comportamiento es igual que las variables locales de cualquier función; o sea desaparecen al terminar su ejecución.

- **Variables globales**

Las variables globales se conocen y se pueden usar en cualquier parte del programa. Además mantienen su valor a lo largo de la ejecución del programa.
Las variables globales se crean justo después de declaración, y por lo general, se hace al principio del programa y fuera de todas las funciones.
Se puede declarar una variable local a una función con el mismo nombre de una variable global existente, pero el compilador las trata distintamente; Dentro de esa función, prevalece la variable local.

**Inicialización de variables**

Todas las variables se pueden inicializar en el momento de su declaración.

☞ **Ejemplo de inicialización en la declaración**

```
float cont=0, max=-1;
char c='a';
int x, y = 6;                  /* sólo inicializa y */
```

## EXPRESIONES

Una expresión se forma combinando constantes, variables, operadores e invocaciones a funciones.

☞ **Ejemplo**

```
s = s + i
n == 0
++i
```

Una expresión representa un valor que es el resultado de realizar las operaciones indicadas siguiendo las reglas de evaluación establecidas en el lenguaje.

## OPERADORES

### Operadores Aritméticos

Los operadores aritméticos comprenden las operaciones básicas: suma (+), resta (-), multiplicación (*), división (/), autoincremento (++), autodecremento (--) y módulo (%).

El operador módulo: % se utiliza para calcular el **resto** del cociente entre dos enteros, y no puede ser aplicado a variables del tipo float ni double.

Se puede observar que no existen operadores de potencia, radicales, logaritmos, etc. En C todas estas operaciones (y muchas otras) se realizan por medio de llamadas a funciones de librería.

Los operadores ++/-- suelen emplearse con variables de tipo **int,** aunque se pueden usar sin problemas con cualquier otro tipo de variable.

La colocación de los operadores como **prefijo** ó **sufijo** son equivalente y tienen el mismo efecto, excepto cuando haya otros operadores; la posición de los operadores (**++,--**) indica el momento de la operación de incremento o decremento con respecto de la otra operación.

☞ **Ejemplo de autoincremento/autodecremento**

```
n = n + 1; es equivalente a n++; o ++n;
m = m - 1; es equivalente a m--; o --m;
m = 4; n = ++m;                 /* Se incrementa m y se asigna su valor a
                                   n; (n = m = 5) */
m = 4; n = m++;                 /* Se asigna m a n y luego se incrementa,
                                   (n = 4 y m = 5).
En ambos casos el valor de m es 5, pero el valor de n es distinto.
```

### Operadores relacionales

Todas las operaciones relacionales dan sólo dos posibles resultados: **falso** o **verdadero**. En C, no existe el tipo booleano, por lo que falso queda representado por cualquier expresión cuyo valor es nulo (cero) y verdadero por cualquier número distinto de cero. No hay que confundir el operador relacional igual que (==) con el de operador de asignación igual a (=).

| Símbolo | Significado | Símbolo | Significado | Símbolo | Significado |
|---|---|---|---|---|---|
| == | igual | < | menor | > | mayor |
| != | distinto | <= | menor o igual | >= | mayor o igual |

### Operadores lógicos

C posee 3 operadores lógicos.

| Símbolo | Significado | Símbolo | Significado | Símbolo | Significado |
|---|---|---|---|---|---|
| && | AND | \|\| | OR | ! | NOT |

### Asignación

☞ Sintaxis: identificador = expresión;

El *identificador* representa generalmente una variable y *expresión* puede ser una constante, una variable o una expresión más compleja. El *identificador* se conoce como **lvalue** y la *expresión* por **rvalue**. Una asignación es simplemente la copia del resultado de una expresión **rvalue** sobre otra **lvalue**, por lo tanto, **lvalue** debe poseer una posición de memoria.

```
a =17;
17 = a;                         /* Incorrecta pues 17 no es lvalue */
```

**Asignación múltiple**

- ☞ Sintaxis: identificador1 = identificador2 = identificador3 = expresión;

En C, se permite la asignación múltiple, y se evalúa de derecha a izquierda.

```
a = b = c = 5;                  /* las variables toman el valor 5 */
```

**Asignación condicional**

- ☞ Sintaxis: identificador = (expresion_1) ? (expresion _2) : (expresion_3) ;

La expresión *expresion_1* que es tipo relacional con un resultado **verdadero** o **falso** se evalúa, y si el resultado es **verdadero**, se asignará al *identificador* el valor de *expresion_2* y si el resultado es **falso**, se le asignará al Identificador valor de *expresion_3*.

- ☞ **Ejemplos de asignación condicional**

```
menor =(a<b)? a:b ;             /* Se elige el menor de a y b */
valor =(a>0)? -a:a;             /* Asigna el negativo */
return (a<b)? a:b ;
```

**Operadores combinados con asignación**

- ☞ Sintaxis: identificador *op*= expresion;

C posee otros operadores de asignación que se combinan con los operadores aritméticos para abreviar las operaciones. Así una operación como a = a + b, se puede abreviar como: a += b.

No se admite un espacio en blanco entre los símbolos. Ésta asignación se extiende a todos los operadores aritméticos.

- ☞ **Ejemplo**

```
int i = 5, j = 7;
float f = 5.5, g = -3.25;
f -= g;                         /* f = f-g; f = 8.75 */
j *= (i-3);                     /* j = j*(i-3); j=14 */
```

**Conversión de tipo**

- ↳ **Conversión Automática**

Cuando dos o más tipos de variables distintas se encuentran dentro de una misma operación o expresión matemática, ocurre una conversión automática del tipo de las variables

- ↳ **Conversión por asignación**

En una asignación, *lvalue = rvalue;* el cálculo de *rvalue* se hace de acuerdo con las reglas de conversión automática, pero luego se ajusta al tipo de *lvalue.*

- ↳ **Conversión explícita (cast)**
  - ☞ Sintaxis: (*tipo de dato*) *operando*;

El lenguaje permite cambiar el tipo de una expresión imponiendo el tipo del resultado de una operación mediante el operador: *(tipo de dato) operando*, donde tipo puede ser (char, int, float, double, long, unsigned, etc.). El *cast* no cambia el tipo de dato de las variables o expresiones, sino cambia el contenido a tomar una forma determinada.

- ☞ **Ejemplos**

```
double x, z, k=2.25;
int y, i=3;
y = k*i;                        /* trunca 6.75 a 6 pues y es entero */
x = (int)(k*i);                 /* trunca 6.75 a 6 pues hay cast */
z = (int)k*i;                   /* trunca k a 2 pues hay cast */
```

**Tabla de precedencia de algunos operadores ordenada de mayor a menor**

| Operador | Asociatividad |
|---|---|
| () [] -> . | I a D |
| sizeof (tipo) ! ++ -- * | D a I |
| * / % | I a D |
| + - | I a D |
| >> << | I a D |
| == != | I a D |
| & | I a D |
| ^ | I a D |
| && | I a D |
| \|\| | I a D |
| ? : | D a I |
| = += *= otros | D a I |

```
a - b / c * d      /* Primero b/c, luego se multiplica por d, y finalmente
                      se resta el producto a a */
m = a*++n;         /* Primero se incrementa n y luego se multiplica por a
                      y luego se asigna a m */
```

## SENTENCIAS

Una sentencia se forma con expresiones. Un bloque empieza con " **{** " y termina con " **}** ", pudiendo contener cualquier número de sentencias y declaraciones.
Todas las sentencias, excepto los bloques, terminan con el símbolo " **;**

## ENTRADA / SALIDA

El conjunto de las funciones de entrada /salida están definidas en el archivo **stdio.h**, pero en algunos compiladores se pude tener algunas funciones de entrada/salida definidas en **conio.h**.
Cuando se emplea alguna función de entrada/salida, se debe incluir el archivo al comienzo del programa mediante la directiva **#include <stdio.h>** y el compilador inserta el código fuente al comienzo del archivo.

### Entrada/Salida con formato

**La función printf()**

Para visualizar datos por la pantalla, se dispone de la función **printf()** que permite formatear la salida. Esta función puede visualizar una combinación de valores numéricos, caracteres y cadenas de caracteres.

- Sintaxis: **printf**(*cadena_de_control*, *arg1*, *arg2*, ...., *argn*);

La *cadena_de_control* contiene la información sobre el formato de las salidas de los *argumentos*, los cuales pueden ser constantes, variables, referencias a funciones, o cualquier expresión C. La *cadena_de_control* está compuesta por grupos de caracteres, donde cada grupo encabezado por el símbolo "**%**" representa el formato del dato de salida. En la *cadena de control*, se puede intercalar texto sin formato para comentar las salidas.
Debe haber igual número de comandos de formato como argumentos. Si hay más argumentos que formatos se descartan los que sobran, pero si hay menos, la salida es indefinida.

**Tabla de los códigos de los formatos posibles**

| Formato | Visualización de datos |
|---|---|
| %c | Carácter |
| %s | Cadena de caracteres |
| %d | Entero decimal con signo |
| %u | Entero decimal sin signo |
| %f | Coma flotante |
| %e | Coma flotante en notación científica. |
| %p | Dirección del puntero. |

| Ejemplo Formato | Otras visualizaciones de datos |
|---|---|
| %10d | escribe un entero en un campo de 10 dígitos. |
| %5.2f | escribe un float en un campo de 5 dígitos donde 2 de ellos son decimales. |
| %06d | rellenara con ceros, en vez de espacios en blanco, los números menores de 6 |
| %-6d | justifica a la izquierda el número en un campo de 6 dígitos |
| %5-7s | escribe una cadena de entre 5 y 7 caracteres. Se truncará si tiene más de 7 y se rellenará con espacios en blanco si tiene menos de 5. |

☞ . **Ejemplos**

```
char ch = 'A';
int m = 4;
float x = 14.250;
printf("ch=%c, m=%d", ch, m);          /* ch=A , m=4 */
printf("ch=%d, x=%f", ch, x);          /* ch=65, x=14.250 */
printf("x=%7.1f, \n m=%05d", x, m);    /* x= 14.2,
                                          m=00004 */
printf("La suma x+m=%f",x+m);          /* La suma x+m=18.250 */
printf("La tecla es %c\n",getch());    /* La tecla es D y baja de linea */
```

**La función scanf()**

Se utiliza para leer los datos desde el teclado y la sintaxis es parecida a la función anterior.

☞ Sintaxis: **scanf**(*cadena_de_control*, *arg1*, *arg2*, ...., *argn*);

La *cadena_de_control* contiene la información sobre el formato de las entradas y son idénticos a los formatos vistos de la función **printf().** Pero, los *argumentos* son direcciones de las variables y para ellos se debe utilizar el operador de las direcciones "&".

Cuando se trata de leer una cadena, el argumento no debe ir precedido por *ampersand*, ya que el propio nombre del arreglo indica la dirección del mismo.

La ausencia del & cuando el mismo sea necesario, no será detectado como error.

☞ **Ejemplos**

```
int n;
float x;
char cadena[20];
........
scanf("%f %d",&x, &n);
scanf("%s ", cadena);
```

**Entrada/Salida sin formato**

Para operar con **caracteres** (teclado/pantalla) se tienen las siguientes funciones:

- ✓ **getche()** espera hasta que se pulse una tecla y devuelve su valor; la tecla pulsada aparece en pantalla.
- ✓ **getch()** opera como la anterior, pero sin eco, es decir el carácter pulsado no aparece en la pantalla.
- ✓ **getchar()** lee un carácter del teclado (con eco) y devuelve su valor cuando se pulsa ENTER.
- ✓ **putchar( )** imprime un carácter en la pantalla.

☞ **Ejemplos**

```
char ch;
ch = getchar();
putchar(ch);
putchar('B');
putchar(98);                    /* Escribe B */
```

Para leer o escribir una **cadena de caracteres** (teclado/pantalla), se tienen las siguientes funciones:

- ✓ **gets( )** lee una cadena de caracteres hasta pulsar ENTER. La cadena se almacena en un arreglo de caracteres y en lugar de nueva línea (\n), se pone el carácter null (\0).
- ✓ **puts ( )** escribe una cadena de caracteres en la pantalla y traduce el \0 por un \n (newline).

☞ **Ejemplos**

```
char linea[80], ch;
........
gets(ch);
gets(linea);
puts(ch);
puts(linea);
puts("UNO");
```

## SENTENCIAS de DECISION

### Sentencia if-else

☞ Sintaxis: **if** (*expresión*) *sentencia1* [*else sentencia 2]*;

### El operador condicional (? :)

☞ Sintaxis: (*condicion*) ? *expresion1* : *expresion2*;

### Sentencia switch

☞ Sintaxis: **switch** (*expresion*) {
**case** *constante1***:** *sentencia1***;**
**break;**
**case** *constante2***:** *sentencia2***;**
**break;**
**……**
**default:** *sentencian***;**
}

La ejecución del **switch** comienza con la evaluación de *expresión* (int o char), y luego se compara sucesivamente con todas las etiquetas**.** Cuando se iguala a una de ellas, se ejecuta la sentencia correspondiente. Si no aparece la palabra **break,** continúa la ejecución del resto de las sentencias, pero si aparece un **break,** se termina la ejecución del **switch**. Al final del **switch**, aparece una opción optativa llamada **default**, que implica que si no se ha cumplido ningún **case**, ejecute lo que sigue. En este caso no es necesario un **break**. Una característica poco obvia del **switch**, es que si se eliminan los **break** al resultar *verdadera* una sentencia de comparación, se ejecutarán las sentencias de ese **case** particular pero también las que se hallen por debajo.

☞ **Ejemplos de equivalencia entre estructuras**

```
if (a<b)
   a;
else
   b;
```

⇨

```
(a<b) ? a : b;
```

```
if (ch == 'a' || ch == 'A')
   conta++;
else
   if (ch == 'e' || ch == 'E')
     conte++;
```

⇨

```
switch (ch) {
 case 'a':
 case 'A':conta++;
          break;
 case 'e':
 case 'E':conte++;
          break;
}
```

## SENTENCIAS de REPETICION

### Sentencia while

☞ Sintaxis: **while** (*expresion*)
*Sentencia*/s;

☞ **Ejemplo** Mostrar los dígitos de 0 a 9

**OPCION 1**

```
int digito = 0;
while(digito <= 9){
  printf("%d \n", digito);
  ++digito;
}
```

**OPCION 2**

```
int digito = 0;
while(digito <= 9)
   printf("%d \n", digito++);
```

### Sentencia do-while

☞ Sintaxis: **do**
*Sentencia*/s;
**while** (*expresion*);

### Sentencia for

☞ Sintaxis: **for**(*expresion1*; *expresion2* ; *expresion3*)
*Sentencia/s*;

dónde *expresion1* es una asignación de una o más variables que equivale a una inicialización de las mismas, *expresion2* representa una condición que debe ser satisfecha para que continúe la ejecución de las iteraciones y *expresion3* es otra asignación, que comúnmente varía alguna de las variables contenida en *expresión2*. Cuando se ejecuta un **for**, en primer lugar se evalúa *expresion2* y se comprueba antes de cada iteración del bucle, y al final de cada pasada se evalúa *expresion3*, por lo tanto un bucle **for** es equivalente a un bucle **while** como se muestra a continuación:

*expresion1*;
**while**(*expresión2*) {
*sentencia/s*;
*expresion3*;
}

☞ **Ejemplo** Mostrar los dígitos de 0 a 9

**OPCION 1**

```
int digito;
for (digito=0;digito<=9;digito++)
  printf("%d \n", digito);
```

**OPCION 2**

```
int digito;
 for (digito=0;digito<=9;)
  printf("%d \n", digito++);
```

**OPCION 3**

```
int digito=0;
for (;digito<=9;)
  printf("%d \n", digito++);
```

## EJERCICIOS

**1)** Leer un número y obtener y mostrar su factorial.

**2)** Leer un conjunto de números positivos y mostrar el factorial de cada uno de ellos.

**3)** Leer nombre y 3 notas de un conjunto de N alumnos, mostrar el nombre del alumno que obtuvo el mejor promedio.

## PUNTEROS

Un puntero es una variable que almacena una dirección de memoria.
Cuando se declara un puntero se reserva espacio para albergar una dirección de memoria, pero **no para almacenar el dato al que apunta el puntero**.
El espacio de memoria reservado para almacenar un puntero es siempre el mismo (4 bytes) independientemente del tipo de dato al que se apunte.
Los operadores ***** y **&** se utilizan para operar con direcciones de memoria. Son unarios (afectan sólo a un operando).

- ✓ el operador **&** permite obtener la dirección de una variable, y
- ✓ el operador ***** es el complementario de éste, pues al evaluarse devuelve el contenido de una dirección de memoria, por lo tanto el operando tiene que ser un puntero. El operador ***** sirve también para declarar una variable de tipo puntero a un tipo de dato determinado.

☞ **Ejemplo**

```
int m=1, n=2,z;
int *p, *q;                    /* p y q son punteros a enteros */

p = &m;                        /* p recibe la dirección de m */
*p = 4;                        /* sería incorrecto p=4; equivale m=4; */
printf("%d %p", *p, p);        /* contenido y la dirección de m*/
printf("%p %p", &m, p);        /* dirección de m dos veces */

q = &n;                        /* q recibe la dirección de n */
z= *q;                         /* equivale z = n; */
printf("%d %d", *p, *q);

p = q;
printf("%d %d", *p, *q);
```

## FUNCIONES

- ✓ Todo programa en C está formado por funciones (al menos *main()*).
- ✓ No se pueden definir funciones anidadas.
- ✓ Desde una función se puede invocar a otras, inclusive a ella misma (recursividad).
- ✓ **El mecanismo de paso es por valor.**

### Prototipos

El **prototipo** de una función es una declaración previa a la definición de dicha función. El prototipo sirve exclusivamente para informarle al compilador el tipo y el número de los argumentos así como el tipo de dato a devolver.
Los prototipos no son obligatorios en C, sin embargo son aconsejables ya que facilitan la comprobación de errores en la invocación respecto de la definición correspondiente.

La declaración de la cabecera, debe anteceder a la propia definición de la función.

☞ Sintaxis: *tipo_dato* nombreFuncion (*lista_tipo_argumentos*);

dónde *tipo_dato* es el tipo de dato devuelto por la función (en caso de obviarse el mismo se toma el tipo **int** por defecto). Para evitar malas interpretaciones es conveniente explicitarlo.
Cuando la función no devuelve ningún valor ("procedimiento"), se indica por medio de la palabra reservada **void** (sin valor).
*lista_tipo_argumentos* representa los tipos de datos de los argumentos que pueden ir acompañados o no de los nombres de los argumentos. Una función que no necesita argumentos se informa mediante la palabra **void** puesta entre los paréntesis.

☞ **Ejemplos**

```
int maximo(int,int);
void saludo();     /*  o void saludo (void);  */
```

**Definición de funciones**

La **definición** de una función puede ubicarse en cualquier lugar del programa, con sólo dos restricciones: debe hallarse después de dar su prototipo, y no puede estar dentro de la definición de otra función (incluida main() ). Es decir, a diferencia de Pascal, en C las definiciones no pueden anidarse.
La definición debe comenzar con un encabezamiento, que debe coincidir totalmente con el prototipo declarado para la misma, y a continuación se escribirán las sentencias que componen el cuerpo de la función que debe ir encerrado por llaves. En la definición de una función, el encabezamiento no termina con un punto y coma.

☞ **Ejemplo**

```
int maximo(int a, int b){
    Sentencias
}
void saludo(){
    Sentencias
}
```

**Salida (Sentencia return)**

Para salir de una función se utiliza la sentencia **return** que produce la salida inmediata de la función hacia la sentencia que la invocó. La sentencia puede devolver un valor acorde con el tipo usado al declarar la función.
En las funciones de tipo **void**, debe obviarse la sentencia **return**, y el control vuelve cuando se alcanza al final de la función.
Si la sentencia **return** incluye una expresión, el valor de la expresión se convierte (si es necesario) al tipo de la función, y ese valor se devuelve al medio que hizo la llamada.

☞ **Ejemplo**

```
int maximo(int a, int b){
    return (a>b)? a : b; */
}
void saludo(){
    printf("Bienvenido");
}
```

**Invocación de una función**

En el cuerpo de alguna otra función se produce la invocación, cuyos *parámetros actuales* deben coincidir en tipo y cantidad con los *parámetros formales* del prototipo de la función. Si los parámetros actuales fueran expresiones, se evalúan antes de copiarlos en los formales.

☞ **Ejemplo**

```
int main(){
  scanf(&a, &b);
  saludo();
  printf("El máximo de los valores leidos es %d", maximo(a,b));
return 0;
}
```

.

Los parámetros de una función pueden ser valores (**pasaje por valor**) o direcciones (**pasaje por referencia**)
En el pasaje de parámetros por **Valor** las modificaciones en los valores de los parámetros formales de la función, no afectan a las variables/constantes/expresiones del medio que hizo la llamada.
Si se quiere conseguir el efecto lateral de afectar a las variables del medio de llamada, hay que inducir el paso por **Referencia**, pasando direcciones de variables (**&**) en vez de sus valores. Entonces, se usan los parámetros formales desreferenciados (*****) dentro de la función.

| PASO POR VALOR | PASO POR REFERENCIA |
|---|---|
| `void doble(int i) {`<br>`  i *= 2;`<br>`}`<br><br>`void main() {`<br>`  int n=5;`<br>`  printf("%d\n", n);  // 5`<br>`  doble(n);`<br>`  printf("%d\n", n);  // 5`<br>`}` | `void doble(int *i){`<br>`  *i *=2;`<br>`}`<br><br>`void main() {`<br>`  int n=5;`<br>`  printf("%d\n", n); // 5`<br>`  doble(&n);`<br>`  printf("%d\n", n); //10`<br>`}` |

**Agrupamiento de los prototipos**

Es recomendable agrupar todos los prototipos de las funciones de un programa en uno o varios archivos de cabecera (**.h**) que luego se incluirán como encabezado en los programas que hagan referencia a las funciones allí declaradas.

**Visibilidad de las funciones**

Una función declarada en un archivo tiene un alcance global; es decir, puede ser invocada desde cualquier punto dentro este archivo. Cuando se quiere acceder a ella desde otro archivo, hay que incluir en este último la declaración (archivo de cabecera) de dicha función.

Si se quisiera restringir el uso de la función a un solo módulo, se usa el modificador **static** en su declaración.

## CADENAS

Una cadena de caracteres es un arreglo de caracteres terminado en el carácter nulo (\0).

Como ya hemos visto **gets()** y **puts**() pueden utilizarse para leer y escribir cadenas sin formato. O **scanf()** y **printf()** con el formato %s.

**Cadenas constantes**

Una cadena constante es una secuencia de caracteres encerrada entre comillas dobles.

Para inicializar una cadena constante:

```
char cad[ ] = ”Hola”;                    /* Ocupa 5 bytes */
```

equivale a:

```
char cad[ ] = {‘H’,‘o’,‘l’,‘a’,‘\0’};    /* Ocupa 5 bytes */
```

equivale a:

```
char *cad = “Hola”;                      /* Ocupa 5 bytes */
```

**Ingreso de cadenas**

Es necesario reservar espacio para una cadena, un modo de hacerlo es en la declaración de la variable.

Para `scanf(“%s”,cad); //o gets(cad);` lo correcto es declarar `char cad[80];` sería incorrecto declarar `char *cad;`

**Librerías**

- **stdio.h** : algunas funciones son **puts**(), **gets**(), **printf**(), **scanf**().
- **string.h** : algunas funciones son
  - ✓ **strlen**(*cad*) : devuelve la longitud de la cadena *cad* sin el \0.
  - ✓ **strcpy**(*cad1*, *cad2*) : copia el contenido de cad2 en cad1. Sería incorrecto cad1 = cad2. Devuelve un char *. Variante: **strncpy**(*cad1*, *cad2, n*).
  - ✓ **strcat**(*cad1*, *cad2*) : concatena el contenido de cad2 en cad1 (sin controlar que haya espacio). Devuelve un char *.
  - ✓ **strcmp**(*cad1*, *cad2*) : compara cad1 con cad2. Sería incorrecto. cad1 != cad2. Retorna -1, 0, 1. Variante: **strncmp**(*cad1*, *cad2, n*).

## ARREGLOS

**Declaración**

☞ Sintaxis: *tipo_dato* nombreVector[*componentes*];
*tipo_dato* nombreMatrizr[*filas*] [*columnas*];

☞ **Ejemplos de declaración de variables y tipos**

```
int vector[3];                 /* posición 0 a la 2 */
int matriz[2][3];              /* la primera posición es la [0][0] */

typedef int Tvector[10];       /* Tvector es un tipo */
typedef int Tmatriz[10][10];   /* Tmatriz es un tipo */
```

**Inicialización**

En la declaración, es posible asignarle un valor a los elementos.

☞ **Ejemplo de declaración con inicialización**

```
int vector[3] = {4, 5, 6};
int matriz[2][3] = { {1,2,3}, {4,5,6} };
```

El compilador puede deducir las dimensiones del arreglo:

```
int vector[] = {1, 2, 3, 5, 7};
```

Cuando se pasa un vector como parámetro, el paso de parámetros es por dirección (un vector es, en realidad, un puntero en C). Por tanto, se debe tener cuidado con los efectos colaterales que se producen si, dentro de un módulo, se modifica un vector que es parámetro.

☞ **Ejemplo de utilización**

```
#include <stdio.h>
void doble(int [], int );

void main() {
    int i, v[] = {1,2,3,4,5};

    doble(v,5);
    for(i=0; i<5; i++)
       printf("%d \t ",v[i]);
 }

void doble(int v[], int n){
  int i;
  for(i=0; i<n; i++)
    v[i]*=2;
}
```

Las cadenas, arreglos y matrices no pueden pasarse por valor

## ESTRUCTURAS

**Declaración**

☞ Sintaxis: **struct** *nombre_estructura* {
*tipo_variable nombre_variable1;*
*tipo_variable nombre_variable2;*
*tipo_variable nombre_variable3;* };

```
struct carta {
     int nro;
     char palo; };
```

Para definir variables de tipo **struct**:

```
struct carta {
     int nro;
     char palo; } c1, c2;

typedef struct {
         int nro;
         char palo; } carta;
carta c1, c2;
```

**Operación con estructuras y campos**

```
c1.palo='O';
c1.nro=2;
c2=c1;
```

**Punteros a estructuras**

```
typedef struct {
    char palo;
    int nro; } carta;

 void lee (carta *);
 void muestra (carta);

 int main() {
    carta c;

    lee(&c);
    muestra(c);
    return 0;
}

void lee (carta * c){
  scanf("%c %d ", &(c->palo), &((*c).nro));
}

void muestra (carta c){
  printf("%c  %d", c.palo, c.nro);
 }
```

## PUNTEROS y MEMORIA DINAMICA

Ya vimos que un puntero es un tipo de datos que almacena una dirección de memoria.
Existe una dirección especial que se representa por medio de la constante NULL (definida en <stdlib.h>) y se emplea cuando se quiere indicar que un puntero no apunta a ninguna dirección.

**Correspondencia entre punteros y vectores**

Al declarar un vector, se reserva memoria para almacenar elementos del mismo tipo. Y como ya hemos visto, el identificador es un puntero.

☞ **Ejemplo**

```
int v[3];
int *ptr;
...
ptr = v;                    // ptr = &v[0]
v[0] = 6;                   // *v = 6;
```

**Aritmética de punteros**

C permite sumar (o restar) valores enteros a punteros, desplazando la dirección tantos elementos como indique el desplazamiento.

Si se declara
*int *ptr*; entonces *ptr + x*, devuelve un puntero a la posición de memoria *sizeof(int)*x* bytes desde *ptr.*

Esto tiene sentido cuando el puntero es la dirección de un arreglo y se permite así el desplazamiento dentro del mismo. C no chequea que el valor del puntero sea válido (apunte a una posición válida dentro del arreglo).

☞ **Ejemplo**

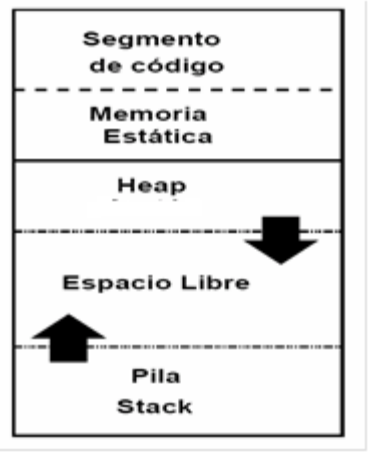
```
int v[10];
int *ptr = v;
          entonces ptr+i apunta a v[i] y *(ptr+i) ≅ v[i]
          ptr++ permite acceder a v[1]
          v++ es inválido, pues es la dirección constante del arreglo
```

Se pueden restar punteros, esto tiene sentido cuando ambos apuntan al mismo arreglo y permite obtener la diferencia entre dos posiciones.
No se pueden sumar punteros.

## GESTION DINAMICA de la MEMORIA

**Organización de la memoria**

- Segmento de código (código del programa).
- Memoria estática (variables globales y estáticas).
- Pila (stack): Variables automáticas (locales).
- Heap (montículo): Variables dinámicas




**Reserva y liberación de memoria**
Cuando se quiere utilizar el *heap*, primero hay que reservar la memoria que se desea ocupar.
Existen dos posibilidades: función **malloc** (ANSI C) y operador **new** (C++)

Tras utilizar la memoria reservada dinámicamente, hay que liberar el espacio reservado: función **free** (ANSI C) y operador **delete** (en C++)
Si no se libera la memoria, ese espacio de memoria no podrá volverse a utilizar.

☞ Sintaxis: *void **malloc** ( tamaño )*;
La función **malloc** (se encuentra en la librería **stdlib.h**) asigna una región de memoria para un objeto de datos de *tamaño* bytes, y devuelve la dirección del primer byte de esa región.
Los valores almacenados en el objeto de datos quedan indefinidos.
Se garantiza que la zona de memoria concedida no está ocupada por ninguna otra variable ni otra zona devuelta por **malloc**. Si no es posible conceder el bloque (p.ej. no hay memoria suficiente), devuelve un puntero NULL.
La creación del objeto y asignación de memoria se hace en tiempo de ejecución.

*Punteros void**
La función *malloc* devuelve un puntero inespecífico (no apunta a un tipo de datos determinado). En C, estos punteros sin tipo se declaran como *void**. Muchas funciones que devuelven direcciones de memoria utilizan los punteros *void**, éstos pueden convertirse a cualquier otra clase de puntero:

☞ **Ejemplo**

```
char *pch = (char *)malloc(1);
int *pint = (int *) malloc(sizeof(int));  o int *pint =(int *)malloc(2);
```

*Operador sizeof*
*malloc* necesita conocer cuántos bytes se quieren reservar. El tamaño en bytes de un elemento de tipo T se obtiene con la expresión *sizeof* (*T*).

☞ Sintaxis: *void* **free** *(puntero)*;
Se libera la asignación de memoria correspondiente al objeto de datos cuya dirección indica *puntero*. Si el puntero tiene valor nulo, no hace nada.

☞ **Ejemplo**

```
int * ptr;
ptr = (int*) malloc (sizeof(int)); /* Espacio para un entero */
……
free (ptr);

ptr = (int*) malloc (30*sizeof(int)); /* Espacio para 30 enteros */
……
free (ptr);
```

Analizar el siguiente código y los posibles errores y efectos no deseados:

```
typedef struct {
   char palo;
   int nro; }carta;

void main() {
   int * Pt1, * Pt2, X=10;
   carta * PtCarta;
   char * cad;

Pt1 = (int *) malloc(sizeof(int));
scanf("%d",Pt1);
Pt2 = &X;                      //No es manejo de memoria dinámica
printf("\n%d  %d", X, *Pt2);
*Pt2= *Pt1 + 4;
printf("\n%d  %d",*Pt1, *Pt2);
Pt2 = Pt1;
printf("\n%d  %d\n",*Pt1, *Pt2);
free(Pt2); free(Pt1);

PtCarta =(carta *)malloc (sizeof(carta));
scanf("%d %c",&(PtCarta->nro), &((*PtCarta).palo));
printf("\n%d  %c\n",(*PtCarta).nro, PtCarta->palo);

cad = (char *)malloc(30);
scanf("%s",cad);
printf("\n%c  %c",cad[0], cad[1]);

free(cad);  free(PtCarta);  // cad=NULL?
}
```

## ESTRUCTURAS DINÁMICAS

Las estructuras dinámicas se construyen con punteros. Un componente del objeto es un puntero que "apunta" a otro objeto del mismo tipo.
Se consigue tener un conjunto de elementos relacionados que puede aumentar y disminuir en tiempo de ejecución. Las estructuras dinámicas se manejan con definiciones de datos recursivas.

☞ **Ejemplo**

```
typedef struct nodo {
    char dato;
    struct nodo *sig;} lista;

typedef lista * Tlista;
```

## ARCHIVOS

Los archivos, en contraposición con las estructuras de datos vistas hasta ahora (variables simples, vectores, registros, etc.), son estructuras de datos almacenadas en memoria secundaria.
El formato de declaración de un archivo es:

FILE * *nombre_archivo*;

En otros lenguajes la declaración del archivo determina (es determinada por) el tipo de datos que se van a almacenar en él. En C la filosofía es distinta, todos los archivos almacenan bytes y es cuando se realiza la apertura y la escritura cuando se decide cómo y qué se almacena en el mismo; durante la declaración del archivo no se hace ninguna distinción sobre el tipo del mismo.

En la operación de apertura se puede decidir si el archivo va a ser de texto o binario, los primeros sirven para almacenar caracteres, los segundos para almacenar cualquier tipo de dato.
Si deseamos leer un archivo como DATOS.txt utilizaremos uno de texto, si queremos leer y escribir registros (struct) usaremos uno binario.

**Apertura y Cierre**

Antes de usar un archivo es necesario realizar una **operación de apertura** del mismo; posteriormente, si se desea almacenar datos en él hay que realizar una **operación de escritura** y si se quiere obtener datos de él es necesario hacer una **operación de lectura**. Cuando ya no se quiera utilizar el archivo se realiza una **operación de cierre** del mismo para liberar parte de la memoria principal que pueda estar ocupando (aunque el archivo en sí está almacenado en memoria secundaria, mientras está abierto ocupa también memoria principal). Las funciones de archivos se encuentran en *stdio.h*

La instrucción para abrir un archivo es ***FILE ** archivo es:*

*archivo* = ***fopen*** *(nombre-archivo, modo);*

La función ***fopen*** devuelve un puntero a un archivo que se asigna a una variable de tipo FILE *. Si existe algún **tipo de error** al realizar la operación, por ejemplo, porque se desee abrir para leerlo y éste no exista, devuelve el valor **NULL.**
El *nombre-archivo* será una cadena de caracteres que contenga el nombre (y en su caso la ruta de acceso) del archivo tal y como aparece para el sistema operativo.
El *modo* es una cadena de caracteres que indica el tipo del archivo (texto o binario) y el uso que se va a hacer de él lectura, escritura, añadir datos al final, etc.

Los modos disponibles son:

| Modo | Significado |
|---|---|
| "r" | abre un archivo para lectura. Si el archivo no existe devuelve error |
| "w" | abre un archivo para escritura. Si el archivo no existe se crea, y si existe se destruye y se crea uno nuevo |
| "a" | abre un archivo para añadir datos al final del mismo. Si no existe se crea. |
| + | símbolo utilizado para abrir el fichero para lectura y escritura |
| b | Indica que el archivo es de tipo binario |
| t | Indica que el archivo es de tipo texto (por defecto) |

Los modos anteriores se combinan para conseguir abrir el archivo en el modo adecuado.
La forma habitual de utilizar la instrucción **fopen** es dentro de una sentencia condicional que permita conocer si se ha producido o no error en la apertura.
Declarando `FILE *arch;` es posible abrir el archivo de algunas de las siguientes formas:

```
arch=fopen ("datos.dat", "rb+"); /* archivo binario ya existente para L/E */
arch=fopen ("datos.txt", "w+");  /* archivo nuevo, si existe se borra su
                                    contenido, para L/E */
arch=fopen ("datos.txt", "a");   /* agregar al final, debe existir */
if ((arch=fopen("nomarch.dat", "r")) == NULL)
   printf("Error en la apertura. Es posible que el archivo no exista");
```

Cuando se termine el tratamiento del archivo hay que cerrarlo: **fclose**(*nombre_archivo*);

**Lectura y Escritura**

Para almacenar datos en un archivo es necesario realizar una operación de escritura, de igual forma que para obtener datos hay que efectuar una operación de lectura. En C existen muchas y variadas operaciones para leer y escribir en un archivo; entre ellas: **fread - fwrite**, **fgetc - fputc**, **fgets - fputs**, **fscanf – fprintf**.

Para leer todos los datos de un archivo basta con realizar lecturas sucesivas hasta que se lee el **final del fichero**. La función ***feof*** *(archivo)* devuelve un valor distinto de 0 cuando se ha alcanzado el final del archivo.

**Acceso Directo a los datos**

Cuando se puede acceder a cualquier dato de un archivo sin tener que pasar por anteriores se está realizando un acceso directo a los datos.
La función que permite situarse en un determinado dato del archivo es **fseek** (*archivo*, *posicion*, *origen*); que coloca el puntero del archivo a tantos bytes del origen como indica posición (positivo o negativo) contando a partir del origen señalado.
Valores posibles para *origen*:

SEEK_SET o 0 : principio del archivo.
SEEK_CUR o 1 : posición actual.
SEEK_END o 2 : final del archivo.

En un archivo *f* que almacene números enteros se podría:

```
fseek(f,0,SEEK-SET); /* coloca el puntero al principio de f */
```

o

```
fseek(f,3*sizeof(int),SEEK-CUR); /* coloca el puntero 3 posiciones más
                                    allá de la posición actual del puntero */
```

Para saber cuál es la posición en la que está el puntero del archivo, C proporciona la función: **ftell** (*archivo*); que devuelve la posición actual en bytes del puntero del archivo con respecto al principio del mismo. Esta función se utiliza sobre archivos binarios (ANSI C). La sentencia `ultimo=ftell(f);`, asigna a ultimo la cantidad de bytes de *f*

**EJERCICIO**

Dado un archivo de texto que contiene datos de clientes de un comercio, en cada línea: nombre, sexo (carácter) y edad.
Obtener la edad promedio de los hombres y guardar sus datos en un archivo de registros

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
FILE * archt, *archb;
int cant, suma, edad;
char sexo, nom[12];
struct dato{
  char nombre[12];
  int edad;} reg;

 archt = fopen("Clientes.txt","rt");
 archb = fopen("Hombres.dat","wb");
 if (archt == NULL) {
    printf("archivo clientes no existe!");
    return 1;
 }
 cant =suma = 0;
 fscanf(archt,"%s %c %d", nom,&sexo, &edad);
 while(!feof(archt)) {
    if (sexo == 'M') {
      cant++;
      suma+=edad;
      strcpy(reg.nombre, nom);
      reg.edad=edad;
      fwrite(&reg, sizeof(struct dato), 1, archb);
    }
   fscanf(archt,"%s %c %d", nom,&sexo, &edad);
}
 if (cant)
     printf("Promedio %i", suma/cant);
 fclose(archt);
 fclose(archb);

 archb = fopen("Hombres.dat","rb");
 if (archb == NULL) {
    printf("archivo clientes no existe!");
    return 1;
 }
 fread(&reg, sizeof(reg), 1, archb);
 while(!feof(archb)) {
      printf("%s %d", reg.nombre, reg.edad);
      fread(&reg, sizeof(reg), 1, archb);
    }
 fclose(archb);
 return 0;
}
```

# PROGRAMACION II

## MATERIAL para TEORIA

## TDA - Ejemplo

**Código Asignatura 6A4**

**Año 2016**

**Segundo Cuatrimestre**

**INTERFACE del TDA COMPLEJO (complejo.h)**

```
typedef struct{
    float r, i;} complex;

void suma (complex c1, complex c2, complex *c3);
void ingresa (complex *c);
void muestra (complex c);
float creal (complex c);
float cimag (complex c);
```

**DESARROLLO del TDA COMPLEJO (complejo.c)**

```
#include "complejo.h"
#include <stdio.h>

void suma (complex c1, complex c2, complex *c3){
  c3->r=c1.r+c2.r;
  c3->i=c1.i+c2.i;
}

void ingresa (complex *c){
printf("Ingrese las componente de un complejo");
scanf("%f %f",&(c->r), &(c->i));
}

void muestra (complex c){
    printf("%5.2f",c.r );
    (c.i > 0) ? printf("+%5.2f i",c.i ): printf("%5.2f i",c.i );
}

float creal (complex c){
 return c.r;
}

float cimag (complex c){
 return c.i;
}
```

**UTILIZACION del TDA COMPLEJO (main.c)**

```
#include <stdio.h>
#include <stdlib.h>
#include "complejo.h"

int main()
{
    complex c1, c2, c3;
    ingresa(&c1);
    ingresa(&c2);
    suma(c1,c2,&c3);
    muestra(c1);
    muestra(c2);
    printf("%5.2f",creal(c3));
    (cimag(c3) > 0) ? printf("+%5.2f i",cimag(c3)): printf("%5.2f
i",cimag(c3));
    return 0;
}
```

# PROGRAMACION II

## MATERIAL para TEORIA

## PILAS

**Código Asignatura 6A4**

**Año 2018**

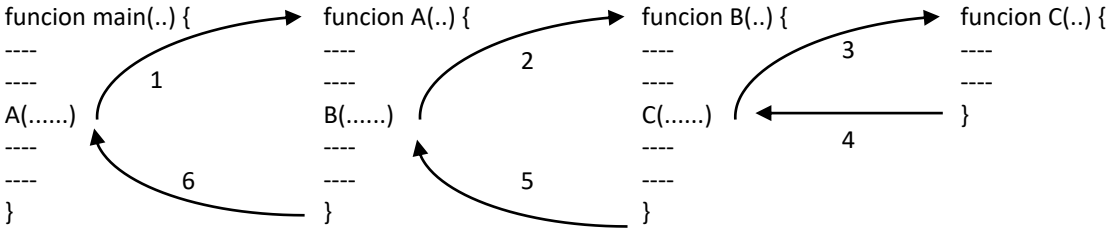
# TIPO de DATOS PILA

Una pila o stack es una estructura de datos que consiste en una colección de elementos, todos del mismo tipo, en la cual las operaciones de inserción y eliminación se realizan por un extremo denominado tope (o cima). Sólo el elemento que está en el tope de la pila puede ser consultado o removido y si se agrega un elemento, éste se incorpora sobre el tope.

Un ejemplo típico es "una pila de platos", en la que ponemos sobre el último apilado y sacamos el que está en el tope (no se incorpora ni se elimina por la base o el medio). También se conoce esta estructura con el nombre de LIFO (Last In First Out), de este modo podemos decir que una pila *invierte* el orden de los elementos que en ella se almacenan. A pesar de su sencillez, es muy adecuada para la información que se comporta de este modo en el almacenamiento y recuperación. Es el caso, por ejemplo de las invocaciones a subprogramas.

Cada subprograma almacena (apila) en el segmento de memoria correspondiente, parámetros, variables locales, temporales y direcciones de retorno.

Por ejemplo, cuando desde un subprograma A se invoca a otro subprograma B, los elementos antes mencionados de B se apilan sobre los de A, quedando en el tope lo concerniente al subprograma activo. Cuando B finaliza, su información se desapila y queda en el tope la información de A, que retoma la ejecución (suspendida por el llamado a B)




## TDA PILA

Para definir un Tipo de Dato Abstracto Pila, es necesario definir el tipo y un conjunto de operadores que permitan declarar y utilizar variables del tipo. Para desarrollar algoritmos que almacenen y recuperen elementos de una pila, es necesario conocer las cabeceras de los operadores (interface), pero no los detalles de su implementación (caja negra)

### OPERADORES del TDA PILA

Siendo P : pila de objetos de tipo "TElementoP" y x : objeto de tipo "TElementoP"

- INICIAP( P ) → Devuelve en P una pila vacía
- VACIAP( P ) → Devuelve Verdadero si la pila P está vacía, y Falso en caso contrario
- CONSULTAP( P ) → Devuelve el valor del elemento que se encuentra en el tope de la pila P
- SACAP( P , x ) → Devuelve en x el elemento que se encuentra en el tope de la pila removiéndolo de P
- PONEP( P , x ) → Inserta el elemento x en el tope de la pila.

### INTERFACE del TDA PILA

void **IniciaP** (TPila * P)

void **poneP** (TPila * P, TElementoP x)

void **sacaP** (TPila * P, TElementoP * x)

TElementoP **consultaP**(TPila P)

int **VaciaP** (TPila P)

## UTILIZACION del TDA PILA

Problema: Reescribir un número decimal en su equivalente binario.

```
#include<stdio.h>
#include<pilas.h>

 int main()  {
   TPila P;
   TElementoP bit;
   int num, numero;

    IniciaP(&P);
    scanf("%d",&numero);
    num=numero;
    while (numero != 0) {
        poneP(&P, numero % 2);
        numero /= 2;
    }
    printf("Representacion de %d en base 2 \n", num);
    while (!VaciaP(P)) {
        sacaP(&P,&bit);
        printf("%d ",bit);
    }
    return 0;
}
```

## IMPLEMENTACION del TDA PILA

### ⇒ IMPLEMENTACION ESTATICA

```
typedef int TElementoP;
typedef struct {
 TElementoP datos[50];
 int tope; } TPila;
void poneP(TPila *P, TElementoP x) {
   if ( ((*P).tope)!=49)
       (*P).datos[++((*P).tope)] = x;
}
void sacaP(TPila *P, TElementoP* x)  {
  if (((*P).tope) != -1)
        (*x) = (*P).datos[((*P).tope)--];
}
TElementoP consultaP(TPila P)  {
  if ((P.tope) != -1)
        return P.datos[P.tope];
}
int VaciaP(TPila P)  {
  return (P.tope == -1);
}
void IniciaP (TPila *P) {
 (*P).tope=-1;
}
```

## ⇒ IMPLEMENTACION DINAMICA

```
typedef int TElementoP;
typedef struct  {
 TElementoP dato;
 struct nodop *sig; } nodop;
typedef nodop *TPila;
void poneP(TPila *P, TElementoP x) {
  TPila N;
  N = (TPila)malloc(sizeof(nodop));
  N->dato = x;
  N->sig = *P;
  *P=N;
}
void sacaP(TPila *P, TElementoP * x) {
  TPila N;
   if (*P) {       // if (*P != NULL)
      N = *P;
      *x = (*P)->dato;
      *P = (*P)->sig;
      free(N);
  }
}
TElementoP consultaP(TPila P)  {
if (P)                  // if (P != NULL)
   return P->dato;
}
int VaciaP(TPila P)  {
 return (P == NULL);
}
void IniciaP(TPila *P) {
 *P =NULL;
}
```

# PROGRAMACION II

## MATERIAL para TEORIA

## COLAS

**Código Asignatura 6A4**

**Año 2018**

# TIPO de DATOS COLA

Una cola es una estructura de datos que consiste en una colección de elementos, todos del mismo tipo, en la cual los datos se incorporan por un extremo y se eliminan por otro. No se permite acceder a elementos intermedios ni recorrerla.

Un ejemplo muy común es la cola que se forma en la caja de un negocio o la ventanilla de un banco. Recibe también el nombre de FIFO (First In First Out), ya que el elemento que se almacenó primero es el primero en removerse, de este modo podemos decir que una cola *preserva* el orden de los elementos que en ella se almacenan. A pesar de su sencillez, es muy adecuada para la información que se comporta de este modo en el almacenamiento y recuperación. Es el caso, por ejemplo de la simulación de procesos de atención (bancos, supermercados, semáforos) y de impresión, entre otros.

## TDA COLA

Para definir un Tipo de Dato Abstracto Cola, es necesario definir el tipo y un conjunto de operadores que permitan declarar y utilizar variables del tipo. Para desarrollar algoritmos que almacenen y recuperen elementos de una Cola, es necesario conocer las cabeceras de los operadores (interface), pero no los detalles de su implementación (caja negra)

### OPERADORES del TDA COLA

Siendo C : cola de objetos de tipo "TElementoC" y x : objeto de tipo "TElementoC"

- INICIAC( C ) → Devuelve en C una cola vacía
- VACIAC( C ) → Devuelve Verdadero si la cola C está vacía, y Falso en caso contrario
- CONSULTAC( C ) → Devuelve el valor del elemento que se encuentra en el primer lugar de la cola C
- SACAC( C , x ) → Devuelve en x el elemento que se encuentra en el primer lugar de la cola removiéndolo de C
- PONEC( C , x ) → Inserta el elemento x en el último lugar de la cola.

### INTERFACE del TDA COLA

void **IniciaC** (TCola * C)

int **VaciaC** (TCola C)

TElementoC **consultaC** (TCola C)

void **sacaC** (TCola *C, TElementoC* x)

void **poneC** (TCola *C, TElementoC x)

**UTILIZACION del TDA COLA**

Problema:

Leer un conjunto de números y mostrar los que son impares y mayores que el promedio, en el orden en que ingresaron.

```
#include <stdio.h>
#include <colas.h>

int main()  {
    TCola Cnros;
    TElementoC num;
    int sum = 0, cuenta = 0;
    float prom;

    IniciaC (&Cnros);
    scanf("%d",&num);
    while (num) {
        sum += num;
        cuenta++;
        if (num % 2)
           poneC (&Cnros, num);
        scanf("%d",&num);
    }
   if (cuenta) {
       prom = sum/cuenta;
       printf("Los numeros impares mayores al promedio son ”);
       while (!VaciaC(Cnros)) {
           sacaC (&Cnros, &num);
           if (num > prom)
               printf("%d  ", num);
      }
   else
    printf("No se registraron numeros impares mayores al promedio”);
 }
 return 0;
}
```

## IMPLEMENTACION del TDA COLA

### ⇒ IMPLEMENTACION ESTATICA

```
typedef int TElementoC;
typedef struct {
 TElementoC datos[50];
    int pri, ult; } TCola;
void IniciaC (TCola *C) {
 (*C).pri=-1;
 (*C).ult=-1;
}
int VaciaC(TCola C){
  return C.pri==-1;
}
void poneC (TCola *C, TElementoC X) {
   if ((*C).ult != 49) {
     if ((*C).pri==-1)
       (*C).pri = 0;
    (*C).datos[++((*C).ult)]=X;
  }
}
void sacaC (TCola *C, TElementoC *X) {
  if ((*C).pri != -1) {      // !vaciaC(*C)
     *dato = (*C).datos[(*C).pri];
    if ((*C).pri == (*C).ult)
       iniciaC(C);
    else
       (*C).pri +=1;
  }
}
TElementoC consultaC (TCola C){
if (C.pri !=-1)
   return C.dato[C.pri];
}
```

## ⇒ IMPLEMENTACION DINAMICA

```
typedef int TElementoC;
typedef struct nodo {
  TElementoC dato;
  struct nodo * sig;} nodo;
typedef struct {
 nodo *pri, *ult;} TCola;
void IniciaC (TCola *C){
 (*C).pri=NULL;
 (*C).ult=NULL;
}
int VaciaC(TCola C){
  return C.pri==NULL;
}
void poneC (TCola *C, TElementoC X) {
   nodo * aux;
   aux = (nodo *) malloc (sizeof(nodo));
   aux->dato = X;
   aux->sig = NULL;
   if ((*C).pri==NULL)
      (*C).pri=aux;
   else
      (*C).ult->sig=aux;
  (*C).ult=aux;
}
void sacaC (TCola *C, TElementoC *X){
  nodo * aux;
  if ((*C).pri !=NULL) {
     aux = (*C).pri;
   *X = aux->dato;
   (*C).pri = (*C).pri->sig;
   If ((*C).pri == NULL)
      (*C).ult = NULL;
   free(aux);
   }
}
TElementoC consultaC (TCola C){
if (C.pri !=NULL)
   return C.pri-> dato;
}
```

## ⇒ IMPLEMENTACION ESTATICA CIRCULAR

```
void poneC (TCola *C, TElementoC X) {
   if  (!(*C).ult==49 && (*C).pri==0 || (*C).ult+1==(*C).pri) {
     if ((*C).pri==-1){
        (*C).pri = 0;
        (*C).ult = 0;
    }
    else
      if (*C).ult == 49)
          (*C).ult = 0;
      else
           (*C).ult += 1;
    (*C).datos[(*C).ult]=X;
  }
}
```

```
void sacaC (TCola *C, TElementoC *X) {
  if ((*C).pri != -1) {
     *X = (*C).datos[(*C).pri];
    if ((*C).pri == (*C).ult)
       iniciaC(C);
    else
      if (*C).pri == 49)
         (*C).pri = 0;
      else
         (*C).pri += 1;
  }
}
```

**Problemas para resolver**

1) En una Cola se han almacenado números enteros positivos, dejar en la cola sólo los pares.

```
void Pares (TCola *C) {
 int num;

    poneC(C, -1);
    sacaC(C, &num);
    while (!vaciaC(*C)){
       if (num % 2 == 0)
          poneC(C,num);
      sacaC(C,&num);
    }
  }
```

2) En una Pila se han almacenado secuencias de al menos un número entero finalizadas en 0, las que sean ascendentes deberán almacenarse en una cola en orden inverso al que tenían.
Por ejemplo:
Si Pila : 1 3 7 0 5 3 5 0 1 4 8 10 0 3 6 1 0 entonces la Cola será : 7 3 1 10 8 4 1

```
void Secu (TPIla *P, TCola *C) {
 TPila PAux;
 int ok, num;

    iniciaC(C);
    iniciaP(&PAux);
    while (!vaciaP(*P)){
       sacaP(P,&num);
       ok=1;
       poneP(&PAux,num);
       while (consultaP(*P)!=0 && ok){
          sacaP(P,&num);
          if (num >= consultaP(AuxP))
            poneP(&PAux,num);
          else
            ok=0;
       }
       if (ok)
         while (!vaciaP(PAux)) {
           sacaP(&PAux, &num);
           poneC(C,num);
         }
       else {
         iniciaP(&PAux);
         while (consultaP(*P)!= 0)
          sacaP(P,&num);
       }
       sacaP(P,&num);
    }
  }
```

3) *Resolver recursivamente*. En una Pila se han almacenado valores enteros, se desea saber cuántos (excepto los dos primeros) son iguales al promedio de los dos anteriores, la pila puede perderse.
Por ejemplo:
Pila : 1 3 **2** 0 **1** 5 **3** 5 Rta: 3

```
void Promedio (TPIla *P, int *cont) { //inicialmente vale 0
int num1, num2;

     if (!vaciaP(*P)){
       sacaP(P,&num1);
       if (!vaciaP(*P)){
          sacaP(P,&num2);
          if (!vaciaP(*P)){
             if (consultaP(*P)=(num1+num2)/2)
               (*cont)++;
             poneP(P,num2);
             Promedio(P,cont);
          }
        }
      }
}
```

4) *Resolver recursivamente* En una Pila se han almacenado valores enteros, se desea saber si están ordenados de modo ascendente. Desarrollar la solución de forma recursiva. No perder la pila.
Por ejemplo:
Pila : 1 3 7 10 Rta: OK

```
void Asc (TPila *P, int *ascOK) {
int num;

     if (!(vaciaP(*P))){ //solo si inicialmente esta vacía
       sacaP(P,&num);
       if (!(vaciaP(*P)))
          if (num <= consultaP(*P))
             Asc(P,ascOK);
          else
             *ascOK=0;
       else
         *ascOK=1;
       poneP(P,num);
     }
     else
       *ascOK=1;
   }
```

# PROGRAMACION II

## MATERIAL para TEORIA

## LISTAS

**Código Asignatura 6A4**

**Año 2017**

**Segundo Cuatrimestre**

# LISTAS SIMPLEMENTE ENLAZADAS

A partir de “nodos” dinámicos, implementados como variables de tipo registro, donde un campo de tipo puntero a registro contiene la dirección del próximo nodo, es posible implementar Pilas, Colas y Listas tomando como enlaces los vínculos en memoria dinámica.

Cada inserción implica crear y vincular nodos y cada eliminación desvincularlos devolviendo al heap los bytes que ocupa. La forma en que se modifican los enlaces o vínculos depende de la forma de operar de la estructura. No se considera la posibilidad de estructura llena, ya que ésta crece y decrece de acuerdo a los requerimientos.

**TIPO LISTA SIMPLEMENTE ENLAZADA**

```
typedef struct nodo{
  char dato[15];
  struct nodo * sig;} nodo;
typedef struct nodo * TLista;
```

**EJERCICIOS**

**1)** Dada una lista simplemente enlazada de cadenas, retornar la cantidad que tienen longitud par
**2)** Dada una lista simplemente enlazada de cadenas, verificar si X está
**3)** Dada una lista ordenada simplemente enlazada de enteros, verificar si X está
**4)** Dada una lista de enteros modificar cada aparición de X por X+1
**5)** Insertar un dato en una lista ordenada simplemente enlazada de enteros
**6)** Dada una lista ordenada simplemente enlazada de enteros, eliminar X

**TAREAS**

**7)** Destruir una lista simplemente enlazada
**8)** Eliminar todas las apariciones de X de una lista simplemente enlazada de enteros
**9)** Eliminar todas las apariciones de X de una lista simplemente enlazada ordenada de enteros

## LISTAS CON SUBLISTAS

**1)** Se tiene una lista L con la siguiente información:

- Codigo de Contenedor (no se repite)
- Destino
- Peso (en toneladas)

Además, una lista de barcos en puerto esperando embarcar los contenedores que están en la lista:

- Codigo de Barco (no se repite)
- Destino (puede repetirse)
- Capacidad
- Sublista de Contenedores
  - Codigo de Contenedor
  - Peso

Se pide: procesar la información de L y distribuir los contenedores en los barcos. Si alguno no pudiera ser embarcado por falta de capacidad en el destino o por falta de barcos a ese destino, quedará en L, de forma tal que al terminar el proceso queden en la misma los contenedores que no pudieron ser despachados.

**TIPO DE DATOS**

```
typedef struct nodo {
  char codC[15], dest [15];
  float peso;
  struct nodo * sig;} nodo;
typedef struct nodo * TListaC;
```

```
typedef struct nodito {
  char codC[15];
  float peso;
  struct nodito * sig;} nodito;
typedef struct nodito * SubLista;

typedef struct nodoB {
  char codB[15], dest[15];
  float capac;
  struct nodoB * sig;
  SubLista sub;} nodoB;
typedef struct nodoB * TListaB;
```

**2)** Se tiene una lista de clientes que registran pagos de un crédito con el siguiente diseño:

- Numero de Cliente (no se repite, ordenado ascendente)
- Total Credito, Total Adeudado (en $)
- Sublista de Pagos
  - Fecha (Ordenada descendente, no se repite)
  - Importe

Se pide:

**a.-** Dado un número de cliente *correcto*, una fecha y un importe, insertar el pago actualizando el valor adeudado.

**b.-** Dado un número de cliente y una fecha, eliminar el pago (si existe) actualizando el valor adeudado.

**c.**- Dado un número de cliente, eliminarlo de la lista

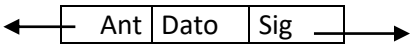
**d.-** Eliminar de la lista los clientes que ya no tienen deuda.

**TIPO DE DATOS**

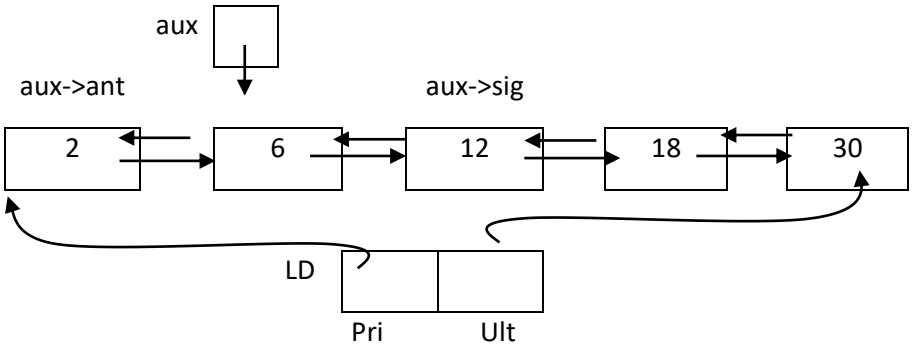
```
typedef struct nodito {
   char fecha[9];
   float imp;
   struct nodito * sig;} nodito;
 typedef struct nodito * SubLista;
typedef struct nodoC {
   int numC;
   float cred, deuda;
   struct nodoC * sig;
   SubLista sub;} nodoC;
typedef struct nodoC * TListaC;
```

# LISTAS DOBLEMENTE ENLAZADAS

En esta estructura cada nodo almacena la dirección del nodo siguiente y del nodo anterior. Esto permite recorrer la lista en ambos sentidos.




Se debe mantener la dirección del primer nodo y del último para acceder a la lista, por el principio y por el final.




**Ejercicio** Se tiene una lista simplemente enlazada que almacena números enteros sin repeticiones. Armar una lista doblemente enlazada ordenada de forma ascendente.
A partir de la lista generada, eliminar nodos de tal modo que en la misma no aparezcan dos valores pares o impares consecutivos.

Ejemplo: 10 7 13 8 12 5 → 5 7 8 10 12 13 → 5 8 13

**TIPO DE DATOS**

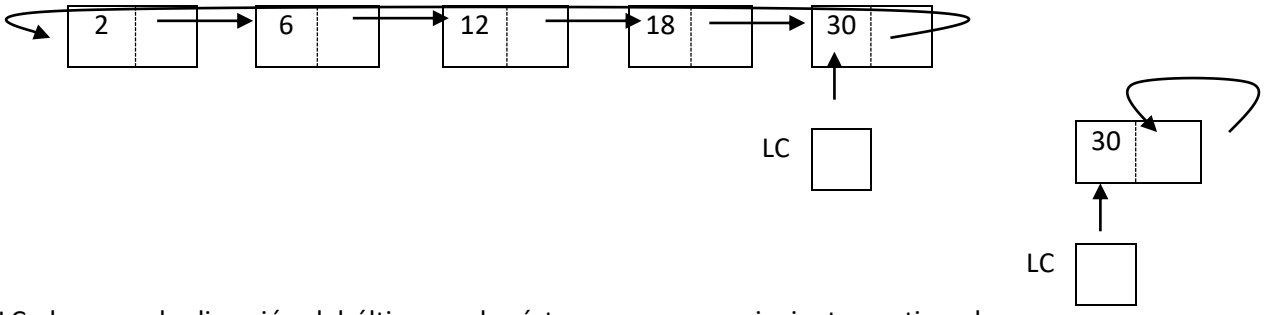
```
typedef struct nodo {
int num;
struct nodo * sig;} nodo;
typedef struct nodo * TLista;

typedef struct nodoD {
  int num;
  struct nodoD *ant, * sig;} nodoD;
typedef struct nodoD * PnodoD;

typedef struct {
    PnodoD pri, ult;} TListaD;
```

# LISTAS CIRCULARES

En esta estructura, el ultimo nodo está vinculado con el primero, permitiendo acceder a cualquier nodo independientemente de dónde se parta.
Ejemplo de dos listas con 5 y 1 nodos respectivamente.




LC almacena la dirección del último nodo, éste en su campo siguiente contiene la dirección del primer nodo de la lista ordenada.

Mantener la dirección del último, en lugar del primero, permite insertar al final de la lista sin tener que recorrerla. Para insertar al comienzo se recurre al último nodo y se modifica su campo siguiente con la dirección del nodo nuevo.

**1)** Mostrar el contenido de una lista circular

**TIPO DE DATOS**

```
typedef struct nodo{
     int dato;
     struct nodo * sig;} nodo;
typedef struct nodo * TListaC;
```

**Ej 2.-** Se tiene una lista circular que almacena palabras (ordenada por este criterio) y cantidad de apariciones de cada una de ellas.

a) Insertar una palabra en la lista.
b) Eliminar una aparición de una palabra

**TIPO DE DATOS**

```
typedef struct nodo {
    int cant;
    char palabra [15];
    struct nodo * sig;} nodo;
typedef struct nodo * TListaC;
```


# PROGRAMACION II

## MATERIAL para TEORIA

## Arboles

**Código Asignatura 6A4**

**Año 2018**

## ARBOLES BINARIOS DE BUSQUEDA

**INSERCION EN ABB**

```
void Inserta (arbol *A, Tdato X) {
 If (*A == null) {
   *A = (arbol) malloc (sizeof (struct nodo));
    (*A)->Dato = X;
    (*A)->Der = null;
    (*A)->Izq = null;
 }
  else
   If  (X>(*A)->Dato)
            Inserta(&((*A)->Der), X);
   else
            Inserta(&((*A)->Izq), X);
}
```

**ELIMINACION EN ABB**

```
void Borrar (arbol * p, arbol aux) {
 If ((*P)->Der != null)
    Borrar(&((*p)->Der),aux)
 else {
       aux->Dato = (*p)->Dato;
       aux = *p;
       (*p) = (*p)->Izq;
       free(aux);
   }
 };

void Elimina (arbol *A, Tdato X) {
arbol aux;
 If (*A != null)
   If (X < (*A)->Dato)
            Elimina (&((*A)->Izq),X);
   else
            If  (X > (*A)->Dato)
               Elimina (&((*A)->Der),X);
            else {
               aux = *A;
               If (aux->Der == null) {
                        (*A) = aux->Izq;
                        free(aux);
               }
               else
                        If (aux->Izq == null) {
                             (*A) = aux->Der;
                           free(aux);
                        }
                        else
                           Borrar(&(aux->Izq), aux);
            }
    };
```

## ÁRBOLES BALANCEADOS POR SU ALTURA (AVL)

La altura H de un árbol binario T se define como:

- 0 si T contiene solo la raíz
- 1 + max($H(T_{izq})$, $H(T_{der})$) en otro caso

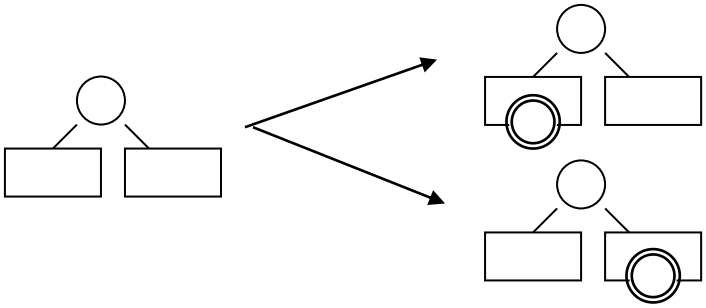
Un árbol AVL (**A**delson-**V**elskii y **L**andis) es un ABB en el que para todo nodo del árbol, la diferencia entre la altura de sus subárboles izquierdo y derecho es a lo sumo 1.

**Situaciones**

Al insertarse o eliminarse un nodo en un árbol AVL, deben diferenciarse los siguientes casos:

✓ Los subárboles izquierdo y derecho tienen la misma altura, por lo que luego de la inserción / eliminación, el árbol se mantiene balanceado:

- Inserción



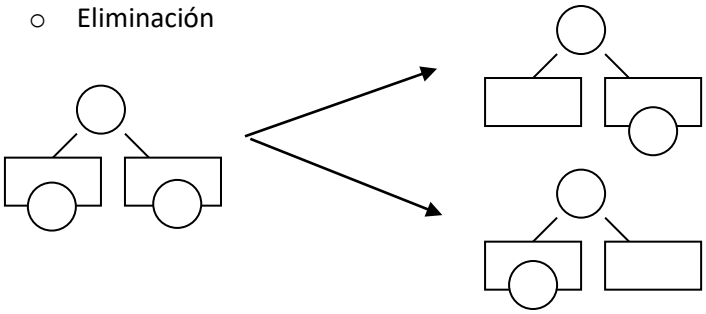
Aumenta la altura del subárbol izquierdo → el árbol se mantiene balanceado

[altura (IZQ) = altura (DER)+1]

Aumenta la altura del subárbol derecho → el árbol se mantiene balanceado

[altura (IZQ)+1 = altura (DER)]

- Eliminación



Disminuye la altura del subárbol izquierdo → el árbol se mantiene balanceado

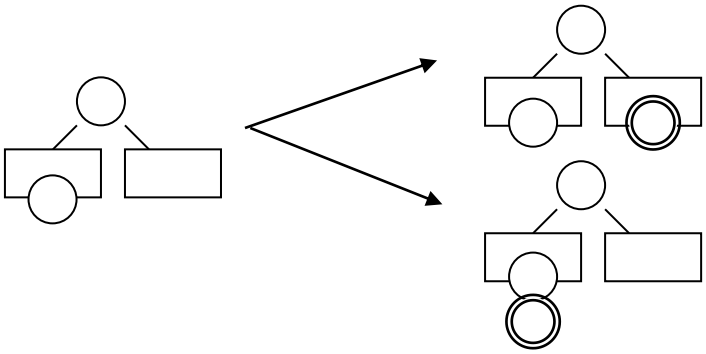
[altura (IZQ) + 1 = altura (DER)]

Disminuye la altura del subárbol derecho → el árbol se mantiene balanceado

[altura (IZQ) = altura (DER) + 1]

✓ La altura del subárbol izquierdo es mayor a la altura del subárbol derecho:

- Inserción:



Aumenta la altura del subárbol derecho → el árbol se mantiene balanceado

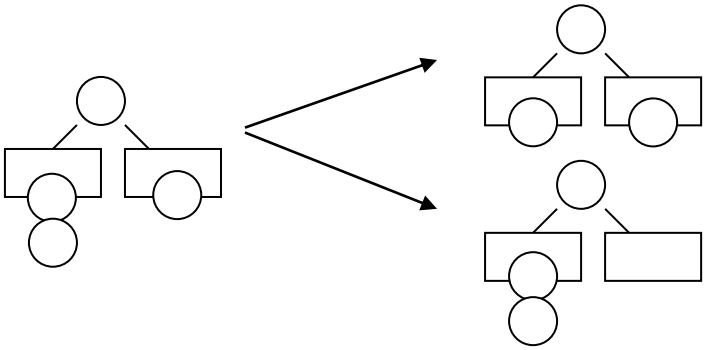
[altura (IZQ) = altura (DER)]

Aumenta la altura del subárbol izquierdo → el árbol se desbalancea

[altura (IZQ) = altura (DER)+2]

**Rebalancear el árbol**

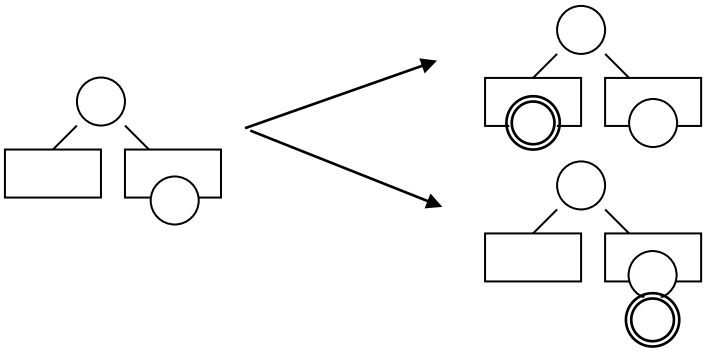
- Eliminación:



Disminuye la altura del subárbol izquierdo → el árbol se mantiene balanceado
[altura (IZQ) = altura (DER)]

Disminuye la altura del subárbol derecho → el árbol se desbalancea
[altura (IZQ) = altura (DER)+2]
**Rebalancear el árbol**

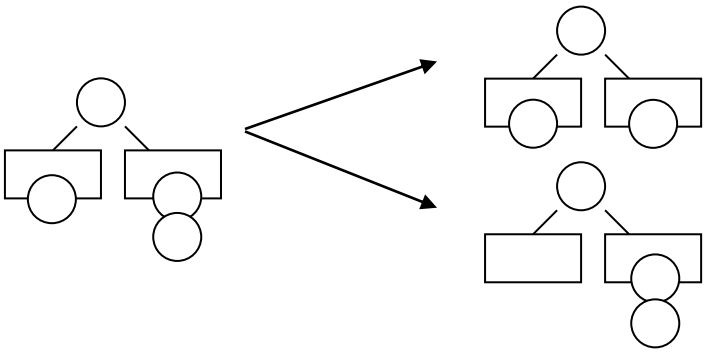
✓ La altura del subárbol derecho es mayor a la altura del subárbol izquierdo:
- Inserción:



Aumenta la altura del subárbol izquierdo → el árbol se mantiene balanceado
[altura (IZQ) = altura (DER)]

Aumenta la altura del subárbol derecho → el árbol se desbalancea
[altura (IZQ)+2 = altura (DER)]
**Rebalancear el árbol**

- Eliminación:



Disminuye la altura del subárbol derecho → el árbol se mantiene balanceado
[altura (IZQ) = altura (DER)]

Disminuye la altura del subárbol izquierdo → el árbol se desbalancea
[altura (IZQ)+2 = altura (DER)]
**Rebalancear el árbol**

**Factor de Equilibrio**

Es la diferencia de altura entre los subárboles izquierdo y derecho:

FE = |altura(IZQ) – altura(DER)| ➔ si el árbol está balanceado → $-1 \leq FE \leq 1$

**Rotaciones**

- ROTACION
  - SIMPLE (Involucra 2 nodos)
    - **DD** (Derecha-Derecha)
    - **II** (Izquierda-Izquierda)
  - COMPUESTA (Involucra 3 nodos)
    - **DI** (Derecha-Izquierda)
    - **ID** (Izquierda-Derecha)

Una vez que se detecta un nodo para el cual el factor de equilibrio es 2 o -2, se debe avanzar (a partir de éste) dos niveles por la rama más larga. El camino que se sigue determina la rotación.

**Algoritmos de Rotación**

- Rotación DD
  DERECHA(Nodo) ← IZQUIERDA(Nodo1)
  IZQUIERDA(Nodo1) ← Nodo
  Nodo ← Nodo1
- Rotación II
  IZQUIERDA(Nodo) ← DERECHA(Nodo1)
  DERECHA(Nodo1) ← Nodo
  Nodo ← Nodo1
- Rotación DI
  IZQUIERDA(Nodo1) ← DERECHA(Nodo2)
  DERECHA(Nodo2) ← Nodo1
  DERECHA(Nodo) ← IZQUIERDA(Nodo2)
  IZQUIERDA(Nodo2) ← Nodo
  Nodo ← Nodo2
- Rotación ID
  DERECHA(Nodo1) ← IZQUIERDA(Nodo2)
  IZQUIERDA(Nodo2) ← Nodo1
  IZQUIERDA(Nodo) ← DERECHA(Nodo2)
  DERECHA(Nodo2) ← Nodo
  Nodo ← Nodo2

**Metodología de Inserción en árboles AVL**

1- Seguir el camino de búsqueda del árbol hasta localizar la ubicación en la que debe insertarse el nuevo elemento.

2- Insertar el nuevo nodo y actualizar su factor de equilibrio (será 0 por ser hoja)

3- Regresar por el camino de búsqueda calculando los FE de los distintos nodos que se visiten a lo largo del mismo. Si alguno de los FE es 2 o -2, debe rebalancearse aplicando la rotación que corresponda.

4- El proceso termina cuando se vuelve al nodo raíz y todos los FE están entre -1 y 1 y no es necesario realizar ninguna rotación; o cuando se haya efectuado alguna rotación (en este caso, no es necesario efectuar el cálculo de los FE del resto de los nodos)

**Metodología de Eliminación en árboles AVL**

1- Seguir el camino de búsqueda del árbol hasta localizar la posición del nodo a eliminar.

2- Eliminar el nodo (según su grado)

3- Regresar por el camino de búsqueda calculando los FE de los distintos nodos que se visiten a lo largo del camino. Si alguno de los FE es 2 o -2, debe rebalancearse aplicando la rotación que corresponda.

4- El proceso termina únicamente cuando se vuelve al nodo raíz y todos los FE están entre -1 y 1.

A diferencia de la inserción, la eliminación puede provocar más de una rotación

## OPERADORES DEL TDA ÁRBOL N – ARIO

Sean **A** variable de tipo árbol general y **p** es variable de tipo posición

**Vacio(A)** → Devuelve verdadero si A es árbol Vacío.
**Nulo(p)** → Devuelve verdadero si p es la posición Nula
**HijoMasIzq(p,A)** → Devuelve la posición del hijo más a la izquierda de p, si p es hoja devuelve una posición nula.
**HermanoDer(p,A)** → Devuelve la posición del hermano a la derecha de p (tiene el mismo padre de p), si p es el de la extrema derecha devuelve una posición nula.
**Info(p,A)** → Devuelve el dato del en la posición p en el árbol A.
**Raiz(A**) → Devuelve una posición que es la raíz del árbol A.

Padre(p,A) → Devuelve la posición del padre de la posición p en el árbol A, si p es la raíz devuelve una posición nula.

# PROGRAMACION II

## MATERIAL para TEORIA

## Grafos

**Código Asignatura 6A4**

**Año 2018**

## REPRESENTACION

**↳ REPRESENTACION EN MEMORIA de GRAFOS**

① Representación Matricial ⇨ Matriz de Adyacencia
② Representación por medio de Listas ⇨ Lista de Adyacencia
⇨ Multilista

**↳ REPRESENTACION EN MEMORIA de DIGRAFOS**

① Representación Matricial ⇨ Matriz de Adyacencia
② Representación por medio de Listas ⇨ Lista de Adyacencia

Se definen:

- ✓ Matriz de Clausura Transitiva

$$CT[i,j] = \begin{cases} 1 \text{ si existe un camino de i a j de longitud} > 0 \\ 0 \text{ en caso contrario} \end{cases}$$

- ✓ Matriz de Clausura Transitiva Reflexiva

$$CTR[i,j] = \begin{cases} 1 \text{ si existe un camino de i a j de longitud} \geq 0 \\ 0 \text{ en caso contrario} \end{cases}$$

## GRAFOS

**Subgrafos**

Sea G = (V,E), un **subgrafo** G' de G es un grafo G' = (V',E') tal que:

- V' $\subseteq$ V
- E' $\subseteq$ E, tal que en E' hay aristas (v,w) $\varepsilon$ E con v y w $\varepsilon$ V'. Si en E' están todas las aristas (v,w) de E con v y w en V', se dice que G' es **subgrafo inducido.**

**Recorridos**

El objetivo es visitar todos los nodos del grafo exactamente una vez (se supone conexo)

- ✓ en *Amplitud*
  - se selecciona un nodo inicial, se visita y se marca como visitado
  - luego todos los nodos no visitados adyacentes al nodo inicial se visitan y se marcan como visitados
  - luego se visitan los nodos no visitados adyacentes a los ya visitados, tomándolos *en el orden* en el que se han visitado
  - se prosigue de la misma manera hasta que no haya nodos sin visitar.
- ✓ en *Profundidad*
  - se selecciona un nodo inicial, se visita y se marca como visitado
  - luego se visita un nodo adyacente al nodo inicial, se visita y se marca como visitado,
  - se prosigue de la misma manera tomando un nodo no visitado adyacente al *último nodo* visitado; en caso de no haber ninguno, se va intentando con los ya visitados en orden inverso al que se los visitó.
  - finaliza cuando no haya más nodos por visitar

En los recorridos antes mencionados, ¿qué pasaría si el grafo fuera no conexo?

**Árboles Abarcadores de Costo Mínimo (AAM)**

Sea G = (V,E) un grafo con aristas ponderadas.
Un árbol abarcador para G es un árbol libre (grafo conexo acíclico) que conecta todos los vértices de V, su costo es la suma de los costos de las aristas del árbol.
Un AAM para G es un árbol abarcador que conecta todos los nodos de V con mínimo costo.

Algoritmos para obtener AAM

- ✓ Algoritmo de **Kruskal**

Se considera la inclusión de las aristas en orden creciente por costo. Una arista se incluye sólo si no forma ciclo (un ciclo implicaría dos caminos entre un mismo par de vértices).

Paso 1: Generar un grafo T=(V,∅) constituido por todos los vértices de V y ninguna arista.

Paso 2: Para construir componentes conexas cada vez más grandes, se examinan las aristas de E en orden creciente de costo. Se agrega la arista si conecta dos vértices que se encuentran en componentes conexas distintas; en caso contrario, se descarta la arista.

Cuando todos los vértices están en una misma componente conexa, T será el AAM

- ✓ Algoritmo de **Prim**

Se inicia un conjunto U con un vértice cualquiera de V. Se consideran las aristas que conectan un vértice de U con un vértice de V-U, de ellas se elige la de menor costo.

Paso 1: Asignar a un conjunto U de vértices un vértice cualquiera, a partir de él se construirá el AAM

Paso 2: localizar la arista (u,v) con menor costo que conecta un vértice de U con un vértice de V-U, agregar v a U

El proceso termina cuando U=V

## DIGRAFOS

**Problemas de los caminos más cortos**

Sea G = (V,E) un digrafo conexo con aristas ponderadas.

- ✓ Algoritmo de **Dijkstra** (caminos mínimos con un solo origen)

Se define S como el *conjunto de vértices cuya distancia más corta desde el origen ya se conoce*. En principio S contiene sólo el origen. En cada paso se agrega algún vértice a S, cuya distancia en la más corta posible pasando sólo a través de vértices de S

Paso 1: Incluir en S el origen.

Paso 2: Calcular distancias (D[w]) desde el origen a cada uno de los vértices que no se encuentran en S (w ε V-S).

Paso 3: Elegir el vértice w (ε V-S) tal que D[w] sea mínima. Agregar w a S

Paso 4: Ir a Paso 2 hasta que S=V

- ✓ Algoritmo de **Floyd** (caminos mínimos entre todos los pares de vértices)

```
Inicializar matriz A
for (k = 1; k <=N ; k++)
   for (i = 1; i<=N ; i++)
      for (j = 1; j <=N ; j++)
          if (A[i,k] + A[k,j] < A[i,j])
              A[i,j] = A[i,k] + A[k,j];
```

- ✓ Algoritmo de **Warshall** (existencia de caminos entre todos los pares de vértices)

```
Inicializar matriz A
for (k = 1; k <=N ; k++)
   for (i = 1; i<=N ; i++)
      for (j = 1; j <=N ; j++)
          if (A[i,k] + A[k,j] !=0 && A[i,k] + A[k,j] != infinito)
              A[i,j] = 1;
```