UNIVERSIDAD AUTONOMA DE TAMAULIPAS
FACULTAD DE ADMINISTRACION Y CIENCIAS SOCIALES
LICENCIATURA EN INFORMATICA

ESTRUCTURA DE DATOS

ING. RAMON V. ROQUE HERNANDEZ, M.C.
ramonroque@hotmail.com

Descripción General de la materia :

La materia aporta herramientas metodológicas valiosas para el desarrollo de sistemas de información computacional, haciendo énfasis en el tratamiento interno de los datos.

Contenido de la materia:

Unidad 0.- Preliminares

- Definiciones importantes para programación
- Repaso de métodos, técnicas y enfoques
- Fundamentos básicos de diagramación

Unidad 1.- Fundamentos de Estructuras de Datos

- Variables y tipos de Datos
- Introducción a las Estructuras de Datos
- Estructuras simples en memoria (Registros)

Unidad 2.- Arreglos

- Unidimensionales
- Bidimensionales
- Multidimensionales

Unidad 3.- Pilas y Colas

- Filosofía de pilas y colas
- Operaciones

Unidad 4.- Listas

- Encadenadas
- Doblemente Encadenadas
- Circulares

Unidad 5.- Modularidad y Recursión

Unidad 6.- Árboles

Dinámica de la materia: Teórica/Práctica. Se explicarán los fundamentos de cada tema, y se enriquecerán con ejemplos prácticos y Ejercicios de aplicación (Algoritmos y Diagramas de Flujo), los cuales se pedirán implementar en un lenguaje de programación.

Forma de Evaluar:

Tareas, Investigaciones, Asistencia, Participaciones, Prácticas, Exámenes Parciales y Ordinario.

Alguna Bibliografía recomendada:

- Estructuras de datos. 2ª. Edición. Autores: Cairó – Guardati Edit. Mc Graw Hill. ISBN: 970-10-3534-8
- Estructuras de datos y algoritmos. M. A. Weiss. Addison-Wesley Iberoamericana. ISBN: 0-201-62571-7
- Data Structures using C. Tenenbaum, Langsam, Augnenstein, Prentice Hall. ISBN: 0-13-199746-7

UNIDAD 0 - PRELIMINARES

- **Sistema.-** Es un conjunto de elementos que interactúan entre si para alcanzar un objetivo común.
- **Algoritmo.-** Es una secuencia lógica y ordenada de pasos para resolver un problema.
- **Diagrama de Flujo.-** Es una herramienta simbólica convencional utilizada en el desarrollo de sistemas para expresar gráficamente un algoritmo.
- **Codificación de un programa (Programa de computadora).-** Conjunto ordenado de instrucciones en un lenguaje de programación para resolver un problema.
- **Lenguaje de Programación.-** Conjunto de símbolos, palabras, instrucciones, y reglas que se utiliza para escribir programas de computadora.

Importantes enfoques y metodologías para el desarrollo de sistemas

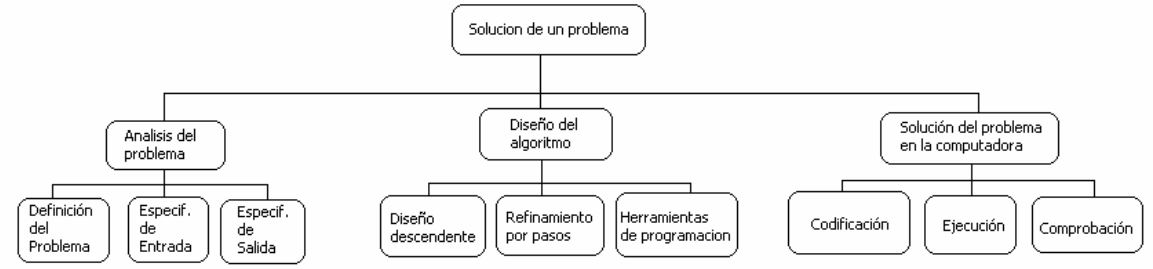
Principio de Modularidad en el desarrollo de sistemas.- Se refiere al proceso de dividir un sistema (programa o diagrama) en varios subprogramas, subtareas o módulos, con el propósito de hacerlo más entendible y fácil de modificar. "Divide y vencerás".

Subrutina o Procedimiento.- Es un fragmento de código que puede o no recibir parámetros, pero nunca regresa un valor como resultado de su ejecución.

Función.- Es un fragmento de código que puede o no recibir parámetros, pero siempre regresa un valor como resultado de su ejecución.

Pasos para resolver un problema con ayuda de la computadora.-

1. Identificar el problema
2. Crear un algoritmo de solución
3. Pasar el algoritmo a un diagrama de flujo
4. Elaborar una prueba de escritorio al algoritmo o diagrama de flujo
5. Pasar el algoritmo (Diagrama de flujo) a un lenguaje de programación (Codificación).
6. Introducir el programa a la computadora.



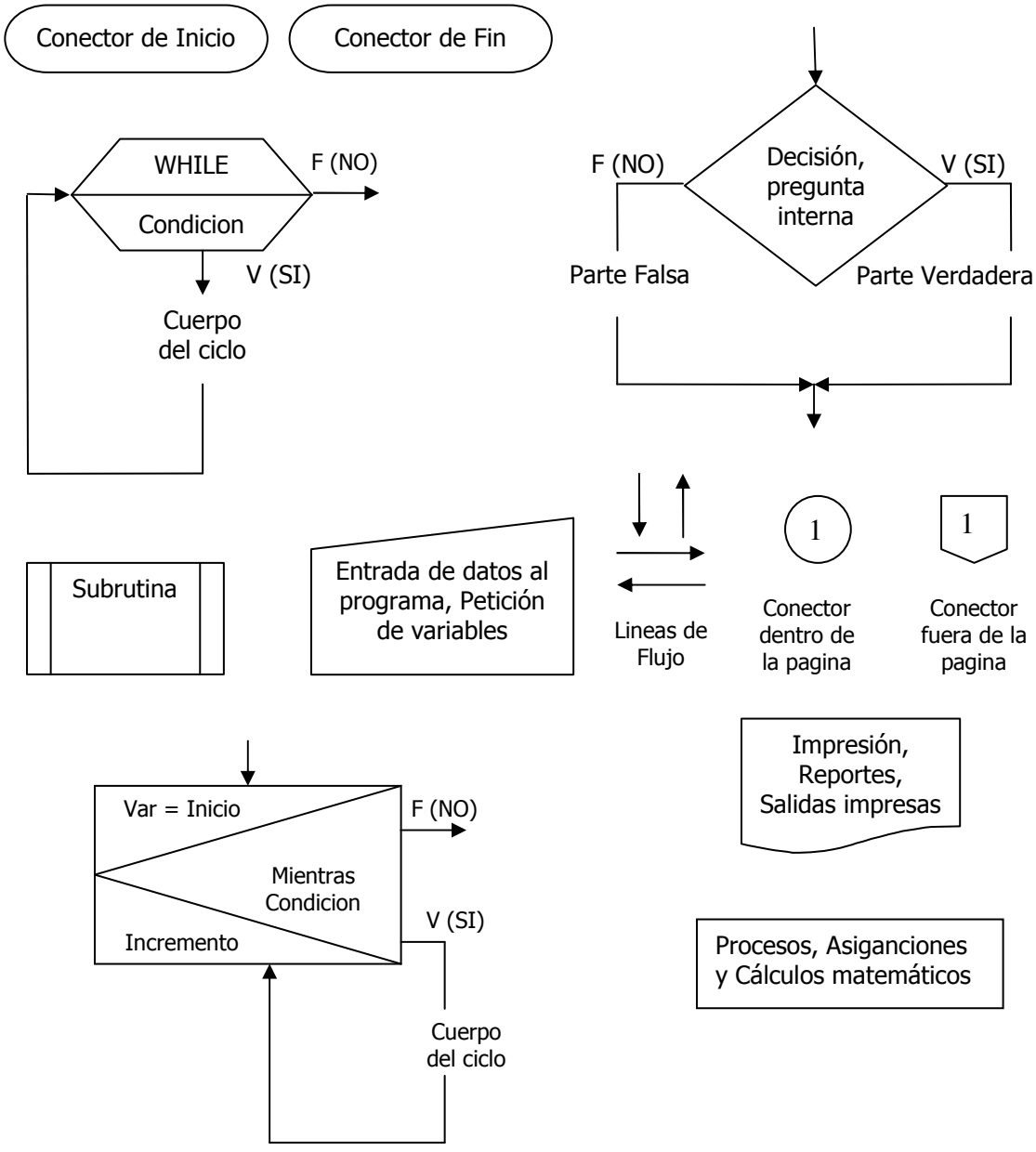
Enfoque de Entrada-Proceso-Salida para la solución de un problema.-
Se refiere a relacionar el problema a solucionar como una caja negra, e identificar:

- Un proceso.- Las actividades principales para resolver el problema.
- Las salidas.- Toda la información que debe generar el proceso para solucionar el problema.
- Las entradas.- Todos los datos que se deben proporcionar o introducir al proceso.

Fundamentos Básicos de Diagramación.-

- Todo diagrama de flujo tiene solo un Inicio y un Final
- Siempre deben estar presentes las líneas de flujo con sus respectivas flechas de dirección
- Un diagrama puede contener varios subdiagramas, o subrutinas
- Un diagrama debe iniciarse de arriba hacia abajo y/o de izquierda a derecha en la hoja.
- Se prefiere utilizar un lápiz en lugar de una pluma para los diagramas iniciales.
- Los ciclos (FOR, WHILE) siempre tiene una salida verdadera y una falsa
- Las decisiones dentro de un diagrama tienen solo dos salidas.
- Es preferible usar conectores en lugar de lineas grandes
- Los letreros de "V" y "F" siempre deben estar presentes en las decisiones y ciclos.

Símbolos utilizados para los diagramas de flujo.-



Repaso de Sintaxis de Lenguaje "C".-

Directrices

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <math.h>
etc.
```

Declaración de variables

```
int x;
char nombre[30];
int x,y,z;
int x[20]={0};
```

Inicio del programa

```
void main(void)
{
        …….
}
```

Instrucciones de Entrada/Salida

```
cout << "Teclee su nombre: " << endl;
cin  >> nombre;

printf ("  Teclee su calificacion:  \n");
scanf (" %i ", &calificación_entera);
```

Operadores

Operadores Relacionales

==, >, < , >= , <=, !=

Operadores Aritméticos

+, -, *, /, %

Operadores Lógicos

&&, ||, !

(AND, OR, NOT respectivamente)

Decisiones

```
 if ( x == 0)
{

}
```

```
if ( x==0 )
{

}
else
{

}
```

```
if ((x == 0 ) && ( y>=4))
{

}
else
{

}
```

Selección múltiple

```
switch (opcion)
{
        case 1 :  cout << "La opcion elegida ha sido 1" ;
                  break;
        case 2 : cout  << "La opcion elegida ha sido 2";
                  break;
        case 3 : cout << "La opcion elegida ha sido 3";
                  break;
        default:  cout << "La opcion es incorrecta";
                  break;
}
```

Ciclos

La instrucción **for** ejecuta instrucciones varias veces. Dentro del paréntesis se indica la variable que controla el ciclo, el valor de inicio de dicha variable y la condición que se estará evaluando cada vez que se incremente/decremente el valor de la variable del ciclo.
Con la instrucción for no es necesario cambiar manualmente el valor de la variable que controla el ciclo.

```
for (x=0; x<=10; x++)
{
        cout << x;
}
```

```
for (x=10; x>=0; x--)
{
        cout << x;
}
```

La instrucción **do..while** y **while** permiten realizar ciclos, evaluando una condición. Al utilizar estas instrucciones, se debe incrementar/decrementar manualmente el valor de la variable que controla el ciclo.

```
x=0;
do
{
        cout << x;
        x++;
}
while (x<=10);
```

```
x=0;
while (x<=10)
{
        cout << x;
        x++;
}
```

La sentencia **break;** permite terminar el ciclo actual.
La sentencia **continue;** permite pasar inmediatamente a la siguiente ejecución del ciclo actual.

Instrucciones especiales de Strings

Cuando se utilicen, siempre Incluir directriz ***#include <string.h>***
Asignación: **strcpy** (destino, fuente);
Comparación: **strcmp**(cadena1, cadena2);
Nota: strcmp Regresa un valor: cero si son iguales
mayor que cero si cadena1 > cadena,
menor que cero si cadena1 < cadena2

UNIDAD 1 - FUNDAMENTOS DE ESTRUCTURAS DE DATOS

Variables y tipos de datos

Durante la ejecución de un programa, se requiere almacenar diferentes datos en memoria para usarlos posteriormente: tal vez para almacenarlos en una base de datos o para realizar cálculos matemáticos con ellos.

En los lenguajes de programación, se utilizan variables para almacenar valores que pueden cambiar mientras el programa está ejecutándose. Las variables hacen posible procesar datos de cualquier tipo.

Es útil pensar en la memoria de la computadora como una bodega donde se almacenan todas las diferentes variables de un programa. Para almacenar un objeto cualquiera en una bodega cualquiera, se debe conocer:

- Que tanto espacio se requiere?
- Que tipo de objeto o articulo se va a almacenar?
- Como se puede localizar ese objeto después de que ha sido almacenado?

De la misma manera en la memoria de la computadora, una variable tiene:

- un nombre único por el cual se identifica (así podemos localizarla)
- un ***tipo de dato*** asociado (que determina la naturaleza del valor almacenado)
- una longitud (que determina la capacidad de almacenamiento)
- un alcance (que determina su disponibilidad)

Una variable puede ser local o global. Una ***variable local*** está disponible únicamente DENTRO del procedimiento, función o modulo en el que fue declarada. Una ***variable global*** está disponible en cualquier lugar dentro del programa en el que fue declarada.

Hay tipos de datos simples y estructurados. Los ***datos simples*** ocupan solo una casilla de memoria, por lo tanto una variable de un tipo de dato simple siempre hace referencia a un valor único a la vez. Los ***datos estructurados*** hacen referencia a un grupo de casillas de memoria. Un dato estructurado tiene varios componentes. Cada uno de los componentes puede ser a su vez un dato simple o estructurado. Ejemplos de datos simples son enteros, reales, carácter, booleanos, etc. Ejemplos de datos estructurados son: arreglos, registros (estructuras), etc.

Ejemplos de tipos de datos simples en algunos lenguajes de programación.-

Visual Basic .NET type	Storage size	Range of values
Boolean	2 bytes	True or False
Date	8 bytes	0:00:00 on January 1, 0001 to 11:59:59 P.M. on December 31, 9999
Decimal	16 bytes	Up to 29 significant digits, with values up to 7.9228×10^{28} (signed)
Double	8 bytes	-4.94065645841246544E-324 to +1.79769313486231570E+308 (signed)
Integer	4 bytes	-2,147,483,648 to +2,147,483,647 (signed)
Single	4 bytes	-3.4028235E+38 to 1.401298E-45 (signed)
String	Varies	0 to approximately 2 billion Unicode characters

Estructuras de Datos.-

- *El término "Estructuras de datos" se refiere a la manera como se organizan, estructuran y manipulan internamente los datos en un programa.*
- *El objetivo de utilizar "Estructuras de datos" es facilitar la manipulación de los datos.*

La información que se procesa en la computadora es un conjunto de datos, que pueden ser simples o estructurados.

Los datos simples son aquellos que ocupan sólo una localidad de memoria, mientras que los estructurados son un conjunto de casillas de memoria a las cuales hacemos referencia mediante un identificador único. Existen tipos de datos simples y estructuras de datos adecuadas para cada necesidad.

Las estructuras de datos son una colección de datos cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y acceder a elementos individuales de datos.
Una estructura de datos se caracteriza por lo siguiente:

- Pueden descomponerse en los elementos que la forman.
- La manera en que se colocan los elementos dentro de la estructura afectará la forma en que se realicen los accesos a cada elemento.
- La colocación de los elementos y la manera en que se accede a ellos puede ser encapsulada.

Necesidad de Utilizar Estructuras de datos.-

Suponer el siguiente problema: Una empresa cuenta con 150 empleados, y desea establecer una estadística sobre los salarios de sus empleados, y quiere saber cual es el salario promedio, y también cuantos y cuales de sus empleados ganan mas de ese promedio.

Para la captura y cálculos deseados, el utilizar tipos de datos simples nos llevaría a un enorme desperdicio de tiempo, almacenamiento y velocidad. Es por eso que en casos similares a este, la mejor opción que tiene un programador es utilizar tipos de datos estructurados.

Ejemplos de algunas estructuras de Datos.-

Registros, Arreglos (Vectores, Matrices), Listas enlazadas, Pilas, Colas, Árboles, Archivos, etc.

Ejemplos de algunas operaciones realizadas en una estructura de datos.-

- Altas (Inserción, adición de un nuevo valor)
- Bajas (Eliminación de un valor de la estructura)
- Búsqueda (Encontrar un determinado valor en la estructura)
- Ordenamiento (Cambio de orden de los elementos pertenecientes a la estructura).
- Unión (Dadas dos estructuras originar una nueva).

Ventajas y Desventajas.-

Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De esta forma, la elección de la estructura de datos apropiada para cada problema depende de factores como las frecuencias y el orden en que se realiza cada operación sobre los datos.

Estructuras simples en memoria (Registros).-

Una estructura en memoria (Registro) es una colección de datos elementales de diferentes tipos, relacionados de manera lógica dentro de un nombre único de variable. A una estructura de este tipo se le da también el nombre de registro.

Crear una estructura es definir un nuevo tipo de datos, denominado "Tipo Estructura". En la definición de una estructura, se especifican los elementos que la componen, así como sus tipos. Cada elemento de la estructura recibe el nombre de miembro o campo.

Se hace referencia a cada uno de los elementos de la estructura por el nombre de la estructura seguido de un punto y el nombre del campo.

El uso de estructuras o registros permite tener los datos organizados en la memoria de una mejor manera y agrupados bajo un mismo nombre. Lenguajes como C y Visual Basic permiten el uso de estructuras.

Ejemplo en Lenguaje C

Declaración de la estructura y de la variable:

```
struct Estructura_Alumno
{
        char nombre[30];
        int   calificacion;
};
struct Estructura_Alumno estudiante;
```

Referencia en Código:

```
cin >> estudiante.nombre;
cin >> estudiante.calificacion;
```

Ejemplo en Visual Basic

Declaración de la estructura y de la variable:

```
Private Type Estructura_Alumno
    nombre As String
    calificacion As Integer
End Type

Dim estudiante As Estructura_Alumno
```

Referencia en código:

```
estudiante.nombre = InputBox("nombre:")
estudiante.calificacion = Val(InputBox("Calificacion:"))
```

Representación de manera gráfica en memoria:

estudiante *(tipo de dato: Estructura_Alumno)*

Nombre (Tipo de dato: String)	*Calificación* (Tipo de dato: Entero)

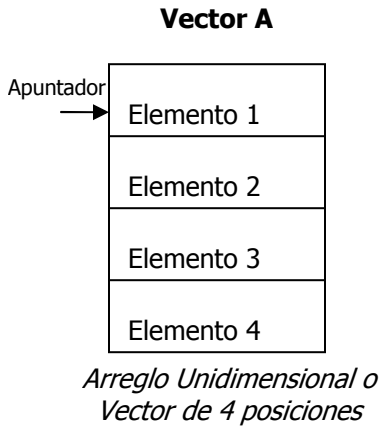
UNIDAD 2 – ARREGLOS

Un arreglo es un conjunto finito de posiciones de memoria consecutivas que tienen el mismo nombre. Los elementos de un arreglo son homogéneos (es decir, del mismo tipo de dato). Un arreglo puede estar compuesto de elementos de tipo cadena, o elementos de tipo numérico, pero no puede tener combinación de ambos.

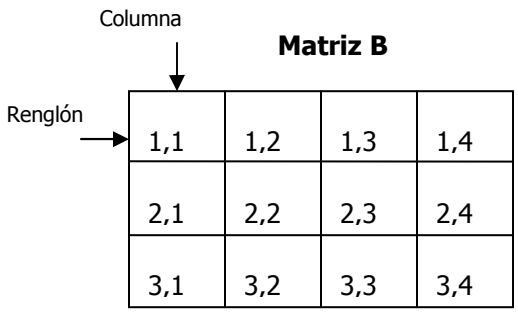
En el área de las matemáticas, los arreglos se conocen también como "vectores" y "matrices"; y como "Tablas" en los cálculos financieros.

El tipo mas simple de arreglo es el unidimensional o ***vector***. Los arreglos de dos dimensiones se conocen como ***matrices***.

El uso de arreglos permite acceso a cualquier tipo de datos almacenados previamente y proporciona una gran capacidad para denominar y manejar grandes posiciones de almacenamiento.



Arreglo Unidimensional o Vector de 4 posiciones



Arreglo Bidimensional o Matriz de 3 x 4

Cuando se utilizan arreglos, un gran número de posiciones de memoria pertenecen al mismo nombre de variable. La localización de cada posición de memoria en el arreglo, se denomina elemento del arreglo, el cual puede referenciarse por su posición relativa en el arreglo a través del uso de un subíndice.

Un subíndice actúa como indicador para localizar un elemento de arreglo específico. Los subíndices se escriben entre paréntesis o corchetes después del nombre del arreglo.

En un arreglo unidimensional o Vector, cada elemento se referencía por ***un subíndice***. En un arreglo bidimensional o Matriz, cada elemento se referencía por ***dos subíndices***.

Ejemplos de referencia a elementos de un vector: *A(2) VectorA(49) Numeros(X)*
Ejemplos de referencia a elementos de una matriz: *M(Ren,Col) Alumnos(3,5) Matriz(1,1)*

Consideraciones al trabajar con arreglos.-

- Los subíndices pueden ser:
  - Constantes enteras
  - Variables que almacenan valores enteros
  - Expresiones cuyo resultado sean valores enteros

- Se puede utilizar un nombre de variable individual como subíndice para referirse a elementos de diferentes arreglos.
- Los subíndices no son parte del nombre del arreglo. Por lo tanto A(X) y A(Y) se refieren al mismo arreglo A. Si X, Y son iguales, entonces A(X) y A(Y) se refieren al mismo elemento en el arreglo A.

Errores comunes al trabajar con arreglos.-

- Confundir el subíndice con el elemento del arreglo al cual se refiere. Es útil hacerse la pregunta: "¿Deseo referirme al subíndice o al elemento del arreglo que señala el subíndice?"
- Hacer referencia a un elemento que excede de la longitud del arreglo, lo cual produce un error en la ejecución del programa.

Técnicas de programación utilizadas cuando se trabaja con arreglos.-

NOTA: *Estas técnicas son válidas independientemente del lenguaje de programación que se utilice.*

- Es común utilizar un ciclo sencillo para manipular arreglos unidimensionales (Vectores).

Ejemplo en Visual Basic:

```
FOR X = 1 TO 10
        VECTOR(X) = X
NEXT X
```

En Lenguaje C:

```
for (x=0; x<=9; x++)
{
        vector[x] = x;
}
```

- Generalmente se utilizan dos ciclos (anidados) para manipular arreglos bidimensionales (matrices). Cada ciclo es controlado por una variable única que sirve como apuntador a la matriz en el formato (renglón, columna)

Ejemplo en Visual Basic:

```
FOR X = 1 TO 10
        FOR Y = 1 TO 10
                SUMA(X,Y) = A(X,Y) + B (X,Y)
        NEXT Y
NEXT X
```

En Lenguaje C:

```
for (x=0; x<=9; x++)
{       for (y=0; y<=9; y++)
        {   suma[x][y] = a[x][y] + b[x][y];
        }
}
```

- Cuando no se conoce la cantidad de elementos sobre los cuales se realizarán operaciones, se utilizan variables para establecer los límites:

Ejemplo en Visual Basic:

```
INICIO = VAL(INPUTBOX "Teclee el valor de Inicio")
FIN  = VAL(INPUTBOX "Teclee el valor de Fin")
FOR X = INICIO TO FIN
        VECTOR(X) = X
NEXT X
```

En Lenguaje C:

```
scanf("%i", &inicio);
scanf("%i", &fin);
for (x=inicio; x<=fin; x++)
{
        vector[x] = x;
}
```

- La mayoría de los lenguajes de programación proporcionan varias opciones para realizar ciclos. Si se conoce el número de iteraciones (o por lo menos es predecible), se utiliza FOR. Si la continuidad del ciclo depende de una condición adicional a la que proporciona el FOR, se utiliza WHILE, DO WHILE, REPEAT, Etc.

- Se utilizan variables mnemónicas para el uso de ciclos y de los arreglos, esto quiere decir, que los nombres de las variables deben ser relacionados con el uso que se les da dentro del programa, lo cual permite recordarlas fácilmente.

Arreglos de Estructuras.-

De la misma manera como se pueden realizar arreglos de otros tipos de datos (entero, caracter, punto flotante, etc.), también son válidos los arreglos de estructuras (también llamados arrays de registros o de estructuras).

Ejemplo en Lenguaje C:

Declaración de la estructura y de la variable:

```
struct  Estructura_Alumno
{
        char nombre[30];
        int   calificacion;
};

struct Estructura_Alumno  estudiante[10];
```

Referencia en Código:

```
for(x=0; x<=9; x++)
{
        cin >> estudiante[x].nombre;
        cin >> estudiante[x].calificacion;
}
```

Ejemplo en Visual Basic:

Declaración de la estructura y de la variable:

```
Private Type Alumno_Estructura
    nombre As String
    calificacion As Integer
End Type

Dim alumno(10) As Alumno_Estructura
```

Referencia en código:

```
For x = 1 To 10
  alumno(x).nombre = InputBox("nombre:")
  alumno(x).calificacion = Val(InputBox("Calificacion:"))
Next x
```

Representación gráfica de un arreglo de estructuras en memoria:

Estudiante *(tipo de dato: Estructura_Alumno)*

0	*Nombre* (Tipo de dato: String)	*Calificación* (Tipo de dato: Entero)
1	*Nombre* (Tipo de dato: String)	*Calificación* (Tipo de dato: Entero)
2	*Nombre* (Tipo de dato: String)	*Calificación* (Tipo de dato: Entero)
…	……	……
9	*Nombre* (Tipo de dato: String)	*Calificación* (Tipo de dato: Entero)

Operaciones comunes con Vectores

Búsqueda Secuencial en un Vector

- Consiste en buscar un elemento determinado, recorriendo una a una cada posición del vector. La búsqueda termina cuando se encuentra el elemento buscado, o bien, cuando todos los elementos del vector terminan de recorrerse.

Ordenamiento de los elementos de un vector

- Existen varios métodos para ordenar los elementos de un vector, el mas conocido, pero no el mas eficiente, es el método de la burbuja.
- El método de la burbuja consiste en comparar el primer elemento del vector con cada uno de los demás elementos del vector, y si es menor (o mayor, según se desee el orden), se intercambia la posición del elemento. Este proceso se repite para cada uno de los elementos en el arreglo.

Determinación del elemento mas grande y mas pequeño de un vector.

- Una técnica muy utilizada es tener dos variables: MAYOR y MENOR y asignarles inicialmente valores de la siguiente manera:
  - MAYOR = el primer elemento del arreglo
  - MENOR = el primer elemento del arreglo

- Recorrer secuencialmente el vector y comparar cada elemento:
  - Si el elemento es mayor que la variable MAYOR, MAYOR = Elemento del Vector.
  - Si el elemento es menor que la variable MENOR, MENOR = Elemento del Vector.
- Al terminar el recorrido, las variables MAYOR y MENOR contendrán los elementos mas grande y mas pequeño del vector, respectivamente.

Operaciones comunes con Matrices

Búsqueda secuencial en una matriz

- Dos ciclos anidados resuelven casi cualquier situación de captura, procesamiento o impresión de datos relacionados con una matriz.

Ordenamiento de los elementos de una matriz

- Un método simple de programación para ordenar los elementos de una matriz es vaciar los datos de la matriz a un vector lineal y aplicarle un proceso de ordenamiento. Posteriormente se regresan los datos a la matriz.

Operaciones matemáticas con matrices

- En la suma, resta y división matricial, las operaciones se realizan elemento por elemento, como normalmente se haría con operandos.
- La multiplicación de matrices está definida únicamente para dos matrices A y B dadas de tamaño n x m y m x p dando como resultado una matriz C de tamaño n x p
- Una matriz A de tamaño n x m tiene su traspuesta AT de tamaño m x n , donde los renglones de A pasan a ser las columnas de AT, y por consecuencia, también las columnas de A son los renglones de AT. Es decir, AT (i,j) = A (j,i)

Importantes notas sobre como se realiza la multiplicación de matrices.-

Para multiplicar 2 matrices, **A** y **B**, el numero de *columnas* de **A** debe ser igual al numero de *filas* de **B**. Así, si **A** es una matriz de *n* x *m* y **B** es de *m* x *p*, el resultado de hacer **AB** sera una matriz de *n* x *p*. La operación consiste en multiplicar los elementos de las *columnas* de la matriz **A**, por los elementos de las *filas* de la matriz **B**:

$$C_{ij} = A_{i1} B_{1j} + A_{i2} B_{2j} + A_{i3} B_{3j} + \ldots + A_{in} B_{nj} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Ejemplo:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

$$A * B = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (3*6)+(4*8) \end{pmatrix}$$

$$A * B = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

UNIDAD 3. PILAS Y COLAS

PILAS

Se denomina ***pila*** a una lista lineal, en la que solamente se puede acceder a sus elementos por un extremo de la misma.

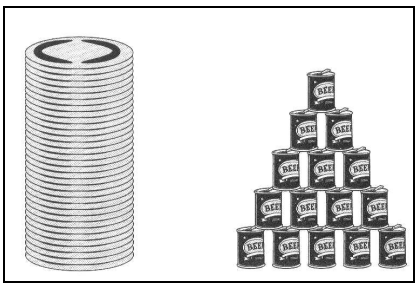
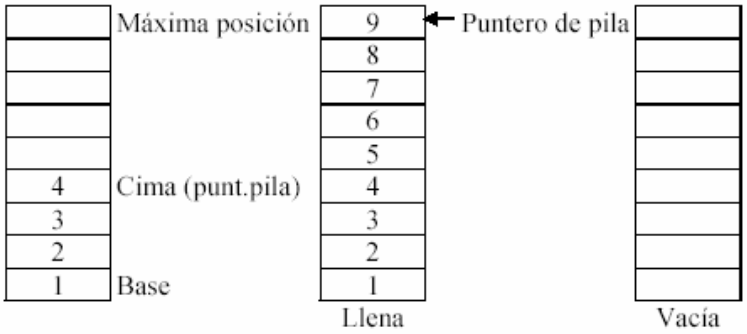
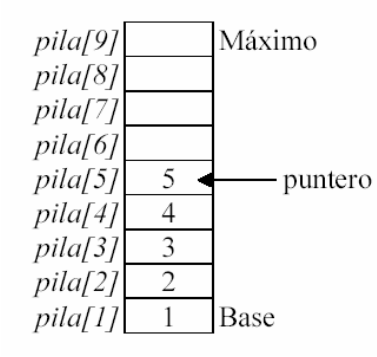


Figura: Ejemplos prácticos de pilas

- El elemento que ocupa este extremo recibe el nombre de ***cima o Tope de la pila***.
- El otro extremo de la pila, que en teoría es inaccesible directamente, recibe el nombre de ***base de la pila***.
- La pila se comporta con filosofía ***LIFO*** (*Last Input, First Output*), es decir, los elementos de una pila serán eliminados en orden inverso al que se insertaron, ya que el último elemento que entra en la pila es el primero que sale de ella.
- En toda pila es necesario el llamado ***puntero de pila***. Este puntero señala o bien la última posición ocupada de la pila, o bien la primera posición libre de la misma, según su implementación.



- En la pila sólo son posibles dos operaciones: **PUSH** (introducir un elemento en la misma) y **POP** (extraer un elemento de ella).
- La posición señalada por el puntero de pila no puede ser mayor que la máxima posición de memoria permitida en la pila, de darse este caso se produciría un desbordamiento (*overflow*) de la pila. De modo similar, la posición de memoria indicada por dicho puntero no puede ser menor que la posición de memoria del elemento base de la pila.
- Si el elemento base no existe la pila está vacía, y si el puntero de la pila es igual a la máxima posición de memoria permitida, la pila está llena.
- Una pila se puede implementar con el uso de un vector. En este caso, el tamaño máximo de la pila estará limitado al que tenga el vector.

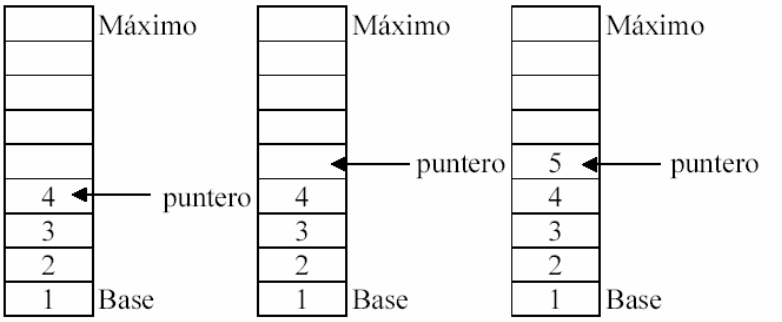


Operaciones con Pilas

PUSH.-

Añadir un elemento a la pila consiste en:

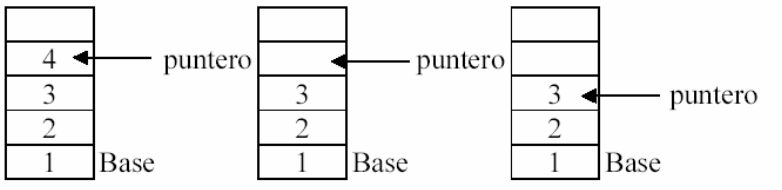
- Verificar si la pila está llena.
- Si no está llena, incrementar en una unidad el puntero de pila.
- Introducir el nuevo elemento en la posición indicada por el puntero de pila.



POP.-

Sacar un elemento de la pila consiste en:

- Verificar si la pila está vacía
- Si no está vacía, extraer el elemento de la pila
- Decrementar el puntero de la pila en una unidad.



COLAS

- *Es una estructura de datos semejante al vector, con la única condición de que el primer elemento que entra, es el primero que sale. Por este comportamiento, se dice que las colas poseen filosofía **FIFO** (First Input, First Output).*

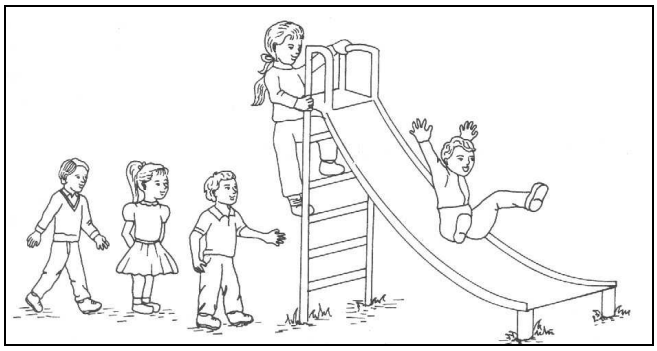


Figura: Ejemplo práctico de Colas

Se denomina cola a una lista, en la que las extracciones se realizan sólo por el principio de la lista y las inserciones solamente por el final de la misma.

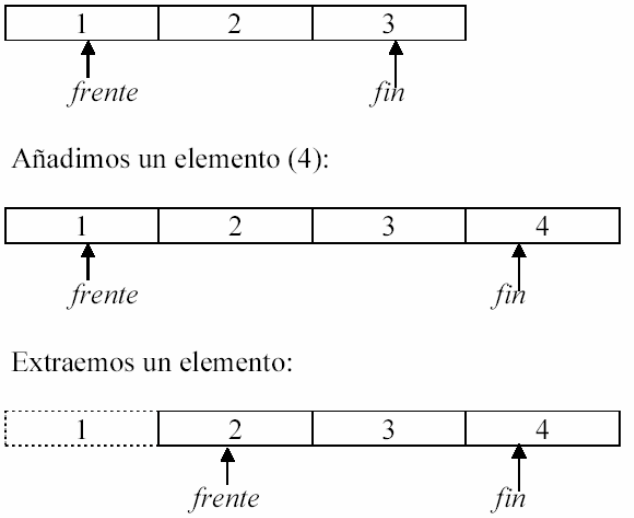
El elemento que está al principio de la cola se denomina ***frente***, y el que está al final se denomina ***final***.

La cola se comporta con filosofía FIFO (F*irst Input, First Output*), es decir, el primer elemento que entra en la cola es el primero que sale de ella.

En la cola sólo son posibles dos operaciones: introducir y extraer un elemento, para realizarlas se precisan ***dos punteros*** para señalar el principio (*frente*) y el final de la cola (*fin*).

Al insertar un elemento en la cola, se debe actualizar el ***Puntero del Final de la cola***.
Al extraer un elemento de la cola, se debe actualizar el ***Puntero del Frente de la cola***.

Las colas, al igual que las pilas, se pueden implementar con el uso de vectores en memoria.



Consideraciones en las operaciones con COLAS.-

Al Ingresar un elemento en la cola

- Si es el primer elemento: FIN =1, FRENTE = 1, PONER ELEMENTO EN LA COLA.
- Los demás casos: FIN = FIN + 1, PONER ELEMENTO EN LA COLA.

Al extraer un elemento de la cola

- Si la cola está vacía: Indicar Mensaje de Error
- Si sale el ultimo elemento: ELIMINARLO, FIN = 0, FRENTE = 0
- Al salir los demás elementos: ELIMINARLO, FRENTE = FRENTE + 1

Como verificar

- Si va a entrar el primer elemento (La cola Está vacía): FRENTE = FIN = 0
- Si sale el ultimo elemento : FRENTE = FIN

APLICACIONES DE PILAS Y COLAS EN LA COMPUTACION

Aplicaciones de Pilas

Las pilas son una estructura de datos muy usada en la solución de diversos problemas como por ejemplo:

- **Llamadas a subprogramas y recursión**.- Cuando se tiene un programa que llama a un subprograma, internamente se usan pilas para guardar el estado de las variables del programa en el momento que se hace la llamada. Asi cuando termina la ejecución del subprograma, los valores almacenados en la pila pueden recuperarse para continuar con la ejecución del programa en el punto en el cual fue interrumpido. Además de las variables, debe guardarse la dirección del programa en la que se hizo la llamada, porque es a esa posición a la que regresa el control del proceso.

- **Tratamiento de expresiones aritméticas**.- Las pilas se utilizan para convertir expresiones en notación infija a su equivalente en notación prefija. También son útiles para determinar si las expresiones matemáticas se encuentran equilibradas en paréntesis.

- **Ordenación.**- Algunos métodos de ordenación utilizan pilas para llevar a cabo su proceso.

Aplicaciones de Colas

El concepto de cola está ligado a la computación. Algunos ejemplos son:

- **Colas de Impresión**.- Cuando hay una sola impresora para atender a varios usuarios, suele suceder que algunos de ellos soliciten los servicios de impresión al mismo tiempo o mientras el dispositivo está ocupado. En estos casos se forma una cola con los trabajos que esperan para ser impresos, éstos se procesarán en el orden en el cual fueron introducidos en la cola.

- **Sistemas de Tiempo Compartido.-** Varios usuarios comparten ciertos recursos, como CPU y memoria de la computadora. Los recursos se asignan a los procesos que están en cola de espera, suponiendo que todos tienen una misma prioridad, en el orden en el cual fueron introducidos en la cola.

UNIDAD 4. LISTAS

▪ Introducción y Generalidades

Una *LISTA* es una colección de elementos llamados *"nodos".* El orden entre los nodos se establece por medio de punteros, es decir, direcciones o referencias a otros nodos.
Un NODO consta de dos campos: DATO y DIRECCION DEL SIGUIENTE ELEMENTO.
El campo "DATO" contiene el elemento en sí, y el campo "DIRECCION" contiene la dirección del siguiente elemento en la lista.
El campo de dirección relaciona los elementos y establece su secuencia, además permite que no sea necesario almacenar físicamente a los nodos en espacios contiguos, ya que basta con seguir el enlace indicado para formar la lista completa.
Una lista sin nodos se conoce como Lista Vacía o Lista Nula.

▪ Tipos de Listas

Una lista puede ser Enlazada simple (Con un solo campo de dirección para el siguiente elemento), o Doblemente Enlazada (Con dos campos de dirección, uno para el siguiente elemento y otro para el anterior, lo cual permite ascender o descender en la lista).
A una lista También se puede agregar la característica Circular (Donde el último elemento apunta al primero),

▪ Implementación de Listas Enlazadas en Lenguajes de Programación

Para implementar las listas enlazadas se hace uso de un VECTOR DE ESTRUCTURAS conteniendo N elementos con los dos campos mencionados (DATO y SIG).
El campo SIG del último elemento de la lista tiene un valor especial único que indica el Fin de la lista. Este valor puede ser NULO, CERO o -1, dependiendo de la implementación.
También se utiliza una variable llamada "APUNTADOR EXTERNO" (APEX) que contiene la dirección de inicio de la lista (Es decir, la dirección del primer elemento).
Otra variable llamada "TOPE" indicará la posición secuencial correspondiente donde se insertará el elemento.

Ejemplo:
Al recorrer la siguiente lista en memoria, obtendríamos la siguiente secuencia de elementos:
ARBOL, CASA, GATO, ZAPATO

APEX = 3

LISTA

	Dato	**Sig**
1	CASA	4
2	ZAPATO	0
3	ARBOL	1
4	GATO	2

Como El apuntador externo vale 3, se inicia el recorrido en esa posición. El primer elemento de la lista es entonces "ARBOL" ya que se encuentra en la posición 3; para obtener el siguiente elemento se consulta el campo "SIG" del elemento 3, el cual contiene un 1; esto indica que el siguiente elemento es "CASA". Al tomar el valor del campo "SIG" del elemento "CASA", se obtiene un 4 por lo que el siguiente elemento es "GATO", quien a su vez hace referencia al elemento 2 el cual es "ZAPATO", siendo este el ultimo elemento de la lista (pues el campo SIG=0).

Operaciones con Listas

Se pueden realizar operaciones como Altas, Bajas, Búsquedas y Listado (Recorrido de la lista).

Altas a Listas

Al ingresar un dato a una lista se conocen cuatro casos:

Alta en una Lista Nula

- Se inserta el elemento en la posición secuencial correspondiente
- El apuntador externo apunta al primer elemento de la lista
- El apuntador al siguiente elemento es NULO o CERO

Alta al Principio de la Lista

- Se inserta el elemento en la posición secuencial correspondiente
- El puntero de ese nodo toma el valor del apuntador Externo
- El apuntador Externo apunta a ese nodo

Alta al Final de la Lista

- Se inserta el elemento en la posición secuencial correspondiente
- El apuntador nulo del ultimo elemento de la lista apunta al nodo insertado
- El campo de dirección del nodo insertado es NULO o CERO

Altas en una Posición Intermedia de la lista

- Se inserta el elemento en la posición secuencial correspondiente
- El apuntador de este nodo toma el valor del apuntador anterior lógico
- El apuntador del nodo anterior lógico apunta al nodo nuevo

Listado (Recorrido de la Lista)

La operación de recorrido consiste en visitar cada uno de los nodos que forman la lista. La visita de un nodo puede definirse por medio de una operación muy simple (por ejemplo, la impresión de la información del mismo), o por medio de operaciones tan complejas como se desee.
Para recorrer todos los nodos de una lista se comienza con el primero. Tomando el valor del campo de dirección de éste se avanza al segundo; a su vez, el campo de dirección del segundo dará acceso al tercero, y así sucesivamente.

Bajas a Listas

Al eliminar nodos, Se identifican dos casos:

Bajas al principio de la Lista

- El apuntador externo toma el valor del apuntador del primer nodo

Bajas al final o intermedias en la Lista

- Se recorre la lista hasta encontrar el elemento a dar de baja; el apuntador del nodo anterior lógico toma el valor del nodo a borrar.

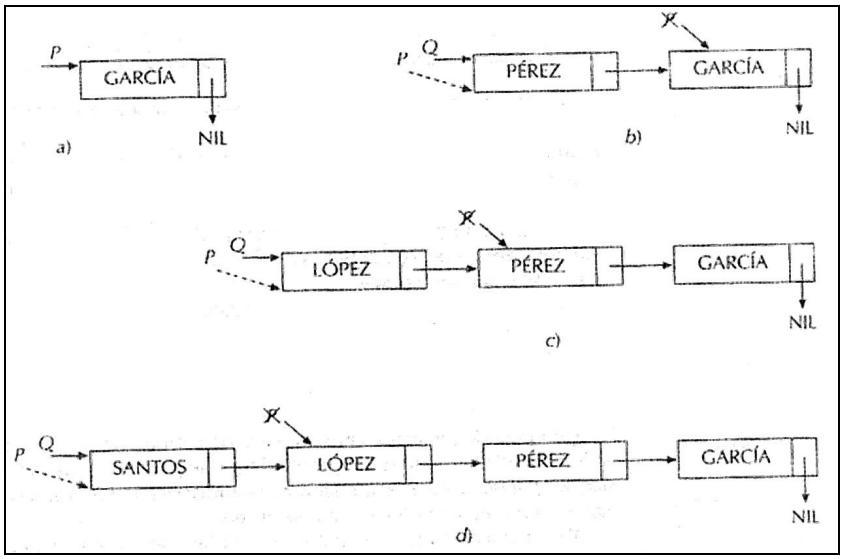


Figura 4.1 Altas al principio de la lista

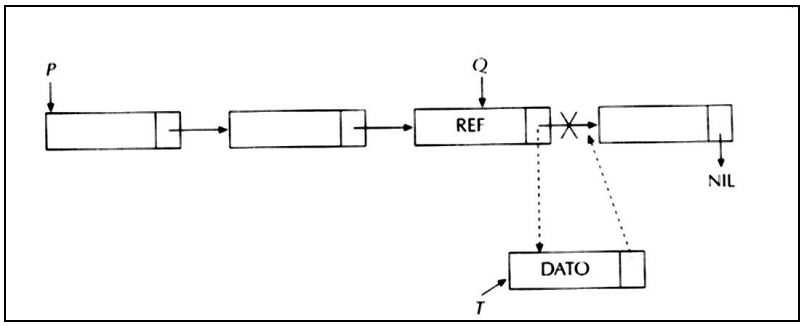


Figura 4.2 Altas en una posición Intermedia de la lista.

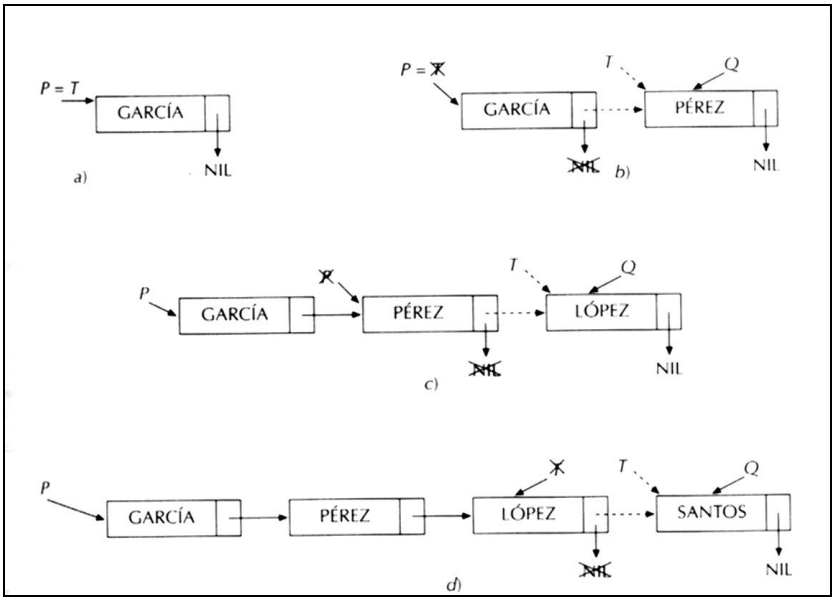


Figura 4.3 Altas al Final de la lista.

Ejercicio: Entendiendo las listas.- Determinar el contenido de las siguientes listas de acuerdo al orden de los apuntadores.

EJERCICIO 1: APEX = 5
TOPE = 6

"LISTA"

	Dato	Sig
1	implementar	0
2	listas	4
3	sencillas	6
4	son	3
5	Las	2
6	De	1

EJERCICIO 2: APEX = 4
TOPE = 10

"LISTA"

	dato	Sig
1	Rosa	3
2	Lodo	8
3	Sal	5
4	Arco	7
5	Taller	9
6	Oso	1
7	Bote	2
8	Mano	10
9	Uva	0
10	niño	6

Ejercicio: Operaciones de Altas a Listas.- Dibujar el contenido de la Lista y de los apuntadores en cada paso indicado.
NOTA: Se inicia con una lista NULA (VACIA)

APEX = 0 TOPE = 0

	dato	Sig

1. Insertar "M" en la lista NULA

APEX = TOPE =

	dato	Sig
1		

2. Agregar "B" al principio de la lista.

APEX = TOPE =

	dato	Sig
1		
2		

3. Agregar "W" al final de la lista

APEX = TOPE =

	dato	Sig
1		
2		
3		

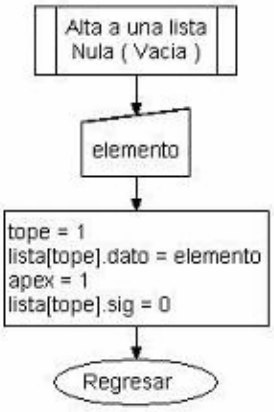
4. Insertar "R" después del elemento "M" (Elemento 1)

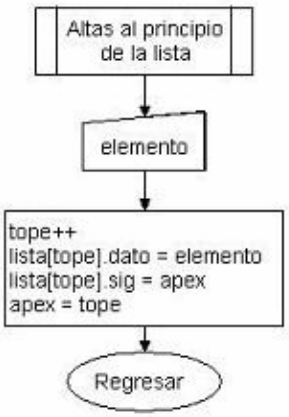
APEX = TOPE =

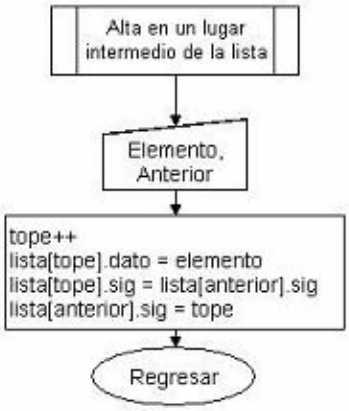
	dato	Sig
1		
2		
3		
4		

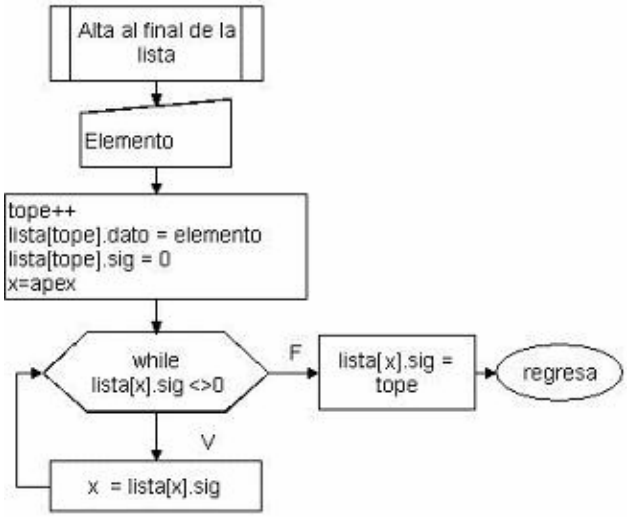
Diagramas de UNIDAD 4 - LISTAS

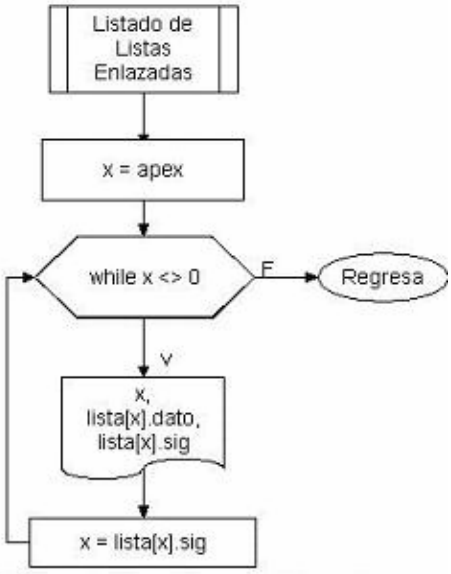
Subject:	Estructuras de Datos	Author/Team:	Ing. Ramon Roque Hernandez,MC
Date:		Student Name:	











UNIDAD 5. MODULARIDAD Y RECURSION

MODULARIDAD

Es el Proceso de dividir un sistema (problema, diagrama o programa) en varias subtareas, subprogramas o módulos de menor tamaño, con el propósito de hacerlo mas entendible y fácil de modificar. Esta división exige la presencia de un módulo denominado módulo base o principal que controle y se relacione con los demás.

Módulo.-

Es aquél que está constituido por una o varias instrucciones físicamente contiguas y lógicamente encadenadas, las cuales se pueden referenciar mediante un nombre y pueden ser llamadas desde diferentes puntos de un programa.

Un módulo puede ser:

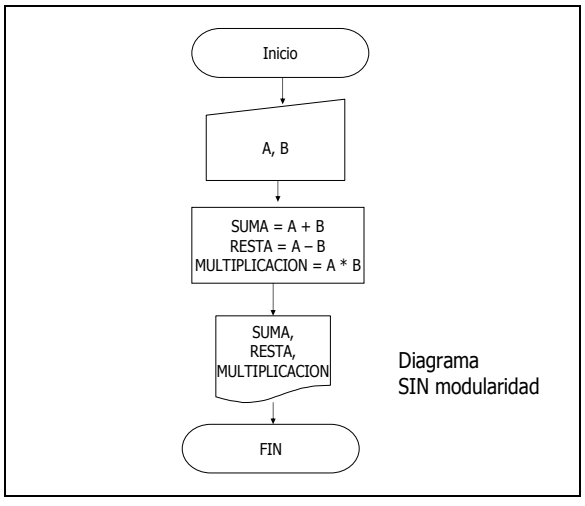
- Un programa
- Una función
- Una subrutina (o procedimiento)

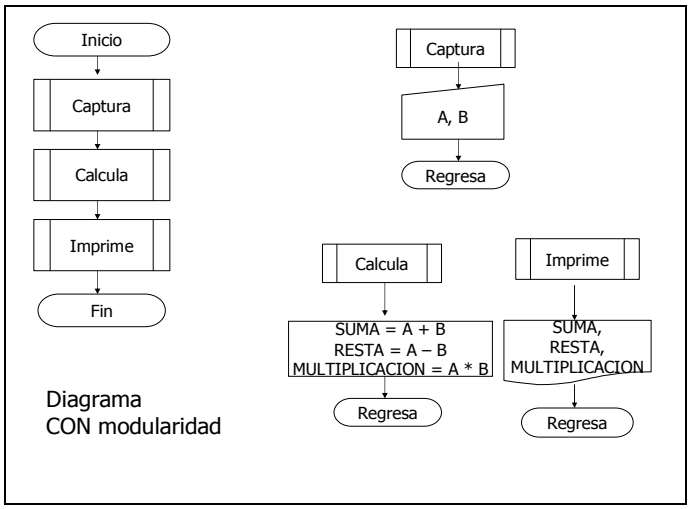
Ventajas de utilizar modularidad.-

- Un programa modular es fácil de mantener, controlar y modificar.
- Un programa modular es más fácil de escribir y depurar (ejecutar, probar y poner a punto).
- El desglose de un problema en módulos permite encomendar los módulos más complejos a los programadores más experimentados y los más sencillos a los programadores principiantes.
- Posibilita el uso repetitivo de las rutinas en el mismo o en diferentes programas.

Desventajas de utilizar modularidad.-

- No se dispone de algoritmos formales de modularidad, por lo que a veces los programadores no tienen claras las ideas de los módulos.
- La programación modular requiere más memoria y tiempo de ejecución.





Reflexiones:

- Si existieran cambios futuros, por ejemplo, agregar mas operaciones matemáticas, o incluir un menú para decidir cuál resultado mostrar, ¿Cuál diagrama sería mas simple de modificar?
- Si se tuviera que dividir este programa para que varios desarrolladores lo implementaran, ¿Cuál diagrama sería mas útil?
- ¿Cual diagrama resulta mas sencillo de entender?

Tipos de "Módulos".-

Los módulos pueden o no recibir "Parámetros" al momento de ser invocados. También pueden o no regresar algún resultado al terminar de ejecutarse.
Un parámetro es un valor que requiere un módulo para poder ejecutarse.
El módulo también puede "entregar" un valor como resultado de su ejecución.
Los módulos que requieren parámetros, y/o regresan valores se les llama comunmente funciones.
Los módulos que no requieren parámetros ni regresan valores se les llama comunmente subrutinas.

Variables Locales y Globales.-

Las variables definidas dentro de un módulo son "locales" y están accesibles solamente dentro del cuerpo de ese módulo.
Las variables definidas fuera de los módulos son "globales" y están accesibles en todos los módulos.

Implementación de la modularidad en Lenguajes de Programación.-

Cada lenguaje de programación tiene su sintaxis propia para el uso de modulos. A continuación se muestran dos maneras de implementar el mismo programa (Operaciones matemáticas con dos numeros) en C++ primero con módulos que no reciben ni devuelven valores, y después con funciones que reciben parámetros y entregan resultados.

```
#include <iostream.h>

                    //**Declaracion de los modulos
void captura (void);
void calcula (void);
void imprime(void);

                    //**Declaracion de Variables GLOBALES
int a,b,suma,resta,multiplicacion;

void main (void)
{
  captura();        //**Se invoca a los modulos
  calcula();
  imprime();
}

                    //**Codigo completo de los módulos
void captura (void)
{
   cout << "Introduce el primer numero: ";
   cin >> a;
  cout << "Introduce el segundo numero: ";
   cin >> b;
}

void calcula (void)
{
  suma = a + b ;
  resta = a – b ;
  multiplicacion = a * b;
}

void imprime (void)
{
   cout << "La suma es: " << suma <<endl;
   cout << "La resta es: " << resta << endl;
   cout << "La multiplicacion es: " << multiplicacion << endl;
}
```

```
#include <iostream.h>

          //**Declaracion de los modulos (FUNCIONES)
int  sumar (int x, int y);
int  restar (int x, int y);
int  multiplicar (int x, int y);

void main (void)
{
                    //**Declaracion de Variables LOCALES
   int a,b,suma, resta, multiplicacion;

                    //***Captura de los numeros
   cout << "Introduce el primer numero: ";
   cin >> a;
   cout << "Introduce el segundo numero: ";
   cin >> b;
                    //**Se invoca a los modulos
   suma = sumar(a,b);
   resta = restar(a,b);
   multiplicacion = multiplicar(a,b);
                    //**Se imprimen resultados
   cout << "La suma es: " << suma <<endl;
   cout << "La resta es: " << resta << endl;
   cout << "La multiplicacion es: " << multiplicacion << endl;
}

                    //***Codigo completo de los módulos
int sumar (int x, int y)
{  return x + y;
}

int restar (int x, int y)
{  return x - y;
}

int multiplicar  (int x, int y)
{  return x * y;
}
```

RECURSION

La recursión aparece en numerosas actividades de la vida diaria, por ejemplo, en una fotografía de una fotografía. Otro caso muy ilustrativo es el que se presenta en los programas de televisión en los cuales un periodista transfiere el control a otro periodista que se encuentra en otra ciudad, y este hace lo propio con un tercero.

La recursión permite definir un objeto en términos de sí mismo. Un subprograma que se llama a si mismo se dice que es recursivo. La recursión en los subprogramas puede darse de dos maneras:

a) **Directa**.- El subprograma se llama directamente a si mismo.
b) **Indirecta**.-El subprograma llama a otro subprograma, y este a su vez, llama al primero.

En toda definición recursiva de un problema se debe establecer un estado básico, es decir, un estado en el cual la solución no se presente de manera recursiva sino directamente. Además, la entrada (datos) del problema debe ir acercándose SIEMPRE al estado básico, de lo contrario, el programa puede llegar a ejecutarse indefinidamente creando resultados no deseados.

Si dada la definición de un problema es posible determinar el estado básico y el acercamiento paulatino al mismo, entonces es posible llegar a una solución. En otro caso se estaría en presencia de una definición circular del problema.

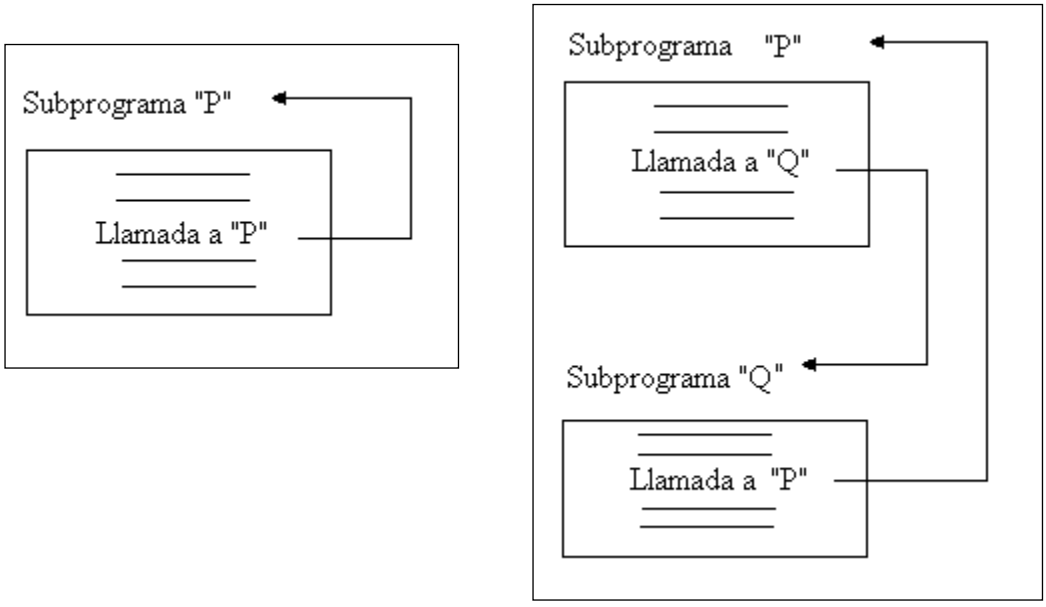


Figura: Recursión Directa y Recursión Indirecta

Ejemplo Recursivo: El Factorial de un número "n"

El factorial de un numero entero positivo "n" se define como el producto de los números comprendidos entre 1 y "n". La expresión n! simboliza el factorial de n. Por definición:

0! = 1
1! = 2
2! = 1 * 2
3! = 1 * 2 * 3
4! = 1 * 2 * 3 * 4
n! = 1 * 2 * 3 * 4 * ….. * (n-1) * n
Definiendo n! en función de (n-1)! Se tiene que:

n! = 1	si n = 0
n! = n * (n-1)!	si n>0

En la definición recursiva del factorial se da un estado básico (si n=0, entonces, factorial =1), en el cual el problema ya no se define en términos de sí mismo sino que se asigna a un valor directamente. En el paso recursivo de la fórmula se utiliza el concepto de factorial aplicado a (n-1). Por ser "n" un numero entero positivo, será (n-1) un valor mas cercano al estado básico (n=0).

Un algoritmo recursivo para calcular el factorial sería:

```
Factorial ( n )
{
        Si N = 0,       Hacer Factorial = 1
        Si no,          Hacer Factorial = N * factorial ( n-1 )
}
```

Funcionamiento Interno de la Recursión.-

Internamente se utiliza una estructura tipo pila para guardar los valores de las variables y constantes locales del programa o subprograma que efectúa la llamada. De esta manera, una vez concluida la ejecución, se puede seguir con los pasos que quedaron pendientes recuperando los valores necesarios de la pila. Con cada llamada recursiva se crea una copia de todas las variables y constantes que estén vigentes, y se guarda esa copia en la pila. Además se guarda una referencia a la siguiente instrucción a ejecutar. Al regreso se toma la imagen guardada en el tope de la pila y se continúa operando. Esta acción se repite hasta que la pila quede vacía. Todo este proceso es totalmente transparente para el programador.

Ventajas y desventajas de la Recursión.-

Usando recursión es posible escribir un código mas compacto, y en ciertos casos de mas fácil comprensión. Sin embargo, debe quedar claro que la recursión no da mayor rapidez a la ejecución del código y tampoco permite ahorrar espacio en memoria. También debe tenerse en cuenta que la recursión para su funcionamiento utiliza una pila interna la cual consume memoria.

En general es recomendable usar recursión cuando el problema por su naturaleza así lo exige, es decir, cuando la solución se simplifica y facilita de modo notable usando esta herramienta. En otro caso, es mas conveniente, computacionalmente hablando, utilizar alguna técnica iterativa para resolver el problema.

Anexo:
Prácticas propuestas

REPORTES DE LA MATERIA DE ESTRUCTURAS DE DATOS

Por cada práctica que se realice en la materia, se debe entregar un reporte escrito con la siguiente información de la práctica realizada:

1) Descripción de la práctica (En que consistió)
2) Antecedentes Teóricos (Conceptos y conocimientos necesarios para realizarla)
3) Diagrama de Flujo y/o lógica de solución (Pasos para resolver el problema)
4) Listado del programa (Código, programación)

RECOMENDACIONES PARA REALIZAR LOS REPORTES

- Incluir siempre una portada donde se indiquen los nombres de las personas que participaron en la elaboración del reporte.
- El reporte puede presentarse impreso en hojas blancas (en una carpeta o engargolado), o vía electrónica como archivo anexo (Email).
- Se prefiere que el alumno utilice su propio estilo para la Descripción y Antecedentes Teóricos; siempre cuidando la redacción.
- En la redacción, utilizar voz pasiva (impersonal) para describir lo realizado. Ejemplo: "Se ejecutó el programa" en lugar de: "Ejecuté el programa".
- Si se considera necesario, incluir figuras, diagramas adicionales, etc.
- Se recomienda que las prácticas y los reportes se realicen en equipos de dos personas.