EVALUACION DEL PERSONAL DOCENTE

CURSO: LENGUAJE C PARTE II, ESTRUCTURAS DE DATOS
FECHA: ABRIL 25 A MAYO 20 DE 1994.

CONFERENCISTA	DOMINIO DEL TEMA	USO DE AYUDAS AUDIOVISUALES	COMUNICACION CON EL ASISTENTE	PUNTUALIDAD
ING. JOSE A. CHAVEZ FLORES (COORD.)				
ING. JESSICA BRISEÑO				
ING. EDWIN NAVARRO PLIEGO				

EVALUACION DE LA ENSEÑANZA

1.- ORGANIZACION Y DESARROLLO DEL CURSO: ☐

2.- GRADO DE PROFUNDIDAD LOGRADO EN EL CURSO: ☐

3.- ACTUALIZACION DEL CURSO: ☐

4.- APLICACION PRACTICA DEL CURSO: ☐

EVALUACION DEL CURSO

CONCEPTO	CALIF.
CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
CONTINUIDAD EN LOS TEMAS	
CALIDAD DEL MATERIAL DIDACTICO UTILIZADO	

ESCALA DE EVALUACION: 1 A 10

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

LANGUAJE -C-
ESTRUCTURA DE DATOS

PARTE II

MATERIAL DIDACTICO

1994

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

LENGUAJE C
PARTE II
ESTRUCTURAS DE DATOS

TEMARIO

1. Arreglos y Apuntadores

2. Estructuras y uniones

3. Listas lineales

4. Recursividad

5. Arboles

6. Métodos de ordenamiento

7. Métodos de búsqueda

8. Manejo de archivos y métodos de acceso

9. Simulación

10. Algoritmos gráficos

LABORATORIO DE PREREQUISITOS

1. Escriba un programa que genere los primeros N números primos.

2. Escriba una función que convierta una cadena a mayúsculas.

3. Escriba un programa que lea una secuencia de nombres de personas, las almacene en un arreglo y se puedan hacer busquedas por nombre en el arreglo. El arreglo no necesita estar ordenado.

ARREGLOS Y APUNTADORES

ARREGLOS

Un arreglo es una colección de elementos del mismo tipo.

Un arreglo se define de la siguiente forma:

```
tipo nombre[tamaño];
```

El identificador de un arreglo es un apuntador constante, que guarda la dirección de inicio del arreglo.

Ejemplo:

```
int x[10];
```

Para accesar a los elementos de un arreglo hay que hacer referencia a su índice.

El primer elemento de un arreglo es el que tiene el índice cero.

En el ejemplo anterior, el primer elemento es x[0] y el último x[9].

Cuando se define un arreglo se reservan localidades continuas de memoria para almacenarlo, aún cuando el arreglo sea multidimensional.

Ejemplo:

```
/* Programa 1

     Este programa muestra el manejo de un arreglo en un
     programa.

*/

#include <stdio.h>
#define   N    100

main()
{
     int vector[N], i;

     for(i=0 ; i < N; i++)
          vector[i] = i;
     for(i=0 ; i < N; i++)
          printf("%c%3d", i%10 ? ' ' : '\n', vector[i]);
}
```

Inicialización de arreglos

Los arreglos externos y estáticos de enteros inicializan sus elementos con cero.

Los arreglos pueden inicializarse de forma explícita de la siguiente forma:

```
int  x[5] = { 2, 6, 8, 12, 28};
```

en este caso:

- El número de inicializadores puede ser menor que el número de elementos en el arreglo; en este caso los elementos restantes se inicializan con cero:

```
int x[10] = { 4, 5, 7};
```

- Es un error el que el número de inicializadores sea mayor que el tamaño del arreglo.

- Cuando se inicializa un arreglo no es necesario especificar su dimensión, se definición será de acuerdo al número de inicializadores:

```
int x[] = { 1, 5, 5, 7};
```

Arreglos de caracteres

En el lenguaje C no existe el tipo "string" o "cadena".

Una cadena puede ser representada con un arreglo de caracteres.

Para la manipulación de cadenas, por convención, el termino de esta se indica con el carácter '\0'.

Por lo tanto, para un arreglo de tamaño N, la longitud máxima de la cadena será de N-1.

Es responsabilidad del programador asegurarse de que no se excedan los límites de la cadena.

Las constantes cadena se escriben entre comillas, por ejemplo, "esto es una cadena", es un arreglo de caracteres de tamaño 19, donde el último elemento es '\0'.

Por lo tanto, las constantes "A" y 'a' no son iguales, la primera de ellas representa un arreglo de dos elementos y la segunda es un carácter.

La inicialización de arreglos de caracteres se puede hacer de una forma semejante a la inicialización de arreglos enteros:

```
char   cadena[] = { 'c', 'u', 'r', 's', 'o', '\0'};
```

o bien:

```
char   cadena[] = "curso";
```

Ejemplo:

```
/* Programa 2

     Este programa muestra el manejo de un arreglo de
     caracteres.

*/

#include <stdio.h>
#define   N    100

main()
{
     int  i=0;
     char nombre[N], c;

     printf("\tDame tu nombre: ");
     while ((c = getchar()) != '\n')
          nombre[i++] = c;
     nombre[i] = '\0';
     printf("\nMuchas gracias %s por haber dado tu nombre\n",
            nombre);
}
```

Arreglos multidimensionales

El lenguaje C permite la definición de arreglos multidimensionales, que en realidad son arreglos de arreglos.

En un arreglo multidimensional, las localidades de memoria que se reservan, al igual que en un arreglo unidimensional, son continuas.

La forma de definir un arreglo multidimensional, por ejemplo de dos dimensiones sería:

```
int  x[10][10];
```

Los arreglos multidimensionales más utilizados son aquellos de dos dimensiones, que permiten la representación de matrices. En este caso el primer índice representa los renglones y el segundo las columnas; sin embargo, esto no implica que el compilador maneje un arreglo de dos dimensiones como un conjunto de renglones y columnas. Este manejo depende totalmente del programador, de tal forma que se podría manipular el arreglo de tal forma que el primer índice representará a las columnas.

La inicialización de arreglos multidimensionales es muy parecida a la de los arreglos unidimensionales:

```
int x[3][3] = { { 3, 6, 9},
                { 8, 5, 1},
                { 1, 1, 5}};

int x[3][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};

int x[][3] = { 3, 6, 9, 8, 5, 1, 1, 1, 5};
```

Cuando se define un arreglo multidimensional, es necesario indicar todas sus tamaños a excepción del primero, de modo que el compilador pueda determinar la función correcta de transformación de almacenamiento.

La función de transformación de almacenamiento se utiliza para calcular la localidad de un elemento en base a sus índices, por ejemplo para un arreglo bidimensional, la función sería la siguiente:

```
dirBase + n*i + j
```

donde:

dirBase = dirección de inicio del arreglo

n = tamaño de la segunda dimensión

i = primer índice del elemento

j = segundo índice del elemento

Para un arreglo definido como: `int x[][10]`, el elemento `x[5][4]` se localizará `10*5 + 4` posiciones después del inicio del arreglo.

APUNTADORES

Todas las variables se almacenan en un cierto número de bytes a partir de cierta dirección de memoria en la máquina.

Un apuntador es una variables que almacena la dirección de memoria de otra variable.

El tipo de la variable para la cual se almacena la dirección puede ser cualquiera y determina el tipo de apuntador: apuntador a entero, apuntador a carácter, apuntador a apuntador, etc. Por ejemplo, para definir un apuntador a entero:

```
int      *p;
```

En este caso se esta definiendo una variable apuntador a entero; sin embargo, la dirección que contiene, hasta después de la definición es una dirección desconocida.

Para asignar a un apuntador una dirección válida se puede utilizar el operador de dirección &. Por ejemplo, supongase que x es una variable entera:

```
p = &x;
```

asigna a p la dirección de x y se puede accesar el valor de la variable x directamente o indirectamente por medio del apuntador p.

El apuntador de dirección & es unuario y solamente se aplica a variables.

Para poder hacer referencia a la dirección que contiene un apuntador, se utiliza el operador de desreferencia o indirección *. Por ejemplo, para cambiar el valor de la variable entera x en forma indirecta:

```
*p = 15;
```

en este caso *p es una variable entera y se puede utilizar en cualquier contexto que acepte valores enteros:

```
int x = 2, y, *p, *q;
double    d;

p = &x;
y = *p + 1;
d = sqrt(*p);

*p = 0;
*p += 1;
(*p)++;

q = p;
```

Los apuntadores se pueden inicializar al momento de ser definidos:

```
int  x = 4, *p = &x;
```

No se puede asignar a un apuntador una dirección de una variable que no es del tipo del apuntador:

```
int       *p;
double    f;

p = &f;      /* no es valido */
```

A un apuntador de le puede asignar la dirección de una localidad de memoria que se reserva al momento de ejecución del programa:

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

Apuntadores como parámetros de funciones

La forma de pasar parámetros a las funciones es por valor, esto implica, que la función no puede cambiar los valores almacenados de los parámetros actuales.

Para que una función cambie el valor de una variable a la cuál se puede hacer referencia desde la función que hace la llamada, es necesario que esta sea definida como externa:

Ejemplo:

```
/* Programa 3

     Este programa muestra el manejo de variables externas
     como una alternativa al paso de parámetros por referencia

*/

#include <stdio.h>
#define   N    10

int  max;
void  maximo(int,int);

main()
{
     int i, num;

     printf("Proporciona 10 numeros:\n");
     scanf("%d", &num);
     max = num;
     for (i=1; i < N; i++) {
          scanf("%d",&num);
          maximo(max, num);
     }
     printf("\nEl numero mayor es: %d\n", max);
}
```

```
void maximo(int x, int y) {

max = x > y ? x : y;

}
```

Otra forma de regresar un valor a la función que hace la llamada es con el uso de la sentencia return en una función.

Ejemplo:

```
/* Programa 3.1

     Este programa muestra el regreso de valores en la
     cláusula return.

*/

#include <stdio.h>
#define   N    10

int maximo(int,int);

main()
{
     int i, num, max;

     printf("Proporciona 10 numeros:\n");
     scanf("%d", &num);
     max = num;
     for (i=1; i < N; i++) {
          scanf("%d", num);
          max = maximo(max, num);
     }
     printf("\nEl numero mayor es: %d\n", max);
}

int maximo(int x, int y) {

return ( x > y ? x : y);
}
```

El uso de variables externas no es muy recomendable por la programación estructurada.

El uso de return solamente permite el regreso de un solo valor.

Muchas funciones necesitan regresar más de un valor.

Cuando se utilizan apuntadores como parámetros, el valor de las variables, direccionadas por el apuntador, puede cambiar en la llamada a una función.

Consideremos una función que intercambia el valor de sus dos parámetros:

Ejemplo:

```
/* Programa 4.1

     Programa que implementa la función swap sin manejo de
     apuntadores

*/

#include <stdio.h>

int swap(int,int);

main()
{
     int i = 10, y = 5;

     swap(i, y);
     printf("\nLos valores son i = %d y y = %d", i, y);
}

int swap(int a, int b) {
     int aux;

     aux = a;
     a = b;
     b = aux;
     printf("\nLos valores son a = %d y b = %d", a, b);
}
```

El mismo ejemplo, implementando paso de parámetros por referencia con el uso de apuntadores:

Ejemplo:

```
/* Programa 4.2

     Programa que implementa la función swap con manejo de
     apuntadores

*/

#include <stdio.h>

int swap(int *, int *);

main()
{
     int i = 10, y = 5;

     swap(&i, &y);
     printf("\nLos valores son i = %d y y = %d", i, y);
}

int swap(int *a, int *b) {
     int aux;

     aux = *a;
     *a = *b;
     *b = aux;
     printf("\nLos valores son a = %d y b = %d", *a, *b);
}
```

En el ejemplo:

- Se deben declarar los parámetros como apuntadores

- Hay que utilizar el operador de indirección en la definición de la función

- El parámetro actual en la llamada a la función es una dirección

Apuntadores y arreglos

Existe una gran relación entre apuntadores y arreglos.

El nombre de un arreglo es una variable donde se guarda la dirección de inicio del arreglo, es decir, es un apuntador constante.

Muchas veces los apuntadores y los arreglos son utilizados con el mismo propósito; sin embargo, cabe recordar que un apuntador es una variable y un arreglo es un apuntador constante.

Cuando se define un arreglo de tamaño N, se reservan N localidades continuas de memoria. Por otra parte, cuando se define un apuntador solamente se reserva el espacio para la variable que representa.

Suponga las siguientes definiciones:

```
int       x[10] = {1,2,3,4,5,6,7,8,9,10}, *p;
```

las siguientes expresiones son validas:

```
p = x;              /* es equivalente a p = &x[0] */

p =  x + 1;        /* utilizando notación de apuntadores */
                    /* p apunta a x[1]                    */

p++;                /* p apunta ahora a x[2] */

*(x + 5) = 10;      /* x[5] = 10             */

*(p + 10) = 15;     /* es válido pero se accesa una   */
                    /* localidad de memoria no válida */
```

la siguientes expresiones no son válidas:

```
x++;                /* x es un apuntador constante    */

*(x + 10) = 20;     /* x es la dirección de un arreglo */
                    /* y la localidad máxima que se    */
                    /* puede accesar es x + 9          */
```

¿ Cuál es la salida del siguiente programa ?

```
main() {

      char      x[] = "ESTA ES UNA CADENA EJEMPLO";
      char      *p = x,
                *q = x + 2;

      printf("%d %c %c %d %d %c",
                *p, p[0], *p + 1, *(p +5),  q[2] + 3, *x);

      p+=2;

      printf("%d %c %c", p[0], **&p + 5, *(p + 4));

      printf("%c %c %c", *(p++), *p, *(p + 1));
}
```

Arreglos como parámetros de funciones

Cuando un arreglo es pasado como parámetro a una función, en realidad se esta pasando una dirección, ya que el nombre de un arreglo es la dirección del primer elemento del mismo.

El parámetro formal puede ser declarado como arreglo a como apuntador.

En el caso de que el parámetro formal sea declarado como un arreglo unidimensional no es necesario especificar el tamaño, en este caso es responsabilidad del programador no rebasar los límites del arreglo.

En el caso de arreglos multidimensionales es necesario especificar todas las dimensiones a excepción de la primera, esta no es necesaria ya que no se utiliza en la formula de transformación de direcciones.

En el caso de arreglos multidimensionales, las dimensiones especificadas en el parámetro formal pueden no ser las mismas que las del parámetro actual. La función lo único que recibe es una dirección de inicio e información para accesar los elementos por medio de la formula de mapeo.

¿ Es correcto el siguiente programa ?

¿ Cuál es la salida ?

```
#include <stdio.h>

void  f(int [][3]);

main() {

     int   matriz[4][4] = { { 1, 2, 3, 4 },
                            { 5, 6, 7, 8 },
                            { 9, 10, 11, 12 },
                            { 13, 14, 15, 16}};

     f(matriz);

}

void f(int a[][3]){

     int  i;

     for(i=0; i < 5; i++)
          printf("%d ", a[i][2]);
}
```

Aritmética de apuntadores

La aritmética de apuntadores es una de las características más eficaces del lenguaje C.

Si p es un apuntador a un tipo de dato e inicialmente tiene una dirección x, por ejemplo la dirección 1876; la expresión p + 1 no es necesariamente la dirección 1877.

El incremento no es unitario necesariamente, sino que depende del número de bytes que se necesiten para almacenar un elemento del tipo al cual direcciona el apuntador.

En el caso de que p fuera una apuntador a int y que el tipo int ocupará 2 bytes p++ ocasionaría que p apuntara dos bytes adelante de su dirección original.

Se pueden manejar sumas y restas de direcciones exclusivamente.

En forma general se puede decir que la aritmética de apuntadores no es igual a la de enteros.

Funciones para manejo de caracteres y cadenas

La siguiente tabla muestra una serie de funciones que permiten determinar la naturaleza de un carácter, todas ellas reciben como parámetro un valor numérico o char.

Para poder utilizar estas funciones es necesario se debe especificar el header ctype.h.

Todas estas funciones regresan como valor cero, si el carácter es del tipo que se esta validando y un valor diferente de cero en cualquier otro caso.

NOMBRE DE FUNCION	PROPOSITO
isalnum(int c)	Determina si c es alfanumérico
isalpha(int c)	Determina si c es letra
iscntrl(int c)	Determina si c es carácter de control
isdigit(int c)	Determina si c es dígito
isprint(int c)	Determina si c es imprimible
islower(int c)	Determina si c es minúscula
isspace(int c)	Determina si c es blanco
isupper(int c)	Determina si c es mayúscula

El lenguaje C cuenta con un conjunto de funciones para manejo de cadenas.

Las cadenas son un arreglo de caracteres terminados con '\0'.

Se debe de indicar el header string.h.

strcpy

```
char *strcpy( char *s1, char *s2)
```

Copia s2 a s1, incluye en s1 el carácter '\0'

Regresa como valor s1

strncpy

```
char *strncpy( char *s1, char *s2, int n)
```

Copia n caracteres de s2 a s1, incluye en s1 el carácter '\0'

Regresa como valor s1

strcat

```
char *strcat( char *s1, char *s2)
```

Concatena s2 a s1 incluye en s1 el carácter '\0'

Regresa como valor s1

strncat

```
char *strncat( char *s1, char *s2, int n)
```

Concatena n caracteres de s2 a s1 incluye en s1 el carácter '\0'.

Regresa como valor s1.

strcmp

```
int strcmp( char *s1, char *s2)
```

Compara las cadenas s1 y s2, no compara la longitud de ellas, sino el orden lexicográfico de cada uno de sus caracteres.

```
Regresa un valor menor a cero si s1 < s2,
                 mayor a cero si s1 > s2,
                 cero si s1 = s2
```

strncmp

```
int strncmp( char *s1, char *s2, int n)
```

Compara a lo más n caracteres de las cadenas s1 y s2

```
Regresa un valor menor a cero si s1 < s2,
                 mayor a cero si s1 > s2,
                 cero si s1 = s2
```

strchr

```
char *strchr( char *s1, int c)
```

Busca la primera ocurrencia del carácter c en s1

Regresa la dirección de la primera ocurrencia de c en s1
Regresa NULL si c no esta en s1

strlen

```
int strlen( char *s1)
```

Calcula la longitud de s1 no contando el terminador

Regresa la longitud de la cadena

A continuación se muestra la implementación de algunas de ellas:

```
int strcmp(char *s, char *t)
{
     for( ; *s == *t ; s++, t++)
          if (*s ==  '\0')
               return 0;
     return *s - *t;
}


char *strcat(char *s1, char *s2)
{
     char      *aux = s1;

     while(*aux++)
          ;
     --aux;
     while (*aux++ = *s2++)
          ;
     return s1;
}


int strlen(char *s)
{
     int  n=0;

     while(*s++)
          n++;
     return n;
}
```

Arreglos de apuntadores

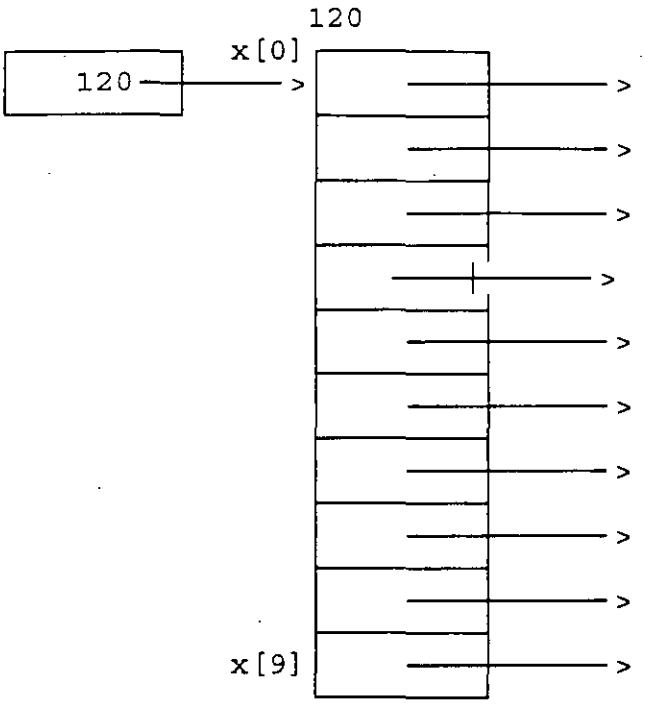
El lenguaje C permite la definición de tipos a partir de los ya definidos, de esta forma se pueden crear arreglos de apuntadores, arreglos de arreglos de apuntadores, etc.

Los arreglos de apuntadores son mucho más flexibles que los arreglos multidimensionales, además de que requieren de menos memoria generalmente.

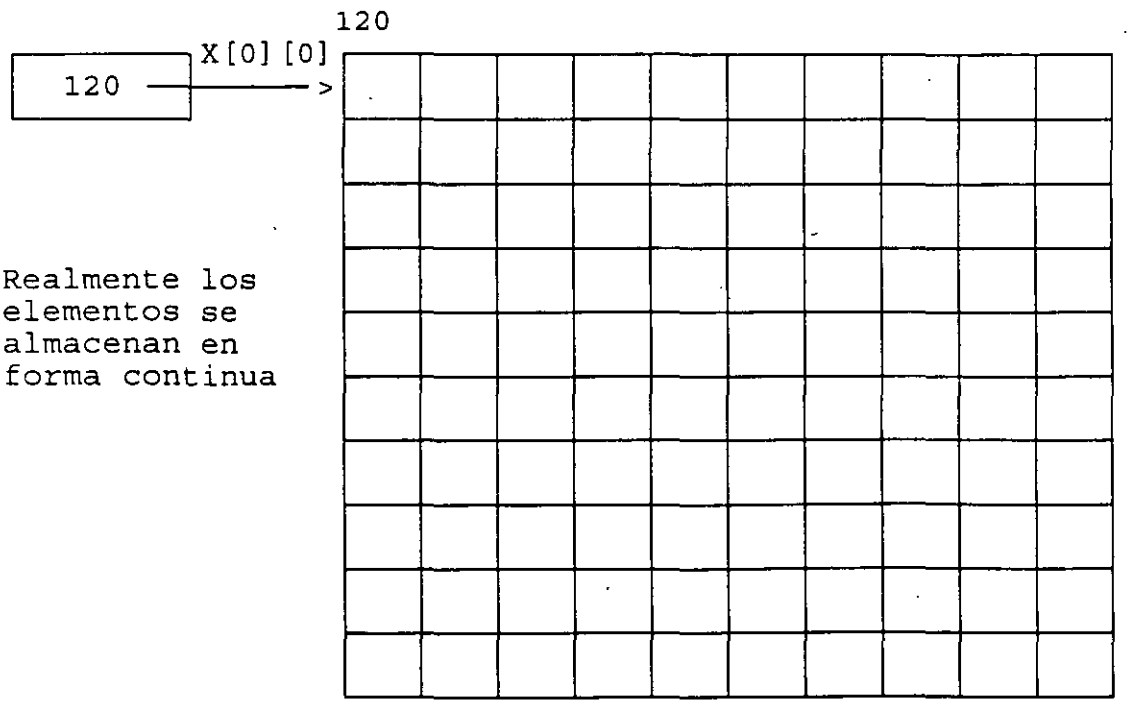
La forma de definir un arreglo de apuntadores a carácter sería de la siguiente:

```
char            *x[10];
```

La representación interna de este arreglo es muy diferente a la de un arreglo bidimensional de caracteres, aunque la forma de hacer referencia a cada uno de sus elementos sea muy parecida.



De inicio los apuntadores apuntan a direcciones no válidas



Realmente los elementos se almacenan en forma continua

Una vez creado el arreglo de apuntadores, se deberá asignar a cada uno de los elementos direcciones de memoria previamente reservadas, esto se puede hacer dinamicamente a tiempo de ejecución de la forma siguiente:

```
x[0] = (char *)malloc(sizeof(N));
```

De esta forma cada uno de los elementos se puede considerar como un arreglo de N caracteres y se puede hacer referencia a cada elemento de un arreglo con la notación utilizada para arreglos bidimensionales.

Ejemplo:

```
/*
     Programa 5

          Este programa muestra el manejo de arreglos de
          apuntadores.

          Se leen n cadenas de caracteres y se reserva memoria a
          tiempo de ejecución.

*/

#include <stdio.h>

#define        MAX        20    /* Máximo número de elementos en */
                                /* el arreglo                    */
#define        MAX_NOM    35    /* Máximo número de caracteres   */
                                /* una cadena                    */

void ordena(char *[], int);
void imprime(char *[], int);

main() {

     char       *agenda[MAX],
                nombre[MAX_NOM];
      int i=0,j;

     printf("\nProporciona el nombre de tus amigos:\n");
      while (gets(nombre) != NULL) {
            agenda[i] = (char *)malloc(sizeof(strlen(nombre)+1));
            strcpy(agenda[i], nombre);
            i++;
      }
      ordena(agenda, i);
     printf("\n\nLista Ordenada:\n");
      imprime(agenda, i);
}
```

```
void ordena(char *x[], int n) {

     int  i,j;
     char aux[MAX_NOM];

     for(i=0; i < n - 1; i++)
          for (j = n - 1; i<j; j--)
               if ((strcmp(x[j-1], x[j])) > 0) {
                         strcpy(aux,x[j-1]);
                         strcpy(x[j-1],x[j]);
                         strcpy(x[j],aux);
               }
}

void imprime(char *x[], int n) {

     int       i;

      for(i=0; i < n; i++)
            printf("%s\n", x[i]);
}
```

Parámetros para main

En la función main se pueden utilizar dos parámetros para establecer comunicación con el sistema operativo al momento que se lleva a cabo la ejecución del programa.

Los argumentos se llaman argc y argv. El primero de ellos almacena el número de argumentos en la línea de comandos que recibe el programa ejecutable y el segundo es una arreglo de apuntadores a char en donde se almacenan dichos argumentos.

El primer elemento argv[0] contiene el nombre del programa ejecutable.

Ejemplo:

```
/*
     Programa 6

          Este programa despliega los parámetros que se le pasan en
          la línea de comandos

*/

#include <stdio.h>

main(int argc, char **argv) {

     int i;

     for(i=0 ; i < argc; i++)
          printf("%s ", argv[i]);
}
```

ESTRUCTURAS Y UNIONES

TYPEDEF

C es un lenguaje que puede ampliarse con facilidad.

La extensión del lenguaje se puede llevar a cabo mediante los #define y creando funciones de propósito general para uso de todos los usuarios.

También puede ampliarse al definir tipos de datos que se construyen con los tipos estándar.

También se pueden definir tipos con componentes no homogéneos con el uso de estructuras.

C proporciona diversos tipos fundamentales, como char, int, double, etc. y varios tipos derivados como arreglos y apuntadores; también proporciona la declaración typedef, que permite la asociación explícita de un tipo con un identificador.

Ejemplos:

```
typedef   int        ENTERO;
typedef   char       CARACTER;
typedef   ENTERO     VECTOR[10];
typedef   CARACTER   *STRING;
typedef   VECTOR     MATRIZ[10];
```

Estos tipos definidos se pueden utilizar para la definición de variables o funciones, de la misma forma en como se utilizan los tipos estándar:

```
STRING    lista[N];
MATRIZ    a,b,c;
```

Las definiciones de tipo permiten la documentación de programas, ya que normalmente las definiciones de tipo junto con los prototipos y las definiciones de símbolos con #define se colocan en archivos header.

Además, cuando existen declaraciones sensibles al sistema, como el caso de int, que en los sistemas UNIX es de cuatro bytes y en otros es de dos, y si estas diferencias son críticas para el programa, el empleo de typedef permite que los programas sean transportables.

ESTRUCTURAS

Las estructuras permiten la representación de elementos de diferentes tipos por medio de un identificador.

Por ejemplo, suponga que se quiere representar mediante una estructura de datos la información de un empleado:

- Número de cuenta (ej. 1287BDG)
- Nombre
- Dirección
- Teléfono
- Sexo
- Tipo (V = vendedor, A = administrativo,
  T = técnico, D = directivo)
- Salario

Se podría utilizar una estructura para definir un tipo que representara a un empleado:

```
struct emp {
     char       noCta[8],
                nombre[35],
                dirección[35],
                teléfono[10],
                sexo,
                tipo;
     float      salario;
};
```

De esta forma se pueden definir variables de tipo struct emp, así como arreglos de esé tipo:

```
struct  emp   lista[N];
struct  emp   empleado1;
```

Se puede hacer uso de typedef para hacer una definición de tipo más manejable:

```
typedef struct {
     char       noCta[8],
                nombre[35],
                dirección[35],
                teléfono[10],
                sexo,
                tipo;
     float      salario;
} EMPLEADO;
```

Con la definición anterior las definiciones de variables serían:

```
EMPLEADO   lista[N];
EMPLEADO   empleado1;
```

En la definición de el tipo EMPLEADO, la etiqueta emp después de struct es opcional, cuando se coloca permite que se pueda utilizar el tipo struct emp para la definición de variables. Es necesaria la etiqueta cuando la estructura tiene elementos del tipo que se esta definiendo.

Los nombres de los miembros de la estructura son únicos dentro de ella.

La forma de accesar los elementos de una estructura es por medio del operador miembro ".".

Una construcción de la forma:

variable_estructura.nombre_miembro

se utiliza como variable de la misma forma que se utiliza una variable simple o un elemento del mismo tipo.

Por ejemplo, para asignar valores a la variable empleado de tipo EMPLEADO:

```
strcpy(empleado.noCta, "811CAFA");
strcpy(empleado.nombre,"C. Jéssica Briseño C.");
empleado.salario = 6500;
```

El valor de una estructura se puede asignar directamente, por ejemplo, si empleado y empleado1 son estructuras del mismo tipo:

empleado1 = empleado

Las funciones pueden recibir parámetros de algún tipo de estructura o regresar el valor de alguna estructura.

Ejemplo:

```
/*  Programa 1

        Programa que implementa el manejo de números complejos
        con el uso de estructuras.

*/

#include <stdio.h>

typedef struct{
        float     real,
                  imaginaria;
} COMPLEJO;

COMPLEJO  suma(COMPLEJO, COMPLEJO);
COMPLEJO  resta(COMPLEJO, COMPLEJO);

main() {

     COMPLEJO  a,b,c;

     printf("\n\nProporciona dos numeros complejos:\n");
     scanf("%f",&a.real);
     scanf("%f",&a.imaginaria);
     scanf("%f",&b.real);
     scanf("%f",&b.imaginaria);
     c = suma(a,b);
     printf("\nLa suma de los numeros:"
            "\n%5.2f + %5.2fi\n%5.2f + %5.2fi\nes:\n\n"
            "%5.2f + %5.2fi\n",
            a.real, a.imaginaria, b.real, b.imaginaria,
            c.real, c.imaginaria);
     c = resta(a,b);
     printf("\n\ny la resta:\n\n%5.2f + %5.2fi\n",
            c.real, c.imaginaria);
}
```

```
COMPLEJO  suma(COMPLEJO  x, COMPLEJO y) {

          COMPLEJO  z;

          z.real = x.real + y.real;
          z.imaginaria = x.imaginaria + y.imaginaria;
          return z;
}

COMPLEJO  resta(COMPLEJO  x, COMPLEJO y) {

          COMPLEJO  z;

          z.real = x.real - y.real;
          z.imaginaria = x.imaginaria - y.imaginaria;
          return z;
}
```

Inicialización de estructuras

Las estructuras pueden ser inicializadas de una forma muy parecida a como se inicializan los arreglos:

```
EMPLEADO  empleado = { "811CAFA",
                       "C. Jéssica Briseño C.",
                       "Norte 86B 4729",
                       "379-00-00",
                       'F',
                       'D',
                       6500 };
```

Arreglos de estructuras

El lenguaje C permite la creación de arreglos de elementos cuyos tipos pueden ser cualquiera previamente definido, por ejemplo, se pueden definir arreglos de estructuras:

```
EMPLEADO        nomina[N];
```

Así, para poder leer una lista de información de empleados:

```
for (i = 0; i < N; i++) {
          printf("Num. de Cta.: ");
          gets(nomina[i].noCta);
          printf("Nombre: ");
          gets(nomina[i].nombre);
          printf("Dirección: ");
          gets(nomina[i].dirección);
          printf("Teléfono: ");
          gets(nomina[i].teléfono);
          printf("Sexo: ");
          scanf("%c",&nomina[i].sexo);
          printf("Tipo: ");
          scanf("%c",&nomina[i].tipo);
          printf("Salario: ");
          scanf("%f",&nomina[i].salario);
}
```

LABORATORIO 1

1. Implemente una agenda que permita almacenar nombre y dirección de sus amigos.

2. Modifique el programa de agenda para que se puedan hacer consultas.

UNIONES

Una unión al igual que una estructura, es un tipo derivado.

Las uniones tienen la misma sintaxis que las estructuras, pero comparten el almacenamiento.

Una unión define a un conjunto de valores alternos que pueden almacenarse en una porción compartida de la memoria.

El compilador asigna una porción de almacenamiento que pueda acomodar al más grande de los miembros especificados.

La notación para accesar a un miembro de una unión es idéntica a la que emplean las estructuras.

El sistema interpreta los valores almacenados de acuerdo al miembro seleccionado, elegir el correcto es responsabilidad del programador.

Ejemplo:

Suponga que se quiere representar la información de los empleados de una empresa; sin embargo, existen diferentes tipos de empleados (vendedor, administrativo, directivo, técnico) y para el calculo de la nómina es importante considerar todos los puntos que influyen en el cálculo de las percepciones mensuales. Los técnicos y directivos reciben un sueldo mensual y un bono adicional, los vendedores perciben además de su sueldo base comisiones y premios, a los administrativos se les paga las horas extras.

Los tipos utilizados serían los siguientes:

```
typedef struct {
     float     sueldoBase;
     float     bono;
}  CONFIANZA;

typedef struct {
     float     sueldoBase;
     int       horasExt;
}  ADMON;

typedef struct {
     float     sueldoBase;
     float     comision;
     float     premio;
}

typedef union {
     CONFIANZA  confianza;
     ADMON      admon;
     VENDEDOR   vendedor;
}  SALARIO;
```

La estructura EMPLEADO se definiría entonces de la siguiente forma:

```
typedef struct {
     char       noCta[8],
                nombre[35],
                dirección[35],
                teléfono[10],
                sexo,
                tipo;
     SALARIO    salario;
} EMPLEADO;
```

De esta forma en base al campo tipo se podría determinar como manejar la unión:

```
EMPLEADO        emp;

...

if (emp.tipo = 'V') {
     emp.salario.vendedor.comision = 1000;
     emp.salario.vendedor.premio = 500;
}

...
```

LABORATORIO 2 (OPCIONAL)

1. Modifique el programa agenda del laboratorio anterior, para que cuando se trate de registrar amigas se almacene su teléfono y fecha de nacimiento y cuando sean amigos el teléfono de su oficina.

APUNTADORES A ESTRUCTURAS

Se pueden definir apuntadores a estructuras para poder referenciarlas indirectamente.

Desde un apuntador también se pueden accesar los miembros de una estructura.

Los apuntadores a estructuras se pueden utilizar para manejar paso de parámetros por referencia cuando se utilizan estructuras como parámetros.

Los apuntadores de estructuras son la base de la implementación más eficiente de estructuras de datos como pilas, listas lineales, gráficas y árboles.

La forma natural de accesar los miembros de una estructura por medio de un apuntador es un poco confusa:

```
EMPLEADO  *p;

...

(*p).tipo = 'D';
```

Debido a que son operaciones muy utilizadas, se utiliza el operador "->":

```
EMPLEADO  *p;

...

p->tipo = 'D';
```

Ejemplo:

```
/*  Programa 2

        Programa que implementa el manejo de números complejos
        con apuntadores a estructuras.

*/

#include <stdio.h>
#define         SIGNO(x)   ((x) >= 0) ? '+' : ' '

typedef struct{
          float     real,
                    imaginaria;
} COMPLEJO;

COMPLEJO   *suma(COMPLEJO, COMPLEJO, COMPLEJO *);
COMPLEJO   *resta(COMPLEJO, COMPLEJO, COMPLEJO *);

main() {

     COMPLEJO  a,b,*c;

     c = (COMPLEJO *)malloc(sizeof(COMPLEJO));

     printf("\n\nProporciona dos numeros complejos:\n");
     scanf("%f",&a.real);
     scanf("%f",&a.imaginaria);
     scanf("%f",&b.real);
     scanf("%f",&b.imaginaria);
     suma(a,b,c);
     printf("\nLa suma de los numeros:"
                "\n%5.2f %c %5.2fi\n%5.2f %c %5.2fi\nes:\n\n"
                "%5.2f %c %5.2fi\n",
                a.real, SIGNO(a.imaginaria), a.imaginaria,
                b.real, SIGNO(b.imaginaria), b.imaginaria,
                c->real, SIGNO(c->imaginaria), c->imaginaria);
      c = resta(a,b, c);
      printf("\n\ny la resta:\n\n%5.2f %c %5.2fi\n",
                c->real, SIGNO(c->imaginaria), c->imaginaria);
}
```

```
COMPLEJO   *suma(COMPLEJO  x, COMPLEJO y, COMPLEJO *z) {

           z->real = x.real + y.real;
           z->imaginaria = x.imaginaria + y.imaginaria;
           return z;
}

COMPLEJO   *resta(COMPLEJO  x, COMPLEJO y, COMPLEJO *z) {

           z->real = x.real - y.real;
           z->imaginaria = x.imaginaria - y.imaginaria;
           return z;
}
```

RESUMEN DE OPERADORES

Operadores	Asociatividad
() [] ->	izquierda a derecha
~ ! ++ -- sizeof -(unario) *(indirección) &(dir.)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
?:	derecha a izquierda
= += -= *= etc.	derecha a izquierda
, (operador coma)	izquierda a derecha

FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

LENGUAJE "C", PARTE II

ESTRUCTURA DE DATOS

LISTAS LINEALES

LISTAS NO LINEALES
ARBOLES

1994.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

LISTAS LINEALES

Las listas son estructuras de datos que dan mayor flexibilidad de programación a los usuarios, con un ahorro considerable de memoria por las operaciones que pueden practicarse sobre ella.

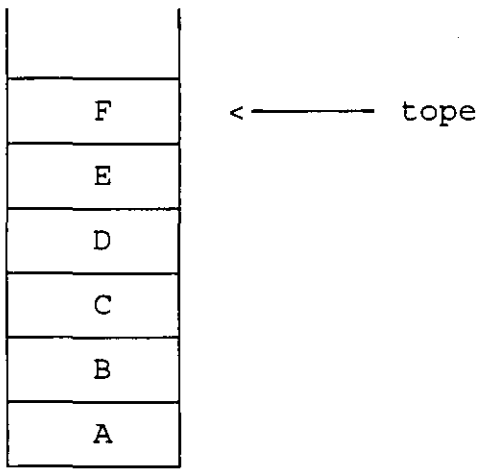
Una lista es una estructura de datos que tiene un número variable de nodos.

Una lista lineal, es una lista cuyos nodos están ordenados por un solo criterio, en donde el último y el primer nodo no tienen sucesor y antecesor respectivamente.

PILA

Una pila o stack es una estructura de datos lineal, en la cual las operaciones se realizan por uno de los extremos de la lista.

Una pila se puede representar mediante la siguiente figura:

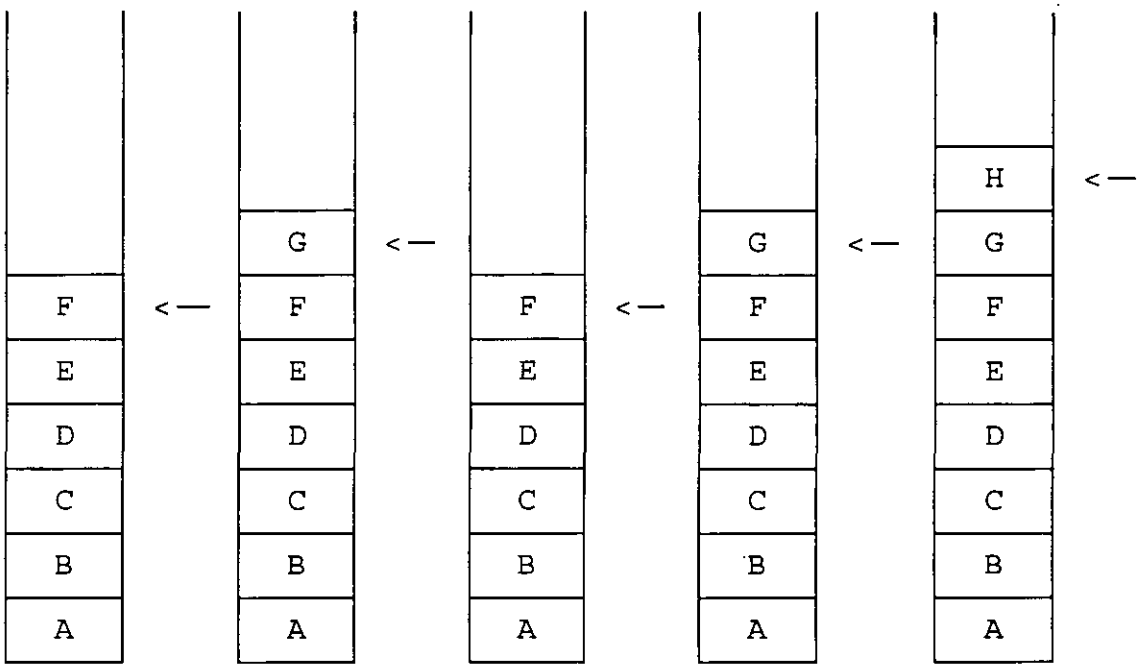


Uno de los extremos de la pila se designa como el tope de la misma y es por donde se colocan nuevos elementos o se retiran.

En el caso de que se agreguen nuevos elementos a la pila, el tope se moverá hacia arriba para indicar al último elemento en entrar a la pila.

En el caso de que se retire un elemento de la pila el tope se mueve hacia abajo, para indicar hacia el nuevo elemento que se encuentra en el extremo de la pila.

En la siguiente figura se muestra el comportamiento de la pila al insertar y borrar elementos de ella.



Por la forma en que se agregan y retiran elementos de la pila, el método ha sido llamado LIFO (las input first output). esto significa que solamente puede ser retirado de la pila el último elemento agregado.

Las operaciones que se llevan a cabo sobre una pila se conocen como **push** (para insertar) y **pop** (para borrar un elemento).

Cuando la pila contiene un solo elemento y se lleva a cabo una operación de pop, la pila resultante no contiene elementos y se llama **pila vacía**.

Aunque la operación push es aplicable teóricamente en cualquier momento, no ocurre lo mismo con la operación pop, la cual no puede aplicarse a una pila vacía.

Representación de pilas en C

Antes de programar la solución de un problema que use una pila, debe decidirse como representarla mediante las estructuras de datos existentes en nuestro lenguaje de programación.

Como se verá, hay varias maneras de representar una pila en el lenguaje C. Consideremos ahora la más sencilla: un arreglo.

Sin embargo, una pila y un arreglo son dos cosas diferentes. El número de elementos de una arreglo es fijo y se asigna en la definición. Por otra parte, una pila es un objeto dinámico cuyo tamaño cambia constantemente mediante las operaciones push y pop.

Si se quiere implementar una pila con un arreglo se deberá definir un arreglo lo bastante grande para admitir el tamaño máximo de la pila. Es necesario otro elemento para guarde la posición del elemento tope de la pila.

Por lo tanto puede declararse una pila, como una estructura que contiene dos elementos: una arreglo para guardar los elementos de la pila y un entero para guardar la posición del elemento tope de la pila:

```
# define STACKSIZE  100

typedef   struct {
                int  tope;
                int  elementos[STACKSIZE];
}    STACK;
```

Para definir una pila:

```
STACK     s;
```

Las funciones push y pop se definirían de la siguiente forma:

```
void push(STACK *s, int dato) {

if ( s->tope == STACKSIZE )
     printf("Pila llena");
else
     s->elementos[s->tope++] = dato;
}


int  pop(STACK *s) {

if ( s->tope == 0 )
     printf("Pila vacía");
else
     return s->elementos[s->tope--];
}
```

Para que no existiera restricción en cuanto al tipo de elementos en la pila se podría definir de la siguiente forma:

```
# define STACKSIZE  100

typedef   union {
          int   intEle;
          float floatEle;
          char  charEle;
} ELEMENTO;

typedef   struct {
               int  tope;
               ELEMENTO  elementos[STACKSIZE];
}    STACK;
```

Pilas dinámicas

Una de las características principales de una pila es que su tamaño es dinámico, es decir su tamaño no esta previamente definido.

Para definir una pila dinámica utilizaremos la siguiente estructura:

```
typedef    struct s {
           int  dato;
           struct s *liga;
} ELEMENTO;

typedef  ELEMENTO   *STACK;
```

De esta forma se puede definir una pila como:

```
STACK      pila= NULL;
```

Inicialmente la pila esta vacía y el único espacio en memoria ocupado es el del apuntador.

Para implementar las funciones push y pop se deben considerar los siguientes puntos:

- La pila esta vacía cuando la variable de tipo STACK tiene como valor NULL.
- Cuando se lleva a cabo la operación pop, se deberá liberar la memoria ocupada por el elemento saliente.
- Cuando se lleva a cabo la operación push se deberá alojar memoria para el elemento que se va a colocar en el tope de la pila.
- Idealmente no existe límite en cuanto al número de elementos que pueden entrar a la pila.

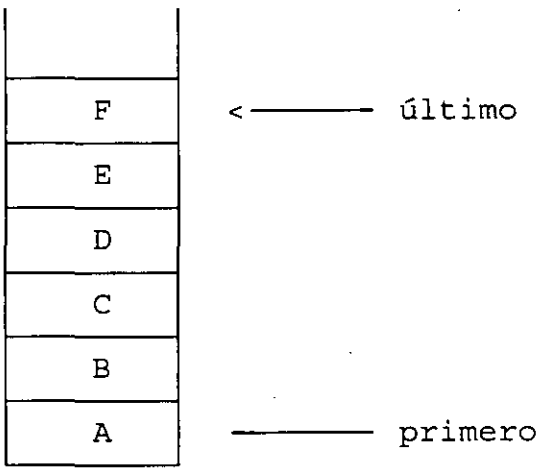
La implementación de estas funciones sería:

```
void push(STACK *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
aux->liga = *s;
*s = aux;
}


int  pop(STACK *s) {

int       dato;
ELEMENTO  *aux;

if ( *s == NULL )
     printf("Pila vacía");
else {
     dato = (*s)->dato;
     aux = *s;
     *s = (*s)->liga;
     free(aux);
     return dato;
}
}
```

COLA

Una cola es una estructura de datos lineal, en la cual la operación de inserción de un elemento se realiza por uno de los extremos de la lista y la extracción por el otro.

Una cola se puede representar mediante la siguiente figura:

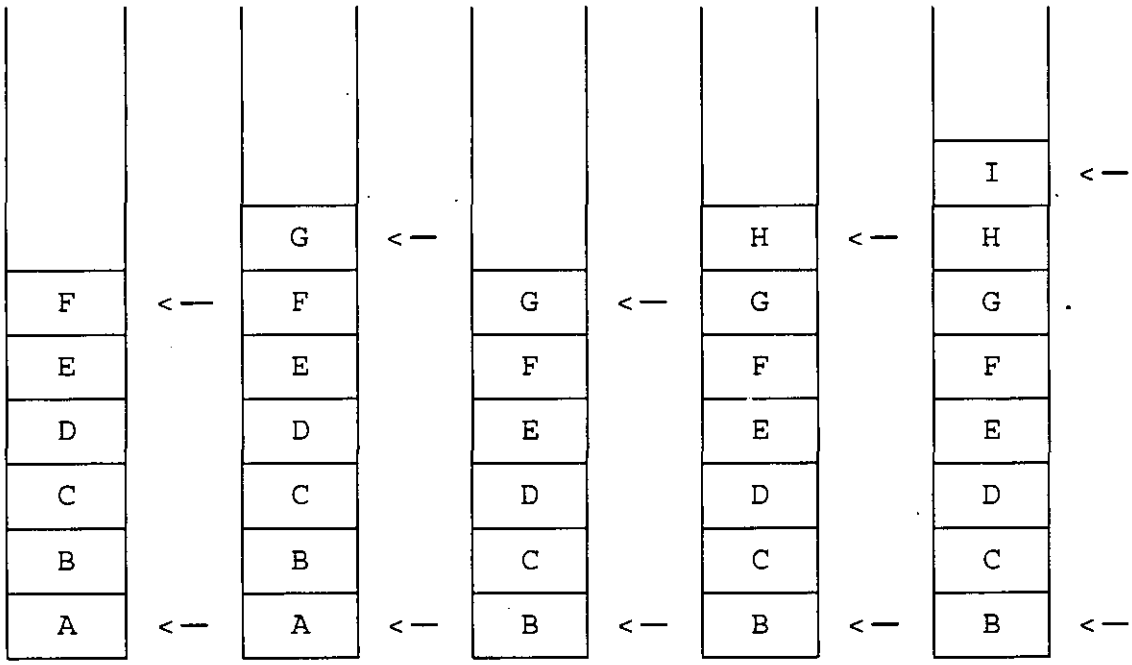


Uno de los extremos de la cola se designa como el último de la misma y es por donde se colocan nuevos elementos.

En el caso de que se agreguen nuevos elementos a la cola, este elemento será el último de la cola.

En el caso de que se retire un elemento de la cola, este elemento será el que entro antes que todos los elementos actuales de la cola. Al retirar al primer elemento de la cola, se mueven hacia abajo los demás elementos.

En la siguiente figura se muestra el comportamiento de la cola al insertar y borrar elementos de ella.



Por la forma en que se agregan y retiran elementos de la cola, el método ha sido llamado FIFO (first input first output). esto significa que solamente puede ser retirado de la cola el primer elemento agregado.

Las operaciones que se llevan a cabo sobre una cola se conocen como **insertar** y **retirar**.

Al igual que con la pila, las formas de representar una cola son muchas; sin embargo debido a que una de las características principales de una cola es que su tamaño es dinámico, consideraremos únicamente esta forma de implementación.

Para definir una cola dinámica utilizaremos la misma estructura que para la pila:

```
typedef   struct s {
          int  dato;
          struct s *liga;
} ELEMENTO;

typedef  ELEMENTO   *COLA;
```

De esta forma se puede definir una cola como:

```
COLA   cola= NULL;
```

Inicialmente la cola esta vacía y el único espacio en memoria ocupado es el del apuntador.

Para implementar las funciones insertar y retirar se deben considerar los siguientes puntos:

- La cola esta vacía cuando la variable de tipo COLA tiene como valor NULL.
- Cuando se lleva a cabo la operación retirar, se deberá liberar la memoria ocupada por el elemento saliente.
- Cuando se lleva a cabo la operación insertar se deberá alojar memoria para el elemento que se va a colocar en uno de los extremos de la cola.
- Idealmente no existe límite en cuanto al número de elementos que pueden entrar a la cola.

La implementación de estas funciones sería:

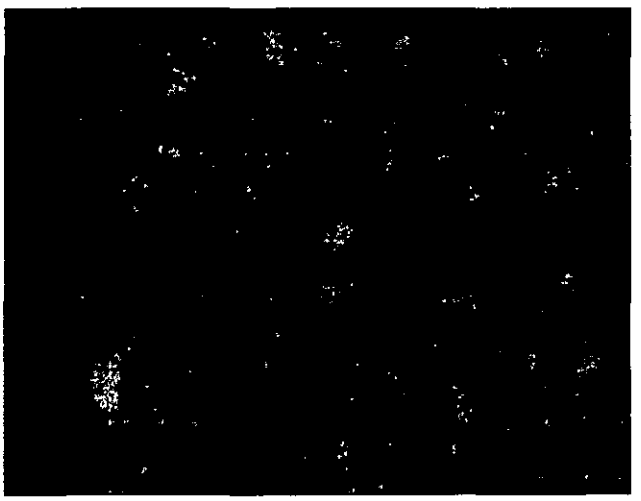
```
void inserta(COLA *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
aux->liga = *s;
*s = aux;
}


int  extrae(COLA *s) {

int       dato;
ELEMENTO  *aux;

if ( *s == NULL )
     printf("Cola vacía");
else {
     aux = *s;
     while ( aux->liga != NULL )
          aux = aux->liga;
     dato = aux->dato;
     if ( aux == *s)
          *s = NULL;
     free(aux);
     return dato;
}
}
```

LISTA CIRCULAR

Una lista circular es aquella estructura de datos que tiene como característica fundamental un orden en el que, al último elemento le sigue el primero.

Las operaciones que se definen sobre la lista circular son las de insertar y extraer y su comportamiento depende de su manejo puede ser como cola o como pila.

Una cola se puede representar mediante la siguiente figura:



Lista Circular

La implementación de una lista circular, considerando la representación como la de una cola es la siguiente:

```
typedef    struct s {
           int  dato;
           struct s *liga;
} ELEMENTO;

typedef  ELEMENTO   *COLA;
```

De esta forma se puede definir una cola como:

```
COLA   cola= NULL;
```

Para implementar las funciones insertar y retirar se deben considerar los siguientes puntos:

- La cola esta vacía cuando la variable de tipo COLA tiene como valor NULL.
- Cuando la cola tiene un solo elemento este apunta así mismo.

La implementación de las funciones sería:

```
void inserta(COLA *s, int dato) {

ELEMENTO  *aux;

aux = (ELEMENTO *)malloc(sizeof(ELEMENTO));
aux->dato = dato;
if ( *s == NULL)
     aux->liga = aux;
else {
     aux->liga = *s;
     while ( (*s)->liga != aux->liga)
          *s = (*s)->liga;
     (*s)->liga = aux;
}
*s = aux;
}
```

```
int  extrae(COLA *s) {

int       dato;
ELEMENTO  *aux;

if ( *s == NULL )
     printf("Cola vacía");
else {
     aux = *s;
     if ( aux->liga == *s) {
          dato = aux->dato;
          free(aux);
          *s = NULL;
     } else {
          while ( aux->liga->liga != *s )
               aux = aux->liga;
          dato = aux->liga->dato;
          free(aux->liga);
          aux->liga = *s;
     }
     return dato;
}
}
```

LISTAS NO LINEALES
ARBOLES

En el capítulo anterior se expuso que una lista lineal es una estructura de datos que expresa las relaciones entre los nodos por un solo criterio o en una sola dimensión.

Las listas no lineales son estructuras de datos más complejas, cuyas relaciones entre sus nodos son en más de una dimensión.

Una de las listas no lineales más utilizadas, principalmente en la implementación de sistemas operativos y DBMS's, son los árboles.

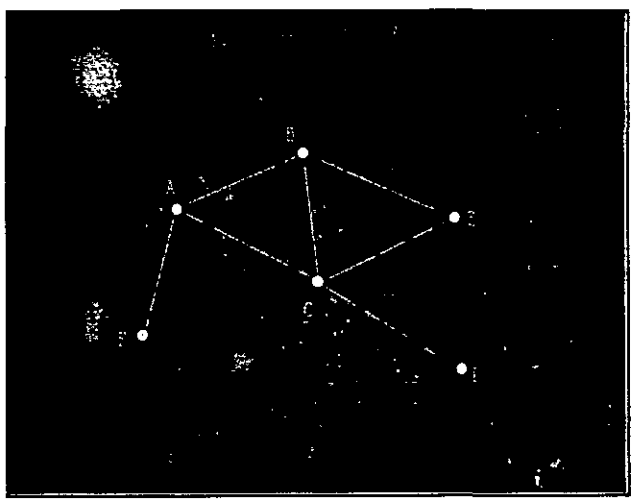
ARBOLES

Para poder definir el concepto de un árbol, tendremos que hacer referencia primero a lo que es una gráfica.

Una gráfica es un conjunto de pares ordenados:

G = { (a,b), (a,c), (c,e), (b,e), (c,d), (a,f) }

representada de la siguiente forma:



Arco dirigido

Si en un par ordenado (a,b) es importante considerar que a es el nodo inicial y b el nodo final, estaremos hablando de un arco dirigido.

Grado externo de un nodo

El grado externo de un nodo u es el número de arcos que salen de él.

Grado interno de un nodo

El grado externo de un nodo u es el número de arcos que llegan a él.

Trayectoria

Si en una gráfica con arcos dirigidos ciertos pares ordenados pueden ser colocados en una secuencia de la forma:

$$(a_1, a_2), (a_2, a_3), (a_3, a_4), \ldots, (a_{n-1}, a_n)$$

el conjunto de arcos es llamado una trayectoria desde a_1 hasta a_n. Si $a_1 = a_n$, la trayectoria es un ciclo.

Cuando los arcos de la secuencia son distintos, la trayectoria es simple. Si los arcos son distintos y contienen a todos los nodos de la gráfica, la trayectoria es hamiltoniana.

La longitud de una trayectoria es el número de arcos que la componen.

Lazo o loop

Un arco que une un vértice consigo mismo se llama lazo. La dirección de una lazo no tiene ningún significado.

Definición de Arbol

Con los conceptos anteriores podemos definir un árbol:

Un árbol es una gráfica en la que:

- El número de nodos es igual al número de arcos más uno.
- Todos los nodos son de grado interno uno, excepto un nodo llamado raíz, de grado cero.
- No hay ciclos.
- Cualquier trayectoria es simple.
- Entre cualquier par de nodos solo hay una trayectoria.
- Cualquier arco es un arco de desconección.

En la terminología que se emplea para el estudio de los árboles encontraremos entre otros, los términos siguientes:

Se define como grado o grado externo de un nodo al número de sus subárboles.

Una hoja o nodo terminal es un nodo de grado cero.

Un nodo ramal es un nodo de grado mayor de cero.

El nivel de un nodo es el nivel de su antecesor directo más uno. El nivel de la raíz es uno.

Es frecuente que los nodos de un árbol reciban nombres, tales como: el nodo a es padre de b, c, d, si existe un arco de a a b, uno de a a c y otro de a a d; que b, c, d son hijos de a.

ARBOLES BINARIOS

Los árboles binarios son aquellos cuyos nodos tienen un grado externo menor o igual a dos.

Los árboles binarios tienen muchas aplicaciones en sistemas operativos y bases de datos.

Las operaciones que se definen para un árbol binario son: inserción y recorrido.

El recorrido de un árbol binario es el procedimiento de visitar cada uno de sus nodos, con el objeto de sistematizar la recuperación de la información almacenada.

Una de las formas más simples de recorrer un árbol es de la raíz hacia los nodos hoja (top-down).

El recorrido top-down de un árbol binario puede ser de tres formas diferentes:

- preorden
- inorden
- postorden

En el recorrido preorden:

- se visita la raíz
- se recorre el subárbol izquierdo
- se recorre el subárbol derecho

En el recorrido inorden:

- se recorre el subárbol izquierdo
- se visita la raíz
- se recorre el subárbol derecho

En el recorrido postorden:

- se recorre el subárbol izquierdo
- se recorre el subárbol derecho
- se visita la raíz

La forma de implementar los algoritmos de recorrido es en forma recursiva y no recursiva; sin embargo, la forma recursiva es mucho más entendible, por lo que es la que se presentará.

Para ello consideremos la siguiente estructura:

```
typedef struct x {
                int       dato;
                struct x  *ligaIzq,
                          *ligaDer;
}     ELEMENTO;

typedef    ELEMENTO   *ARBOL;
```

Para definir una variable:

```
ARBOL      arbol=NULL;
```

Las funciones de recorrido se muestran a continuación:

```
int preOrden(ARBOL  a) {

if ( a != NULL ) {
     printf("%d ", a->dato);
     preOrden(a->ligaIzq);
     preOrden(a->ligaDer);
}
}
```

```
int inOrden(ARBOL  a) {

if ( a != NULL ) {
     inOrden(a->ligaIzq);
     printf("%d ", a->dato);
     inOrden(a->ligaDer);
}
}


int postOrden(ARBOL  a) {

if ( a != NULL ) {
     postOrden(a->ligaIzq);
     postOrden(a->ligaDer);
     printf("%d ", a->dato);
}
}
```