ESTRUCTURAS DE DATOS DINÁMICAS

8.1.- Introducción

El tipo de almacenamiento convencional se denomina ***estático***, porque la memoria destinada a las variables no cambia durante la ejecución del programa.

Otra forma de almacenar las variables, llamada almacenamiento dinámico, permite reservar zonas de memoria en cualquier momento de la ejecución de un programa. El lenguaje C permite crear variables dinámicas mediante la biblioteca de funciones. Se pueden utilizar las funciones **malloc** y **free** entre otras para reservar y liberar memoria.

En algunas aplicaciones, es necesario organizar los datos de forma dinámica de manera que la estructura que los almacene pueda ampliarse o reducirse según las necesidades del programa.

Las variables astáticas se almacenan en la zona de almacenamiento estático donde están todas las variables globales y estáticas del programa. Las variables dinámicas, por su parte, se almacenan en una zona llamada **montículo** o zona de almacenamiento dinámico

Se dice que una estructura de datos es dinámica cuando la aplicación le asigna el espacio que requiere de memoria en tiempo de ejecución, en contraposición con las estructuras de datos estáticas en las que el tamaño de la estructura se define en tiempo de compilación.

8.2.- Funciones de asignación dinámica

Cuando se define una matriz, también se define su tamaño, que permanece invariable en todo el programa. La matriz definida de esta forma se almacenará en una zona de memoria llamada ***zona de asignación estática***, donde residen todas las variables globales y estáticas del programa.

Las otras zonas de memoria que contiene un programa ejecutable son la ***pila***, que está limitada según la memoria disponible; el ***código del programa*** y la ***zona de almacenamiento dinámico***.

Según las características del programa, es posible que una matriz únicamente se necesite en un momento determinado de su ejecución. Para estos casos se emplea otra forma de almacenamiento de datos, llamado ***almacenamiento dinámico***. Las variables dinámicas se emplean utilizando un puntero que referencia a la zona dinámica de memoria (llamada también ***montículo o montón de memoria***) donde residen todas las variables dinámicas del programa.

El montículo de memoria es memoria libre, no utilizada por el programa, por el sistema operativo ni por cualquier otro programa que se esté ejecutando en el ordenador. El tamaño del montículo no se conoce por anticipado, pero generalmente contiene gran cantidad de memoria libre, pero aunque muy grande, es finito y se puede agotar.

El núcleo del sistema de asignación dinámica de C está compuesto por las funciones **malloc()** y **free()**. Estas funciones trabajan juntas usando la región de memoria libre para

establecer y gestionar una lista de memoria disponible. La función **malloc()** asigna memoria y la función **free()** la libera (devuelve). Esto es, cada vez que se hace una petición de memoria con **malloc()**, se asigna una parte de la memoria restante. Cada vez que se libera memoria con **free()**, se devuelve la memoria al sistema. Cualquier programa que use estas funciones debe incluir el archivo de cabecera **<stdlib.h>**

El prototipo de la función malloc() es:

void *malloc(size_número_de_bytes)

Donde **size_número_de_bytes** representa el número de bytes de memoria que se quieren asignar.

La función devuelve un puntero de tipo **void*** lo que significa que puede apuntar a cualquier tipo de puntero. Tras una llamada fructífera, **malloc()** devuelve un puntero al primer byte de la región de memoria dispuesta en el montón. Si no hay suficiente memoria para satisfacer la petición de **malloc()**, se produce un fallo de asignación y **malloc()** devuelve el puntero nulo.

El programa siguiente ilustra la utilización de la función **malloc()**.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main (void)
{
    int *p;
    int i;
    p = (int*)malloc(20*sizeof(int));
    if (p == NULL)
    {
         puts ("No hay memoria disponible");
          exit(1);
    }

    for (i = 0; i < 20; i++)
         p[i] = i;
    printf ("Direccion de p: %p\n",&p);
    printf ("p apunta a : %p\n\n", p);
    for (i = 0; i < 20; i++)
    printf ("Direccion de p[%2d]: %p        Contenido: %d\n", i, &p[i],p[i]);
    free(p);
    printf ("Pulse una tecla para finalizar");
    getche();
}
```

La función **realloc()** permite reasignar un bloque de memoria previamente asignado, cuyo prototipo es:

void* realloc(void * *pBlomem,* **size_número_de_bytes)**

Donde *pBlomen* es un puntero que apunta al comienzo del bloque de memoria actual. Si este puntero es NULL, la función se comporta exactamente igual que la función **malloc()**. Si el puntero no es un puntero nulo, entonces tiene que ser devuelto por las funciones **malloc(), calloc()**, o por la propia función **realloc()**. El siguiente programa muestra como realizar una reasignación de memoria y pone de manifiesto que después de una reasignación, la información no varía en el espacio de memoria conservado. Por último, el bloque de memoria es liberado.

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



void main (void)
{
        int *p = NULL, *q = NULL;
    int Nbytes = 100;
    // Asignación de memoria para Nbytes

    if ((p = (int*)malloc(Nbytes)) == NULL)
        {
        printf ("Insuficiente espacio de memoria");
          exit(1);
       }

    printf ("Se han asignado %d bytes de memoria \n");

    // Operaciones sobre el bloque de memoria
    // ...
    // ...

    // Reasignación del bloque para contener mas datos
    if (Nbytes == 0)
        {
                free(p);
          printf("\nEl bloque se ha liberado\n");
          exit(1);
       }
    q =(int*)realloc(p, Nbytes*2);
        if (q == NULL)
        {
                printf ("La reasignación no ha sido podsible\n");
             printf ("Se conserva la memoria original");
          }
       else
       {
        p = q;
          printf ("Bloque reasignado\n");
```

```
            printf("Nuevo tamaño %d bytes\n", Nbytes*2);
        }

    // Operaciones sobre el bloque de memoria
    // ...
    // ...
    free(p);
    printf("El bloque ha sido liberado");
          gotoxy(15,20);printf ("Pulse una tecla para finalizar");
    getche();
    }
```

8.3.- Matrices dinámicas

Hasta ahora todas las matrices que se han utilizado han sido estáticas, lo que exigía conocer, en el momento de escribir el código del programa, cuál era la dimensión de la matriz y expresar esta dimensión como una constante entera:

```
#define MAX 100

// …

int matriz[MAX];
```

Asignando memoria dinámicamente, puede decidirse durante la ejecución del programa cuantos elementos puede tener la matriz. Este tipo de matrices recibe el nombre de **matrices dinámicas,** porque se crean durante la ejecución del programa. Al igual que ocurría con las matrices estáticas, los elementos de las matrices dinámicas pueden ser de cualquier tipo.

Para asignar memoria dinámicamente para una matriz, además de la función **malloc()** la biblioteca de C (archivo de cabecera <stdlib.h>) proporciona la función **calloc()** cuya sintaxis es:

void* calloc(size_número_elementos, size_elemento);

donde size_numero_elementos especifica el número de elementos de la matriz y size_elemento representa el tamaño en bytes de cada elemento.

La función **calloc()** devuelve un puntero a void que referencia el espacio de memoria asignado o NULL si no existe un bloque de memoria del tamaño solicitado.

Según esto, los bloques de código

```
int *p;
    p = (int*)malloc(200*sizeof(int));
if (p == NULL)
{
      puts ("No hay memoria disponible");
       exit(1);
}
```

y

```
int *p;
    p = (int*)calloc(200, sizeof(int));
if (p == NULL)
{
     puts ("No hay memoria disponible");
      exit(1);
}
```

son absolutamente equivalentes, ya que ambas versiones solicitan reservar memoria para 200 * sizeof(int) bytes consecutivos., o, lo que es lo mismo, para una matriz de 200 elementos enteros.

8.3.1.- Matrices dinámicas numéricas

Para crear una matriz durante la ejecución de un programa basta con declarar una variable que apunte a los objetos del tipo de los elementos de la matriz e invocar a alguna de las funciones **malloc, calloc** o **realloc** vistas anteriormente. Para liberar el espacio de memoria asignado cuando ya no se necesite la matriz se utilizará la función **free**.

Matrices dinámicas numéricas de una dimensión

Un ejemplo es el programa siguiente:

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

void main (void)
{
        int *m = NULL;
    int NBytes = 0, correcto = 0, Nelementos = 0, i;

    while (!correcto || Nelementos < 1)
    {
     printf ("Numero de elementos de la matriz:   ");
    correcto = scanf("%d", &Nelementos);
    fflush(stdin);
    }
    // Tamaño del bloque de memoria
    NBytes = Nelementos * sizeof(int);


    m = (int*)malloc(NBytes);
    if (m == NULL)
    {
     puts ("No hay memoria disponible");
    exit(1);
    }
    printf ("Se han asignado %u bytes de memoria\n", NBytes);
```

```
//Puesta a 0 de los elementos de la matriz
for (i = 0; i < Nelementos; i++)
      m[i] = 0;

free(m);
gotoxy(15,20); printf ("Pulse una tecla para finalizar");
getche();
}
```

Es importante observar que se puede utilizar la indexación se la matriz, ***m[i]***, para acceder a los elementos de la misma. También es importante saber que cuando se accede a un elemento de una matriz dinámica a través de un puntero, C toma como referencia el tipo del objeto apuntado para asignar el número de bytes correspondiente a ese tipo. En el ejemplo anterior, *m* es un puntero a un entero por lo que *m[i]* accede a un número de bytes igual al tamaño de un entero (4 bytes) localizados a partir de la dirección *m+i*

Matrices dinámicas numéricas de dos dimensiones

Para asignar memoria para una matriz de dos dimensiones, el proceso se divide en dos partes:

- **Asignar memoria para una matriz de punteros**, cuyos elementos referenciarán cada una de las filas de la matriz de dos dimensiones que se desea crear

- **Asignar memoria a cada una de las filas.** El número de elementos de cada fila puede ser variable

El programa que se muestra a continuación crea dinámicamente una matriz entera de dos dimensiones, pone todos los elementos a cero y posteriormente libera la memoria.

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

void main (void)
{
        int **m = NULL; // puntero que referencia la matriz de punteros
    int Filas = 0, Columnas = 0;
    int correcto = 0, fil = 0, col = 0,   i;

    while (!correcto || Filas < 1)
    {
         printf ("Numero de filas de la matriz:   ");
        correcto = scanf("%d", &Filas);
        fflush(stdin);
    }

    correcto = 0;
    while (!correcto || Columnas < 1)
    {
         printf ("Numero de columnas de la matriz:   ");
        correcto = scanf("%d", &Columnas);
```

```
        fflush(stdin);
    }

// Asignar memoria a la matriz de punteros
        if ((m = (int**)malloc(Filas * sizeof(int))) == NULL)
    {
        printf ("Insuficiente espacio de memoria");
       exit(1);
    }

// Asignar memoria para cada una de las filas
        for (fil = 0 ; fil < Filas; fil++)
    {
        if ((m[fil] = (int*)malloc(Columnas * sizeof(int))) == NULL)
       {
        printf ("Insuficiente espacio de memoria");
       exit(1);
       }
     }
     //Puesta a 0 de los elementos de la matriz
    for (fil = 0; fil < Filas; fil++)
            for(col = 0; col < Columnas; col++)
                m[fil][col] = 0;
     // Imprimir los elementos de la matriz
    for (fil = 0; fil < Filas; fil++)
                {
            for(col = 0; col < Columnas; col++)
          printf ("%d     ",m[fil][col]);
       printf ("\n");
        }
     //Liberar la memoria asignada a cada fila
     for (fil = 0; fil < Filas; fil++)
                free(m[fil]);
     //LIberar la memoria asignada al la matriz de punteros
    free(m);
    gotoxy(15,20); printf ("Pulse una tecla para finalizar");
    getche();

}
```

Debe observarse que para liberar la memoria ocupada por la matriz debe seguirse un proceso lógicamente inverso al de crear la matriz, esto es, liberando primero la memoria asignada a cada una de las filas y después la asignada a la matriz de punteros.

8.3.2.-Matrices dinámicas de cadenas de caracteres

Una matriz dinámica de cadena de caracteres una matriz de dos dimensiones cuyos elementos son de tipo **char**. Por tanto, su constricción es idéntica a las matrices que se acaban de ver en el apartado anterior. Sin embargo, cada fila tiene un tamaño variable, que depende sólo del número de caracteres que se almacene en la fila mas el carácter nulo de

terminación., Lo que implica una asignación dinámica de memoria diferente para cada fila, como se ilustra en el programa siguiente, que lee, ordena y visualiza cadenas de caracteres:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
int leercadenas (char **nombre, int Filas);
void ordenarcadenas(char **nombre, int fil);
void visualizarcadenas(char **nombre, int fil);

void main (void)
{
        char **nombre = NULL; // puntero que referencia la matriz de punteros
   int Filas = 0;
   int correcto = 0, fil = 0, f = 0;
   while (!correcto || Filas < 1)
   {
        printf ("Numero de filas de la matriz de caracteres:   ");
      correcto = scanf("%d", &Filas);
      fflush(stdin);
   }

// Asignar memoria a la matriz de punteros
        if ((nombre = (char**)malloc(Filas * sizeof(char*))) == NULL)
   {
        printf ("Insuficiente espacio de memoria");
      exit(1);
   }
   // Operaciones a realizar
   fil = leercadenas (nombre, Filas);
   if (fil == -1 )
   {
        printf("Se ha producido un error en la lectura de cadenas");
      exit(1);
   }
   ordenarcadenas(nombre, fil);
   visualizarcadenas(nombre, fil);
      // Liberar la memoria asignada a cada una de las filas

    for (f = 0; f < Filas; f++)
                free(nombre[f]);
    //LIberar la memoria asignada al la matriz de punteros
   free(nombre);
   gotoxy(15,20); printf ("Pulse una tecla para finalizar");
   getche();
   }

int leercadenas (char **nombre, int Filas)
{
```

```
        int f = 0, longitud = 0;
        char cadena[81];
        printf ("Introduzca cadenas de caracteres.\n");
        printf ("para finalizar introduzca una cadena nula\n");
        printf ("Es decir, pulse INTRO\n\n");

        while (f < Filas && (longitud = strlen(gets(cadena)))>0)
        {
                //asignar espacio para la cadena de caracteres
        if ((nombre[f] = (char*)malloc(longitud+1)) == NULL)
        {
                        printf("Insuficiente espacio en memoria");
        exit(1);
        }
        //Copiar la cadena en el espacio reservado para ella
        strcpy(nombre[f], cadena);
        f++;
        }
        return (f);
}


void ordenarcadenas(char **nombre, int fil)
{
        char *aux; //puntero auxiliar
    int i = 0, s = 1;
    while ((s == 1) && (--fil > 0))
    {
        s = 0; // no permutacion
       for (i = 1 ; i <= fil ; i++)
        if (strcmp (nombre [i-1], nombre[i]) > 0)
          {
                aux = nombre[i - 1];
            nombre[i - 1] = nombre[i];
            nombre[i] = aux;
            s = 1; //permutacion
          }
    }
}


void visualizarcadenas(char **nombre, int fil)
{
        int f = 0;
    printf("\n\n\n");
        for (f=0; f < fil; f++)
        printf ("%s\n", nombre[f]);
}
```

8.4.-Estructuras de datos y programación dinámica

Los datos dentro de un programa que estén relacionados y tengan un tratamiento común se pueden agrupar formando matrices. En estas natrices el acceso es directo y se realiza a través de un índice, pero sin ninguna relación posterior entre el valor o valores de la matriz con sus respectivos índices, salvo el de la relación posicional. Esto es, el índice indica la posición del valor respecto del comienzo de la matriz donde se ha almacenado un elemento del conjunto.

Por ejemplo, un programa que almacene llamadas telefónicas puede tener en memoria una matriz de registros (estructuras), cada uno de ellos almacenará los detalles de una llamada:

```
struct
{
        char hora;
        char min;
        char seg;
        char dia;
        char mes;
        char anio;
} FECHA;

struct
{
        int extension;
        int pasos;
        float importe;
        char nummarcado;
        int enlace;
        int duracion;
        FECHA fechallamada;
        char destino[25];
} LLAMADA;

LLAMADA llamadas[50];
```

El mayor inconveniente de una agrupación lineal de datos, es que el tamaño de la estructura de datos permanece fijo a lo largo de todo el programa, pues cuando se define una matriz, también se define su tamaño, que a partir de ese momento, permanecerá invariable.

En muchas aplicaciones, los conjuntos de datos contienen relaciones elementales, y se prestan a estructurarlos en matrices. En estos casos el empleo de matrices presenta algunas ventajas:

Claridad, por la agrupación física de los datos.

Construcción inmediata

Acceso directo a los elementos de la matriz

Facilidad de uso.

Un inconveniente de las matrices es su falta de encadenamiento. Por ejemplo, si se quiere ordenar por el campo de "extensión" la matriz de llamadas del ejemplo anterior, se deben trasladar los registros de una posición a otra hasta quedar ordenados. Dependiendo del algoritmo de ordenación que se utilice, el proceso llevará mayor o menor tiempo y mayor o menor número de traslados de registros.

Una mejora de la matriz de estructuras es crear otra matriz con valores enteros. Cada elemento de la nueva matriz será un índice ordenado que indique la posición de la matriz original. Por ejemplo, para clasificar la matriz en orden ascendente, si el contenido de la matriz original (campo de "extensión") es:

200

400

100

300

. . .

El contenido de la nueva matriz será:

3

1

4

2

. . .

De esta forma, para ordenar la matriz original, sólo será necesario permutar los índices correspondientes a la nueva matriz. Esta mejora evita tener que realizar traslados de registros de una posición a otra.

Si se modifica la matriz de llamadas de forma que queden encadenados los registros, añadiendo a cada registro un campo que apunte al siguiente, se estaría creando una estructura de otro tipo, con otra forma de acceso y de obtención de información.

Según sea la forma en que se relacionen los elementos, se pueden organizar los datos de distintos modos, creando diferentes estructuras de datos:

Pilas

Colas

Listas

Árboles binarios

Etc.

Las dos operaciones elementales sobre estas estructuras de datos son:

Almacenar un elemento de la estructura y

Eliminar un elemento de la estructura

Aunque en cada implementación se pueden añadir otras funciones básicas.

Las características generales de las estructuras mas conocidas son las siguientes:

PILAS: las adiciones y eliminaciones se realizan al final de la pila (llamada también **cabecera o cabeza** de la pila). De esta forma, la pila se mantiene en orden inverso al de creación de sus elementos. En muchos casos esta estructura sólo emplea un puntero, el que apunta a la cabeza de la pila; en otros, se emplea un puntero de encadenamiento para cada elemento. Se pueden emplear en programas que evalúan expresiones

COLAS: Las adiciones se realizan al final y las eliminaciones en la cabecera. Mantiene el orden de creación de los elementos.

LISTAS: Los elementos pueden estar encadenados de forma bidireccional, secuencial, doble, etc. Cada implementación se adapta a distintas necesidades.

ÁRBOLES BINARIOS: A cada elemento se le den0omina **nodo**. Asociado al nodo hay un valor que se emplea como **clave o índice**. Cada nodo puede tener cero, uno o dos descendientes. Al primer nodo se le llama **raíz**. El árbol se mantiene ordenado a través de la clave, haciendo que el descendiente izquierdo del nodo sea menor que el derecho. Los árboles binarios tienen aplicaciones en inteligencia artificial, en la construcción de compiladores, analizadores sintácticos, etc.

Estas estructuras de datos tienen como característica principal la ***flexibilidad***, en lo que se refiere a la inserción y borrado de sus elementos. Los datos no tienen por qué estar agrupados físicamente, como las matrices, sino que se encadenan.

Una cola o pila se puede programar

Declarando una matriz de tamaño variable, empleando programación dinámica

Declarando una matriz de tamaño fijo

En este último caso, la ventaja es el acceso directo a la matriz, a través del índice. Para ciertas aplicaciones esta implementación es suficiente. Aunque el almacenamiento estático tiene el inconveniente de que el máximo número de elementos se define al comenzar el programa, y por lo tanto, en ciertas ocasiones, sólo se utilizará una pequeña parte de la memoria reservada al principio.

La ventaja de la programación dinámica frente a la estática es que el número de elementos de la estructura de datos se ajusta en cada instante a las necesidades del programa. La estructura dinámica de datos aumenta y disminuye según las operaciones que se lleven a cabo, y, cuando ya no es necesaria la estructura, se puede liberar toda la memoria que ocupa.

Si se quiere almacenar en una matriz estática un conjunto de elementos ordenados, números enteros, por ejemplo, cada posición física de la matriz se relaciona con el orden de los elementos; de forma que para ordenarlos es preciso moverlos de una posición a otra. Si se amplía este caso a elementos que contienen mas información, por ejemplo, que cada uno sea un registro, se comprenderá que una implementación que realice continuamente el transvase de la información de una posición a otra no puede ser muy eficiente.

Este problema se puede solucionar de varias formas; por ejemplo, introduciendo un campo de encadenamiento en cada registro. Así, cuando se deba actualizar la estructura, sólo se modifican los punteros. El último paso, mas general, es que cada registro se almacene en la zona dinámica de datos, de forma que la estructura pueda crecer indefinidamente. No obstante, la conveniencia o no de utilizar programación dinámica depende de la aplicación en sí pues ambas formas de implementación se ajustan a diferentes problemas de programación

Las principales ventajas de la programación dinámica son:

Ahorro de tiempo, es decir, aumento en la velocidad de proceso al reajustar los punteros, no los contenidos de los registros.

Ahorro de memoria, al ajustarse al número de elementos que se necesitan en cada momento.

La mayor desventaja es que al ser mas compleja la estructura, el programa que la emplea es mas difícil de leer.

La solución de pasar una matriz de elementos ordenados por posición física a una matriz de elementos encadenados, de manera que cada elemento apunte al siguiente, crea una estructura de tipo **cola**. Le faltaría solamente un puntero que señale al primer elemento.

Lo general en programación dinámica es que la estructura pueda crecer indefinidamente hasta llenar toda la memoria disponible del ordenador. Para prevenir esto, se debe controlar el número máximo de elementos que se desean emplear o, de un modo mas genérico, controlar exclusivamente las funciones de asignación de memoria. Así, cuando se vayan a almacenar datos nuevos, se comprueba si hay suficiente cantidad de memoria para albergarlos. Por ejemplo, la función **malloc()** devuelve un puntero nulo si no ha podido reservar toda la memoria requerida.

Las estructuras de datos que se han mencionado se pueden considerare como tipos abstractos, pues proporcionan unos datos organizados de una cierta forma y unas operaciones para manipularlos. El tratamiento de estos datos sólo podrá realizarse a través de las operaciones asociadas.

8.5.- Pilas

Una pila es una estructura del tipo "***último en entrar, primero en salir***" , también llamada estructura **LIFO** (last in, first out), pues el último elemento que se introduzca en la pila será el primero en salir de ella. A este elemento que se extrae se le denomina ***cabeza de la pila***. Las operaciones básicas sobre una pila son:

Introducir.- Almacenar un elemento en una pila

Sacar.- Extraer un elemento de una pila.

El siguiente ejemplo ilustra el efecto de introducir y sacar de una pila:

Operación	**Contenido de la pila**
Introducir 1	1
Introducir 2	2 1
Introducir 3	3 2 1
Sacar (extrae 3)	2 1
Introducir 4	4 2 1
Sacar (extrae 4)	2 1
Sacar (extrae 2)	1
Sacar (extrae 1)	Pila vacía

Para programar una pila se necesita una zona de memoria para almacenarla. Se pueden emplear varios métodos.

Definir una matriz

Definir una matriz en la zona dinámica de memoria

Crear cada elemento en la zona dinámica de memoria a medida que se vayan necesitando.

En el primer caso la memoria para la pila se declara directamente como una matriz , que tendrá la misma longitud durante todo el programa. La matriz se indexa para almacenar o extraer los elementos.

En el segundo caso también se reserva una zona de memoria para toda la pila y esta zona será, así mismo, contigua físicamente. La diferencia sustancial con el caso anterior es que, al emplear memoria dinámica, se puede liberar toda la memoria que ocupa la pila cuando ya no es necesario utilizarla.

Una forma mas general de programar una pila es reservando la memoria a medida que se necesite. En este caso, la memoria que ocupa la pila crece o decrece, según el numero de elementos que se almacenan en cada momento.

El siguiente programa crea una pila de 20 elementos declarandodirectamente una matriz de enteros:

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

#define MAXPILA 20
int pila[MAXPILA];
int pospila = 0;

int introducir(int i);
int sacar();

void main(void)
{
    int i, j, k;
        introducir(1);
    introducir(2);
    introducir(3);

    i = sacar();
    j = sacar();
    k = sacar();

    printf("\n%d %d %d \n",i, j, k);
    getch();
}

int introducir(int i)
{
        if (pospila >= MAXPILA) return 0; //pila llena
    pila[pospila] = i;
    pospila++;
    return 1;
}

int sacar()
{
        pospila--;
    if (pospila <= 0)
    {
        puts ("Error, pila vacia");
       exit(1);
    }
    return (pila[pospila]);
}
```

La pila se ha definido como entera para almacenar variables enteras y se ha seguido el método estático de almacenamiento.

El siguiente programa emplea el segundo método para programar la pila, pues emplea una matriz de tamaño MAXPILA en la zona dinámica de memoria. Posteriormente libera la memoria ocupada y reserva memoria para otra pila:

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

#define MAXPILA 20

char *finpila,*principiopila;

char introducir(char **p,char car);
char sacar(char **p);
void main(void)
{
        char *pila, *p;
  char a, b, c;
   /* Asigna a la pila memoria dinámica*/
   p = (char*) malloc(MAXPILA);
   if (p == NULL)
   {
        puts("No hay memoria disponible");
       exit(1);
   }
        pila = p;
   /*introducir elementos a la pila*/
        introducir (&pila,'A');
        introducir (&pila,'B');
        introducir (&pila,'C');

   a = sacar(&pila);
        printf("%c\n",a);
   b = sacar(&pila);
   printf("%c\n",b);
   c = sacar(&pila);
   printf("%c\n",c);
   // Se libera la pila
   free(p);
   getch();
   // Se crea una pila nueva
   p = (char*) malloc(MAXPILA);
   if (p == NULL)
   {
        puts("No hay memoria disponible");
       exit(1);
   }
        pila = p;
   principiopila = pila;
   finpila = pila + MAXPILA - 1 ;
```

```
    introducir(&pila,'M');
    a = sacar(&pila);
    printf("%c\n",a);
    free(p);
    getch();

}
char introducir(char**p,char car)

{
        if (*p> finpila) return 0; //pila llena
    **p = car;
    (*p)++;
    return 1;
}


char sacar(char **p)

{

        //pospila--;

    if ((*p)-- < principiopila )

    {

         puts ("Error, pila vacia");

        exit(1);

    }

    return (**p);

}
```

Otra forma de diseñar la pila es forzando a que la estructura crezca y decrezca según las operaciones de introducir y sacar, formando una estructura totalmente dinámica. En esta implementación cada elemento de la pila será un registro con dos partes bien diferenciadas: primero, los campos de datos y segundo un campo que encadena la estructura con los otros elementos.

También debe existir siempre un puntero que señale a la cabeza de la pila , es decir, al elemento mas reciente.

Para programar la pila se deben crear dos funciones, una para cada operación básica.

La función **introducir()** (tradicionalmente se utiliza el nombre inglés **push()**) debe:

1º.- Reservar sitio para el nuevo elemento

2º.- Almacenar la información del nuevo elemento

3º.- Encadenar el nuevo elemento

4º.- Apuntar a la nueva cabeza de la fila:

La función **sacar()**(su nombre inglés es **pop()**) deberá seguir los siguientes pasos:

1º.- Emitir un mensaje de error si no hay ningún elemento en la pila

2º.- Extraer el elemento cabeza

3º.- El puntero dirigido al elemento que se extrae deberá apuntar al siguiente elemento, la nueva cabeza de la pila.

4º.- Liberar la memoria que ocupaba el elemento extraído

5º.- Devolver la información extraída.

Por otro lado, también puede añadirse una función para examinar la cabeza de la pila. Esta función, **ver()**, debe:

1º.- dar un mensaje de error si no hay ningún elemento en la pila

2º. – Devolver la información concerniente al elemento cabeza.

Cada elemento de la pila se define como una estructura tipo PILA que contiene un campo de información y un campo para enlazar con el siguiente elemento de la pila. Para definir este campo de encadenamiento es necesario realizar una **definición recursiva**, ya que el tipo es el mismo que el de la estructura:

```
struct PILA
{
   int informacion;
   struct PILA *lazo;
};
```

Los prototipos de las funciones son:

```
int introducir(struct PILA **p, int info);

char sacar(struct PILA **p, int *error);

  int ver (struct PILA *p, int *error);
```

Debe observarse que mientras las dos primeras funciones utilizan como parámetro la dirección del puntero a la pila, la función **ver()** tiene como parámetro directamente el puntero a la pila, que np pasa su dirección ya que esta no se va a modificar.

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

```
#include <conio.h>
#include <stdlib.h>
#include <alloc.h>

struct PILA
{
    int informacion;
    struct PILA *lazo;
};

int introducir(struct PILA **p,int info);
char sacar(struct PILA **p, int *error);
int ver (struct PILA *p, int *error);

void main(void)
{
        int x, y, z, u, v;
    int error;
    PILA *p;
        p = NULL;
       introducir(&p,1);
       introducir(&p,2);
       introducir(&p,3);
       introducir(&p,4);
       introducir(&p,5);
       x = sacar (&p,&error);
       printf("\n%d",x);
       y = sacar (&p,&error);
       printf("\n%d",y);
       z = sacar (&p,&error);
       printf("\n%d",z);
       u = sacar (&p,&error);
       printf("\n%d",u);
       v = sacar (&p,&error);
       printf("\n%d",v);
       getch();
}


int introducir(struct PILA **p, int info)
{
        struct PILA *nuevoelemento;
    //devuelve 0 si no hay memoria disponible, 1 en caso contrario
    nuevoelemento = (PILA*)malloc(sizeof(PILA));
    if (nuevoelemento == NULL) return 0;
    //Se almacena la información para el nuevo elemento
    nuevoelemento->informacion = info;
    //El nuevo elemento apunta a la antigua cabeza de la pila
    nuevoelemento -> lazo = *p;
    //El puntero a la pila apunta a la nueva cabeza
```

```
    *p = nuevoelemento;
    return 1;
}

char sacar(struct PILA **p, int *error)
{
      struct PILA *elementoasacar = *p;
    int infoasacar = 0;
    //Error = 1 si la pila esta vacia 0 si no hay errores
    if (*p)
    {
      //infoasacar recibe la informacion de la cabeza de la pila
      infoasacar = elementoasacar->informacion;
      // el puntero a la pila debe apuntara la nueva cabeza
      *p = (*p)->lazo;
      //Se libera la memoria del elemento
      free(elementoasacar);
      *error = 0;
    }
    else
      *error = 1;
    return(infoasacar);
}

int ver (struct PILA *p, int *error)
{
      //Error = 1 si la pila esta vacia 0 si no hay errores
    *error = 0;
    if (p)
      //Devuelve la informacion
      return(p ->informacion);
      *error = 1;
      return (0);
    }
```

La salida de este programa es:

5 4 3 2 1

es decir, la estructura pila devuelve los elementos en orden inverso a como los recibe.

8.6.- Colas

Una cola es una estructura del tipo "***primero en entrar, primero en salir***", conocida también como estructura FIFO (firs in, first out). El primer elemento de la cola, el mas antiguo es el primero en salir. Las operaciones básicas sobre una cola son:

Introducir.- Almacenar un elemento en la cola

Sacar.- Extraer un elemento de la cola.El siguiente ejemplo muestra el efecto de estas operaciones en la cola:

Operación	Contenido de la pila
Introducir 1	1
Introducir 2	1 2
Introducir 3	1 2 3
Sacar (extrae 1)	2 3
Introducir 4	2 3 4
Sacar (extrae 2)	3 4
Sacar (extrae 3)	4
Sacar (extrae 4)	Pila vacía

Al igual que con la pila, la estructura COLA puede programarse de forma totalmente dinámica, para admitir cualquier número de entradas y salidas; de esta forma, la única limitación es la memoria disponible. En este diseño, cada elemento de la cola tiene dos campos, uno con la información a almacenar y otro que encadena los elementos de la cola.

Las declaraciones previas son:

```
struct COLA
{
    int informacion;
    struct COLA *lazo;
};
```

siendo los prototipos de las funciones los siguientes:

```
int introducir(struct COLA **p, int info);
int sacar(struct COLA **p, int *error);
```

La función **introducir()** (tradicionalmente se utiliza el nombre inglés **push()**) debe:

1º.- Recorrer la cola, a través del encadenamiento, hasta el final, para ello, debe existir un puntero que indique el comienzo de la cola

2º.- reservar memoria para el nuevo elemento

3º.- Almacenar la información del nuevo elemento;

4º.- Enlazar el último elemento de la cola con el nuevo elemento

La función **sacar()**(su nombre inglés es **pop()**) deberá seguir los siguientes pasos:

1º.- Dar un mensaje de error si no hay elementos en la cola.

2º.- Extraer el elemento y guardar la información que contiene

3º.- El puntero dirigido al elemento que se extrae deberá apuntar al siguiente elemento, que será el mas antiguo de la cola.

4º.- Liberar la memoria que ocupaba el elemento extraído

5º.- Devolver la información extraída.

Como puede apreciarse, la operación de insertar un nuevo elemento en la cola es mucho mas lenta que la de extraer. Esto se puede solucionar añadiendo otro puntero a la cola, de manera que apunte siempre al final, al último elemento de la cola ; de esta forma se evita el bucle que busca el último elemento.

Así mismo pueden realizarse otros cambios como mantener un contador para admitir un número máximo de elementos; así como declarar las estructuras directamente como matrices sin emplear asignación dinámica.

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

struct COLA
{
        int informacion;
    struct COLA *lazo;
};

int introducir(struct COLA **p,int info);
char sacar(struct COLA **p, int *error);


void main(void)
{
        int x, y, z, u, v;
    int error;
    COLA *p;
        p = NULL;
       introducir(&p,1);
       introducir(&p,2);
       introducir(&p,3);
       introducir(&p,4);
       introducir(&p,5);
       x = sacar (&p,&error);
       printf("\n%d",x);
       y = sacar (&p,&error);
       printf("\n%d",y);
       z = sacar (&p,&error);
       printf("\n%d",z);
       u = sacar (&p,&error);
       printf("\n%d",u);
```

```
        v = sacar (&p,&error);
        printf("\n%d",v);
        getch();
}


int introducir(struct COLA **p, int info)
{
         struct COLA *nuevoelemento;
    struct COLA *actual = *p;
    struct COLA *anterior = NULL;
    // Busca el ultimo elemento de la cola
    while (actual)
    {
         anterior = actual;
        actual = actual->lazo;
    }
    //devuelve 0 si no hay memoria disponible, 1 en caso contrario
    nuevoelemento = (COLA*)malloc(sizeof(COLA));
    if (nuevoelemento == NULL) return 0;
    //Se almacena la información para el nuevo elemento
    nuevoelemento->informacion = info;
    if (anterior)
    {
         nuevoelemento->lazo = anterior->lazo;
        anterior->lazo = nuevoelemento;
    }
    else
    {
         //La cola esta vacia. Es el primer elemento de la cola
        //No hay encadenamiento
        *p = nuevoelemento;
        (*p)->lazo = NULL;
    }
    return 1;
}

char sacar(struct COLA **p, int *error)
{
         struct COLA *elementoasacar = *p;
    int infoasacar = 0;
    //Error = 1 si la pila esta vacia 0 si no hay errores
    if (*p)
    {
         //infoasacar recibe la informacion de la cabeza de la pila
        infoasacar = elementoasacar->informacion;
        // el puntero a la pila debe apuntara la nueva cabeza
        *p = (*p)->lazo;
        //Se libera la memoria del elemento
        free(elementoasacar);
```

```
        *error = 0;
    }
    else
        *error = 1;
    return(infoasacar);
}
```

La salida de este programa es:

1 2 3 4 5

es decir, la estructura cola devuelve los elementos en el mismo orden que los recibe.

8.6.1.- Colas circulares

Una variación de la estructura de cola es la cola circular. La característica de esta estructura es que posee dos punteros, en este caso, dos índices a las posiciones de la cola. El primer índice, *pmete,* indica la posición donde se almacenan los elementos. El segundo índice, *psaca*, indica la posición desde la que se pueden extraer los elementos. La cola estará:

vacía si pmete = psaca

llena si pmete + 1 = psaca

Esto significa que si la cola tiene 50 posiciones, puede almacenar hasta 50 - 1 elementos.

Una utilidad muy común de una pila circular es en los sistemas operativos, para mantener la información que se lee o escribe en un disco o en cualquier otro dispositivo de entrada y salida.

El siguiente ejemplo mantiene una cola circular con los caracteres que se escriben desde el teclado. Cuando la cola se llena, el programa lista todo lo que hay dentro de ella. La función **sacar_cola**() devuelve el carácter de la cola que corresponde al índice o devuelve 0 si la cola está vacía. La función **introducir_cola** devuelve 0 si la cola está llena o un 1 en caso contrario.

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

int introducir_cola(char car);
char sacar_cola(void);

#define MAXCOLA 20
char cola[MAXCOLA+1];
int pmete = 0;
int psaca = 0;

void main(void)
```

```
{
        char car;
        char *pcad;
        puts (pcad);
  for (;;)
        {
                car = getchar();
                if (!introducir_cola(car)) break;
        }

        printf ("\n\nContenido de la cola:   ");
        while ((car = sacar_cola()))
                putchar(car);

}

int introducir_cola(char car)
{
        int tmp;
        tmp = pmete + 1;

        if (tmp > MAXCOLA) tmp = 0;
        if (tmp == psaca) return 0;
        cola[pmete] = car;
        pmete = tmp;
        return 1;
}

char sacar_cola(void)
{
        int tmp;
        char car;
        if (pmete == psaca) return 0;
        car = cola[psaca];
        psaca++;
        if (psaca > MAXCOLA) psaca = 0;
        return car;
}
```

8.7.- Listas

Una estructura mas general que las anteriores es la formada por los elementos de una lista ordenada, elementos que se identifican por un campo o clave y que se insertan y extraen según esa clave identificativa. Las funciones que debe tener una estructura **lista** son:

Insertar: Añadir un elemento a la lista

Eliminar: Elimina un elemento de la lista

Examinar: Averigua si un elemento está presente en la lista

Listar: Lista todos los elementos de la lista

Borrar: Borra toda la lista.

Todas estas funciones deben mantener el orden correcto de la lista. Se pueden añadir otras funciones, para extraer la información, realizar consultas, etc.

La lista debe crearse como una estructura que, de forma recursiva, mantenga un campo para encadenar los elementos de la lista. En el ejemplo a continuación se construye una lista en la que cada elemento de la lista forma una estructura que contiene la siguiente información:

```
struct LISTA
{
        char titulo[60];     //Título de un libro
        struct LISTA *lazo;   //Campo para encadenar los elementos de la lista
        char autor[40];   //Nombre del autor
        int numpag;   //Número de páginas
        float importe; //Precio del libro
        char codigo[20]; //Código del libro
        char editorial[40];//Nombre de la Editorial
}
```

El campo "Titulo del libro" se emplea en el ejemplo como *clave* para mantener ordenada la lista.

Los prototipos de las funciones a emplear son:

```
int insertar (LISTA **, LISTA *);
void listar (LISTA *);
int examinar(LISTA *, LISTA *);
int eliminar(LISTA **, LISTA *);
void borrar (LISTA **);
LISTA *crear(LISTA *);
```

La función **insertar()** recorre la lista hasta encontrar un registro que sea mayor o igual que el título que se va a añadir; de esta forma se mantiene ordenada la lista . La comparación se realiza mediante la función de librería **strcmp()** que devuelve un valor mayor que 0 si el primer argumento es mayor que el segundo; 0 si los dos argumentos son iguales y un valor menor que cero si el segundo argumento es mayor que el primero.

Una vez encontrada la posición adecuada, se llama a la función **crear()** para crear un registro nuevo, que reserva memoria para el nuevo registro y almacena toda la información correspondiente. Una vez hecho esto, lo concatena con la lista:

```
nuevo -> lazo = actual;
```

Y si hay un registro anterior, deberá apuntar al nuevo:

```
anterior -> lazo = nuevo;
```

La función **listar()** escribe en la pantalla los campos de todos los registros de la lista.

La función **borrar()** recorre toda la lista, registro a registro, liberando la memoria ocupada por cada uno..

La función **eliminar()**recorre la lista hasta encontrar el registro buscado o dar con el final de la lista (lazo nulo). Si al encontrar el registro buscado no hay ninguno posterior, el puntero que hace referencia al comienzo de la lista deberá modificarse para que apunte al siguiente registro:

***p = actual -> lazo;**

Si existe un registro anterior, el código

anterior -> lazo = actual -> lazo

encadena el registro anterior para que enlace con el posterior al que se extrae.

El codigo del programa sería:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
struct LISTA
{
        char titulo[60];
        LISTA *lazo;
        char autor[40];
        int numpag;
        float importe;
        char codigo[20];
        char editorial[40];
};
int insertar (LISTA **, LISTA *);
void listar (LISTA *);
int examinar(LISTA *, LISTA *);
int eliminar(LISTA **, LISTA *);
void borrar (LISTA **);
LISTA *crear(LISTA *);

void main(void)
{
        LISTA *tmp1rg, *tmp2rg, *tmp3rg;
        LISTA *p;
        p = NULL;
        tmp3rg = (LISTA *)malloc(sizeof(LISTA));
        if (!tmp3rg)
        {
                puts("No hay memoria disponible");
```

```
            exit(1);
        }
        strcpy(tmp3rg->titulo, "El codigo da Vinci");
        strcpy(tmp3rg->autor, "Dan Brown");
        strcpy(tmp3rg->codigo, "A003 123");
        strcpy(tmp3rg->editorial, "Umbriel");
        tmp3rg->numpag = 557;
        tmp3rg->importe = 25;
        insertar(&p,tmp3rg);
        tmp2rg = (LISTA *)malloc(sizeof(LISTA));
        if (!tmp2rg)
        {
            puts("No hay memoria disponible");
            exit(1);
        }
        strcpy(tmp2rg->titulo, "El ultimo Caton");
        strcpy(tmp2rg->autor, "Matilde Asensi");
        strcpy(tmp2rg->codigo, "B003 124");
        strcpy(tmp2rg->editorial, "Debolsillo");
        tmp2rg->numpag = 632;
        tmp2rg->importe = 20;
        insertar(&p,tmp2rg);
        tmp1rg = (LISTA *)malloc(sizeof(LISTA));
        if (!tmp1rg)
        {
            puts("No hay memoria disponible");
            exit(1);
        }
        strcpy(tmp1rg->titulo, "Yo Augusto");
        strcpy(tmp1rg->autor, "Ernesto Ekaizer");
        strcpy(tmp1rg->codigo, "C003 125");
        strcpy(tmp1rg->editorial, "Aguilar");
        tmp1rg->numpag = 1020;
        tmp1rg->importe = 50;
        insertar(&p,tmp3rg);
         clrscr();
        listar(p);
        getch();
        clrscr();
        eliminar(&p, tmp1rg);
        listar(p);
        getch();
        clrscr();
        eliminar(&p, tmp2rg);
        listar(p);
        getch();
        clrscr();
        eliminar(&p, tmp3rg);
        listar(p);
}
```

```
int insertar(LISTA **p, LISTA *elemento)
{
        char tmp[40];
        LISTA *actual = *p;
        LISTA *anterior = NULL;
        LISTA *nuevo;
        //almacena el titulo en la matriz temporal
        strcpy(tmp, elemento -> titulo);
        //busca un titulo mayor o igual
        while (actual != NULL && strcmp(actual -> titulo, tmp) < 0)
                {
                        anterior = actual;
                        actual = actual -> lazo;
                }
        //crea un nuevo registro
        nuevo = crear(elemento);
        if ((nuevo == NULL)) return 0; // No hay memoria suficiente
        nuevo ->lazo = actual;
        if (anterior == NULL)
                //el puntero a la lista apunta al nuevo registro
                *p = nuevo;
        else
                //el registro anterior apunta al nuevo registro
                anterior -> lazo = nuevo;
        return 1;
}

void listar(LISTA *p)
{
        LISTA *actual = p;
        puts("\n***");
        while (actual)
        {
                printf ("\n%s\n", actual ->titulo);
                printf ("AUTOR              :%s\n",actual -> autor);
                printf ("Nº de PAGINAS   :%d\n",actual -> numpag);
                printf ("Importe           :%-9.2f\n",actual -> importe);
                printf ("Codigo            :%s\n",actual -> codigo);
                printf ("Editorial         :%s\n",actual -> editorial);
                actual = actual -> lazo;
        }
}

int examinar(LISTA *p, LISTA *elemento)
{
        LISTA *actual = p;
        while (actual)
                {
       //busca un registro con el mismo titulo
```

```
                        if(!strcmp(elemento -> titulo, actual -> titulo)) break;
                        actual = actual -> lazo;
                }
                return ((actual == NULL)?(0):(1));
}

int eliminar (LISTA **p, LISTA *elemento)
{
        LISTA *actual = *p;
        LISTA *anterior = NULL;
        //recorre la lista comparando el campo titulo
        while (actual != NULL && strcmp(actual->titulo, elemento->titulo) != 0)
        {
                anterior = actual;
                actual = actual -> lazo;
        }
        if (actual != NULL && anterior == NULL)
  {
        //el registro a eliminar es el primero de la lista.
        //el puntero a la lista apuntara al siguiente registro
                *p = actual -> lazo;
                free(actual);
        }
        else if (actual != NULL && anterior != NULL)
        // el registro anterior apuntara al siguiente
        {
                anterior -> lazo = actual -> lazo;
                free (actual);
        }
}

void borrar(LISTA **p)
{
        LISTA *actual = *p;
        LISTA *anterior;
         //elimina cada elemento de la lista
        while (actual != NULL)
        {
                anterior = actual;
                actual = actual -> lazo;
                free(anterior);
        }
        *p = NULL;
}

LISTA *crear(LISTA *elemento)
{
        LISTA *registro;
        registro =(LISTA*)malloc(sizeof(LISTA));
        if (registro == NULL) return NULL;
```

```
        *registro = *elemento; //copia los campos
        return registro;
}
```

8.8.- Árboles binarios

Un árbol es una estructura que contiene un registro raíz del que pueden partir varios caminos. Cada camino a su vez puede tener un registro con otros caminos. A cada registro se le llama **nodo**. En un árbol binario, los nodos pueden crear 0, 1 o 2 subárboles

El acceso a los nodos se puede realizar de diferentes formas, por ejemplo:

Recorrido prefijo: Se comienza en la raíz , se sigue por el subárbol izquierdo y luego por el derecho.

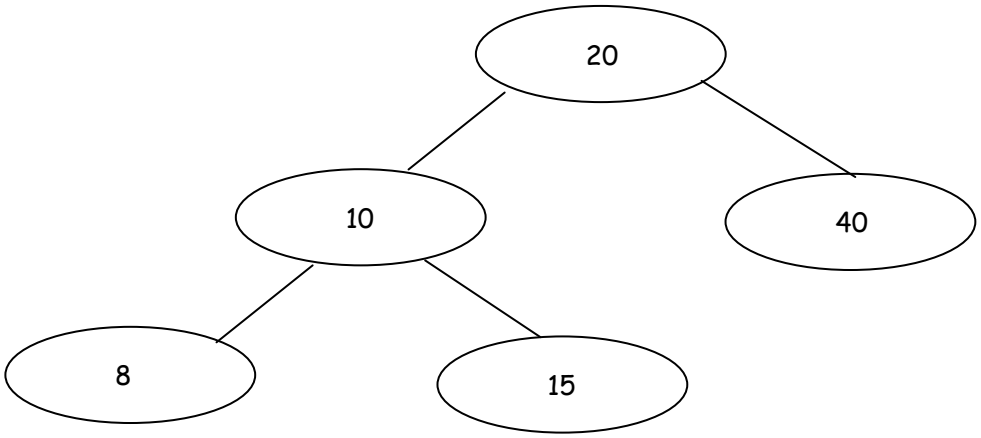
Recorrido postfijo: Se comienza por el subárbol izquierdo, después el subárbol derecho y finalmente la raíz.

Recorrido infijo: Se comienza por el subárbol izquierdo, luego la raíz y finalmente el subárbol derecho.

Si cada nodo o registro tiene, entre otros datos, un campo que se indexa como una clave, se puede organizar el árbol para que el lado izquierdo del nodo sea siempre menor que el índice y el lado derecha mayor. De esta manera las operaciones de búsqueda, inserción y borrado se agilizarán bastante, pues dependerán de *la profundidad* del árbol (el **nivel**), que, al estar ordenado, cumplirá la relación:

$$\textbf{numero de niveles} = \log_2(\textbf{numero de nodos})$$

Lo que significa que si un árbol contiene un millón de nodos y está ordenado, sólo tendrá 20 niveles. Debido a esto, los algoritmos se deben diseñar dependiendo exclusivamente del número de niveles, no del número de nodos.



Las operaciones básicas sobre un árbol binario serán similares a las de una lista:

Insertar: Añade un nodo en el árbol

Listar: Lista todos los nodos del árbol

Examinar: Indica si un nodo está presente

Eliminar: Elimina un nodo de un árbol

Borrar: Borra todo el árbol

Si se considera un ejemplo análogo al de la cola, la definición de la estructura del árbol es:

```
struct ARBOL
{
        char titulo[60];
        ARBOL *lazoizq;
        ARBOL *lazoder;
        char autor[40];
        int numpag;
        float importe;
        char codigo[20];
        char editorial[40];
};
```

y los prototipos de estas funciones básicas son;

```
int insertar(ARBOL**, ARBOL*);
void listar_arbol (ARBOL*);
int presente(ARBOL*, ARBOL*);
int eliminar(ARBOL**, ARBOL*);
void borrar(ARBOL*);
ARBOL *crear(ARBOL*);
void listar_info(ARBOL*);
```

Las variables son similares a las del programa de la estructura LISTA, la diferencia estriba en los dos lados de cada registro, necesarios para encadenar el árbol binario.

También se considerará que el índice a ordenar será el mismo que en el caso anterior: el título.

Una característica de estas funciones es la presencia de la *recursividad*. La función ***listar_arbol()*** , por ejemplo, se llama a sí misma dos veces, el algoritmo es el siguiente:

```
if (raiz)
{
        listar_arbol(raíz -> lazoizq);
        //escribir información del registro
        listar_arbol(raíz -> lazoder);
}
```

La recursión termina cuando se llega al final del camino, es decir cuando se llega a un nodo que no tiene hijos, donde los lazos izquierdo y derecho son nulos.

La función **insertar()** recorre el árbol comenzando por la raíz y siguiendo el órden del índice, por el lado izquierdo si es menor y por el lado derecho si es mayor. Si se

encuentra un campo que es igual la función termina, indicando la duplicidad. El bucle termina cuando se llega al fondo del árbol. En ese momento el último nodo se encadena con el nodo recién creado; a través del lazo izquierdo si el nuevo valor es menor y con el lado derecho si es mayor.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
struct ARBOL
{
        char titulo[60];
        ARBOL *lazoizq;
        ARBOL *lazoder;
        char autor[40];
        int numpag;
        float importe;
        char codigo[20];
        char editorial[40];
};
int insertar(ARBOL**, ARBOL*);
void listar_arbol (ARBOL*);
int presente(ARBOL*, ARBOL*);
int eliminar(ARBOL**, ARBOL*);
void borrar(ARBOL*);
ARBOL *crear(ARBOL*);
void listar_info(ARBOL*);

void main(void)
{
        ARBOL *tmp1rg, *tmp2rg, *tmp3rg;
        ARBOL *p;
        p = NULL;
        tmp3rg = (ARBOL *)malloc(sizeof(ARBOL));
        if (!tmp3rg)
        {
                puts("No hay memoria disponible");
                exit(1);
        }
        strcpy(tmp3rg->titulo, "El codigo da Vinci");
        strcpy(tmp3rg->autor, "Dan Brown");
        strcpy(tmp3rg->codigo, "A003 123");
        strcpy(tmp3rg->editorial, "Umbriel");
        tmp3rg->numpag = 557;
        tmp3rg->importe = 25;
        insertar(&p,tmp3rg);

        tmp2rg = (ARBOL *)malloc(sizeof(ARBOL));
        if (!tmp2rg)
```

```
        {
                puts("No hay memoria disponible");
                exit(1);
        }
        strcpy(tmp2rg->titulo, "El ultimo Caton");
        strcpy(tmp2rg->autor, "Matilde Asensi");
        strcpy(tmp2rg->codigo, "B003 124");
        strcpy(tmp2rg->editorial, "Debolsillo");
        tmp2rg->numpag = 632;
        tmp2rg->importe = 20;
        insertar(&p,tmp2rg);

        tmp1rg = (ARBOL *)malloc(sizeof(ARBOL));
        if (!tmp1rg)
        {
                puts("No hay memoria disponible");
                exit(1);
        }
        strcpy(tmp1rg->titulo, "Yo Augusto");
        strcpy(tmp1rg->autor, "Ernesto Ekaizer");
        strcpy(tmp1rg->codigo, "C003 125");
        strcpy(tmp1rg->editorial, "Aguilar");
        tmp1rg->numpag = 1020;
        tmp1rg->importe = 50;
        insertar(&p,tmp3rg);
   clrscr();
        listar_arbol(p);
        getch();
   clrscr();
        eliminar(&p, tmp1rg);
        listar_arbol(p);
        getch();
   clrscr();
        eliminar(&p, tmp2rg);
        listar_arbol(p);
        getch();
   clrscr();
        eliminar(&p, tmp3rg);
          listar_arbol(p);
}

int insertar (ARBOL**raiz, ARBOL*elemento)
{
        //devuelve 0 si no hay memoria disponible
        //devuelve 1 se se ha insertado el nodo en el arbol
        ARBOL *p = NULL;
        ARBOL *actual = *raiz;
        ARBOL *nuevo;
        int encontrado = 0;
        int i;
```

```
        while (actual   != NULL && !encontrado)
        {
                if (!(i = strcmp(elemento->titulo, actual -> titulo))) encontrado = 1;
                else
                {
                        p = actual;
                        if (i < 0)
                                actual = actual -> lazoizq;
                        else
                                actual = actual -> lazoder;
                }
        }
        if(!encontrado)
  {
                if (p == NULL)
                {
                        *raiz = crear(elemento);
                        if (raiz == NULL)
                        return 0;
                        (*raiz) -> lazoizq = (*raiz) -> lazoder = NULL;
                }
                else
                {
                        nuevo = crear(elemento);
                        if (nuevo == NULL)
                                return 0;
                        nuevo -> lazoizq = nuevo -> lazoder = NULL;
                        if (strcmp(elemento ->titulo, p -> titulo) < 0)
                                p -> lazoizq = nuevo;
                        else
                                p -> lazoder = nuevo;
                }
        }
        return 1;
}

void listar_arbol(ARBOL *raiz)
{
        if (raiz)
        {
                listar_arbol(raiz -> lazoizq);
                listar_info(raiz);
                listar_arbol(raiz -> lazoder);
        }
}

void listar_info(ARBOL *raiz)
{
        puts ("\n***");
        printf("\n%s\n",raiz ->titulo);
```

```
        printf ("AUTOR              :%s\n",raiz -> autor);
        printf ("Nº de PAGINAS   :%d\n",raiz -> numpag);
        printf ("Importe          :%-9.2f\n",raiz -> importe);
        printf ("Codigo           :%s\n",raiz -> codigo);
        printf ("Editorial        :%s\n",raiz -> editorial);
}

int presente (ARBOL *raiz, ARBOL *elemento)
{
        //devuelve 1 si se ha encontrado el elemento y 0 si no
        ARBOL *actual = raiz;
        int encontrado = 0;
        int i;
        while (actual != NULL && !encontrado)
        {
                if (!(i = strcmp(elemento -> titulo, actual -> titulo)))
                        encontrado = 1;
                else
                {
                        if (i < 0)
                                actual = actual -> lazoizq;
                        else
                                actual = actual -> lazoder;
                }
        }
        return encontrado;
}

int eliminar (ARBOL **raiz, ARBOL *elemento)
{
        //devuelve 1 si se ha encontrado el nodo y 0 si no
        ARBOL *anterior = NULL;
        ARBOL *actual = *raiz;
        ARBOL *r, *s, *p;
        int devuelve = 0;
        int i;
        while (actual != NULL && !devuelve)
        {
                if(!(i = strcmp(elemento -> titulo, actual -> titulo))) devuelve = 1;
                else
                {
                        anterior = actual;
                        if (i < 0) actual = actual ->lazoizq;
                        else actual = actual -> lazoder;
                }
        }

        if (devuelve)
        {
                if(actual ->lazoizq == NULL) r = actual -> lazoder;
```

```
                else
                if(actual ->lazoder == NULL) r = actual -> lazoizq;
                else
                {
                        p = actual;
                        r = actual -> lazoder;
                        s = r -> lazoizq;
                        while (s != NULL)
                        {
                                p = r;
                                r = s;
                                s = r -> lazoizq;
                        }
                        if (p != actual)
                        {
                                p -> lazoizq = r -> lazoder;
                                r -> lazoder = actual ->lazoder;
                        }
                        r -> lazoizq = actual -> lazoizq;
                }
                if (anterior == NULL) *raiz = r;
                else if (actual == anterior ->lazoizq) anterior -> lazoizq = r;
                else anterior -> lazoder = r;
    free (actual);

        }
  return devuelve;
}

void borrar(ARBOL *raiz)
{
        if (raiz)
        {
                borrar (raiz -> lazoizq);
                borrar (raiz -> lazoder);
                free(raiz);
        }
        raiz = NULL;
}

ARBOL *crear(ARBOL *elemento)
{
        ARBOL *nodo;
        nodo = (ARBOL*)malloc(sizeof(ARBOL));
        if (nodo == NULL) return NULL; //no hay memoria disponible
        *nodo = *elemento; //almacena la informacion
        return nodo;
}
```

8.9.- Construcción de una lista genérica

Una lista que pueda servir para diferentes tipos de registros evita tener que programar una estructura particular para cada aplicación. Para crearla, será necesario definir un registro inicial (o cabeza de la lista) que contenga el tamaño de los nodos que vaya a tener la lista (suponiendo que todos son de igual tamaño). También se deberá tener un puntero al primer nodo de la lista. Por último, no podrá escribir la información que contienen los nodos, pues los campos dentro de cada uno variarán de una aplicación a otra, por lo que habrá que apuntar a una función dada por el usuario, que será el encargado de programar dicha función.

La estructura de los nodos comprende:

un **campo de encadenamiento**

un **campo de datos**

El campo de datos debe definirse como un puntero a caracteres puesto que el tamaño puede cambiar en cada aplicación. Las operaciones básicas sobre esta lista serán:

Crear la estructura lista

Insertar un elemento

Listar la información

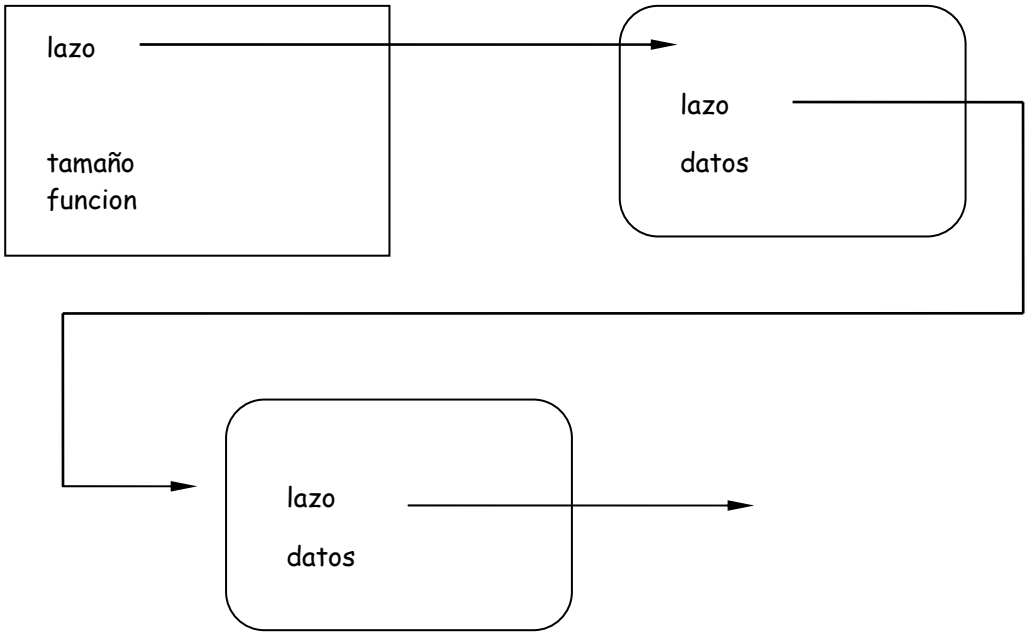
Extraer un elemento

La lista genérica se puede generalizar aún mas con otras operaciones, añadiendo funciones que permitan extraer el primer elemento, o algún elemento en particular, si se mantuviese ordenada la lista con el criterio de una función definida por el usuario.

Las declaraciones y prototipos de las funciones básicas se muestran a continuación:

```
//Definiciones de la lista genérica
typedef void (*funcion_lis)(char *);
struct NODO
{
        NODO *lazo;
        char *info;
}
struct LISTA
{
        NODO *lazo;
        int lg_nodo;
        fncion_lis visualizar;
}
// Prototipos
int crear_lista (LISTA**, int, funcion_lis);
int insertar_lista(LISTA**, char *);
```

```
void listar (LISTA *);
char *extraer_lista(LISTA**);
```



La función **crear_lista** crea el registro inicial de la lista con los siguientes campos:

***lazo** Puntero al primer nodo de la lista que se inicializa a nulo

lg_nodo Tamaño en bytes, definido por el usuario que ocupa la información que almacenan los elementos de la lista

visualizar Puntero que señala a la función definida por el usuario para listar la información de los nodos.

La función **insertar_lista()** reserva espacio en memoria para insertar el nuevo nodo. Como cada nodo no contiene la información, sino un puntero para referenciarla, debe reservar otra zona de memoria para almacenar la información relativa al nuevo nodo . La información se traspasa byte a byte, por medio de un bucle **for** hasta mover **lg_nodo** bytes. Para encadenar el nuevo nodo, la función recorre la lista hasta el final, haciendo que el último nodo apunte al nuevo. Si el nuevo nodo es el primero de la lista, se debe modificar el puntero inicial.

La función **extraer_lista()** devuelve un puntero dirigido a la información del primer nodo de la lista . Para ser genérico, este puntero debe ser de tipo **char**. La información se traspasa a través de un bucle **for** byte a byte, desde el nodo que se va a eliminar a otra zona de memoria, que se reserva con una llamada previa a **malloc()**. Una vez hecho el traspaso de información, el puntero a la lista deberá apuntar al siguiente nodo. También se deberá liberar la memoria que ocupaba el nodo.

La función **listar()** recorre la lista e invoca en cada paso a la función de visualización definida por el usuario.

La programación de las funciones básicas es la siguiente:

```
 /*Funciones de la lista generica*/

int crear_lista (LISTA **p, int n, funcion_lis fun_lis)
{
        *p = malloc (sizeof(LISTA));
        if(p == NULL) return 0; //no hay memoria disponible
        (*p) -> lazo = NULL;//se inicializa a nulo
        (*p) -> lg_nodo = n;//tamaño en bytes
        (*p) -> visualizar = fun_lis; //funcion del usuario
        return 1;
}

int insertar_lista(LISTA**p, char *datos)
{
        NODO *nuevo;
        NODO *actual;
        NODO *anterior;
        int i;
        //reserva memoria para el nuevo nodo
        nuevo = (NODO*)malloc(sizeof(NODO));
        if ((nuevo == NULL)return 0;
        // reserva memoria para la informacion del nuevo nodo
        nuevo -> info = (car*)malloc((*p)->lg_nodo);
        if (nuevo ->infor == NULL)return 0;
        //se insertara al final de la lista
        nuevo -> lazo = NULL
        //almacena la nueva informacion
        for (i = 0; i < (*p)->lg_nodo; i++)
                nuevo ->info[i] = datos[i];
        //busca el ultimo nodo de la lista
        if ((*p) -> lazo)
        {
                anterior = (*p) -> lazo;
                actual = anterior -> lazo;
                while (actual != NULL)
                {
                        anterior = actual;
                        actual = actual -> lazo;
                }
                //encadena el nuevo nodo: es el ultimo de la lista
                anterior -> lazo = nuevo;
        }
        else
                //el nuevo nodo es el primero de la lista
                (*p) -> lazo = nuevo;
        return 1; //terminacion correcta
```

```
}

void listar (LISTA *p)
{
        NODO *actual = p -> lazo;
        //recorre la lista y llama a la funcion definida por el usuario
        while (actual)
        {
                p -> visualizar(actual -> info);
                actual = actual -> lazo;
        }
}

char *extraer_lista(LISTA**p);
{
        NODO *a_extraer = (*p) -> lazo;
        char *devuelve;
        int i;
        if (a_extraer == NULL) return NULL; //lista vacia
        devuelve = (char*)malloc((*p) ->lg_nodo);
        if (devuelve == NULL) return NULL //no hay memoria disponible
        //almacena la informacion del nodo que se va a extraer
        for (i = 0; i < (*p) ->lg_nodo; i++)
                devuelve[i] = a_extraer -> info[i];
        (*p) -> lazo = a_extraer -> lazo;
        free (a_extraer); //libera la memoria
        //la funcion devuelve un puntero a la informacion que contenia el modulo
        return (devuelve);
}
```
