ORGANIZACIÓN LÓGICA DE DATOS. ESTRUCTURAS ESTATICAS

A. INTRODUCCION

Las estructuras estáticas son conjuntos de datos homogéneos que se trata como una sola unidad. Hay dos tipos de estructuras: internas y externas (en soportes físicos).

Internas: se almacenan en la memoria principal del ordenador, que es la memoria RAM.

Array o tabla o matriz: se utilizan en cualquier lenguaje de programación para almacenar datos que son del mismo tipo o demás.

B. Conceptos y definiciones

1. Tabla o array: consiste en un (conjunto) numero fijo y ordenado de elementos, todos del mismo tipo y bajo un mismo nombre.

2. Componentes: son los elementos que forman el array.

3. Índices: la posición de cada componente dentro del array viene determinado por el índice o índices (uno o varios).

4. Dimensiones: de una tabla o array es el número de índices que se utiliza.

5. Longitud: es el tamaño de la tabla o array, el número de elementos que contiene el array.

6. Tipo de array: el tipo de sus componentes.

Todos los componentes de un array los vamos a utilizar como variables y se va a poder realizar operaciones con los arrays.

Representación grafica de un array:

Alumnos

Array	Jose	Rosa	Luis	Lola	Andres	Maria

Nombre del array: Alumnos.

Componentes: Alumnos(1), Alumnos(2), ... , Alumnos(7).

Índices: 1

Dimensión:1

Longitud: 7

Tipo de array: Alfanumérico.

C. TIPOS DE ARRAYS

- **Unidimensionales**
- **Bidimensionales**
- **Multidimensionales**

1. Unidimensionales: van a tener una sola dirección y por lo tanto un solo índice. Se les llama a este tipo de arrays Vectores.

Si queremos intercambiar el contenido entre dos vectores, vamos a declarar primero una variable auxiliar.

AUX=Alumnos(1)

Alumnos(1)=Alumnos(2)

Alumnos(2)=AUX

SWITCH

Es un campo de memoria que puede tomar dos valores exclusivos (uno u otro) y se utiliza para lo siguiente:

− Recordar en un determinado punto de un programa la ocurrencia o no de un suceso anterior.

− Salir de un bucle.

− Para decidir en una instrucción alternativa que acción realizar.

− Para hacer que dos acciones diferentes se ejecuten alternativamente dentro de un bucle.

a) OPERACIONES CON VECTORES: asignación, lectura/escritura, recorrido y actualización.

- Asignación: es dar un valor a algún elemento del vector. Los vectores son sucesiones ordenadas de elementos. Cuando veamos x[1], x[2], ... , x[n] sabremos que es un vector. En los vectores no es obligatorio empezar desde 1, pueden empezar desde 0, un número negativo o un número positivo distinto de 1.

Tipo

Array[dimensión]tipo de datos: nombre del array

Var

Nombre del array: N (variable que se va a utilizar para cada elemento del array).

- Lectura y escritura:

Leer (A): leer el array entero.

Escribir(A): escribir el array entero.

Leer V(1): leer el elemento 1 del array.

- Recorrido o acceso secuencial al vector: es la operación con la que vamos a realizar una acción general sobre todos los elementos del vector. Dependiendo de la dimensión del array se hace de una

manera o de otra. En los bidimensionales se realiza de forma más compleja. Se puede leer del principio a fin o del final al principio.

- Actualización: esta compuesta de tres operaciones: añadir, insertar y borrar.

    - ♦ Añadir: significa agregar un elemento al final del array. Se trata de una asignación. Esto lo podemos hacer cuando tenemos uno o más elementos al final del array vacíos. Tenemos que fijarnos en la dimensión.

    - ♦ Insertar: introducir un elemento entre los ya existentes. Tiene que haber elementos vacíos y tenemos que cambiar los que van después del que insertamos.

2. Bidimensionales: Son arrays de dos dimensiones y se les conoce con el nombre de matrices. Tienen 2 índices por lo que cada componente de la matriz se direcciona con el nombre de la matriz seguido de dos índices separados por comas.

Tratamiento secuencial o recorrido de una matriz

El recorrido de una matriz se realiza mediante el anidamiento de dos bucles desde. El bucle externo recorre cada una de las filas y el interno todos los componentes de una fila. Este tratamiento también se puede realizar por columnas en casos en los que nos interese y además en ambos casos en orden creciente y decreciente para las filas y columnas.

3. Arrays MULTIDIMENSIONALES: Son los arrays desde 3 hasta n dimensiones. También se les llama poliedros. Es necesario que los n subíndices estén declarados o identificados, para poder localizar un elemento individual del poliedro. Un array de n dimensiones se puede identificar de la siguiente manera.

A [P1:U1, P2:U2, ... , PN:UN]

A [I1, I2, ... , In] (estos I son los índices que nos van a ayudar a localizar un elemento).

Componentes:

P[1,1,1], P[1,1,2]....P[1,1,n]

P[1,2,1], P[1,2,2]....P[1,2,n]

P[1,n,n],.....P[n,n,n].

Índices:

[1.....n]

[1.....n]

[1.....n]

Longitud: N.

Tratamiento secuencial o recorrido de un poliedro

Requiere tantos bucles desde anidados como dimensiones tenga el poliedro. Cada índice de un bucle recorre una dimensión y puede hacerse en cualquier orden de anidamiento.

SUBALGORITMOS O SUBPROGRAMAS: PROCEDIMIENTOS Y FUNCIONES

El método para solucionar un programa complejo es dividirlo en subprogramas que sean sencillos de resolver. Cuando se trabaja de esta manera existirá un algoritmo principal o conductor que transfiera el control a los distintos subalgoritmos, los cuales, cuando terminen su tarea, devolverán el control al algoritmo que los llamo. Existen dos tipos de subprogramas: funciones y procedimientos.

1. FUNCIONES

Una función a partir de uno o más valores llamados argumentos o parámetros devuelve un resultado en el nombre de la función.

Declaración de funciones: una función constaría de:

a) Cabecera: también se le llama definición de la función. En la definición de la función deben figurar una serie de parámetros, llamados parámetros formales para que cuando se llame a la función se pueda establecer una correspondencia 1 a 1 y de izquierda a derecha entre los parámetros actuales y formales.

b) En el cuerpo de la función estarán el bloque de declaraciones y el bloque de instrucciones. Debe incluir una instrucción mediante la cual la función tomaría un valor para devolverlo al programa principal o llamador.

Función < nombre de la función > (lista de parámetros formales): tipo de dato devuelto.

Declaración de las variables locales.

Instrucciones (Inicio).

Los tipos de datos validos como resultado de una función son: ordinales, reales, cadenas, punteros y lógicos.

En una lista de parámetros formales es posible separar estos parámetros con **;** y hay que indicar cada tipo.

2. PROCEDIMIENTOS

Un procedimiento es un subprograma o subalgoritmo que realiza una tarea especifica y que puede ser definido mediante 0, 1 o n parámetros.

Tanto la entrada de información al procedimiento como la devolución de resultados desde el procedimiento al programa principal o llamador o conductor se realiza a través de los parámetros. El nombre de un procedimiento no esta asociado a ninguno de los resultados que obtiene. La llamada a un procedimiento se realiza con la instrucción Llamar_a o directamente con el nombre del procedimiento, es decir:

<nombre del procedimiento> (lista de parámetros que se llaman actuales o reales).

Declaración de procedimientos: es similar a la de una función excepto que el nombre del procedimiento no se encuentra asociado a ningún resultado.

Procedimiento <nombre del procedimiento>(lista de parámetros formales)

(declaración de variables locales)

Inicio

.

.

.

Fin_procedimiento

La lista de parámetros formales se debe separar por **;** y debemos indicar el tipo de cada uno de ellos.

3. VARIABLES LOCALES Y GLOBALES

Las variables utilizadas en los programas principales y en los subprogramas se clasifican en dos tipos: variables locales y variables globales.

- Variables locales: una variable es local si esta declarada y definida dentro de un subprograma y es distinta de las variables que tengan el mismo nombre que hayan sido declaradas en el programa principal.

Cuando otro subprograma utiliza el mismo nombre al declarar una variable ambas serán distintas y tendrán diferentes posiciones de memoria, por lo tanto solo tendrán vigencia en el subprograma donde han sido declaradas. El uso de variables locales tiene la ventaja de hacer a los subprogramas independientes y la comunicación con el programa principal se realizara exclusivamente a través de los parámetros.

- Variables globales: una variable es global cuando el ámbito (SCOPE) en el que dicha variable se conoce es el programa completo y debe haber sido declarada en el programa principal. Tienen la ventaja de compartir información de diferentes subprogramas. Sin una correspondiente entrada en la lista de parámetros.

4. COMUNICACIÓN CON SUBPROGRAMAS: PASO DE PARÁMETROS

Cuando un programa llama a un subprograma, la información se comunica a través de una lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. Los parámetros actuales son sustituidos o utilizados en lugar de los parámetros formales. La declaración del subprograma se hace:

Procedimiento <nombre>(tipo de dato: dato; tipo de dato: dato; ...) Parámetros formales.

Y la llamada al subprograma se hace con:

Llamar_a <nombre del procedimiento> (valores que va a tomar el procedimiento:)

Existen dos métodos para establecer la correspondencia de parámetros:

- Correspondencia posicional: es la que se utiliza en C. La correspondencia se establece emparejando los parámetros reales y formales, según su posición en las listas. Fi se corresponde con Ai (F: formal, A: actual).
- Correspondencia por el nombre explicito o método de paso de parámetros por el nombre. En este método en

las llamadas se indica explícitamente la correspondencia entre los parámetros reales y formales.

5. TRANSMISIÓN DE PARÁMETROS

Los métodos de transmisión de parámetros más utilizados son:

- Por valor: el paso de un parámetro por valor significa que el valor del argumento (parámetro actual o real) se asigna al parámetro formal, es decir, antes de que el subprograma comience a ejecutarse el argumento toma un valor especifico. Este valor se copia en el correspondiente parámetro formal dentro del subprograma. Una vez que el procedimiento empieza a ejecutarse, cualquier cambio de valor de uno de los parámetros formales no se refleja en un cambio en el correspondiente argumento (parámetros reales), es decir, que cuando el subprograma termine de ejecutarse el argumento actual (parámetros actuales) tendrá el mismo valor que cuando el subprograma comenzó, independientemente de lo que les haya sucedido a los parámetros formales. A estos parámetros se les llama parámetros valor. Y cuando hagamos la declaración tendremos que hacerlo a través de E (entrada).
- Por referencia: el paso de parámetros por referencia, que se les llama también parámetros variables, significa que la posición o dirección de (no del valor) del argumento o parámetro actual se envía al subprograma, es decir, que si a un parámetro formal se la da el atributo de parámetro por referencia y si el parámetro actual es una variable, entonces un cambio en el parámetro formal se refleja en un cambio en el correspondiente parámetro actual, porque ambos tienen la misma posición de memoria. Para indicar que deseamos transmitir un parámetro por dirección cuando hagamos la declaración de los parámetros, lo haremos a través de E/S o S.

6. EFECTOS LATERALES

Las modificaciones que se produzcan mediante una función o procedimiento en los elementos situados fuera del subprograma se llaman efectos laterales. En algunos casos son beneficiosos para la programación pero no es recomendable no utilizarlos.

- Efectos laterales en procedimientos: la comunicación del procedimiento con el resto del programa se tiene que hacer a través de los parámetros. Cualquier otra comunicación entre el procedimiento y el resto del programa son efectos laterales. Estos efectos son perjudiciales en la mayoría de los casos. Si un procedimiento modifica una variable global (distinta de un parámetro actual) esto es un efecto lateral, por ello excepto en contadas ocasiones, no debe aparecer en la declaración del procedimiento. Si se necesita una variable temporal en un procedimiento es mejor utilizar una variable local, que habría que declarar en el procedimiento.

Si queremos que el programa modifique el valor de una variable global es mejor utilizar un parámetro formal, variable en la declaración del procedimiento, y después usar la variable global como el parámetro actual en una llamada al procedimiento.

En general, se debe seguir la regla de no utilizar ninguna variable global en procedimiento, aunque esto no significa que los procedimientos no puedan manipular variables globales.

En el cuerpo de un procedimiento es mejor utilizar un parámetro formal o una variable local. En los lenguajes donde es posible declarar constantes, se puede utilizar constantes globales ya que no pueden ser modificadas por el procedimiento.

- Efectos laterales en funciones: Una función toma los valores de los argumentos y devuelve un único valor. Sin embargo, al igual que en los procedimientos, una función puede hacer cosas similares a ellos.

Una función puede tener parámetros variables, además de parámetros valor (no devuelven nada) en una lista

de parámetros formales. Una función puede cambiar el contenido de una variable global y ejecutar instrucciones de E/S. Estas operaciones se conocen como parámetros laterales y se deben evitar. Toda la información que se transmite entre procedimientos y funciones debe realizarse (al igual que en los procedimientos) a través de la lista de parámetros y no a través de las variables globales. Esto convertirá al procedimiento o función en módulos independientes que pueden ser comprobados y depurados por si solos, lo que evitara preocuparnos por el resto de las partes del programa.

7. RECURSIVIDAD

Un subprograma puede llamar a cualquier otro subprograma y este a otro y así sucesivamente, que es lo que se llama anidación. Pero también se pueden llamar a si mismos.

- llamar_a A

_____

- Llamar_a B
- Llamar_a A

Otra función o procedimiento que se puede llamar así mismo se llama recursivo. Es una herramienta muy potente en algunas aplicaciones, sobre todo en calculo, y puede ser utilizada como una alternativa a la repeticion o estructura repetitiva. La escritura de un procedimiento o función recursiva es similar a los procedimientos o funciones normales, sin embargo, para evitar que la recursion continúe indefinidamente es preciso incluir una condición de terminación.

La razón de que existan lenguajes que admitan la recursividad se debe a la existencia de estructuras especificas tipo pilas y memorias dinámicas. Las direcciones de retorno y el estado de cada subprograma se guardan en estructuras tipo pilas.

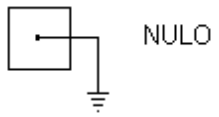
TEMA 7: ESTRUCTURAS DINAMICAS LINEALES DE DATOS

Hay otro tipo de estructuras que pueden ampliar o limitar su tamaño mientras se ejecuta el programa: estructuras dinámicas

ESTRUCTURAS DINAMICAS

Estas estructuras son generadas mediante un tipo de dato conocido con el nombre de puntero. Una variable de tipo puntero almacena la dirección o posición de otra variable. La principal ventaja que tiene usar este tipo de datos es que se pueden adquirir posiciones de memoria cuando se necesitan y liberarlas cuando ya no se necesitan. De esta manera se pueden crear estructuras dinámicas que se expandan o contraigan según agreguemos o eliminemos elementos. Un puntero es una variable que almacena la posición de una variable dinámica de un tipo determinado y llega a la variable numérica apuntada. El tipo de dato puede ser simple o estructurado y podemos realizar las siguientes operaciones:

- Inicialización:



- Comparación: p=q, p<>q

- Asignación: p!q

Creación de variables dinámicas: consiste en reservar un espacio en memoria para la variable dinámica. Reservar (p).

Eliminación de variables dinámicas: consiste en desocupar el espacio en memoria que está siendo utilizado por las variables dinámicas. Liberar (p)

Una variable dinámica es una variable simple o estructurada de datos sin nombre y creada en tiempo de ejecución, por lo tanto, para poder acceder a una variable numérica apuntada como no tiene nombre usaremos "!". Las variables dinámicas pueden intervenir en cualquier operación o expresión para una variable estática de su mismo tipo. Pueden ser de dos tipos: lineales y no lineales.

1. LINEALES

a) LISTAS

Son secuencias de 0 o más elementos de un tipo de datos almacenado en memoria. Son estructuras lineales donde cada elemento de una lista excepto el primero tiene un único predecesor y cada elemento de la lista excepto el ultimo tiene un sucesor. Al número de elementos de una lista se le llama longitud, y decimos que una lista está vacía si tiene 0 elementos. Se pueden añadir nuevos elementos o suprimirlos de cualquier posición.

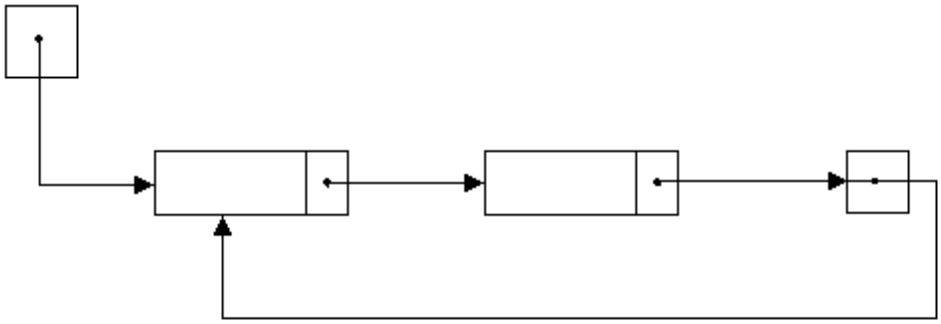
Tipos de listas lineales:

- Listas contiguas: los elementos son adyacentes en la memoria del ordenador y tienen unos limites (izquierdo y derecho o inferior y superior) que no pueden ser rebasados cuando se añade un nuevo elemento. Se implementan a través de arrays y la inserción o eliminación de un elemento necesitará una traslación por parte de los elementos de la lista, excepto para la cabecera y el final de la lista.
- Listas enlazadas: los elementos se almacenan en posiciones de memoria que no son adyacentes o contiguas, por lo que cada elemento necesita almacenar la posición del siguiente elemento de la lista. Son bastante más flexibles y potentes que las listas contiguas y la inserción o el borrado de un elemento no requiere el desplazamiento de los demás elementos de la lista. Se implementan, normalmente, de forma dinámica, pero si el lenguaje de programación no lo permite se utilizaran arrays, con lo cual tendremos limitaciones en cuanto al número de elementos que pueda contener la lista y además establece una ocupación de memoria constante.

Gráficamente se representa de la siguiente forma:

- Listas circulares: son una modificación de las listas enlazadas en las que el puntero del ultimo elemento apunta al primero de la listas.

Gráficamente se representa de la siguiente forma:



Debemos diseñar un nodo especial llamado nodo cabecera que está permanentemente asociado a la existencia de la lista y cuyo campo para almacenar información no se utiliza. Así, al efectuar un recorrido de la lista, el nodo cabecera nos permitirá detectar cuándo han sido visitados todos los demás nodos.

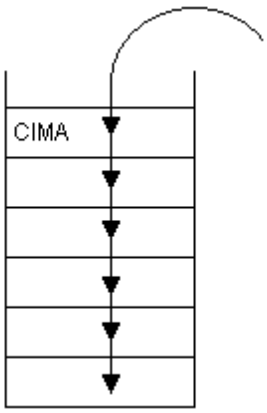
- Listas doblemente enlazadas: Se pueden recorrer tanto del final al principio como de principio a fin. Cada nodo de las listas de este tipo consta de un campo con información y de otros dos campos que son de tipo puntero y que podríamos denominar anterior y siguiente, uno desde su nodo anterior y otro de su nodo sucesor.

- Listas doblemente encadenadas circulares: en este tipo de listas el campo anterior del primer nodo apunta al ultimo nodo y el campo siguiente del ultimo nodo apunta al primero.

b) PILAS

Son unas estructuras que son más utilizadas siempre que se quieran recuperar una serie de elementos en orden inverso a como se introdujeron.

Tanto la extracción como la inserción de un elemento en la pila se realiza por la parte superior, por lo tanto, el único elemento al que podemos acceder es el ultimo, y como el ultimo elemento que se pone en la pila es el primero que se puede sacar, a estas estructuras dinámicas llamadas pilas se les conoce como LIFO (Last In First Off). Las pilas se deben implementar de forma dinámica utilizando punteros, pero si el lenguaje de programación que estamos utilizando no admite punteros, entonces tendremos que utilizar arrays, y además una variable que normalmente se le da el nombre de `cima' y es la que apunta al último elemento de la pila.



Aplicaciones de las pilas: el uso más común que suele darse a las pilas es cuando hacemos una llamada desde un programa a otro subprograma. Se utiliza las pilas para guardar el lugar desde donde se hizo la llamada y para almacenar el estado de las variables en el momento en que se hace la llamada. También se utilizan en

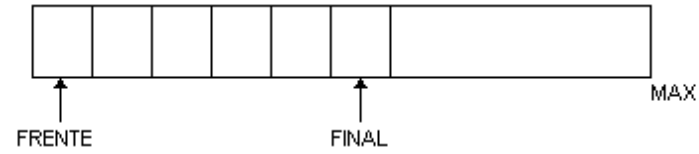
todos los algoritmos recursivos para almacenar el valor de las variables y los parámetros.

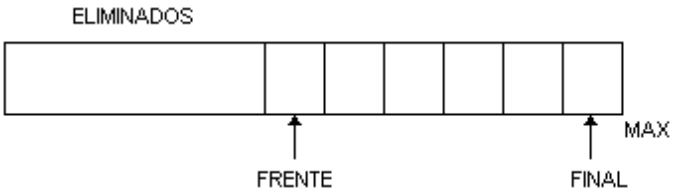
c) COLAS

Una cola es también una estructura de datos lineal en donde las eliminaciones se realizan por uno de sus extremos que normalmente se llama frente, y las inserciones se realizan por el otro extremo que normalmente se llama final. A estas estructuras se les llama FIFO (First In First Out).

Se tiene que implementar de forma dinámica usando punteros a menos que el lenguaje que utilicemos lo permita, y entonces tendremos que utilizar arrays.

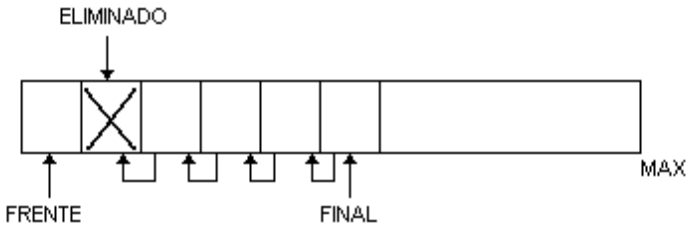
Representación grafica de colas hechas con arrays.



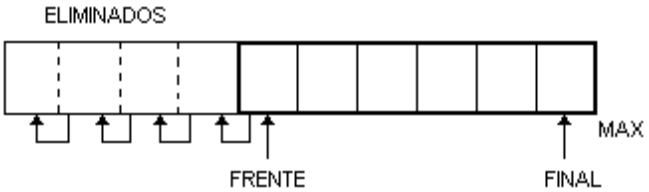


SOLUCIONES en el caso de que la variable `final' coincida con el máximo del array aunque haya posiciones libres.

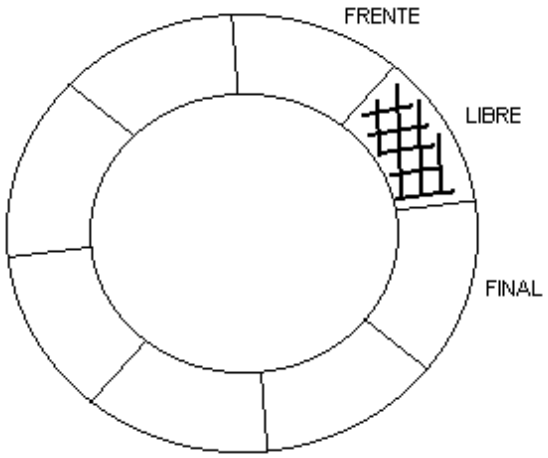
- Retroceso: consiste en mantener fijo a 1 el valor de la variable `frente' realizando un desplazamiento de una posición para todos los elementos ocupados cada vez que se produce una eliminación.



- Reestructuración: la variable `final' llega al máximo de los elementos, los elementos ocupados se deben desplazar hacia atrás, las posiciones que hagan falta para que el principio de la lista coincida con el principio del array, es decir:



- Mediante un array circular: se considera que el elemento primero sigue al elemento ultimo. Para esta implementación debemos dejar una posición libre que separe el principio y el final de la cola.



MALLOC

```
#include <stdio.h>

#include <stdlib.h>

void main ()

{

int i,j,fl,nel=0;

int **ptabla;

printf("\n\t Escribe el numero de nombres \n");

scanf("%i",&fl);

printf("\n\t Escribe el numero de columnas \n");

scanf("%i",&cl);

ptabla=(int**)malloc(fl*sizeof(int));

for (i=0;i<cl;i++)

{

ptabla[i]=(int*)malloc(cl*sizeof(int));

}

for (i=0;i<fl;i++)
```

```
{
for(j=0;j<cl;j++)
{
printf("Escribe el elemento \n");
scanf("%i",&ptabla[i][j]);
}
}
for (i=0;i<fl;i++)
{
for(j=0;j<cl;j++)
{
printf("%4i",*(*ptabla)++);
}
printf("\n");
*ptabla−=cl;
ptabla++;
}
ptabla−=fl;
for (i=0;i<fl;i++)
{
free(ptabla[i]);
}
free(ptabla);
}
```

MALLOC (2ª PARTE)

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

```
#include <stdlib.h>

crear(int **ptabla,int *cuan)

{

*ptabla=(int *)malloc(sizeof(int)**cuan);

}

rellenar(int *ptabla,int cuan)

{

int i;

for (i=0;i<cuan;i++)

{

printf("\n\t escribe el elemento: ");

scanf("%i",ptabla++);

}

}

listar(int *ptabla,int cuan)

{

int i;

for(i=0;i<cuan;i++)

{

printf("\n\t el elemento es: %i",*ptabla++);

}

fflush(stdin);

getchar();

}

ordenar(int *ptabla,int cuan)

{
```

```
fflush(stdin);
getchar();
for (i=0;i<cuan;i++)
{
for(j=0;j<cuan−1;j++)
{
if(*ptabla<*(ptabla+1))
{
aux=*ptabla+1;
*(ptabla+1)=*ptabla;
*(ptabla)=aux;
}
ptabla++;
}
ptabla−=cuan;
}
}
}
void main()
{
int opci;
int* ptabla;
int cuantos;
printf("\n\t escribe cuantos elementos \n");
scanf("%i",&cuantos);
do
```

```
{
system("clear");
printf("\n\t 1 para crear tabla");
printf("\n\t 2 para rellenar tabla");
printf("\n\t 3 listar tabla");
printf("\n\t 4 para ordenar");
printf("\n\t 5 para salir \n");
scanf("%i",&opci);
switch (opci)
{
case 1:
crear(&ptabla,&cuantos);
break;
case 2:
rellenar(&ptabla,cuantos);
break;
case 3:
listar(&ptabla,cuantos);
break;
case 4:
ordenar(&ptabla,cuantos);
break;
case 5:
free(*ptabla);
exit(0);
}
```

```
}while(opci!=5 );

}
```

ESTRUCTURAS

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

union t_variable

{

char autor[15];

int edicion;

};

struct t_biblio

{

int codigo;

char nombre[30];

char tipo;

union t_variable unvariable;

};

void listar(struct t_biblio alma[],int *cant);

void alta(struct t_biblio alma[],int *cant);

void main()

{

struct t_biblio fichas[10];

int cantidad=0,opcion;

do

{
```

```
system("clear");
printf("\n\t Escribe lo que quieres hacer 1 listar 2 para dar de alta\n\t ");
printf("\n\t Escribe la opcion \t");
scanf("%i",&opcion);
if (opcion==1)
{
listar(fichas,&cantidad);
}
else
{
alta(fichas,&cantidad);
}
}
while(opcion!=199);
}
void listar(struct t_biblio alma[],int *cant)
{
int i;
for(i=0;i<*cant;i++)
{
system("clear");
printf("\n\tnombre: %s ",alma[i].nombre);
printf("\n\tcodigo: %i ",alma[i].codigo);
printf("\n\ttipo: %c ",alma[i].tipo);
if(alma[i].tipo=='l')
{
```

```
printf("\n\tautor: %s ",alma[i].unvariable.autor);
}
else
{
printf("\n\tedicion: %i ",alma[i].unvariable.edicion);
}
fflush(stdin);
printf("\n\t pulsa una tecla para continuar");
getchar();
}
}
void alta(struct t_biblio alma[],int *cant)
{
char opci;
system("clear");
printf("\n\tescribe el tipo p(periodicos) l(libros): ");
fflush(stdin);
scanf("%c",&opci);
switch (opci)
{
case 'p':
printf("\n\tescdribe el nombre: ");
fflush(stdin);
gets(alma[*cant].nombre);
printf("\n\t escribe el codigo : ");
fflush(stdin);
```

```
scanf("%i",&alma[*cant].codigo);
printf("\n\t escribe la edicion: ");
fflush(stdin);
scanf("%i",&alma[*cant].unvariable.edicion);
alma[*cant].tipo='p';
(*cant)++;
break;
case 'l':
printf("\n\t escribe el nombre: ");
fflush(stdin);
gets(alma[*cant].nombre);
printf("\n\t escribe el codigo: ");
fflush(stdin);
scanf("%i",&alma[*cant].codigo);
printf("\n\t escribe el autor: ");
fflush(stdin);
gets(alma[*cant].unvariable.autor);
alma[*cant].tipo='l';
(*cant)++;
break;
}
}
```

STRING DINAMICAS

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

```
void main ()
{
int i,j,fl,nel=0;
char nombre[50];
char **ptabla;
printf("\n\t Escribe el numero de nombres \n");
scanf("%i",&fl);
ptabla=(char**)malloc(fl*sizeof(char));
for (i=0;i<fl;i++)
{
printf("Escibe el nombre");
gets(nombre);
ptabla[i]=(char*)malloc((strlen(nombre)+1)*sizeof(char));
strcpy(ptabla[i],nombre);
}
for (i=0;i<fl;i++)
printf("%4s\n",*ptabla++);
ptabla−=fl;
for (i=0;i<fl;i++)
{
free(ptabla[i]);
}
free(ptabla);
}
```

RECURSIVIDAD

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

```
#include <stdlib.h>

void main()

int factorial(int n)

{

res=num*factorial−1;

return(res);

}

{

int num,res;

printf("Escribe el numero");

scanf("%i",&num);

if (num > 1)

{

res=num*factorial(num−1);

}

return res;

printf("resultado es %i",res);

}
```

MENU

```
#include <stdlib.h>

#include <stdio.h>

void maximo(int datos[],int cantidad)

{

int i,mayor=0;

system("clear");

for (i=0;i<cantidad;i++)
```

```
{
if (datos[i]>=mayor)
mayor=datos[i];
}
printf("el mayor es %i \n",mayor);
fflush(stdin);
getchar();
system("clear");
}
void leer(int numeros [],int *cantidad)
{
int i;
char op='n';
system("clear");
while(op!='s')
{
printf("escribe un numero −s− para salir \n");
scanf("%i",&numeros[*cantidad]);
(*cantidad)++;
printf("escribe s para salir \n");
scanf("%s",&op);
}
system("clear");
}
void menor(int numeros [],int cantidad)
{
```

```
int i;
long menor=100000;
system("clear");
for (i=0;i<cantidad;i++)
{
if(numeros[i]<=menor)
menor=numeros[i];
}
printf(" el menor es %i \n",menor);
fflush(stdin);
getchar();
system("clear");
}
void ordenar(int numeros [],int cantidad)
{
int i,c,aux;
system("clear");
for(c=0;c<cantidad;c++)
{
for (i=0;i<cantidad−1;i++)
{
if (numeros[i]>numeros[i+1])
{
aux=numeros[i+1];
numeros[i+1]=numeros[i];
numeros[i]=aux;
```

```
}
}
}
system("clear");
}
void visualizar(int numeros[],int cantidad)
{
int i;
system("clear");
for(i=0;i<cantidad;i++)
{
printf("\n el elemento %i es %i ",i,numeros[i]);
}
fflush(stdin);
getchar();
system("clear");
}
void anadir(int numeros[],int *cantidad)
{
if(*cantidad<=10)
{
system("clear");
printf(" dime el numero \n");
scanf("%i",&numeros[*cantidad]);
(*cantidad)++;
}
```

```
else
{
printf("\n\t no se pueden anadir mas numeros ");
}
}
void eliminar(int numeros[], int *cantidad)
{
int pos,i;
if (*cantidad!=0)
{
printf("dime la posicion que quieres borrar hay %i posiciones",*cantidad);
scanf("%i",&pos);
for(i=pos−1;i<*cantidad−1;i++)
{
numeros[i]=numeros[i+1];
}
(*cantidad)−−;
}
else
{
printf("\n\tno hay valores que eliminar");
}
}
void main()
{
int datos[10],cant;
```

```
char opci;
system("clear");
do
{
printf(" \n\t pulsa 1 para introducir datos \n");
printf(" \n\tpulsa 2 para ordenar los datos \n");
printf(" \n\tpulsa 3 para ver el mayor \n");
printf(" \n\tpulsa 4 para ver el menor \n");
printf(" \n\tpulsa 5 para ver los datos \n");
printf(" \n\tpulsa 6 para anadir \n");
printf(" \n\tpulsa 7 para eliminar \n");
printf(" \n\tpulsa 8 para salir \n");
printf(" \n\telige una opcion :");
scanf("%s",&opci);
switch (opci)
{
case '1':
leer(datos,&cant);
break;
case '2':
ordenar(datos,cant);
break;
case '3':
maximo(datos,cant);
break;
case '4':
```

```
menor(datos,cant);
break;
case '5':
visualizar(datos,cant);
break;
case '6':
anadir(datos,&cant);
break;
case '7':
eliminar(datos,&cant);
break;
case '8':
exit(0);
default:
continue;
}
}
while(opci>=1||opci<=7);
}
```

FICHEROS

```
#include <stdlib.h>
#include <stdio.h>
void maximo(int datos[],int cantidad)
{
int i,mayor=0;
system("clear");
```

```
for (i=0;i<cantidad;i++)
{
if (datos[i]>=mayor)
mayor=datos[i];
}
printf("el mayor es %i \n",mayor);
fflush(stdin);
getchar();
system("clear");
}
void leer(int numeros [],int *cantidad)
{
int i;
char op='n';
system("clear");
while(op!='s')
{
printf("escribe un numero −s− para salir \n");
scanf("%i",&numeros[*cantidad]);
(*cantidad)++;
printf("escribe s para salir \n");
scanf("%s",&op);
}
system("clear");
}
void menor(int numeros [],int cantidad)
```

```
{
int i;
long menor=100000;
system("clear");
for (i=0;i<cantidad;i++)
{
if(numeros[i]<=menor)
menor=numeros[i];
}
printf(" el menor es %i \n",menor);
fflush(stdin);
getchar();
system("clear");
}
void ordenar(int numeros [],int cantidad)
{
int i,c,aux;
system("clear");
for(c=0;c<cantidad;c++)
{
for (i=0;i<cantidad−1;i++)
{
if (numeros[i]>numeros[i+1])
{
aux=numeros[i+1];
numeros[i+1]=numeros[i];
```

```
numeros[i]=aux;
}
}
}
system("clear");
}
void visualizar(int numeros[],int cantidad)
{
int i;
system("clear");
for(i=0;i<cantidad;i++)
{
printf("\n el elemento %i es %i ",i,numeros[i]);
}
fflush(stdin);
getchar();
system("clear");
}
void anadir(int numeros[],int *cantidad)
{
if(*cantidad<=10)
{
system("clear");
printf(" dime el numero \n");
scanf("%i",&numeros[*cantidad]);
(*cantidad)++;
```

```
}
else
{
printf("\n\t no se pueden anadir mas numeros ");
}
}
void eliminar(int numeros[], int *cantidad)
{
int pos,i;
if (*cantidad!=0)
{
printf("dime la posicion que quieres borrar hay %i posiciones",*cantidad);
scanf("%i",&pos);
for(i=pos−1;i<*cantidad−1;i++)
{
numeros[i]=numeros[i+1];
}
(*cantidad)−−;
}
else
{
printf("\n\tno hay valores que eliminar");
}
}
void main()
{
```

```
int datos[10],cant;
char opci;
system("clear");
do
{
printf(" \n\t pulsa 1 para introducir datos \n");
printf(" \n\tpulsa 2 para ordenar los datos \n");
printf(" \n\tpulsa 3 para ver el mayor \n");
printf(" \n\tpulsa 4 para ver el menor \n");
printf(" \n\tpulsa 5 para ver los datos \n");
printf(" \n\tpulsa 6 para anadir \n");
printf(" \n\tpulsa 7 para eliminar \n");
printf(" \n\tpulsa 8 para salir \n");
printf(" \n\telige una opcion :");
scanf("%s",&opci);
switch (opci)
{
case '1':
leer(datos,&cant);
break;
case '2':
ordenar(datos,cant);
break;
case '3':
maximo(datos,cant);
break;
```

```
case '4':
menor(datos,cant);
break;
case '5':
visualizar(datos,cant);
break;
case '6':
anadir(datos,&cant);
break;
case '7':
eliminar(datos,&cant);
break;
case '8':
exit(0);
default:
continue;
}
}
while(opci>=1||opci<=7);
}
```

FICHEROS (2ª PARTE)

```
#include <stdio.h>
struct tdatos{
int nume;
float real;
};
```

```
void main(){

struct tdatos dato[10];

FILE *pfich;

pfich=fopen("ficher","w");

dato.nume[0]=1;

dato.real[0]=23.4;

fwrite(&dato,sizeof(dato),1,pfich)

fclose("ficher");

pfich=fopen("ficher","r");
```

PASAR ARGUMENTOS DE L DE COMANDO

```
#include <stdio.h>

int main(int c,char* args[])

{

int i;

/* mostrar los argumentos pasados atraves de la linea de comandos */

while (i<c)

{

printf("los argumentos pasados son :%s\n",args[i]);

i++;

}

}
```

25