COSAS IMPORTANTES MP

-COMO CONSEGUIR NÚMEROS ALEATORIOS: **--->** #include <time.h> / #include <stdlib.h>

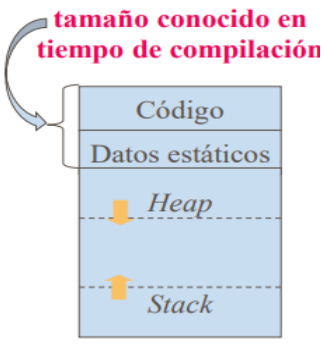
time_t t; // tipo definido en time.h
srand ((int)time(&t)); // Fija la semilla del generador:
int valor_max=10;
int valor_min=-3;

int num_rand = rand()%(valor_max+1-valor_min)+valor_min; // Este número será un numero entre 10 y -3

-ORGANIZACIÓN MEMORÍA DE UN PROGRAMA:

Al ejecutar un programa, la memoria se divide (de manera lógica, no física) en:

- **Código ejecutable**
  - Código máquina, constantes y literales (sólo lectura)
  - Se puede situar en una zona de memoria estática (permanente durante toda la ejecución)
- **Datos**
  - Variables globales y estáticas (lectura/escritura)
- La **pila o *stack*** de control que controla las ejecuciones de los procedimientos
  - Cuando se produce una llamada, se interrumpe la ejecución de una activación y se guarda en la pila
    - Variables locales y resultados intermedios
    - Información sobre el estado de la máquina
- **Montículo o *heap***
  - Guarda el resto de la información generada en tiempo de ejecución
  - Bloques de memoria direccionados por punteros

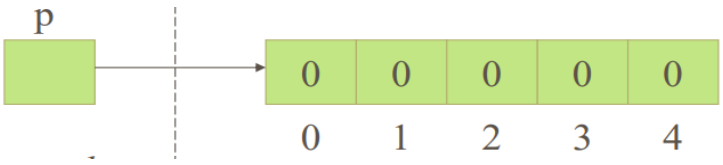


El tamaño del *stack* y el *heap* varia durante la ejecución del programa. Como la memoria reservada para ambos es fija, si para una se necesita mucho espacio, se reduce el espacio para la otra y viceversa

Cuando alojemos memoria para vectores dinámicos, lo haremos en el Heap

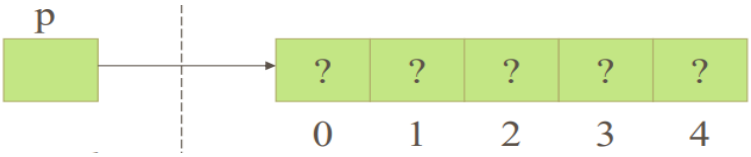
-RESERVA DE MEMORIA: **--->** #include <stdlib.h>

·**CALLOC(**int tamaño_cadena, int tamaño_dato**):**
Esta función sirve para crear un vector con un tamaño_cadena del tipo que quieras. El campo de **tamaño_dato=sizeof(tipo_dato)**. Esta función inicializará los elementos del vector con ceros.



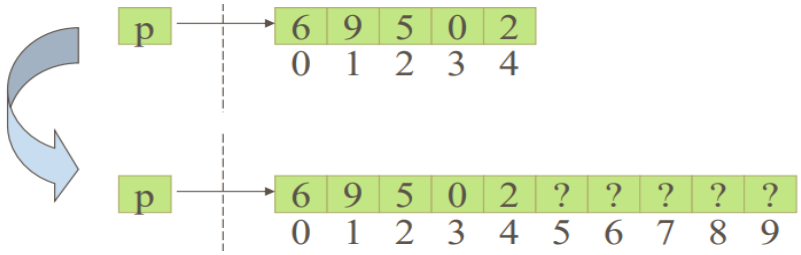
·MALLOC(int tamaño_cadena * tamaño_dato**):**

Esta función hace lo mismo que la anterior, pero no inicializará los elementos del vector.



·REALLOC(tipo_dato* vector, int tamaño_cadena * tamaño_dato**):**

Esta función cambia el tamaño de un vector dinámico. Los nuevos elementos estarán sin inicializar



·FREE(tipo_dato* vector**):**

Esta función hace lo mismo libera la memoria del vector. Borrará la información a la que el puntero apunta, pero no pondrá al puntero a NULL. IMPORTANTE: Siempre se hará al final del programa, sin excepción.

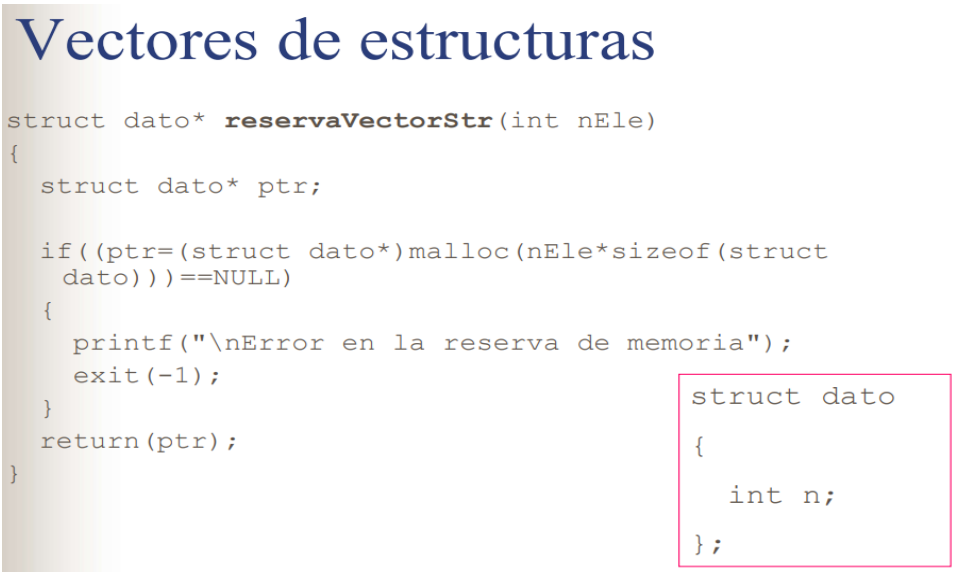
CÓDIGO:

```
int *v=NULL;
int tamaño=10; //Ejemplo
v=(int *)malloc(tamaño*sizeof(int)); //Reserva de memoria con malloc

if (v==NULL){
        printf("Ha ocurrido un error con la reserva de memoria");
        exit(-1);
}

int nuevo_tamaño=15; //Se le asignará un nuevo tamaño al vector

if((v=(int *)realloc(v, nuevo_tamaño*sizeof(int)))==NULL){ //De 10 a 15

        printf("Ha ocurrido un error con la reserva de memoria");
        exit(-1);
}

free(v);
```

IMPORTANTE: Comprobar siempre que la reserva de memoria se haya hecho bien y free(v) al final

-RECOMENDACION CREAR VECTOR EN UNA FUNCIÓN:

Yo lo que haría es crear un puntero en el main (**int *v=NULL**) y crear una función del tipo del dato que queremos el vector dinámico, que nos devuelva el vector ya creado.



También podemos pasar el puntero por referencia a una función y ahí modificarlo, pero no lo recomiendo debido a la gran cantidad de * que habrá q poner.

-RESERVA/LIBERACIÓN DE MATRICES POR FILAS:

Las matrices en C punteros que apuntan a un puntero, es decir, un vector de vectores. Cada elemento del primer vector (fila) apunta a un vector (columna)

·CÓDIGO:

```
int **m=NULL;
int tam=3;

if((m=(int**)malloc(tam*sizeof(int*)))==NULL){ //tam=nFil
        printf("Error");
        exit(-1);
}

for(int i=0;i<tam;i++){ //tam=nFil
        if((m[i]=(int*)malloc(tam*sizeof(int)))==NULL){ //tam=nCol
                printf("Error");
                exit(-1);
        }
}
```

```
for(int j=0;j<tam;j++){ //tam=nFil
        free(m[j]);
}
free(m);
```

-RESERVA/LIBERACIÓN DE MATRICES EN UN SOLO BLOQUE:

·**CÓDIGO:**

```
int **m=NULL;
int nFil=3;
int nCol=4;

if((m=(int**)malloc(nFil*sizeof(int*)))==NULL){
        printf("Error");
        exit(-1);
}

if((m[0]=(int*)malloc(nFil*nCol*sizeof(int)))==NULL){ //Se multiplíca doble
        printf("Error");
        exit(-1);
}

for(int i=1;i<nFil;i++){
        m[i]=m[i-1]+nCol;
}

free(m[0]);
free(m);
```

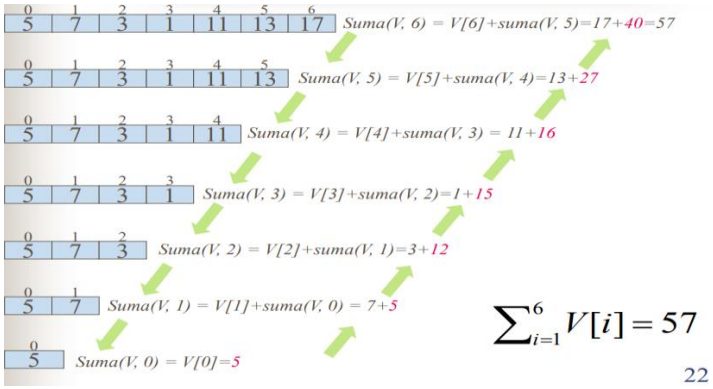
-RECURSIVIDAD:

La recursividad consiste en una función que se llama a sí misma, con el fin de obtener el resultado inicial. Estas funciones tienen un caso base, y un cuerpo.

·**CÓDIGO:**

```
int factorial (int n) {
        int resultado;
        if (n==0) {resultado = 1;} //Caso base
        else{ resultado = n*factorial(n-1);} //Cuerpo
        return(resultado);
```

}



-FICHEROS DE TEXTO: ---> #include <stdlib.h>

·fopen(char* nombre_fich, char* tipo_apertura);

Sirve para abrir el fichero. Hay que hacerlo siempre antes de modificarlo. Hay varias formas de abrirlo.

Modo	Acciones	Cursor	Si existe	Si no existe
r	Lectura	Inicio	Abre	Código error
w	Escritura	Inicio	Borra contenido	Crea
a	Adición	Final	Abre. Agrega al final	Crea
r+	L/E	Inicio	Abre. Agrega al inicio, sobreescribiendo	Código error
w+	L/E	Inicio	Borra contenido	Crea
a+	L/Adición	Final	Abre. Agrega al final	Crea

·fclose(FILE* fich);

Sirve para abrir el fichero. Hay que hacerlo siempre al acabar de tratar con él, al final del programa, ya que si no puede corromperse.

·fprintf(FILE* fich, "contenido", variables);

Sirve para escribir en el fichero. También sirve para interactuar con el usuario, como un printf (stdout) o mostrar errores (stderr).

·fscanf(FILE* fich, "contenido", variables);

Sirve para tomar datos del fichero. Tomará el número de datos y el tipo de datos especificado, escritos en "variables". Devuelve el número de elementos leídos o EOF si ha llegado al final del fichero. Útil para lectura de varios elementos, y dividirlos en distintas variables.

·fgets(char* contenido, int tam_contenido, FILE* fich);

Lee hasta tam_contenido-1 elementos, o hasta llegar a un salto de línea **\n** (si ultimo elemento es '\n', cambiarlo por '\0'). Guarda la información en una string contenido. Útil para leer una línea completa

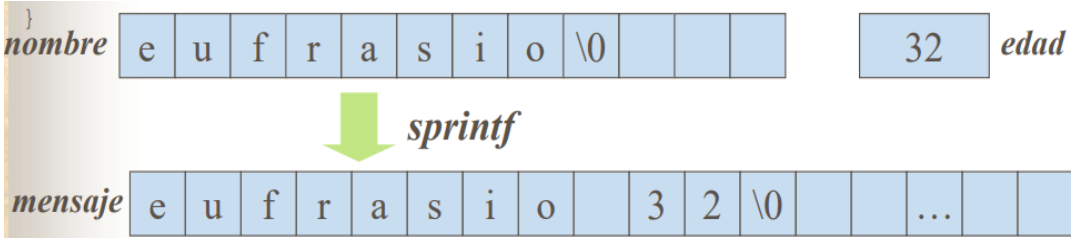
·fputs(char* contenido, int tam_contenido, FILE* fich);

Escribe una cadena (contenido) en el fichero y coloca un salto de línea (\n) al final.

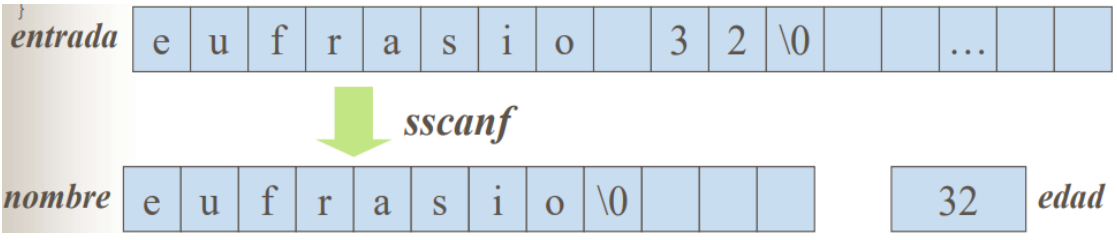
IMPORTANTE: Si cambiamos la "s" de estas funciones, podemos leer un solo carácter.

```
while(fgets(cadena, 30, fich)!=NULL)
{ if(cadena[strlen(cadena)-1]=='\n')
    cadena[strlen(cadena)-1]='\0';
  printf("\nNombre y apellidos: <%s>\n", cadena);

  fgets(cadena, 30, fich);
  sscanf(cadena, "%d", &edad); // edad=atoi(cadena);
  printf("\n\tEdad: <%d>\n", edad);

  fgets(cadena, 30, fich);
  sscanf(cadena, "%f", &saldo); // saldo=atof(cadena);
  printf("\n\tSaldo: <%f>\n", saldo);
```

·sprintf(char* mensaje_salida, "suma entradas", entrada1, entrada2, ...);
Sirve para concatenar varias entradas en una string mensaje_salida.



·sscanf(char* mensaje_entrada, "contenido", salida1, salida2, ...);
Sirve para dividir una string mensaje_entrada en varias salidas.



·remove(char* nombre_fich);

·**rename(char* nombre_viejo, char* nombre_nuevo);**

·**fflush(FILE* fich);**

Liberar el buffer asociado al fichero de salida. NUNCA usar con ficheros de entrada (stdin).

-FICHEROS BINARIOS: **--->** #include <stdlib.h>

Son un tipo de ficheros donde los datos se almacenan de forma binaria, donde cada dato ocupa sizeof(dato) bytes.

·**fopen(char* nombre_fich, char* tipo_apertura);**

Sirve para abrir el fichero. Hay que hacerlo siempre antes de modificarlo. Hay varias formas de abrirlo.

Modo	Acciones	Cursor	Si existe	Si no existe
rb	Lectura	Inicio	Abre	Código error
wb	Escritura	Inicio	Borra contenido	Crea
ab	Adición	Final	Abre. Agrega al final	Crea
r+b	L/E	Inicio	Abre. Agrega al inicio, sobreescribiendo	Código error
w+b	L/E	Inicio	Borra contenido	Crea
a+b	L/Adición	Final	Abre. Agrega al final	Crea

·**fread(char* contenido, int tam_dato, int numero, FILE* fich);**

Sirve para leer los datos del fichero. Leerá tantos datos como la variable "numero" diga. Devolverá el número de datos leídos. Si ponemos un vector, será capaz de leer todos sus elementos en una línea.

·**fwrite(char* contenido, int tam_dato, int numero, FILE* fich);**

Sirve para escribir datos en el fichero. Escribirá tantos datos como la variable "numero" diga. Devolverá el número de datos escritos. Si ponemos un vector, será capaz de escribir todos sus elementos en una línea.

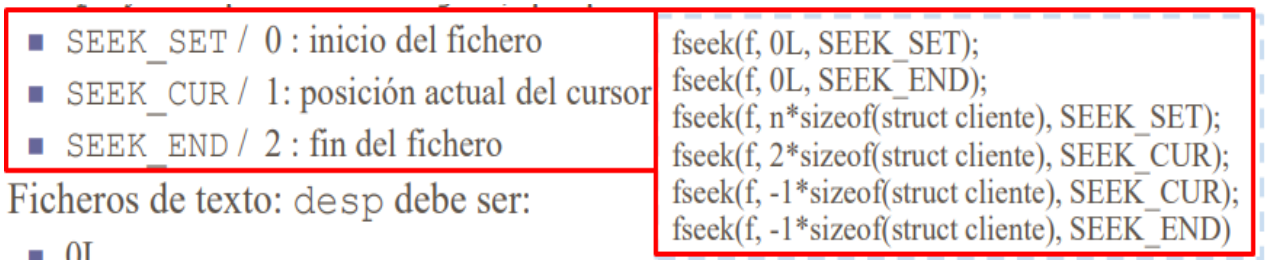
·**fteel(FILE* fich);**

Nos devolverá el nº de bytes desde el principio a la posición del cursor.

·**fseek(FILE* fich, long desplazamiento, long origen);**

Establecera la posición del cursor respeto a un origen, desplazándolo x bytes.

Muy útil, junto ftell, para calcular el nº total de elementos de un fichero.



·**CÓDIGO:(contador registros)**

```
FILE *f;
char* nombre=”archivo.bin”;
if((f=fopen(nombre, “rb”))==NULL)        {
        fprintf(stderr, “ERROR”);
        exit(-1);
}
if(fseek(f, 0L, SEEK_END)){ //Devuelve distinto de 0 si falla (true)
        fprintf(stderr, “ERROR”);
        exit(-1);
}
long numBytes=ftell(f);
long numReg=numBytes/sizeof(int) //sizeof(dato), en este caso un int
rewind(f); //Equivalente a fseek(f,0,SEEK_SET);
```

·BORRADO LÓGICO

Consiste en revisar en un archivo todos los datos hasta encontrar el que queremos borrar. En ese momento, retrocedemos el cursor, cambiamos ese valor por un espacio en blanco y seguimos escribiendo los demás

·BORRADO LÓGICO

Consiste en ir copiando la información que queremos en un archivo aparte, borrar el original, y sustituirlo por el nuevo.

-MÉTODOS DE BUSQUEDA:

Con búsqueda nos referimos al hecho de buscar un elemento entre todos los elementos de un vector. Para ello, tenemos varios métodos, más o menos eficientes, para hacerlo:

·TRATAMIENTO SELECTIVO:

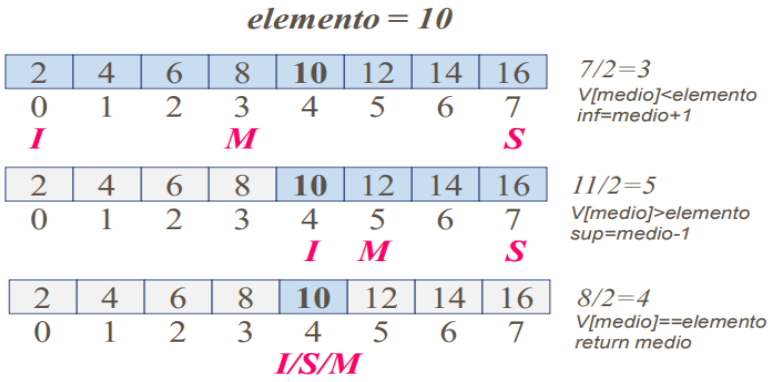
Usado para vectores donde la información puede aparecer más de una vez y requiere ser usada (ej: suma de los 2 en un vector). Debemos recorrer todo el vector.

·SECUENCIAL O LINEAL:

Usado para vectores donde la información aparecerá, como mucho, una sola vez. La búsqueda se detendrá cuando encontremos al elemento y lo devolverá. Si no es encontrado, se devolverá **–1.**

·DICOTÓMICA O BINARIA: ---> O(log n)

Necesitamos tener el vector ordenado. Lo que hacemos aquí es partir el vector, mirando si el elemento del medio es mayor o menos que el que buscamos. Así, iremos reduciendo la búsqueda hasta encontrarlo o hasta no tener más elementos.



-MÉTODOS DE ORDENACIÓN:

Con ordenación nos referimos a ordenar los elementos de un vector numérico de forma ascendente o descendente. Para ello, tenemos varios métodos, más o menos eficientes, para hacerlo:

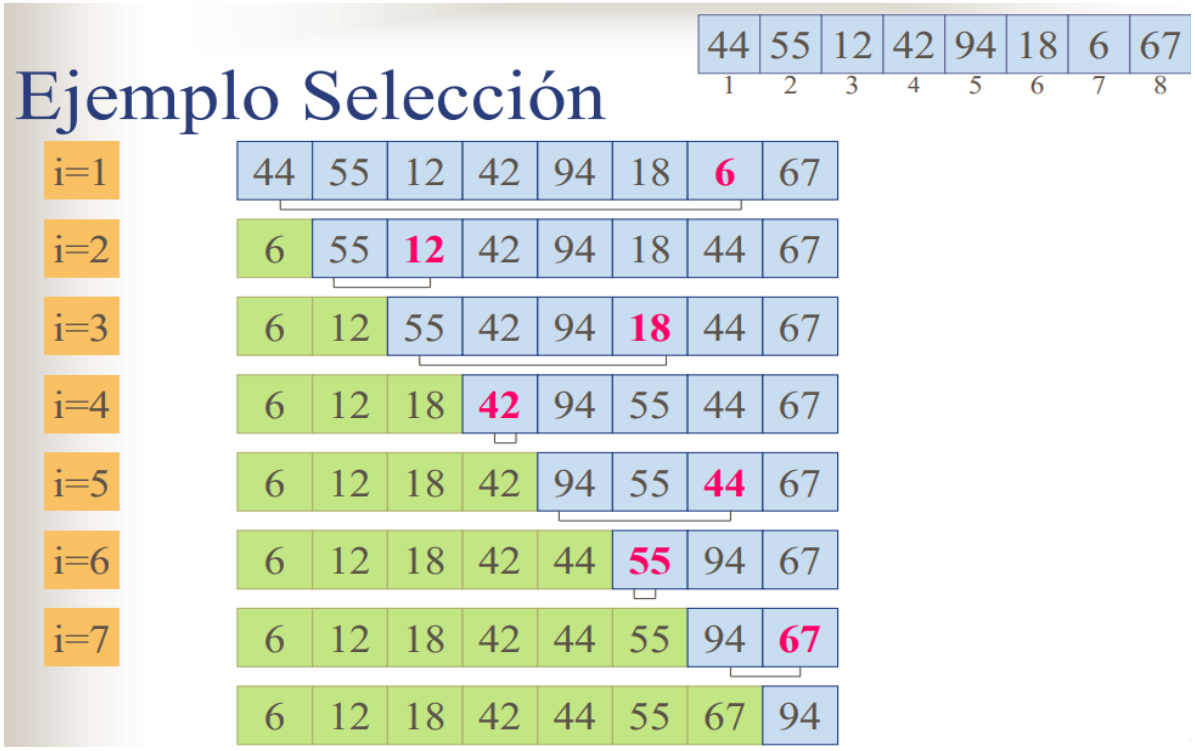
·ORDENACION POR SELECCIÓN: (Selection sort)

Consiste en coger el primer elemento del vector e ir viendo los demas elementos. Iremos guardando el índice del elemento menos a nuestro elemento inicial y, cuando lleguemos al final del vector, intercambiaremos estos elementos.

·CÓDIGO:

```
for(int i=0; i<tam-1; i++){
        int menor=i;
```

```
            for(int j=i+1; j<tam;j++){
                if(v[j]<v[menor]){
                    menor=j;
                }
            }
        aux=v[i];
        v[i]=v[menor];
        v[menor]=aux;
}
```



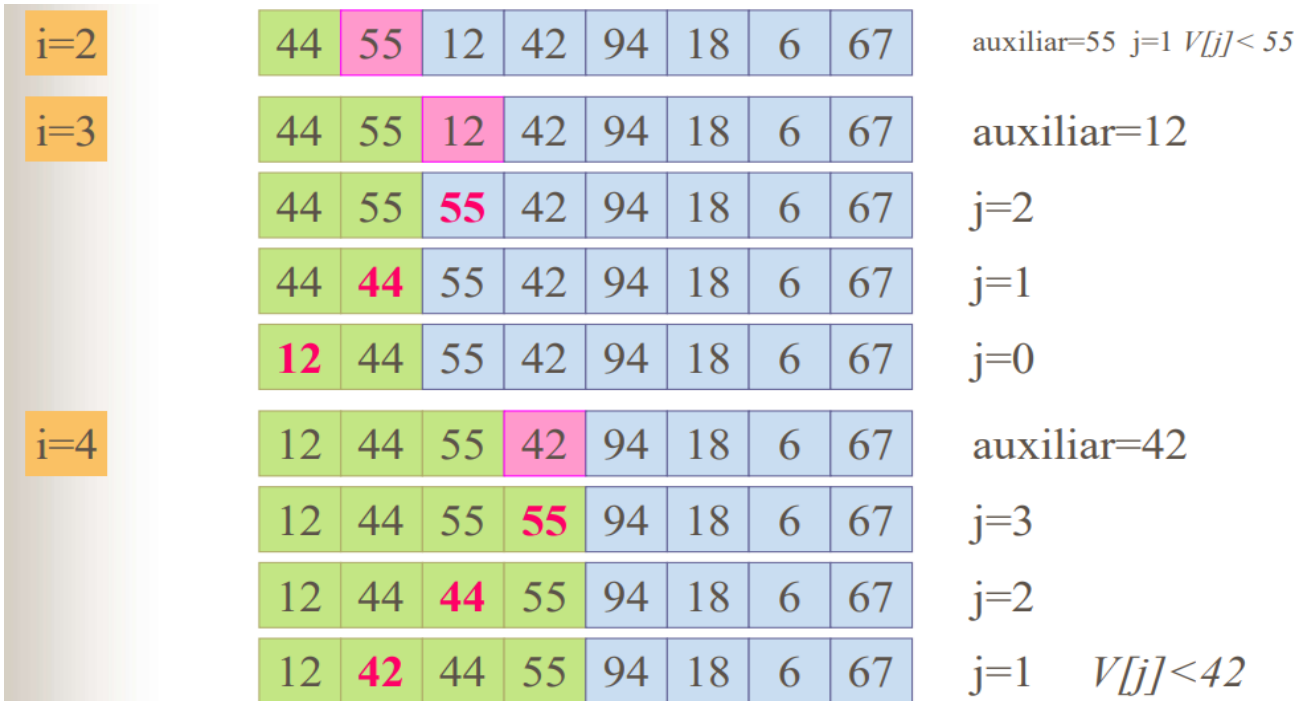
·ORDENACION POR INSERCIÓN: (Selection sort)

La idea principal de este algoritmo es ordenar los elementos del vector respecto a los demas. Cogemos un primer elemento y lo comparamos con su siguiente. Si es menor, cambian sus posiciones. Ahora, cogemos el siguiente elemento y lo comparamos con los anteriores. Así hasta acabar con el ultimo elemento.

·CÓDIGO:

```
for(int i=1; i<tam; i++){
        int aux=v[i];
        int j=i-1;
```

```
        while(j > 0 && v[j] > aux) {
                v[j + 1] = v[j]; // Desplazamiento
                j--;
        }
        v[j + 1] = aux;
}
```



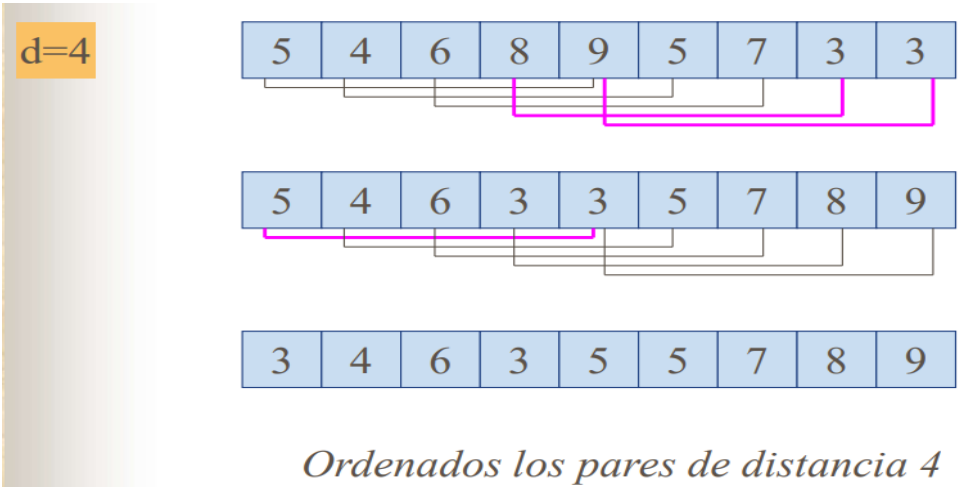
·ORDENACION POR INSERCIÓN SHELL: (Shell sort)

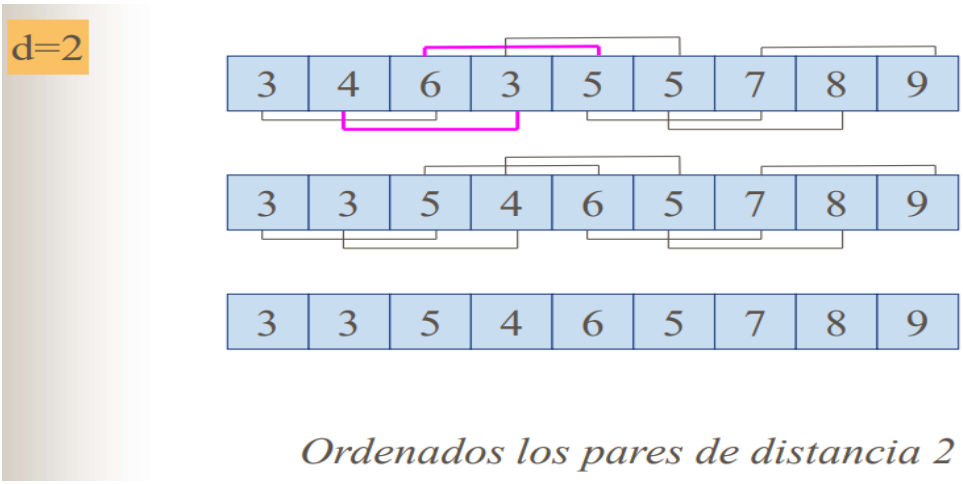
Este método trata de ir comparando cada elemento **i** con el elemento **i+n/2**, sustituyendo n, al terminar de comparar, por el resultado de n/2. Asi, hasta llegar a n=1. Este proceso se hara de izquierda a derecha, para asi garantizar q los menores quedaran en la izquierda.

·CÓDIGO:

```
for (int h = tam / 2; h > 0; h /= 2) {
        for (int i = h; i < tam; i++) {
```

```
                int actual = v[i];
                int j = i;
                while (j >= h && v[j - h] > actual) {
                        v[j] = v[j - h];
                         j -= h;
                }
                v[j] = actual;
        }
}
```





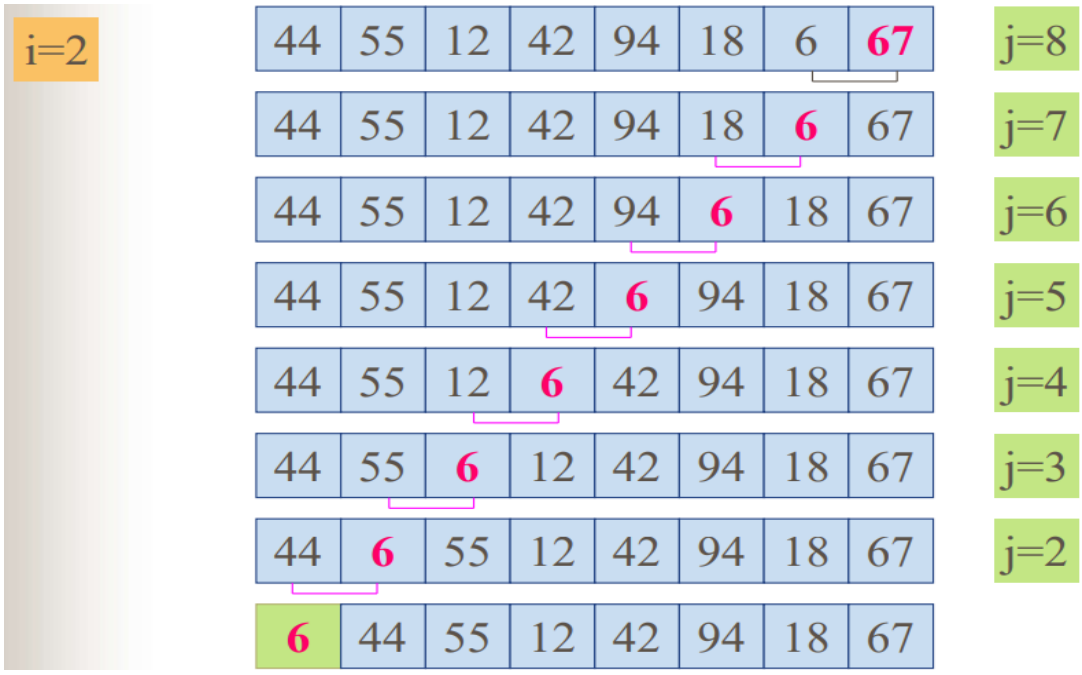
·ORDENACION POR BURBUJA: (Bubble sort)

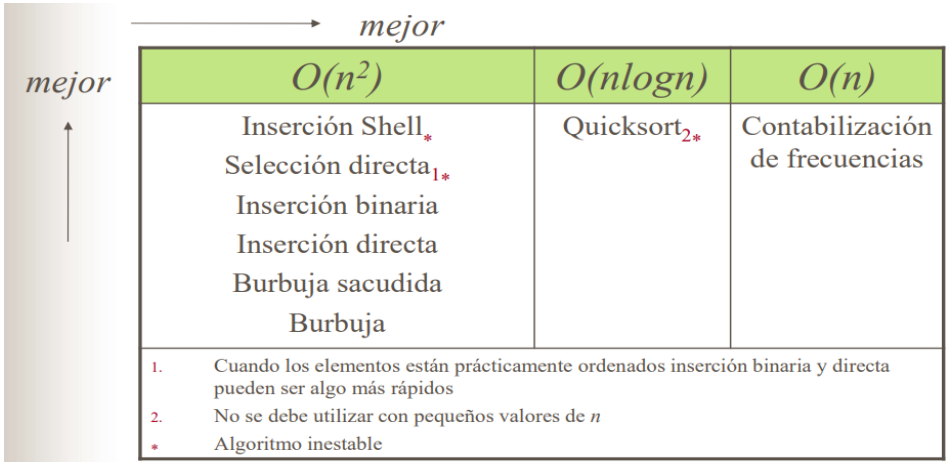
La base de este método es la comparación e intercambio de pares de elementos adyacentes, hasta que todos los elementos del vector están ordenados. Se empieza por la derecha y se va comparando con su adyacente a la izquierda.

·CÓDIGO:

```
for (int i=2, i<n, i++){
        for (int j=n-1; j>i; j--){
```

```
            if (v[j]<v[j-1]){
                aux=v[j-1];
                v[j-1]=v[j];
                v[j]=aux;
            }
        }
}
```



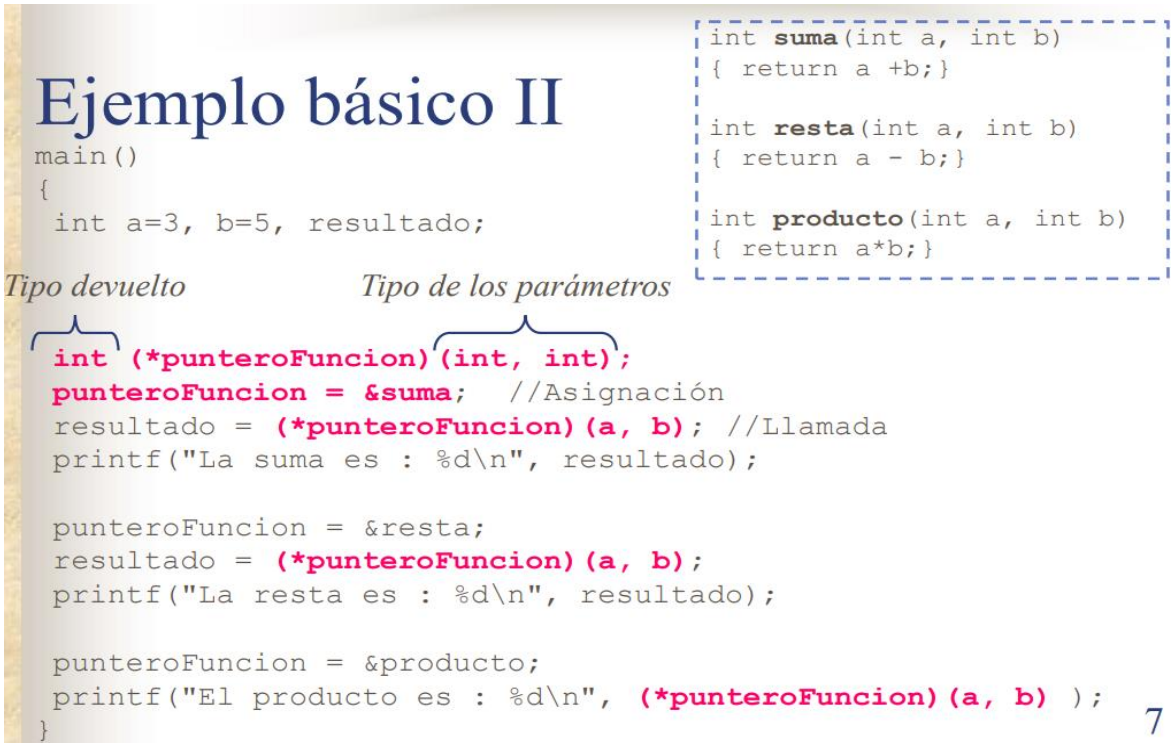


-PUNTEROS A FUNCIONES:

Esto es, como su nombre indica, un puntero que apunta a otra función. Esto nos sirve para poder pasar funciones como parámetros y generalizarlas.

-ESTRUCTURA:

Tipo _retorno (*nombre_puntero) (tipo_parametro, ...)



Utilización como argumentos de una función

- Ejemplo: Función general que calcule la suma *f(1)+f(2)+ ... + f(n)* para cualquier función *double f(int)*

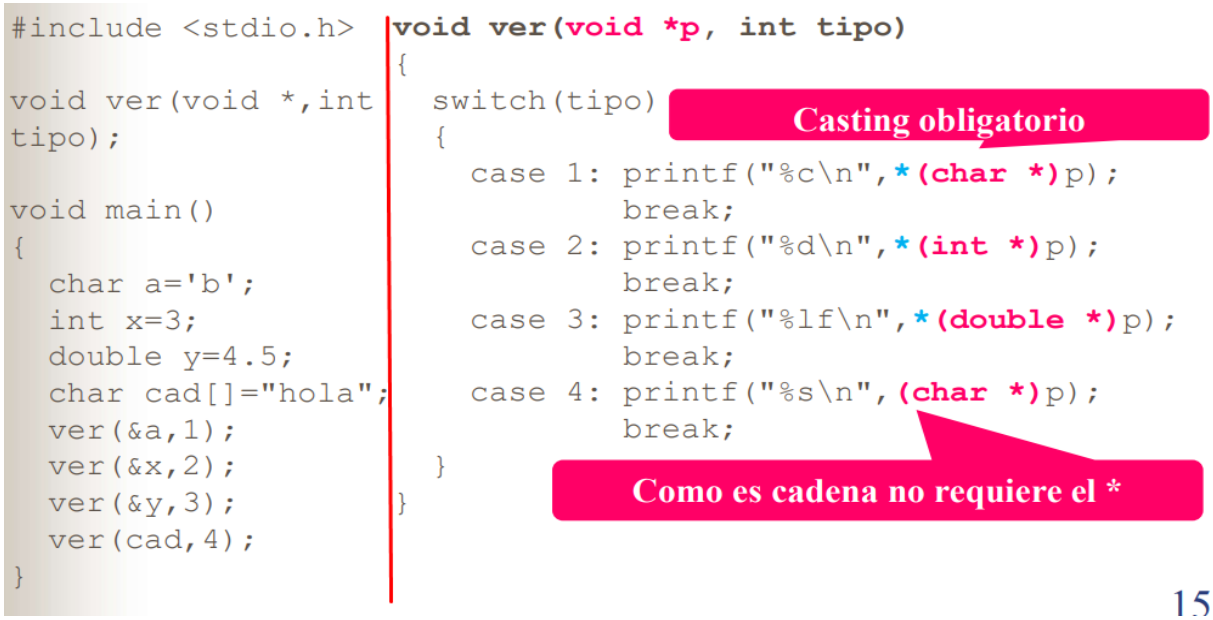
```
double funcSuma(int n, double (*f)(int));

void main()
{
   double (*pf)(int);

   //A) pasamos el puntero
   pf=&inverso;
   printf("Suma 2 inversos:%lf\n",  funcSuma(2,pf));
   pf=&cuadrado;
   printf("Suma 3 cuadrados:%lf\n", funcSuma(3,pf));

   //B) pasamos la direccion de la funcion
   printf("Suma 4 inversos:%lf\n",  funcSuma(4, &inverso));
   printf("Suma 5 cuadrados:%lf\n", funcSuma(5, &cuadrado));
}
```

```
double funcSuma(int n, double (*f
{
   double s=0;
   int i;
   for (i=1;i<=n;i++)
      s+=(*f)(i);
   return(s);
}
double inverso (int k)
{
   return 1.0/k;
}
double cuadrado (int k)
{
   return (double)k*k;
}
```

En este caso, en vez de tener x funciones de suma distinta, hace una función de suma general y va cambiando los números que suma, pasando por parámetros una función que invierte o eleva los números al cuadrado.

-PUNTEROS VOID A FUNCIONES:

Estos punteros son útiles a la hora de recibir datos cuando no sabemos el tipo de este mismo. Para ello, es obligatorio usar "casting", que se refiere a, con un switch, tomar las posibles entradas de datos válidas y realizar una función según el tipo.



-QSORT: (en stdlib.h)

Es una función incluida en la librería "stdlib.h" que nos permite realizar el algoritmo de ordenación **quicksort**. Esta es su estructura:

qsort(vector, tam, sizeof(tipo_dato), &FuncComparaEnteros);

La función de comparar enteros será:

```
int comparaEnteros(const void* e1 , const void* e2) {
        *int* a, *b;
        a = (int*)e1;
        b = (int*)e2;
         if(*a<*b)
                return (-1);
        else if(*a==*b)
                 return(0);
        else
                return(1);
}
```

-MAKEFILE:

```
inicio: saluda crearejecutable
saluda:
@echo "Ejecutando makefile de
crearejecutable: main.o funciones.o
gcc main.o funciones.o -o eje
main.o: main.c funciones.h
gcc -c main.c
funciones.o: funciones.c funciones.h
gcc -c funciones.c
```

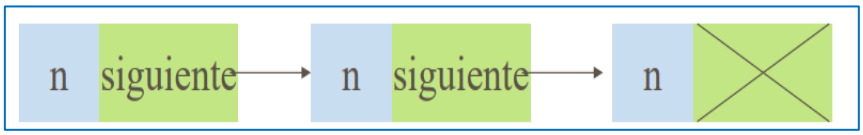
20:23

```
.PHONY: clean
clean:
@echo Borrando ficheros.o ...
@rm *.o
```

-LISTAS:

Son un tipo de estructura que tiene como elementos un dato, y un puntero a estructura llamado siguiente, que lo que hace es apuntar a otro elemento. De la forma:

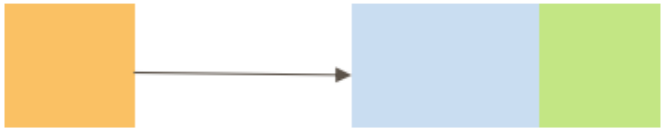
```
struct lista{
        int n;
        struct lista*
sig;
};
```



Se diferencian con los vectores en que no tienen índice, lo que quiere decir que no podemos acceder fácilmente al elemento i-enésimo, y en que tienen un borrado mucho menos costoso, ya que no hace falta reescribir toda la lista para borrarlo.

-FUNCIONES:

```
struct lista *nuevoElemento(){
        return ((struct lista *) malloc(sizeof(struct lista)));
}
```

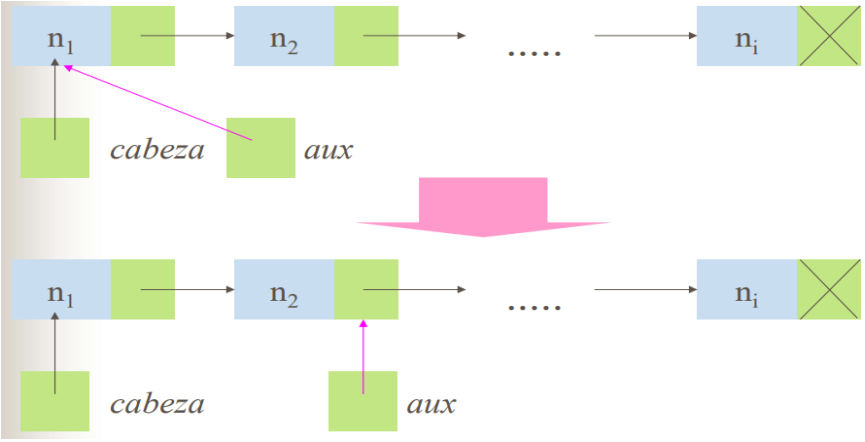


```
void insertarDelante(struct lista** cabeza, int n){
        struct lista *nuevo=NULL;

        nuevo=nuevoElemento();
        elemento->n=n;

        elemento->sig=*cabeza;
        *cabeza= nuevo;
}
```

Esto lo que hace es crear un nuevo elemento, hacer que apunte al primer elemento, al que apunta *cabeza, y apuntar cabeza al nuevo elemento.

```
void imprimirLista(struct lista* cabeza){
        struct lista *aux=NULL:

        aux=cabeza;
        while(aux!=NULL){
                printf("%d", aux->n);
                aux=aux->sig;
        }
}
```

Lo que hace es hacer una lista auxiliar que es igual a cabeza. Esto se hace para no mover el puntero de la lista cabeza. Ahora, con la aux, recorremos toda la lista imprimiendo los datos.

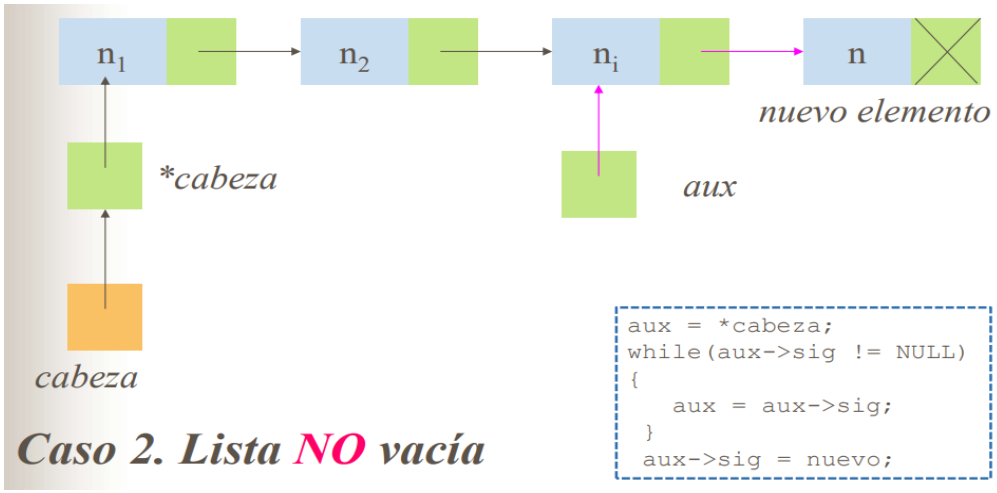



```
void imprimirListaInverso(struct lista *elemento) {
        if (elemento != NULL) {
                imprimirListaInverso(elemento->sig);
                printf(" %d \n", elemento->n);
```

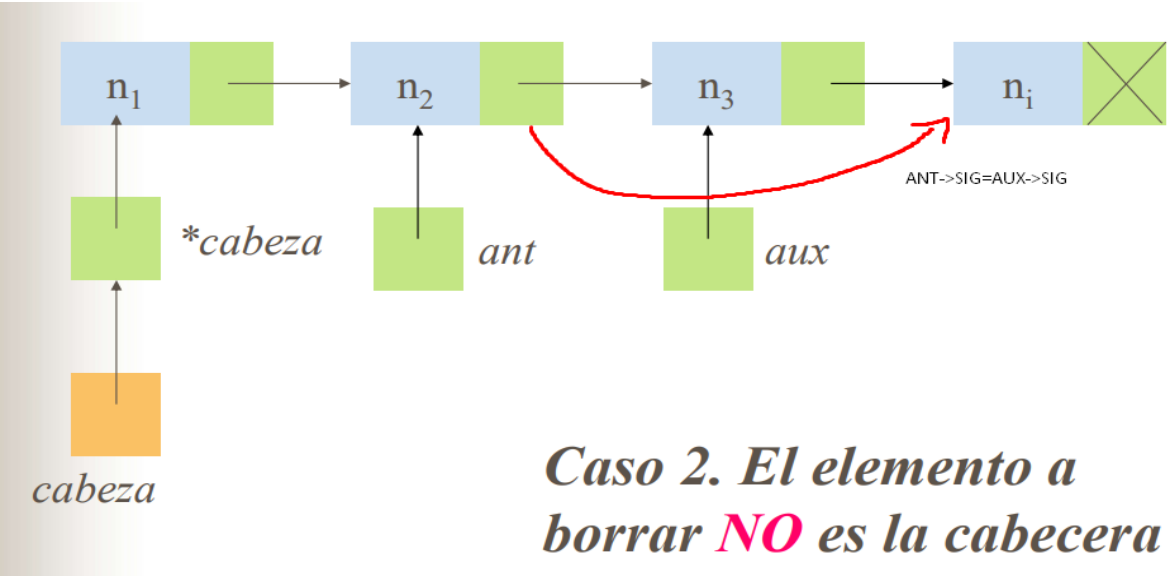
```
        }
    }
```

Aquí, para mostrarlo de forma inversa, usa recursividad. Misma forma.

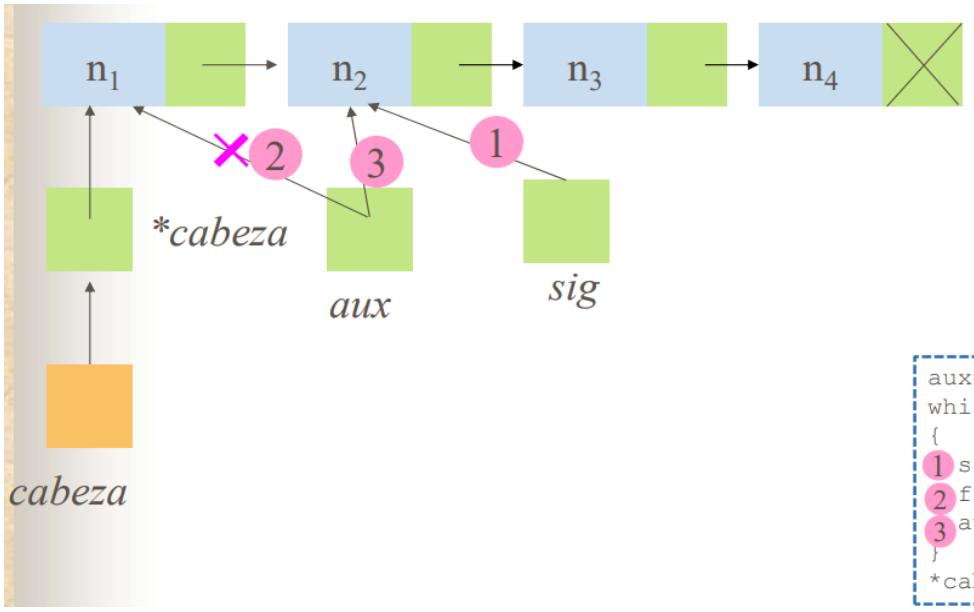
```
int buscarElemento(struct lista *cabeza, int n) {
        int encontrado = 0;
        struct lista *aux = NULL;
        aux = cabeza;

        while (aux!=NULL && encontrado==0){
            if (aux->n==n){
                encontrado=1;
            }else{
                aux=aux->sig;
            }
        }
    }


void insertarDetras(struct lista **cabeza, int n){
        struct lista* aux=NULL;
        struct lista* nuevo=NULL;
        nuevo=nuevoElemento();
        nuevo->n=n;
        nuevo->sig=NULL;                //Hacerlo a mano

        if (*cabeza==NULL){             //Lista nueva
            *cabeza=nuevo;
        }else{                          //Lista con elementos
            aux=cabeza*
            while(aux->sig!=NULL){
                aux=aux->sig;
            }
            aux->sig=nuevo;
        }
    }
```




```
void borrarElemento(struct lista **cabeza, int n){
        struct lista *ant = NULL;          /*anterior al que se borra*/
        struct lista *aux = NULL;          /* elemento a borrar */

        aux=*cabeza;
        while(aux->n!=n){                  // Buscar elemento
                ant=aux;
                aux=aux->sig;
        }

        if (aux=*cabeza){
                *cabeza=aux->sig;
                free(aux);
        }
        else{
                ant->sig=aux->sig;
                free(aux);
        }
```




```
void borrarLista(struct lista **cabeza){
        struct lista * aux, * sig;
        aux=*cabeza;

        while(aux!=NULL) {
                sig=aux->sig;
                free(aux);
                aux=sig;
        }
        *cabeza=NULL;                   //Lista vacia
}
```

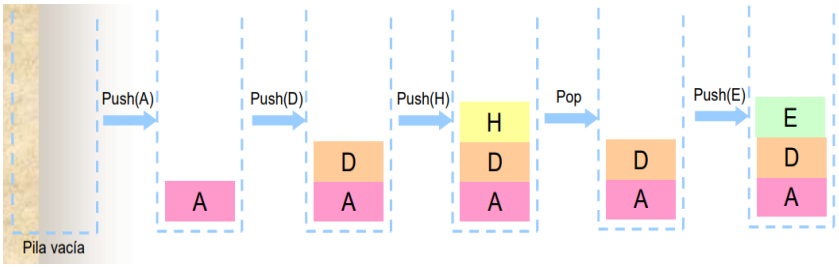



```
aux=*cabeza;
while(aux!=NULL)
{
1 sig=aux->sig;
2 free(aux);
3 aux=sig;
}
*cabeza=NULL;
```

-PILAS:

Esto es otro tipo de estructura que funciona como una lista LIFO (Last Input First Output). Esto quiere decir que el ultimo que entra, es el primero en mostrarse. Esto, hablando de algo cotidiano, es como una pila de libros. Su estructura es:

```
struct pila{
        char
nombre[20];
        struct pila* sig;

};
```



·FUNCIONES:

```
struct pila * nuevoElemento();
int vacia(struct pila *cabeza);
void verCima(struct pila *cabeza, char *nombre);
void apilar(struct pila **cabeza, char *nombre);
void desapilar(struct pila **cabeza, char *nombre);
```

-COLAS:

Una cola es otro tipo especial de lista en la cual los elementos se insertan por el extremo anterior (por el principio) y se suprimen por el posterior (por el final). Son listas FIFO (First Input First Output)

Colas

```
struct punto{
   float x;
   float y;
};
struct cola{
   struct punto p;
  struct cola *sig;
};
```

- `struct cola *nuevoElemento();`
- `void insertarCola(struct cola **cabeza, struct punto p);`
- `struct punto extraerCola(struct cola **cabeza);`
- `int contiene(struct cola *cabeza);`