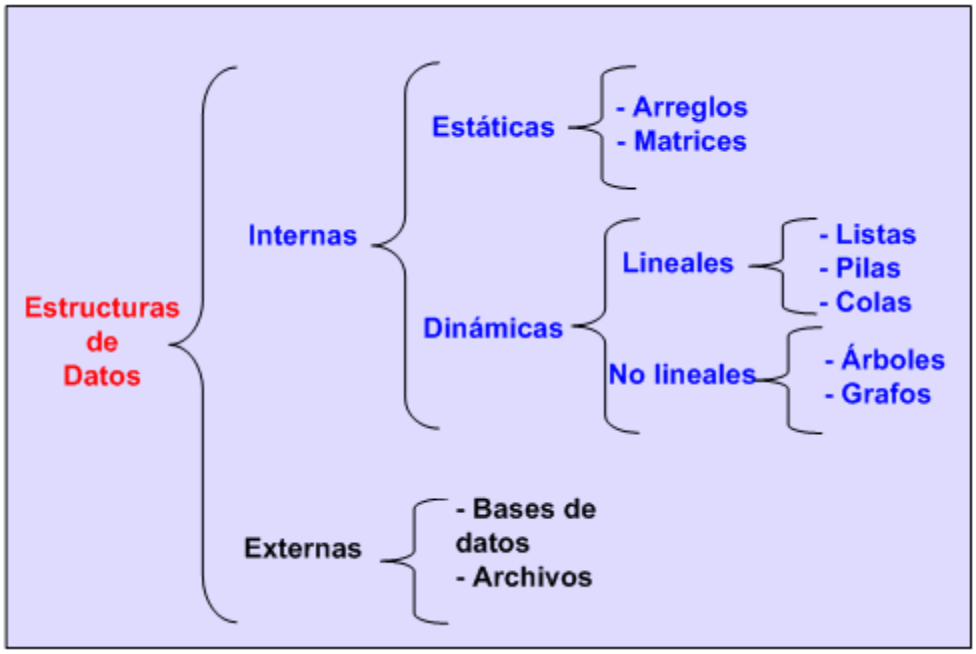
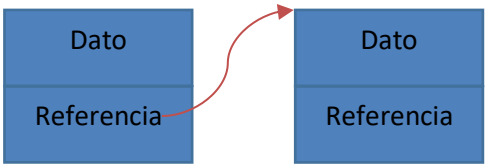
Estructuras de datos dinámicas en C – listas ligadas

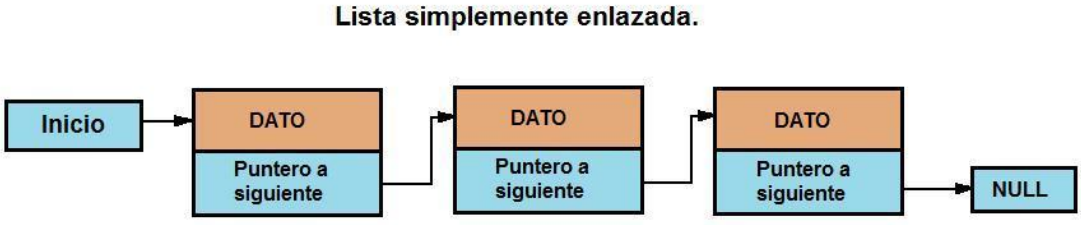


Son colecciones de datos homogéneas en las que cada elemento o nodo contiene una referencia a la posición en memoria del resto. De esta manera cada uno puede estar ubicado en posiciones de memoria no necesariamente consecutivas, sino que se puede reservar y liberar memoria de manera dinámica si se necesita agregar o sacar nuevos elementos.

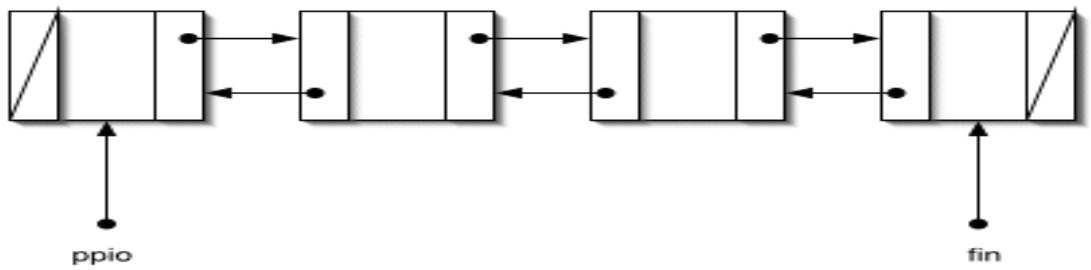


Según el tipo de referencia o ligadura se armarán estructuras lógicas de distintos tipos:

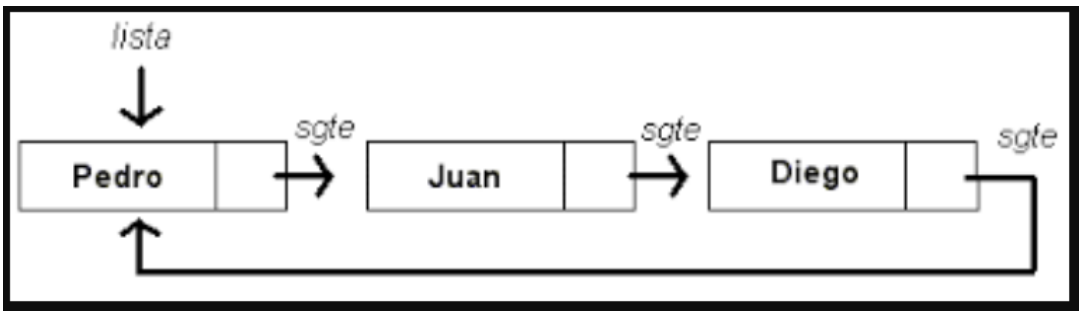
Lineales simplemente enlazadas:



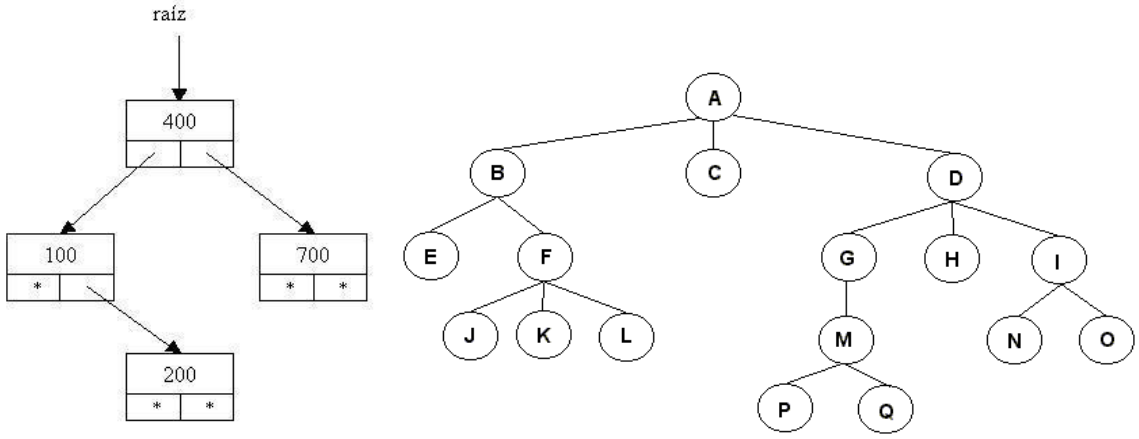
Lineales doblemente enlazadas:



Circulares:



Arboles:



Nodos: estructuras autoreferenciadas

Para implementar los nodos se pueden utilizar estructuras autoreferenciadas, es decir, estructuras que contienen al menos dos elementos: la información o dato a almacenar y un puntero a una estructura de su mismo tipo. Ejemplo:

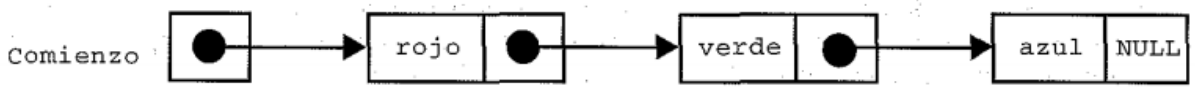
```
struct Nodo {
int Dato;
struct Nodo * ptr;};
```

El enlace o ligadura entre los nodos se realiza asignando al puntero de un nodo la dirección de memoria en donde esté ubicado otro nodo. En caso de que el nodo no deba hacer referencia a otro se puede asignar ***NULL***, por ejemplo para indicar que es el último elemento de una lista.

Como la intención es disponer una estructura para almacenar datos de manera dinámica, es decir, que se puedan agregar o eliminar elementos a demanda, se van a utilizar funciones de memoria dinámica para crear o eliminar nodos.

Ejemplo de creación de una lista enlazada simple:

EJEMPLO 11.30. La Figura 11.3(*a*) muestra una lista enlazada con tres componentes. Cada componente consta de dos elementos: una cadena de caracteres y un puntero que referencia el siguiente componente dentro de la lista. Así, el primer componente contiene la cadena `rojo`, el segundo contiene `verde` y el tercero `azul`. El principio de la lista es indicado por un puntero separado, etiquetado como `comienzo`. También el final de la lista está marcado por un puntero especial llamado `NULL`.



a)

Definición de estructura nodo:

```
//definicion de nodo:
struct nodo
{
    char color[10];
    struct nodo *ptr;
};
```

Función para crear un nodo nuevo:

```
struct nodo * CrearNodo(char * dato)
{
    struct nodo * pnodo = (struct nodo *) malloc(sizeof(struct nodo));
    if(pnodo != NULL)
    {
        strcpy(pnodo->color,dato);

        pnodo->ptr = NULL;
    }
    return pnodo;
}
```

Función para recorrer la lista completa y mostrar en pantalla el contenido de cada nodo:

```
void MostrarLista(struct nodo * pnodo)
{
    //struct nodo * pnodo = pnodo;
    if(pnodo != NULL)
    {
        do
        {
            printf("%s, ",pnodo->color);
            pnodo = pnodo->ptr;
        }
        while(pnodo != NULL);
        printf("\n");
    }
}
```

Programa que crea los 3 nodos indicados en la figura, luego inserta el nodo Blanco y finalmente elimina el nodo Verde:

```
//necesito almacenar la direccion del primer elemento:
struct nodo * inicio;

//tambien necesito un puntero para almacenar la direccion de cada nuevo
elemento y manipularlo:
struct nodo * pnodo;

//creo primer nodo:
pnodo = CrearNodo("Rojo");

//guardo esta primer direccion como inicio de la lista:
inicio = pnodo;

//creo segundo nodo y debo enlazar este nuevo nodo al primero
pnodo->ptr = CrearNodo("Verde");
//como el primer nodo ya está enlazado al segundo, actualizo el puntero pnodo:
pnodo = pnodo->ptr;

//creo tercer nodo y debo enlazar este nuevo nodo al segundo
pnodo->ptr = CrearNodo("Azul");
//como el primer nodo ya está enlazado al segundo, actualizo el puntero pnodo:
pnodo = pnodo->ptr;
//entonces cada vez que agrego un nodo nuevo pnodo apunta al ultimo y este
contiene referencia a NULL

MostrarLista(inicio);
```

Añadamos ahora otro componente, cuyo valor es `blanco`, entre `rojo` y `verde`. Para hacer esto simplemente cambiamos los punteros, como se ilustra en le Figura 11.3(*b*). Análogamente, si elegimos borrar el elemento cuyo valor es `verde`, sencillamente cambiamos el puntero asociado con el segundo componente, como se muestra en la Figura 11.3(*c*).

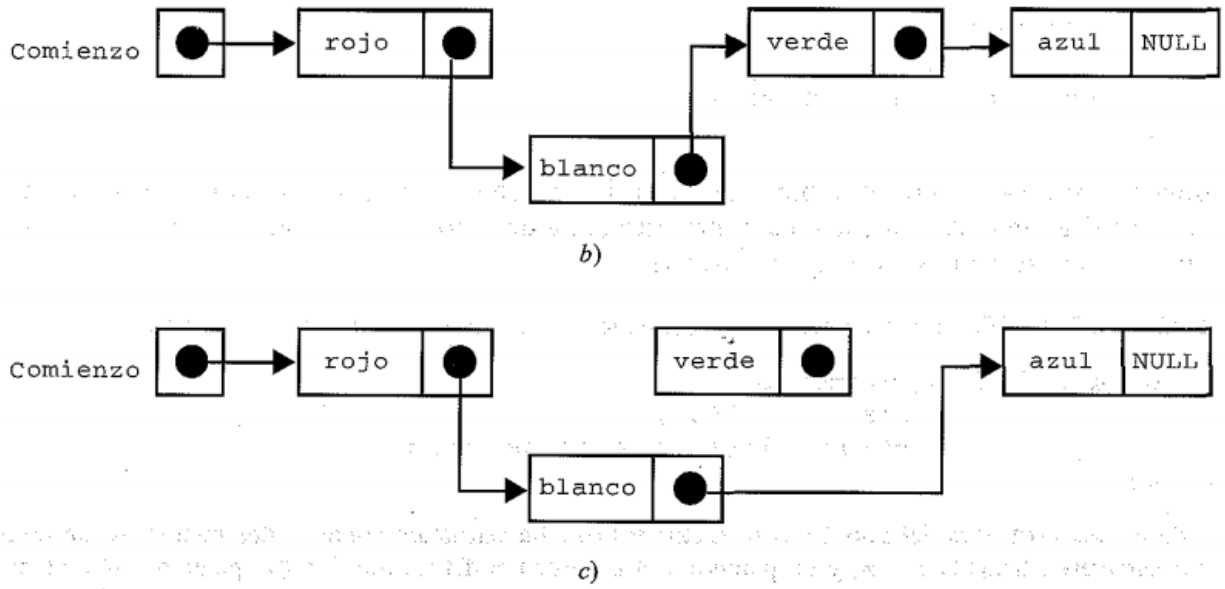


Figura 11.3.

```
//insertar nodo Blanco despues del rojo
struct nodo * nuevo = CrearNodo("Blanco");
//busco nodo Rojo, parto desde inicio
pnodo = inicio;
do{
    if(strcmp( pnodo->color, "Rojo" ) == 0 )
    {
        //encontre el nodo rojo
        nuevo->ptr = pnodo->ptr;//enlazo blanco a verde
        //enlazar rojo a blanco
        pnodo->ptr = nuevo;
    }
    pnodo = pnodo->ptr;
}while(pnodo !=NULL);

MostrarLista(inicio);
```

```c
//eliminar el nodo verde de la lista
//buscar nodo verde
pnodo = inicio;
struct nodo * panterior = inicio;
do{
    if(strcmp( pnodo->color, "Verde" ) == 0 )
    {
        //pnodo es el nodo verde
        //panterior hace referencia a verde
        //unir panterior a pnodo->ptr
        panterior->ptr = pnodo->ptr;
        //eliminar verde liberando la memoria
        strcpy(pnodo->color, "XXXXXXXXXX");
        free(pnodo);
    }
    panterior = pnodo;
    pnodo = pnodo->ptr;
}while(pnodo !=NULL);

MostrarLista(inicio);
```

Ejemplo de lista enlazada doble:

EJEMPLO 11.31. En la Figura 11.4 vemos una lista lineal enlazada similar a la mostrada en la Figura 11.3(*a*). Sin embargo, ahora hay *dos* punteros asociados con cada componente: un puntero al siguiente y un puntero al anterior. Este doble conjunto de punteros permite recorrer la lista en cualquier dirección, de principio a fin, o del fin al principio.

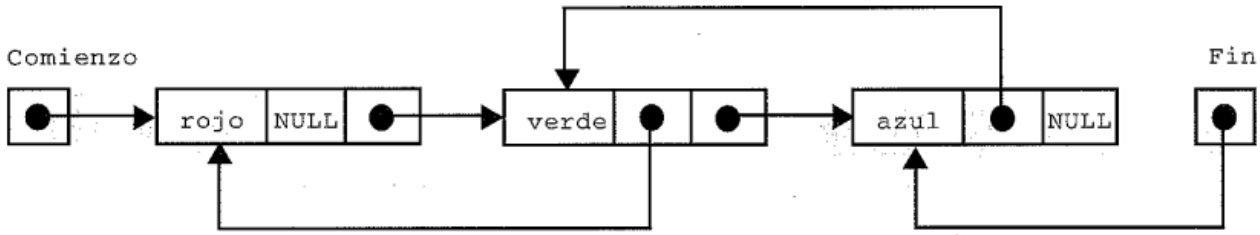


Figura 11.4.

```c
//definicion de nodo:
struct nodo
{
    char color[10];
    struct nodo *panterior;
    struct nodo *psiguiente;
};
```

Crear nodo:

```c
struct nodo * CrearNodo(char * dato)
{
    struct nodo * pnodo = (struct nodo *) malloc(sizeof(struct nodo));
    if(pnodo != NULL)
    {
        strcpy(pnodo->color,dato);

        pnodo->psiguiente = NULL;
        pnodo->panterior = NULL;
    }
    return pnodo;
}
```

Crear la misma lista del ejemplo anterior, ahora doblemente enlazada:

```
//creo primer nodo:
pnodo = CrearNodo("Rojo");

//guardo esta primer direccion como inicio de la lista:
inicio = pnodo;

//creo segundo nodo y debo enlazar este nuevo nodo al primero
pnodo->psiguiente = CrearNodo("Verde");
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
(pnodo->psiguiente)->panterior = pnodo;//enlace de verde a rojo

pnodo = pnodo->psiguiente;//puntero actual a verde

//creo tercer nodo y debo enlazar este nuevo nodo al segundo
pnodo->psiguiente = CrearNodo("Azul");
(pnodo->psiguiente)->panterior = pnodo;//enlace de azul a verde
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
pnodo = pnodo->psiguiente;
//entonces cada vez que agrego un nodo nuevo pnodo apunta al
ultimo y este contiene referencia a NULL

MostrarLista(inicio);
```

Ejemplo de lista circular:

Consideremos ahora la lista mostrada en la Figura 11.5. Esta lista es similar a la mostrada en la Figura 11.3(*a*), excepto que el último elemento (`azul`) apunta al primer elemento (`rojo`). Por tanto esta lista no tiene principio ni fin. Tales listas son denominadas *listas circulares*.

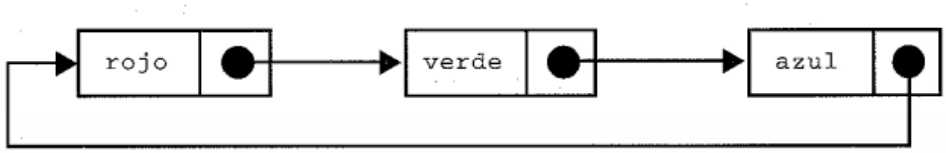


Figura 11.5.

En este caso el código es idéntico al primer ejemplo, solo que al crear el ultimo nodo lo enlazo al primero:

```
//creo primer nodo:
pnodo = CrearNodo("Rojo");

//guardo esta primer direccion como inicio de la lista:
inicio = pnodo;

//creo segundo nodo y debo enlazar este nuevo nodo al primero
pnodo->ptr = CrearNodo("Verde");
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
pnodo = pnodo->ptr;

//creo tercer nodo y debo enlazar este nuevo nodo al segundo
pnodo->ptr = CrearNodo("Azul");
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
pnodo = pnodo->ptr;
pnodo->ptr = inicio;
```

Si intento utilizar la misma función para mostrar la lista obtengo la siguiente salida:

```
valentin@DESKTOP-US8MTTG:/mnt/e/Codes/2020/INFO2/24-6$ gcc listas1c.c
valentin@DESKTOP-US8MTTG:/mnt/e/Codes/2020/INFO2/24-6$ ./a.out
Hola mundo!
Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azu
l, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde,
Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verd
e, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, V
erde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo, Verde, Azul, Rojo
```

Es decir, se va a leer de manera infinita el contenido de la lista.

Ejemplo de árbol:

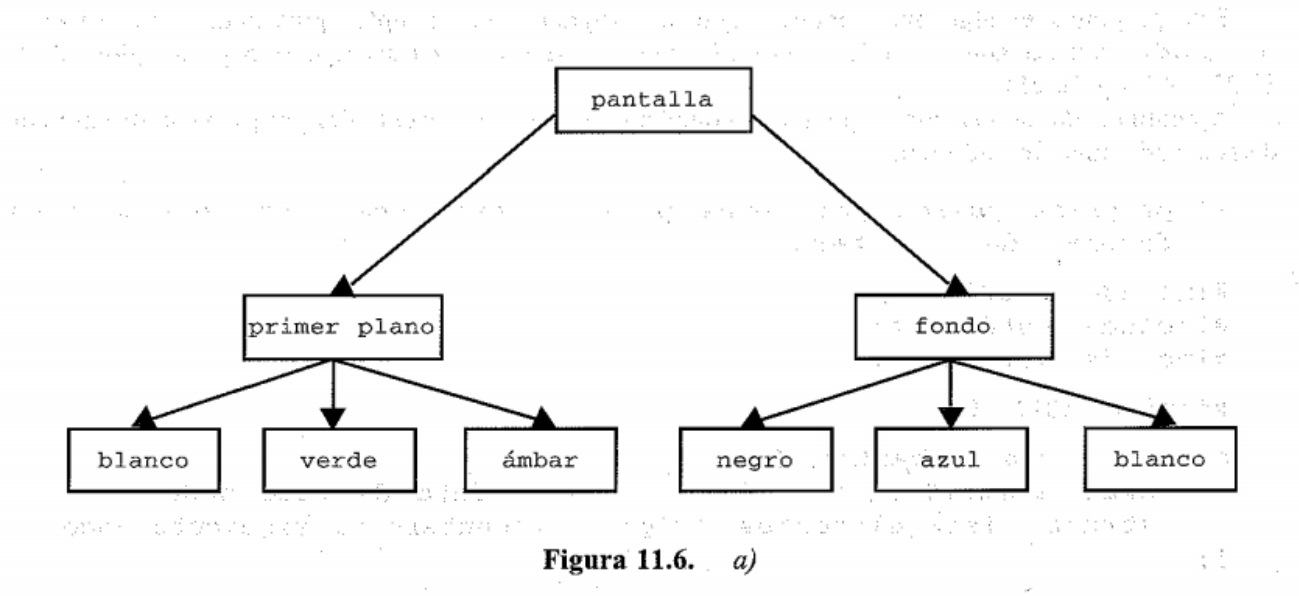


Figura 11.6. *a)*

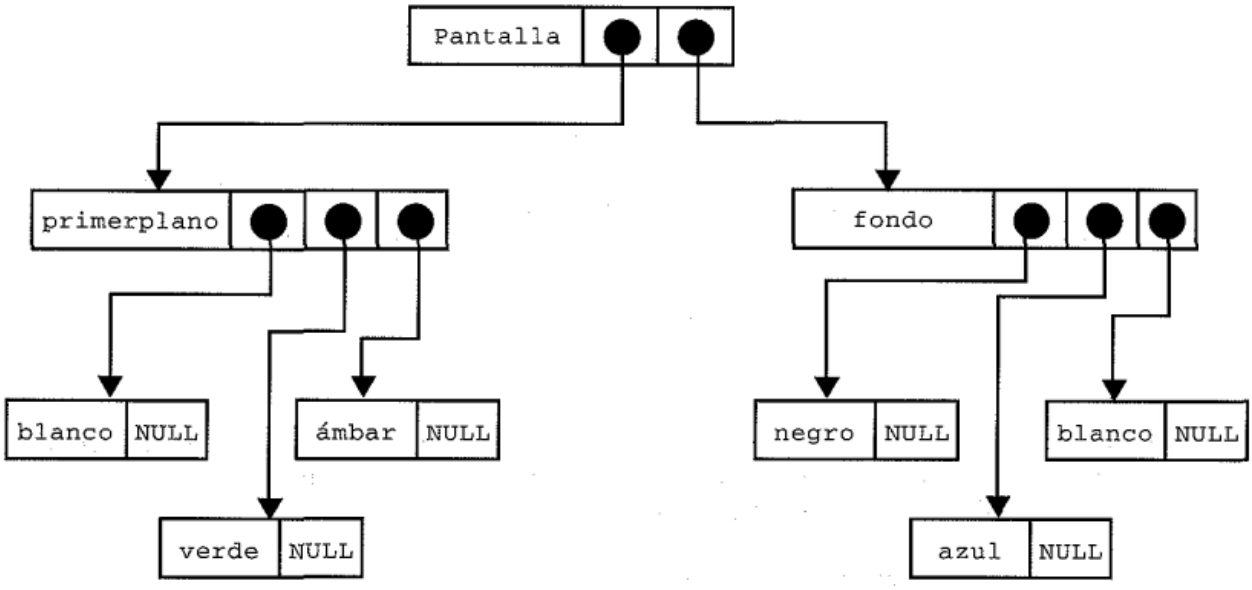


Figura 11.6. *b)*

Ejemplo de una estructura nodo para un árbol:

```c
//definicion de nodo:
struct nodo
{
    char color[10];
    struct nodo *padre;
    struct nodo *hijo_1;
    struct nodo *hijo_2;
    struct nodo *hijo_3;
};
```

Otra forma un poco más versátil de definir las estructuras:

```
//definicion de nodo:

typedef struct{int a; float b;} Dato;

typedef struct pnodo
{
    Dato Info;
    struct pnodo *ptr;
}nodo;
```

Con esta definición si deseo cambiar el tipo de dato que almacena la lista solo debo modificar la primer línea y las funciones para crear, insertar y eliminar un nodo quedarían igual.

Ahora la función Crear nodo quedaría:

```
nodo * CrearNodo(Dato d)
{
    nodo * pnodo = (nodo *) malloc(sizeof(nodo));

    if(pnodo != NULL)
    {
        pnodo->Info = d;
        pnodo->ptr = NULL;
    }

    return pnodo;
}
```

Y la función mostrar lista:

```
void MostrarLista(nodo * pnodo)
{
    //struct nodo * pnodo = pnodo;
    if(pnodo != NULL)
    {
        do
        {
            printf("%d,%f; ",pnodo->Info.a,pnodo->Info.b);
            pnodo = pnodo->ptr;
        }
        while(pnodo != NULL);
        printf("\n");
    }
}
```

El programa main:

```
printf("Hola mundo!\n");
Dato s1 = {1,2};
Dato s2 = {3,4};
Dato s3 = {5,6};
//necesito almacenar la direccion del primer elemento:
 nodo * inicio;

//tambien necesito un puntero para almacenar la direccion de cada
nuevo elemento y manipularlo:
 nodo * pnodo;

//creo primer nodo:
pnodo = CrearNodo(s1);

//guardo esta primer direccion como inicio de la lista:
inicio = pnodo;

//creo segundo nodo y debo enlazar este nuevo nodo al primero
pnodo->ptr = CrearNodo(s2);
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
pnodo = pnodo->ptr;

//creo tercer nodo y debo enlazar este nuevo nodo al segundo
pnodo->ptr = CrearNodo(s3);
//como el primer nodo ya está enlazado al segundo, actualizo el
puntero pnodo:
pnodo = pnodo->ptr;
//entonces cada vez que agrego un nodo nuevo pnodo apunta al
ultimo y este contiene referencia a NULL

MostrarLista(inicio);
```

Ejercicios de listas:

1- Escriba un programa en C que concatene (una) dos listas ligadas de caracteres. El programa debe incluir la función concatenar que tome como argumentos apuntadores a ambas listas y concatene la segunda a la primera.
2- Escriba un programa que mezcle dos listas ordenadas de enteros en una sola lista ordenada de enteros. La función mezclar debe recibir apuntadores al primer nodo de cada lista a mezclar y debe devolver un apuntador al primer nodo de la lista mezclada.
3- Escriba un programa que inserte 25 enteros al azar, del 0 al 100, en una lista ligada. El programa debe calcular la suma de los elementos y el promedio de ellos en punto flotante.
4- Modifique el programa de gestión de alumnos desarrollado previamente para que utilice una lista simplemente enlazada en lugar de un arreglo de estructuras:
    a. Defina la estructura adecuada para almacenar los datos de un alumno (nombre, apellido, legajo, notas) y la estructura autorreferenciada que permita su inserción en una lista
    b. Implemente una función de carga de datos de los alumnos que inserte cada nuevo alumno de manera tal que la lista siempre se encuentre ordenada por legajo
    c. Implemente una función que calcule el promedio de cada alumno y de todo el curso
    d. Implemente una función que muestre en pantalla el listado completo de alumnos ordenado alfabéticamente o por legajo

Estructuras dinámicas particulares:

Son versiones restringidas de listas dinámicas. Es decir que poseen un mecanismo particular para la inserción, extracción y recorrido de nodos.

Pilas: tipo Ultima en entrar, primero en salir (LIFO)

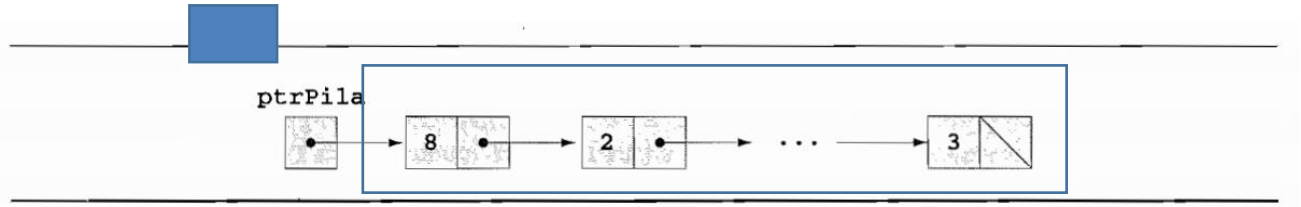


Figura 12.7 Representación gráfica de una pila.

Colas: tipo primero en entrar, primero en salir (FIFO)

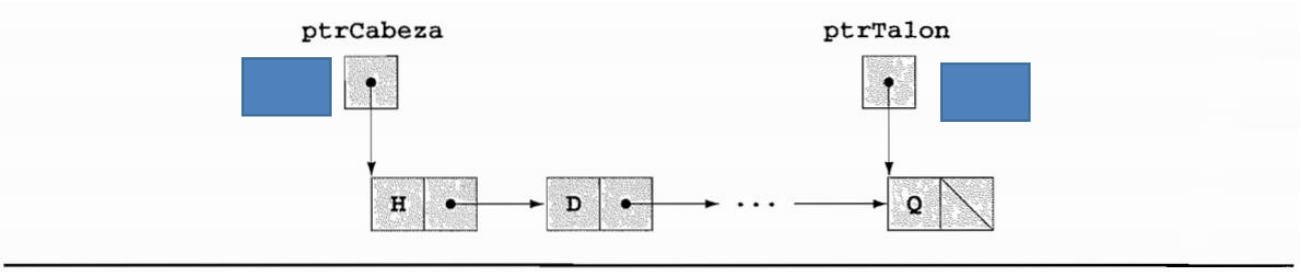


Figura 12.12 Representación gráfica de una cola.

Arboles: estructura no lineal

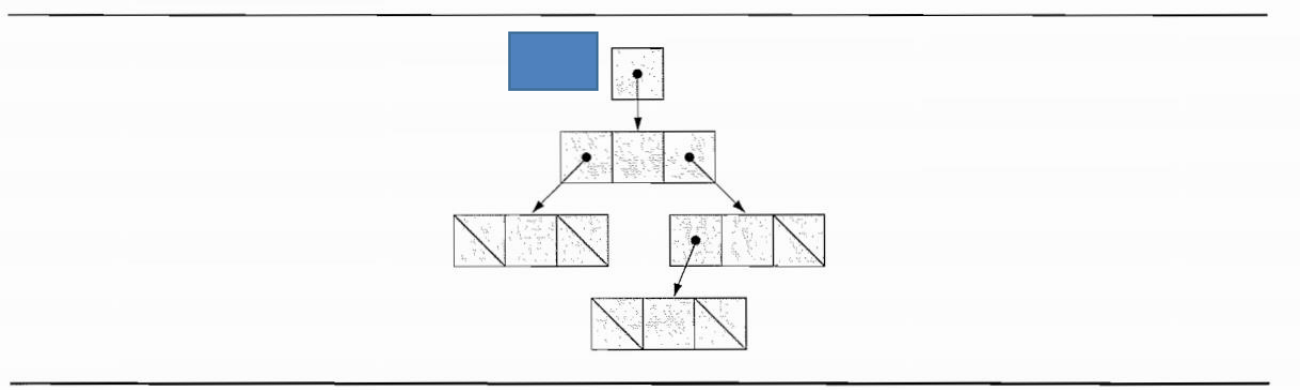


Figura 12.17 Representación gráfica de un árbol binario.

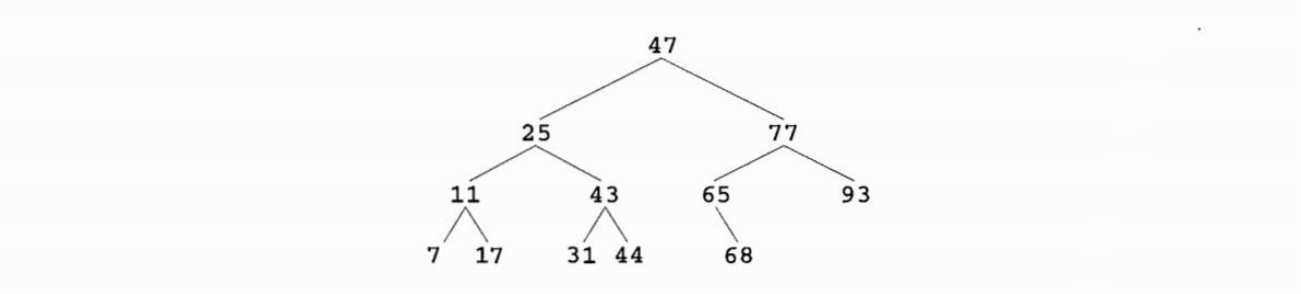


Figura 12.18 Árbol binario de búsqueda.

Operaciones básicas:

Pilas:

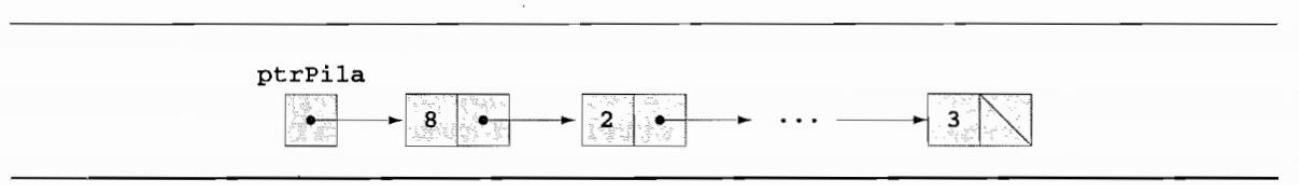
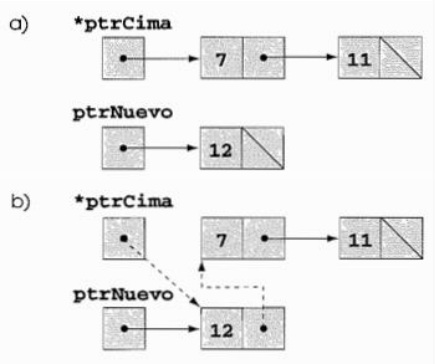


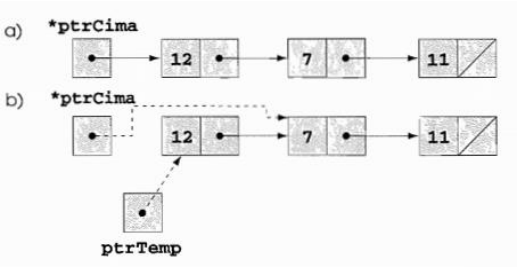
Figura 12.7 Representación gráfica de una pila.

Inserción de nodo (push):




```c
void push(ptrNodoPila *ptrCima, int info)//push
{
    //insertar un nuevo dato en la pila:
    //crear un nuevo nodo

    //crear una variable con puntero a structura nodo:
    ptrNodoPila ptrNuevo;//puntero a nodo


    //Reservo memoria de manera dinamica
    ptrNuevo = (ptrNodoPila) malloc(sizeof(NodoPila));

    //inserto el nodo en la cima de la pila
    if(ptrNuevo != NULL)
    {
        ptrNuevo->dato = info;
        ptrNuevo->ptrSiguiente = *ptrCima;
        *ptrCima = ptrNuevo;
    }
    else
        printf("No hay memoria disponible.\n");

}
```

Extracción de nodo (pop):




```c
int pop(ptrNodoPila *ptrCima)
{
    ptrNodoPila ptrTemp = *ptrCima;//guardo temporalmente el ultimo nodo

    int valor = (*ptrCima)->dato;//extraigo su informacion

    //actualizar el puntero a la cima de la pila
    *ptrCima = (*ptrCima)->ptrSiguiente;
    //libero memoria del nodo elimado
    free(ptrTemp);
    return valor;

}
```

Recorrer y mostrar una pila:

```c
void imprimePila(ptrNodoPila ptrActual)
{
    if(ptrActual == NULL)
    {
        printf("Pila vacia.\n");
    }
    else
    {
        printf("La pila es: ");

        while(ptrActual != NULL)//si nodo actual es nulo, he llegado al fin de la pila
        {
            printf("%d --> ", ptrActual->dato);
            ptrActual = ptrActual->ptrSiguiente;
        }
        printf("NULL\n");
    }
}
int estaVacia(ptrNodoPila ptrCima){return ptrCima == NULL;}
```

Colas:

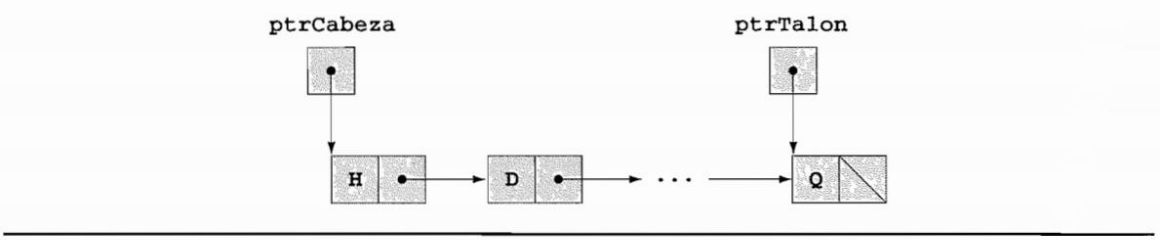
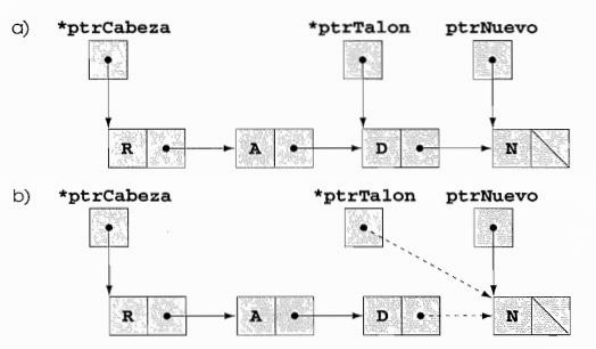


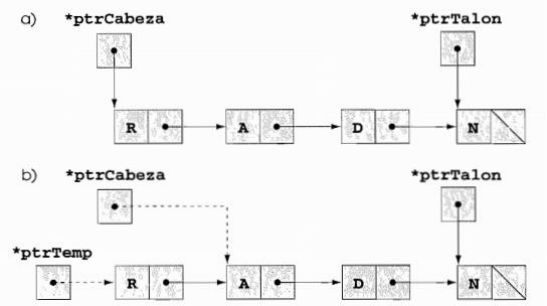
Figura 12.12 Representación gráfica de una cola.

Insertar nodo:




```c
void encolar(ptrNodoCola *ptrCabeza, ptrNodoCola *ptrCola, char value)
{
   ptrNodoCola newPtr = malloc(sizeof(NodoCola));

   if (newPtr != NULL) { // is space available
      newPtr->data = value;
      newPtr->ptrSig = NULL;

      // if empty, insert node at head
      if (estaVacia(*ptrCabeza)) {
         *ptrCabeza = newPtr;
      }
      else {
         (*ptrCola)->ptrSig = newPtr;
      }

      *ptrCola = newPtr;
   }
   else {
      printf("%c not inserted. No memory available.\n", value);
   }
}
```

Extraer nodo:




```c
char desencolar(ptrNodoCola *ptrCabeza, ptrNodoCola *ptrCola)
{
   char value = (*ptrCabeza)->data;
   ptrNodoCola tempPtr = *ptrCabeza;
   *ptrCabeza = (*ptrCabeza)->ptrSig;

   // if queue is empty
   if (*ptrCabeza == NULL) {
      *ptrCola = NULL;
   }

   free(tempPtr);
   return value;
}
```

Recorrer y mostrar contenido de una cola:

```c
void imprimirCola(ptrNodoCola currentPtr)
{
   // if queue is empty
   if (currentPtr == NULL) {
      puts("Queue is empty.\n");
   }
   else {
      puts("The queue is:");

      // while not end of queue
      while (currentPtr != NULL) {
         printf("%d --> ", currentPtr->data);
         currentPtr = currentPtr->ptrSig;
      }

      puts("NULL\n");
   }
}
```

Arboles binarios de búsqueda (ABB):

Definición de nodo:

```
struct nodoArbol
{
    struct nodoArbol *ptrIzq;
    int dato;
    struct nodoArbol *ptrDer;
};

typedef struct nodoArbol NodoArbol;
typedef NodoArbol *ptrNodoArbol;
```

Inserción de nuevo nodo en un ABB manteniendo el orden:

```
void insertarNodo(ptrNodoArbol *ptrArbol, int valor)
{
    //Si el arbol esta vacio-> este es el nodo raiz (o un padre)
    if(*ptrArbol == NULL)
    {
        *ptrArbol = malloc(sizeof(NodoArbol));
        if(*ptrArbol != NULL)//memoria ok
        {
            (*ptrArbol)->dato = valor;
            (*ptrArbol)->ptrDer = NULL;
            (*ptrArbol)->ptrIzq = NULL;
        }
        else
        {printf("No hay memoria disponible. \n");}

    }
    else//si el arbol no esta vacio el nodo es un hijo, debo determinar si es hijo izq o der
    {
        if (valor < (*ptrArbol)->dato)
        {
            insertarNodo(&((*ptrArbol)->ptrIzq),valor);
        }
        else
            if (valor > (*ptrArbol)->dato)
            {
                insertarNodo(&((*ptrArbol)->ptrDer),valor);
            }
            else
            {printf("Valor ya existente.\n");}
    }
}
```

Recorrido del ABB en los tres modos posibles:

```
void inOrden(ptrNodoArbol ptrArbol)
{
   // si el arbol no es vacio lo puedo recorrer o es NULL cuando llego a un nodo final u hoja
   if (ptrArbol != NULL) {
      inOrden(ptrArbol->ptrIzq);

      //if(ptrArbol->dato > 7) si solo quiero mostrar los mayores a 7
          printf("%3d", ptrArbol->dato);


      inOrden(ptrArbol->ptrDer);
   }
}


void preOrden(ptrNodoArbol ptrArbol)
{
    // si el arbol no es vacio lo puedo recorrer o es NULL cuando llego a un nodo final u hoja
   if (ptrArbol != NULL) {
      printf("%3d", ptrArbol->dato);
      preOrden(ptrArbol->ptrIzq);
      preOrden(ptrArbol->ptrDer);
   }
}


void postOrden(ptrNodoArbol ptrArbol)
{
    // si el arbol no es vacio lo puedo recorrer o es NULL cuando llego a un nodo final u hoja
   if (ptrArbol != NULL) {
       postOrden(ptrArbol->ptrIzq);
       postOrden(ptrArbol->ptrDer);
      printf("%3d", ptrArbol->dato);
   }
}
```

Ejercicios (extraídos del Deitel y guía Ing. Carrera):

Pilas:

1) Se desea codificar una pila dinámica. Los nodos contienen números enteros. Cargar la pila con 10 nodos, recorrerla y mostrar su contenido en pantalla.
2) Codificar un programa que permita cargar una pila dinámica con N caracteres ingresados por teclado, recorrerla y mostrar su contenido en pantalla.
3) Se desea codificar una pila dinámica. Los nodos contienen Apellido y edad de personas. Cargar la pila hasta que se ingrese la edad 999. Se desea recorrerla y mostrar por pantalla la información de los nodos que representen personas mayores de edad (21 años).
4) Se crean dos pilas, una de ellas con 10 lecturas de concentración de gas provenientes de un sensor y la otra con 10 lecturas del nivel de líquido de gas del cartucho que lo contienen (que se corresponden una a una con las lecturas de concentración). Se desea tener una sola estructura de tipo pila con ambas informaciones.

Colas:

1) Se desea codificar una cola dinámica. Los nodos contienen números enteros. Cargar la cola con 10 nodos, recorrerla y mostrar su contenido en pantalla.
2) Se tiene cargada una estructura de tipo cola. La información de los nodos son números enteros, la cola admite tener nodos con datos repetidos. Generar otra estructura del mismo tipo sin las repeticiones. Recorrer las dos estructuras dinámicas y mostrar por pantalla sus contenidos.

Arboles binarios:

1) Implementar un árbol de búsqueda binaria con 10 nodos, los nodos contienen números enteros sin repetición. Se ingresan por teclado de forma desordenada.
2) Implementar una función recursiva que permita recorrer el árbol anterior in orden para mostrar el contenido del árbol. Luego implementar la lectura en pre orden y post orden.
3) Se tiene una estructura dinámica de tipo pila ya almacenada en memoria de sistema. Con los datos almacenados en esta generar un árbol de búsqueda binaria.
4) Implementar un árbol binario de búsqueda con registros de personas (nombre, edad, sexo, ID) los nodos se organizan por ID. Recorrerlo y mostrar por pantalla.
5) Implementar un árbol binario de búsqueda con registros de alumnos (nombre, edad, sexo, promedio general, numero de inasistencias, ID) los nodos se organizan por ID. Recorrerlo y mostrar por pantalla:
    a. Los alumnos con promedio mayor a 7
    b. Los alumnos con menos de 5 inasistencias
6) Escriba una función llamada profundo, que reciba un árbol binario y determine cuantos niveles tiene
7) Escriba una función BusquedaArbolBinario que intente localizar un valor especificado en un árbol binario de búsqueda. La función debe tomar como argumentos un apuntador al nodo raíz del árbol binario y una clave de búsqueda a localizar. Si se encuentra el nodo con la clave de búsqueda la función debe devolver un puntero hacia ese nodo, de lo contrario devolver NULL.

Referencias:

- Programación en C, Arquitectura de Sistemas UC3M, http://www.it.uc3m.es/pbasanta/asng/course_notes/c_programming_part_es.html
- Fundamentos de programación. Piensa en C. O. C. Battistutti, 2006
- Como programar en C, C++ y Java 4ta ed. Deitel y Deitel, 2004
- Programación en C. Serie Schaum. 2da edición, 2005
- https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_de_archivos