Unidad 7
ESTRUCTURAS DINÁMICAS
Desarrollo de la unidad :

Prácticas : Implementación de algoritmo de lista en C , Pilas y Colas,
Ejercicios : Cola de Impresión, Servidor de Internet, Invertir un fichero
Conceptos:
Memoria dinámica, tablas con memoria dinámica,
Listas encadenadas: Listas, Anillos, Dobles, Pilas y Colas
Árboles, Binarios, Ordenados, Equilibrados
Grafos

Introducción

Cuando definimos en un programa una variable estática estamos fijado previamente cual va a ser su espacio en memoria y cuales van a ser los posibles valores que puede tomar a lo largo de la ejecución del programa. Existen problemas complejos que se resuelven más eficazmente mediante la utilización de variables que cambien dinámicamente la cantidad o el tipo de datos que pueden contener. Este tipo de estructuras de datos se denomina estructuras dinámicas.

Características:

- Pueden variar el tamaño (Número de datos) a lo largo de la ejecución el programa
- Se adaptan mejor a la naturaleza del problema, aprovechando más eficientemente los recursos de memoria.
- Son más complejas de manejar pues debemos controlar el crecimiento o reducción del espacio de memoria que ocupan.
- Se almacenan en memoria principal
- Una vez definidas se utilizan como cualquier variable estática.

Operaciones especiales

Creación : Petición al sistema de una cantidad determinada de memoria para almacenar la nueva variable
En Lenguaje C disponemos de la función malloc (alloc memory) pedir memoria, nos devuelve la dirección de memoria donde podemos guardar los datos, un puntero a una nueva zona de memoria o NULL en caso de que no exista memoria disponible.

Ejemplo:

```
char *cadena;
float *preal;
TipoDato *pdato;

cadena = ( char * ) malloc ( 30 * sizeof(char) );   Espacio para 30 caracteres
preal  = ( float * ) malloc ( 100 * sizeof(float));  Espacio para 100 número reales
pdato =  ( TipoDato *)  malloc ( sizeof(TipoDato); Espacio para una variable de TipoDato
```

Una vez asignadas podemos trabajar como cualquier variable tipo puntero

```
strcpy(cadena, "Hola pepe);
preal[10] = 3.1416;
pdato->edad = 15;
```

Destrucción: Devolución de la memoria, una vez libera la memoria no podemos volver a utilizar la variable, hasta que no reservemos de nuevo espacio.

free (puntero);

```
free (cadena );
cadena[3] = 'a'; // Produciría error de  ejecución.
```

ESTRUCTURAS DINÁMICAS

Tablas dinámicas:

- Tablas donde se define su tamaño en tiempo de ejecución.

No son estructuras dinámicas puras pues una vez definidas no permiten el cambio de tamaño.

Ejemplos de la web:

1. Crear un una tabla de un tamaño determinado y mostrar su contenido
2. Unir Tablas char * UnirCadenas (char *cad1, char * cad2);
3. Cargar un fichero de texto con información de configuración y consultar su contenido
4. Ordenar un fichero de texto con una tabla de cadenas creadas por memoria dinámica

Estructuras dinámicas verdaderas

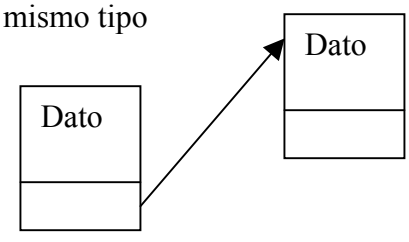
La tablas creadas mediante memoria dinámica no son verdaderas estructuras dinámicas pues una vez definidas en tiempo de ejecución, no pueden cambiar su tamaño, sólo liberarse todo el espacio que ocupas y volver a definirse.

Tipos: Listas, árboles y grafos.

La mayor parte de la estructura dinámicas se realizan mediante estructuras auto-referenciadas. Son un tipo de estructuras que contienen punteros a elementos de su mismo tipo

Ej.-

```
struct  SElemento {
                    TipoDato dato;
                    struct SElemento *sig;  // Un puntero a estructuras del su mismo tipo
                  };

struct Melemento {
                  TipoDato dato;
                   struct Melemento *pun[4]; // 4 Punteros
                   }

typedef struct SElemento TipoDinámicoAuto1;
typedef struct Melemento TipoDinámicaAuto4;
```



LISTAS ENCADENADAS

Definición: Es una secuencia de cero o más elementos de un mismo tipo, en donde cada elemento tiene un elemento anterior y un posterior o siguiente. El orden de los elementos se establece mediante punteros.

Existe un primer elemento que no tiene anterior y un último que no posee posterior,

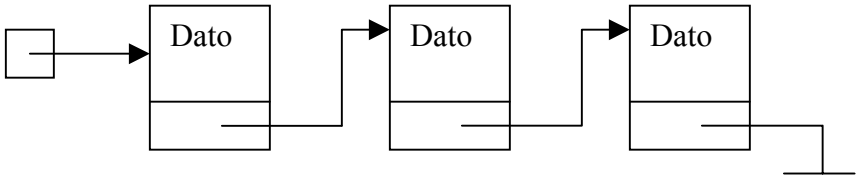
- Una lista puede crecer o decrecer con facilidad sin tener que desplazar su elementos, está limitada por la cantidad máxima de memoria que disponga el proceso.
- Las listas se pueden implementación mediante estructuras autoreferenciadas o mediante una tablas, estando limitado en este caso el número máximo de elementos al tamaño de la tabla.

Implementación mediante estructuras autoreferenciadas:

```
typedef int TipoDato; // TipoDato puede ser cualquier tipo.
struct SEle{
    TipoDato valor;
    struct SEle *sig;
    };

typedef struct SEle Elemento;

Elemento *Base;
```

Representación Gráfica:

Base



Operaciones Básicas:

1. Creación
2. Primero
3. Ultimo
4. Poner Al principio
5. Poner Al Final
6. Borrar Al principio
7. Borrar Al final
8. Borrar Toda la Lista
9. Inserta Elemento
10. Borrar Elemento

Ejercicios propuestos:

- Unir dos listas.
- Crear dos listas una de pares y otra de impares ceder elemento de una a otra.
- Crear una lista a partir de una cadena de caracteres.
- Invertir una lista.

```
/* ------------------------------------------------------------ */
/* IMPLEMENTACION DE FUNCIONES BASICAS UNA LISTA ENCADENADA     */
/* ------------------------------------------------------------ */

#include <stdio.h>
#include <alloc.h>

typedef enum {FALSE=0,TRUE=1} boolean;

/* Tipo de dato que almacena cada elemento de la lista */
typedef int TipoDato;

/* Estructura autoreferenciada */
struct SEle{
      TipoDato valor;
      struct SEle *sig;
      };

typedef struct SEle Elemento;

/* Variable global Puntero de Inicio de la Lista */
Elemento  *Base;

/* ------------------------------------------------- */
/* Funciones de Manejo de UNA lista encadenada       */
/* ------------------------------------------------- */
void     Crear   (void);
boolean  Vacia   (void);
TipoDato Primero (void);
TipoDato Ultimo  (void);
boolean  PonAlFinal       (TipoDato Dato);
boolean  PonAlPrincipio    (TipoDato Dato);
boolean  BorrarAlFinal     (void);
boolean  BorrarAlPrincipio (void);
boolean  InsertarEnOrden   (TipoDato Dato );
boolean  BorrarElemento    (TipoDato Dato );
void     VerLista    (void);
void     VaciarLista (void);

/*--------------------------------------------------------------*/
/* Inicializa el puntero de la Lista */
void Crear(void)
{
  Base = NULL;
}

/*--------------------------------------------------------------*/
/* Indica si esta o no vacía */
boolean Vacia(void)
{
  return ( Base == NULL)?TRUE:FALSE;
}

/*--------------------------------------------------------------*/
/* Devuelve el valor almacenado en
   el primer elemento de la Lista */
TipoDato Primero(void)
{
  return Base->valor;
}

/*--------------------------------------------------------------*/
/* Devuelve el valor almacenado en
   el último elemento de la Lista */
TipoDato Ultimo(void)
{
  Elemento *paux;
  paux = Base;
  while ( paux->sig != NULL )
    {
    paux = paux->sig;
    }
  return paux->valor;
}
```

```
/*-----------------------------------------------------------------*/
/* Crea un nuevo elemento al principio de la Lista */
boolean PonAlPrincipio ( TipoDato Dato)
{
  Elemento *pnuevo;
  boolean resu;

  resu = FALSE;
  pnuevo = malloc( sizeof(Elemento) );
  if ( pnuevo != NULL )
    {
    pnuevo->sig = Base;
    pnuevo->valor = Dato;
    Base = pnuevo;
    resu = TRUE;
    }
  return resu;
}

/*-----------------------------------------------------------------*/
/* Crea un nuevo elemento al final de la lista */

boolean PonAlFinal  ( TipoDato Dato)
{
   Elemento *paux,*pnuevo;

   /* Creo el elemento nuevo */
   pnuevo = malloc( sizeof(Elemento));
   if ( pnuevo == NULL )
     {
     /* Si no pude crear el elemento termina
      la función */
     return FALSE;
     }

   /* Relleno el nuevo elemento */
   pnuevo->valor = Dato;
   pnuevo->sig   = NULL; /* Va a ser el último */

  /* Si no está vacia recorro la lista
    hasta alcanzar el último elemento */
  if ( Base != NULL )
    {
    paux = Base;
    while ( paux->sig != NULL )
                    {
                    paux = paux->sig;
     }
    /* El puntero siguiente del último
      elemento señala al nuevo */
    paux->sig   = pnuevo;
    }
 else
    {
    /* La lista está vacia */
    /* Base señala al nuevo elemento */
    Base  = pnuevo;
    }
 return TRUE;
}
/*-----------------------------------------------------------------*/
/* Elimina el primer elemento
   de la lista */
boolean BorrarAlPrincipio   (void)
{
  Elemento *paux;

  paux = Base;
  if ( !Vacia() )
    {
    /* Señala al siguiente y libero memoria */
    Base = Base->sig;
    free(paux);
    return TRUE;
    }
 return FALSE;
}
```

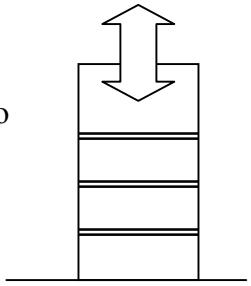
```
/*-----------------------------------------------------------------*/
/* Borra el último elemento de la lista */
boolean BorrarAlFinal(void)
{
  Elemento *paux1,*paux2;
  if ( Vacia() )
    {
     return FALSE;
    }
  else
    {
    /* Si solo hay uno */
    if ( Base->sig == NULL )
                                {
                                free(Base);
                                Base = NULL;
                                }
              else
                                {
               paux1 = Base;      /* Primer elemento */
                         paux2 = Base->sig; /* Segundo elemento */
               while ( paux2->sig != NULL )
              {
         paux1 = paux2;
         paux2 = paux2->sig;
         }
        /* Pon el puntero siguiente del
          elemento anterior a NULL */
        paux1->sig = NULL;
        /* Libero el espacio del último elemento */
        free(paux2);                          }
    return TRUE;
    }
}
/*-----------------------------------------------------------------*/
/* Coloca un elemento en la lista
   insertando en orden creciente */

boolean InsertarEnOrden( TipoDato Dato )
{
  Elemento *paux1, *paux2, *pnuevo;

  // Creo el espacio para el nuevo elemento
  pnuevo = malloc (sizeof (Elemento) );
  if ( pnuevo == NULL )
    {
    return FALSE;
    }
  pnuevo->valor = Dato;

  /* Si está vacia */
  if ( Base == NULL )
    {
    Base = pnuevo;
    pnuevo->sig = NULL;
    }
  else
    {
    /* Si está antes del primero */
    if ( Base->valor > pnuevo->valor )
      {
      pnuevo->sig = Base;
      Base = pnuevo;
      }
    else
      {
      paux1 = Base;      /* Primero */
      paux2 = Base->sig; /* Segundo */
      while ( paux2 != NULL )
      {
       if ( paux2->valor > pnuevo->valor )
         {
         break;
         }
       paux1 = paux2;
       paux2 = paux2->sig;
      }
      // Inserto
      paux1->sig = pnuevo;
      pnuevo->sig = paux2;
      }
    }
   return TRUE;
}
```

```
/*-------------------------------------------------------------*/
/* Borra un elemento determinado */
boolean BorrarElemento ( TipoDato Dato )
{
   Elemento *paux1,*paux2;
  /* Si no hay elementos */
  if ( Base == NULL )
     {
     return FALSE;
     }
  /* Si es el primero */
  if ( Base->valor == Dato )
     {
     paux1 = Base;
     Base = Base->sig;
     free(paux1);
     }
  else
     {
     /* Busco el elemento */
     paux1 = Base;
     paux2 = Base->sig;
     while ( paux2 != NULL )
      {
      if ( paux2->valor == Dato )
         {
         break;
         }
      paux1 = paux2;
      paux2 = paux2->sig;
      }
     if ( paux2 == NULL )
        {
        /* No está */
        return FALSE;
        }
     else
        {
        paux1->sig = paux2->sig;
        free(paux2);
        }
     }
return TRUE;
}
/*-------------------------------------------------------------*/
/* Muestra el contenido de la lista */
void VerLista ( void )
{
  Elemento *paux;

  printf("\n LISTA: [");
  paux = Base;
  while ( paux != NULL )
   {
   printf("->%d ",paux->valor);
   paux = paux->sig;
   }
  printf("]\n");
}


/*-------------------------------------------------------------*/
/* Borrar todos los elemento de la lista */
void VaciarLista(void)
{
   Elemento *paux;
   while ( Base != NULL )
    {
    paux = Base;
    Base = Base->sig;
    free(paux);
    }
```

ESTRUCTURAS ABSTRACTAS RELACIONADAS: PILAS Y COLAS

Las Pilas y Colas son tipos abstractos de datos utilizados en algoritmos para resolver un gran número de problemas, especialmente en software de sistemas. Su implementación más común es mediante listas encadenadas.

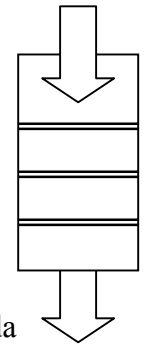
Operaciones básicas sobre pilas y colas:
Crear, Destruir, Poner_Elemento, Sacar_Elemento, Preguntar_Si_Vacia.



Pila: protocolo LIFO *(least Input First Output) Ultimo al entrar primero al salir.*

Primero se atiendes los más recientes, Se extraen y se introducen elementos por el mismo extremo.

Ejemplos
- Pila ejecución con las llamadas a funciones en la ejecución de un programa
- Pila de Ventanas a la hora de visualizar en la pantalla
- Los cuentos de las mil y una noches



Cola : protocolo FIFO *(First Input First Output) Primero al entrar, primero al salir*

Primero se atiende el que lleva más tiempo, Un extremo de entrada y otro de salida

Ejemplos
- Cola para pedir un ticket
- Cola de impresión
- Colas se peticiones de envío de correo
- Procesos en espera de CPU

Otras: Colas no estrictas o con prioridades : Cuando cada elemento tiene una prioridad determinada y el orden de salida no viene sólo por la posición de la cola sino por la prioridad.

Ejercicios sobre pilas y colas

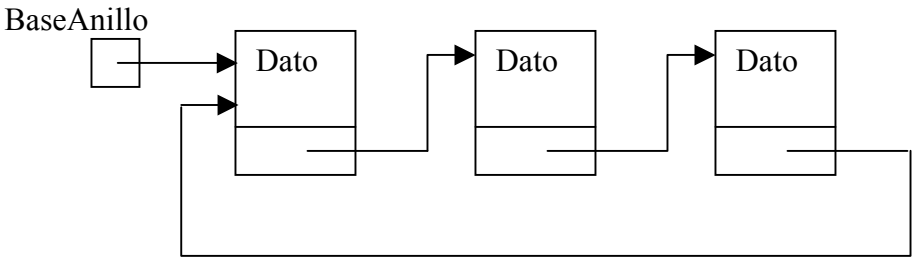
- Implementación de una Pila mediante listas encadenadas
- Implementación de una Cola mediante listas encadenas
- Implementación de una Cola con prioridades
- Implementación de una Cola mediante una tabla.
- Implementación de una Pila mediante una tabla
- Invertir un fichero mediante una pila de cadenas
- Guardar las cinco últimas líneas de un fichero mediante una Cola.

Otros ejercicios propuestos

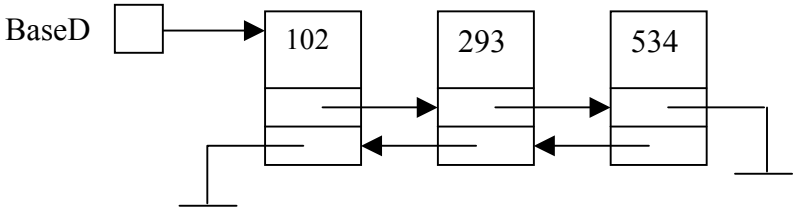
- Gestionar un cola de Impresión de fichero
- Cola de envío de mensajes por Internet
- Análisis de expresiones en notación polaca.

OTRAS ESTRUCTURAS CON LISTAS ENCADENADAS

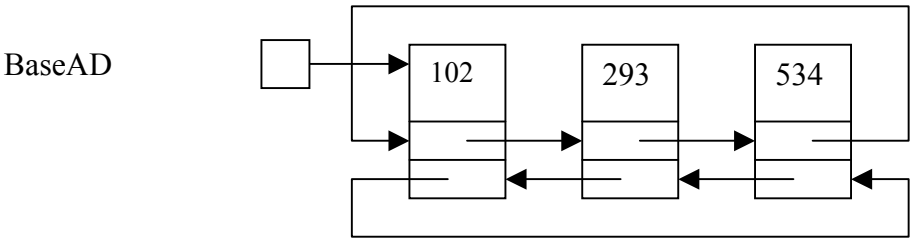
- **Listas circulares o Anillos:** Son lista encadenadas donde el último elemento señala al primero



- **Listas doblemente enlazadas.** Son listas donde cada elemento tiene dos punteros, uno señalando al elemento siguiente y otros señalando al anterior. Permite un recorrido en ambos sentidos,



- **Anillos doblemente enlazados.** Anillos con dos punteros: anterior y siguiente.



ÁRBOLES

Concepto:

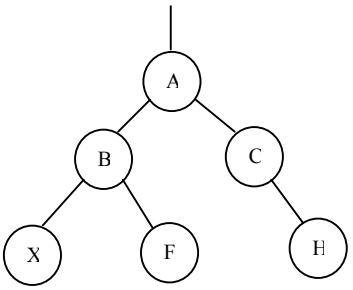
Es una estructura dinámica donde cada elemento tiene un elemento anterior y cero, uno o varios posteriores. Existe un elemento raíz que no tiene antecesores.

La estructura árbol se utiliza en multitud de estructuras de datos Permite representar estructuras jerárquicas, facilitar la ordenación y búsqueda de datos, representar expresiones matemáticas, semánticas, evolución de un sistemas, deducciones lógicas, árbol de directorios, etc.

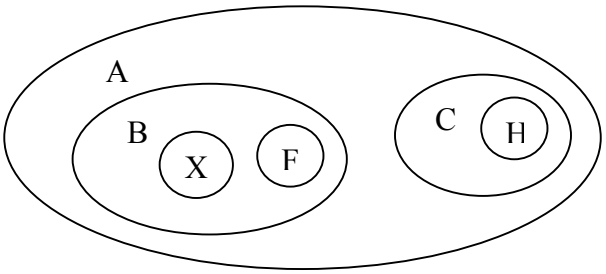
Una lista es un caso especial de árbol con una sola rama.

Representación gráfica

- Nodos enlazados



- Conjunto concéntricos



- Lista con paréntesis

(A (B ((X) (F))) (C (H)))

Ej.- La expresión matemática (80 * (3 +4)) / 15 en estructura árbol
(/ (15) (* (80) (+ (3) (4))))

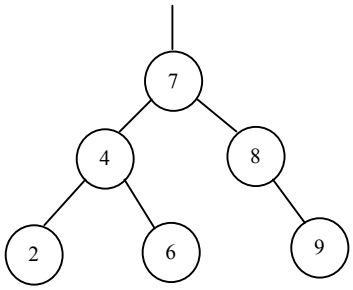
Elementos y propiedades de un árbol

- **Nodo:** Cualquier elemento del árbol: A,B,X,F,C,H
- **Raíz :** Primer elemento sin elemento anterior: A
- **Hoja:** Elemento sin elemento posterior: X,F,H
- **Subárbol**: Rama de un árbol (Ej- (B (X) (F))
- **Nivel de un elemento**: Distancia entre la raíz y un elemento: Nivel(A) = 1, Nivel(X) = 3
- **Grado de un árbol** : Número máximo de sucesores: En el ejemplo el grado = 2
- **Profundidad de un árbol:** Nivel máximo entre las hojas y la raiz: Profundidad = 3
- **Padre de un nodo** : Antecesor. Padre de F es B
- **Hijos de un nodo** : Sucesores. Hijos de B son X y F

TIPOS DE ÁRBOLES

Árboles binarios

Árboles de grado 2, cada nodo tiene 0,1 o 2 nodos hijos o descendientes
Raíz, subárbol derecho y subárbol Izquierdo.



Ej. Árbol binario ordenado y equilibrado.

Árbol binario de búsqueda (ordenado)

Un árbol de grado 2 donde sus elementos están ordenados de tal forma que la búsqueda de un elemento se realiza fácilmente.

Regla:

Los sucesores de un elemento por la izquierda son menores que él y los sucesores por la derecha son mayores.

Árbol equilibrado:

Si para cada nodo, el número de nodos que hay en el subárbol derecho y el izquierdo difieren como mucho en una unidad.

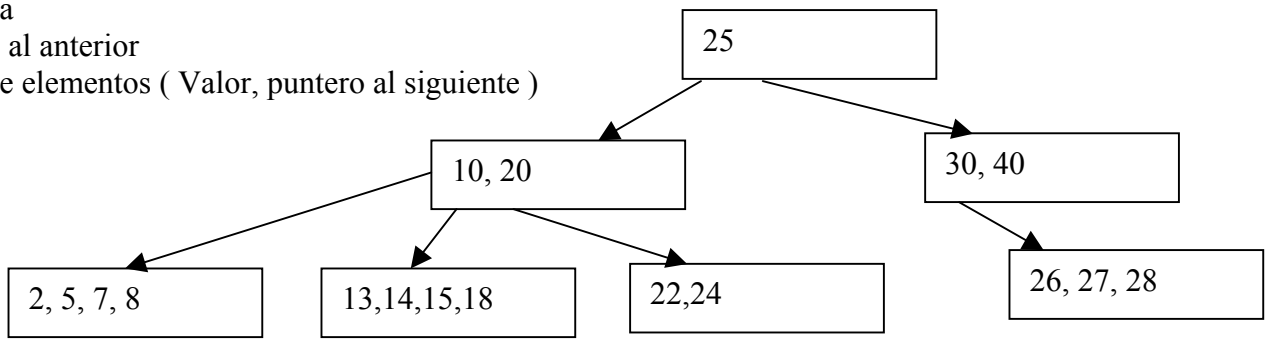
" Un árbol equilibrado tiene la mínima profundidad, poco ramificado, y si está ordenado la búsqueda será los más rápida posible "

Ventajas de los árboles binarios: Rápido acceso a la información, eficiente en memoria
Desventajas : Complejos método de altas y bajas si se quiere mantener el árbol equilibrado, Hay que mantener toda la estructura en memoria principal

Árboles B, B+

El mantener un árbol perfectamente equilibrado es bastante complejo y necesita de algoritmos no muy eficientes. Solución: Los árboles B diseñados por R. Bayer (Tipo especial de árboles múltiples) se agrupan los subárboles en páginas (Tablas ordenadas) , garantizando que todas las páginas almacenaban entre n y n/2 nodos (salvo la raíz) siendo N valor fijo. Las páginas pueden estar parte en memoria principal o en memoria secundaria, leyendo o grabando la página cuando sea necesario. Este tipo de estructura es muy utilizada para la indexación de grandes bases de datos.

Cada página
puntero al anterior
Tabla de elementos (Valor, puntero al siguiente)



Recorridos de un árbol Binario

(A (B (D E)) (C ((F (G H)) X)))

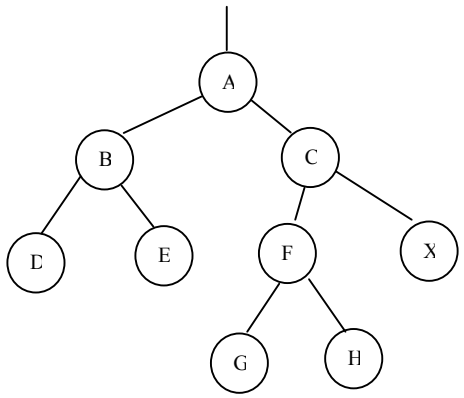
- En Amplitud (Por Niveles) A B C D E F X G H

- En profundidad:

Pre-Orden (A B D E C F G H X)
- Padre
- Rama Izda.
- Rama Dcha.

Orden Central (D B E A G F H C X)
- Rama Izda.
- Padre
- Rama Dcha.

Post-orden (D E B G H F X C A)
- Rama Izda.
- Rama Dcha.
- Padre.



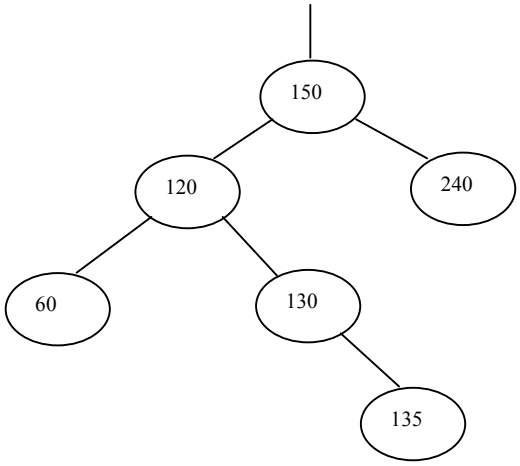
Ej.- Árbol binario no ordenado y no equilibrado

IMPLEMENTACIÓN EN LENGUAJE C

- **Mediante tablas:**
-

Valor, índice izquierda, índice derecha

	Valor	**Izda**	**Dcha**
1	150	3	3
2	120	4	5
3	240	0	0
4	60	0	0
5	130	0	6
6	135	0	0
..	..		
..	..		

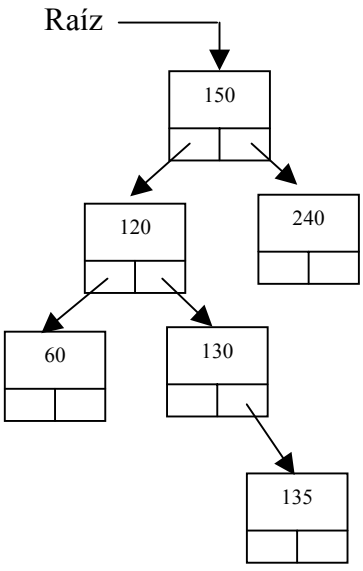


Ej.- Árbol binario ordenado y no equilibrado

- **Estructuras autoreferenciadas**

Valor, Puntero Derecha, Puntero Izquierda

```
struct  sarbol {
        TipoDato valor;
        struct sarbol *dcha;
        struct sarbol *izda;
        }
```



Algoritmos básicos sobre árboles binarios

Los algoritmos sobre árboles generalmente recursivos.

- Búsqueda de un elemento
- Recorridos: En Amplitud, En Profundidad (PreOrden, InOrden, PostOrden)
- Inserción de un elemento
- Borrado de un elemento
- Calculo de nivel, profundidad, etc.

Ejercicios planteados:

1.- Contar el número de nodos que tiene un árbol
2.- Contar el número de hojas de un árbol
3.- Obtener el elemento mayor.
4.- Diferencia entre el mayor y el menor

Ejercicios desarrollados con árboles:

- Generación de un índice de las palabras a partir de un fichero de texto
- Recorrido de un árbol de directorios
- Cargar los registros de un fichero en un árbol y volcarlo a un fichero orden

ALGORITMOS BÁSICOS DE ÁRBOLES IMPLEMENTADOS EN LENGUAJE C

```
/* BÚSQUEDA INTERATIVA DE  UN ARBOL BINARIO ORDENADO */
boolean BuscarI ( Elemento *parbol , TipoDato dato )
{
  Elemento *paux;
  boolean encontrado = FALSE;

  paux = parbol;

  while (( paux != NULL ) && ( !encontrado ) )
    {
    if ( dato == paux->valor )
      {
      encontrado = TRUE;
      }
    else
      {
      if ( dato > paux->valor )
        {
        paux = paux->dcha;
        }
      else
        {
        paux = paux->izda;
        }
      }
    }
  return encontrado;
}
```

```
/*
 * BUSQUEDA DE UN ELEMENTO DE FORMA RECURSIVA
 */

boolean BuscarR  ( Elemento *parbol, TipoDato dato )
{
  if ( parbol == NULL )
     {
     return FALSE;
     }
  else
     {
     if ( dato == parbol->valor )
       {
       return TRUE;
       }
     else
       {
        if ( dato > parbol->valor )
          {
           return  BuscarR(parbol->dcha,dato);
           }
         else
          {
          return  BuscarR(parbol->izda,dato);
         }
       }
     }
}
```

/* BUSCA RECURSIVA DE UN ELEMENTO SIN TENER EN CUENTA QUE EL ÁRBOL ESTÁ ORDENADO */

```
boolean BuscarNoOrden ( Elemento *parbol, TipoDato dato )
{
  if ( parbol == NULL )
    {
    return FALSE;
    }
  else
    {
    if ( parbol->valor == dato )
      {
      return TRUE;
      }
    else
      {
      return BuscarNoOrden(parbol->dcha, dato ) ||
            BuscarNoOrden(parbol->izda, dato );
      }
    }
}
```

```
/*
 EJEMPLOS DE RECORRIDOS RECURSIVOS
 */

void EnPreOrden ( Elemento *parbol )
{
  if ( parbol != NULL )
    {
    printf( " %d,", parbol->valor);
    EnPreOrden(parbol->izda);
    EnPreOrden(parbol->dcha);
    }
}
/* ---------------------------------------- */
void EnInOrden ( Elemento *parbol )
{

  if ( parbol != NULL )
    {
    EnInOrden(parbol->izda);
    printf( " %d,", parbol->valor);
    EnInOrden(parbol->dcha);
    }
}
/* ---------------------------------------- */
void EnPostOrden ( Elemento *parbol )
{
  if ( parbol != NULL )
    {
    EnPostOrden(parbol->izda);
    EnPostOrden(parbol->dcha);
    printf( " %d,", parbol->valor);
    }
}
```

```
/* ---------------------------------------- */
/* Cuenta cuantas hojas tiene el árbol      */
/* ---------------------------------------- */
int ContarHojas ( Elemento *parbol )
{
  if ( parbol == NULL )
    {
    return 0;
    }
  else
    {
    if ( ( parbol->dcha == NULL ) &&
        ( parbol->izda == NULL ) )
     {
     return 1; // Es una hoja
     }
    else
     {
     return ContarHojas(parbol->dcha) +
          ContarHojas(parbol->izda);
     }
    }
}
```

```
/* ---------------------------------------- */
/* Cuenta cuantos nodos tiene el árbol      */
/* ---------------------------------------- */
int ContarNodos ( Elemento *parbol )
{
   if ( parbol == NULL )
     {
     return 0;
     }
   else
     {
     return 1 + ContarNodos(parbol->dcha)
            + ContarNodos(parbol->izda);
     }
}
```

```
/* ---------------------------------------- */
/* Calcula la profundidad que tiene el árbol   */
/* ---------------------------------------- */

int Profundidad ( Elemento *parbol )
{
  int prodcha, proizda;
  if ( parbol == NULL )
     {
     return 0;
     }
  prodcha = 1 + Profundidad(parbol->dcha);
  proizda = 1 + Profundidad(parbol->izda);
  // Devuelvo la mayor profundidad
  return ( prodcha > proizda )? prodcha:proizda;

}
```

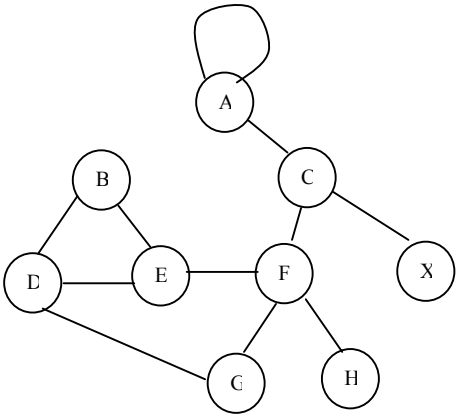
```
// ---------------------------------------------------------
//  INSERTA ORDENADAMENTE UN NUEVO ELEMENTO
// ---------------------------------------------------------
Elemento *  Insertar ( Elemento *parbol, TipoDato dato )
{
  Elemento *paux;
  paux = parbol;
  if ( paux == NULL )
    {
    paux = malloc ( sizeof ( Elemento ) );
    if ( paux != NULL )
       {
       paux->valor = dato;
       paux->dcha = NULL;
       paux->izda = NULL;
       }
    }
  else
    {
    if ( dato > paux->valor )
       {
       paux->dcha =  Insertar( paux->dcha,dato);
       }
    else
       {
       paux->izda = Insertar (paux->izda,dato);
       }
    }
  return paux;
}
```

GRAFOS

Concepto:

Conjuntos de elementos relacionados, donde cada elemento puede tener varios sucesores y varios antecesores.

Los árboles y las listas son un tipo especial de grafo.



Grafo no dirigido, no etiquetado, no valorado, incompleto, conexo y simple.

Ejemplos

- Líneas de Metro, Recorrido de una ciudad
- Relaciones entre elementos
- Estados de un sistema
- Esquema de una red de datos

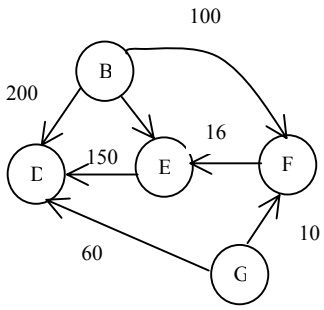
Componentes

- Nodos / Vértices : Cada uno de los elementos de un grafo
- Aristas / Arcos : Enlaces entre cada par de nodos
- Camino: Secuencia de aristas que conectan dos nodos
- Longitud de camino: Número de aristas o arcos que tiene un camino
- Bucle : Arco que conecta un nodo consigo mismo
- Ciclo: Camino cerrado con longitud mayor que 2

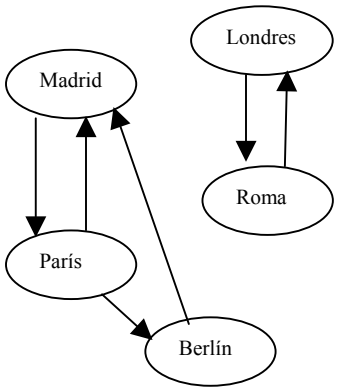
Tipos de grafos

- Dirigido / No Dirigido: Las aristas tiene un sentido, flechas)
- Etiquetado / No Etiquetado (Cada arista tiene un nombre)
- Valorado / No Valorado (Cada arista tiene un valor o peso)
- Completo / Incompleto (Cada nodo esta unido con todos los demás)
- Conexo-conectado / Inconexo-desconectado (Existe un camino entre cualquier par de nodos)
- Simple / Múltiple (una arista entre cada nodo)

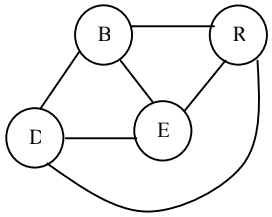
Ejemplos:



Grafo dirigido, no etiquetado, valorado, incompleto, conexo, simple.



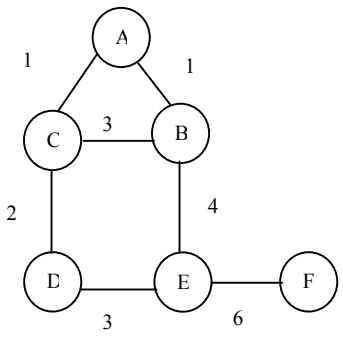
Grafo dirigido, no etiquetado ,no valorado, incompleto ,inconexo, múltiple.



Grafo no dirigido, no etiquetado ,no valorado, completo ,conexo ,simple.

ALGORITMOS BÁSICOS:

Recorrido de un grafo: obtener un camino entre dos nodos, en un grafo valorado obtener el camino de coste mínimo.

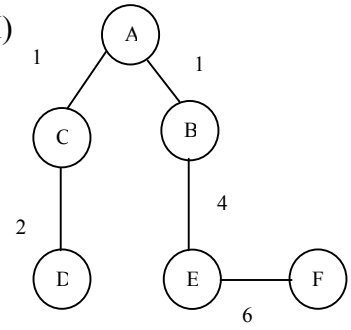


Caminos posibles de A a D:

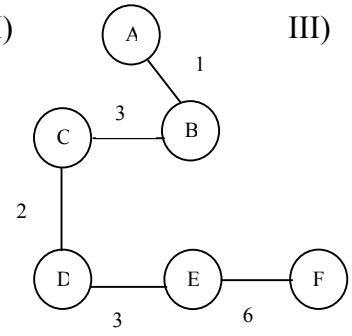
A C D = 1 + 2 = 3 (*Camino de coste mínimo*)
A B C D = 1 + 3 + 2 = 5
A B E D = 1 + 4 + 3 = 8
A C B E D = 1 + 3 + 4 + 3 = 11

Árboles de recubrimiento : Un árbol basado en la estructura de grafo que permite conectar a todos los nodos (Establece un único camino entre dos nodos)
Árboles de recubrimiento de coste mínimo, cuando el grafo es valorado es el árbol de recubrimiento cuya suma del valor de las aristas sea mínimo.

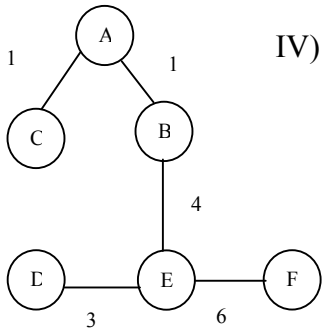
I)



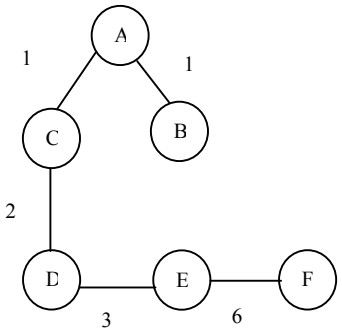
II)



III)



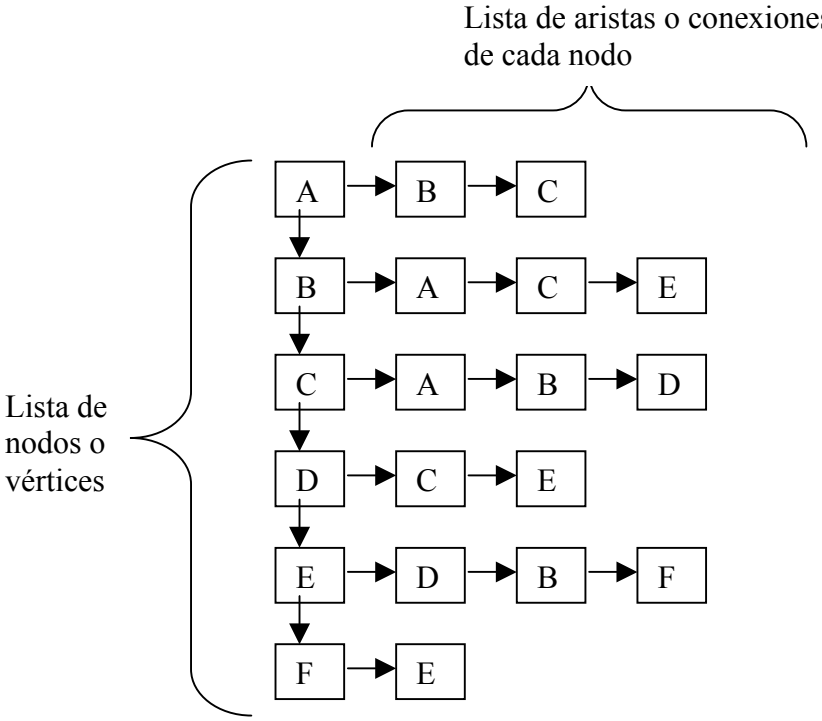
IV)



I) 1 + 1 + 2 + 4 + 6 = 14
II) 1 + 3 + 2 + 3 + 6 = 15
III) 1 + 1 + 4 +3 + 6 = 15
IV) 1 + 1 + 2 + 3 + 6 = 13 (Árbol de recubrimiento de coste mínimo)

IMPLEMENTACIÓN

1. Listas de adyacencias : Mediante una serie de listas encadenadas: Una lista principal con los nodos o vértices del grafo y otra con cada las aristas que salen de cada nodo.



2. Matrices de adyacencias: Mediante una tabla de N x N de valores lógicos, siendo N es número de nodos que tiene el grafo. *Si el grafo es no dirigido, como en este caso la matriz será simétrica.*

	A	**B**	**C**	**D**	**E**	**F**
A		X	X			
B	X		X		X	
C	X	X		X		
D			X		X	
E		X		X		X
F					X	

3. Estructuras autoreferenciadas con una tabla o array de punteros.