La memoria dinámica:

Hemos mencionado antes que la memoria principal utilizada durante la ejecución de un programa tiene cuatro segmentos: el segmento de código, en donde está el programa en ejecución, el segmento de datos, donde están los datos, estáticos‟ del programa, el segmento de pila, que se utiliza para la ejecución de funciones llamadas, por ejemplo desde main, y el segmento ‟extendido‟ o memoria ‘‘heap‟, ‘‘montículo‟ o ‘‘montón‟. De éste último hablaremos ahora.

De la zona de memoria llamada ‘‘heap‟ se puede tomar tanto como sea necesario para almacenar datos, los cuales pueden ser de cualquier tipo simple o estructurado.
Esto se lleva a cabo con ayuda de punteros; cuando el espacio usado para el almacenamiento de datos ya no es más necesario, se ‘‘libera‟ ese espacio, el cual queda disponible para que sea usado nuevamente.

¿Cómo se solicita memoria dinámica?

Consideremos tener un puntero a entero p,

```
int *p;
```
la ejecución de esta sentencia:

```
p=(int *) malloc (sizeof(int));
```

solicita memoria del heap de tamaño suficiente como para qu ese almacene un entero.
Ëste espacio de memoria se manipulará desde p.

Por ejemplo, la sentencia
```
*p=3;
```
almacena 3 en el espacio de memoria generado, y cuya dirección está en p.

El manejo de la „memoria dinámica‟ permite extender el espacio que el programa utiliza para datos, de la manera más eficiente.

Luego de usada la memoria, la sentencia

```
free(p);
```

libera la memoria del heap cuya dirección estaba almacenada en p.

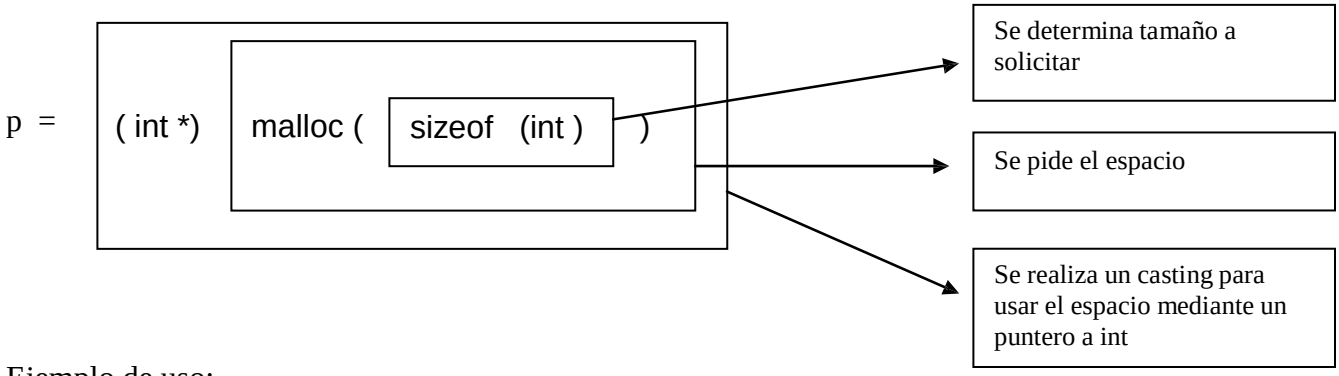
¿Qué hace exactamente malloc?

malloc es una de las varias funciones que el C tiene para habilitar el uso de la memoria dinámica.

malloc solicita memoria dinámica de un tamaño *en bytes* indicado en el argumento. Por eso, hemos indicado como argumento, sizeof(int), ya que sizeof es una función de C que retorna el número de bytes que ocupa el tipo pasado por argumento.

malloc retorna, si no hay espacio de memoria disponible, un cero, que corresponde a un puntero nulo (corresponde a la constante NULL). En el caso de que la solicitud de memoria termine de manera exitosa, se retorna la dirección de comienzo de la zona de memoria otorgada, dirección que será almacenada en la variable puntero colocada a izquierda. Esta dirección se retorna como puntero void, por lo cual se debe realizar un "casting" o moldeo al tipo correspondiente al puntero.

Es decir:



Ejemplo de uso:

```
int *p, *q;

p= (int *) malloc (sizeof(int));
*p= 5;

//si se ejecuta
q=p;

//tendremos a p y a q apuntando a la misma dirección
q= 2;

// al ejecutarse las siguientes líneas comprobamos que p y q apuntan a la misma dirección

printf("p apunta a la dirección %d, que contiene %d ", p, *p);
printf("q apunta a la dirección %d, que contiene %d ", q, *q);

//si ahora hacemos

q=(int *) malloc (sizeof(int));
*q=8;
```

```
// p y q apuntaràn a distintas direcciones; una contiene 5 y la otra 8
//Comprobamos que son direcciones diferentes, y cuál es el contenido, ejecutando
//nuevamente estas líneas

printf(“p apunta a la dirección %d, que contiene %d ”, p, *p);
printf(“q apunta a la dirección %d, que contiene %d ”, q, *q);

//Para liberar la memoria usada por p, ejecutamos

free(p);

//el haber liberado a p hace que ya no podamos utilizar la expresión *p
//observar que free no „limpia‟ el contenido de p, la variable continúa conteniendo una
dirección de memoria, que ahora es no válida, ya que ha sido „liberada‟
```

Para colocar un valor nulo en p, hacemos

```
p=NULL;
```

o bien

```
p=0;
```

que es una expresión análoga a la anterior.

Problemas de pérdida de memoria:

Es muy importante liberar la memoria dinámica luego de que se ha utilizado. Hay que tener en cuenta que la no liberación puede llevar a perder la posibilidad de usar ciertas zonas. Por ejemplo, en el siguiente código:

```
int main ()
{
 int a;

printf(“Ingrese un entero; si es positivo se solicitará espacio de memoria dinámica para un
entero”)
scanf(“%d”, &a);
if(a>0)
{
int  *p;
p= (int *) malloc (sizeof(int));
//guardamos un valor en* p
*p=1;
} //(1)

// (1) Al cerrarse el bloque correspondiente a  p, no se puede  liberar la memoria dinámica
solicitada mediante esa variable
```

//La memoria solicitada mediante p no fue liberada y ha quedado inaccesible.
//Nos ha quedado una ‟laguna de memoria‟, hemos “perdido memoria‟
}

La línea que falta antes de que se cierre el bloque en el que p está definida es
free(p);

Solicitud de memoria para un array:

Si en lugar de requerir espacio para un entero necesitamos espacio para, por ejemplo un array de 20 enteros, efectuamos:

p= (int *)malloc (20 * sizeof (int));

Luego, podemos referirnos a cada una de las 20 componentes del array mediante
p[0], p[1], etc
o bien mediante
*p, *(p+1), etc

Es decir, se puede manipular un array generado en memoria dinámica, o “array dinámico‟, tal como lo hacíamos con un array “estático‟.
Luego de usado, liberamos el espacio d ememoria que usó el array mediante:
free [ ] p;

Ejercicio:

Ingresar un entero positivo N desde teclado, y luego de validado el número, generar un array de N reales, cargar el array mediante una función adecuada, ordenar el vector y emitirlo ordenado.

Posible esquema de solución:

Usaremos con una matriz dinámica, de F filas y C columnas.

Declaramos una varible M como puntero a puntero a reales.

float **M;

Generamos espacio para el array de punteros, con F posiciones, siendo F un valor entero ingresado previamente.

M= (float **) malloc (N * (sizeof (float *)));

Luego generamos cada fila de la matriz. Podemos hacerlo con un ciclo de F iteraciones, que en cada una de ellas ejecute:

M [i] = (float*) malloc (C * (sizeof (float));

Luego, cargamos la matriz con valores, tal como lo hacíamos con una matriz estática, y emitirla por pantalla.

Finalmente, hay que liberar el espacio de memoria dinámica ocupado por cada fila, ejecutando un ciclo de tantas ieraciones como filas tiene la matriz, que contenga como sentencia free [ ] M [i] ;
y se libera el espacio usado por el array dinámico de punteros mediante
free [ ] M ;

Estructura de lista ligada:

Vamos a considerar una declaración como la siguiente:

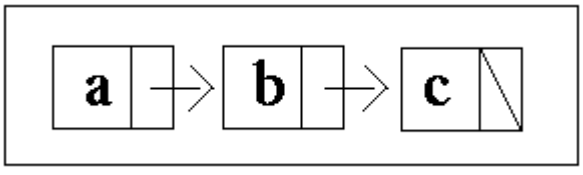
```
struct  nodo{
            char c;          //campo de  información, en este ejemplo un carácter
            nodo * sig;    //campo conteniendo el puntero, o enlace al siguiente nodo
          }
```

Un "nodo" de este tipo nos permite "encadenar" instancias de nodo, de tal modo de poder construir una estructura que, si reside en memoria dinámica, puede aumentar y disminuir su tamaño de acuerdo con las necesidades de la aplicación.

A una estructura con estas características se la llama estructura de lista ligada. Los nodos forman una secuencia y están enlazados por punteros.

En las listas ligadas cuando los nodos de la misma están distribuidos de modo aleatorio en la memoria dinámica (aquella que se gestiona en tiempo de ejecución) teniendo cada uno de ellos la dirección del siguiente nodo de la lista.

El gráfico muestra la organización de una lista ligada simple:



Siendo **a**, **b** y **c** datos almacenados en los nodos. Las flechas representan punteros no nulos; la diagonal del último nodo indica un puntero nulo.

Es posible también definir esta estructura de manera recursiva, del siguiente modo:
Una estructura de lista ligada o bien está vacía, siendo en ese caso nulo el "puntero de entrada" o puntero al primero de los nodos de la lista ligada, o bien está formada por un

nodo que contiene al menos dos campos: un campo de información y un campo conteniendo la dirección de una lista.

Una posible declaración de la estructura del nodo es :

```
struct  nodo{
            char c;          //campo de  información, en este ejemplo un carácter
            nodo * sig;    //campo conteniendo el puntero, o enlace al siguiente nodo
        }
```

Para manipular una lista con estas características utilizamos la dirección del primer nodo, la cual se almacena en una variable tipo puntero.
En una lista tal se reconoce el final porque en el último nodo el campo siguiente tiene un puntero nulo. Si el puntero de entrada estuviera nulo, esto indicaría que la lista está vacía.

Las operaciones básicas de las listas son aquellas que permiten:

- Crear una lista
- Recorrer una lista
- Determinar si una lista está vacía
- Insertar nodos en una lista (puede ser al principio, al final, en orden, etc.)
- Determinar si un dato está o no en una lista
- Borrar nodos de una lista
- Eliminar una lista

Nota: en los códigos que se desarrollan a continuación, en algunos casos se ha desarrollado código iterativo y recursivo para la implementación de los algoritmos. En general los códigos recursivos tienen menos líneas y un desarrollo más elegante que los iterativos correspondientes, pero en muchos casos implican también un mayor consumo de memoria durante su ejecución (a menos que se trate de "recursividad de cola"). Es importante recordar que todo código recursivo puede reescribirse iterativamente y viceversa (lo cual se demuestra en uno de los teoremas fundamentales de la Ciencia de la Computación).

Creación de una lista.

Crear una lista consistirá en inicializar el puntero de entrada, es decir poner a NULL el puntero al primer nodo, tal como lo hace el siguiente código:

```
 void crea(struct nodo ** p )   //recibe la direccion de la variable puntero para modificar el
                                //contenido  del mismo
{
*p = NULL;
}
```

Recorrer una lista:

Consideraremos las versiones iterativa, y recursiva.

Versión iterativa del recorrido que emite el contenido de cada nodo :

```
void recorrer ( struct  nodo *p )
{
while ( p )
 //o bien   while (p != NULL)
 //este ciclo se ejecuta mientras no sea nulo el puntero p
  {
   printf ("%c\n", p->car);
   p = p->sig;
 // se reasigna el puntero p, con la dirección del campo siguiente del  nodo actual,
 //produciendo el  avance en la lista
  }
}
```

Versión recursiva del recorrido que emite los contenidos de los nodos:

```
void recorrer( struct lista *p )
{
  if (p ) //mientras no sea nulo p
  {
    printf("%c, ",p->car);
    recorrer( p->sig );  // invocación recursiva
  }
}
```

Determinar si una lista esta vacía

El código analiza el estado del puntero de entrada y retorna un entero según el estado del mismo.

```
int vacia (struct nodo * p)
{
   return (p = = NULL);   //  retorna 1 si p es nulo y 0 si no lo es
}
```

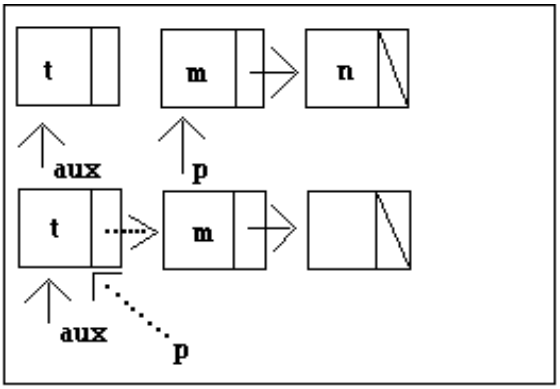
Inserción de elementos en una lista

Consideraremos entonces estos casos de inserción en una lista:
a) inserción al principio de la lista
b) inserción al final de la lista, con sus versiones iterativa y recursiva
c) inserción en orden

a) Inserción al principio de la lista

Se trata de incorporar a la lista un nuevo nodo, el cual contendrá en su campo de enlace la dirección del anterior primer nodo de la lista. En consecuencia, se modifica el contenido del puntero de entrada a la lista.

El gráfico siguiente ilustra el algoritmo:



El nodo apuntado por aux contiene el dato „t" y se insertara al principio de la lista.

Para ello, se asignara al campo sig del nodo apuntado por aux el valor de p, direccion actual de comienzo de la lista,y posteriormente se reasignara p con la direccion contenida en aux; de este modo, el primer nodo de la lista sera el que contenga a „t". Se han indicado las reasignaciones con línea de puntos.

```
void altapri (struct nodo **pp , char c)
    {
     struct nodo * aux;  // puntero auxiliar para el nodo nuevo
     aux = (nodo *) malloc (sizeof (nodo));   //se solicito espacio para el nuevo nodo
     if (aux)    //si se ha podido crear el nodo nuevo
       {
         aux->car = c ;          //se almacena el dato en el campo de información
         aux ->sig = *pp ;       //el campo siguiente recibe la dirección del primer nodo
         *pp =aux;               //se modifica el contenido del puntero de entrada
       }
  }
```

Otra versión: usando dos funciones: una crea un nodo y lo retorne con el dato a cargar ya almacenado y otra que inserta ese nodo en la lista.

```
nodo*  nodonuevo (  char c )
{
  struct nodo * nue ;
  nue = (nodo *) malloc (sizeof (nodo)) ;
  if (nue ) nue ->car = c ;
  return (nue) ;
}

void insertap ( struct nodo ** pp, char letra)
 {
  struct nodo  * aux ;
  aux = nodonuevo ( letra);
  aux->sig = *pp ;
  *pp = aux ;
}
```

b) Inserción al final de una lista

Versión iterativa

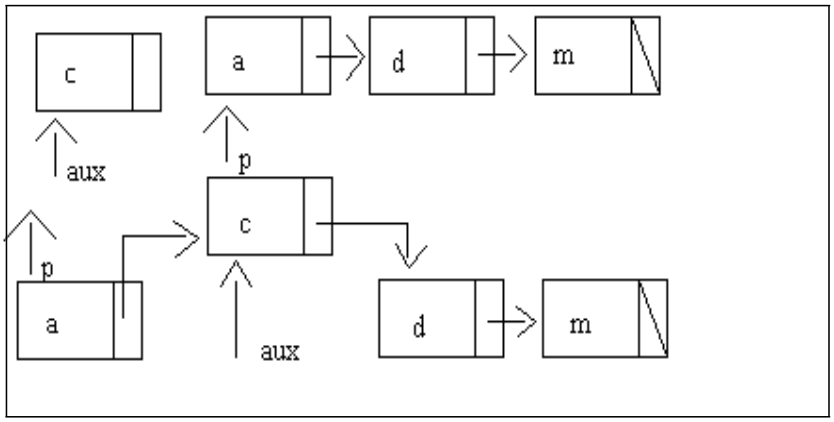
```
void insertafin (struct nodo ** pp , char letra )
{
nodo *aux;
if (*pp= =NULL)
 {
 *pp=(nodo *)malloc(sizeof (nodo));
 (*pp)->info=d;
 (*pp)->sig=NULL;
 }
else
 {
 aux=*pp;
 while (aux->sig!=NULL)
  {
   aux=aux->sig;
  }
 aux->sig= (nodo *)malloc(sizeof (nodo));
 aux=aux->sig;
 aux->info=d;
 aux->sig=NULL;
 }
```

Versión recursiva

```
void insertafine (struct nodo ** pp , char letra)
{
 if ( *pp!=NULL)       // si la lista es nula
     {
        (*pp) = (nodo *) malloc (sizeof (nodo));  //se genera el nuevo nodo
        (*pp)->car =letra;  //se almacena la letra
        (*pp)->sig=NULL;  // es pone a nulo el campo sig del ultimo nodo
      }
 else  insertafine (&((*pp)->sig)), letra);

}
```

c) Inserción en orden



Versión iterativa

```
void insertaord (struct nodo **pp, char letra)
{ struct nodo *actual, *anterior, *nuevo;
   //actual apuntara al nodo que contenga al dato siguiente a letra
   //anterior apuntara al nodo que preceda a actual
   //inicialmente anterior y actual apuntan al comienzo de la lista
  anterior = actual = *pp;
  while ((actual != NULL ) && (actual->car < letra) )
   {
     anterior = actual;
     actual = actual->sig;   //se avanzan los punteros
   }
// el bucle termina cuando se localiza la posición a insertar o se encuentra el final de la lista
   nuevo = (struct nodo *) malloc (sizeof (struct nodo));
```

```
  //genera espacio para el nuevo nodo
  if  (nuevo)   //si el nuevo nodo se ha creado con éxito
{
  nuevo->car =letra;  //se almacena el dato
  if ((anterior = = NULL) || (anterior = = actual))
  //se analiza si la lista es nula o si tiene un solo nodo
  {
    nuevo->sig = anterior;
    *pp = nuevo;
  }
 else {
        nuevo->sig = actual;
        anterior->sig = nuevo;
   //se modifican los enlaces para incorporar al nuevo nodo
      }
}
}
```

Versión recursiva de la inserción en orden

Según se observa en este código, se procede a rastrear utilizando una técnica recursiva, aquel nodo que contiene un dato mayor que el que se quiere insertar; una vez apuntado este nodo, se realiza una inserción al principio de la sublista de la cual es cabeza. De no encontrarse un valor mayor que el del dato a insertar, se ubica éste al final de la lista.

```
void insertaord (struct nodo **pp, char letra)
{
  struct nodo *aux;
  if ((**pp) && (letra > (*pp)->car))
      insertaord (&(*pp)->sig , letra );  //invocación recursiva
  else
     { aux =(nodo*) malloc (sizeof (nodo)) ;
       if (aux)   // si se ha generado el nodo
                  { aux->car =letra ;
                    aux->sig = *pp ;
                    *pp= aux;
                  }
     }
}
```

Determinar si un dato está en la lista

Se trata de analizar si en alguno de los nodos de una lista se encuentra determinado dato. Se supone en los casos considerados una lista no ordenada. La búsqueda termina exitosamente retornándose 1 cuando el dato se encuentra o 0 si el dato no está.

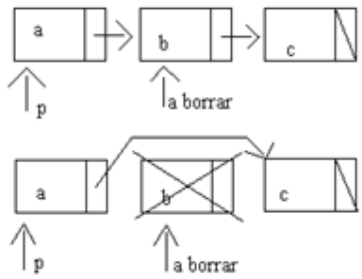
Versión iterativa

```
int esta(struct nodo *p , char ch)    //retorna 1 si el dato esta y 0 si no
{
  while (p && ( p->car != ch)) p= p->sig ;
      // mientras p no sea nulo y en el nodo apuntado no esté el dato, avanza al siguiente nodo
      // el bucle termina si el puntero es nulo o se encuentra el dato
   if (! p) return 0;  // si p es nulo, el dato no está
              return 1;  // valor retornado si el dato está
}
```

Version recursiva

```
int esta(struct nodo *p , char ch)  //retorna 1 si el dato esta y 0 si no
{
  if  (p)
     {if  ( p->car = = ch)  // analiza si el dato
               esta return 1;
       else   return ( esta (p->sig , ch));
    // retorna el resultado recibido de la invocación recursiva a la misma función
       }
  else    return 0;      //si el puntero esta nulo el dato no esta
}
```

Eliminar un nodo con un valor determinado:



Se quiere eliminar el nodo que contiene ‘b‟.
Se reasigna el puntero siguiente del nodo que precede al que contiene el dato a eliminar.
Posteriormente se libera el espacio utilizado por el nodo.

Versión iterativa:

```
void Borrar(nodo ** pp, char ch)
{ struct nodo *anterior, *aborrar;
aborrar =*pp; anterior = NULL;
while(aborrar && (aborrar->car < v))
  {
    anterior = aborrar;
    aborrar = aborrar->siguiente;
  }
  if(aborrar && (aborrar->car = = v))
   //si se encontró un nodo con el elemento buscado, se eliminará
   {
    if(!anterior)     //analiza si es el primer elemento
       *pp = aborrar->siguiente;
    else              //no es el primer elemento
       anterior->siguiente = aborrar->siguiente;
free (aborrar)
   }
}
```

Versión recursiva

```
void borra (struct nodo** pp, char ch)
{
 struct nodo *aux;
 if ( (* pp) && (*pp)->car = = ch) //si se localizó el nodo buscado
    {
       aux = *pp;
       *pp= *pp ->sig;
       free(aux);                  //libera el espacio usado por el nodo
     }
else borra (&(*pp)->sig , ch);
}
```

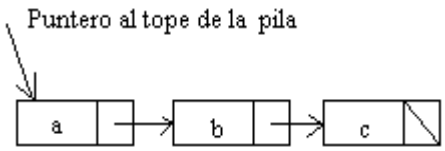
Estructura de lista ligada usada como Pila:

Si en una lista imponemos restricciones que hagan que el alta y la baja se realicen siempre por el mismo extremo, la estructura estará funcionando como una pila.

Las pilas tienen restricciones en la inserción y el borrado de elementos. Las altas y bajas deben realizarse por el mismo extremo. Son estructuras “last in, first out” (L.I.F.O.).

El último elemento en entrar, se dice que está en el tope de la pila, y el único que puede salir en una pila es también el que está en la posición del tope.
La implementación de una pila con estructuras dinámicas requiere de una lista ligada en la cual un extremo se utiliza para las altas y las bajas. Por razones de eficiencia se elige el

principio de la lista como extremo por el cual se realizaran las altas mediante la inserción de un nodo al principio y las bajas mediante el borrado del primer nodo.



Las operaciones básicas de las pilas son:

Crear una pila (idéntica a crear una lista)
Emitir el contenido del tope de la pila (emitir el contenido del primer nodo)
Determinar si la pila está vacía. (analizar si el puntero de entrada está nulo)
Agregar datos a la pila, o 'push' (equivalente al alta al principio de una lista)
Borrar el contenido del tope de la pila, o 'pop' (equivalente a borrar el primer nodo de la lista; es posible sólo si la lista no está vacía)
Determinar si la pila está llena (esta operación consiste en determinar si en la memoria dinámica hay espacio suficiente como para que un nuevo nodo sea creado).

Los códigos correspondientes a estas operaciones ya se han desarrollado para las listas, al comienzo del apunte.

Colas

Si imponemos a la estructura de la lista ligada para que por un extremo se hagan las altas y por otro las bajas, la estructura estará funcionando como cola. Las colas son estructuras "first in, first out" (F.I.F.O.).

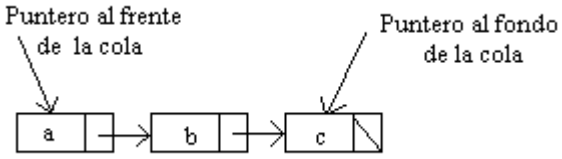
El último elemento en entrar, se dice que está en el fondo de la cola, y el único que puede salir en una cola es el que está en la posición del frente de la misma.

Las operaciones básicas de las colas son:

Crear una cola
Emitir el contenido del frente de la cola (emitir el contenido del primer nodo)
Determinar si la cola está vacía. (analizar si el puntero de entrada está nulo)
Agregar datos a la cola (equivalente al alta al final de una lista, proceso facilitado por la existencia del puntero al último nodo de la lista; posible sólo si la cola no está llena)
Dar de baja a un elemento de la cola (equivalente a borrar el primer nodo de la lista; es posible sólo si la lista no está vacía)
Determinar si la cola está llena (esta operación consiste en determinar si en la memoria dinámica hay espacio suficiente como para que un nuevo nodo sea creado)

Para implementar una cola con estructuras dinámicas se considera una lista ligada en la cual los ingresos se realizarán por un extremo y las bajas se realizarán por el otro.
Debido a que los dos extremos de una cola se requieren en las operaciones básicas, se diseñará la cola como una lista con dos punteros: uno al comienzo o "frente" de la cola, y el otro apuntando al último nodo, o "fondo" de la cola.
En el estado inicial, con la cola vacía ambos punteros estarán nulos.
Cuando la cola contenga un solo nodo, ambos punteros apuntarán a ese único nodo. Para realizar un alta con la cola no vacía, el puntero requerido es el que apunta al fondo; mediante él se generará el nuevo nodo y se "moverá" el puntero del fondo al nodo nuevo, que constituirá el nuevo fondo.
Para realizar una baja, se debe discriminar si la cola contiene un solo nodo o más: si el nodo es único, se procederá a eliminarlo, poniendo luego los punteros al frente y al fondo en NULL; en cambio si la cola contiene más de un nodo se procederá a eliminar el primer nodo de la misma, adelantando asimismo el puntero al frente al siguiente nodo.
Las operaciones más relevantes se desarrollarán a continuación.

Nota: el nodo de la cola es idéntico al de la lista ligada desarrollada anteriormente.



Creación de una cola

```
void crearcola (nodo**  frente , nodo **fondo)
{
  *frente=NULL;
   *fondo =NULL;
}
```

Alta en una cola

```
void alta (nodo ** frente , nodo ** fondo , char dato)
{
 if (*frente = NULL)        // si cola está vacía
     {
       *frente = (nodo *) malloc (sizeof (nodo));  //se genera un nodo
         if (*frente)
        {
```

```
        *fondo = *frente;                    //frente y fondo apuntan a ese único nodo
        (*frente)->car=dato;                  //se carga el dato
        (*frente)->sig =NULL;                //el campo sig queda nulo
       }
     }
else {
        ( *fondo)->sig =(nodo *) malloc (sizeof (nodo));

        if (*fondo)
          {
           * fondo=(*fondo)->sig;
              (*fondo)->car=dato;                  //se carga el dato
            (*fondo)->sig =NULL;                //el campo sig queda nulo
          }
        }
}
```

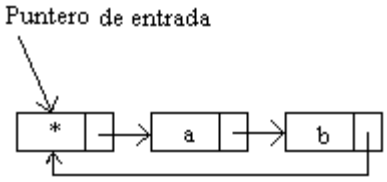
Baja en una cola

```
void bajacola (nodo **frente , nodo ** fondo)
{
  nodo *aux;
  if (*frente)                          // ¿hay nodos en la cola?
     { if (*frente = = *fondo)       // ¿hay un solo nodo en la cola?
          {
            *fondo = NULL;
        //si hay un solo nodo, frente y fondo le apuntan; ambos deben terminar en NULL
                                  //y el espacio del nodo debe ser liberado
          free( *frente);  //se libera el nodo
         *frente=NULL;
        }
       else            // si hay más de un nodo
        {
          aux = *frente;                  //auxiliar para la destrucción del primer nodo
          frente =  (*frente)->sig;      //avanza el frente
          free (aux);                     //destrucción del primer nodo
           }
       }
}
```

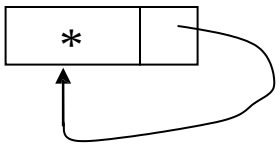
Listas circulares en C

Una lista circular ligada es un caso particular de lista ligada en el cual el último nodo en vez de tener un puntero nulo en el campo siguiente, contiene la dirección del primer nodo, es decir apunta al primer nodo.

Para reconocer el primer nodo (y, por ejemplo, no pasar "de largo" al realizar un recorrido de la lista) suele colocarse una "marca" en el primer nodo en el campo de datos. Esta marca es un valor diferente de los valores que pueden tener los datos almacenados en la lista. Por ejemplo, si la lista contuviera datos enteros positivos, la marca podría ser un 0, si almacenara caracteres o strings podría ser un "*".



Esto implica que cuando una lista construida con estas características no contenga datos, debe tener de todos modos un nodo, el de la marca. El proceso de creación de tales listas consiste en generar un nodo, asignar el valor de la marca al campo de datos y colocar la dirección del mismo nodo en su campo de enlace, es decir que tenemos un nodo con un puntero que apunta al mismo nodo.



Podemos declarar el nodo de esta forma:

```
struct nodo {
      int info;
      nodo *sig;
                  }
```

Observar que la declaración del nodo de la lista ligada circular es idéntica a la del nodo de una lista no circular. Es el modo de realizar las operaciones lo que determinará su condición de "circular"

Las operaciones básicas con las listas ligadas circulares son las mismas que se han enumerado para las listas ligadas no circulares. Se desarrollarán los códigos más relevantes.

Creación de una lista ligada circular con marca indicadora de primer nodo

```
nodo * crealistacir ()
{
   nodo * aux = (nodo*) malloc (sizeof(nodo));
```

```
  aux -> info = -1;            // -1 será la marca del primer nodo
  aux ->sig= aux;
  return aux;
}
```

Alta al principio de una lista circular con marca de primer nodo

Dado que una vez creada una lista circular la dirección de comienzo no varía (en la implementación que se ha elegido), puesto que nunca se eliminará el nodo que contenga la marca, no es necesario pasar como parámetro la dirección del puntero de entrada. Por eso se pasará sólo la dirección del primer nodo.

```
void altapliscir ( nodo * p , int dato)
{
nodo * aux = (nodo *) malloc (sizeof(nodo));
aux->info  =dato;
aux->sig=p->sig;
p->sig=aux;
}
```

Alta al final de una lista circular con marca de primer nodo

Para llevar a cabo esta operación, se debe detectar el nodo que contenga en el campo sig la dirección del que tiene la marca.
Se ha desarrollado una versión iterativa de la función correspondiente.

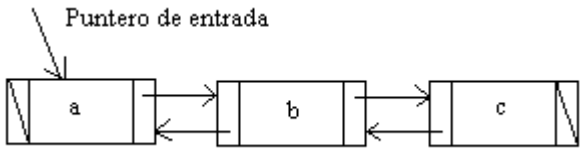
```
void altafliscir(nodo *p, int dato)
{
nodo * aux =(nodo *)malloc (sizeof(nodo));
aux->info=dato;
p=p->sig;    //avanza un nodo para recorrer los que contienen información hasta detectar el último
while (p-> info != -1) p=p->sig;
aux->sig=p;
p=aux;
}
```

Se deja como ejercicio el diseño y codificación de las restantes operaciones.

Listas doblemente ligadas

Se trata de listas en las que cada nodo contiene dos campos puntero: siguiente y anterior.

El campo siguiente de cada nodo (excepto el último nodo) contiene la dirección del siguiente nodo.
El campo anterior de cada nodo (excepto el primer nodo) contiene la dirección del nodo anterior.
El campo anterior del primer nodo y el campo siguiente del último nodo contienen NULL.
El doble enlace permite recorrer la lista en ambos sentidos.



La declaración del nodo podría ser

```
struct  nodolisd
                {
                  int valor;
                  struct  nodolisd  *siguiente, *anterior ;
                   };
```

Cuando la lista doble esté vacía, el puntero de entrada estará nulo.
Cuando contenga un solo nodo, los campos anterior y siguiente del mismo estarán en NULL.

Las operaciones básicas con listas dobles son las habituales en listas. Como en los casos anteriores, sólo se desarrollaran las más relevantes.

Alta al principio en lista ligada doble

Versión iterativa

```
void altaplisd (nodo ** pp, int dato)
 {
   nodo *aux ;                              // puntero para generar el nodo nuevo
   aux = (nodo *)malloc(sizeof(nodo));
   nuevo->info = dato;
   nuevo->ant=NULL;
   nuevo->sig=*pp;
   (*pp)->ant=aux;
   *pp=aux;
}
```

Alta al final en lista doblemente ligada

Versión iterativa

```
void altaplisd (nodo ** pp, int dato)
 {
  nodo *aux, *aux2;                              // puntero para generar el nodo nuevo
  aux = (nodo *)malloc(sizeof(nodo));
  nuevo->info = dato;
  nuevo->sig=NULL;
  if (! (*pp))
            {
             *pp= aux;
             (*pp)->ant=NULL;
              }
   else
            {
              aux2 = *pp;          //con aux2 se reconocerá el último nodo
              while (aux2->sig != NULL) aux2 = aux2->sig;
                                   // al finalizar el ciclo , aux2 apuntará al último nodo
              aux2->sig = aux;  //se enlaza a continuación el nuevo nodo
              aux->ant = aux2;  //se asigna al campo anterior del último nodo la dirección de
aux2
            }
}
```

Borrado en una lista ligada doble

Versión iterativa

```
 void Borrar(nodo **pp, int dato)
{
  nodo *aux , *auxant ;
  aux = *pp;
  while(aux && aux->info!=dato) aux=aux->sig;  //avanza hasta apuntar al nodo con el
dato
  if  (aux)          //si el valor buscado está en la lista
      {
        auxant = aux->ant;       //apunta al nodo anterior al que contiene el dato
        if ( ! auxant)               // si está en el primer nodo
            {
             *pp = (*pp) ->sig;
              (*pp)->ant = NULL;
              free (aux);
            }
        else if (! aux->sig )       //si está en el último nodo
            {
              auxant ->sig =NULL;
```

```
                free(aux) ;
              }
         else                        //si está en otro nodo
              {
                auxant ->sig = aux ->sig;
                aux->sig->ant = auxant;
                free (aux);
              }
     }
}
```

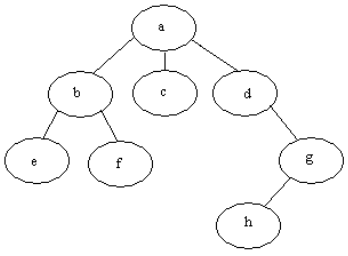
Se deja como ejercicio el diseño e implementación de las restantes operaciones.

Arboles

Una estructura arborescente o árbol es una estructura jerárquica definida sobre un conjunto de elementos llamados nodos; una determinada relación impone la jerarquía; uno de los nodos del árbol se distingue por ser la raíz.
Estas estructuras pueden definirse recursivamente del siguiente modo:
Un árbol es una estructura vacía o bien un conjunto finito de uno o más nodos, en el cual un nodo es llamado raíz y los restantes nodos están separados en conjuntos disjuntos en número mayor o igual que 0 cada uno de los cuales es a su vez un árbol (subárboles del nodo raíz).

El gráfico que aparece a continuación muestra un árbol genérico; el nodo raíz contiene el dato "a", sus tres nodos hijos contienen "b", "c" y "d" respectivamente.
Los nodos que contienen los datos "e", "f" y "h" son hojas por no tener hijos.

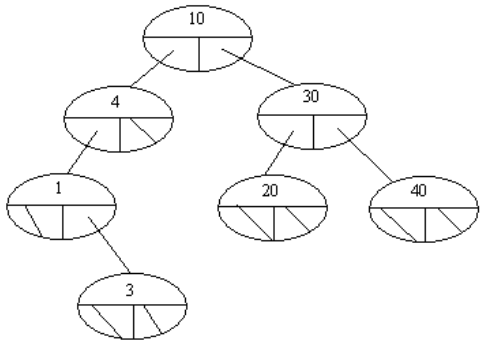
Arboles binarios
Si cada nodo de un árbol tiene dos hijos (pudiendo ser uno de ellos o ambos nulos), dicho árbol es binario. Estos árboles suelen implementarse mediante estructuras dinámicas compuestas por nodos en los cuales hay al menos un campo de datos y dos punteros a los nodos hijos.



Arboles binarios de búsqueda

Dentro de los árboles binarios interesan particularmente aquellos en los cuales los datos no están repetidos y están ordenados, puesto que facilitan las búsquedas de datos en el conjunto. En estos árboles, se verifica para cada nodo que los datos almacenados en el subárbol izquierdo son menores que el dato del nodo y los datos almacenados en el subárbol derecho son mayores que el dato del nodo.

A continuación, se muestra un gráfico de un árbol binario de búsqueda:



Consideremos un árbol que almacena en cada nodo un dato entero. La declaración del struct para el nodo puede ser tal como se indica a continuación:

```
struct nodoarbol
                { int dato;
                  nodoarbol *izq, *der ;
                 };
```

Operaciones propias de los árboles binarios:
Creación
Destrucción
Alta
Recorridos: inorden, preorden, postorden
Baja de un elemento particular
Determinación de si un dato está o no en el árbol
Determinación de si un árbol está vacío

Alta en un árbol binario de búsqueda

Por razones de simplicidad de código, se ha decidido desarrollar un código recursivo.

```
void alta (nodoarbol **pp , int dato)
{
 if (! *pp)   //si el puntero es nulo, se genera el nodo y se almacena el dato
            {
               *pp= (nodoarbol *) malloc (sizeof (nodoarbol ));
                (*pp)->info = dato;
                 (*pp)->izq = NULL;
                 (*pp)->der = NULL;
             }
 else          // si el puntero no es nulo
     if (dato <(*pp)->info)   // si el dato a agregar no es mayor que el del nodo
                                  alta ( &((*pp)->izq) , dato);
     else  if    (dato > (*pp)->info)                          // si el dato es mayor que el del nodo
                                  alta ( & ((*pp)->der), dato);
}
```

Recorridos en un árbol binario:

Un ABB puede recorrerse en profundidad o en anchura. Los recorridos en profundidad tienen tres formas básicas:

Inorden: para cada nodo se procede a
Recorrer primero el subárbol izquierdo,
Luego se visita el nodo
Y luego recorrer el subárbol derecho

Preorden: para cada nodo se procede a
Visitar primero el nodo,
Recorrer luego el subárbol izquierdo,
Y luego recorrer el subárbol derecho

Postorden: para cada nodo se procede a
Recorrer primero el subárbol izquierdo,
Recorrer luego el subárbol derecho
Y luego visitar el nodo

Código del recorrido inorden emitiendo el contenido de los nodos:

```
void  inorden (nodoarbol * p)
{
   if (p)           // si el puntero no es nulo
          { inorden (p->izq);      //invocación recursiva para bajar por izquierda
            printf ( "%d", p->info);
            inorden (p->der);     //invocación recursiva para bajar por derecha
}
```

Código del recorrido en preorden emitiendo el contenido de los nodos:

```
void  preorden (nodoarbol * p)
{
   if (p)          // si el puntero no es nulo
          {
             printf ( "%d", p->info);
             preorden (p->izq);      //invocación recursiva para bajar por izquierda
             preorden (p->der);     //invocación recursiva para bajar por derecha
          }
}
```

Código del recorrido en postorden emitiendo el contenido de los nodos:

```
void  postorden (nodoarbol * p)
{
   if (p)          // si el puntero no es nulo
          {
             postorden (p->izq);      //invocación recursiva para bajar por izquierda
             postorden (p->der);     //invocación recursiva para bajar por derecha
             printf ( "%d", p->info);
          }
}
```

Nota:

Los códigos correspondientes a los recorridos anteriores se pueden reescribir utilizando estructuras de ciclo y una pila (la cual permitirá almacenar las sucesivas posiciones en el descenso a las hojas del árbol). Corresponden todos ellos a un recorrido en profundidad de la estructura.

Si se quiere recorrer un árbol por niveles (es decir, pasando primero por la raíz, luego por los nodos del nivel 1, luego por todos los del nivel 2 y así sucesivamente), se debe usar una estructura cola , y realizar un recorrido en amplitud.

Baja de un elemento particular en un árbol binario de búsqueda

Para que la baja de un elemento del árbol sea eficiente, se procede del siguiente modo:

Una vez apuntado el nodo conteniendo el dato a eliminar, si éste es una hoja, se libera el espacio utilizado por la misma, colocándose en NULL el campo que le apuntaba desde su nodo padre (a menos que se tratara de la raíz, en cuyo caso el árbol quedaría vacío)

Si no se trata de una hoja, se reemplaza el contenido de dicho nodo por el del mayor de los hijos menores, o por el del menor de los hijos mayores, dado que esto permite que se mantengan ordenados los datos del árbol.
Si el reemplazo se realizó con el mayor de los menores, éste nodo sólo puede tener un hijo a derecha. El padre del nodo usado para el reemplazo deberá enlazarse con el hijo derecho del reemplazante.
Si se utilizó el menor de los mayores, éste nodo sólo puede tener un hijo a izquierda. El padre del reemplazante deberá enlazarse con él adecuadamente. La implementación de la baja se deja como ejercicio.

Búsqueda de un dato en un árbol binario de búsqueda:

La búsqueda de un dato en un árbol binario de búsqueda es eficiente si el árbol está suficientemente balanceado, es decir si los nodos se distribuyen de modo relativamente uniforme ocupando la disponibilidad de los niveles.

```
int esta (nodoarbol *p , int dato)
{
 if (p)            // si el puntero no es nulo
   {
     if (p->info = = dato) return 1;   // dato encontrado : retorna 1
     else
       if (dato < p->info)              //si el dato es menor, continúa buscando por izquierda
            {return (esta (p->izq , dato) );}
       else
                 //si el dato es mayor, continúa buscando por derecha
            {return(esta (p->der , dato));}
        }
  else   return 0;                      //dato no encontrado : retorna 0
}
```

Quedan como ejercicios el diseño e implementación de las restantes operaciones básicas

Recorrido en amplitud de un árbol binario

Este recorrido se lleva a cabo visitando, en orden, los nodos de cada nivel del árbol.
Primero se visitan los nodos de nivel 0 (raíz), después los nodos de nivel 1, y así sucesivamente hasta agotar el conjunto de nodos.
En general para este recorrido se utiliza una estructura cola y un algoritmo iterativo.
El método consiste en guardar en la cola la raíz (o su dirección) y luego, mientras no se haya vaciado la cola, se elimina el primero de la cola y se almacenan los subárboles hijos

(o sus direcciones) del que se ha dado de baja de la cola y así se prosigue hasta que la cola esté vacía.

```
void amplitud (n_arbol *a)
{
  Cola c;   /* las claves de la cola serán de tipo árbol binario */
  n_arbol *aux;

   if (a != NULL) {
    CrearCola(&c);
    Altacola(&c, a);
    while (!Colavacia(c)
    {
      Bajacola(&c, aux);
      Emitir(aux);
      if (aux->izq != NULL) Altacola(&c, aux->izq);
      if (aux->der != NULL) Altacola(&c, aux->der);
    }
  }
}
```