Estructuras de datos lineales

Estructuras estáticas

- Básicamente, cuando hablamos de “estructura” nos referimos a “colección”. De hecho, la palabra *struct* de C nos permite definir un tipo de datos nuevo con base en una colección de otros tipos de datos ya existentes.

- Una estructura de datos estática es una colección cuya capacidad máxima debe definirse previamente.

- Por ejemplo, al declarar una variable de tipo Persona[10] (un arreglo con capacidad para contener hasta 10 personas) el programa reservará una cantidad de memoria fija más allá de que durante su ejecución la utilice o no.

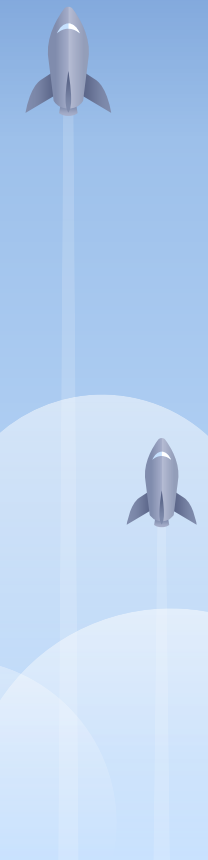
```
typedef  struct Persona
{
      int dni;                //2 bytes
      char nombre[20];        //20 bytes
      long fechaNac;          //4 bytes
}Persona;
```

Cada registro Persona utiliza 26 bytes.

Así un arreglo de 10 Personas ocupará 260 bytes de memoria.

Estructuras dinámicas

- El concepto de “estructura dinámica” también se refiere a una colección de valores del mismo tipo.
- La diferencia esta dada en que la cantidad de elementos de la colección puede variar durante la ejecución del programa aumentando o disminuyendo y, en consecuencia, utilizando mayor o menor cantidad de memoria.



El nodo

- Las estructuras dinámicas se forman “enlazando” nodos.
- Un nodo representa un conjunto de uno o más valores, más un puntero haciendo referencia al siguiente nodo de la colección.
- Veamos como definir un nodo en C.

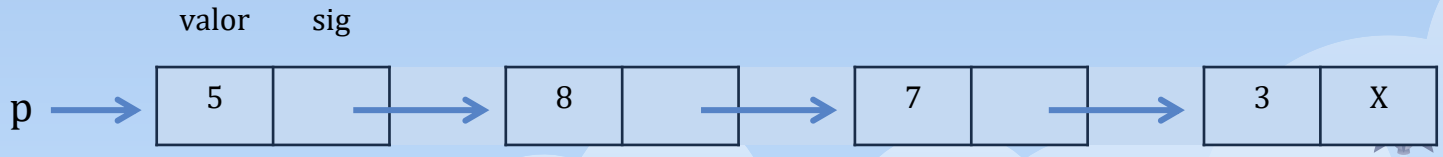
```
typedef struct Nodo
{
    int valor;
    struct Nodo* sig;
}Nodo;
```

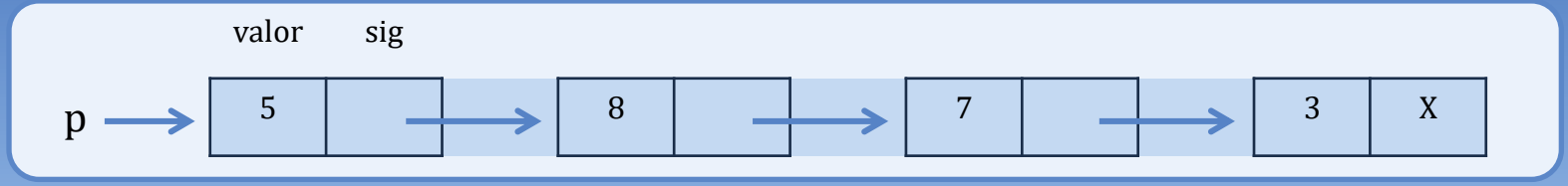
Un nodo simplemente es una estructura que define valores más una referencia (puntero de tipo Nodo*) para apuntar al siguiente nodo de la colección.

Listas

Listas enlazadas

- Con la estructura ***Nodo*** ya definida y analizada, fácilmente podemos visualizar una lista enlazada de nodos en la que cada nodo contiene un valor y una referencia al siguiente elemento de la colección.





- En la figura vemos una lista formada por un nodo con valor de 5, cuya referencia apunta a un nodo con valor de 8, cuya referencia apunta a un nodo con valor de 7, cuya referencia apunta a un nodo con valor de 3, cuya referencia tiene un valor nulo por tratarse del último nodo de la lista o último elemento de la colección.
- También podemos ver una variable ***p*** que apunta al primer nodo.
- Es decir, ***p*** debe de ser una variable de tipo ***Nodo**** de forma tal que pueda contener la dirección del primer elemento de la lista.
- Como el último nodo no tiene "elemento siguiente", el valor de su referencia al siguiente nodo debe de ser NULL.

Estructuras de datos dinámicas lineales

- Una lista enlazada formada por una colección de nodos, constituye una estructura de datos dinámica lineal.
- La linealidad de la estructura se refiere al hecho de que cada elemento de la colección tiene un único elemento anterior y un único elemento posterior salvo, obviamente, los casos particulares del primer y último elemento.

Punteros por referencia

- Si necesitamos hacer que una función f (o cualquier otra función) pueda modificar el contenido de sus parámetros entonces tendremos que trabajar con sus direcciones de memoria.

```
void f(int * x)
{
    *x = 3;
}
```

```
int main()
{
    int a=10;
    f(&a);
    printf("%d\n", a);
}
```

- Ahora tendremos que analizar que sucederá si necesitamos que una función modifique el valor de un parámetro de tipo puntero.
- Por ejemplo, la función *asignarMemoria* recibe un puntero a entero *(int*)* al que le asigna la dirección de un espacio de memoria recientemente “asignado”.

Incorrecto	**Correcto**
void asignarMemoria(int * p) { p = (int*)malloc(sizeof(int)); }	void asignarMemoria(int ** p) { *p = (int*)malloc(sizeof(int)); }

- En el **primer** caso (Incorrecto), la función recibe un puntero a entero, esto es: la referencia a un valor entero pero no una copia de su dirección de memoria. Podemos modificar el valor referenciado por **p** pero no podemos modificar su propio valor (la dirección que contiene).

- En el **segundo** caso (correcto), recibimos un "puntero a entero". Esto es una referencia a la dirección de un valor entero o, en otras palabras, un puntero por referencia. Luego, dentro de la función, ***p** representa el contenido de **p** que simplemente es la dirección de un valor de tipo *int*.

Operaciones sobre listas enlazadas

- Se definirán un conjunto de operaciones que nos facilitarán la manipulación de los elementos de las listas.

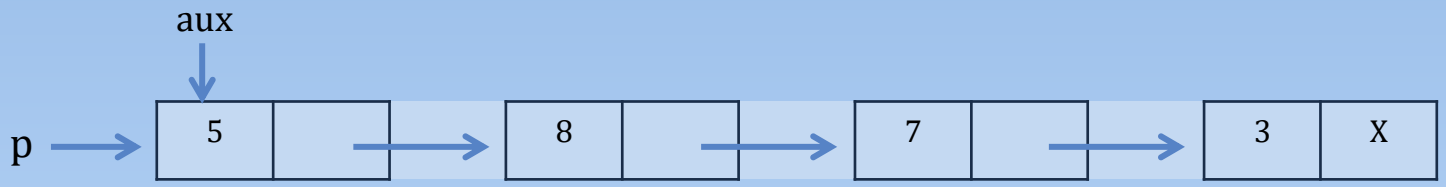
- Se trabajará sobre listas de enteros cuyos nodos respetan la estructura *Nodo*.

Agregar un elemento nuevo al final de una lista

- Analizaremos un algoritmo para agregar un elemento nuevo (nodo) al final de una lista enlazada.

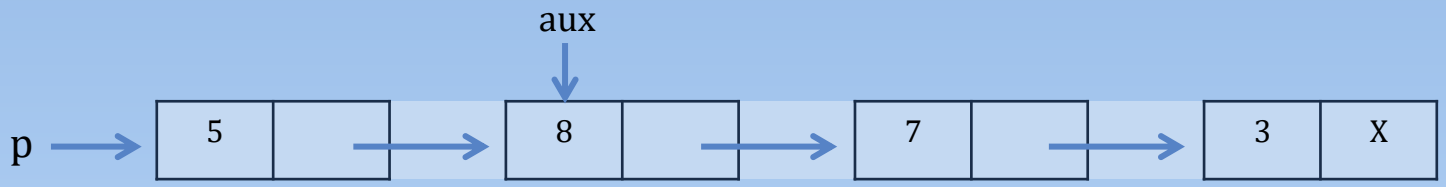
- La idea es recorrer la lista avanzando sobre cada uno de sus nodos hasta llegar al último, que es fácilmente identificable por tener el valor nulo NULL en su referencia al siguiente nodo.

- Para esto, utilizaremos una variable auxiliar ***aux*** y le asignaremos como valor inicial la dirección contenida por **p**.
- Luego ***aux*** apuntará al primer nodo de la lista.



- Para saber si ***aux*** apunta al último nodo de la lista simplemente preguntamos si ***aux->sig*** es NULL.
- Según el gráfico, ***aux->sig*** (el siguiente de ***aux***) no es NULL ya que tiene la dirección del nodo con valor 8 (el segundo elemento).

- El próximo paso será hacer que ***aux*** apunte al siguiente nodo. Esto lo logramos asignando a ***aux*** la dirección de su propio campo ***sig*** haciendo ***aux = aux->sig***.

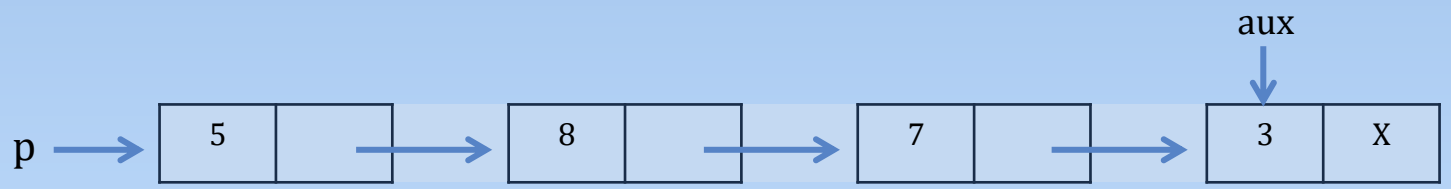


- En realidad, este proceso lo haremos dentro de un ciclo de repeticiones que itere mientras que "el siguiente de ***aux***" sea distinto de NULL.

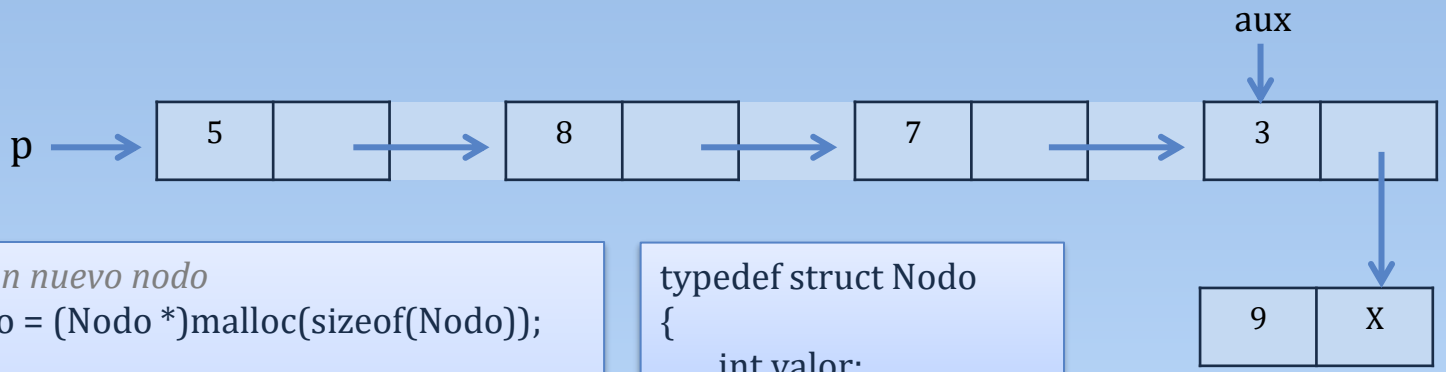
```
aux = p;
while( aux->sig!=NULL )
{
    aux = aux->sig;
}
```

- Como podemos ver, ***aux*** comienza apuntando al primer nodo de la lista y luego de cada iteración apuntará al siguiente.

- La condición del *while* se dejará de cumplir cuando ***aux*** apunte a un nodo sin elemento siguiente que, según nuestro ejemplo, es el nodo con valor 3.



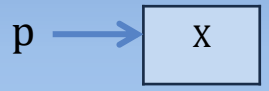
- El próximo paso será crear un nuevo nodo y “enlazarlo” al final. esto es convertirlo en “el siguiente” del último nodo de la lista que, en este momento, esta siendo apuntado por ***aux***.



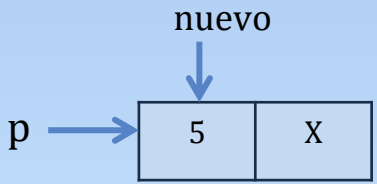

```
//creamos un nuevo nodo
Nodo* nuevo = (Nodo *)malloc(sizeof(Nodo));

//Asignamos su valor y NULL en su siguiente
nuevo->valor = 9;
nuevo->sig = NULL;

//lo enlazamos como siguiente de aux
aux->sig = nuevo;
```

```
typedef struct Nodo
{
    int valor;
    struct Nodo* sig;
}Nodo;
```

- Este algoritmo funciona perfectamente si la lista ya tiene al menos un elemento.
- Analicemos ahora que sucederá si la lista sobre la que vamos a agregar un nuevo nodo aún está vacía.



- En la figura vemos que **p** (el puntero al primer nodo de la lista) tiene la dirección nula NULL, lo que indica que la lista está vacía.
- En este caso, luego de crear el nuevo nodo debemos hacer que **p** lo apunte. Para esto, tenemos que asignarle a **p** la dirección memoria del nuevo nodo ya que, este también será el primero.



- Como este algoritmo se va a encapsular dentro de una función, resulta que si **p** es NULL tendremos que modificar su valor para asignarle la dirección de memoria del nuevo nodo.

- Es decir, la función recibirá a **p** por referencia y, dado que **p** es de tipo **Nodo***, entonces debemos tratarlo como un valor de tipo Nodo**.

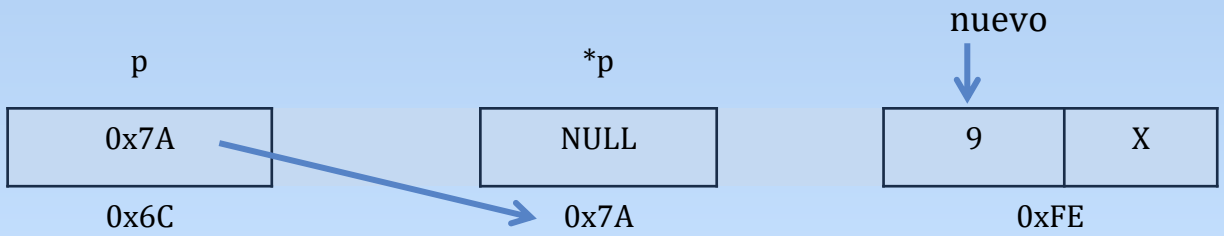
- Veamos el desarrollo de la función ***agregar*** que recibe por referencia el puntero al primer nodo de la lista más un valor de tipo *int* para agregarlo como último elemento.

- Lo primero que hacemos en ***agregar*** es crear el nuevo nodo y asignarle el valor **x** que recibimos como parámetro y NULL como referencia al siguiente.
- Luego entramos a un **if** para detectar el caso particular en el que la lista esta vacía.
- Si entramos por la parte derecha del **if** será porque la lista ya tiene, al menos, un elemento. Entonces la recorremos hasta llegar al último nodo que, como observamos más arriba, podemos identificar porque su puntero al siguiente es NULL.
- Una vez que encontramos el último nodo (apuntado por ***aux***) simplemente asignamos a **aux->sig** la dirección del nuevo.
- Con esto, el nuevo nodo pasará a ser el último elemento de la lista.

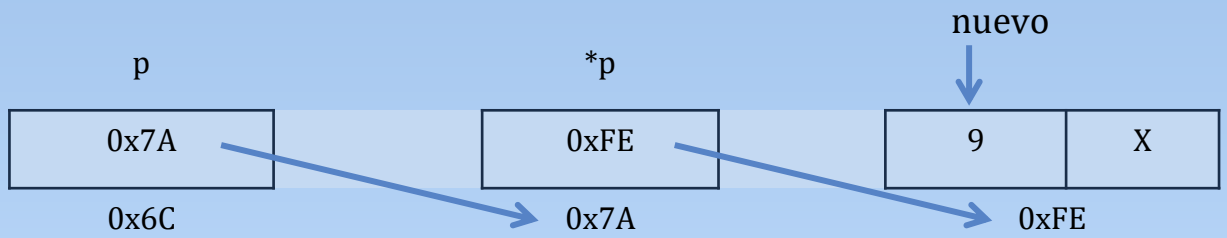
- Analicemos ahora el caso en el que p es NULL. En este caso, entraremos por la parte izquierda del ***if***, donde tenemos que hacer que el nuevo nodo sea también el primero asignado a ***p*** la dirección que contiene nuevo.

- Hacer que **p** apunte al nuevo nodo implica modificar su valor. Recordemos que dentro de la función ***agregar***, **p** es la dirección del puntero que apunta al primer nodo de la lista.

- Entonces, **p*** es el puntero al primer nodo de la lista.



- En la figura vemos que la variable **p** contiene la dirección de un espacio de memoria en el que actualmente se aloja el valor nulo NULL.
- Tenemos acceso a este espacio a través de ***p**.
- Luego para colocar el nuevo nodo como primer elemento de la lista tenemos que hacer que ***p** deje de tener NULL y pase a tener la dirección del nuevo elemento que, según la figura, es 0xFE.



- Luego de la asignación ***p = nuevo** el nuevo nodo pasará a ser el primer y único elemento de la lista.

```
void agregar(Nodo** p, int v)
{
        // creamos el nuevo nodo
        Nodo *nuevo = (Nodo*) malloc(sizeof(Nodo));
        nuevo->valor = v;
        nuevo->sig = NULL;
        // si la lista esta vacia entonces hacemos que p apunte al nuevo nodo
        if( *p == NULL )
        {
                *p = nuevo;
        }
        else
        {
                Nodo* aux = *p;
                // recorremos la lista hasta llegar al ultimo nodo
                while( aux->sig != NULL )
                {
                        // avanzamos a aux al proximo nodo
                        aux = aux->sig;
                }
                // como aux apunta al ultimo entonces su siguiente sera el nuevo nodo
                aux->sig = nuevo;
        }
}
```

```
int main()
{
        // inicializamos la lista
        Nodo* p = NULL;

        // le agregamos valores a través de la función agregar
        agregar(&p, 5);
        agregar(&p, 8);
        agregar(&p, 7);
        agregar(&p, 3);
        agregar(&p, 9);

        // mostramos por pantalla el contenido de lista
        // la función mostrar la analizaremos a continuación
        mostrar(p);

        // antes de finalizar el programa liberamos la memoria
        // que ocupan los nodos de la lista
        liberar(&p);
        return 0;
}
```

- Aquí definimos la variable **p** de tipo Nodo*.
- Durante el programa **p** será el puntero al primer nodo de la lista.
- Luego invocamos a la función ***agregar*** para agregarle elementos.
- Al final, invocamos a las funciones ***mostrar*** y ***liberar***.
- La primera para mostrar por pantalla cada uno de los elementos de la lista y la segunda para liberar la memoria que ocupan.



Recorrer una lista para mostrar su contenido

- Para recorrer una lista y mostrar su contenido, simplemente, tenemos que posicionarnos en el primer nodo, mostrar su valor y avanzar al siguiente.

- Repetimos este proceso hasta llegar al final de la lista.

```
void mostrar(Nodo *p)
{
        Nodo* aux = p;
        // recorremos la lista hasta llegar al ultimo nodo
        while( aux != NULL )
        {
                printf("%i\n", aux->valor);
                // avanzamos a aux al próximo nodo
                aux = aux->sig;
        }
}
```

Notemos que en esta función simplemente recibimos el puntero al primer nodo de la lista (de tipo **Nodo***), no su dirección(**Nodo****).

Esto es porque aquí **no** vamos a modificarlo.

Simplemente, recorremos la lista pasando por cada uno de sus nodos para mostrar su valor por pantalla.

Liberar la memoria que utilizan los nodos de una lista enlazada

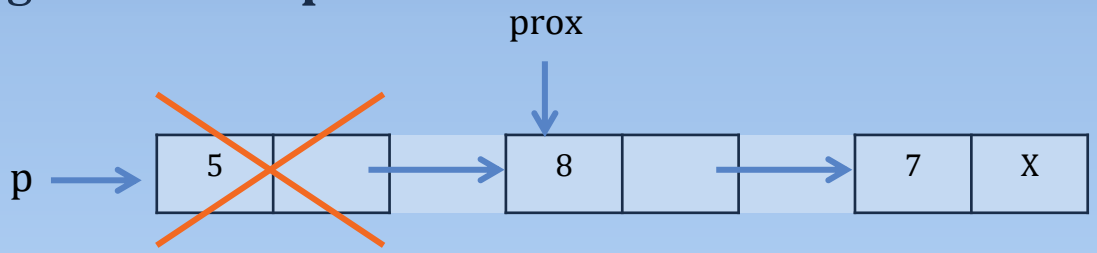
- La memoria necesaria para alojar los elementos que agregamos a la lista se gestiona, dinámicamente, a través de la función de **C** *malloc*.
- La memoria gestionada por *malloc* es persistente y permanece "asignada" durante toda la ejecución del programa.
- Debido a lo anterior, será nuestra responsabilidad liberar la memoria cuando ya no la necesitemos.
- Para esto, desarrollaremos la función liberar que recorre la lista enlazada liberando la memoria que ocupan cada uno de sus nodos.

- Esta función libera la memoria que utiliza la lista enlazada direccionada por **p** (puntero al primer nodo de la lista).
- Además al finalizar debe asignar el valor NULL a **p** ya que luego de liberar la memoria la lista quedará vacía.
- Esto significa que tenemos que recibir a **p** por referencia (**Nodo****).

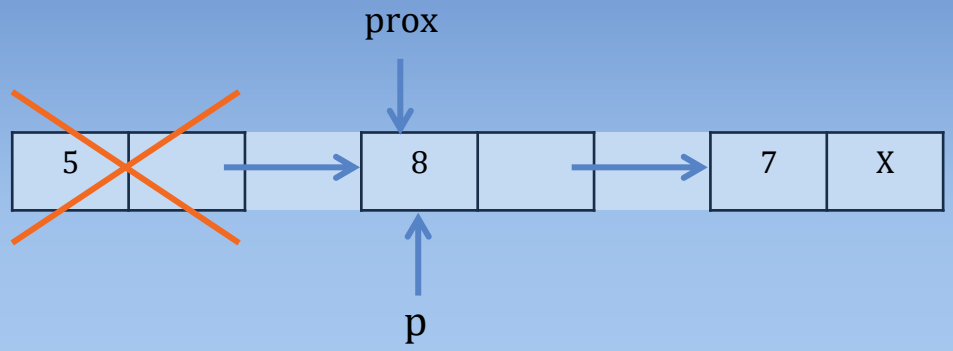
- El algoritmo finalizará cuando **p** tenga el valor NULL.
- Esto, dentro de la función, se traduce como **while(*p != NULL)**.
- Recordemos que recibimos a **p** por referencia; por lo tanto, la dirección del primer nodo de la lista es ***p**.

- Como *p es la dirección del primer nodo entonces **(*P)->sig** es la dirección del segundo elemento de la lista.
- La asignación **prox = (*p)->sig** asigna la variable **prox** la dirección del segundo elemento de la lista.
- Los paréntesis son necesarios ya que el operador “flecha” tiene precedencia sobre el operador “asterisco”.
- Si no utilizamos paréntesis entonces “estaríamos hablando del siguiente de **p**”.
- Recordemos que, dentro de la función, **p** es de tipo **Nodo****, no tiene campo **sig**.
- Quien tiene ese campo es ***p**.

- Luego de la asignación **prox = (*p)->sig** podemos liberar la memoria direccionada por ***p** ya que la dirección del siguiente nodo esta resguardada en **prox.**



- Ahora debemos hacer que el primer nodo de la lista sea el que está siendo apuntado por **prox.**
- La asignación ***p = prox** descarta, definitivamente, el primer nodo (ya liberado mediante la función free) y hace que la lista comience desde el segundo.



- Notemos que en la figura mantuvimos el dibujo del “viejo primer nodo de la lista” ya que el hecho de que lo hayamos liberado con ***free*** no implica que la información se haya perdido o borrado.
- Simplemente, ese espacio de memoria ya no pertenece más a nuestro programa y, de hecho, no tenemos forma de accederlo porque no lo tenemos apuntado con ningún puntero. Quedo desreferenciado.

- Función **liberar**(Nodo ** p)

```
void liberar(Nodo** p)
{
        while( *p != NULL )
        {
                Nodo* prox = (*p)->sig;
                free(*p);
                *p = prox;
        }
}
```

- Cuando **p** apunte al último nodo de la lista **prox=(*p)->sig** asignará el valor NULL a prox.
- Luego, al hacer *p = prox estaremos asignando NULL a **p** con lo que, finalmente, la lista quedará vacía y la memoria que ocupaba estará liberada.

Determinar si la lista contiene un valor determinado

- Naturalmente, si se tiene una lista de elementos en algún momento vamos a necesitar determinar si esta contiene o no un cierto valor.

- Para esto, desarrollaremos la función buscar que realiza una búsqueda secuencial sobre los nodos de la lista hasta encontrar aquel nodo cuyo valor sea el que estamos buscando.

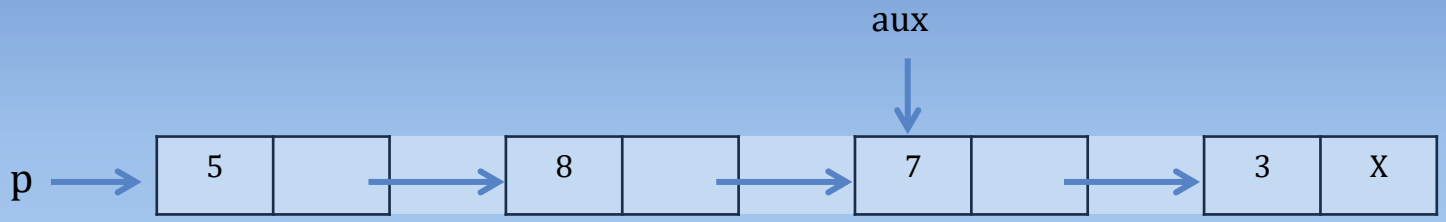
- Veamos la siguiente lista enlazada y supongamos que queremos determinar si contiene un nodo cuyo valor sea 7.

- Para desarrollar un algoritmo que lo pueda determinar tendremos que analizar el primer nodo y, si no contiene el valor que estamos buscando, pasar al siguiente y así hasta analizar el último elemento de la lista.

- Utilizaremos un puntero auxiliar **aux**, inicialmente, apuntando al primer nodo.

- Si el nodo apuntado por **aux** no tiene el valor que estamos buscando entonces lo haremos avanzar para que apunte al siguiente.

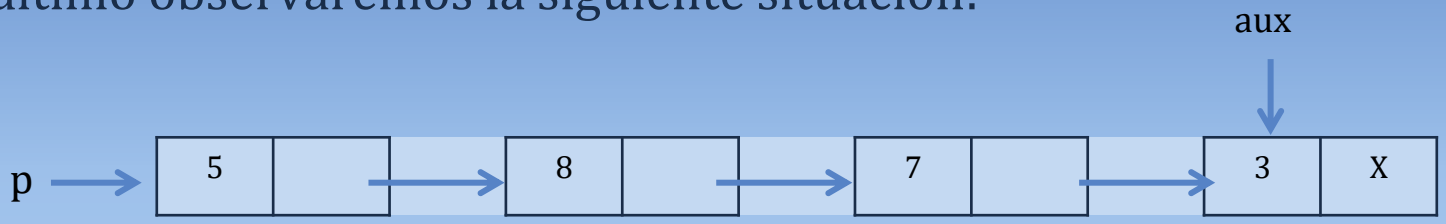
- Si el valor de alguno de los nodos de la lista coincide con el que estamos buscando entonces podemos dar por finalizado el algoritmo.



- La función retornará un puntero al nodo cuyo valor sea el que buscamos.

- Ahora bien, ¿Qué sucederá cuándo busquemos un elementos que no esté en la lista, por ejemplo el 25?

- Luego de analizar y avanzar sobre todos los nodos de la lista, al llegar al último observaremos la siguiente situación:



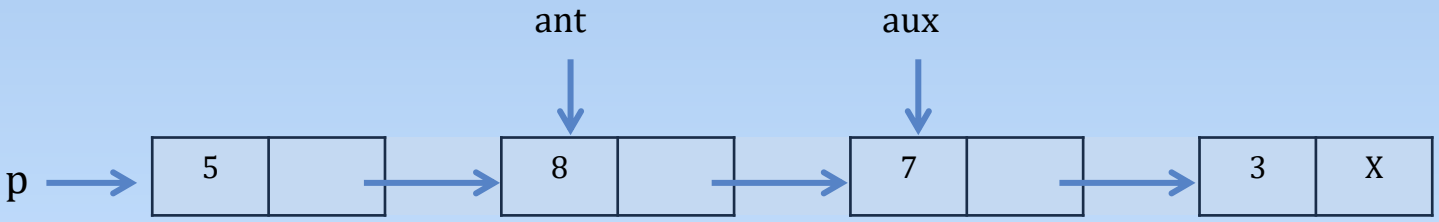
- En este caso, el nodo apuntado por **aux** tiene el valor de 3, que no coincide con el valor 25 que estamos buscando.
- Luego, al avanzar el puntero haremos **aux=aux->sig**.
- Como **aux** apuntaba al último nodo de la lista resulta que su referencia al siguiente es NULL; por lo tanto, le estaremos asignando el valor NULL a **aux**, lo que hará finalizar el ciclo iterativo y, al retornar **aux**, estaremos retornando NULL.

- Luego, la función ***buscar*** retorna un puntero al nodo que contiene el valor que buscamos o NULL si ningún nodo de la lista contiene dicho valor.

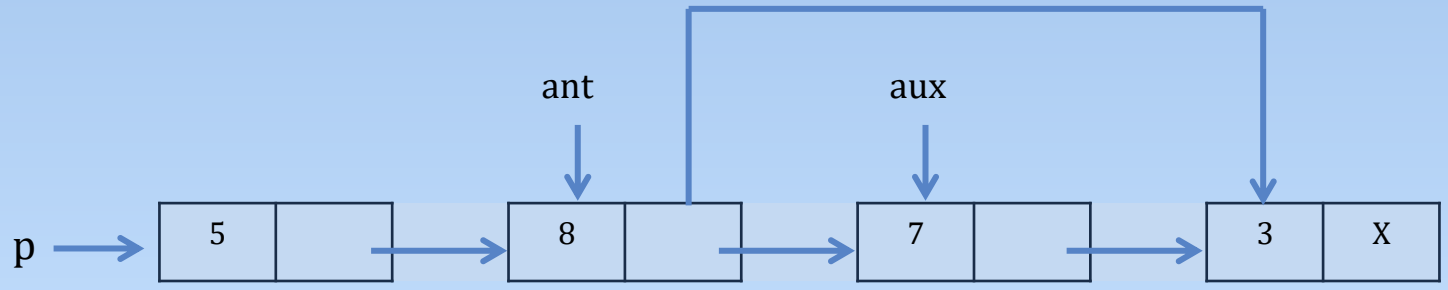
```
Nodo* buscar(Nodo* p, int v)
{
    Nodo* aux = p;
    while( (aux != NULL) && (aux->valor!=v) )
    {
        aux = aux->sig;
    }
    return aux;
}
```

Eliminar un elemento de la lista

- El siguiente algoritmo nos permitirá eliminar un nodo de la lista cuyo valor coincida con el que especifiquemos como argumento.
- Dada la siguiente lista enlazada, supongamos que queremos eliminar el nodo cuyo valor es 7.



- Para eliminar el elemento cuyo valor es 7 tenemos que obtener dos punteros: uno que apunte al nodo que vamos a eliminar (***aux***) y otro que apunte al nodo anterior (***ant***).
- Una vez logrado esto eliminaremos el nodo que está siendo referenciado por ***aux*** haciendo que el siguiente de ***ant*** apunte al siguiente de ***aux***.



- Luego de hacer que el siguiente de ***ant*** apunte al siguiente de ***aux***, el nodo referenciado por ***aux*** dejó de ser parte de la lista, pero aún ocupa memoria.
- El próximo paso será liberar ese espacio de memoria haciendo *free*(***aux***).
- Para obtener la referencia al nodo que vamos a eliminar y la referencia al nodo anterior recorreremos secuencialmente la lista y, antes de avanzar al siguiente nodo, asignaremos en el puntero ***ant*** el valor actual de ***aux***.
- Notemos que la función recibe a **p** por referencia porque en el caso de eliminar al primer nodo **p** debe pasar a apuntar al siguiente y si la lista tiene un único nodo entonces **p** debe quedar en NULL.

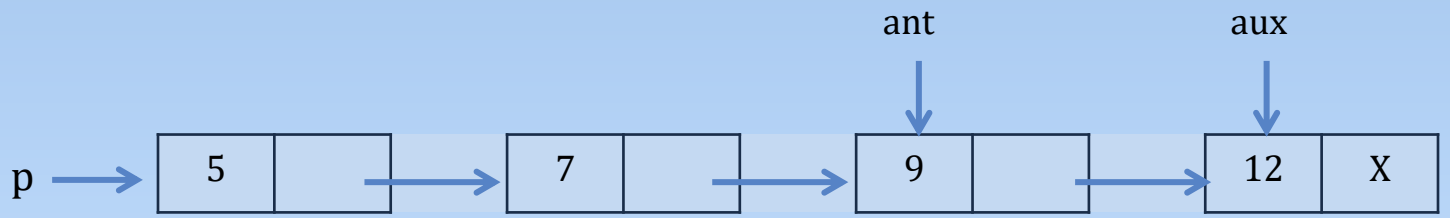
Función eliminar

```
void eliminar(Nodo** p, int v)
{
	Nodo* aux = *p;
	Nodo* ant = NULL;
	while( (aux!=NULL) && (aux->valor!=v) )
	{
		ant = aux;
		aux = aux->sig;
	}

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

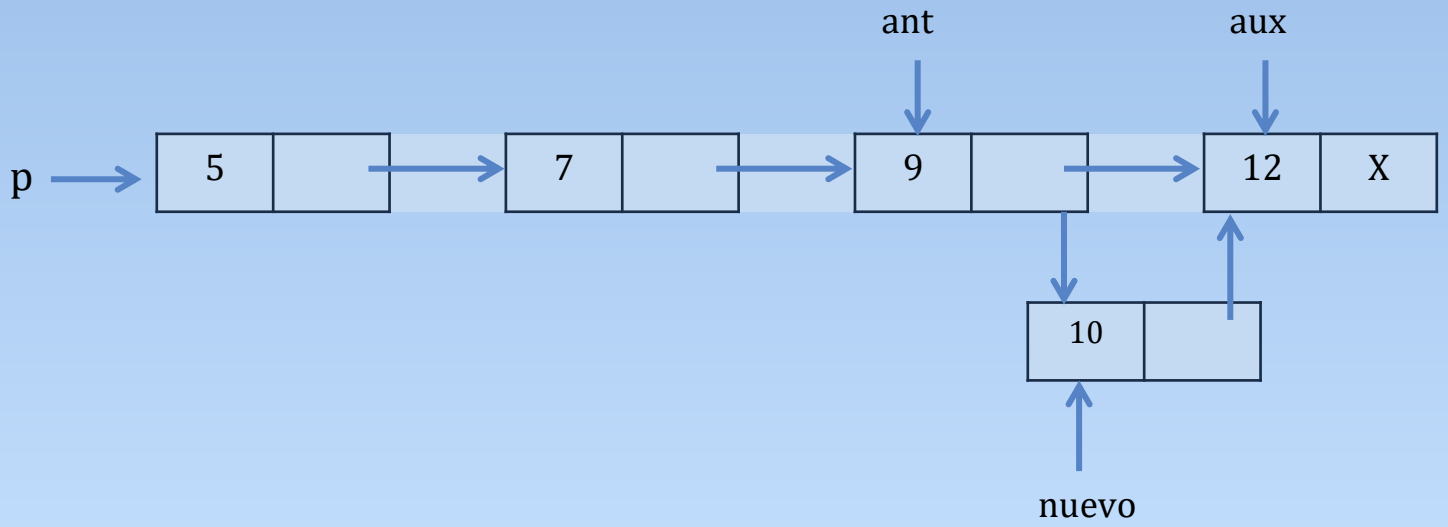
Insertar un valor respetando el ordenamiento de la lista

- Analizaremos ahora un algoritmo que nos permita insertar un valor dentro de una lista enlazada de forma tal que su ubicación respete el orden natural de esta. Veamos un ejemplo:

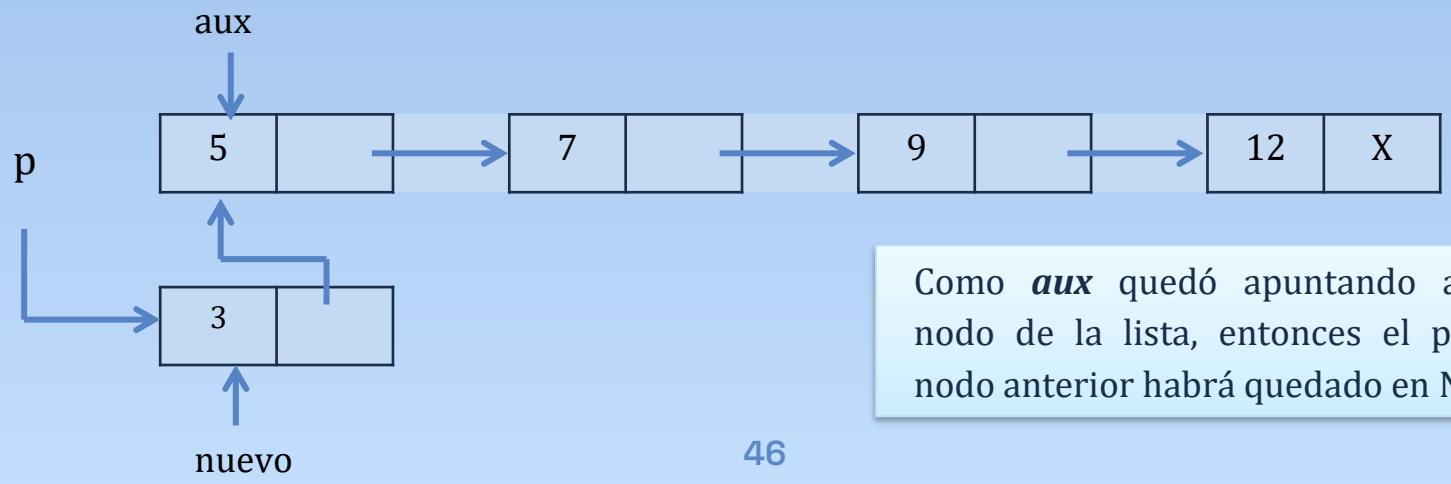
- Los elementos representados en los nodos de esta lista se encuentran ordenados naturalmente.
- Para insertar un nuevo nodo, por ejemplo, con valor 10 tenemos que recorrer la lista hasta encontrar el primer elemento mayor que, en este caso, es 12 y quedarnos con un puntero al elemento anterior (9).



- Luego, para insertar el nodo con valor 10 en la posición que corresponda debemos hacer que el siguiente del nuevo nodo apunte a **aux** y también que el siguiente de **ant** apunte al nuevo nodo.

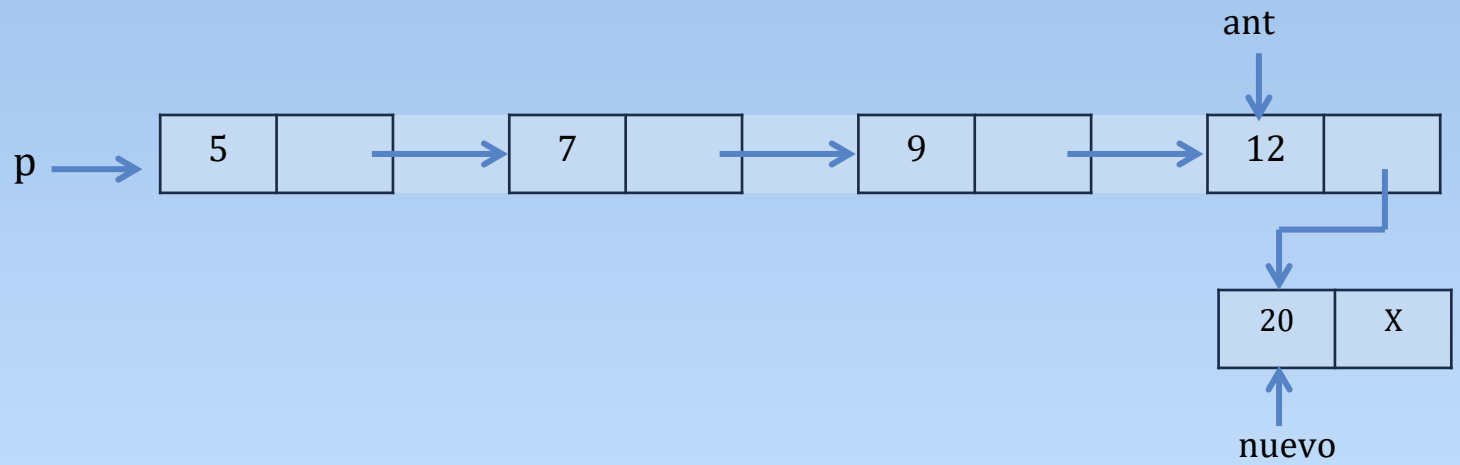


- Tenemos que analizar los siguientes casos particulares:
- El primero se da cuando el elemento que vamos a insertar es menor que el primer elemento de la lista.
- En este caso tenemos que modificar el valor de **p** para hacerlo apuntar al nuevo nodo; el siguiente del nuevo debe apuntar al viejo valor de **p**.
- Supongamos que vamos a insertar el valor de 3, entonces:



Como ***aux*** quedó apuntando al primer nodo de la lista, entonces el puntero al nodo anterior habrá quedado en NULL.

- El otro caso particular se da cuando el elemento que vamos a insertar es mayor que todos los elementos de la lista, por lo que tendremos que ubicarlo al final.
- En este caso, ***ant*** quedará apuntando al último nodo y **aux** habrá quedado en NULL.



Función insertarOrdenado( )

```
Nodo* insertarOrdenado(Nodo **p, int v)
{
	Nodo* nuevo = (Nodo *) malloc(sizeof(Nodo));
	nuevo->valor = v;
	nuevo->sig = NULL;

	Nodo* aux = *p;
	Nodo* ant = NULL;

	while( (aux!=NULL) && (aux->valor <= v) ) {
		ant = aux;
		aux = aux->sig;
	}

	if(ant == NULL){
		*p = nuevo;
	}
	else{
		ant->sig = nuevo;
	}
	nuevo->sig = aux;
	return nuevo;
}
```

Insertar un valor solo si la lista aún no lo contiene

- Este algoritmo permite insertar (en orden) un valor solo si la lista aún no lo contiene.
- Se implementará como una función que retornará un puntero al nodo que contiene dicho valor, si la lista ya lo contenía.
- En caso contrario, insertará el valor y retornará un puntero al nodo recientemente insertado.



- Para determinar si el valor ***v*** ya estaba contenido o se agregó luego de haber invocado a la función, recibiremos el parámetro por referencia ***enc*** (encontrado) donde asignaremos ***true*** o ***false*** según corresponda.

```
Nodo* buscarEInsertarOrdenado(Nodo **p, int v, int* enc)
{
	Nodo* r = buscar(*p, v);
	*enc = r!=NULL;		//Vale 1 si lo encuentra y 0 si no lo encuentra

	if(!*enc)
	{
		r = insertarOrdenado(p, v);
	}

	return r;
}
```