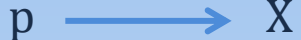
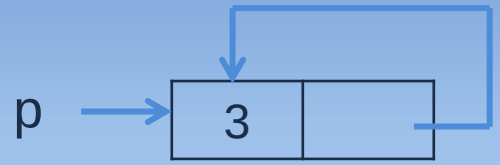
Lista Enlazada Circular

Lista Enlazada Circular

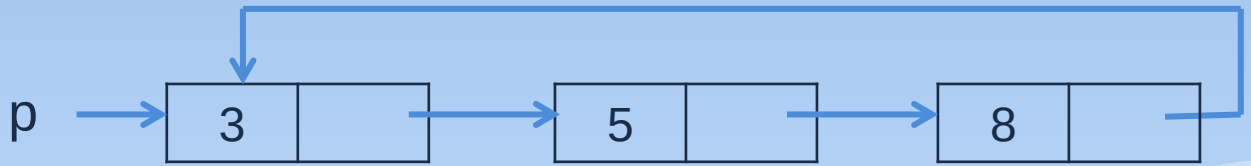
- Una lista circular es una lista enlazada en la que el último nodo apunta al primero.



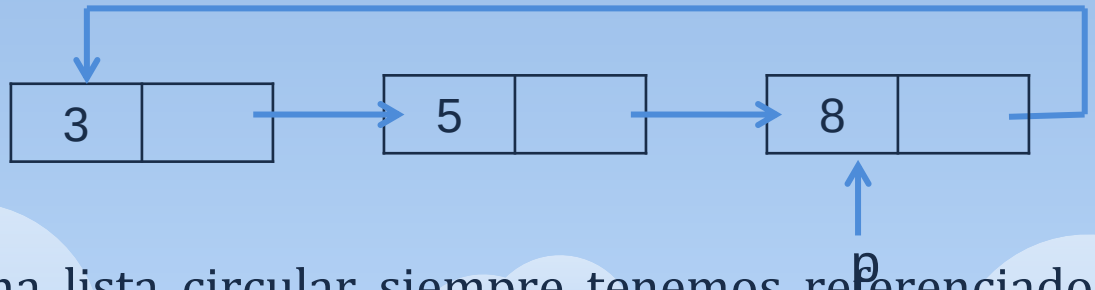
- Una lista circular con un único elemento.



- Una lista circular con más de un elemento.



- Dado que en una lista circular el último nodo tiene una referencia al primero resultará más provechoso mantener a **p** apuntando al último nodo de la lista ya que desde allí podremos acceder al primero, que estará referenciado por **p->sig**.

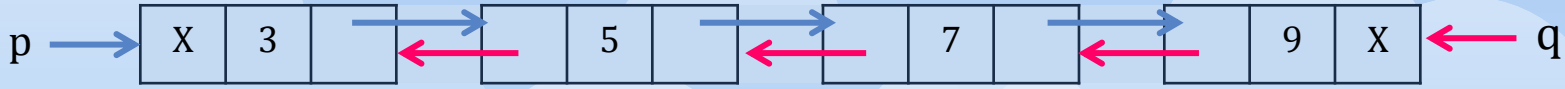


- Así, en una lista circular siempre tenemos referenciado el último nodo que agregamos (**p**) e, indirectamente, también tenemos referenciado el primero (**p->sig**)

Lista doblemente enlazada

Listas doblemente enlazadas

- Así como en una lista cada nodo tiene un puntero al siguiente, en una lista doblemente enlazada cada nodo tiene dos punteros: uno apuntando al siguiente y otro apuntando al anterior.

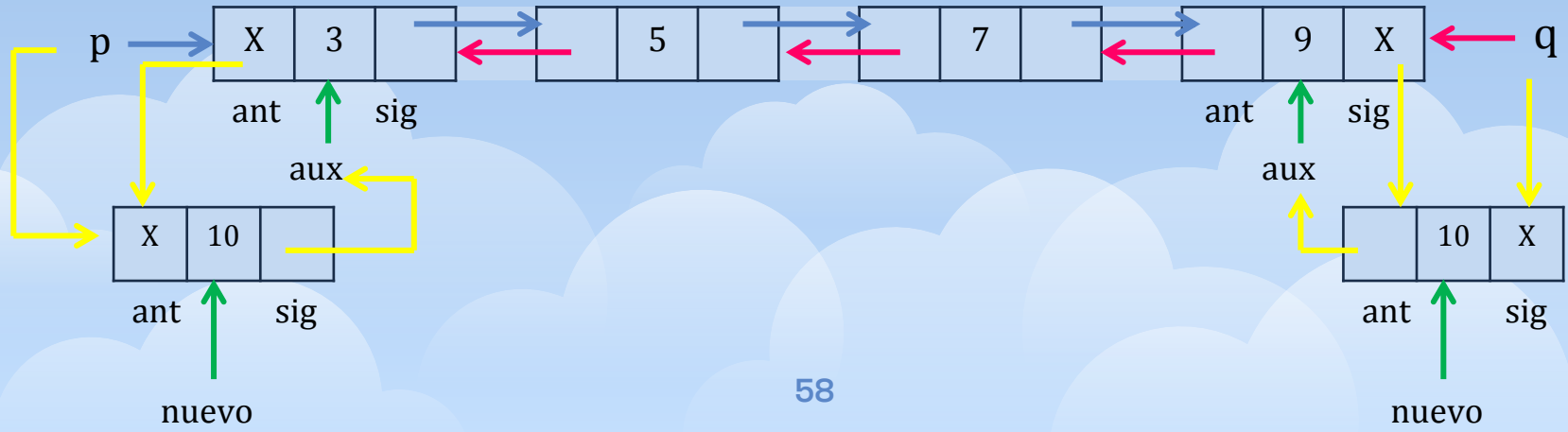


- En la figura vemos una lista doblemente enlazada: cada nodo tiene un puntero al siguiente y un puntero al anterior.
- Obviamente, el puntero al anterior del primer nodo y el puntero al siguiente del último son NULL.
- También vemos que manejamos dos punteros p y q que apuntan al primer y al último nodo de la lista respectivamente.
- La lista doblemente enlazada utiliza más memoria que la lista enlazada, pero ofrece las siguientes ventajas:
  - La lista puede recorrerse en ambas direcciones.
  - Las operaciones insertar y eliminar utilizan menor cantidad de instrucciones ya que el mismo nodo tiene la dirección del siguiente y del anterior.

Agregar elementos al inicio y al final

Es el primer nodo de la lista

Hay más de un nodo en la lista



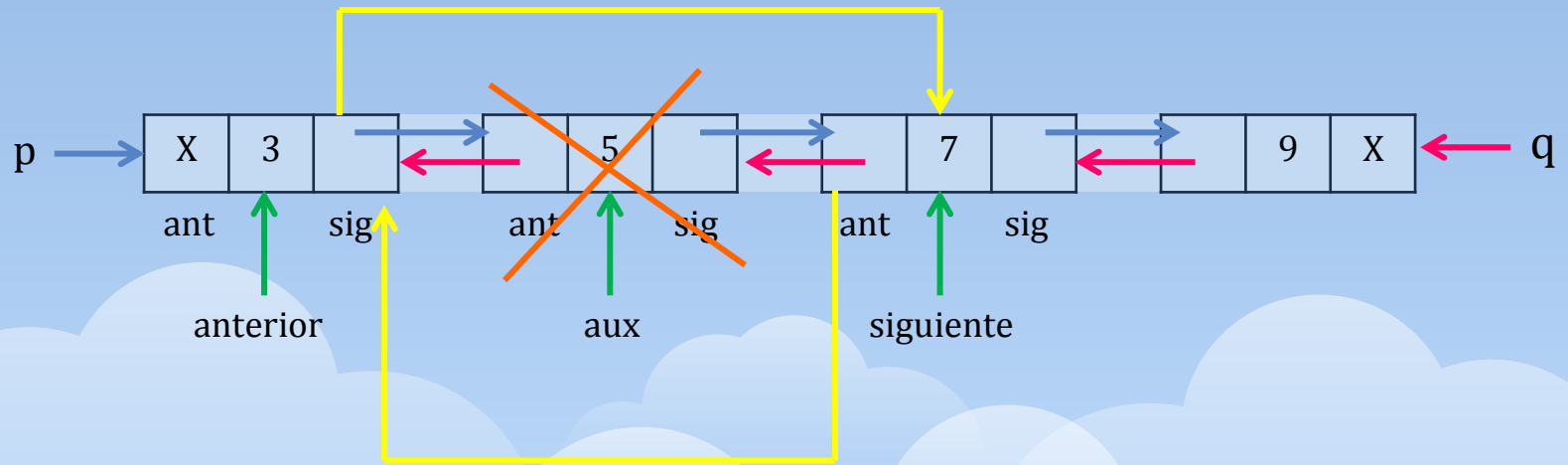
Agregar elementos al inicio

```
void agregarAlInicio(Nodo** p, Nodo** q, int v)
{
        // creamos el nuevo nodo
        Nodo *nuevo = (Nodo*) malloc(sizeof(Nodo));
        nuevo->valor = v;
        nuevo->sig = NULL;
        nuevo->ant = NULL;

        // si la lista esta vacia entonces hacemos que p y q apunten al nuevo nodo
        if( *p == NULL )
        {
                *p = nuevo;
                *q = nuevo;
        }
        else
        {
                Nodo* aux = *p;
                // como aux apunta al primero entonces su anterior sera el nuevo nodo
                nuevo->sig = aux;
                aux->ant = nuevo;
                *p = nuevo;
        }
}
```

Agregar elementos al final

```
void agregarAlFinal(Nodo** p, Nodo** q, int v)
{
        // creamos el nuevo nodo
        Nodo *nuevo = (Nodo*) malloc(sizeof(Nodo));
        nuevo->valor = v;
        nuevo->sig = NULL;
        nuevo->ant = NULL;

        // si la lista esta vacia entonces hacemos que p y q apunten al nuevo nodo
        if( *p == NULL )
        {
                *p = nuevo;
                *q = nuevo;
        }
        else
        {
                Nodo* aux = *q;
                // como aux apunta al ultimo entonces su siguiente sera el nuevo nodo
                aux->sig = nuevo;
                nuevo->ant = aux;
                *q = nuevo;
        }
}
```

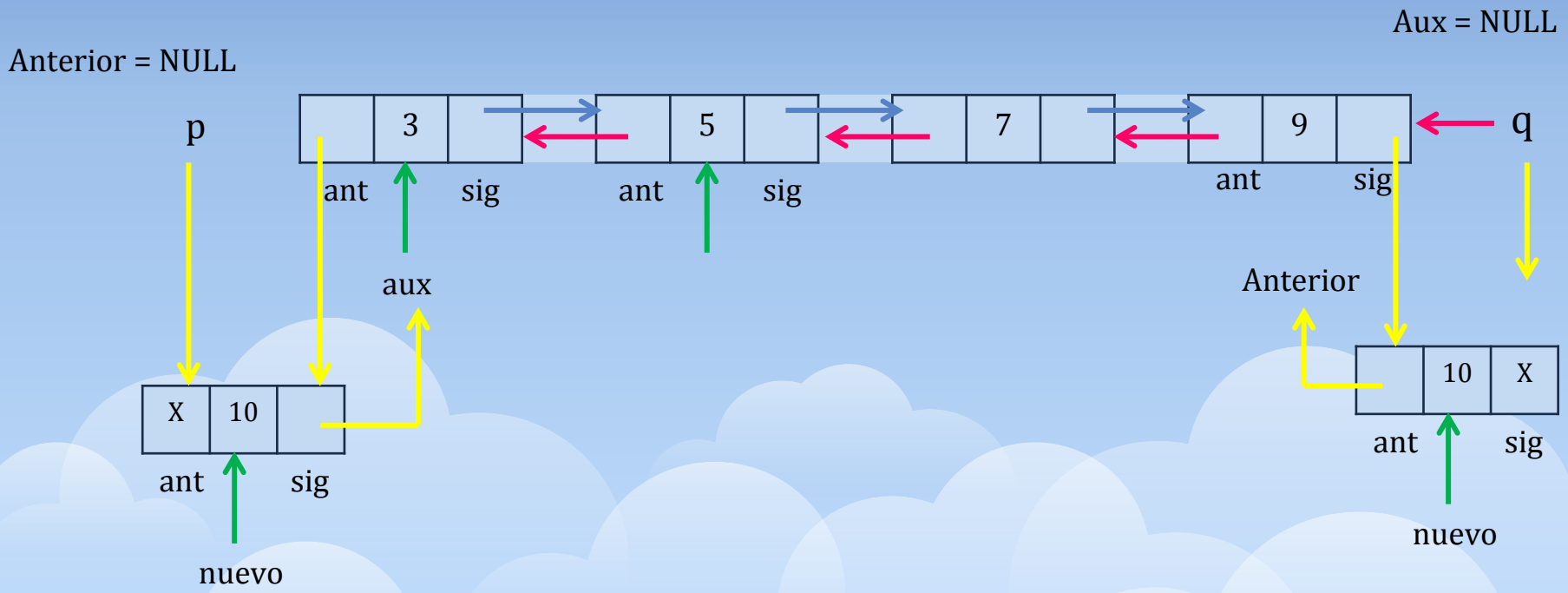
Eliminar nodo



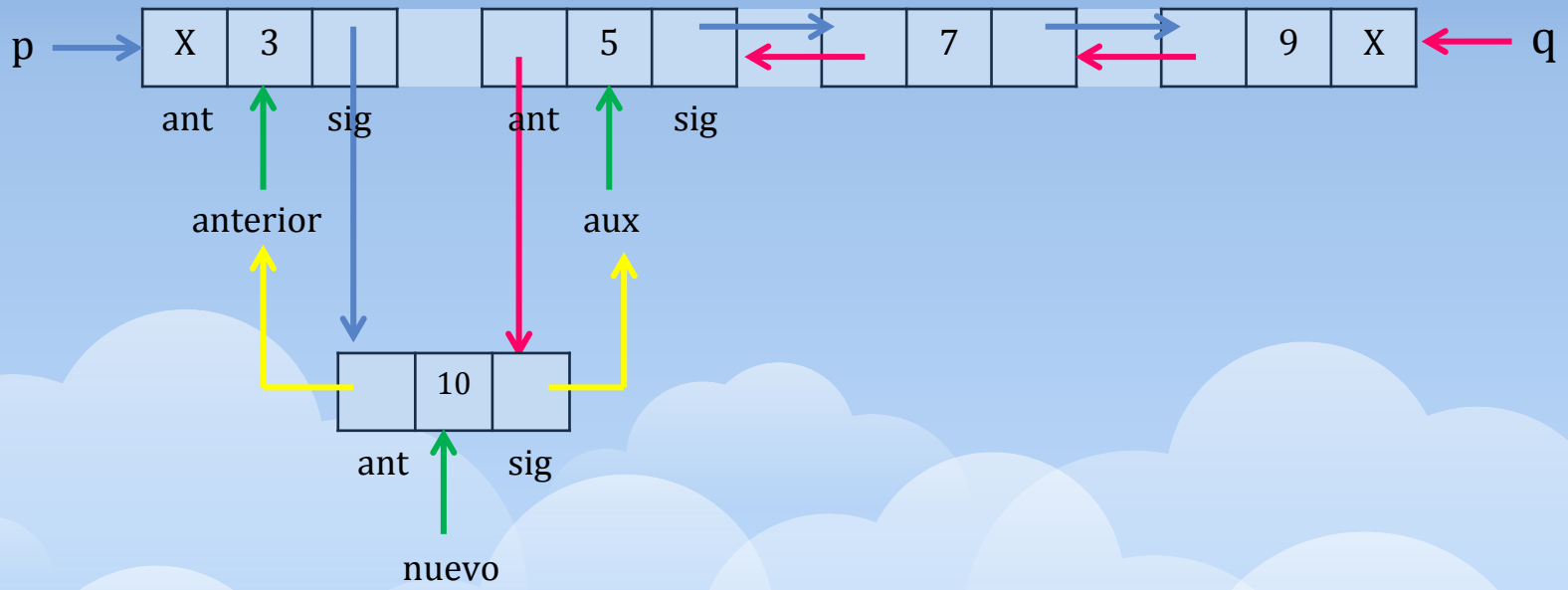
Eliminar nodo

```
void eliminar(Nodo** p, Nodo ** q, int v)
{
        Nodo* aux = *p;
        Nodo* anterior = NULL;
        Nodo* siguiente = NULL;

        while( (aux!=NULL) && (aux->valor!=v) )
                aux = aux->sig;

        anterior = aux->ant;
        siguiente = aux->sig;

        if(aux!=NULL) //Si la lista no esta vacia
        {
                if(aux->ant != NULL) //Si no es el primer elemento de la lista
                        anterior->sig = aux->sig;
                else            //Es el primero de la lista, entonces movemos  p de inicio al segundo elemento
                        *p = aux->sig;

                if(aux->sig != NULL) //Si no es el ultimo elemento de la lista
                        siguiente->ant = aux->ant;
                else        //Es el ultimo elemento de la lista, entonces movemos q del final al penultimo elemento
                        *q = aux->ant;
                free(aux);
        }
}
```

Insertar ordenado
Al inicio o al final



Insertar ordenado
Al inicio o al final



Insertar nodo ordenado

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

        while( (aux!=NULL) && (aux->valor <= v) )
        {
                anterior = aux;
                aux = aux->sig;
        }
        if(anterior == NULL) //Se va a insertar al inicio de la lista
        {
                *p = nuevo;
                aux->ant = nuevo;
                nuevo->sig = aux;
        }
        if(aux == NULL) //Se va a insertar al final de la lista
        {
                *q = nuevo;
                anterior->sig = nuevo;
                nuevo->ant = anterior;
        }
        //Se va a insertar en medio de la lista
        if(anterior != NULL && aux!=NULL)
                {
                anterior->sig = nuevo;
                aux->ant = nuevo;
                nuevo->ant = anterior;
                nuevo->sig = aux;
        }

        return nuevo;
}
```

Recorrer 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 proximo nodo
            aux = aux->sig;
    }
}
```

```
void mostrarOrdenInverso(Nodo *q)
{
    Nodo* aux = q;

    // recorremos la lista hasta llegar al primer nodo
    while( aux != NULL )
    {
            printf("%i\n",aux->valor);
            // avanzamos a aux al proximo nodo
            aux = aux->ant;
    }
}
```

Buscar nodo en lista

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

```
Nodo* buscarInverso(Nodo* q, int v)
{
      Nodo* aux = q;
      while( (aux != NULL) && (aux->valor!=v) )
      {
                  aux = aux->ant;
       }
       return aux;
}
```

Nodos con múltiples campos.
Nodo con un único valor de tipo Struct

Opción 1

Nodo con múltiples campos

- La primera solución para este problema consiste en agregar al nodo todos los campos de la estructura que queremos guardar. En el caso de struct Persona el nodo quedaría así:

```
typedef struct Nodo
{
        int dni;              //datos de la persona
        char nombre[20];      //datos de la persona
        long fechaNac;        //datos de la persona
        struct Nodo *sig;     //referencia al siguiente nodo
}Nodo;
```

- Para implementar una pila con nodos de este tipo, las operaciones poner y sacar deberían plantearse de la siguiente manera:

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

```
void poner(Nodo** p, Persona v)
{
                Nodo* nuevo = (Nodo*) malloc(sizeof(Persona));

                //asignamos uno a uno los campos de v a nuevo
                nuevo->dni = v.dni;
                nuevo->fechaNac = v.fechaNac;
                strcpy(nuevo->nombre, v.nombre);

                //ahora si enlazamos el nodo al principio de la lista
                nuevo->sig = *p;
                *p = nuevo;
}
```

```
Persona sacar(Nodo** p)
{
        //Definimos una variable de tipo Persona
        Persona v;

        //Asignamos uno a uno los campos
        v.dni = (*p)->dni;
        v.fechaNac = (*p)->fechaNac;
        strcpy(v.nombre, (*p)->nombre);

        //Ahora desenlazamos el primer nodo
        Nodo* aux = *p;
        *p = (*p)->sig;
        free(aux);

        return v;
}
```

Opción 2

Nodo con un único valor de tipo Struct

- La otra opción para que un nodo permita guardar varios valores es definirle un único campo de tipo Struct. Con esta alternativa podemos implementar una pila de personas de la siguiente manera:

```
typedef struct Nodo
{
        Persona info;                   //Todos los datos de la persona
        struct Nodo *sig;               //Referencia al siguiente nodo
}Nodo;
```

```
void poner(Nodo** p, Persona v)
{
        Nodo* nuevo = (Nodo*) malloc(sizeof(Persona));

        //asignamos los datos de la persona
        nuevo->info = v;

        //ahora si, enlazamos el nodo al principio de la lista
        nuevo->sig = *p;
        *p = nuevo;
}
```

Si bien las dos soluciones son correctas, es evidente que la segunda opción implica menos trabajo y es más fácil de implementar.

```
Persona sacar(Nodo** p)
{
        //Rescatamos los datos del primer nodo
        Persona v = (*p)->info;

        //Ahora desenlazamos el primer nodo
        Nodo* aux = *p;
        *p = (*p)->sig;
        free(aux);

        return v;
}
```

- Respecto de la función ***buscar*** podríamos implementarla de dos maneras:

En esta implementación recibimos (por referencia) el puntero al primer nodo de la lista y el DNI de la persona que queremos buscar. Esto es correcto si el DNI es suficiente para identificar a una persona.

```
Nodo* buscar(Nodo** p, int dni)
{
        //….
}
```

Sin embargo, si necesitamos más datos o todos los datos de la estructura para identificar a una persona entonces el prototipo de la función *buscar* debería ser el siguiente:

```
Nodo* buscar(Nodo** p, Persona v)
{
        //….
}
```

- En todos los casos la función *buscar* retorna un *Nodo** ya que si encuentra a la persona que estamos buscando retornará un puntero al nodo que contiene sus datos, pero si ningún nodo tiene la información de dicha persona entonces retornará NULL.