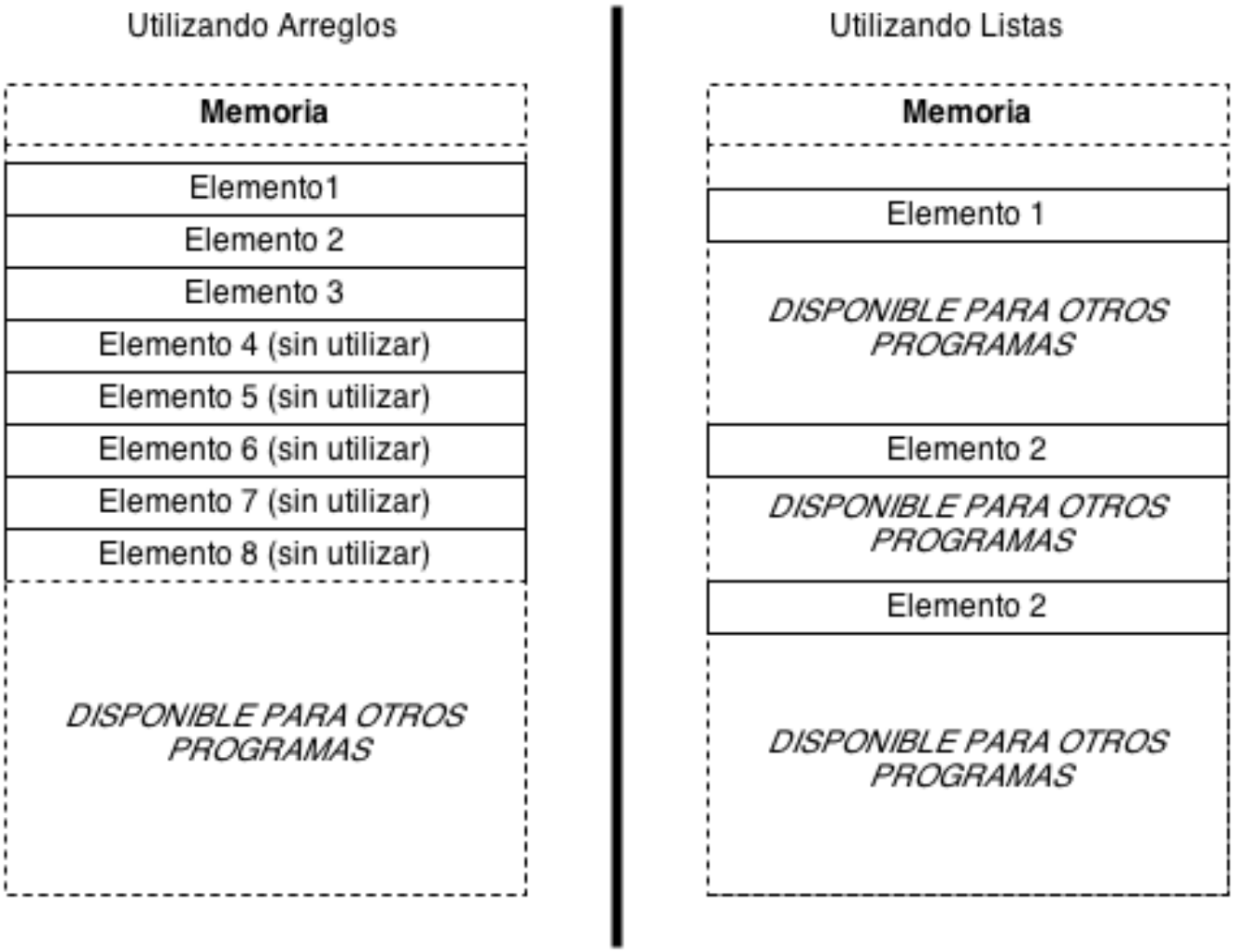
Listas en C

Listas, ¿Para qué?

Para no requerir grandes porciones de memoria contigua se utilizan las listas.

Al utilizar punteros y funciones para administrar memoria (malloc, free), los datos cargados en el programa pueden crecer indefinidamente, sin necesidad de recompilar ni estar copiando matrices de un lado a otro de la memoria.

Comparación entre Listas y Arreglos



Listas, ¿Cómo?

Existen muchas formas de utilizar listas, como siempre la que elijamos dependerá de los requerimientos que tenemos.

Un ejemplo básico sería:

- Poder recorrer la lista en un solo sentido.
- Poder ingresar elementos nuevos en la lista.
- Saber la cantidad de elementos que hay en la lista.

Para reciclar el código, separamos la implementación de la lista de los datos a ordenar mediante un puntero a un struct.

Structs para manipular listas

Para empezar vamos a definir dos structs para trabajar con listas.

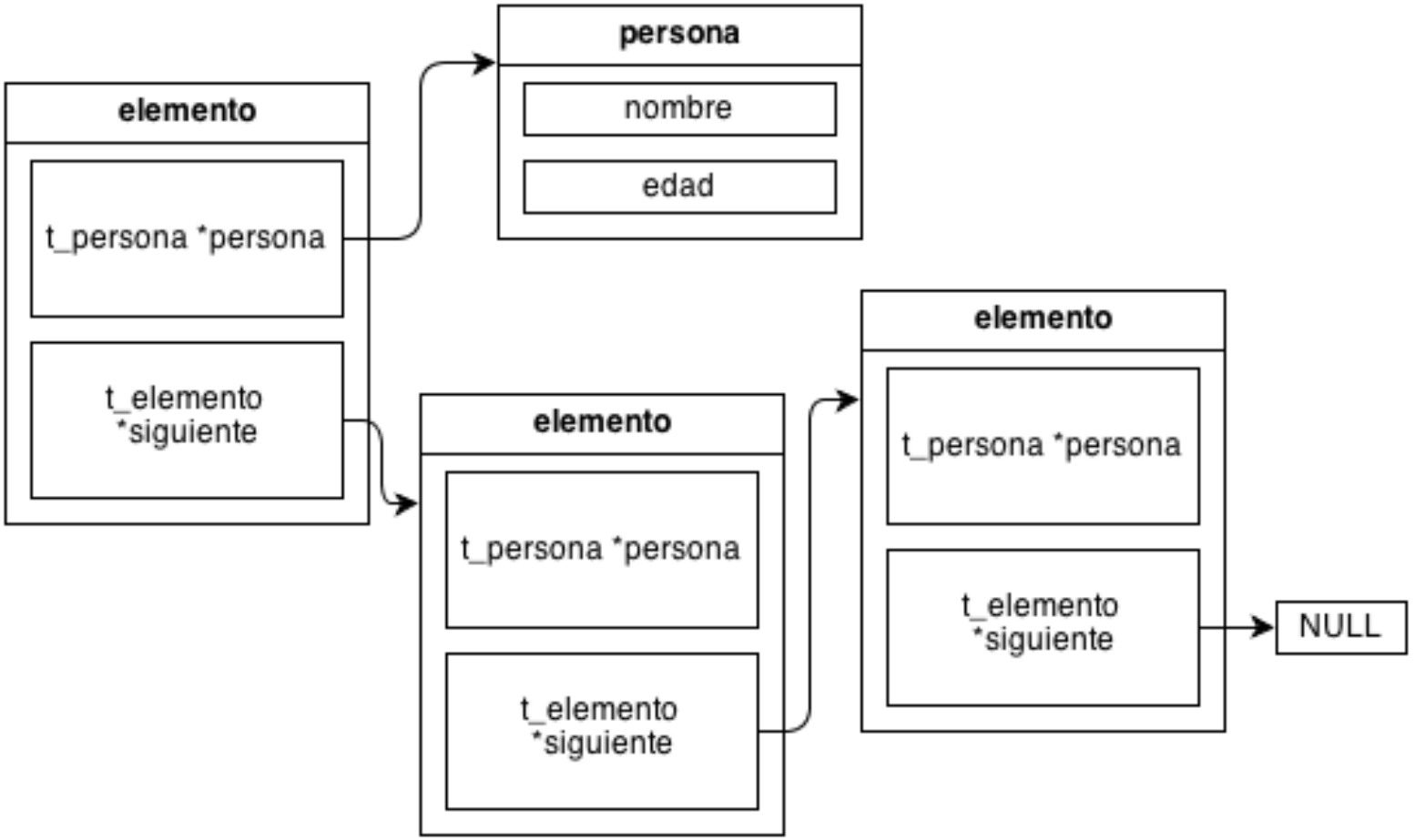
```
typedef struct t_elemento {
    t_persona *persona; //un puntero al struct con datos a ordenar
    struct t_elemento *siguiente;
} t_elemento;

typedef struct t_manipulador {
    t_elemento *primero; //un puntero al primer elemento de la lista
    t_elemento *ultimo; //un puntero al último elemento
    unsigned int cantidad; //un contador de elementos
} t_manipulador;
```

Lógica de las listas

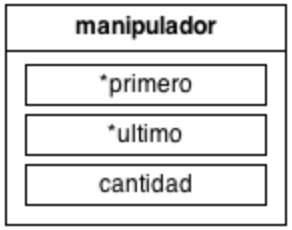
- Exceptuando el primer elemento, todos los demás están referenciados por el elemento anterior.
- Exceptuando el último elemento, todos los demás tienen un puntero al elemento siguiente.
  En caso de tratarse del último elemento, esa referencia apunta a NULL.

Lógica de las listas



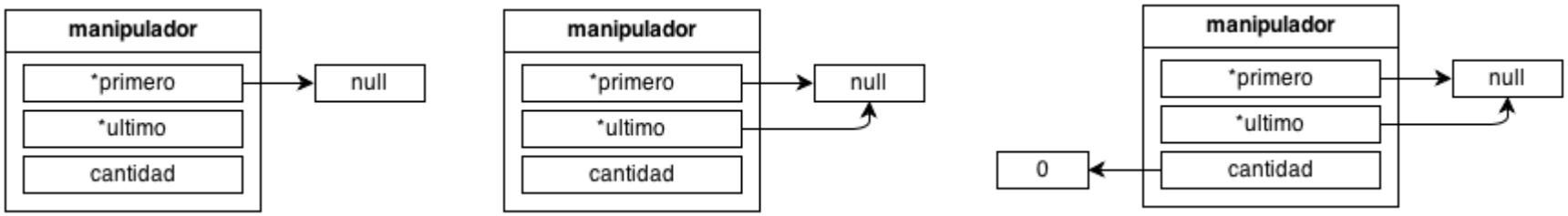
Crear una lista nueva

Inicio un nuevo Manipulador




```
t_manipulador manipulador;
```

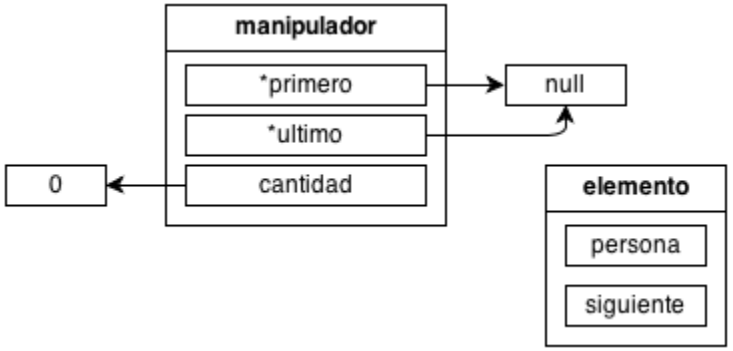
Apunto los "índices" a Null




```
manipulador.primero=Null;
manipulador.ultimo=Null;
manipulador.cantidad=0;
```

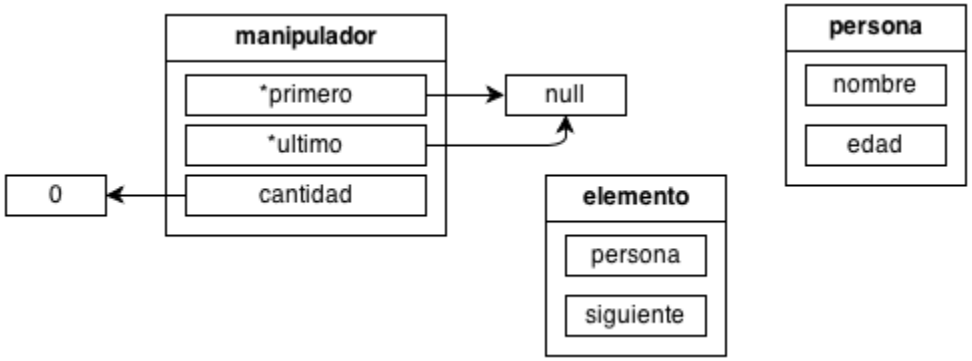
Agregar un primer elemento

Inicio un nuevo elemento



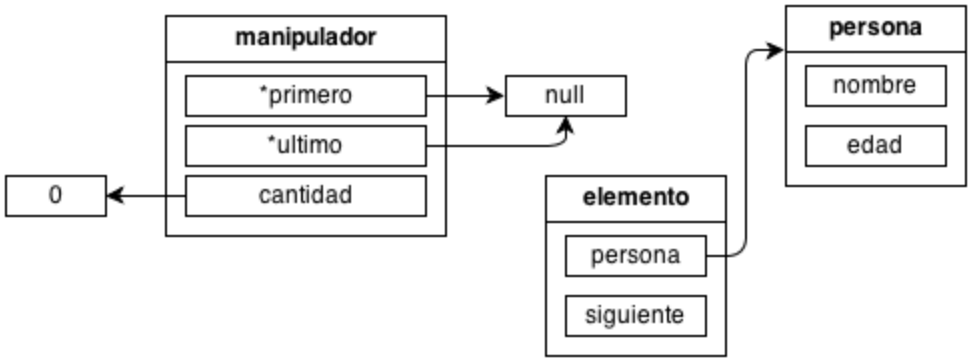

```
t_elemento *elemento;
elemento=malloc(sizeof(t_elemento));
```

Inicio una nueva persona



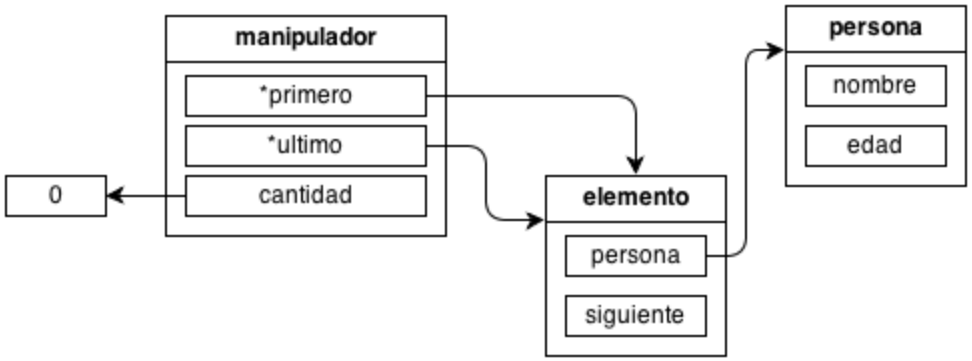

```
t_persona *persona;
persona=malloc(sizeof(t_persona));
```

Apunto a la persona correspondiente del elemento



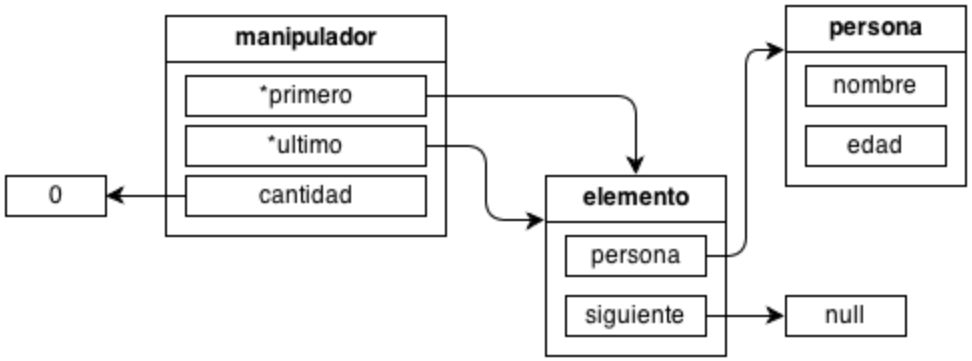

```
elemento->persona=persona;
```

Apunto los dos "índices" al elemento



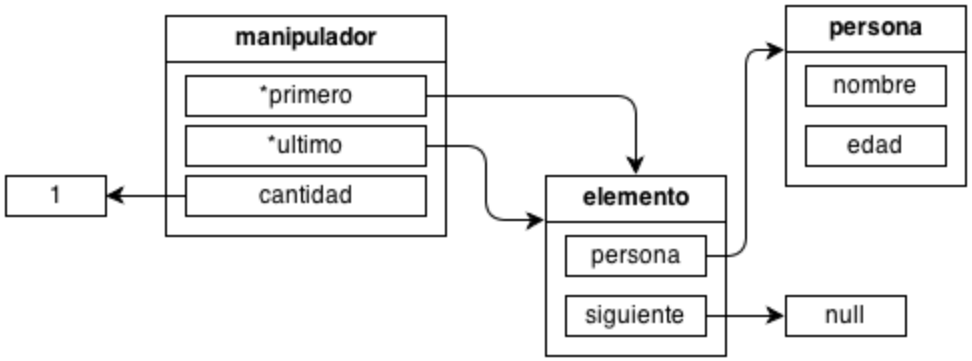

```
manipulador.primero=elemento;
manipulador.ultimo=elemento;
```

Apunto a Null el campo siguiente del elemento.




```
elemento->siguiente=Null;
```

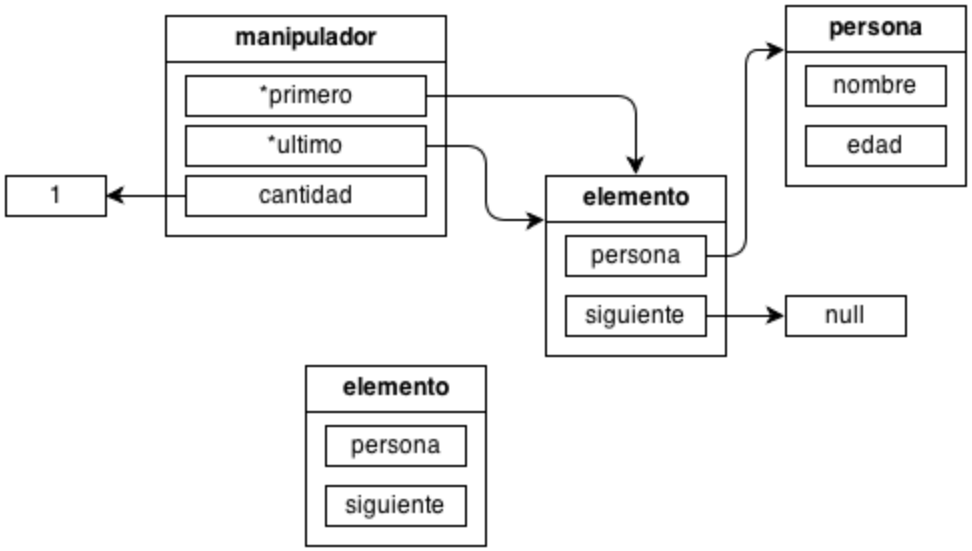
Incremento el contador de elementos.




```
manipulador.cantidad++;
```

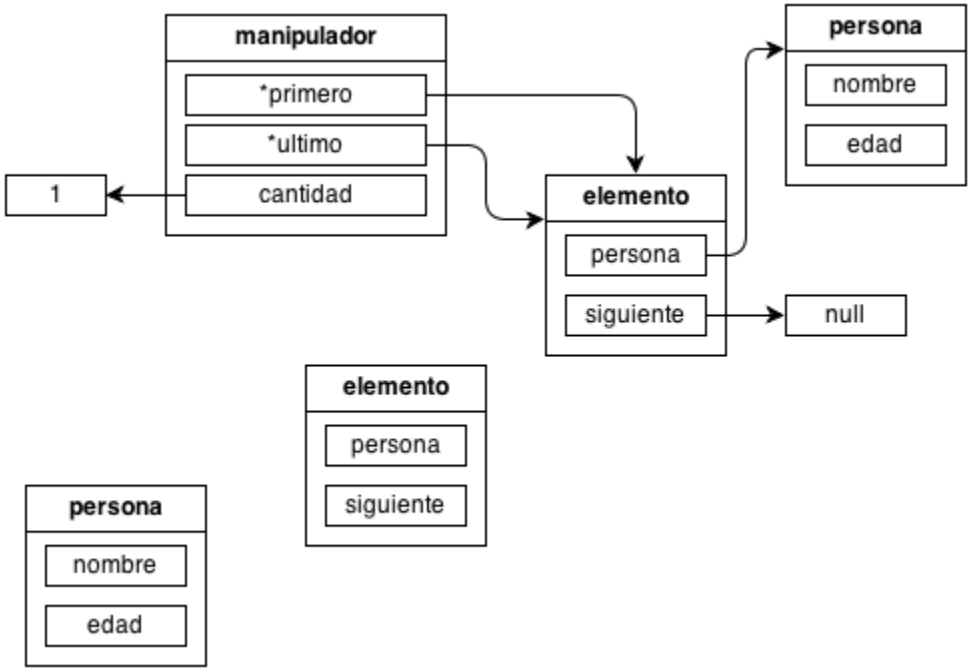
Agregar un elemento al final de la lista

Inicio un nuevo elemento



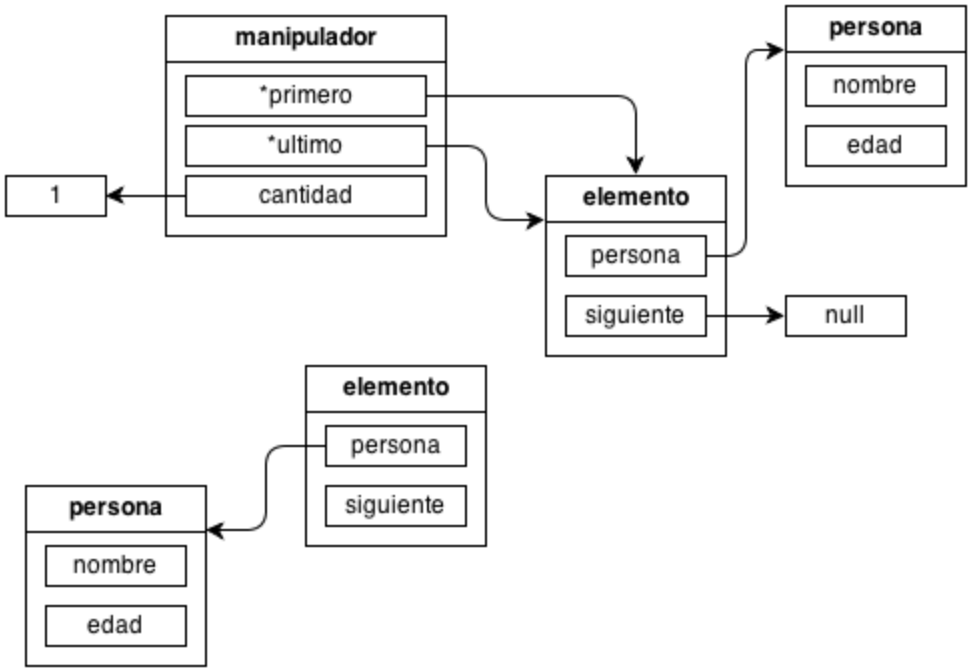

```
t_elemento *nuevo_e;
nuevo_e=malloc(sizeof(t_elemento));
```

Inicio una nueva persona



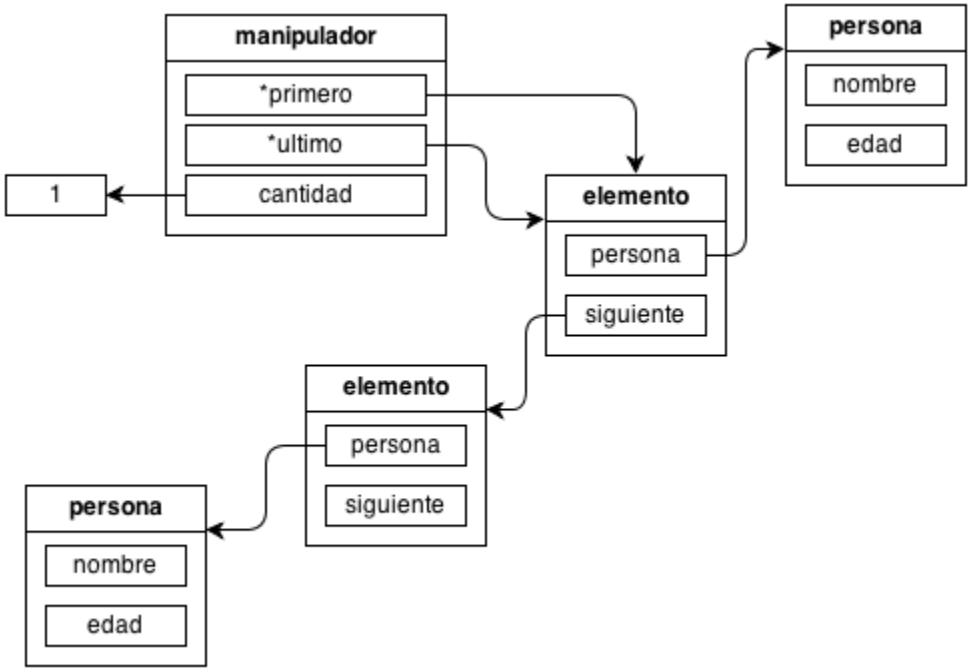

```
t_persona *nueva_p;
nueva_p=malloc(sizeof(t_persona));
```

Apunto a la persona en el elemento



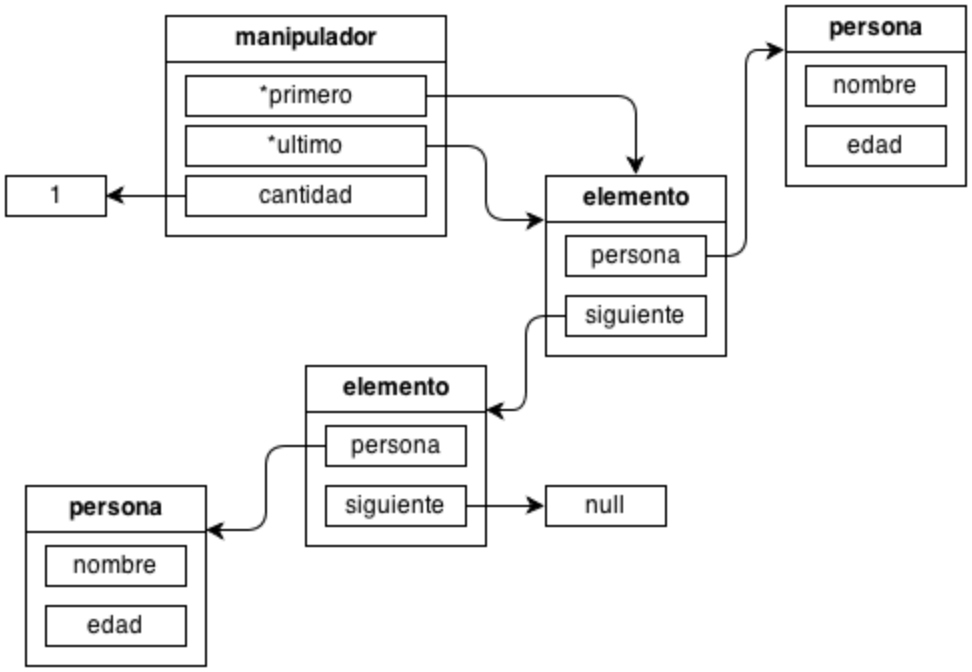

```
nuevo_e->persona=nueva_p;
```

Apunto al nuevo elemento en el hasta ahora último elemento



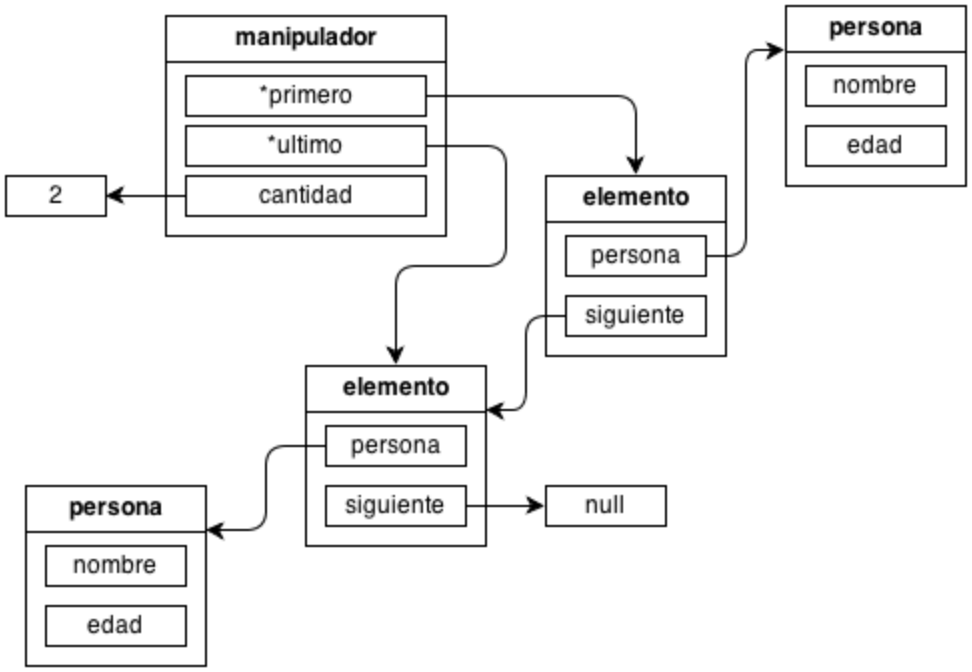

```
elemento->siguiente=nuevo_e;
```

Apunto a Null el campo siguiente del elemento nuevo



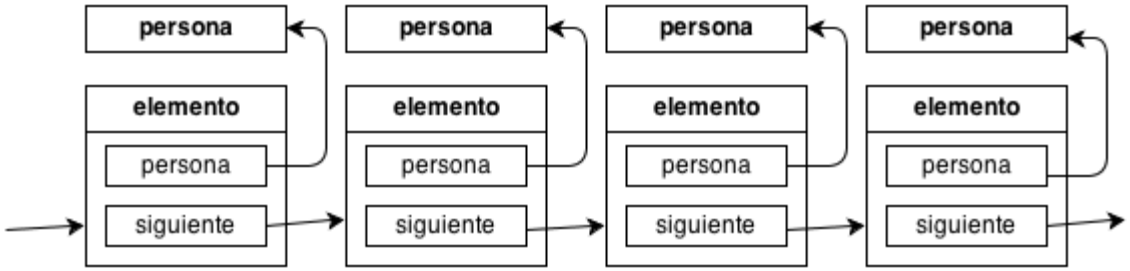

```
nuevo_e->siguiente=Null;
```

Apunto al nuevo elemento el índice correspondiente e incremento el contador



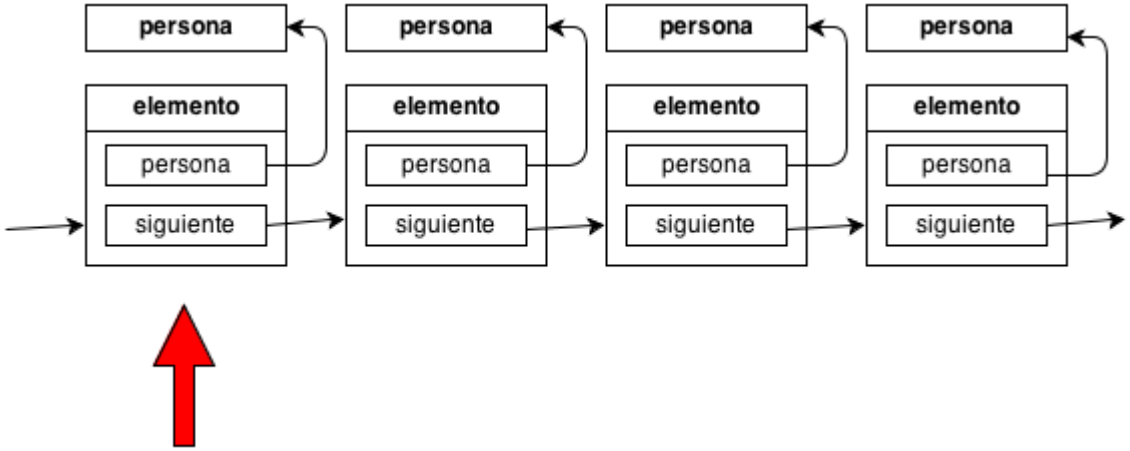

```
manipulador.ultimo=nuevo_e;
manipulador.cantidad++;
```

Borrar un elemento n de la lista



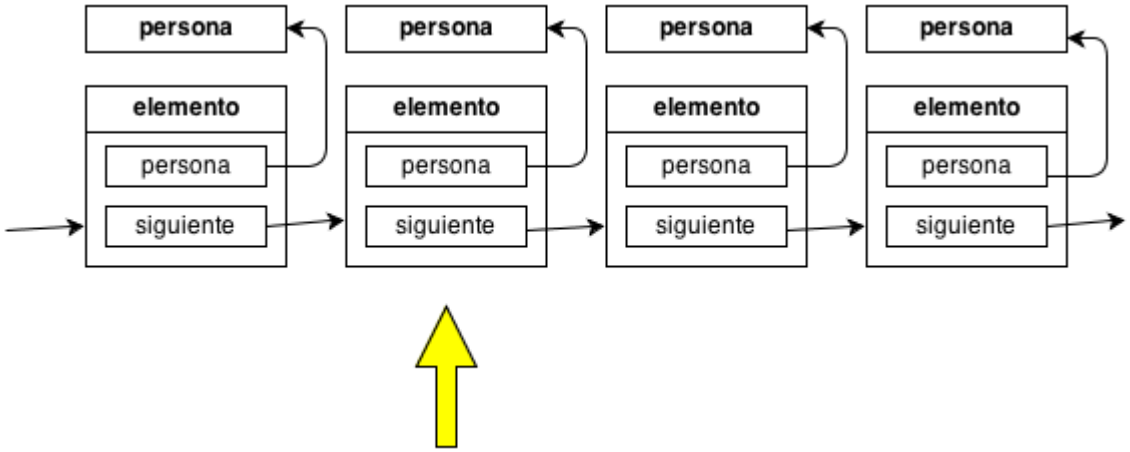
Como solo tengo referencia al primer elemento, para buscar el elemento n a borrar necesito iterar por la lista.

Itero por la lista buscando el elemento a borrar.

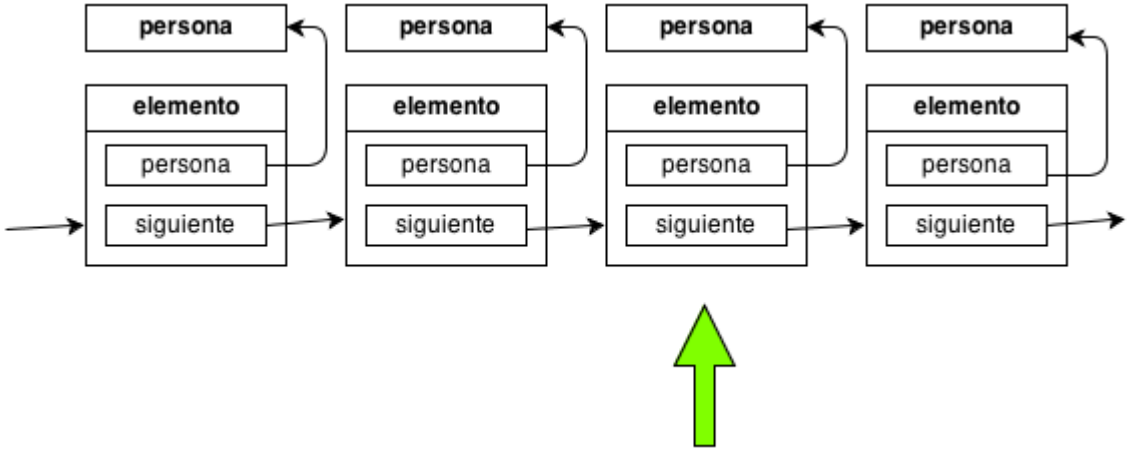



```
t_elemento *elemento=lista->primero;
```

Encuentro el elemento anterior al que quiero eliminar

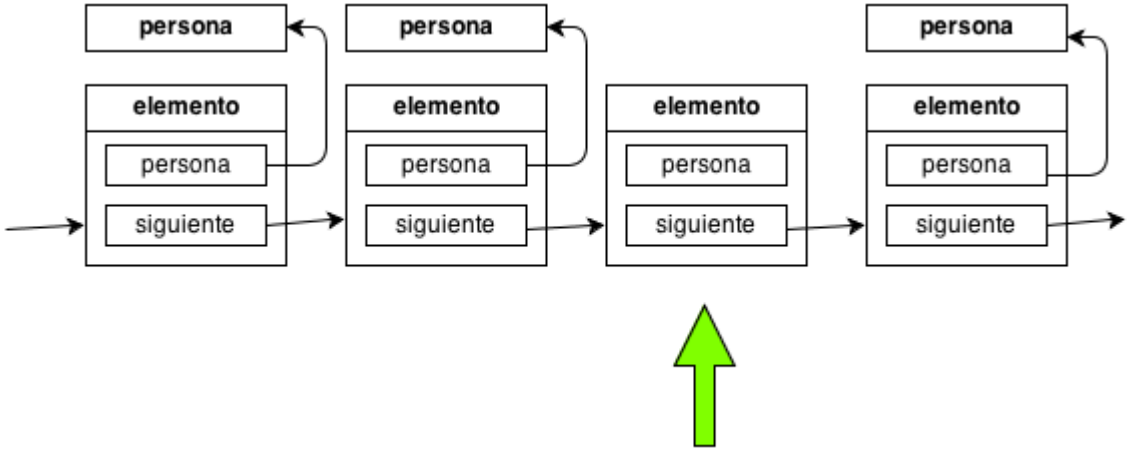



```
unsigned int i;
for ( i=0; i != pos-1; i++ )
{ elemento=elemento->siguiente; }
```



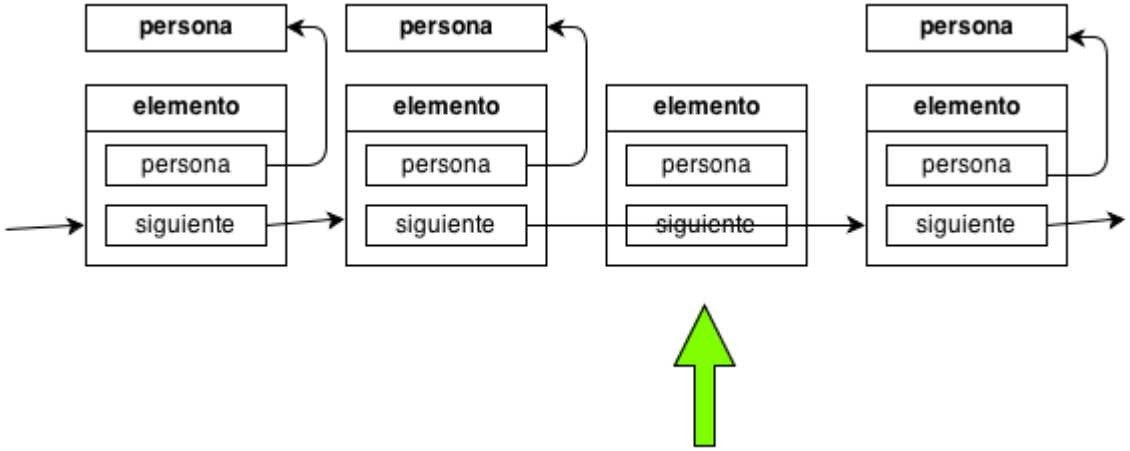

```
t_elemento *aborrar=elemento->siguiente;
```

Elimino los datos referenciados



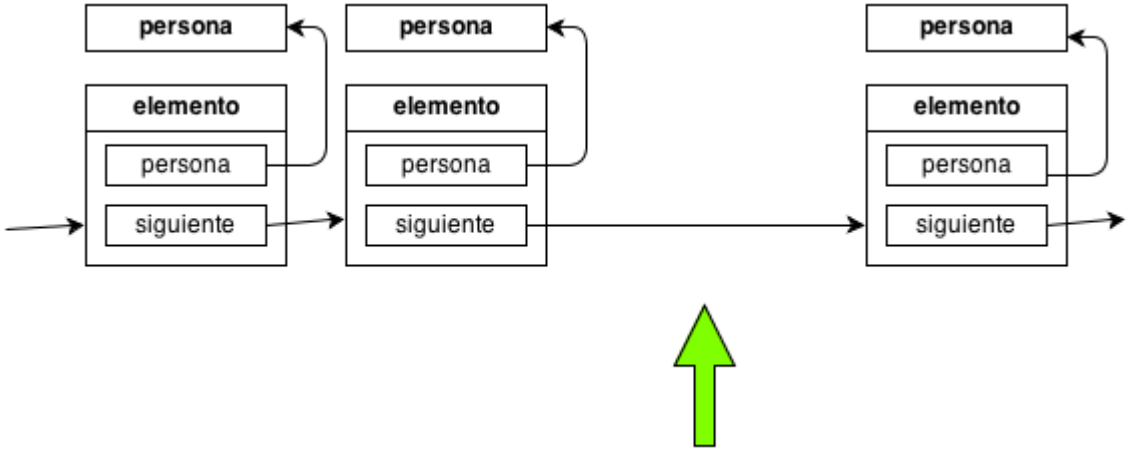

```
free(aborrar->persona);
```

Conecto la lista por fuera del elemento a borrar




```
elemento->siguiente=aborrar->siguiente;
```

Borro el elemento




```
free(aborrar);
```