Estructuras de Datos

Listas

Como ya hemos visto, un array, de 100 elementos por ejemplo, requiere un bloque de memoria contigua equivalente, aproximadamente, a:

```
100 * sizeof(dato)
```

Donde dato corresponde al tipo de dato que se desee almacenar en el arreglo.

Suponiendo que usamos malloc para solicitar nuestro bloque de memoria, el tamaño del mismo sólo se puede modificar (agrandar):

- usando realloc
- pidiendo otro bloque de memoria más grande (con malloc), copiando la información posición por posición y liberando el bloque anterior (con free).

Todo esto, es sumamente costoso. Una alternativa es usar estructuras de datos dinámicas, que nos den flexibilidad para modificar su tamaño. La más sencilla de ellas son las Listas. Comencemos con ella.

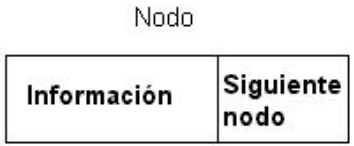
Listas

Una lista enlazada es una estructura dinámica formada por una serie de nodos, conectados entre sí a través de una referencia (puntero), en donde se almacena la información de los elementos de la lista.

Por lo tanto, los nodos de una lista enlazada se componen de dos partes principales:

- la información que queremos guardar.
- un puntero al siguiente elemento.

Se lo suele representar como:



Listas

Una definición de lista enlazada suele ser similar a esta:

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

Como podemos ver, la información a almacenar es un entero (dato) y, el puntero al siguiente elemento (sig).

Suele resultar confusa este tipo de definición así que analicemos por parte la estructura.

Vemos que la definición comienza con la palabra clave typedef. La misma permite definir un alias para un tipo de dato.
Un ejemplo sencillo de su uso sería:

```
typedef int* punteroInt;
punteroInt a = malloc(sizeof(int));
*a = 4;
```

Donde definimos a punteroInt como un puntero a un entero. Luego de esto, lo puedo usar como un tipo de dato ya que, cuando se compile, se reemplaza por el tipo de dato original.

Ahora que sabemos en qué consiste typedef, lo que estamos teniendo es que toda esta estructura termina siendo SNodo.

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

Otra opción de hacer esto sería:

```
struct _SNodo {
  int dato;
  struct _SNodo *sig;
}

typedef struct _SNodo SNodo;
```

Listas

Ya entendimos una parte de la definición, vayamos ahora por analizar el contenido de la estructura.

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

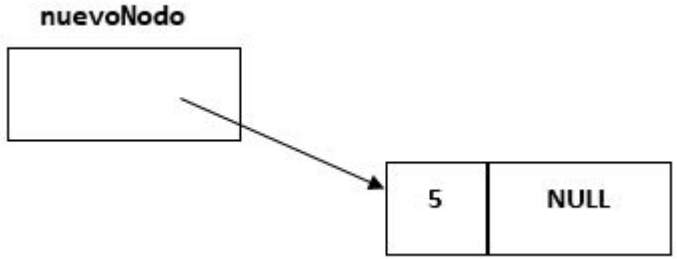
Como podemos ver, la información a almacenar es un entero (dato) y, el puntero al siguiente elemento (sig).
Suele resultar confuso que el tipo de sig es un puntero a la estructura (struct _SNodo) que estoy definiendo pero, sabemos que un puntero es solamente una variable que guarda una dirección de memoria y, que el tipo asociado es usado para que luego se sepa, en la ejecución, qué se supone que hay ahí.

Un ejemplo de su uso es:

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = 5;
nuevoNodo->sig = NULL;
```

Acá estamos pidiendo memoria para un SNodo y, luego almacenando un 5 en el entero y, NULL en el puntero.

Es decir, tendríamos:



Hasta acá tenemos un nodo inicializado pero, ¿cómo hacemos para construir la lista? Tenemos dos opciones:

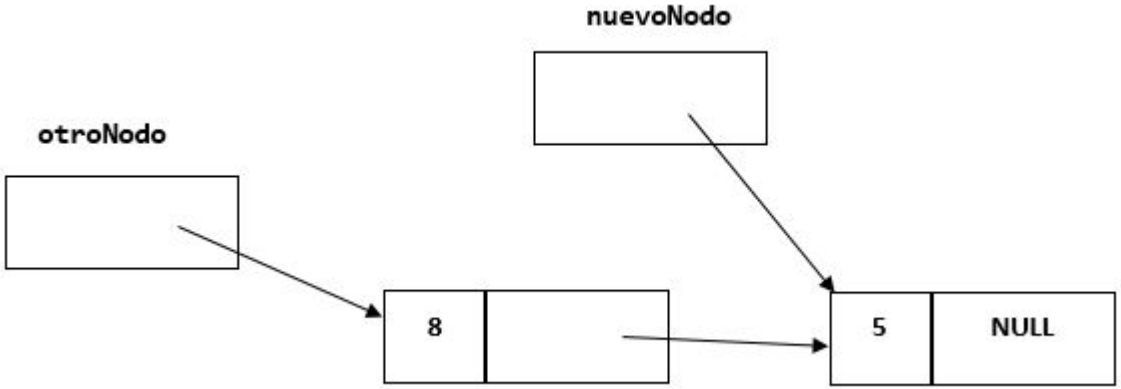
- crear un nuevo nodo inicial y, hacer que su siguiente (sig) apunte al nodo anterior
- ir guardando en el puntero sig del último nodo de la lista, la dirección del siguiente nodo

¿Qué significa crear un nuevo nodo inicial y, hacer que su siguiente apunte al nodo anterior? Vamos a verlo en código:

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = 5;
nuevoNodo->sig = NULL;
SNodo* otroNodo = malloc(sizeof(SNodo));
otroNodo->dato = 8;
otroNodo->sig = nuevoNodo;
```

Es decir, creamos un nuevo nodo, y su sig apunta al anterior inicio de la lista. O sea, estamos agregando al inicio.

En un diagrama, lo que vimos en código, se vería reflejado como:

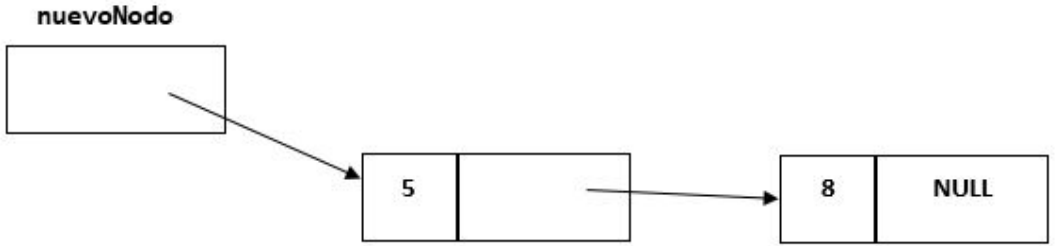


Ahora bien, tenemos que tener cuidado porque se cambia el inicio, es decir, si tenemos una variable que guarda el inicio de la lista, dicha variable debería ser actualizada luego de agregar el nuevo nodo (otroNodo).

¿Qué significa ir guardando en el puntero sig del último nodo de la lista la dirección del siguiente nodo? Vamos a verlo con un ejemplo:

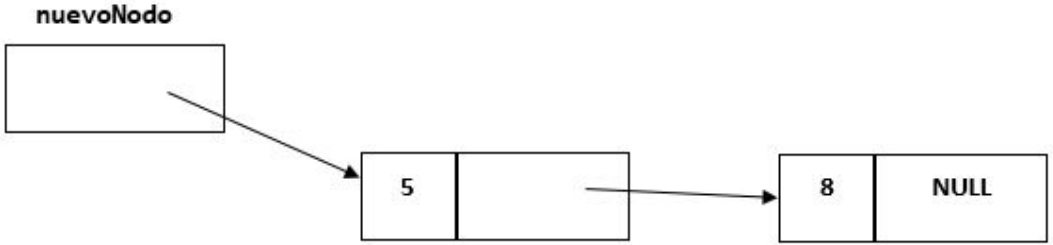
```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = 5;
nuevoNodo->sig = malloc(sizeof(SNodo));
nuevoNodo->sig->dato = 8;
nuevoNodo->sig->sig = NULL;
```

Es decir, en memoria tenemos:



Ahora bien, para eso tenemos que ir hasta el último nodo. Eso nos genera una pregunta, ¿cómo hacemos para determinar si un nodo es el último?
Como podemos suponer, la respuesta resulta simple, es aquel Nodo cuyo atributo sig vale NULL.

Es decir, tenemos que iterar preguntando si el valor de sig es NULL.



Analicemos el siguiente bloque de código:

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = dato;
nuevoNodo->sig = NULL;

if (inicio == NULL) inicio = nuevoNodo;
else {
  SNodo* temp = inicio;
  for(;temp->sig!=NULL; temp=temp->sig);
  temp->sig =  nuevoNodo;
}
```

La primer parte crea un Nodo (nuevoNodo) y, lo inicializa con un valor y con siguiente valiendo NULL.

Este nodo va a ser el último nodo de la lista.

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = dato;
nuevoNodo->sig = NULL;
```

Lo que nos resta es posicionarnos en el último actual y ponerle como siguiente a nuevoNodo.

Lo que puede pasar es que la lista esté vacía, para esto me fijo si el puntero que apunta al inicio de la lista es NULL (inicio). En este caso, simplemente el nuevo nodo creado (nuevoNodo) es el inicio de la lista (inicio).

```
if (inicio == NULL) inicio = nuevoNodo;
```

Si la lista no está vacía, usamos un puntero auxiliar para recorrer la lista (temp).

Nos vamos desplazando nodo por nodo hasta llegar al último nodo. Notemos que el for escrito no tiene sentencias, sino que sólo ejecuta el bloque de incremento hasta que la condición sea falsa.

Al salir del for, estamos posicionados en el último nodo actual de la lista, por lo que a sig le asignamos el nuevoNodo, agregándolo al final de la lista.

```
else {
  SNodo* temp = inicio;
  for(;temp->sig!=NULL; temp=temp->sig);
  temp->sig =  nuevoNodo;
}
```

Estructuras de Datos

Listas

Ahora que ya entendimos las ideas básicas de cómo trabajar con listas enlazadas, pasemos a pensar funciones que nos permitan modularizar el comportamiento.

Comencemos con agregar datos a una lista al inicio.

Recordemos que el ejemplo que habíamos mencionado en la presentación anterior era:

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = 5;
nuevoNodo->sig = NULL;
SNodo* otroNodo = malloc(sizeof(SNodo));
otroNodo->dato = 8;
otroNodo->sig = nuevoNodo;
```

Para poder modularizar y abstraer este comportamiento debemos pensar en cómo sería su uso.

Listas

Ante todo, recordemos la definición de estructura que tenemos.

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

Una lista, no es otra cosa que un puntero al comienzo de la misma, por lo que, para definir una lista hacemos:

```
SNodo* lista = NULL;
```

Con esto tenemos un puntero que va a recordar el comienzo de la lista. Ahora que está definida, podemos pensar en la función que agrega al inicio.

Analicemos el siguiente prototipo de función:

```
void slist_agregar_inicio(SNodo* lista, int dato);
```

Es correcto?

No, no lo es. ¿Por qué? Recordemos que se hace una copia de los argumentos por lo que, al querer modificar lista (lo que va a pasar en cada llamada a esta función) lo vamos a poder hacer dentro de la función pero, al volver al lugar de la llamada la variable tendrá su valor anterior (el que nunca cambió).
Por lo que nuestros prototipos podrían ser:

```
SNodo* slist_agregar_inicio(SNodo* lista, int dato);

void  slist_agregar_inicio(SNodo** lista, int dato);
```

Pensemos un poco, ¿cuál es la diferencia entre ellas?

La primera retorna el nuevo puntero al inicio mientras, que la segunda al agregar una referencia más, podemos modificar el valor de la variable que nos pasan.

```
SNodo* slist_agregar_inicio(SNodo* lista, int dato);

void  slist_agregar_inicio(SNodo** lista, int dato);
```

Vamos a ver como sería el código en cada caso.

Acá podemos ver el código de la función para agregar al comienzo un nodo.
En la primera línea creo un nuevo nodo, luego copio el dato que me pasan en el atributo homónimo y, pongo como siguiente (sig) de este nuevo nodo, el valor que guarda lista (el inicio de la lista hasta el momento); para finalizar, retorno la dirección del nuevoNodo creado.

```
SNodo* slist_agregar_inicio(SNodo* lista, int dato) {
  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = lista;
  return nuevoNodo;
}
```

Un ejemplo del uso de esta función sería, por ejemplo:

```
int main() {
  SNodo* lista = NULL;
  lista = slist_agregar_inicio(lista, 5);
  lista = slist_agregar_inicio(lista, 8);
}
```

Esta otra versión nos muestra que, en lugar de retornar vamos a guardar la dirección del nodo creado (nuevoNodo).

```
void  slist_agregar_inicio(SNodo** lista, int dato) {
  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = *lista;
  *lista = nuevoNodo;
}
```

Notemos que estoy usando *lista para acceder al valor y lugar original de la variable que me están pasando.

Un ejemplo, similar al anterior, ahora sería:

```
int main() {
  SNodo* lista = NULL;
  slist_agregar_inicio(&lista, 5);
  slist_agregar_inicio(&lista, 8);
}
```

Para agregar datos al final de una lista, el código que vimos en el slide anterior era:

```
SNodo* nuevoNodo = malloc(sizeof(SNodo));
nuevoNodo->dato = dato;
nuevoNodo->sig = NULL;

if (inicio == NULL) inicio = nuevoNodo;
else {
  SNodo* temp = inicio;
  for(;temp->sig!=NULL; temp=temp->sig);
  temp->sig =  nuevoNodo;
}
```

escribamos esto, dentro de una función.

Analicemos la siguiente función.

```
SNodo* slist_agregar_final(SNodo* lista, int dato) {

  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = NULL;

  if (lista == NULL) return nuevoNodo;
  else {
    SNodo* temp = lista;
    for(;temp->sig!=NULL; temp=temp->sig);
    temp->sig =  nuevoNodo;
  }
  return lista;
}
```

Podemos ver que creo un nodo (nuevoNodo), en el cual copio el dato que me pasan en el atributo homónimo y, pongo como siguiente (sig) de este nuevo nodo NULL, ya que va a ser el último nodo de la lista.

En el caso que la lista esté vacía, nuevoNodo es el primer y último nodo de la lista por lo que retorno su dirección. Este es el único caso en el que retorno un argumento diferente del valor que se tenía.
En caso contrario, me desplazo por la lista hasta poder agregar nuevoNodo y, retorno el valor con el que lista contaba.

```
SNodo* slist_agregar_final(SNodo* lista, int dato) {

  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = NULL;

  if (lista == NULL) return nuevoNodo;
  else {
    SNodo* temp = lista;
    for(;temp->sig!=NULL; temp=temp->sig);
    temp->sig =  nuevoNodo;
  }
  return lista;
}
```

Esta versión retorna el nuevo valor del puntero, ¿cómo sería la función con retorno void? Acá podemos verla:

```
void slist_agregar_final(SNodo** lista, int dato) {

  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = NULL;

  if (*lista == NULL) *lista = nuevoNodo;
  else {
    SNodo* temp = *lista;
    for(;temp->sig!=NULL; temp=temp->sig);
    temp->sig =  nuevoNodo;
  }
}
```

Parecería que ya finalizamos los casos posibles para agregar un nodo al final de la lista, sin embargo queda una pregunta: ¿se podrá hacer una función recursiva que agregue un nodo al final de la lista?

La respuesta es que sí, podemos ver el código de la función a continuación.

```
SNodo* slist_agregar_finalR(SNodo* lista, int dato) {
  if (lista == NULL) {
    SNodo* nuevoNodo = malloc(sizeof(SNodo));
    nuevoNodo->dato = dato;
    nuevoNodo->sig = NULL;
    return nuevoNodo;
  }
  else {
    lista->sig = slist_agregar_finalR(lista->sig, dato);
    return lista;
  }
}
```

En el caso que lista sea NULL, simplemente se crea un nuevoNodo y lo retorno.
En el caso de no serlo, hago la llamada recursiva con el siguiente elemento pero, tengo que guardar el retorno de esa llamada. ¿Dónde lo guardo? en el mismo argumento que paso para poder construir la lista con el nuevo nodo creado, siguiendo el razonamiento que hicimos anteriormente.

```
SNodo* slist_agregar_finalR(SNodo* lista, int dato) {
  if (lista == NULL) {
    SNodo* nuevoNodo = malloc(sizeof(SNodo));
    nuevoNodo->dato = dato;
    nuevoNodo->sig = NULL;
    return nuevoNodo;
  }
  else {
    lista->sig = slist_agregar_finalR(lista->sig, dato);
    return lista;
  }
}
```

Podemos hacer un par de comentarios sobre este código:

- el else es innecesario ya que tenemos un return dentro del if.
- es innecesario el definir una nueva variable dentro del if, podemos usar a lista, que es NULL, para esto.

```
SNodo* slist_agregar_finalR(SNodo* lista, int dato) {
  if (lista == NULL) {
    SNodo* nuevoNodo = malloc(sizeof(SNodo));
    nuevoNodo->dato = dato;
    nuevoNodo->sig = NULL;
    return nuevoNodo;
  }
  else {
    lista->sig = slist_agregar_finalR(lista->sig, dato);
    return lista;
  }
}
```

Acá tenemos una nueva versión con el segundo punto implementado.

```
SNodo* slist_agregar_finalR(SNodo* lista, int dato) {
  if (lista == NULL) {
    lista = malloc(sizeof(SNodo));
    lista->dato = dato;
    lista->sig = NULL;
  }
  else {
    lista->sig = slist_agregar_finalR(lista->sig, dato);
  }
  return lista;
}
```

Vemos que, al hacer esto, podemos unificar en un único return y, al realizar esto debo dejar el else. La otra opción es poner ambos return sin el else. En mi opinión, esta variante es más clara.

Ahora bien, ¿podemos evitar recorrer toda la lista para llegar hasta el último nodo? No sin modificar la forma en la que estamos viendo las listas, qué pasaría si tenemos estas definiciones:

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;

typedef struct SList {
  SNodo *primero;
  SNodo *ultimo;
} SList;
```

La primera es la misma que teníamos pero, la segunda nos presenta una estructura para representar las listas que nos permite tener dos punteros: uno al primer elemento y, otro al último.
De esta manera podemos acceder tanto al inicio como al final de una lista sin necesidad de recorrerla por completo.

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;

typedef struct SList {
  SNodo *primero;
  SNodo *ultimo;
} SList;
```

Se deja como ejercicio implementar las funciones que agregan al inicio y, al final con estas nuevas estructuras.

También, les dejo una pregunta: ya que tenemos un puntero al inicio y, al final de la lista, ¿por qué recorrerla sólo en un sentido?

Esta pregunta nos va a llevar a desarrollar la idea de Listas Doblemente Enlazadas en la próxima presentación.

Estructuras de Datos

Listas

Retomando las ideas de la presentación anterior, teníamos estas estructuras definidas:

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;

typedef struct SList {
  SNodo *primero;
  SNodo *ultimo;
} SList;
```

Para continuar, vamos a crear nuevas estructuras que nos permitan trabajar con las listas pero, agregándole la posibilidad de recorrerlas en ambos sentidos.

Consideremos las siguientes definiciones:

```
typedef struct _DNodo {
  int dato;
  struct _DNodo* sig;
  struct _DNodo* ant;
} DNodo;

typedef struct {
  DNodo* primero;
  DNodo* ultimo;
} DList;
```

Podemos notar que la única diferencia sustancial (además de los nombres), es que la estructura DNodo contiene otro puntero además de sig: ant.

El objetivo de este puntero es que cada nodo apunte a su inmediato anterior para que, de esta forma, cada nodo conozca a su anterior y a su siguiente en la lista.
Usándolo vamos a poder recorrer la lista desde el último al primero.

```
typedef struct _DNodo {
  int dato;
  struct _DNodo* sig;
  struct _DNodo* ant;
} DNodo;

typedef struct {
  DNodo* primero;
  DNodo* ultimo;
} DList;
```

Pero, antes de hacer eso, tenemos que pensar: ¿cómo adaptamos la función agregar y eliminar que teníamos a esta nueva estructura?

Comencemos con la función agregar y, su versión al inicio. Esto era lo que teníamos.

```
SNodo* slist_agregar_inicio(SNodo* lista, int dato) {
  SNodo* nuevoNodo = malloc(sizeof(SNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = lista;
  return nuevoNodo;
}
```

Lo que nos faltaría en este caso es:

- inicializar el puntero ant a NULL. ¿Por qué NULL? Porque el primer nodo de la lista no tiene un anterior.
- en caso de que ya hubiera un nodo en la lista, el anterior del primer nodo debe ser nuevoNodo.

Con esos cambios, la función quedaría así:

```
DNodo* dlist_agregar_inicio(DNodo* lista, int dato) {
  DNodo* nuevoNodo = malloc(sizeof(DNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = lista;
  nuevoNodo->ant = NULL;
  if (lista!=NULL)
    lista->ant = nuevoNodo;
  return nuevoNodo;
}
```

Ahora, escribamos una versión usando la estructura DList.

Con la estructura, si agrego un nodo al inicio, debo modificar el puntero a primero y, actualizar el puntero al último si este es NULL(eso indicaría que el nodo que acabo de agregar es el primer y último nodo de la lista).

```
DList* dlist_agregar_inicio(DList* lista, int dato) {
  DNodo* nuevoNodo = malloc(sizeof(DNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = lista->primero;
  nuevoNodo->ant = NULL;
  if (lista->primero != NULL)
    lista->primero->ant = nuevoNodo;
  if (lista->ultimo == NULL)
    lista->ultimo = nuevoNodo;
  lista->primero = nuevoNodo;
  return lista;
}
```

Quedan pendientes las funciones para agregar al final usando y, sin usar, la estructura DList.

En el primer caso, voy a tener que iterar o hacer una recursión mientras que en el segundo es muy similar a la función que acabamos de escribir.

Vamos a escribir la versión de DNodo iterando hasta posicionarnos en el último nodo.

```
DNodo* dlist_agregar_final(DNodo* lista, int dato) {

  DNodo* nuevoNodo = malloc(sizeof(DNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = NULL;
  nuevoNodo->ant = NULL;

  if (lista == NULL) return nuevoNodo;
  else {
    SNodo* temp = lista;
    for(;temp->sig!=NULL; temp=temp->sig);
    temp->sig =  nuevoNodo;
    nuevoNodo->ant = temp;
  }
  return lista;
}
```

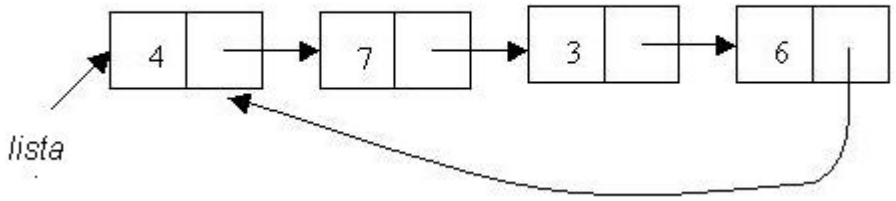
Comparen este código con el de la función slist_agregar_final.

Finalmente, vamos a agregar al final usando la estructura DList:

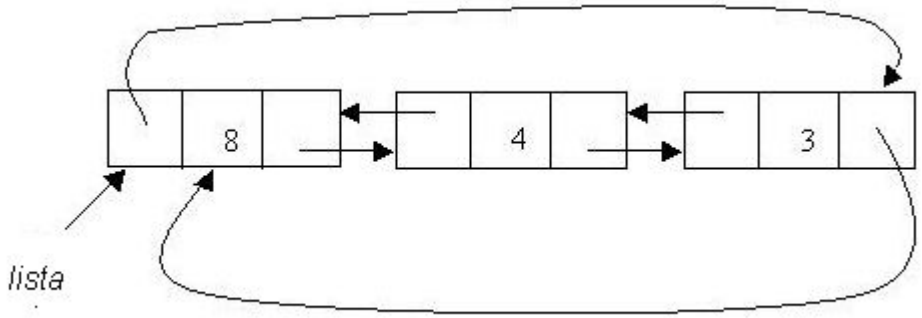
```
DList* dlist_agregar_final(DList* lista, int dato) {
  DNodo* nuevoNodo = malloc(sizeof(DNodo));
  nuevoNodo->dato = dato;
  nuevoNodo->sig = NULL;
  nuevoNodo->ant = lista->ultimo;
  if (lista->ultimo != NULL)
    lista->ultimo->sig = nuevoNodo;
  if (lista->primero == NULL)
    lista->primero = nuevoNodo;
  lista->ultimo = nuevoNodo;
  return lista;
}
```

La última variante que vale la pena mencionar es cuando el siguiente del último elemento de la lista apunta al primero y, en caso de que sea doblemente enlazada, el anterior del primero es el último de la lista; la lista con esta característica se denomina Lista Circular.

Acá podemos ver una representación gráfica de una lista circular simplemente enlazada:



Una doblemente enlazada sería:



Podemos pensar que, en este tipo de lista, no necesitamos otra estructura, además de la definición propia de Nodo, para acceder, sin iterar o usar recursividad, al último nodo de la lista.