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.