Ayudantía 4, parte 1 - Estructura de Datos y Algoritmos ELO320

Estructuras de datos dinámicas

Werner Creixell, Felipe Vera

24 de abril del 2013

En la ayudantía pasada se repasó **memoria dinámica** y **listas simplemente enlazadas**. En la segunda se repasó conceptos de **punteros**. Este conocimiento es fundamental para comprender bien este tema, por lo que recomiendo dominar bien esos conceptos antes de intentar entender el contenido de esta ayudantía.

1. Reglas y consejos

Para idear funciones que modifiquen o usen datos de las estructuras de datos a continuación, se debe tener las siguientes cosas en mente:

1. ¿En alguna circunstancia es necesario que la función modifique el nodo apuntado por el puntero a primer elemento? (o último, si corresponde) Dependiendo de eso se pasa ese puntero por valor (`lista *`) o por referencia (`lista **`).
2. Crear variables temporales en los cuales almacenar punteros a elementos en caso de modificar la estructura de una lista o árbol.
3. Siempre se debe liberar la memoria usada (con la función `free`) al terminar la ejecución del programa.
4. Inicializar SIEMPRE las listas vacías con punteros en `NULL`, de lo contrario los métodos que la recorran hasta el final terminarán en *Segmentation fault*.
5. Muchas veces una recursividad bien pensada puede ser mejor que una iteración (un `for`). Sin embargo hay que pensar bien en como implementarlas, y en el caso de que se ramifiquen (como en la búsqueda de elementos en un árbol binario) sirve de mucho hacer un ruteo con lápiz y papel.
6. Siempre es buena idea hacer un dibujo de qué es lo que debe hacer su algoritmo y cómo procederá, y de acuerdo a ese dibujo escribir el algoritmo.

2. Listas simplemente enlazadas

A pesar de que se repasó conceptos básicos de listas simplemente enlazadas, hay varios conceptos sin repasar.

2.1. Construcción

Consiste en un dato y un puntero hacia el *siguiente* nodo de la lista.

```
struct lista_simple
{
    [datos]
    struct lista *siguiente;
};
```

La lista comienza con un puntero a la cabeza (o primer elemento) de la lista y se inicializa con `NULL`. Cada nodo es declarado como un **puntero a** `struct lista` también.

Regla 4 Se debe inicializar en `NULL`

```
struct lista_simple *cabeza = NULL;
```

Pro-tip Puede escribir un nombre más corto para el tipo de datos de su lista enlazada empleando `typedef`.

```
typedef struct lista_simple lsimple;

lsimple *cabeza = NULL;    //Lo mismo que struct lista_simple *cabeza
[...]
```

2.2. Asignación de memoria

Antes de insertar el nodo en la lista, se le debe asignar la memoria necesaria empleando `malloc`.

```
lsimple *nodo1 = (lsimple*) malloc (sizeof (lsimple));
nodo1->siguiente = NULL;
```

Nota Si el nodo contiene arreglos, también se les debe asignar memoria aparte.

```
typedef struct _lista_simple
{
    char *cadena;    //Cadena de caracteres
    struct _lista_simple *siguiente;
} lsimple2;

lsimple2 *nodo2 = (lsimple*) malloc (sizeof (lsimple));
nodo2->cadena = (char*) malloc (100);    //Almacena 100 bytes
```

Usualmente esto se hace en una función aparte.

2.3. Inserción de un nodo

2.3.1. Al principio

Regla 1 Luego de correr el algoritmo, el puntero a la cabeza de la lista debería apuntar a un nuevo nodo. Por lo tanto **se debe pasar el puntero a la cabeza de la lista por referencia**.

Regla 2 Como se debe cambiar la estructura de la lista, antes de apuntar el puntero a la cabeza, se debe almacenar el *antiguo* primer elemento en una variable temporal para luego enlazarlo como segundo elemento.

```
void
insertarPrincipio (lsimple **cabeza, [datos])
{
    lsimple *oldfirst = (*cabeza);    //Regla 2

    (*cabeza) = asignarMemoria ([datos]);
    (*cabeza)->siguiente = oldfirst;
}
```

2.3.2. Al final

Regla 1 Si la lista está vacía será necesario cambiar el puntero a la cabeza de la lista. Entonces **se debe pasar por referencia**.

```
void
insertarNodo (lsimple **cabeza, [datos])
{
    if((*cabeza) == NULL)
        (*cabeza) = asignarMemoria ([datos]);
    else
        insertarNodo (&((*cabeza)->next), [datos]);
}
```

Como se puede notar, la función `insertarNodo` es recursiva. Se llama a sí misma si el puntero a nodo pasado por referencia `cabeza` no es nulo, derivando al siguiente elemento, y así sucesivamente hasta que ese puntero sea nulo y la recursividad termine, agregando el elemento.

2.4. Eliminación de un nodo

2.4.1. Al principio

Regla 1 Se debe enlazar el puntero a la cabeza de la lista al segundo elemento. Se debe pasar por referencia.

Regla 2 Se debe guardar un puntero al segundo elemento antes de eliminar el primero.

```
void
eliminarPrimero (lsimple **cabeza)
{
    if((*cabeza) == NULL)
        return;    //Sale de la función si la lista está vacía.

    lsimple *segundo = (*cabeza)->next;
    free(*cabeza);
    (*cabeza) = segundo;
}
```

Nota Esta es una lista incompleta de las funciones, pero sirve para dar un ejemplo de cómo buscar, insertar y eliminar nodos en las estructuras de datos. El resto de funciones se implementan de manera similar.

2.5. Eliminar la lista

Simplemente se sigue la base de la función anterior y se hace recursiva. Como no se necesita modificar el enlace al puntero cabeza, esta función no requiere que se pase dicho puntero por referencia, sino **por valor**.

```
void
eliminarLista (lsimple *cabeza)
{
    if(cabeza != NULL)
    {
        lsimple *signodo = cabeza->siguiente;
        free(cabeza);
        eliminarLista(signodo);
    }
}
```

Nota Se ocupa una variable temporal para rescatar el puntero al siguiente elemento, puesto que al momento de eliminar el nodo también se elimina dicho puntero.

3. Listas doblemente enlazadas

La diferencia de estas listas con las simplemente enlazadas es que estas tienen un puntero al elemento anterior aparte del puntero al elemento siguiente. Esto permite que se pueda acceder a toda la lista empleando un puntero a cualquier elemento de ella. Comúnmente se usa un puntero al principio (`head`) y otro al final de la lista (`tail`).

```
struct lista_doble
{
    [datos]
    struct lista_doble *anterior;
    struct lista_doble *siguiente;
}

typedef struct lista_doble ldoble;
```

3.1. Asignar memoria

Se hace de la misma forma que las listas simplemente enlazadas, con la diferencia de que el puntero al elemento anterior también debe inicializarse en `NULL`.

3.2. Insertar elemento

Dependiendo de si se inserta al principio, en la mitad o al final de la lista, se debe reordenar los punteros de los nodos adyacentes.

3.2.1. Al principio

Primero, se debe asegurar de que el puntero `anterior` del primer elemento es nulo. Luego el puntero cabeza se guarda en una variable temporal, y se enlazan tanto el nuevo primer elemento como el antiguo primer elemento según corresponda.

Esta función está diseñada para un puntero a cualquier elemento de la lista.

Comprobaciones Para evitar desagradables *Segmentation faults*, hay que planear cómo evitar acceder a punteros nulos.

1. El puntero a un elemento de la lista (`*cabeza`) podría ser `NULL`. (lista vacía)
2. El puntero a un elemento anterior también podría ser `NULL`.

Dependiendo de la implementación de la función se debe tener cuidado con dichas situaciones. En la mostrada a continuación, a partir de la primera se toman las precauciones necesarias para hacerse cargo de la segunda.

```
void
insertarPrincipio (ldoble **cabeza, [datos])
{
    if((*cabeza) != NULL)
    {
        insertarPrincipio((*cabeza)->anterior, [datos]);
    }
    else
    {
        ldoble *segundo = (*cabeza);
        (*cabeza) = asignarMemoria([datos]);
        (*cabeza)->siguiente = segundo;
        segundo->anterior = (*cabeza);
    }
}
```

3.2.2. Al final

Se sigue un procedimiento similar que insertando al principio, con la diferencia de que la función se hace recursiva dirigiéndose al elemento siguiente, y se intercambian los punteros de enlace a siguiente y anterior elemento según corresponde.

3.3. Eliminar elemento

El procedimiento es:

1. Asegurarse de que está al principio o final de la lista (según corresponda)
2. Avanzar la cantidad de elementos necesaria dentro de la lista si el elemento a eliminar está en medio de esta (cuidado si hay enlaces `NULL`)
3. Liberar la memoria correspondiente al elemento.

3.4. Eliminar lista

Véase *Eliminar lista* simplemente enlazada. El procedimiento es similar, con la diferencia de que hay que asegurarse que comenzar al principio de ella.

4. Buffers circulares

Si se sabe la cantidad máxima de elementos que va a tener una lista, y se necesita un acceso rápido a sus elementos, un Buffer circular es una buena idea. También es conveniente en el caso de que se necesite programar en entornos que no posean la función `malloc`, como en el caso de microcontroladores.

Es simplemente un arreglo que se accede de forma especial. Se debe tener dos variables de apoyo. Una con un indicador al **principio del buffer** circular y otro con el indicador al **final**. Las funciones que manejen el buffer circular también deben recordar el **tamaño** de este.

Los siguientes son implementaciones con arreglos de 50 enteros.

```
#define TAM_BUFFER    50

int principio = 0;
int final = 0;

int buffer1 [TAM_BUFFER];    //Impl. 1

int* buffer = (int*) malloc(TAM_BUFFER * sizeof(int)); //Impl. 2
```

4.1. Agregar un elemento

Dependiendo si es al principio o al final se hace lo siguiente:

4.1.1. Al principio

```
buffer[principio] = [datos];
principio = (principio - 1 + TAM_BUFFER) % TAM_BUFFER;
```

4.1.2. Al final

```
buffer[final] = [datos];
final = (final + 1) % TAM_BUFFER;
```

Sin embargo, esta implementación no tiene medidas de precaución en el caso de que se ingrese un dato con el buffer lleno.

4.2. Obtener el valor de un elemento

Siguiendo la implementación anterior para agregar un elemento...

4.2.1. Al principio

```
dato = buffer[principio + 1];
```

4.2.2. Al final

```
dato = buffer[final - 1];
```

5. Stacks

Un *stack* es simplemente una implementación *LIFO (Last in, First out)* de una lista simplemente enlazada, que sigue dos reglas:

- Sólo se insertan elementos al principio de la lista (*push*).
- Sólo se puede obtener el valor de un elemento sacando el elemento del principio de la lista (*pull* o *pop*).

6. Queues

Un *queue* es una implementación *FIFO (First in, First out)* de una lista doblemente enlazada, que sigue dos reglas:

- Sólo se insertan elementos al final de la lista (*enqueue*).
- Sólo se puede obtener el valor de un elemento sacando el elemento del principio de la lista (*dequeue*).

Ejercicios

1. (2012-2) Haga una función en C que combine dos listas ordenadas ascendentemente en una única lista ordenada ascendentemente. La función debe retornar un puntero a la lista combinada. Su solución **no debe** utilizar memoria adicional para la lista resultante.

   ```
   struct lista_simple
   {
       float dato;
       struct lista_simple *next;
   }

   typedef lista_simple lsimple;

   lsimple *sortedMerge(lsimple *a, lsimple *b);
   ```

2. (2012-1) Haga una función que invierta una lista simplemente enlazada empleando `lsimple` como tipo de dato para ella. Utilice el siguiente prototipo para la función:

   ```
   void reverse (nodo **headref);
   ```

3. LISTAS DOBLEMENTE ENLAZADAS: Cree una función con el siguiente prototipo:

   ```
   struct lista_doble
   {
       float dato;
       struct lista_doble *prev;
       struct lista_doble *next;
   }

   typedef lista_doble ldoble;

   ldoble *insertarElemento(float nuevodato, ldoble *head, ldoble *tail);
   ```

   Esta función debe insertar un elemento al principio de la lista si el *dato* es menor al primer elemento de la lista, o al final si es mayor al último elemento de la lista. Haga que regrese una referencia al nuevo dato ingresado dentro de la lista si se insertó, o `NULL` si no cumplió ninguno de esos criterios. Ejecute las medidas de protección necesarias para evitar *Segmentation faults*.

4. STACK CON LISTAS Y BUFFER CIRCULAR: Implemente una calculadora que reciba comandos en un archivo, de modo que:

   - Si recibe un número, guárdelo en un stack.

- Si recibe un signo (+, -, *, /) realizar la operación que corresponda sacando los dos últimos números del stack e insertando el resultado de la operación

Los comandos están en el archivo *calculadora.txt*. Se sugiere emplear la función `atoi` de la librería `stdlib.h`.

Intente implementar este programa usando listas (decida usted si son simple o doblemente enlazadas) y stacks.