Capítulo 1 Listas abiertas:

1.1 Definición

La forma más simple de estructura dinámica es la lista abierta. En esta forma los nodos se organizan de modo que cada uno apunta al siguiente, y el último no apunta a nada, es decir, el puntero del nodo siguiente vale NULL.

En las listas abiertas existe un nodo especial: el primero. Normalmente diremos que nuestra lista es un puntero a ese primer nodo o llamaremos a ese nodo la cabeza de la lista. Eso es porque mediante ese único puntero podemos acceder a toda la lista.

Cuando el puntero que usamos para acceder a la lista vale NULL, diremos que la lista está vacía.

El nodo típico para construir listas tiene esta forma:

```
struct nodo {
   int dato;
   struct nodo *siguiente;
};
```

En el ejemplo, cada elemento de la lista sólo contiene un dato de tipo entero, pero en la práctica no hay límite en cuanto a la complejidad de los datos a almacenar.

1.2 Declaraciones de tipos para manejar listas en C:

Normalmente se definen varios tipos que facilitan el manejo de las listas, en C, la declaración de tipos puede tener una forma parecida a esta:

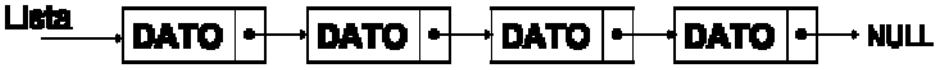
```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
```

tipoNodo es el tipo para declarar nodos, evidentemente.

pNodo es el tipo para declarar punteros a un nodo.

Lista es el tipo para declarar listas, como puede verse, un puntero a un nodo y una lista son la misma cosa. En realidad, cualquier puntero a un nodo es una lista, cuyo primer elemento es el nodo apuntado.



Es muy importante que nuestro programa nunca pierda el valor del puntero al primer elemento, ya que si no existe ninguna copia de ese valor, y se pierde, será imposible acceder al nodo y no podremos liberar el espacio de memoria que ocupa.

1.3 Operaciones básicas con listas:

Con las listas tendremos un pequeño repertorio de operaciones básicas que se pueden realizar:

- Añadir o insertar elementos.
- Buscar o localizar elementos.
- Borrar elementos.
- Moverse a través de una lista, anterior, siguiente, primero.

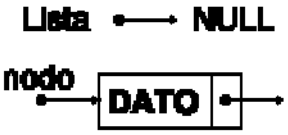
Cada una de estas operaciones tendrá varios casos especiales, por ejemplo, no será lo mismo insertar un nodo en una lista vacía, o al principio de una lista no vacía, o la final, o en una posición intermedia.

1.4 Insertar elementos en una lista abierta:

Veremos primero los casos sencillos y finalmente construiremos un algoritmo genérico para la inserción de elementos en una lista.

Insertar un elemento en una lista vacía:

Este es, evidentemente, el caso más sencillo. Partiremos de que ya tenemos el nodo a insertar y, por supuesto un puntero que apunte a él, además el puntero a la lista valdrá NULL:



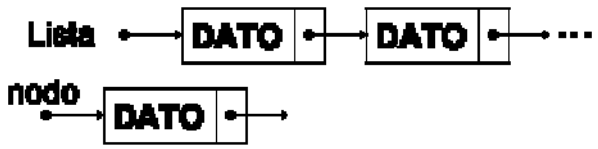
El proceso es muy simple, bastará con que:

1. nodo->siguiente apunte a NULL.
2. Lista apunte a nodo.

Insertar un elemento en la primera posición de una lista:

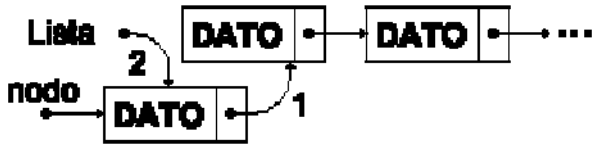
Podemos considerar el caso anterior como un caso particular de éste, la única diferencia es que en el caso anterior la lista es una lista vacía, pero siempre podemos, y debemos considerar una lista vacía como una lista.

De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una lista, en este caso no vacía:



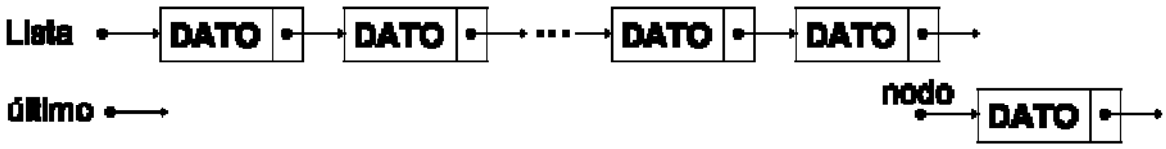
El proceso sigue siendo muy sencillo:

1. Hacemos que nodo->siguiente apunte a Lista.
2. Hacemos que Lista apunte a nodo.



Insertar un elemento en la última posición de una lista:

Este es otro caso especial. Para este caso partiremos de una lista no vacía:

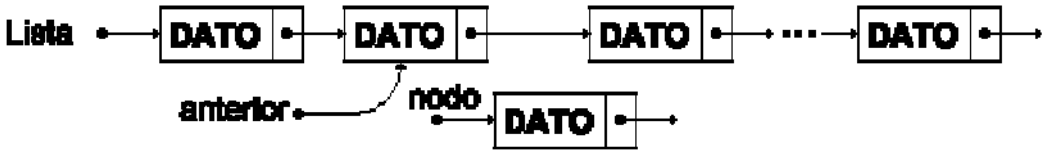


El proceso en este caso tampoco es excesivamente complicado:

1. Necesitamos un puntero que señale al último elemento de la lista. La manera de conseguirlo es empezar por el primero y avanzar hasta que el nodo que tenga como siguiente el valor NULL.
2. Hacer que nodo->siguiente sea NULL.
3. Hacer que ultimo->siguiente sea nodo.

Insertar un elemento a continuación de un nodo cualquiera de una lista:

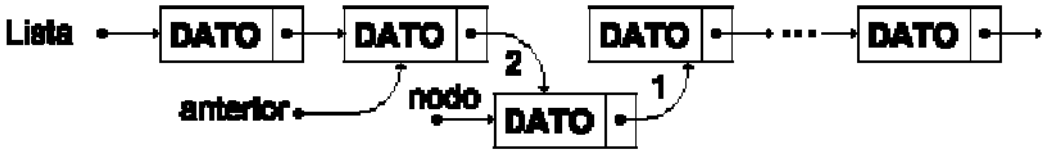
De nuevo podemos considerar el caso anterior como un caso particular de este. Ahora el nodo "anterior" será aquel a continuación del cual insertaremos el nuevo nodo:



Suponemos que ya disponemos del nuevo nodo a insertar, apuntado por nodo, y un puntero al nodo a continuación del que lo insertaremos.

El proceso a seguir será:

1. Hacer que nodo->siguiente señale a anterior->siguiente.
2. Hacer que anterior->siguiente señale a nodo.



1.5 Localizar elementos en una lista abierta:

Muy a menudo necesitaremos recorrer una lista, ya sea buscando un valor particular o un nodo concreto. Las listas abiertas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente, pero no se puede obtener, por ejemplo, un puntero al nodo anterior desde un nodo cualquiera si no se empieza desde el principio.

Para recorrer una lista procederemos siempre del mismo modo, usaremos un puntero auxiliar como índice:

1. Asignamos al puntero índice el valor de Lista.
2. Abriremos un bucle que al menos debe tener una condición, que el índice no sea NULL.
3. Dentro del bucle asignaremos al índice el valor del nodo siguiente al índice actual.

Por ejemplo, para mostrar todos los valores de los nodos de una lista, podemos usar el siguente bucle en C:

```
typedef struct _nodo {
   int dato;
   struct _nodo *siguiente;
} tipoNodo;

typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;
...
pNodo indice;
...
indice = Lista;
while(indice) {
```

```
   printf("%d\n", indice->dato);
   indice = indice->siguiente;
}
...
```

Supongamos que sólo queremos mostrar los valores hasta que encontremos uno que sea mayor que 100, podemos sustituir el bucle por:

```
...
indice = Lista;
while(indice && indice->dato <= 100) {
   printf("%d\n", indice->dato);
   indice = indice->siguiente;
}
...
```

Si analizamos la condición del bucle, tal vez encontremos un posible error: ¿Qué pasaría si ningún valor es mayor que 100, y alcancemos el final de la lista?. Podría pensarse que cuando indice sea NULL, si intentamos acceder a indice->dato se producirá un error.

En general eso será cierto, no puede accederse a punteros nulos. Pero en este caso, ese acceso está dentro de una condición y forma parte de una expresión "and". Recordemos que cuando se evalúa una expresión "and", se comienza por la izquierda, y la evaluación se abandona cuando una de las expresiones resulta falsa, de modo que la expresión "indice->dato <= 100" nunca se evaluará si indice es NULL.

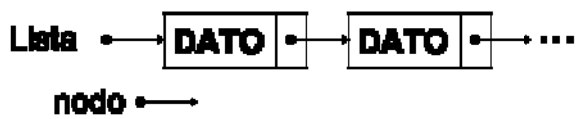
Si hubiéramos escrito la condición al revés, el programa nunca funcionaría bien. Esto es algo muy importante cuando se trabaja con punteros.

1.6 Eliminar elementos en una lista abierta:

De nuevo podemos encontrarnos con varios casos, según la posición del nodo a eliminar.

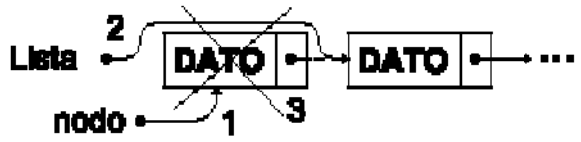
Eliminar el primer nodo de una lista abierta:

Es el caso más simple. Partiremos de una lista con uno o más nodos, y usaremos un puntero auxiliar, nodo:



1. Hacemos que nodo apunte al primer elemento de la lista, es decir a Lista.
2. Asignamos a Lista la dirección del segundo nodo de la lista: Lista->siguiente.
3. Liberamos la memoria asignada al primer nodo, el que queremos eliminar.

Si no guardamos el puntero al primer nodo antes de actualizar Lista, después nos resultaría imposible liberar la memoria que ocupa. Si liberamos la memoria antes de actualizar Lista, perderemos el puntero al segundo nodo.

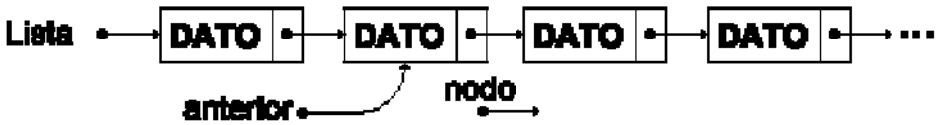


Si la lista sólo tiene un nodo, el proceso es también válido, ya que el valor de Lista->siguiente es NULL, y después de eliminar el primer nodo la lista quedará vacía, y el valor de Lista será NULL.

De hecho, el proceso que se suele usar para borrar listas completas es eliminar el primer nodo hasta que la lista esté vacía.

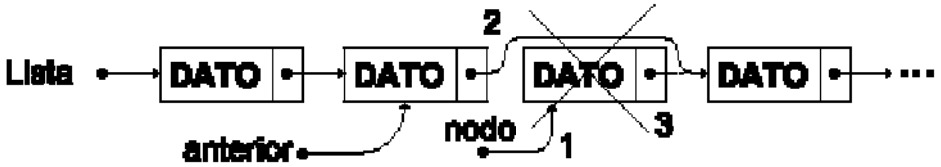
Eliminar un nodo cualquiera de una lista abierta:

En todos los demás casos, eliminar un nodo se puede hacer siempre del mismo modo. Supongamos que tenemos una lista con al menos dos elementos, y un puntero al nodo anterior al que queremos eliminar. Y un puntero auxiliar nodo.



El proceso es parecido al del caso anterior:

1. Hacemos que nodo apunte al nodo que queremos borrar.
2. Ahora, asignamos como nodo siguiente del nodo anterior, el siguiente al que queremos eliminar: anterior->siguiente = nodo->siguiente.
3. Eliminamos la memoria asociada al nodo que queremos eliminar.



Si el nodo a eliminar es el último, es procedimiento es igualmente válido, ya que anterior pasará a ser el último, y anterior->siguiente valdrá NULL.

1.7 Moverse a través de una lista abierta:

Sólo hay un modo de moverse a través de una lista abierta, hacia delante.

Aún así, a veces necesitaremos acceder a determinados elementos de una lista abierta. Veremos ahora como acceder a los más corrientes: el primero, el último, el siguiente y el anterior.

Primer elemento de una lista:

El primer elemento es el más accesible, ya que es a ese a que apunta el puntero que define la lista. Para obtener un puntero al primer elemento bastará con copiar el puntero Lista.

Elemento siguiente a uno cualquiera:

Supongamos que tenemos un puntero nodo que señala a un elemento de una lista. Para obtener un puntero al siguiente bastará con asignarle el campo "siguiente" del nodo, nodo->siguiente.

Elemento anterior a uno cualquiera:

Ya hemos dicho que no es posible retroceder en una lista, de modo que para obtener un puntero al nodo anterior a uno dado tendremos que partir del primero, e ir avanzando hasta que el nodo siguiente sea precisamente nuestro nodo.

Último elemento de una lista:

Para obtener un puntero al último elemento de una lista partiremos de un nodo cualquiera, por ejemplo el primero, y avanzaremos hasta que su nodo siguiente sea NULL.

Saber si una lista está vacía:

Basta con comparar el puntero Lista con NULL, si Lista vale NULL la lista está vacía.

1.8 Borrar una lista completa:

El algoritmo genérico para borrar una lista completa consiste simplemente en borrar el primer elemento sucesivamente mientras la lista no esté vacía.

1.9 Ejemplo de lista abierta ordenada en C:

Supongamos que queremos construir una lista para almacenar números enteros, pero de modo que siempre esté ordenada de menor a mayor. Para hacer la prueba añadiremos los valores 20, 10, 40, 30. De este modo tendremos todos los casos posibles. Al comenzar, el primer elemento se introducirá en una lista vacía, el segundo se insertará en la primera posición, el tercero en la última, y el último en una posición intermedia.

Insertar un elemento en una lista vacía es equivalente a insertarlo en la primera posición. De modo que no incluiremos una función para asignar un elemento en una lista vacía, y haremos que la función para insertar en la primera posición nos sirva para ese caso también.

Algoritmo de inserción:

1. El primer paso es crear un nodo para el dato que vamos a insertar.
2. Si Lista es NULL, o el valor del primer elemento de la lista es mayor que el del nuevo, insertaremos el nuevo nodo en la primera posición de la lista.
3. En caso contrario, buscaremos el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior->siguiente no sea NULL y el dato que contiene anterior->siguiente sea menor o igual que el dato que queremos insertar.
4. Ahora ya tenemos anterior señalando al nodo adecuado, así que insertamos el nuevo nodo a continuación de él.

```
void Insertar(Lista *lista, int v)
{
   pNodo nuevo, anterior;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Si la lista está vacía */
   if(ListaVacia(*lista) || (*lista)->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = *lista;
      /* Ahora, el comienzo de nuestra lista es en nuevo nodo */
      *lista = nuevo;
   }
   else {
      /* Buscar el nodo de valor menor a v */
      anterior = *lista;
      /* Avanzamos hasta el último elemento o hasta que el
         siguiente tenga un valor mayor que v */
      while(anterior->siguiente && anterior->siguiente->valor <=
v)
         anterior = anterior->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior */
      nuevo->siguiente = anterior->siguiente;
      anterior->siguiente = nuevo;
   }
}
```

Algoritmo para borrar un elemento:

Después probaremos la función para buscar y borrar, borraremos los elementos 10, 15, 45, 30 y 40, así probaremos los casos de borrar el primero, el último y un caso intermedio o dos nodos que no existan.

Recordemos que para eliminar un nodo necesitamos disponer de un puntero al nodo anterior.

1. Lo primero será localizar el nodo a eliminar, si es que existe. Pero sin perder el puntero al nodo anterior. Partiremos del nodo primero, y del valor NULL para anterior. Y avanzaremos mientras nodo no sea NULL o mientras que el valor almacenado en nodo sea menor que el que buscamos.
2. Ahora pueden darse tres casos:
3. Que el nodo sea NULL, esto indica que todos los valores almacenados en la lista son menores que el que buscamos y el nodo que buscamos no existe. Retornaremos sin borrar nada.
4. Que el valor almacenado en nodo sea mayor que el que buscamos, en ese caso también retornaremos sin borrar nada, ya que esto indica que el nodo que buscamos no existe.
5. Que el valor almacenado en el nodo sea igual al que buscamos.
6. De nuevo existen dos casos:
7. Que anterior sea NULL. Esto indicaría que el nodo que queremos borrar es el primero, así que modificamos el valor de Lista para que apunte al nodo siguiente al que queremos borrar.
8. Que anterior no sea NULL, el nodo no es el primero, así que asignamos a anterior->siguiente la dirección de nodo->siguiente.
9. Después de 7 u 8, liberamos la memoria de nodo.

```
void Borrar(Lista *lista, int v)
{
   pNodo anterior, nodo;

   nodo = *lista;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
   else { /* Borrar el nodo */
      if(!anterior) /* Primer elemento */
         *lista = nodo->siguiente;
      else  /* un elemento cualquiera */
         anterior->siguiente = nodo->siguiente;
      free(nodo);
   }
}
```

Código del ejemplo completo:

```
#include <stdlib.h>
#include <stdio.h>


typedef struct _nodo {
   int valor;
   struct _nodo *siguiente;
} tipoNodo;


typedef tipoNodo *pNodo;
typedef tipoNodo *Lista;

/* Funciones con listas: */
void Insertar(Lista *l, int v);
```

```
void Borrar(Lista *l, int v);

int ListaVacia(Lista l);

void BorrarLista(Lista *);
void MostrarLista(Lista l);

int main()
{
   Lista lista = NULL;
   pNodo p;

   Insertar(&lista, 20);
   Insertar(&lista, 10);
   Insertar(&lista, 40);
   Insertar(&lista, 30);

   MostrarLista(lista);

   Borrar(&lista, 10);
   Borrar(&lista, 15);
   Borrar(&lista, 45);
   Borrar(&lista, 30);
   Borrar(&lista, 40);

   MostrarLista(lista);

   BorrarLista(&lista);

   system("PAUSE");
   return 0;
}

void Insertar(Lista *lista, int v)
{
   pNodo nuevo, anterior;

   /* Crear un nodo nuevo */
   nuevo = (pNodo)malloc(sizeof(tipoNodo));
   nuevo->valor = v;

   /* Si la lista está vacía */
   if(ListaVacia(*lista) || (*lista)->valor > v) {
      /* Añadimos la lista a continuación del nuevo nodo */
      nuevo->siguiente = *lista;
      /* Ahora, el comienzo de nuestra lista es en nuevo nodo */
      *lista = nuevo;
   }
   else {
      /* Buscar el nodo de valor menor a v */
      anterior = *lista;
      /* Avanzamos hasta el último elemento o hasta que el
         siguiente tenga un valor mayor que v */
      while(anterior->siguiente && anterior->siguiente->valor <=
v)
         anterior = anterior->siguiente;
      /* Insertamos el nuevo nodo después del nodo anterior */
      nuevo->siguiente = anterior->siguiente;
      anterior->siguiente = nuevo;
   }
}
```

```
void Borrar(Lista *lista, int v)
{
   pNodo anterior, nodo;

   nodo = *lista;
   anterior = NULL;
   while(nodo && nodo->valor < v) {
      anterior = nodo;
      nodo = nodo->siguiente;
   }
   if(!nodo || nodo->valor != v) return;
   else { /* Borrar el nodo */
      if(!anterior) /* Primer elemento */
         *lista = nodo->siguiente;
      else  /* un elemento cualquiera */
         anterior->siguiente = nodo->siguiente;
      free(nodo);
   }
}

int ListaVacia(Lista lista)
{
   return (lista == NULL);
}

void BorrarLista(Lista *lista)
{
   pNodo nodo;

   while(*lista) {
      nodo = *lista;
      *lista = nodo->siguiente;
      free(nodo);
   }
}

void MostrarLista(Lista lista)
{
   pNodo nodo = lista;

   if(ListaVacia(lista)) printf("Lista vacía\n");
   else {
      while(nodo) {
         printf("%d -> ", nodo->valor);
         nodo = nodo->siguiente;
     }
     printf("\n");
   }
```