5 Estructuras Enlazadas

Introducción:
En el presente trabajo se incorpora el concepto de asignación dinámica en memoria. Esto permite romper con la limitación de tamaño fijo que proporcionan las tablas cuando es necesario trabajar con colección de datos el mismo tipo en memoria.
Para esto se incorporan los conceptos de estructuras enlazadas, punteros y asignación dinámica en memoria**.**

Estructuras enlazadas vs. estructuras indexadas
Como vimos un arreglo es una secuencia de datos del mismo tipo donde a cada elemento se puede acceder a través de un índice. En esta estructura las posiciones de memoria son contiguas y se las puede ir recorriendo con solo incrementar el índice. En estas estructuras el primer elemento lógico coincide con el primero físico, es decir se comienza desde la primera posición, y el siguiente lógico coincide con el siguiente físico, es decir al próximo elemento se accede incrementando en uno el índice. Estas estructuras, con esta organización son estructuras indexadas. Las estructuras pueden organizarse de forma diferente. El primer elemento lógico puede no coincidir con el primero físico, en ese caso, para conocer donde comienza

lógicamente se debe saber la posición de ese primero. El siguiente elemento lógico no necesariamente debe estar en la posición contigua, para saber donde esta será necesario tener algo que referencie la posición del siguiente elemento. Una estructura con esta organización es una estructura enlazada. Las estructuras enlazadas tienen la particularidad que se necesita conocer la posición del primer elemento y en cada posición se tiene un registro, llamado nodo, con al menos dos campos, uno que contenga la información y otro que indique o referencie donde se encuentra el siguiente que no necesariamente debe ser el contiguo.
Por otro lado vimos que si se quiere definir en memoria una colección de datos del mismo tipo mediante un arreglo se establecer una capacidad máxima. Estas son estructuras con tamaño fijo, el mismo no puede ser modificado en tiempo de ejecución. Existe entonces riesgo que la cantidad de posiciones definidas en tiempo de declaración no alcancen, o, que se utilicen muy pocas de las posiciones definidas haciendo un uso poco eficiente de los recursos.

Estructuras enlazadas con asignación dinámica en memoria
Una estructura con asignación dinámica es un conjunto de elementos de tipo homogéneo donde los mismos no ocupan posiciones contiguas de memoria, pueden aparecer físicamente dispersos manteniendo un orden lógico y son instanciados y/o liberados en tiempo de ejecución.

Operaciones con listas

- Crear una estructura
- Recorrerla, parcial o totalmente
- Agregar o insertar nuevos elementos o nodos en distintos lugres y con criterio variado.
- Localizar un elemento en particular en la estructura
- Borrar un elemento de posicion particular o previamente desconocida

La mayor facilidad de las listas es el "enlace dinámico" a través de punteros que permiten intercalar o eliminar valores sin un movimiento masivo de memoria, con solo pedir memoria o liberarla y hacer simplemente los enlaces de los nodos involucraos.

Punteros

Los punteros son particularmente Tipos de datos. Para la definición precisa de un tipo de dato se debe definir dos conjuntos, el conjunto de valores que ese dato puede contener y el conjunto de operaciones que se puede hacer con ese dato. Pongamos por ejemplo los enteros, lo valores que puede tener son 2,3,4…. Y las operaciones +,-, *,/ ente otras. En el caso de los punteros, los valores que contienen son direcciones de memoria y las operaciones son operaciones de asignacion

Es posible definir estructuras de datos complejas cuyo tamaño puede cambiar en tiempo de ejecución, por lo que son denominadas *estructuras de datos dinámicas*. Estas se instancian a través de punteros.

Los punteros son tipos de datos, en el caso de C/C++ se disponen de dos operadores el operador de dirección (&) y el operador de indireccion o de desreferenciacion

Se se tiene int a = 10, *p = &a

Nombre	tipo	valor
a	entero	10
p	puntero	la dirección de memoria donde se encuentra a
*p	entero	valor contenido en la dirección que apunta p

Mediante la utilización de array se puede definir una secuencia de valores como:

```
Tipo Vector[LongitudMaxima];
```

Con esta declaración se reserva espacio fisico para una cantidad fija de elementos. Esta cantidad esta dada por el valor de LongitudMaxima.
Otra forma de definir una secuencia de valores seria por inducción o recursion:

La secuencia vacía es una secuencia

Un elemento concatenado a una secuencia es otra secuencia

De esta forma se declara el tipo sin especificar ninguna capacidad máxima, se define en forma recursiva.
En esta definición es necesario que cada elemento referencie con precisión el siguiente elemento si es que este existe.
Una secuencia, con esta organización, es una concatenación de registros llamados nodos de dos campos: uno para la información y otro para un indicador al siguiente elemento. El siguiente elemento al que se referencia es un registro del mismo tipo del actual, por referenciar a una estructura de su mismo tipo estas estructuras se conocen como estructuras autoreferenciadas.

Para la asignación dinámica en memoria, los elementos se enlazan a través de un determinado tipo de dato que disponen la mayoría de lenguajes de programación y que se denominan *puntero* o *apuntador*, cuyos valores son referencias o punteros a variables de un tipo. Por ejemlo, en el caso de C

```
int* p;
```

Se hace una definición de un tipo de dato con la particularidad que tendrá como valor una dirección de memoria. Se dirá entonces que es un puntero al tipo de dato o que apunta a determinado tipo de dato. P es entonces un apuntador a un entero, tendrá la dirección de memoria en la que se aloja un dato entero
Los posibles valores de p no serán enteros, sino referencias a enteros; esto sería la dirección del espacio de memoria en la que se almacena el entero.
Ya se definió *variable* como un identificador que denota un espacio de memoria en el que se almacena un valor del tipo asociado en la dirección.

Un valor apuntador es la dirección de una variable "anónima", se le asigna la dirección o se crea en tiempo de ejecución.

En el caso de C
int a = 10;//declaración de a con valor 10
int *p // declaración de p como puntero a entero.
A p se le puede asignar:

1. p = &a, por lo que p apuntara a la dirección de a, en este caso *p será el valor contenido en la dirección a la que apunta p que como apunta a 'a' su valor será 10
2. p = null; se le asigna a p el valor nulo, por lo que no tendrá contenido
3. p = new int() se crea en tiempo de ejecución un unevo espacio de memoria al que apunta p. *p es el acceso a ese espacio de memoria, instanciado por p que no tiene nombre propio como cualquier variable por lo que se llama variable anónima, solo se accede a ella a través del puntero, en este caso p, por el que fue creada.

También un puntero puede apuntar a una estructura
TipoRegistro* pr =&R;
Pr en este caso es un puntero a un registro al que se le asigno la dirección de R, por lo que *pr es la estructura que esta en esa dirección, es decir un redistro.
pr es un puntero a un registro
*pr es el registro R, al que apunta pr
(*pr).c1 es el campo c1 del registro R al que apunta pr, esto también puede escribirse pr->c1. Por tanto el operador -> "flecha" es el operador a un miembro de un rehistro apuntado por un puntero,
Desde luego que los registros también pueden ser creados en tiempos de ejecución pr = new Registro(); es la forma de crear esa instancia.

Para indicar que una variable puntero no referencia o apunta a ninguna a otra variable se le debe asignar el valor NULO.

```
struct Nodo{
   int info;
   Nodo* sgte
}
```
Define un nodo con la información, un int, y la referencia al siguiente que es un puntero a la siguiente dirección de memoria de la secuencia definida.
Se trata de una definición recursiva o autorreferenciada siguiendo la definición inductiva de secuencia.
Una estructura enlazada utilizando punteros debe tener una variable que controle la estructura, en general debe contener la dirección de memoria del primer nodo de la misma, que debe ser de tipo puntero e inicialmente debe ser creada haciendo apuntar a la misma al valor NULO, lo que indica que la secuencia esta vacía, a esto llamamos crear la esructura.
Con este tipo de estructura no es necesario especificar una capacidad máxima en la declaración, sino que se asigna espacio en tiempo de ejecución, conforme se necesita, a través de los apuntadores, por lo que se obtienen dos importantes beneficios.

- No se desaprovecha espacio cuando la secuencia tiene menos elementos que el tamaño máximo establecido.
- El tamaño de la secuencia no está limitado (salvo los limites de la memoria disponible).

Solo se podría mencionar como desventaja el hecho que cada elemento adema de tener la información propia debe destinar recursos para referenciar al siguiente, situación que puede estar a favor de los arreglos. Con las particularidades de cada estructura es responsabilidad del programador seleccionar la más idónea para cada caso que deba resolver.
Como ya mencionamos estas estructuras de datos cuyo tamaño puede variar en tiempo de ejecución se denominan *estructuras de datos dinámicas*, que son básicas en programación, como: *listas*, *pilas*, *colas* y *árboles binarios de búsqueda y grafos.*

El tipo de dato *Puntero*

Se ha definido un tipo de datos puntero como aquel cuyo dominio de valores está formado por referencias o punteros a variables (direcciones de memoria) de un tipo existente.

Si los tipos de datos se definen por sus valores y sus operaciones los valores que pueden tener los punteros son los de direcciones de memoria o el valor NULO en caso de no estar apuntando a una dirección definida y, entre las operaciones se pueden mencionar la asignación interna, la creación o la destrucción, los lenguajes de programación ofrecen los mecanismos para estas operaciones.

Constantes y operaciones con punteros

No es posible el uso de literales para representar valores de tipo puntero salvo las constantes simbólicas que representan el valor NULO con sintaxis diferente según el lenguaje de programación que se trate. NULO representa solo eso y no contiene un valor de dirección de memoria válido. Esta constante NULO es compatible de asignación y comparación con cualquier tipo de dato apuntador, con independencia a cualquiera sea el tipo de valores a los que apunta éste.

Los valores de tipo apuntador son datos internos propios de cada ejecución del programa, solo aceptan asignación interna, no es correcto hacer asignaciones externas leer o escribir con variables de tipo apuntador.

Las operaciones elementales de una variable de tipo apuntador son las de *creación* de un nuevo objeto y *destrucción* del objeto apuntado.

Para la creación de un nuevo objeto apuntado dispondremos de la acción Nuevo (new en C++). Esta acción tiene un único parámetro de tipo Apuntador a T, cualquiera sea el tipo de T, y su efecto es el siguiente:

Reserva espacio de memoria suficiente para alojar una variable de tipo T

Asigna al el valor de la dirección de memoria que acaba de reservar, si es que se dispone de espacio en memoria o el valor NULO en caso de no poder ejecutar la acción de creación.

El efecto de invocar a la acción p = new tipo(), sería la creación de una nueva variable anónima del tipo al que apunta el puntero, y la asignación a pe de la dirección de memoria donde se encuentra dicha variable. La nueva variable se denomina anónima porque no tiene identificador propio, y únicamente será accesible a través de la dirección que la acción new ha asignado al puntero. La nueva variable está recién creada y, por tanto, sin inicializar; no tiene un valor definido mientras no se le asigne nada.

Aunque los lenguajes de programación tienden a aliviar al programador de la responsabilidad de liberar el espacio reservado a variables de tipo apuntador e incorporan rutinas de *recolección de basuras* para liberar los espacios de memoria reservados que ya no están siendo utilizados, de cualquier modo es erróneo el realizar reservas de espacios de memoria que posteriormente queden reservados e inaccesibles por la pérdida de valor de la dirección donde se encuentran.

Cuando un espacio de memoria que ha sido reservado con la acción Nuevo no es ya necesario, podemos *destruir(delete en C++)* las variables en él contenidas y liberar ese espacio para que quede disponible para futuras reservas. Para ello, utilizaremos la acción delete, que ejecuta la operación inversa de Nuevo: recibe como parámetro una variable de tipo apuntador que contiene una dirección válida que previamente ha sido reservada con la acción new, y libera el espacio de memoria al que apunta el puntero, es decir libera la instancia. Cuando programemos, no debemos suponer que la acción Destruir modifica el valor del parámetro que recibe, sino que una buena práctica es asignar el valor NULO a una variable de tipo apuntador tras invocar Destruir.

delete no modifica el valor del puntero, este tiene un valor de dirección de memoria que no es válida, pues ha sido liberada. Se dice que queda como un *apuntador colgante*, ya que no apunta a nada, o mejor dicho, apunta a una dirección que ya no es válida, pues ha sido liberada. Es responsabilidad del programador el mantener de manera conveniente los valores de los apuntadores y no acceder a direcciones que no son válidas.

Acceso a datos mediante apuntadores

Los lenguajes de programación como C/C++ utilizan el operador * en notación prefija (*Puntero) para referenciar al objeto apuntado por el puntero. La expresión *Puntero hará referencia al contenido de la dirección apuntada por el puntero y será un identificador de la variable de tipo TipoBase a que se refiere el apuntador puntero.

Respecto al tema de la asignación debe diferenciarse si lo que se asigna es el contenido o lo que se asigna es la dirección. Si p y q son dos variables de tipo puntero que apuntan al mismo tipo base hay que diferenciar que es lo que ocurre con las asignaciones siguientes:

```
*p = *q
```

En este caso se le asigna al contenido de p el contenido de q. P y Q siguen apuntando a su propia dirección de memoria, i los contenidos de esas dos direcciones es lo que se iguala.

Por otro lado la asignación:

```
p = q
Hace que p apunte a la dirección apuntada por q.
```

En el caso en que el tipo base de un determinado tipo apuntador no sea un tipo simple, sino un tipo estructurado, el acceso se realiza con los mismos operadores de acceso que los datos estáticos. Se agrega, como se vio el operador flecha ->

Algunos lenguajes de programación, como C/C++ permiten utilizar apuntadores para referenciar variables no anónimas declaradas en el léxico. Para ello disponen de un operador dirección_de (el operador & en el caso del lenguaje de programación C) que dada una variable, devuelve su dirección.

Obviamente, cuando una variable de tipo apuntador haga referencia a una variable declarada en el léxico, como en el caso citado, no será posible destruir su contenido mediante la acción Destruir, ya que solo pueden destruirse las instancias que se crean en tiempo de ejecución con la acción Nuevo.

Tipos de datos autorreferenciados o recursivos

Hemos mostrado que con punteros se pueden definir tipos que se autorreferencian, lo que permite que el tamaño de una estructura enlazada sólo esté limitado por la disponibilidad de memoria principal. Con una única variable de tipo puntero será posible referenciar a una secuencia cuyo número de elementos podrá crecer o disminuir en tiempo de ejecución.

En tipo de dato recursivo o autorreferenciado es un registro, llamado nodo, que contiene entre sus elementos al menos dos campos, uno con la información y el otro es un puntero a una estructura del mismo tipo. La bibliografía denomina nodo para referir a ese elemento ya que es el término utilizado habitualmente para referirse a los elementos de una estructura dinámica.

```
Struct Nodo{
 int info;
 Nodo* sgte;]
```

Cada vez que reservemos espacio para una variable se crea una nueva variable de tipo Nodo. Es posible construir sucesivamente una secuencia indefinidamente larga de datos de tipo Nodo, en la que el campo sig de cada uno de ellos apunta al siguiente elemento de tipo Nodo creado. El último elemento de tipo Nodo contiene en su campo sig el valor NULO para indicar que no existen más elementos siguientes.

El tipo Nodo es, por tanto, un tipo de dato *autorreferenciado*.

La siguiente función CrearSecuencia lee de la entrada n valores de tipo entero, crea una secuencia con esos valores en el orden inverso de entrada y devuelve como resultado un apuntador al primer elemento de la secuencia. La variable r se utiliza para apuntar al nodo creado en cada paso y p apunta

al primer elemento de la secuencia. Cada nuevo nodo se añade delante del primero.

```
Nodo* p, q, r;
int i, v;
ALGORITMO
p = null //Crea la estructura
cin>>n //Leer(n);
for (i=0; i<N; i++){
 cin>> v; // leer(v);
 r = new nodo(); // instancia y pide memoria
 r->info = v;
 r->sgte = p;
 p ← r;
}
```

Guarda en el registro apuntado por r el valor v y en el campo siguiente el puntero p, este puntero contiene la direccion del que estaba en primer lugar por lo que se establece como siguiente el que era primero antes.

El ámbito de una variable local es la acción o función en la que se declara. En cambio, aun cuando todas las variables apuntador usadas en la función anterior, es decir, p y r, son parte de su léxico local y, por tanto, desaparecen tras concluir la llamada a ésta. No sucede así con las variables dinámicas creadas a través de ellas con las llamadas a Nuevo. La secuencia creada por la función sería accesible fuera de ella a través del valor TPNodo que retorna.

```
Escribir(q↑.dato);
q ← q↑.sig
FIN_MIENTRAS
FIN
```

Estructuras de datos dinámicas lineales

Dependiendo del número de punteros y de las relaciones entre nodos, podemos distinguir varios tipos de estructuras dinámicas

El tipo pila

Una *pila* es una colección de elementos de un mismo tipo, posiblemente vacía, sobre la que podemos hacer operaciones de inserción de un nuevo elemento, eliminación de un elemento. Una pila es una estructura de tipo *LIFO* (del inglés, ***L**ast-**I**nput, **F**irst-**O**utput*), lo cual significa que los elementos siempre serán eliminados de ella en orden inverso al que fueron colocados, de modo que el último elemento en entrar será el primero en salir. A este tipo de colección de datos se la conoce por el nombre de "pila" precisamente por esta característica. A la hora de retirar elementos, únicamente podremos tomar el que está en la *cima* de la pila, que será el último que fue apilado.

Para definir el tipo de pila debemos, por tanto, diseñar las siguientes operaciones:

PilaVacia: → *Pila*
EsPilaVacia: Pila → *Booleano*
Push : Pila X TipoBase → *Pila*
Pop : Pila → *Pila*

Esta lista de operaciones disponibles para un tipo se conoce con el nombre de *signatura*, y en ella se establecen los nombres de las operaciones y el número y tipo de parámetros de cada una de ellas. En la signatura de un cierto tipo T se distinguen tres tipos de operaciones:

- Operaciones *constructoras*: son aquellas en las que el tipo T aparece como resultado devuelto, pero no como parámetro de la operación.
- Operaciones *modificadoras*: son aquellas en las que el tipo T aparece tanto en la lista de parámetros como en el resultado devuelto.
- Operaciones *de consulta*: son aquellas en las que el tipo T aparece únicamente en la lista de parámetros.

En el caso del tipo pila anterior, la única operación constructora sería *PilaVacia*; las operaciones *Push*, *Pop* serían modificadoras, y las operaciones *EsPilaVacia* sería de consulta. La operación *Pop* tendrá como

precondición que la pila que reciben como parámetro no sea vacía, pues para una pila vacía no estaría definida su *cima* ni se podría desapilar.

Es importante que se sepa que cuando se definen tipos de datos basados en estructuras de datos complejas, las operaciones usuales de igualdad o desigualdad, que se realizan por defecto mediante los operadores booleanos = y ≠, producirían resultados que no se corresponden con la semántica del tipo implementado. Por ejemplo, suponga que declaramos dos variables, p1 y p2 de tipo PilaCars, y efectuamos una comparación como la siguiente:

```
SI p1 = p2 ENTONCES
...
FIN_SI;
```

Lo que estamos comparando en realidad no son las pilas, es decir, si constan de los mismos elementos y están en idéntico orden, sino los apuntadores p1 y p2, puesto que ambas variables son de tipo PilaCars el cual, a su vez, es un tipo apuntador. La comparación p1 = p2 devolverá Verdadero si y sólo si los apuntadores p1 y p2 apuntan a la misma dirección de memoria.

Insertar elemento en una pila:

Apilar en una pila vacía:

Partiendo que ya tiene el nodo a insertar y un puntero que apunte a él, además el puntero a la pila valdrá NULO:

El proceso es:

1. Nodo->siguiente apunte a NULO
2. Pila apunte a nodo.

Meter en una pila no vacía:

Se puede considerar el caso anterior como un caso particular de éste, la única diferencia es el siguiente será pila. Se puede utilizar este procedimiento para ambos casos.

El proceso sigue siendo muy sencillo:

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

Desapilar: leer y eliminar un elemento
Sólo existe un caso posible, en las pilas sólo se puede leer y sacar desde le extremo de la pila.

- Se utiliza un puntero auxiliar.
- Se hace apuntar al auxiliar al primer elemento de la pila, es decir a Pila.
- Se avanza con la pila, asignando a Pila la dirección del segundo nodo de la Pila = Pila->siguiente.
- Se conserva el contenido del nodo para devolverlo como retorno. La operación sacar equivale a leer y borrar.
- Se libera la memoria asignada al auxiliar, que es el nodo a liberar.
- Si la pila sólo tiene un nodo, el proceso sigue siendo válido, ya que el valor de Pila->siguiente es NULO, y después de eliminar el último nodo la pila quedará vacía, y el valor de Pila será NULO.

Algoritmo de la función "push" Apilar:

- Se Crea un nodo para el valor que se colocara en la pila.
- Se hace que nodo->siguiente apunte a Pila.
- Pila debe apuntar al nuevo nodo.

Algoritmo de la función "pop" Desapilar:

- Se hace que nodo apunte al primer elemento de la pila, es decir a Pila.
- Se asigna a Pila la dirección del segundo nodo de la pila: Pila→siguiente.
- Se guarda el contenido del nodo retornarlo.
- Se libera la memoria asignada a nodo.

El tipo cola

El tipo *cola* difiere del tipo *pila* únicamente en la forma de inserción y extracción de elementos. Una cola es una estructurade tipo *FIFO* (del inglés, ***F**irst-**I**nput*, ***F**irst-**O**utput*), es decir, primero en entrar, primero en salir. Se conoce con el nombre de cola por ser la que habitualmente se utiliza en las colas de la vida cotidiana: el primero que llega (que entra en la cola) es el primero en ser atendido (en ser eliminado de la cola).

```
TPNodoColaCars = TIPO Apuntador a NodoColaCars;
NodoColaCars = TIPO < dato: carácter; sig TPNodoColaCars >;
ColaCars = TPNodoColaCars;
```

Las operaciones propias del tipo cola son las siguientes:

ColaVacia : → Cola
EsColaVacia : Cola → Booleano
Agregar : Cola X TipoBase → Cola
Suprimir : Cola → Cola

Operaciones básicas con colas

Por las restricciones a esta estructura tienen muy pocas operaciones disponibles. Las colas sólo permiten añadir y leer elementos:

- Añadir: Inserta un elemento al final de la cola.
- Leer: Lee y elimina un elemento del principio de la cola.

Añadir un elemento Encolar:

Las operaciones con colas son muy sencillas, prácticamente no hay casos especiales, salvo que la cola esté vacía.

Añadir elemento en una cola vacía:

Se parte de tener el nodo a insertar y un puntero que apunte a él, además los punteros que definen la cola, primero y ultimo que valdrán NULO
El proceso es muy simple, bastará con que:

- Nodo->siguiente apunte a NULO.

- Y que los punteros primero y último apunten a nodo.

Añadir elemento en una cola no vacía:
De nuevo partiremos de un nodo a insertar, con un puntero que apunte a él, y de una cola, en este caso, al no estar vacía, los punteros primero y último no serán nulos:
El proceso sigue siendo muy sencillo:

- Nodo->siguiente apunte a NULO.
- Ultimo->siguiente apunte a nodo.
- Y se actualiza último, haciendo que apunte a nodo.

Añadir elemento en una cola, caso general:
Para generalizar el caso anterior, sólo necesitamos añadir una operación:
Siempre nodo->siguiente apunta a NULO.
Si ultimo no es NULO, entonces ultimo->siguiente apunta a nodo.
Si no el primero también es NULO, significa que la cola estaba vacía, así que primero también apunta a nodo
Y se actualiza último, haciendo que apunte a nodo.

Leer un elemento de una cola Eliminar primero:
Ahora también existen dos casos, que la cola tenga un solo elemento o que tenga más de uno.

Leer un elemento en una cola con más de un elemento:

- Se utilaza puntero auxiliar:
- Se hace que auxiliar apunte al primer elemento de la cola, es decir a primero.
- Se asigna a primero la dirección del segundo nodo de la cola: primero=primero->siguiente.
- Se guarda el contenido del auxiliar para devolverlo, la operación de lectura en colas implican también borrar.
- Se Libera la memoria asignada a nodo.

Leer un elemento en una cola con un solo elemento:

- También se requiere un puntero auxiliar:
- Se Hacemos que auxiliar apunte al primer elemento de la cola.
- Se le asigna NULO
- Se guarda el contenido del auxiliar para retornarlo.
- Se libera la memoria asignada al primer nodo, el que queremos eliminar.
- El último debe apuntar a NULO.

El tipo lista

El tipo lista es el caso más general de estructura de datos lineal. No existe una forma prefijada de inserción y extracción de elementos, de modo que éstos pueden ser insertados en, y también eliminados de, cualquier posición arbitraria.

Listas simplemente enlazadas

Con las listas existe un repertorio más amplio 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.

Insertar un elemento en una lista vacía:

El proceso es muy simple

1 Crear el nodo	Nuevo(Nodo)
2 Guardar la información	Nodo^←<Valor, NULO>
3 nodo^.siguiente apunte a NULO	
4 Lista apunte a nodo	Lista ← Nodo

Insertar un elemento en la primera posición de una lista:
El proceso sigue siendo muy sencillo:

- Crear el Nodo
- Guardar la información
- Nodo^.siguiente apunte a Lista, que contiene la dirección del anterior primero.
- Lista debe apuntar al nuevo nodo.

Insertar un elemento en la última posición de una lista:
Se supone una lista no vacía, el proceso en este caso tampoco es complejo, se necesita 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 NULO.

- Auxiliar ← primero
- Al Nodo se le asigna el valor y NULO como siguiente Nodo^. ← <valor, NULO>
- Avanzar hasta que Auxiliar^.siguiente apunte al valor NULO.
- Hacer que Auxiliar^.siguiente apunte al nuevo Nodo y así queda enlazado.

Insertar un elemento a continuación de un nodo cualquiera de una lista:
En este caso, siguiendo el criterio del ejemplo anterior se hace que el nodo "anterior" sea aquel a continuación del cual insertaremos el nuevo nodo:
Si ya se dispone del anterior y del nuevo nodo, el proceso a seguir será:

- Hacer que Nodo^.siguiente apunte a anterior^.siguiente.
- Hacer que anterior^.siguiente señale Nodo.

Recorrer una lista:
Las listas simplemente enlazadas sólo pueden recorrerse en un sentido, ya que cada nodo apunta al siguiente.
Para recorrer una lista se utiliza un puntero auxiliar como índice:

- Se le asigna al puntero índice el valor de Lista.
- Se recorre con un ciclo de repetición que al menos debe tener una condición, que el índice no sea NULO.
- Dentro del Ciclo se asigna al índice el valor del nodo siguiente al índice actual.

Eliminar elementos en una lista

Eliminar el primer nodo de una lista abierta:
Es el caso más simple. Se parte de una lista con uno o más nodos, y se utiliza un puntero auxiliar, Nodo:

- Se hace apuntar a Nodo al primer elemento de la lista, es decir a Lista.
- Se le asigna a Lista la dirección del segundo nodo de la lista: Lista=Lista->siguiente.
- Se libera la memoria asignada al primer nodo, apuntado por Nodo.
- Si la lista sólo tiene un nodo, el proceso es también válido, ya que el valor de Lista^.siguiente es NULO, y después de eliminar el primer nodo la lista quedará vacía, y el valor de Lista será NULO.

Eliminar un nodo cualquiera de una lista simplemente enlazada:
En todos los demás casos, eliminar un nodo se puede hacer siempre del mismo modo. Supongamos que tener 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:
Se busca que Nodo apunte al nodo a eliminar**(1)**.
Se asigna como nodo siguiente del nodo anterior, el siguiente al que se quiere eliminar: anterior->siguiente = nodo->siguiente.**(2)**
Se elimina la memoria asociada al nodo a eliminar.**(3)**

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 NULO.

Moverse a través de una lista
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.
1. 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, teniendo un puntero auxiliar para contener el anterior.

```
Nodo = Primero
MIENTRAS no sea el nodo buscado HACER
        Auxiliar = Nodo
        Nodo =  Nodo->siguiente
FIN_MIENTRAS
```

Auxiliar apuntara al nodo anterior al buscado

Último elemento de una lista:
Para obtener un puntero al último elemento de una lista se parte del primer nodo y se avanza hasta que el nodo apunte a NULO, el auxiliar apuntara al último. También se puede preguntar anticipadamente por el valor NULO evitándose de este modo el puntero auxiliar
Suponiendo la lista no vacía
Nodo = Primero;
MIENTRAS Nodo->siguiente <> NULO HACER
    Nodo ← Nodo->siguiente
FIN_MIENTRAS
En este caso Nodo, al final del ciclo apunta l último nodo de la estructura

Como determinar si una lista está vacía:
Basta con comparar el puntero Lista con NULO si Lista vale NULO la lista está vacía.

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.

Algoritmo de inserción:

- El primer paso es crear un nodo para el dato que vamos a insertar.
- Si Lista es NULO, 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.
- En caso contrario, se debe buscar el lugar adecuado para la inserción, tenemos un puntero "anterior". Lo inicializamos con el valor de Lista, y avanzaremos mientras anterior^.siguiente no sea NULO y el dato que contiene anterior^.siguiente sea menor que el dato a insertar.

- Ahora anterior señalando al nodo previo al insertar, por lo que se enlazan lo puntero de modo que el siguiente del nuevo sea el que era siguiente del anterior y al siguiente del anterior se lo hace apuntar al nuevo nodo. Se debe recordar que anterior contiene el valor inmediato anterior al valor insertado y el siguiente es mayor. El nuevo nodo debe intercalarse entre ambos.

En el capitulo de algoritmos puntuales se ofrecen varios ejemplos para insertar nodos en situaciones distintas y con distintos fines

Algoritmo para borrar un elemento:

Para eliminar un nodo se necesita disponer de un puntero al nodo anterior.

- Lo primero será localizar el nodo a eliminar, si es que existe. Pero sin perder el puntero al nodo anterior.
- Se parte del nodo primero, y del valor NULO para anterior.
- Se avanza mientras nodo no sea NULO o mientras que el valor almacenado en nodo sea menor que el que buscamos.
- Ahora pueden darse tres casos:
- Que el nodo sea NULO, esto indica que todos los valores almacenados en la lista son menores que el que buscamos y el nodo que buscamos no existe. Entonces se retorna sin borrar nada.
- Que el valor almacenado en nodo sea mayor que el buscado, en ese caso también se retorna sin borrar nada, ya que esto indica que el nodo que buscado no existe.
- Que el valor almacenado en el nodo sea igual al buscado, nuevo existen dos casos:
    a. Que anterior sea NULO. Esto indicaría que el nodo que se quiere borrar es el primero, así que se modifica el valor de Lista para que apunte al nodo siguiente al que se quiere borrar.
    b. Que anterior no sea NULO, el nodo no es el primero, así que se asigna a anterior^.siguiente la dirección de nodo^.siguiente.

c. Después de 7 , se libera la memoria de nodo.

Acciones y funciones para pilas

Pila estructura del tipo LIFO el ultimo en entrar es el primero en salir, en general el uso de esta estructura es adecuado cuando se persigue el propósito de invertir el orden de una estructura dada. Su uso es reservado para los compiladores que los utilizan para recursividad y las notaciones sufijas, por ejemplo.

CrearPila(Dato_Resultado Pila: TPNodo): una Accion

Las estructuras deben ser creadas, esto es hacer apuntar al valor NULO, o inicializar la estructura antes de comenzar a trabajar con ellas.

```
ALGORITMO
        Pila = NULO;
FIN
```

Simplemente asigna el valor NULO al puntero que controla el inicio de la estructura

EsVacia(Dato Puntero: TPNodo) : Boolean una Funcion

Esta función determina si hay nodos en la pila, es una forma de verificar la existencia de elementos en la estructura para no producir errores en tiempo de ejecución como por ejemplo intentar sacar cuando ya no hay datos.

```
ALGORITMO
        EsVaciaPila ← Puntero = NULO;
FIN
```

Retona verdadero si el puntero a punta al valor NULO y falso en caso contrario. La asignación d una expresión logica asigna verdadero al identificador booleano si se cumple la condicion y falso en caso contrario

Meter(Dato_Resultado Pila: TPNodo; Dato Valor: Tinfo) una accion

Este procedimiento inserta un nuevo nodo en la pila, se encarga de pedir memoria, guardar la información y actualizar los punteros. En esta estructura siempre la pila retorna el valor del primer nodo, y como se

agrega adelante, el procedimiento siempre retorna en pila el nodo que acaba de crear.

LEXICO

Pre Pila Apunta al Inicio o a NULO si esta vacía

Pos Pila apunta al primer elemento de la estructura (coincide con el creado

La estructura queda perfectamente enlazada.

```
Ptr : TPNodo;
ALGORITMO
        Nuevo(Ptr);
        Ptr^ ← <Valor, Pila>;
        Pila ← Ptr
FIN.
```

Pide memoria, guarda el valor en el nuevo nodo, hace que el siguiente de este nodo sea pila, es ex primero (porque agrega delante del primero) y pila apunta al nodo creado

Sacar(Dato_Resultado Pila: TPNodo; Dato_Resultado Valor: Tinfo) una accion

Este procedimiento libera el nodo apuntado por pila, se encarga de guardar la información contenida en el primer nodo en valor, por lo que este es un parámetro variable y actualizar los punteros.

LEXICO

PRE: Pila Apunta al Inicio y no esta vacía

POS: Libera el nodo de la cima y retorna en valor la informacion. Pila apunta al nuevo primer elemento de la estructura o a NULO si queda Vacia.

La estructura queda perfectamente enlazada.

```
Ptr : TPNodo;
ALGORITMO
        Ptr ← Pila;
        Valor ←Ptr^ .info;
        Pila ← Pila^.sgte;
        Destruir(Ptr);
FIN
```

Utiliza un puntero auxiliar para conservar el inicio de la pila, guarda el valor contenido en ese puntero, avanza con la estructura y libera el puntero auxiliar.

Acciones y funciones para Colas

CrearCola(Dato_Resultado ColaFte,ColaFin: TPNodo): una Accion
ALGORITMO
    ColaFte = NULO;
    ColaFin = NULO;
FIN

Agregar(Dato_Resultado Colafte, Colafin: TPNodo; Dato Valor: Tinfo) una accion

Este procedimiento inserta un nuevo nodo en la cola, se encarga de pedir memoria, guardar la información y actualizar los punteros. En esta estructura siempre ColaFte retorna el valor del primer nodo, y como se agrega después del último, el procedimiento siempre retorna en colafin el nodo que acaba de crear.

LEXICO
    PRE Colafte Apunta al Inicio o a NULO si esta vacía, Cola Fin al final o NULO
    POS ColaFte apunta al primer elemento de la estructura.
        ColaFin apunta al ultimo elemento de la estructura (coincide con el creado)
La estructura queda perfectamente enlazada.
Ptr : TPNodo;
ALGORITMO
    Nuevo(Ptr);
    Ptr^ ← <Valor, NULO>;
    SI (Colafte = NULO)
    ENTONCES
        Colafte ← Ptr
    SI_NO
        ColaFin^.sgte = Ptr;

Se pide memoria se guarda la información y el siguiente del nuevo nodo siempre es el valor NULO. Para enlazarlo, como vimos, hay dos situaciones posibles cuando es el primer nodo, en ese caso el puntero al inicio apunta al nuevo nodo y cuando ya hay por lo menos un nodo, en ese caso se debe enlazar el nuevo nodo a continuación del anterior ultimo. En todos los casos el puntero al ultimo siempre apuntara al nodo creado.

FIN_SI;
ColaFin ← Ptr
FIN.

Suprimir(Dato_Resultado Colafte, ColaFin: TPNodo; Dato_Resultado Valor: Tinfo) una accion
Este procedimiento libera el nodo apuntado porcolafte, se encarga de guardar la información contenida en el primer nodo en valor, por lo que este es un parámetro variable y actualizar los punteros
LEXICO
PRE ColaFte Apunta al Inicio y no esta vacia
POS Libera el nodo de la cima y retorna en valor la informacion.
Colafte apunta al nuevo primer elemento de la estructura o a NULO si queda Vacia, en ese caso, ColaFin tambien apuntara a nulo.
La estructura queda perfectamente enlazada.
Ptr : TPNodo;
ALGORITMO
Ptr ← Colafte;
Valor ←Ptr^ .info;
ColaFte ← ColaFte^.sgte;
SI (ColaFte = NULO)
ENTONCES
ColaFin = NULO
FIN_SI;
Destruir(Ptr);
FIN

Utiliza un puntero auxiliar para conservar el inicio de lacolaa, guarda el valor contenido en ese puntero, avanza con la estructura y libera el puntero auxiliar Si el puntero al inicio apunta a NULO (cuando saca el ultimo valor), entonces tambien se hace apuntar al puntero auxiliar al valor NULO. La unica vez que el puntero al final se modifica, cuando se saca, es en este ultimo caso.

Acciones y funciones para Listas Ordenadas enlazadas
CrearLista(Dato_Resultado Lista: TPNodo): una Accion
ALGORITMO
Lista = NULO;
FIN

InsertaNodo(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

LEXICO

PRE Lista Apunta al Inicio o a NULO si esta vacia.

POS Lista apunta al primer elemento de la estructura.

No retorna la direccion del nodo creado, salvo si es el primero

La estructura queda perfectamente enlazada y ordenada creciente.

Ptr, PtrNuevo : TPNodo;

ALGORITMO

SI (lista = NULO o Valor < Lista^.info

ENTONCES

Nuevo(Ptr);

Ptr^ ← <Valor, Lista>;

Lista ← Ptr

Si la lista esta vacia, o el valor a guardar es menor que el que esta en la primera posición se coloca delante del primero con un procedimiento identico al de insertar en una

SI_NO

Se pide memoria y se guarda la informacion

Nuevo(PtrNuevo);

PtrNuevo^ ← <Valor, NULO>;

Ptr = Lista;

MIENTRAS (Ptr^.sgte <>NULO y Valor > Ptr^.sgte^.info)

HACER

Ptr ← ptr^.sgte

FIN_MIENTRAS

Pregunta anticipadamente por el valor contenido en el nodo siguiente al apuntado por ptr. Si la informacióna ingresar es mayor a la del nodo siguiente debe avanzar una posicion

PtrNuevo`.sgte ← Ptr^.sgte;
Ptr^.sgte ← Ptrnuevo;
FIN_SI;

FIN.

Se enlazan los punteros, como se pregunta por adelantado, el siguiente del nuevo sera el que es siguiente al nodo donde salimos que es el inmediato anterior al nueo valor. El siguiente de ese nodo sea ahora el nuevo. El nuevo lo colocamos entre el que señalamos con ptr y el siguiente

InsertaPrimero(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

LEXICO
PRE Lista Apunta a NULO porque esta vacia.
POS Lista apunta al primer elemento de la estructura.

Ptr: TPNodo;
ALGORITMO
Nuevo(Ptr);
Ptr^ ← <Valor, NULO>;
Lista ← Ptr
FIN.

InsertaDelante(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

LEXICO
PRE Lista Apunta al Inicio y noesta vacia.
POS Lista apunta al primer elemento de la estructura.

Ptr: TPNodo;
ALGORITMO
Nuevo(Ptr);
Ptr^ ← <Valor, Lista>;

Lista ← Ptr
FIN.

InsertaEnMedio(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion
LEXICO
PRE Lista Apunta al Inicio no esta vacia.
POS Lista apunta al primer elemento de la estructura.
No retorna la direccion,queda perfectamente enlazada y ordenada.
Ptr, PtrNuevo : TPNodo;
ALGORITMO
Nuevo(PtrNuevo);
PtrNuevo^ ← <Valor, NULO>;
Ptr = Lista;
MIENTRAS (Ptr^.sgte <>NULO y Valor > Ptr^.sgte^.info)
HACER
Ptr ← ptr^.sgte
FIN_MIENTRAS
PtrNuevo`.sgte ← Ptr^.sgte;
Ptr^.sgte ← Ptrnuevo;
FIN.

InsertaAl Final(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

LEXICO
PRE Lista Apunta al Inicio o a NULO si esta vacia.
POS Lista apunta al primer elemento de la estructura.
No retorna la direccion del nodo creado.
Ptr, PtrNuevo : TPNodo;
ALGORITMO

```
        SI (lista = NULO o Valor < Lista^.info
ENTONCES
Nuevo(Ptr);
                Ptr^ ← <Valor, Lista>;
                Lista ←  Ptr
        SI_NO
          Nuevo(PtrNuevo);
  PtrNuevo^ ← <Valor, NULO>;
  Ptr = Lista;
  MIENTRAS (Ptr^.sgte <>NULO)
  HACER
                Ptr ← ptr^.sgte
          FIN_MIENTRAS
          PtrNuevo`.sgte ← Ptr^.sgte;
          Ptr^.sgte ← Ptrnuevo;
FIN_SI;

FIN.
```

InsertaNodoPorDosCampos(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

```
LEXICO
        PRE    Lista Apunta al Inicio o a NULO si esta vacia.
        POS    Lista apunta al primer elemento de la estructura.
               Info es un rec¡gistro con al menos dos campos para ordenar
               No retorna la direccion del nodo creado, salvo si es el
primero
La estructura queda perfectamente enlazada y ordenada creciente.
Ptr, PtrNuevo : TPNodo;
ALGORITMO
        SI (lista = NULO) o (Valor.C1 < Lista^.info.C1) o
```

(Valor.C1 =Lista^.info.C1)y(Valor.C2<Lista^.info.c2)
ENTONCES
Nuevo(Ptr);
Ptr^ ← <Valor, Lista>;
Lista ← Ptr
SI_NO
Nuevo(PtrNuevo);
PtrNuevo^ ← <Valor, NULO>;
Ptr = Lista;
MIENTRAS (Ptr^.sgte <>NULO) y ((Valor > Ptr^.sgte^.info) o ((valor.C1 = Lista*.info.C1)y(Valor.C2 >Lista*.info.C2))
HACER
Ptr ← ptr^.sgte
FIN_MIENTRAS
PtrNuevo`.sgte ← Ptr^.sgte;
Ptr^.sgte ← Ptrnuevo;
FIN_SI;

FIN.

BuscarNodo(Dato Lista: TPNodo; Dato Valor: Tinfo): TPNodo una funcion

LEXICO
PRE Lista Apunta al Inicio y no esta vacia.
Se busca encontrar el nodo que contenga valor como informacion
POS Lista apunta al primer elemento de la estructura.
Retorna la direccion del nodo con el valor buscado o NULO esta

Ptr : TPNodo;
ALGORITMO

```
        Ptr = NULO;
  MIENTRAS (Lista <>NULO  Y  Ptr = NULO)  HACER
                SI valor = Lista ^.info
                ENTONCES
                        Ptr = Lista {lo encontro y sale}
                SINO
                        Lista = Lista ^.sgte {avanza al proximo nodo}
FIN_SI
        FIN_MIENTRAS
  BuscarNodo = Ptr;

FIN.
```

BuscaOInserta(Dato_Resultado Lista,Ptr: TPNodo; Dato Valor: Tinfo) una accion

```
LEXICO
        PRE    Lista Apunta al Inicio o a NULO si esta vacia.
        POS    Lista apunta al primer elemento de la estructura.
               Retorna la direccion del nodo creado, salvo si es el primero
La estructura queda perfectamente enlazada y ordenada creciente.
No se repite la clave
PtrNuevo : TPNodo;
ALGORITMO
        SI (lista = NULO o Valor < Lista^.info
ENTONCES
Nuevo(Ptr);
                Ptr^ ← <Valor, Lista>;
                Lista ←  Ptr
        SI_NO
          Ptr = Lista;
```

```
 MIENTRAS (Ptr^.sgte <>NULO  y Valor >= Ptr^.sgte^.info)
 HACER
            Ptr ← ptr^.sgte
        FIN_MIENTRAS
        SI (Ptr^.Info <> Valor)
        ENTONCES
            PtrNuevo^.sgte ← Ptr^
            Ptr^.sgte ← Ptrnuevo;
      Ptr ← PtrNuevo;
 FIN_SI;
FIN_SI;
```

Similar al procedimiento inserta nodo, pero como no debe insertar siempre se cambia el orden de las accines cuando se debe insertar en el medio. En este caso se busca primero y si no esta se crea el nodo y se enlazan los punteros. Como ptr en este estado no puede valer NULO y el siguiente si el ciclo de repetición se hace modificando el operador de relacion por >= en lugar de >. Al salir del ciclo si el valor contenido en el nodo actual es distinto al buscado significa de ese dato no esta en la lista y se debe inserta, de lo contrario no se inserta.

FIN.

InsertaNodo2(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

LEXICO
PRE Lista Apunta al Inicio o a NULO si esta vacia.
POS Lista apunta al primer elemento de la estructura.
No retorna la direccion del nodo creado, salvo si es el primero
La estructura queda perfectamente enlazada y ordenada creciente.
P,Q,Ptr : TPNodo;
ALGORITMO
Nuevo(Ptr);
Ptr^ ← <Valor, NULO>;
P ← Lista;
Q ← NULO;
MIENTRAS P <> NULO y Valor > p^.info HACER
Q ← P;
P ← P^.sgte;

```
        FIN_MIENTRAS;

        SI (P = Lista)
        ENTONCES
                Lista ← Ptr;
        SI_NO
                Q^.sgte ← Ptr;
        FIN_SI;
        Ptr^.sgte ← P;
FIN.
```

SuprimeNodo(Dato_Resultado Lista: TPNodo; Dato Valor: Tinfo) una accion

```
LEXICO
        PRE    Lista Apunta al Inicio y no esta vacia
        POS    Libera el nodo si encentra el valor.
P, Q : TPNodo;
ALGORITMO
        P ← Lista;
Q ← NULO:
MIENTRAS (P <> NULO y Valor > P^.info) HACER
        Q ← P;
        P ← P^.sgte:
FIN_MIENTRAS;
SI (P <> NULO y Valor = P^.info
ENTONCES
        SI Q <> NULO
        ENTONCES
                Q^.sgte ← P^.sgte
        SI_NO
                Lista ← P^.sgte;
```

FIN_SI:
Destruir(P):
FIN_SI;
FIN

Implementaciones

Definición del Nodo

```
struct Nodo
{
   int info;
   Nodo* sgte;
};
```

Pilas

Push

```
void push Nodo*& p, int v){
      Nodo*q = new Nodo();
      q->info = v;
      q->sgte= p;
      p = q;
      return;
}
```

Pop

```
int pop(Nodo*& p){
      int v;
      Nodo*q = p;
      v = q->info;
      p = q->sgte;
      delete q;
      return v;
}
```

Colas

agregar

```
void agregar(Nodo* &fte, Nodo* & fin, int v){
	Nodo* p = new Nodo();
	p->info = v;
	p->sgte = NULL;
	if (fte == NULL)
		fte = p;
	else
		fin->sgte = p;
	fin = p;
	return;
}
```

Suprimir

```
int suprimir(Nodo*& fte, Nodo*& fin){
	int v;
	Nodo*q = fte;
	v = q->info;
	fte = q->sgte;
	if(fte == NULL) fin = NULL; //si queda vacia el fin
apunta a null
	delete q;
	return v;
}
```

```
	fte = q->sgte;
	if(fte == NULL) fin = NULL; //si queda vacia el fin
apunta a null
	delete q;
	return v;}
```

Listas ordenadas simplemente enlazadas

Insertar el primer nodo

```
Nodo* insertaPrimero(Nodo*& l, int x){
   Nodo* p = new Nodo();
   p->info = x;
   p->sig = NULL;

   l = p;
   return p;
}
```

La función anterior es un caso particular de la siguiente, porque?

Insertar delante del primer nodo

```
Nodo* InsertaDelante(Nodo*& l, int x){
   Nodo* p = new Nodo();
   p->info = x;
   p->sgte = l;

   l = p;
   return p;
}
```

Insertar alfinal (supone la lista no vacia)

```
Nodo* InsertaAlFinal(Nodo*& l, int x){
      Nodo* nuevo = new Nodo();
      Nodo*p= l;
      nuevo->info = x;


      while(p->sig!=NULL ){
         p = aux->sig;
      }
      nuevo->sgte = NULL;
      p->sgte = nuevo;
      return nuevo;
      }
```

Insertar en medio de dos conservando el orden creciente (supone la lista no vacia)

```
Nodo* InsertaEnMedioDeDos(Nodo*& l, int x){
      Nodo* nuevo = new Nodo();
      Nodo*p= l;
      nuevo->info = x;
      while(x>p->sgte->info){
         p = aux->sgte;
      }
      nuevo->sgte = p->sgte;
      p->sgte = nuevo;
      return nuevo;
      }
```

La función que sigue contiene a las dos anteriores, está de acuerdo? Justifique.

Insertar en medio (supone la lista no vacia)

```
Nodo* InsertaEnMedio(Nodo*& l, int x){
      Nodo* nuevo = new Nodo();
      Nodo*p= l;
      nuevo->info = x;
      while(p->sgte!=NULL && x>p->sgte->info){
         p = aux->sgte;
      }
      nuevo->sgte = p->sgte;
      p->sgte = nuevo;
      return nuevo;
}
```

Mostrar

Muestra el contenido sin perder la lista, se envía por valor

```
void mostrar(Nodo* l){
      while( aux!=NULL ){
      cout << aux->info << endl;
      aux = aux->sig;
   }
}
```

Vaciar contenido de una lista

```
void eliminar(Nodo*& l)
{
   Nodo* p;
   while( l!=NULL ){
      p = l;
      l = p-> sgte;
      delete p;
    }
}
```

Insertar un nodo con un dato simple y orden creciante

```
Nodo* insertarOrdenado(Nodo*& l, int v){
   Nodo* nuevo = NULL;
   If(l==NULL || v < l->info)
      Nuevo = insertaPrimero(l,v);
   else
      Nuevo = insertaEnMedio(l,v);
   return Nuevo;
}
```

Buscar En Lista

Busca un valor, retornando la posición, en caso de encontrarlo o NULL en caso que este no esté.

```
//pregunto por distinto a efectos de independizarme del orden
Nodo* buscar(Nodo* l, int v)
{
   Nodo* p = l;
   while( p!=NULL && p->info!=v )
      p = p->sgte;

   return aux;
}
```

Insertar sin repetir la clave

```
Nodo* buscaOInserta(Nodo*& l, int v){
   Nodo* x = buscar(l,v);//lo busca…

   if( x ) x = insertarOrdenado(p,v);//si no lo encuentra lo
inserta

   return x;
}
```

Esta función es de importancia en particula en dos patrones algorítmicos particulares, cargar sin repetir para acumular y lista de listas, una estructura compleja que en el campo de la información tiene un puntero a una lista auxiliar

Eliminar de Nodo

Elimina de la lista el valor x, en caso de encontrarlo, retorna verdadero en esta caso y falso en caso contrario

```
bool eliminar(Nodo*& l, int x){
   Nodo* p = l;
   Nodo* q = NULL;
   bool enc = false;
   while( p!=NULL && p->info!= x ){
      q = p;
      p = p->sgte;
   }

   if( p!=NULL ){//en este caso esta en P y hay que
eliminarlo
      enc = true;
      if(p==l)//pero hay que verificar si no esta en la
primera posicion
       l = p->sgte;
      else
       q->sgte = p->sgte;
      delete p;
 return enc;
}
```

Plantillas

Este tema NO se evalúa, se marca en rojo y subrayado lo que se modifica respecto del que no utiliza plantillas
El nodo

```
template <typename T> struct Nodo
{
   T info;
   Nodo<T>* sig;
};
```

Listas ordenadas simplemente enlazadas

Insertar delante del primer nodo

```
template<typenam<T> Nodo<T>* InsertaDelante(Nodo<T>*& l, T
x){
   Nodo<T>* p = new Nodo();
   p->info = x;
   p->sgte = l;
   l = p;
   return p;
}
```

Insertar alfinal (supone la lista no vacia)

```
template<typenam<T> Nodo* InsertaAlFinal(Nodo<T>*& l, T x){
      Nodo<T>* nuevo = new Nodo<T>();
      Nodo<T>*p= l;
      nuevo->info = x;
```

```
        while(p->sig!=NULL ){
           p = aux->sig;
        }
        nuevo->sgte = NULL;
        p->sgte = nuevo;
        return nuevo;
        }
```

```
template <typename T> void push(Nodo<T>*& p, T v){

        Nodo<T>* q = new Nodo<T>();
        q->info = v;
        q->sgte= p;
        p = q;
        return;
}
```

```
template <typename T> T push(Nodo<T>*& p){

        T v;
        Nodo<T>*q = p;
        v = q->info;
        p = q->sgte;
        delete q;
        return v;
}
```

```
Ejemplo de uso
Nodo* p = new Nodo()// invocación a la función sin plantilla
Nodo<int>* q = new Nodo<int>(); invoca a la funcion con
plantilla con int
Nodo<float>* r = new Nodo<float>(); invoca a funcion con
plantilla con float
int a = pop(p);
int b = pop<int>(q);
float c = pop<float>(R);
```

Esta función agrega un nuevo concepto, con que criterio se ordena. Se debe determinar si es creciente o decreciente, si es un dato simple o un campo de una struct, esto se resuelve con un nuevo parámetro,un puntero a una función.

Insertar en medio (supone la lista no vacia)

```
template<typenam<T> Nodo* InsertaEnMedio(Nodo*& l, T x, int (*criterio)(T,T)){
      Nodo<T>* nuevo = new Nodo<T>();
      Nodo<T>*p= l;
      nuevo->info = x;
      while(p->sgte!=NULL && criterio(v,p->sgte->info)>0){
         p = p->sgte;
      }
      nuevo->sgte = p->sgte;
      p->sgte = nuevo;
      return nuevo;
}
```

```
Ejemplo de uso
Struct tr{ int C1; float C2};

Int CriterioDatoSimpleCrec(int a, int b{
 return a-b;//si el valor a colocar es mayor a lo del nodo sgte sigue
}
Int CriterioDatoSimpleDecreciente(int a, int b{
 return b-a;//al reves
}
Int CriterioC1Creciente(tr a, tr b{
 return a.C1-b.C1;es por un campo determinado y creciente
}

Nodo<int>* p=InsertarEnMedio(L1,4,CriterioDatoSimpleCrec)
En forma similar para todos los demás.
```