Unidad 11 Punteros, asignación dinámica en memoria y estructuras enlazadas lineales

11

Contenidos analíticos

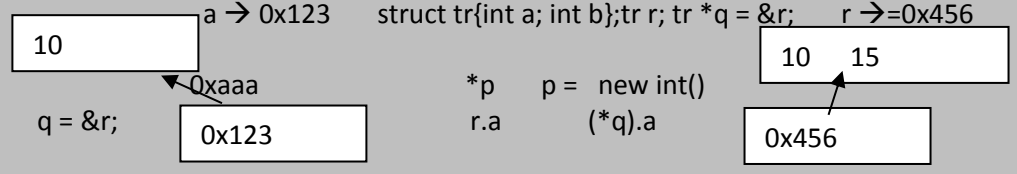
Parte IV Almacenamiento Físico - Estructuras con asignación dinámica
Punteros, asignación dinámica en memoria y estructuras enlazadas lineales

Definición de punteros. La necesidad de la asignación dinámica en memoria. Constructores y destructores. Concepto de instancia y operadores de dirección y desreferenciacion Punteros a datos y punteros a código. Estructuras enlazadas con asignación dinámica en memoria: Estructuras lineales, ventajas y debilidades. Definiciones y declaraciones
Estructura de tipo pila: Patrones de carga y eliminación de nodos
Estructura de tipo cola: Patrones de carga y eliminación de nodos.
Estructura de tipo lista: listas simplemente enlazadas, listas circulares, listas doblemente enlazadas y listas circulares doblemente enlazadas.
Patrones de carga sin restricciones, de carga sin repetir del dato, de búsqueda, de recorrido (completo, parcial, con corte de control o apareo) de eliminación de nodos

Punteros y estructuras enlazadas lineales

Conceptos preliminares
Un puntero es un tipo de dato que como valor contiene una dirección de memoria en la que puede alojarse un dato simple una estructura o un determinado código (puntero a una función)

Puntero
Valores → Direcciones de memoria donde un dato o código
Operaciones → asignación →&, new(tipo) NULL
Ejemplo int* p;
int a = 10; // declaración de una variable de tipo entero inicializada en 10
int *p=NULL,*p2;// declaración de puntero a
entero podrá tener la dirección de memoria donde se aloja un int



Asigna la dirección de a()
p = &a;
q->a;
*p

Nombre	tipo de dato	valor	q=new tr();
a	int	10	delete (p)
p	puntero a int	0x123	
*p	int	10 → valor contenido en el lugar de memoria donde apunta p	
r	tr	10, 15	r.a r.b
q	puntero a tr	0x456	
*q	tr	10, 15	(*q).a q->a (*q).b q->b
(*q).b	int	15	(*q).a
q->b	int ->	15 →operador de acceso a un miembro de struct apuntada por puntero.	

Operadores

& → de dirección → se aplica sobre un identificador

* → de indireccion o desreferenciacion → se aplica sobre un puntero

-> operador de acceso a un campo de un registro apuntado por un puntero equivale a (*puntero).campo

Asignaciones

p = NULL → asigna el valor nulo a un puntero

p2 = p; le asigna a p2 el valor de p, ambos punteros son del mismo tipoy ahora apuntan a la misma dirección

p = q; INCORRECTO los punteros con tipo y p y q apuntan a tipos de datos distintos

*p = q->b le asigna al entero apuntado por p el campo b del registro apuntado por r

p = new int() crea un nuevo espacio de memoria en TE para almacenar un entero → asignación dinámica

q= new tr(); crea un nuevo espacio de memoria en TE para almacenar un tr → asignación dinámica

Los espacios de memoria creados con new son INSTANCIAS de los punteros, y se referencian mediante el mismo. Al no tener nombre propio se los llama variables anónimas.

Los punteros son variables estáticas, las instancias son dinámicas, se pueden crear y liberar en TE

delete (p); libera la instancia a la que apunta p, similar seria delete(q);

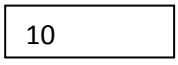
(*q).c1 q->c1

Prioridad asociatividad operadores

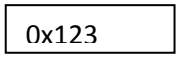
PRIORIDAD AS.	OPERADOR/ES
G1 S/A	::
G2 I→D	. -> [ ] ( ) ++ -- posfijo
G3 D→I	sizeof ++ -- prefijo ! + - unario &direccion * indirección
G4 I→D	* / %
G5 I→D	+ - suma resta

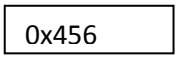
int a = 10, v[4] = {6,7,8,9}, v1[4],*p1,*p2;

a 0x123 *(p2+2) v[2] *(v+2) *(p2+2) v[2] p2++



p2 = &v[0], o p2 = v





p1 = & a;

0XABC

?			

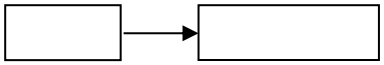
V1 = V → INCORRECTA PORQUE TRATA DE ASIGNAR DIRECCIONES Y EN LOS VECTORES

0x456

v[0] 6	**v[1] 7**	**v[2] 8**	**v[3] 9**

v equivalente a &v[0]
(*p1)++ *p1 *(v+2) v[2]
*p2 6 p2++ avanza el puntero v1++ produce error porque v es puntero constante
*(p2+2) = 8 = v[2] p2[2]
*(2+p2) = 8 = 2[v]
p2++;
*p2, *p2++, (*p2)++; ++*p2; ++(*p2)

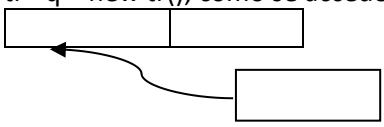
int a, v[10]
Suponiendo las siguientes declaraciones con las correspondientes definiciones
struct tr{int a, int b}; tr r;
int cantRegistros(FILE *);
void ordenarVector(tr [], int);
Interrogantes
int *p = new int(); genera en tiempo de ejecución lo que sigue?



p *p
p = new int[10] es correcto? de ser asi se puede acceder a cada miembro? justifique

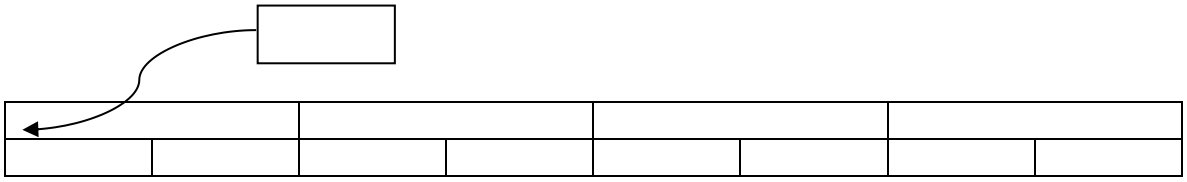
?									

tr *q = new tr(); como se accede a la estructura, y a cada miembro? tr v4[4]



q

q = new tr[4]; es correcto, genera lo que sigue? se puede acceder a la estructura o a cada miembro? como?



con un flujo de registros como el descripto, abierto para lectura, es posible la siguiente secuencia de código?

```
int n = cantRegistros(f);
q = new tr[n];
fread(q, sizeof (tr), n,f);
ordenarVector(q, n);
mostrarVector( tr q);
struct articulo{
        char s[10];
        int precio;
};
```

```
struct vendedor{
        char nombre[20];
        articulo productos[10]
};

vendedor vendedores[30];

vendedor matrizComplicada[5][10][10];
```

vendedor * punteroDificil = new vendedor[5][10][10]; es posible? como se accede a cada miembro? se puede hacer lo mismo que con la estructura anterior? cual es, si la hay, la diferencia conceptual?

Punteros a funciones

Los punteros contienen direcciones de memoria, las que pueden ser a un tipo de dato, o a código, en este caso se los denomina punteros a funciones

```
int  *fptr (int);   // E.2
```

es el prototipo de una función que recibe un int y devuelve puntero-a-int

```
int (*fptr)(int);   // E.1
```

es la declaración de un puntero-a-función que recibe un int y devuelve un int.

```
void func(int (*fptr)(int));   // E.3: Ok.
```

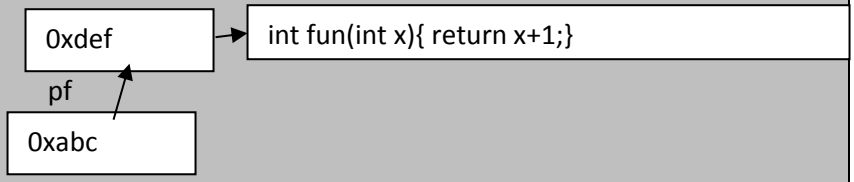
La expresión E.3 es el prototipo de una función que no devuelve ningún valor y recibe como argumento un puntero-a-función.

```
#include <iostream>
using namespace std;

int* fun0(int i) {          //
  cout << i << endl;
  return &i;}
//recibe un int y retorna un puntero a int

int fun1(int i) {           //
  cout << i << endl;
  return 10*i;} // recibe un int y retorna un int

void fun3(int (*pf)(int)) { pf(20); }      // int* pf es distinto a int (*pf)
```

Función que retorna valor ausente, recibe un puntero-a-función que evalua un int y devuelve un int. Ejecutan la función señalada por su argumento

```
int main(void) {
  int x = 10;
  int y = fun1(x);          //asigna a y el resultado de fun1
  fun1(fun1(y));            // invoca fun1 recursivamente
  int (*pf1)(int) = &fun1;     // pf1 es el puntero a la función fun1
  pf1(x);                   // se invoca a la fun1 mediante el puntero
  fun3(pf1);                // se invoca fun3 con el puntero pf1 que ejecuta fun1
  int* (*pf2)(int) = &fun0;    // pf2 es puntero-a-función que recibe un int y devuelve un puntero-a-
int.  Lo iniciamos con la dirección de la función fun0
  pf2(y);                   // Invocamos fun0 utilizando su puntero

int fun(int x){return x+1;}
```

fun → 0xabc 0xdef




```
int (*pf)(int) = &fun;
```

Sintesis

Puntero es un tipo de dato (tipado) que apunta a direcciones de memoria en la que se aloja dato o código

Apuntar, metafóricamente, significa que contiene la dirección de memoria en la que puede haber un dato o una porción de código (puntero a función)

Se puede asignar dirección de memoria de datos estáticos o, mediante new esta asignación puede ser dinámica.

Asignacion dinámica crea instancias que son, justamente, dinámicas, es decir direcciones que pueden crearse o liberarse en tiempo de ejecución.

Estructuras enlazadas

Una estructura de tipo vector, si se considera que el primer elemento lógico coincide con el primer elemento físico, por ejemplo, en v[o], y el siguiente elemento lógico es el siguiente físico a v[i] le sigue v[i+1] es una estructura secuencial. pero para recorrer un vector, por ejemplo, no es imprescindible recorrerlo secuencialmente.

2	
3	
5	
1	
4	

esta estructura puede ser mostrada ordenada, por ejemplo, sin necesidad de ordenarla.
Si se tiene referencia al primer elemento lógico índice 3 en este caso se muestra 1. Si luego se accede a la posición indicada por el otro campo, cero en este caso se puede acceder al siguiente elemento lógico. Esto es una estructura enlazada. Se referencia al primero y cada elemento es un registro con información y referencia

1
2
3
4
5

2	**1**
3	4
5	-1
1	0
4	2

POSICION DEL PRIMERO 3

Nombre	Cargan	sacan
Pilas	Delante del primero	De la primera posición
Colas	Después del ultimo Prioridad	Primera posición Según la prioridad
Listas simplemente enlazadas Listas circulares Listas doblemente enlazadas Listas doblemente enlazadas	Según criterio	Según ese criterio

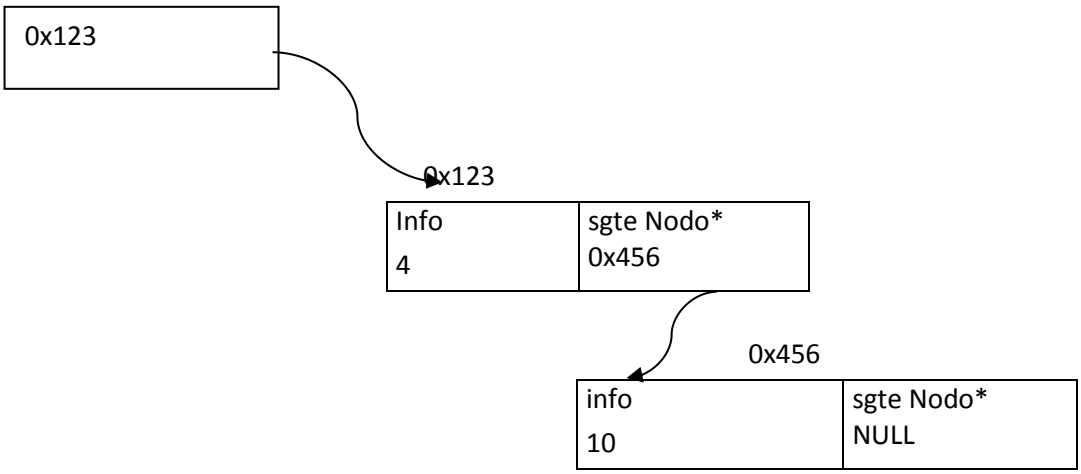
Las estructuras enlazadas pueden ser lineales, vinculan un elemento con uno, pueden ser arbóreas, vinculan un elemento con mas de uno, estos son los arboles, o pueden vincularse entre varios estos son grafos
en algoritmos veremos estructuras lineales de según como se carguen los elementos y como se los obtengan pueden ser pilas, colas o listas.

Diferentes estructuras lineales

Pilas: el nuevo elemento se agrega delante del primero, en el momento de sacar se extrae el que esta en la primera posición (LIFO) → pila de platos →recursividad
Colas: el nuevo elemento se agrega despues del ultimo, en el momento de sacar se extrae el que esta en la primera posición (FIFO) → cola de colectivo utilización de un recurso
Listas: el nuevo elemento se agrega siguiendo un determinado criterio, en general de orden. Para sacar se lo hace buscando el elemento y se lo extrae del lugar donde esta, si es que esta. esta estructura a diferencia de las otras dos se la puede recorrer.
Las pilas y colas solo pueden ser accedidas par poner un nuevo dato o sacarlo, esto de los lugares fijos donde es posible hacerlo.

las listas pueden ser: Simplemente enlazadas, doblemente enlazadas, circulares, circulares doblemente enlazadas. En AyED profundizaremos el estudio de las simple y doblemente enlazadas.
Las estructuras enlazadas se pueden implementar con vectores como hemos visto, pero esto no resolvería el problema del tamaño fijo en tiempo de ejecución. Es por ello que las implementaremos con punteros y asignación dinámica en memoria para ir generando los registros (al que llamaremos NODO) que vayamos necesitando en tiempo de ejecución. Estos nodos generaran lo que se conoce como estructuras autoreferenciadas. Tendran un campo con la información y otro que indicara la siguiente dirección de memoria donde habrá un nodo del mismo tipo o la constante NULL que indicara donde finaliza la estructura. Veamos

PUNTERO CONTROL DE LA ESTRUCTURA Nodo *

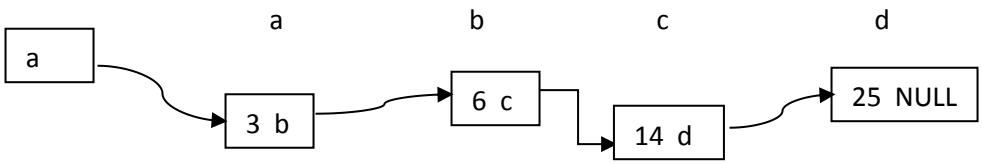



```
struct Nodo{
int info;
Nodo* sgte;
}
Nodo *pila;
```

int v[4]

3
6
14
25

Lista



while(p)			while(p->sgte)	
	p=p>sgte		p=p->sgte	
p	p->info	p->sgte	p-sgte->info	x>p->sgte->info

Nombre	Tipo	valor
v	vector	3, 6, 14, 25
lista	puntero	Direccion de memoria a
v[0]	int	3
lista→info	int	3
lista→sgte	puntero	Direccion de memoria b
v[1]	int	6
lista→sgte→info	int	6
lista→sgte→sgte	puntero	Direccion de memoria C
v[2]	int	14
lista→sgte→sgte→info	int	14
lista→sgte→sgte→sgte	puntero	Direccion de memoria D
v[3]	int	25
lista→sgte→sgte→sgte→info	int	25

Pasos para la creación y carga de estas estructuras:

1. Definir la estructura autorreferenciada→ el registro que llamaremos NODO

Este tendrá dos campos: Uno con la información y el otro con un puntero a una estructura momo la que se esta definiendo.

Ejemplo para una estructura de enteros → **struct Nodo{ int info; Nodo* sgte};**

2. declarar un identificador que contenga la dirección de memoria del inicio de la estructura, un puntero

Nodo* p = NULL; //p tendrá la referencia a la dirección de memoria del primer elemento, para crearlo es necesario hacerlos apuntar a NULL

3. tomar el dato para cargar en la estructura
4. pedir memoria
5. guardar el nuevo dato en la instancia creada

6 actualizar los punteros

Los pasos para todas las estructuras son los mismos, solo se diferencian en la forma en que se actualizan los punteros.

Se necesita un puntero para conocer donde comienza la estructura. El resto de los nodos se van vinculando con el campo siguiente de cada nodo intermedio. El valor del campo siguiente del ultimo nodo es NULL lo que significa que alli termino la estructura. Si el puntero de control de la estructura, cualquiera sea esta apunta a NULL significa que la estructura esta vacia

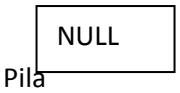
Cargar nodos en una pila

definir el Nodo

struct Nodo{ int info, Nodo* sgte}; *// crear el nodo con los dos campos*

crear la estructura

Nodo* Pila = NULL; //declarar el puntero que apuntara al inicio de la estructura. Se debe inicializar con NULL



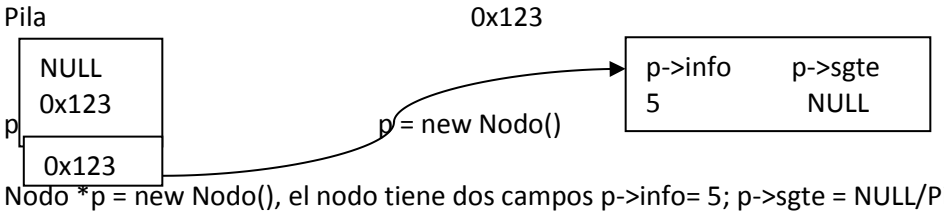
En esta estructura pueden darse dos situaciones diferentes

1 Que la pila este vacia, es decir que apunte a NULL. Al insertar un nodo este va a ser el primero y único

2 que la pila ya tenga nodos por lo que el que se genera va a ser el rimero pero no será el único

Insertar valor 5 en una Pila de enteros que esta vacia

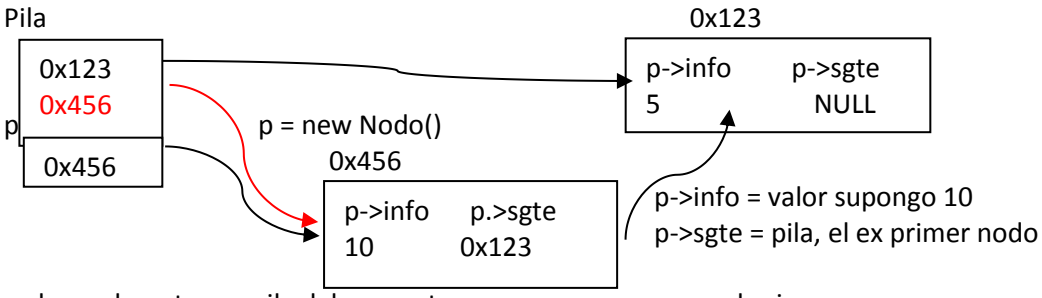
Pila si la pila esta vacia el puntero apunta a NULL



Nodo *p = new Nodo(), el nodo tiene dos campos p->info= 5; p->sgte = NULL/Pila

Ahora que se agrego el nodo la pila deja de estar vacia, el nodo creado es el primero por lo que el puntero Pila debe apuntar a esa instancia a la que apunta p, entonces Pila = p

Si la pila ya tiene nodos el procedimiento es similar



y ahora el puntero a pila debe apuntar a p ya que pasa a ser el primero

Ambas son idénticas excepto el la línea que se asigna p->sgte pero asignar NULL o pila cuando se inserta el primer nodo tienen idéntica semantica

```
void push(Nodo* & Pila, int x){          se invoca con la pila, que siempre se modifica, y el valor a guardar x
 Nodo*p = new Nodo();                    pedir memoria
  p→info = x;                            guardar la informacion
  p→sgte =Pila;                          enlazar el nuevo nodo con la estructura existente delante de Pila
  Pila = p;                              el nodo creado pasa a ser el primero
 return;
}
```

Esto vale para las dos situaciones, la sentencia p→sgte = pila; hace que el nodo siguiente al creado sea el que estaba en primer lugar, el apuntado por pila. si la pila esta vacia su valor es nul, por lo

que asignaría NULL al campo siguiente del nodo creado indicando con esto que la estructura finaliza en ese nodo.

En el caso de necesitar eliminar un nodo de la pila y retorna el valor allí guardado la secuencia será:

```
int pop(Nodo* &pila){   recibe la pila como parámetro y retorna el valor contenido en el primer nodo
  int x;                el tipo de dato de la información para retornarlo
  Nodo* p = Pila;       un puntero al comienzo de la estructura para luego eliminar ese nodo
  x = pila→info;        conservar el valor del primer nodo para retornarlo
  pila = p→sgte;        Avanzar con la pila un nodo para eliminar el nodo que estaba en el tope
  delete p;             eliminar el que era primero, la pila esta apuntando al que era segundo o a NULL
  return x;             retornar la información que estaba en el primer nodo
}
```

Tenga en cuenta que para utilizar esta función la pila debe tener al menos un nodo. La pila solo permite push o pop, no admite recorrido. En push y en pop el puntero a pila SIEMPRE cambia. La pila invierte el orden

1
2
3
4
5

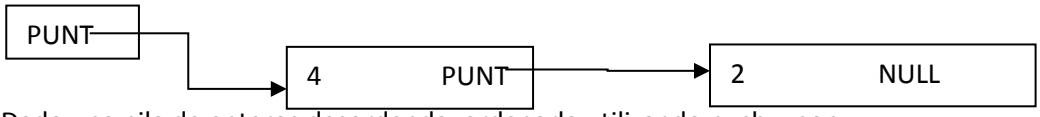
Secuencial

- **1er elemento lógico → primero físico**
- **sgte elemento lógico → siguiente físico**

0	2	1
1	3	4
2	5	-1
3	1	0
4	4	2

Enlazadas

- **posición 1er elemento logico**
- **c/elemento registro → nodo → info, siguente elemento logico**



Dado una pila de enteros desordenda, ordenarla utilizando push y pop

6
9
3
14
2

2
3
6
9
|4

Repasamos y avanzamos con colas

Puntero

Tipo de dato que contiene dirección de memoria a un dato o a un segmento de codigo

Operadores de dirección & indireccion * y →

```
int a = 10;
int* p = &a;
cout << *p << endl;

tr*q=new tr();
*q refiere al registro apuntado por q
(*q).c1 o su equivalente q→c1 acceso al camp1 del registro apuntadp por q
```

Asignación dinámica de memoria

```
int* p = new int();
*p = 10;
delete p;
```

Estructuras autoreferenciadas

Las estructuras enlazadas tienen, para cada elemento, una struct, que se llama nodo con un campo con la información y el otro con una referencia a una estructura del mismo tipo. En el caso de la implementación con asignación dinámica en memoria un campo con la información y otro campo, un punyero, con la dirección de memoria del siguiente nod, o el valor NUU en caso de no haber siguiente.

```
struct Nodo
{
   int info;  // valor que contiene el nodo
   Nodo* sig; // puntero al siguiente nodo
};
```

Punteros a estructuras: operadores de acceso a un miembro

```
Nodo* p = new(Nodo);
(*p). info; equivale a p->info
Lo dicho vale para el campo sig
```

Punteros a vectores dinámicos

```
tr* q = new tr[10]
```

Punteros y funciones: Punteros a funciones

int *fptr (int); // E.2
es el prototipo de una función que recibe un int y devuelve puntero-a-int
int (*fptr)(int); // E.1
es la declaración de un puntero-a-función que recibe un int y devuelve un int.
void func(int (*fptr)(int)); // E.3: Ok.
La expresión E.3 es el prototipo de una función que no devuelve ningún valor y recibe como argumento un puntero-a-función.

Estructuras enlazadas Aspectos generales

A diferencia de las estructuras secuenciales en las que el primer elemento lógico coincide con el primero físico y el siguiente lógico con el siguiente físico, por ejemplo si se quiere recorrer un vector secuencialmente, una estructura enlazada es una estructura tal que en cada posición tiene un registro al que se llama NODO, con al menos dos campos → uno con la información y el otro con la referencia al siguiente nodo. Estas estructuras además del nodo con esas características deben tener la referencia al primer elemento lógico qu puede o no ser el primero físico. Se pueden implementar con vectores pero lo implementaremos con asignación dinámica en memoria.

Los pasos son:

1. crear el nodo
2. declarar un puntero para controlar la estructura.
3. tomar la información
4. pedir memoria
5. guardar la información
6. actualizar los punteros

Estas estructuras pueden ser lineales, arbóreas o grafos, en EyED abordamos lineales que a su vez pueden ser Pilas, Colas y Listas, se diferencian, además del uso conceptual, por la forma en que se carga un nuevo dato o se extrae uno en particular.

Pilas → con un dato simple en el campo info

struct Nodo{ int info, Nodo* sgte}; →crear el nodo

Nodo* Pila = NULL; → "inicializar la estructura" hacer apuntar a NULL el puntero de control

```
void push(Nodo* &Pila, int x){
   Nodo * p = new Nodo();
   p→info = x;
   p→sgte = Pila;
   Pila = p;
   return;
}

int pop(Nodo* & Pila){
   int x;
   Nodo* p = Pila;
   x = p→info;
   Pila = p→sgte;
   detete p;
   return;
}
```

Pila → con una struct en el campo info

Crear la struct para la información: **struct tipoInfo{int c1, int c2}**

crear el nodo con una struct tipoInfo en el campo de la información: **struct Nodo{tipoInfo info; Nodo* sgte};**

los procedimientos push y pop solo difieren en el tipo de dato:

1. **push** solo en el prototipo cambiando el tipo de dato de x a tipoInfo
2. **pop** en el valor de retorno de la función debe retornar tipoInfo en lugar de int y, por supuesto, "x" debe ser de tipoInfo.

Esto requeriría desarrollar estos procedimientos para cada tipo de dato que querramos utilizar salvo que se desarrollen con plantillas utilizando plantillas o templates en C++

Para generalizar el tipo de dato con plantillas se requiere:

Nodo

```
template <typename T> struct Nodo
{
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};
```

push

```
template <typename T> void push(Nodo<T>*& Pila, T x)
{
  Nodo<T>* p = new Nodo<T>();
  p ->info = x;
  p->sgte = Pila;
  Pila = p;
  return;
}
```

pop

```
template <typename T> T pop(Nodo<T>*&)
{
  T x;
  Nodo<T>* p = Pila;
  x = p ->info;
  Pila = p->sgte;
  delete p;
  return x;
}
```

Estructura tipo cola

Esta estructura desde el punto de vista conceptual esta compuesta por un conjunto de nodos. Difiere de la pila en la forma en que se agrega un nuevo elemento o se extrae uno. El agregado se hace después del ultimo elemento, la razón de este nombre de la estructura es justamente por eso, como cualquier cola en cualquier contexto lo razonable es que el ingreso sea ubicándose después del ultimo. para extraer un elemento, siguiendo el criterio también de las colas se hace desde el tope, es decir el primero que se libera es el primero que ingreso por eso se las identifica como FIFO por las siglas en ingles. a los efectos de la implementación se dividirá en las dos situaciones diferentes que pueden presentarse para la carga, uniéndose luego en una función que contemple ambos casos. Lo mismo será tara eliminar nodos. La estructura pila requiere un puntero al principio ya que desde allí se inserta un elemento nuevo o se extrae uno. en el caso de la cola como se inserta después del ultimo es conveniente tener un puntero también a esta dirección.

Implementacion de una cola con datos simples

Declaracion del Nodo → sin diferencias a lo realizado para pilas

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

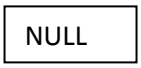
definicion de los punteros que controlan la estructura

Nodo* frente = NULL; Puntero al comienzo de la estruntura inicializada en NULL

Nodo* fin = NULL; Puntero al final de la estructura también inicializado en NULL

El procedimiento para agregar un dato en una cola lo llamaremos **queue** por su nombre en ingles. Para extraer un nodo lo llamaremos unqueue.

Nodo* frente=NULL



Nodo * fin = NULL

Agregar a una cola vacia donde frente y fin apuntan a NULL

Nodo * p = new Nodo();

p→info = x;

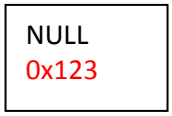
p→sgte = NULL; como p va a ser el ultimo nodo no tendra siguiente por lo que apunta a NULL

frente = p; como no había nodos el frente debe apuntar a este recine creado

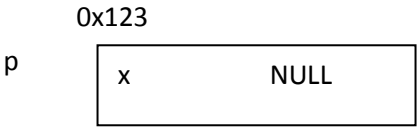
fin = p; el puntero al fin siempre apuntara a este nodo ya que fin apunta, al final que es donde se agrega

en este caso se modifican el inicio y el final, no había nodos, ahora hay uno que es primero y ultimo

frente



fin



p->info = x esto es siempre

p->sgte = NULL esto también es siempre

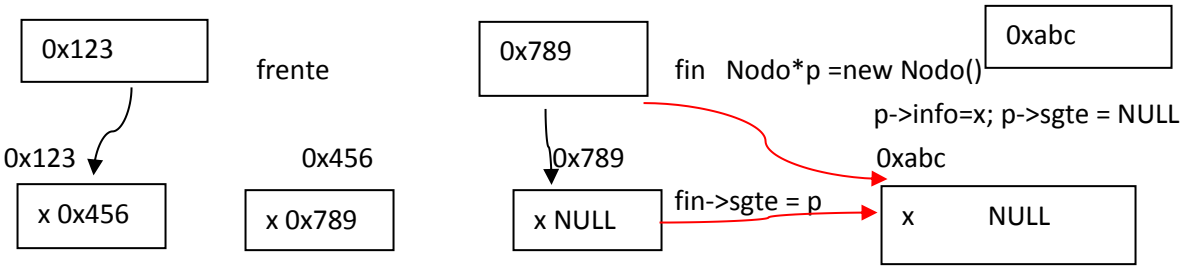
Nodo * p = new Nodo()

ahora el frente debe apuntar al nodo creado p. Solo en esta situación se modifica el frente

frente = p;

el puntero fin debe apuntar al nodo creado, esto ocurre siempre que se agrega un nodo pues se agrega al final

De este modo queda actualizada la cola con el primer nodo



Agregar a una cola con datos donde frente apunta al primer nodo y fin al ultimo

Nodo * p = new Nodo();

p→info = x;

p→sgte = NULL;

fin→sgte = p; se enlaza el nuevo nodo a continuación del que era el ultimo

fin = p; se redefine el nuevo fin apuntando a este nuevo nodo

Las cargas según lo visto solo cambian en la línea 4, en caso que la cola estuviera vacia y frente apuntara a NULL ahora, con un nodo debería apuntar a el. En el segundo caso el frente no cambia pero se hace necesario enlazar el nuevo nodo con el ex ultimo nodo que estaba apuntado por fin. Luego de esto debe cambiarse la dirección a la que este puntero apunta para que lo siga haciendo al ultimo

con dato de tipo simple

```
void queue(Nodo*&frente, Nodo* &fin, int X){
   Nodo * p = new Nodo();
   p→info = x;
   p→sgte = NULL;
   if (frente==NULL) frente = p; else fin→sgte = p;
   fin = p; se redefine el nuevo fin apuntando a este nuevo nodo
   return;
}
```

con plantilla

Nodo

```
template <typename T> struct Nodo {
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

template <typename T> void queue(Nodo<T>*& frente, Nodo<T>*&fin, T x)
{
  Nodo<T>* p = new Nodo<T>();
  p ->info = x;
  p->sgte = NULL;
  if (frente==NULL) frente = p; else fin→sgte = p;
  return;
}
```

Un análisis similar se puede hacer para eliminar nodos de una cola, la eliminación es similar al procediento pop de una pila, con la iferencia que al eliminar el ultimo nodo se debe hacer apuntar a NULL sl pintero al final

con dato de tipo simple

```
int unqueue(Nodo*&frente, Nodo* &fin){
   int x;
   Nodo * p = frente;
   x = p→info;
   frente = p→sgte;
   if (frente==NULL) fin = NULL;
   delete p;
   return x;
}
```

con plantilla

Nodo

```
template <typename T> struct Nodo {
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

template <typename T> T unqueue(Nodo<T>*& frente, Nodo<T>*&fin)
{
  T x;
  Nodo<T>* p = frente;
  x = p ->info;
  frente = p->sgte;
  if (fte==NULL) fin = NULL;
  delete p;
  return x;
}
```

```
Ejemplos de uso con datos simples
int main()
{
   Nodo<int>* pila = NULL;
   Nodo<int>* frente = NULL;
   Nodo<int>* fin = NULL;

   push<int>(pila,10);
   push<int>(pila,20);
   push<int>(pila,30);
   while( pila!=NULL )cout << pop<int>(pila) << endl;

   queue<int>(frente, fin ,20);
   queue<int>(frente, fin ,30);
   while( frente!=ULL )cout << unqueue<int>(frente, fin) << endl;

   return 0;
}



Ejemplo con estructuras
struct Alumno
{
   int legajo;
   string nombre;
};

// función auxiliar para cargar los datos de la struct
Alumno cargarRegistro(int leg, string nom)
{
   Alumno a;
   a.leg = leg;
   a.nom = nom;
   return a;
}
```

```
int main()
{
   Nodo<Alumno>* pila = NULL;
   Alumno a = cargarRegistro(1,"Roxana");
   push<Alumno>(pila a;

   Alumno a = cargarRegistro(2,"Oscar");
   push<Alumno>(pila a;


   while( pila )
   {
      a = pop<Alumno>(pila);
      cout << a.leg << ", " << a.nom << endl;
   }
   return 0;
}
```

listas ordenadas simplemente enlazadas

Como hemos visto las estructuras enlazadas pueden ser lineales (pilas, colas, listas) orboreas o grafos.

En las estructuras lineales lo que las diferencia es donde se agrega un dato nuevo y/o de qué lugar se lo extrae. en el caso de las pilas y las colas tienen lugares específicos para ello: En pilas se agrega delante del primer nodo (la metáfora de la pila de platos), en las colas detrás del ultimo (metáfora de cola en los colectivos), en ambas estructuras también se extraen los datos de un lugar determinado: el tope.

Las listas no tienen esta restricción; se puede insertar un valor delante del primero, al final de la estructura o en medio de dos valores particulares. En general se las inserta siguiendo un determinado criterio de ordenamiento. Esta estructura puede ser recorrida con propósitos diversos, se puede modificar un dato particular, se puede eliminar un nodo que este al principio, o al final, o en medio de dos particulares.

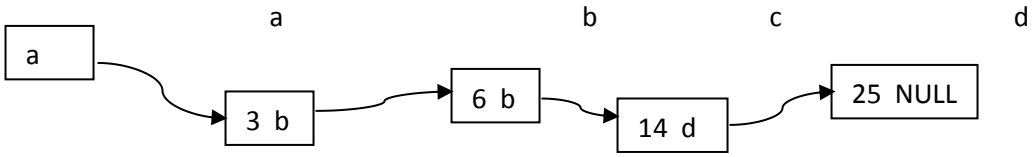
Esta estructura esta en memoria, pero los datos no necesariamente están en posiciones contiguas como ocurre con los vectores.

Veamos

int v[4]

3
6
14
25

Lista

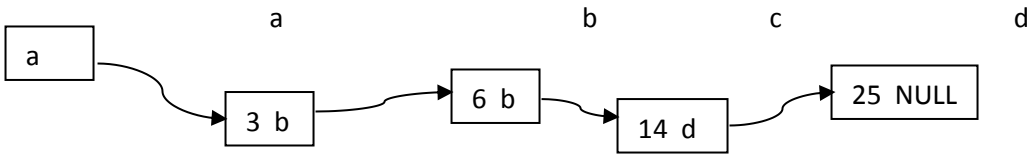


Nombre	Tipo	valor
v	vector	3, 6, 14, 25
lista	puntero	Direccion de memoria a
v[0]	int	3
lista→info	int	3
lista→sgte	puntero	Direccion de memoria b
v[1]	int	6
lista→sgte→info	int	6
lista→sgte→sgte	puntero	Direccion de memoria C
v[2]	int	14
lista→sgte→sgte→info	int	14
lista→sgte→sgte→sgte	puntero	Direccion de memoria D
v[3]	int	25
lista→sgte→sgte→sgte→info	int	25

Abordaremos distintos patrones para insertar nodos en una lista, supondremos ordenadas en forma creciente. Abordaremos la implementación con datos simples y luego con plantillas Los patrones a abordar serán:

- ✓ Insertar el primer nodo de una lista o delante del primero
- ✓ insertar después del ultimo
- ✓ Insertar en una posición determinada
- ✓ Insertar ordenado → donde corresponda según el criterio de ordenamiento
- ✓ buscar un valor en la lista
- ✓ insertar ordenado sin repetir la clave
- ✓ cambiar el criterio de ordenamiento
- ✓ Eliminar nodos
  - el primero
  - el ultimo
  - uno determinado según su valor
  - todos los nodos de la estructura
- ✓ Mostrar todos los valores de la lista
  - eliminando el mostrado
  - sin eliminar el mostrado

Lista




```
p = lista;                  p=lista;
while(p)                    while (p->sgte)
        p=p->sgte;                  p=p->sgte;
 X=8;          X=30;
```

p	p->info	p->sgte	p->sgte->info
v		vector	3, 6, 14, 25

○

<table>
<tr><th>Con datos simples</th><th>Con plantillas y/o estructuras</th></tr>
<tr><td>Insertar delante del primero: similar a push
retorna la dirección del nodo creado

Nodo * insertaPrimero(int*& l, int x)
{
  int* p = new Nodoint();
  p ->info = x;
  p->sgte = l;
  l = p;
  return; p
}</td><td>Nodo
template <typename T> struct Nodo{
T info;   // valor que contiene el nodo
Nodo<T>* sig; // puntero al siguiente nodo
};

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

Ejemplo de uso
struct tr{int c1; int c2};
Nodo<int>* l1 = NULL;
Nodo<tr>* l2 = NULL;
tr r; r.c1 = 2; r.c2 = 15;
insertaPrimero<int>(l1, 25);
insertaPrimero<tr>(ls, a);</td></tr>
<tr><td>Agregar al final: Se debe recorrer la lista hasta llegar a apuntar al ultimo nodo, de este se debe apuntar al nuevo

Nodo* insertarAlFinal(int*& l, int x){
  Nodo* nuevo = new Nodo();
  nuevo->info = x;
  nuevo->sig = NULL;

  if(l==NULL) {
    l = nuevo; }
  else {
    Nodo* aux = l;
    while(aux->sig!=NULL)
      aux = aux->sig;

    aux->sig = nuevo;
  }
return nuevo;
}</td><td>template<typename T>
Nodo<T>* insertarAlFinal(Nodo<T>*& l, T x){
  Nodo<T>* nuevo = new Nodo<T>();
  nuevo->info = x;
  nuevo->sig = NULL;

  if(l==NULL) {
    l = nuevo; }
  else {
    Nodo* aux = l;
    while(aux->sig!=NULL)
      aux = aux->sig;

    aux->sig = nuevo;
  }
return nuevo;
}</td></tr>
</table>

Buscar: si un valor determinado se encuentra en la lista retorna el puntero a ese nodo; de lo contrario retorna NULL

Version 1: se supone No Ordenada y dato simple

```
Nodo* buscar(Nodo* l, int v){
  Nodo* aux = l;

while( aux!=NULL && aux->info!=v )   {
     aux=aux->sig;
  }
  return aux;
}
```

Versión 2 si esta ordenada puede salir antes del final

```
Nodo* buscar(Nodo* l, int v){
  Nodo* aux = l;

while( aux!=NULL && aux->info!=v )   {
     aux=aux->sig;
  }
  return (uux!=NULL&&aux->info==v)?aux:NULL);
}
```

Ejemplo completo de busqueda

```
int main(){
  Nodo<Alumno>* p = NULL;

agregar<Alumno>(p,CargarRegistro(1,"Roxana") );

agregar<Alumno>(p,cargarRegistro(2,"Oscar"));

  int leg;
  cout << "Ingrese el legajo a buscar: ";
  cin >> leg;

   // busca por legajo
  Nodo<Alumno>* r =
 buscar<Alumno,int>(p,leg,criterioAlumnoLeg);
  if( r!=NULL )
    cout << r->info.leg << ", " << r->info.nom << endl;

  string nom;
```

```
template <typename T>
Nodo<T>* buscar(Nodo<T>*l, T v){
Nodo<T>* aux = l;

while( aux!=NULL && aux->info!=v )   {
     aux=aux->sig;
  }
  return aux;
}
template <typename T>
Nodo<T>* buscar(Nodo<T>*l, T v){
Nodo<T>* aux = l;

while( aux!=NULL && aux->info!=v )   {
     aux=aux->sig;
  }
return (uux!=NULL&&aux->info==v)?aux:NULL);
}
```

Version 3 con un criterio diferente de ordenamiento utilizando punteros a funciones

```
template <typename T, typename K>
Nodo<T>*buscar(Nodo<T>*l, K v, int (*criterio)(T,K) )
{
  Nodo<T>* aux = p;
  while( aux!=NULL && criterio(aux->info,v)!=0 )   {
     aux = aux->sig;
  }
  return aux;
}
```

Ejemplo de uso

```
// funciones con distintos criterios de busqueda
int criterioAlumnoLeg(Alumno a,int leg){
  return a.leg-leg;
}
```

```
  cout << "Ingrese el nombre a buscar: ";
  cin >> nom;

   // busc por nombre
  r =

buscar<Alumno,string>(p,nom,criterioAlumnoNom);

  if( r!=NULL )
    cout << r->info.leg << ", " << r->info.nom << endl;

  return 0;
}
```

```
int criterioAlumnoNom(Alumno a,string nom){
  return strcmp(a.nom.c_str(),nom);
}
```

La invocación

Se pasa la lista, el valor buscado y el puntero a la función que lo evaluara

```
Nodo<Alumno>* r =

buscar<Alumno,int>(l,leg,criterioAlumnoLeg);

Nodo<Alumno>* r =

buscar<Alumno,int>(l,nom,criterioAlumnoNom);
```

Mostrar el contenido de una lista,

Version 1 conservando los valores

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

Version 2 Eliminando el nodo luego de mostrar

```
void mostrar(Nodo* & l) {

 while( l )
    cout << pop(l) << endl;

  return;
}
```

Liberar la memoria de todos los nodos de una lista

Version 1 avanzando el puntero manualmente

```
void suprimirr(Nodo*& l)
{
  Nodo* ant;
  while( l!=NULL )  {
    ant = l;
    l = l->sig;
```

```
    delete ant;
  }
 return
}
```

Version 2 usando funciones ya desarrolladas

```
void liberar(Nodo*& p){
  while( l )
    pop(l);
  return;
}
```

eliminarNodo elimina un nodo con un valor dado si lo encuentra

```
void eliminarNodo(Nodo*& l, int v){
  Nodo* actual = l;
  Nodo* ant = NULL;

  while( actual!=NULL && aux->info!=v ) {
    ant = actual;
    actual = actual->sig;
  }
  If(actual->info!=v return; //no lo encontro
  if( ant!=NULL )//si no es el primer nodo
    ant->sig = actual->sig;
  else//si se elimina el primero
    l = actual->sig;//actualiza puntero al inicio

  delete actual;
return;
}
```

```
template <typename T, typename K>
void eliminar(Nodo<T>*& p, K v, int
(*criterio)(T,K)){
  Nodo<T>* actual = l;
  Nodo<T>* ant = NULL;

  while( actual!=NULL && criterio(aux-
>info,v)!=0 )   {
    ant = aux;
    aux = aux->sig;
  }
  If(actual->info!=v return;
  if( ant!=NULL )
    ant->sig = aux->sig;
   else
    p = aux->sig;

  delete aux;
return;
}
```

Insertar ordenado

```
Nodo* insertarOrdenado(Nodo*& l, int v){

 Nodo* nuevo = new Nodo();
  nuevo->info = v;
  nuevo->sig = NULL;

  Nodo* ant = NULL;
  Nodo* actual = l;

  while( actual!=NULL && aux->info<v ) {
    ant = actual;
    actual = actual->sig;
  }

  if( ant==NULL )
     l = nuevo;
```

```
template <typename T>
Nodo<T>* insertarOrdenado(Nodo<T>*& p,
T v, int (*criterio)(T,T) ){
  Nodo<T>* nuevo = new Nodo<T>();
  nuevo->info = v;
  nuevo->sig = NULL;

  Nodo<T>* ant = NULL;
  Nodo<T>* actual = p;

 while( aux!=NULL && criterio(aux-
>info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }

  if( ant==NULL )
```

```
  else
    ant->sig = nuevo;

  nuevo->sig = actual;

  return nuevo;
}
```

```
    l = nuevo;
  else
    ant->sig = nuevo;

  nuevo->sig = aux;

  return nuevo;
}
```

Insertar sin repetir la clave: Inserta solo en caso que el valor a agregar no se encuentre en la lista retorna
(encontró el valor? puntero a ese nodo:puntero al nuevo que inserta)

```
Nodo* cargarSinRepetir(Nodo*& l,tr v,
int (*criterio)(T,T)){
  Nodo* x = buscar(l,v,);

  if( x == NULL )
  {
    x = insertarOrdenado(p,v,);
  }

  return x;
}
```

```
template <typename T>
Nodo<T>* cargarSinRepetir(Nodo<T>*& l,T
v,
int (*criterio)(T,T)){
  Nodo<T>* x = buscar<T,T>(p,v,criterio);

  if( x == NULL )
  {
    x = insertarOrdenado<T>(p,v,criterio);
  }

  return x;
}
```

Una aplicación completa

```
struct TipoRegistro // para un nodo con una strucct en el campo info
{
  int c1;
  string c2;
}

template <typename T> struct Nodo
{
  T info;      // valor que contiene el nodo
  Nodo<T>* sig; // puntero al siguiente nodo
};

// para dato simple será:        Nodo<int>* l1 = NULL;
// para nodo con struct será:    Nodo<TipoRegistro>* l2 = NULL;
```

```
//Insertar ordenado con plantilla y criterio
template <typename T>
Nodo<T>* insertarOrdenado(Nodo<T>*& p, T v, int (*criterio)(T,T) ){
  Nodo<T>* nuevo = new Nodo<T>();
  nuevo->info = v;
  nuevo->sig = NULL;
```

```
  Nodo<T>* actual = p;
  Nodo<T>* ant = NULL;
  while( actual!=NULL && criterio(actual->info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }
  if( ant==NULL )  p = nuevo;
   else ant->sig = nuevo;
  nuevo->sig = actual;
  return nuevo;
}
```

```
Nodo* insertarOrdenado(Nodo*& p, TipoRegistro v) {
  Nodo* nuevo = new Nodo();
  nuevo->info = v;
  nuevo->sig = NULL;
  Nodo<T>* actual = p;
  Nodo<T>* ant = NULL;
  while( actual!=NULL && criterio(actual->info,v)<=0 ) {
    ant = actual;
    actual = actual->sig;
  }
  if( ant==NULL )  p = nuevo;
   else ant->sig = nuevo;

  nuevo->sig = actual;
  return nuevo;
}
```

```
// funciones auxiliares de criterio
int criterioCampo1Asc(TipoRegistro a,int c1) {
  return a.c1-c1;      // si el dato a cargar es mayor retorna negativo y sigue en el while
}
int criterioCampo1Desc(TipoRegistro a,int c1) {
  return c1 - a.c1;  // si el dato a cargar es menor retorna positivo  y sale del while
}
int criterioCampo2(TipoRegistro a,string c2){
  return strcmp(a.c2.c_str(),c2);
}
```

```
TipoRegistro cargarRegistro(int c1, string c2)
{
  Alumno a;
  a.leg = leg;
  a.nom = nom;
  return a;
}
```

```
int main()
{
  Nodo<TipoRegistro>* l1 = NULL;        // lista con plantilla por c1 asc
  Nodo* l2 = NULL;                      // lista sin plantilla
  Nodo<TipoRegistro>* l3 = NULL;        // lista con plantilla por c1 desc
  Nodo<TipoRegistro>* l4 = NULL;        // lista con plantilla por c2


 // carga ordenado por c1 ascendente
  insertarOrdenado<TipoRegistro>(l1,cargarRegistro(1,"Roxana")criterioCampo1Asc);
  insertarordenado<TipoRegistro>(l1,cargarRegistro(2,"Oscar")criterioCampo1Asc);

// carga ordenado por c1 descendente
  insertarOrdenado<TipoRegistro>(l3,cargarRegistro(1,"Roxana")criterioCampo1Desc);
  insertarordenado<TipoRegistro>(l3,cargarRegistro(2,"Oscar")criterioCampo1Desc);

// carga ordenado por c2
  insertarOrdenado<TipoRegistro>(l4,cargarRegistro(1,"Roxana")criterioCampo2);
  insertarordenado<TipoRegistro>(l4,cargarRegistro(2,"Oscar")criterioCampo2);

 // carga sin plantilla
  insertarOrdenado(l2,cargarRegistro(1,"Roxana"));
  insertarordenado(l4,cargarRegistro(2,"Oscar"));


……..
```

Ejercicio Nro. 1:

Dada una pila y un valor X colocar el vañor x en la tercera posicion se la pila, retornando un parámetro con valor **'S'** o **'N'** según haya sido exitoso o no el requerimiento. (Definir parámetros y codificar).

Ejercicio Nro. 2:

Dada una pila y dos valores **X** e **I**, desarrollar un procedimiento que inserte el valor **X** en la posición **I** de la pila si es posible. (Definir parámetros y codificar).

Ejercicio Nro. 3:

Dada una pila y un valor **X**, desarrollar un procedimiento que inserte el valor **X** en la última posición de la pila y la retorne. (Definir parámetros y codificar).

Ejercicio Nro. 4:

Dada una pila y dos valores **X** e **Y**, desarrollar un procedimiento que reemplace cada valor igual a **X** que se encuentre en la pila por el valor **Y** retornando la pila modificada. En caso de no haber ningún valor igual a **X** retornar la pila sin cambio. (Definir parámetros y codificar).

Ejercicio Nro. 5:

Definir una función **INVERSA** que evalúe dos conjuntos de caracteres separados por un punto y retorne True si los conjuntos son inversos (ej: ABcDe.eDcBA) o False si no lo son. Los conjuntos deben ingresarse por teclado. (Definir parámetros y codificar).

Ejercicio Nro. 6:

Desarrollar un procedimiento que ingrese por teclado un conjunto de Apellidos y Nombre de alumnos y los imprima en orden inverso al de ingreso. (Definir parámetros y codificar).
Ejercicio Nro. 7:

Dada una pila desarrollar un procedimiento que ordene la misma de acuerdo al valor de sus nodos y la retorne. Solo se deben usar pilas. (Definir parámetros y codificar).

Ejercicio Nro. 8:

Dada una cola (nodo = registro + puntero), desarrollar y codificar un procedimiento que elimine 2 nodos de la misma (indicar con un parámetro 'S'/'N' si ello fue, o no posible)

Ejercicio Nro. 9:

Dada una cola (nodo = registro + puntero), desarrollar y codificar una función que devuelva la cantidad de nodos que tiene.

Ejercicio Nro. 10:

Dadas dos colas **COLA** y **COLB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere una única cola **COLAB** a partir de ellas. (Primero los nodos de **COLA** y luego los de **COLB**).

Ejercicio Nro. 11:

Dada una cola (nodo = registro + puntero), imprimirla en orden natural si tiene más de 100 nodos, caso contrario imprimirla en orden inverso.

Ejercicio Nro. 12:

Dadas dos colas **COLA** y **COLB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere otra cola **COLAB** por apareo del campo **ARRIBO** del registro (define orden creciente en ambas).
Nota: **COLA** y **COLB** dejan de ser útiles después del apareo.

Ejercicio Nro. 13:

Dado un archivo de registros de alumnos, donde cada registro contiene:
a) Apellido y Nombre del alumno (35 caracteres) b) Número de legajo (7 dígitos)
c) División asignada (1 a 100)
ordenado por número de legajo, desarrollar el algoritmo y codificación del programa que imprima el listado de alumnos por división, ordenado por división y número de legajo crecientes, a razón de 55 alumnos por hoja.

Ejercicio Nro. 14:

Idem Ejercicio Nro. 13, pero el listado de alumnos por división debe realizarse ordenado creciente por división y decreciente por número de legajo.

Ejercicio Nro. 15:

Idem Ejercicio Nro. 13 pero considerando que las divisiones asignadas son 100 y se identifican con un código de 4 caracteres.

Ejercicio Nro. 16:1

Dado un arreglo de **N** (< 30) colas (nodo = registro + puntero), desarrollar y codificar un procedimiento que aparee las colas del arreglo en las mismas condiciones que las definidas en el Ejercicio Nro. 12.
Nota: Retornar la cola resultante y no mantener las anteriores.

Ejercicio Nro. 17:

Dada una lista (nodo = registro + puntero), desarrollar y codificar una función que devuelva la cantidad de nodos que tiene.

Ejercicio Nro. 18:

Dadas dos listas **LISTA** y **LISTB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere una única lista **LISTC** a partir de ellas. (Primero los nodos de **LISTA** y luego los de **LISTB**).

Ejercicio Nro. 19:

Dada una **LISTA** (nodo = registro + puntero), imprimirla en orden natural si tiene más de 100 nodos, caso contrario imprimiría en orden inverso.

Ejercicio Nro. 20:

Dadas dos listas **LISTA** y **LISTB** (nodo = registro + puntero), desarrollar y codificar un procedimiento que genere otra lista **LISTC** por apareo del campo **LEGAJO** del registro (define orden creciente en ambas).
Nota: **LISTA** y **LISTB** dejan de ser útiles después del apareo).

Implementacion en C de listas circulares

```
struct Nodo {
   char dato[50+1];
   Nodo* siguiente;
};
```

La lista circular, para poder controlarse requiere un puntero al inicio de la estructura y un puntero al final. En este ejemplo se utiliza una struct que contiene este puntero. Esto es una decisión particular, desde luego que se puede optar por dos punteros separados, creemos que concentrarlos en una struct es mas ordenado y simple

```
 struct Lista {// el tipo creado, como se observa, se escribe con MAYUSCULA inicial
   Nodo* inicio;
   Nodo* fin;
   int tamanio;
};
Lista lista; //lista, en minuscula, es una struct con puntero al inicio, al final y cantidad de elementos
```

Operaciones sobre las listas circulares
Inicialización
Prototipo de la función:

```
void inicialización (Lista& lista); // recibe una struct pasado por referencia
```

Definicion de La función:

```
void inicialización (Lista& lista){
   lista.inicio = NULL;
   lista.fin = NULL;
   lista.tamanio = 0;
}
```

```
Nodo* insListaCircVacia(Lista& lista, char dato[]){
Nodo *nuevo = new(Nodo);
nuevo->siguiente = nuevo;
strcpy(nuevo->dato, dato);
lista.inicio = nuevo;
lista.fin = nuevo;
lista.tamaño=1;
return nuevo;
}
```

```
Nodo* insListaCircNoVacia(Lista& lista, char dato[]){
Nodo* nuevo = new Nodo();
strcpy (nuevo->dato, dato);
nuevo->siguiente = actual->siguiente;
actual->siguiente = nuevo;
lista->fin = nuevo;
lista->tamaño++;
return nuevo;
}
```