Listas

DOCENTE – FEDERICO VILENSKY

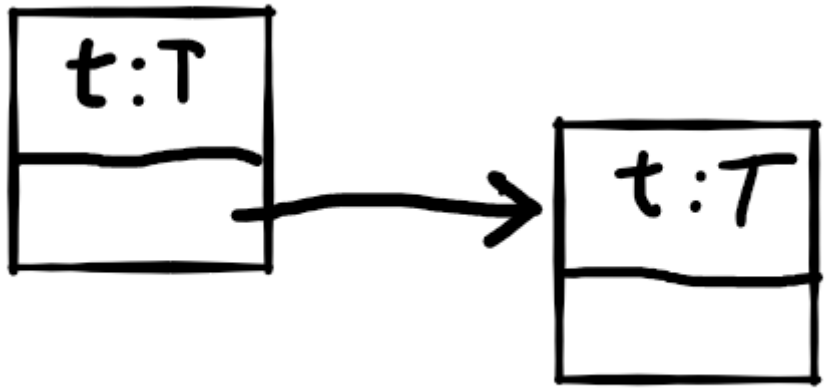
Lista encadenada

- Esta es la primera estructura que vamos a ver en el curso
- Una lista encadenada es una estructura lineal
  - Esta formada por una cadena de nodos
  - Cada nodo apunta a su sucesor en la lista

Lista encadenada

- Podemos definir a los nodos de la lista como
- ```
  struct nodoLista {
      T info;
      nodoLista * sig;
  }

  typedef nodoLista * Lista
  ```

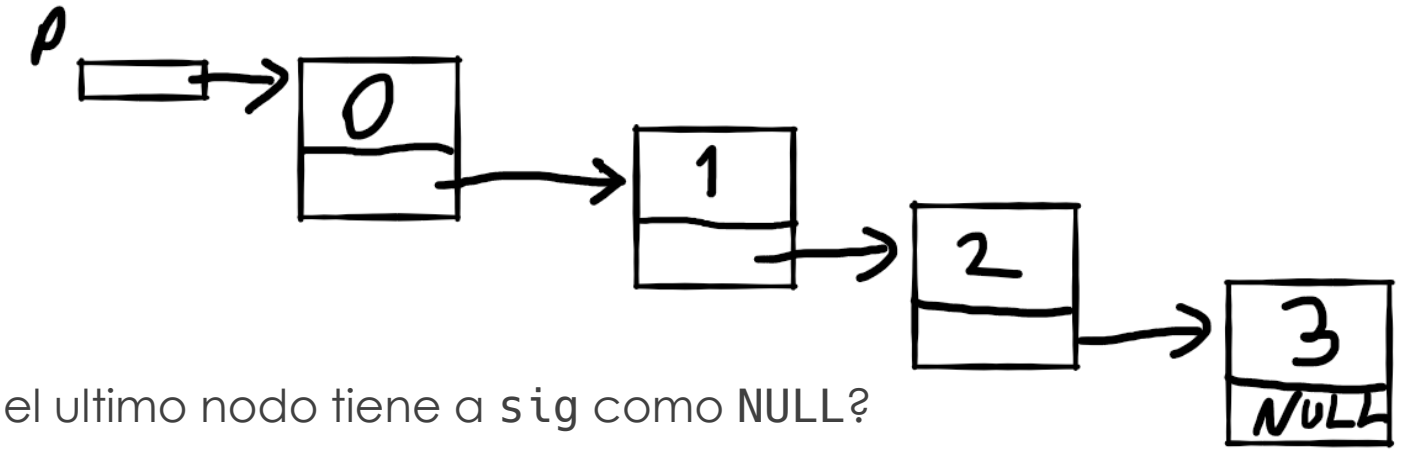

  ```

Lista encadenada

- Para la definición asumimos que los datos contenidos en cada uno de los nodo son de tipo T
- Para acceder a la lista, vamos a tener una variable de tipo puntero a nodoLista
  - `Lista p;`
  - Para esto es que hicimos la línea `typedef nodoLista * Lista`
  - Es lo mismo que haber puesto `nodoLista * p;`
- Esta variable va a apuntar a la dirección en memoria del principio de la lista (al primer nodo)

Lista encadenada

- Veamos que pasa si decimos que T es `int`
- Un ejemplo de esto sería



- Por qué el ultimo nodo tiene a `sig` como `NULL`?
  - Esto indica que no tiene ningún elemento que le sigue
  - O sea que es el ultimo elemento

Operaciones sobre listas

- Vamos a ver algunas operaciones que nos sirven para manipular las listas
- Vamos a estar trabajando con un puntero `p`, y eventualmente con otros punteros auxiliares
  - `Lista p;`
- `p` apunta al principio de la lista
- La primera operación que vamos a ver es la más simple, que es la creación de una lista vacia
  - `p = NULL;`

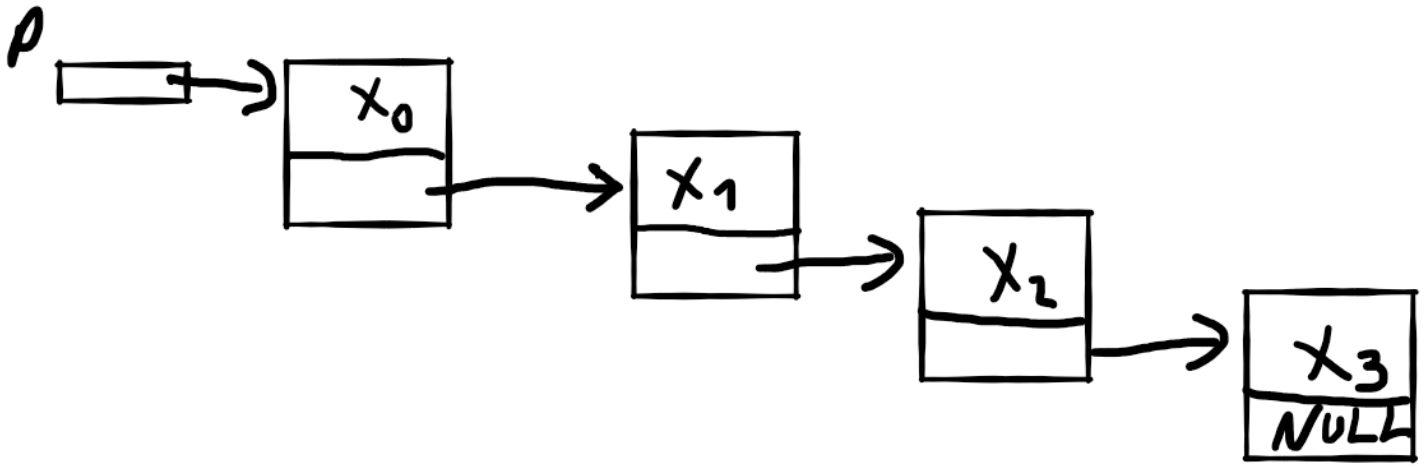
Operaciones sobre listas – esVacia

- Dada una lista, retorna true sii la lista está vacía

```
bool esVacia(Lista p){
    return p == NULL;
}
```

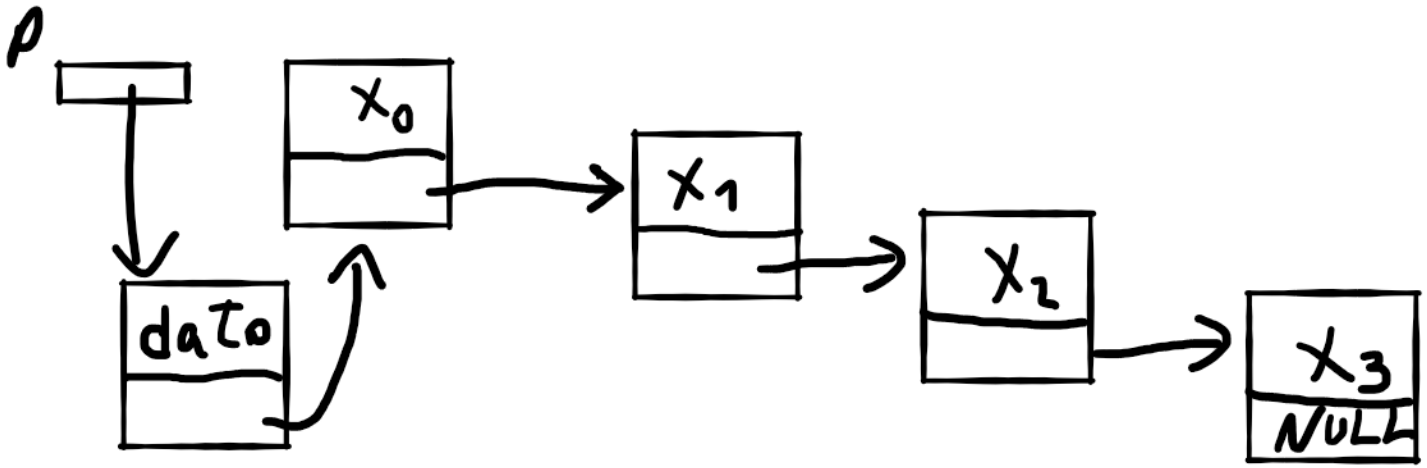
Operaciones sobre listas – insertar

- Dada una lista, queremos agregar un nuevo nodo al principio de la lista
- Supongamos que el nuevo dato está en una variable `dato` de tipo T



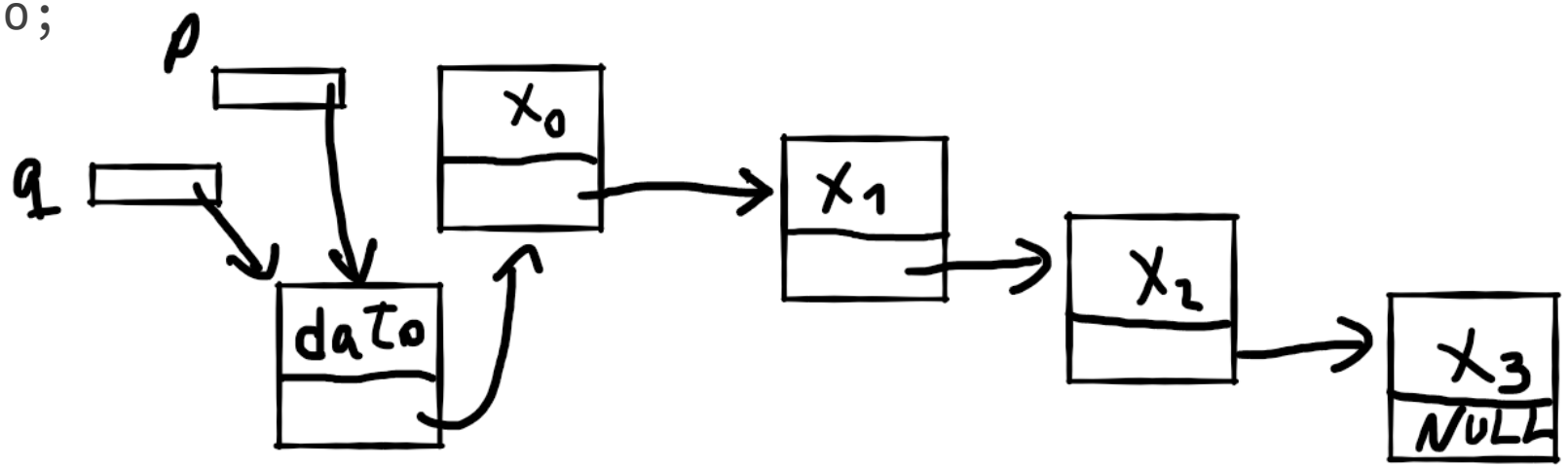
Operaciones sobre listas – insertar

- Dada una lista, queremos agregar un nuevo nodo al principio de la lista
- Supongamos que el nuevo dato está en una variable `dato` de tipo T



Operaciones sobre listas – insertar

- Vamos a definirnos `q`, un puntero a un `nodoLista`, que vamos a usar como variable auxiliar
-
```
nodoLista *q;
q = new nodoLista;
q -> info = dato;
q -> sig = p;
p = q;
```

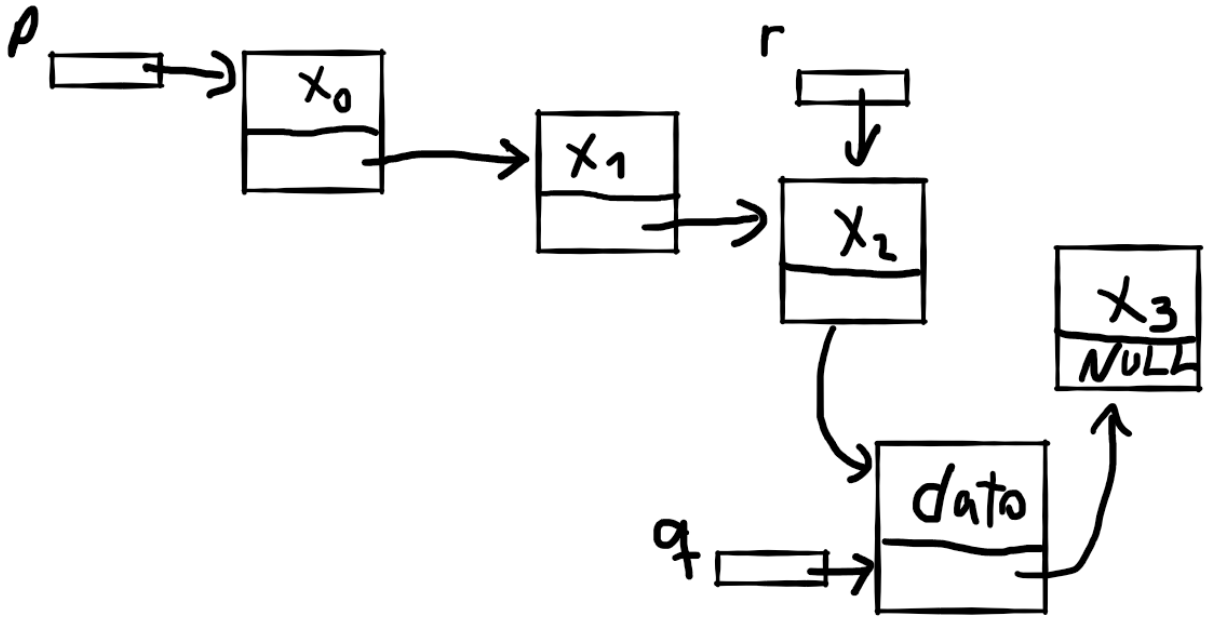


Armar lista de 0 a n

- Asumiendo que T es int
- Cómo podemos utilizar este procedimiento, de ir agregando al principio de la lista, para armar una lista de 0 a n?
- ```
  p = NULL;
  nodoLista * q;
  while (n > 0)
  {
      q = new nodoLista;
      q->info = n;
      q->sig = p;
      p = q;
      n--;
  }
  ```
  ```

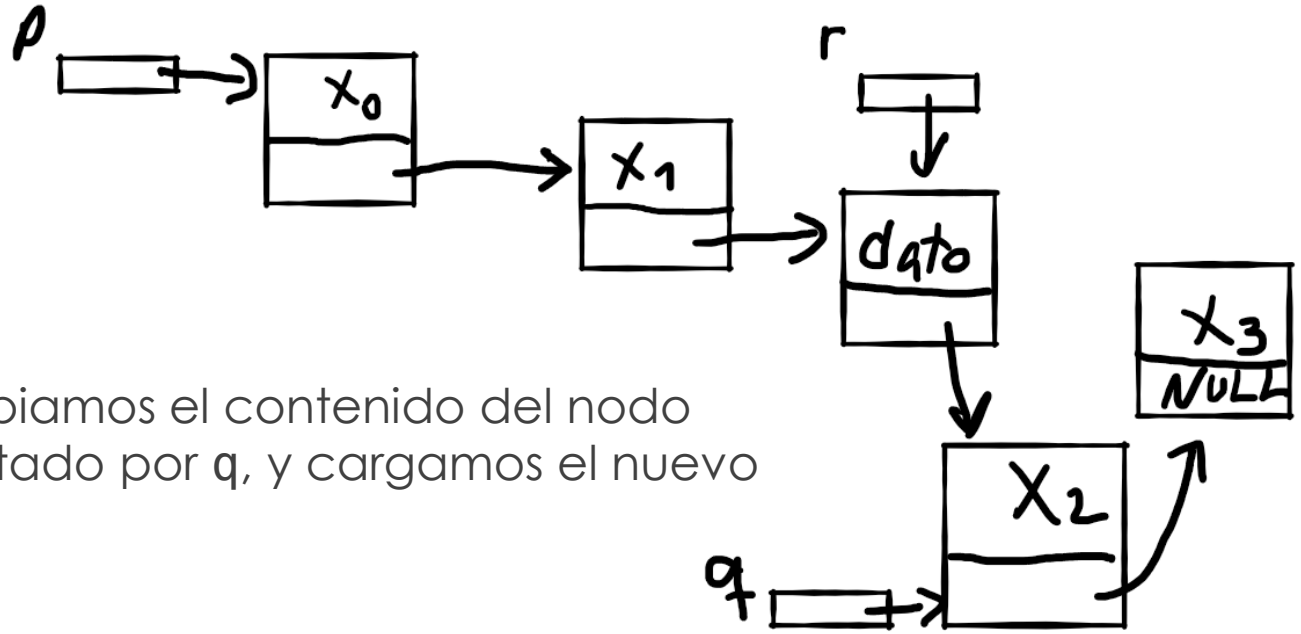
Operaciones sobre listas – insertar luego de r

- A veces nos interesa agregar un nuevo nodo en la mitad de una lista
- Imaginemos nos dan `r`, un puntero a un `nodoLista`, y nos piden insertar un nuevo nodo con `dato` de tipo T, después del nodo a donde apunta `r`
  - ```
    nodoLista * q;
    q = new nodoLista;
    q->info = dato;
    q->sig = r->sig;
    r->sig = q;
    ```


    ```

Operaciones sobre listas – insertar antes de r

- Otro caso es que nos pidan insertar el `dato` antes del apuntado por `r`
- ```
  nodoLista *q;
  q = new nodoLista;
  q->info = r->info;
  q->sig = r->sig;
  r->info = dato;
  r->sig = q;
  ```
- Lo que estamos haciendo acá es copiamos el contenido del nodo apuntado por `r` al nuevo nodo apuntado por `q`, y cargamos el nuevo dato en `r`

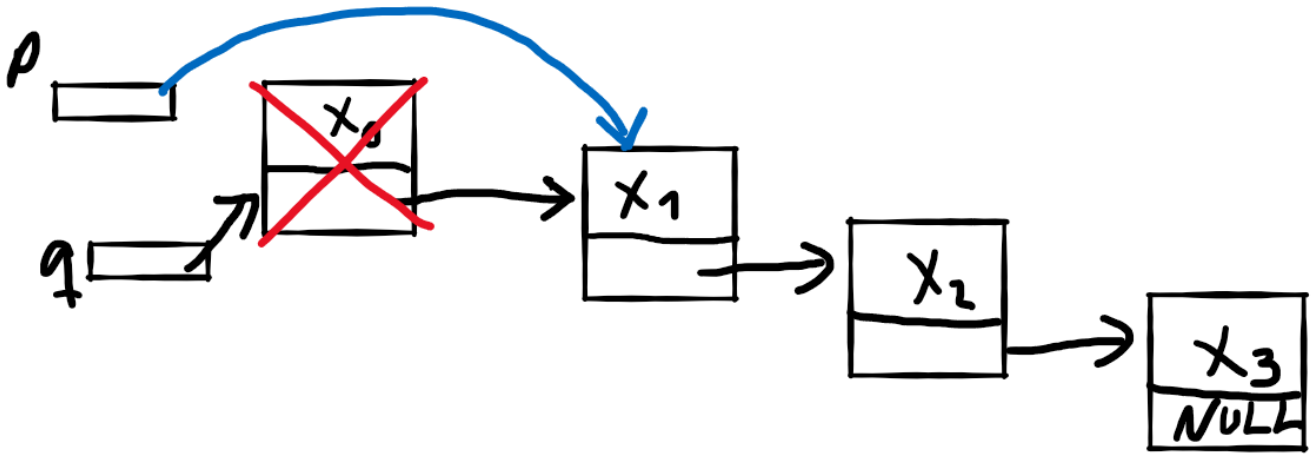

  ```

Operaciones sobre listas – borrar

- Lo siguiente que vamos a ver es como borrar elementos de una lista
- Vamos a empezar por los casos mas simples
  - Borrar el primer elemento
    - Suponiendo que la lista no es vacía
  - Borrar el sucesor de un elemento
    - Suponiendo que no es el ultimo

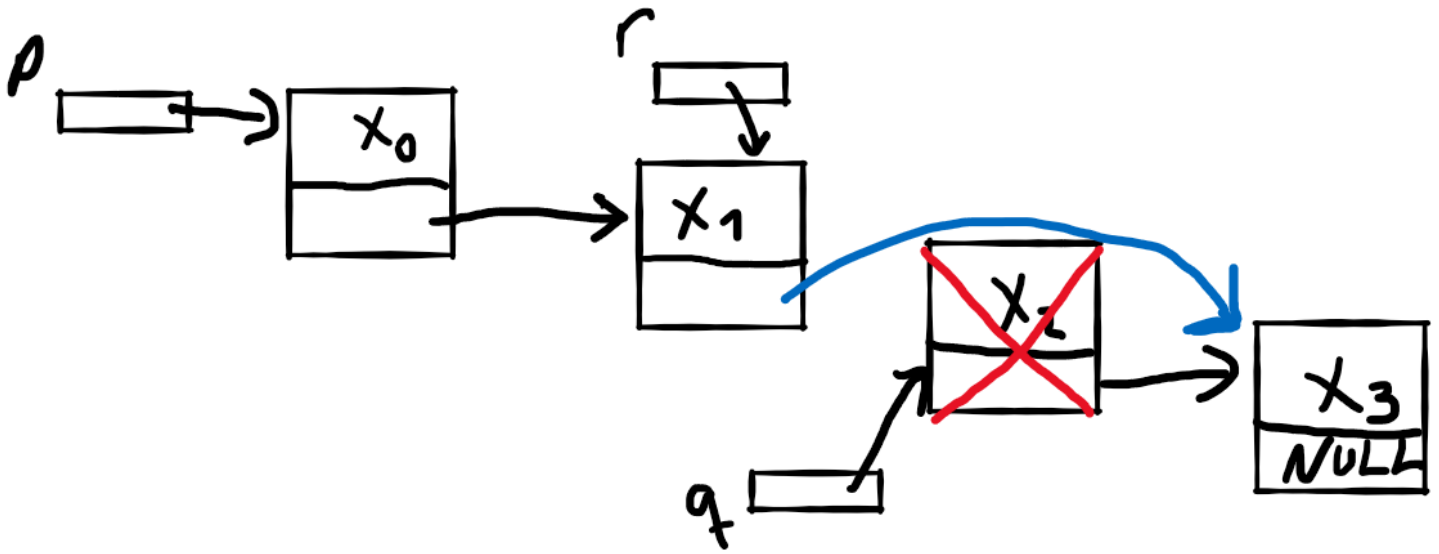
Operaciones sobre listas – borrar 1o

- Para borrar el primer elemento de la lista hacemos
- ```
  nodoLista *q;
  q = p;
  p = p->sig; //borrado lógico
  delete q; //borrado físico
  ```


  ```

Operaciones sobre listas – borrar siguiente

- Ahora queremos borrar el siguiente al nodo apuntado por r
- nodoLista *q;
  q = r->sig;
  r-sig = q->sig;
  delete q;



Dudas hasta ahora?

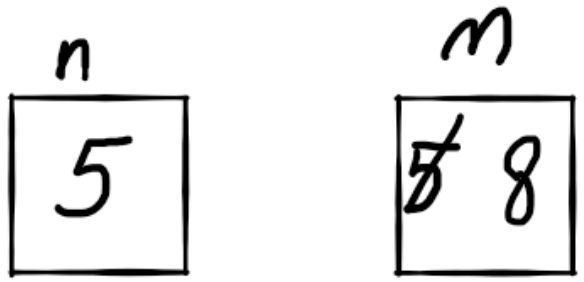
Paréntesis

- Vamos a hacer un paréntesis para hablar del pasade de parámetros a funciones
- El pasaje puede ser
  - Por valor
  - Por referencia

Pasaje por valor

- ```
  void proc(int m){ m = 8; }

  int main()
  {
      int n = 5;
      proc(n);
      cout << n << endl;
  }
  ```

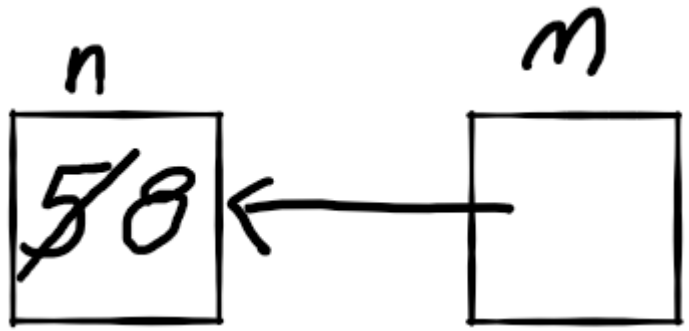


- Qué imprime?
  - 5
- En C/C++ el pasaje de parámetros es siempre por valor
  - Cuando pasamos un argumento a una función, se realiza una copia del mismo, y esa copia es la que recibe la función
  - Esto significa que si modificamos la copia, el original no se va a ver afectado
  ```

Pasaje por referencia

- ```
  void proc(int *m){ *m = 8; }

  int main()
  {
      int n = 5;
      proc(&n);
      cout << n << endl;
  }
  ```



- Qué imprime?
  - 8
- Cuando queremos poder modificar el argumento que recibe la función, pasamos la dirección de memoria a la variable (es decir que el parámetro de la función es un puntero)
  - Luego usando el operador *, podemos acceder al valor original
  - La dirección de memoria **&n** se pasa por valor de todas formas
  ```

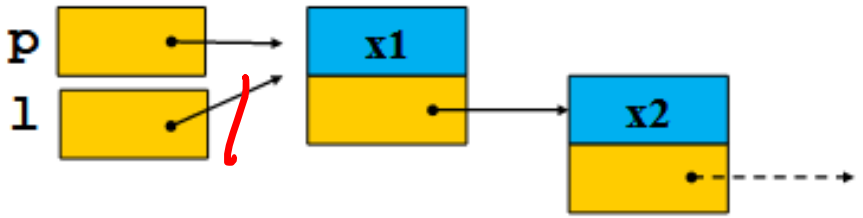
Pasaje por referencia (C++)

- C++ (no C) nos permite una forma alternativa de hacer el pasaje por referencia de otra forma
-
```
void proc(int &m){ m = 8; }

int main()
{
    int n = 5;
    proc(n);
    cout << n << endl;
}
```
- Esto imprime 8
- Vamos a estar utilizando esta notación, por comodidad, pero ambas son validas

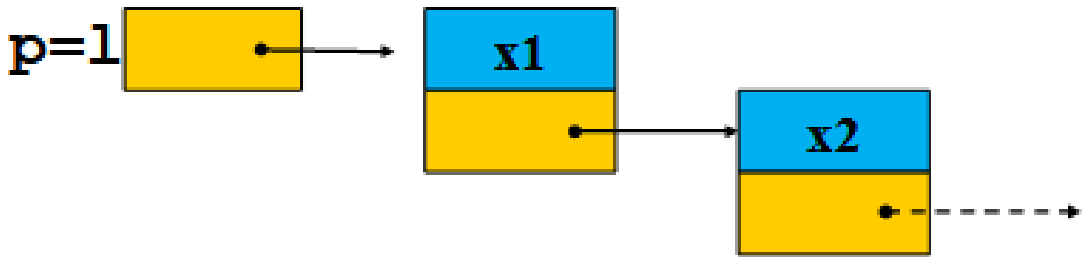
Ejemplos con listas

- Vamos a ver cuatro ejemplos distintos para ver que pasa con cada uno cuando le pasamos una lista no vacía p
- ```
  void p1 (Lista l) { l = NULL; }
  void p2 (Lista &l) { l = NULL; }
  void p3 (Lista &l) { l -> sig = NULL; }
  void p4 (Lista l) { l -> sig = NULL; }
  ```


  ```

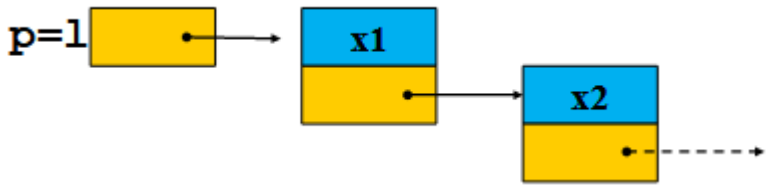
Ejemplos con listas

- Vamos a ver cuatro ejemplos distintos para ver que pasa con cada uno cuando le pasamos una lista no vacía p
- ```
  void p1 (Lista l) { l = NULL; }
  void p2 (Lista &l) { l = NULL; }
  void p3 (Lista &l) { l -> sig = NULL; }
  void p4 (Lista l) { l -> sig = NULL; }
  ```


  ```

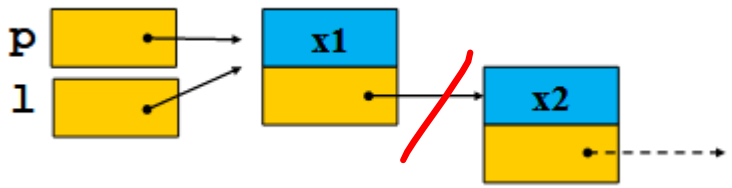
Ejemplos con listas

- Vamos a ver cuatro ejemplos distintos para ver que pasa con cada uno cuando le pasamos una lista no vacía p
- ```
  void p1 (Lista l) { l = NULL; }
  void p2 (Lista &l) { l = NULL; }
  void p3 (Lista &l) { l -> sig = NULL; }
  void p4 (Lista l) { l -> sig = NULL; }
  ```


  ```

Ejemplos con listas

- Vamos a ver cuatro ejemplos distintos para ver que pasa con cada uno cuando le pasamos una lista no vacía p
- ```
  void p1 (Lista l) { l = NULL; }
  void p2 (Lista &l) { l = NULL; }
  void p3 (Lista &l) { l -> sig = NULL; }
  void p4 (Lista l) { l -> sig = NULL; }
  ```


  ```


//Passing by value
void func(int data, double moreData) {
data *= data;
moreData++;
}
//Passing by reference
void func(int &data, double &moreData) {
data *= data;
moreData++;
}
//Passing the responsibility
void func(/*TODO: Code this*/) {
//do something
}

Dudas?

Volvemos a las listas

- Queremos ejecutar una operación proc sobre cada elemento de una lista
- Asumimos que p apunta al principio de una lista
- ```
  q = p
  while(q != NULL)
  {
      proc(q->info);
      q = q->sigg;
  }
  ```
  ```

Impresión de una lista

- Uno de los ejemplos típicos de recorrida de una lista es para imprimir los elementos de la lista.
- Donde la operación `proc` viene dada por una función `impDatos` que sabe imprimir los datos de tipo `T` de forma linda
- Para hacer esto vamos a recibir la lista "por valor" y vamos a ir recorriéndola hasta llegar a el final, imprimiendo el valor de cada uno de los elementos

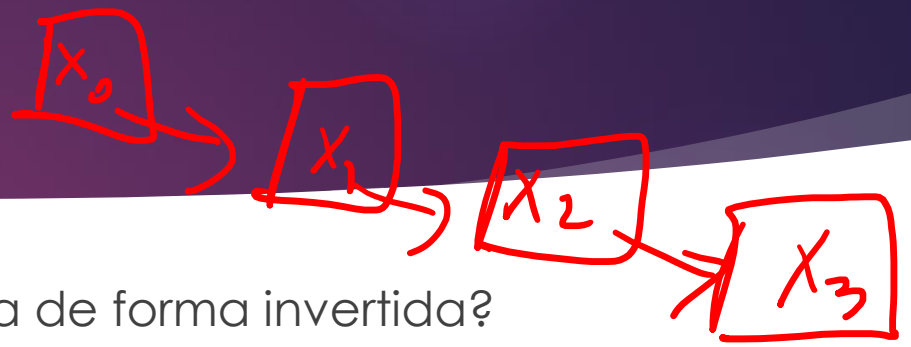
Impresión de una lista

- ```
  void impLista(Lista p)
  {
      while(p != NULL)
      {
          impDatos(p->info);
          p = p->sig; // por que podemos hacer esto?
      }
  }
  ```
- Podemos hacer la asignación p = p->sig, porque el puntero p es una copia del puntero original, por lo que solo se ve afectado en el scope de la función
  ```

Impresión de una lista

- Esta misma función podemos hacerla de forma recursiva
- ```
  void impLista(Lista p)
  {
      if(p != NULL)
      {
          impDatos(p->info);
          impLista(p->sig);
      }
  }
  ```
  ```

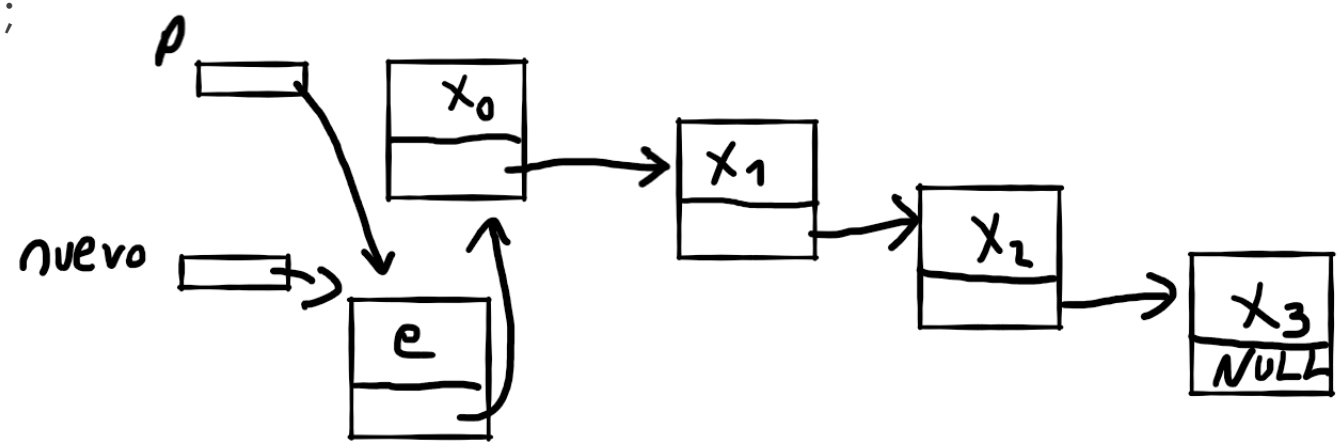
Impresión de una lista



- Se les ocurre como hacer para imprimir la lista de forma invertida?
-
```
void impLista(Lista p)
{
    if(p != NULL)
    {
        impLista(p->sig);// los dimos vuelta
        impDatos(p->info);// los dimos vuelta
    }
}
```

Inserción al principio de lista

- void insPrincipio(Lista &l, T e)
  {
  Lista nuevo = new nodoLista;
  nuevo->info = e;
  nuevo->sig = l;
  l = nuevo;
  }



Concatenar dos listas

- Otro procedimiento que vamos a usar bastante seguido es concatenar dos listas, o sea, agarrar el final de una de las listas y ponerla al final de la otra
- Vamos a recibir dos listas, y vamos a poner la segunda al final de la primera
- Usando la notación matemática de tipos inductivos, como escribiríamos estas funciones
- ```
  concat: Lista[a]xLista[a] ->Lista[a]
  concat([], l2)=...
  concat(x.l1, l2)=...
  ```
  ```

Concatenar dos listas

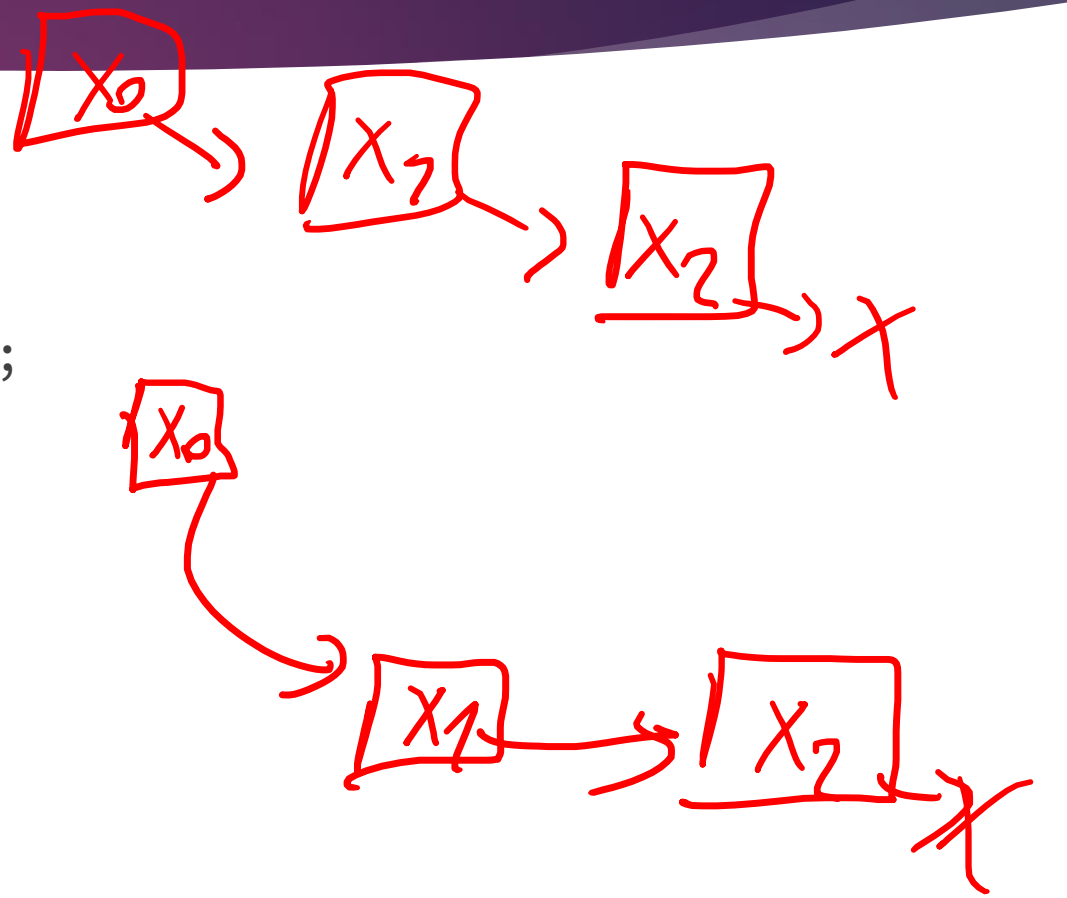
- Otro procedimiento que vamos a usar bastante seguido es concatenar dos listas, o sea, agarrar el final de una de las listas y ponerla al final de la otra
- Vamos a recibir dos listas, y vamos a poner la segunda al final de la primera
- Usando la notación matemática de tipos inductivos, como escribiríamos estas funciones
- ```
  concat: Lista[a]xLista[a] ->Lista[a]
  concat([], l2)= l2
  concat(x.l1, l2)= x.concat(l1,l2)
  ```
  ```

Concatenar dos listas

- Vamos a pasarlo entonces a C++
- ```
  void concat(Lista &l1, Lista l2)
  {
      if (l1 == NULL) l1 = l2; //l1 = copia(l2)
      else concat(l1->sig, l2);
  }
  ```
- `copia` es una función que recibe una lista y nos retorna una copia que no comparte memoria
  - Hay casos donde hay que tener cuidado de que no compartan memoria, por eso nos puede interesar hacer una copia
  ```

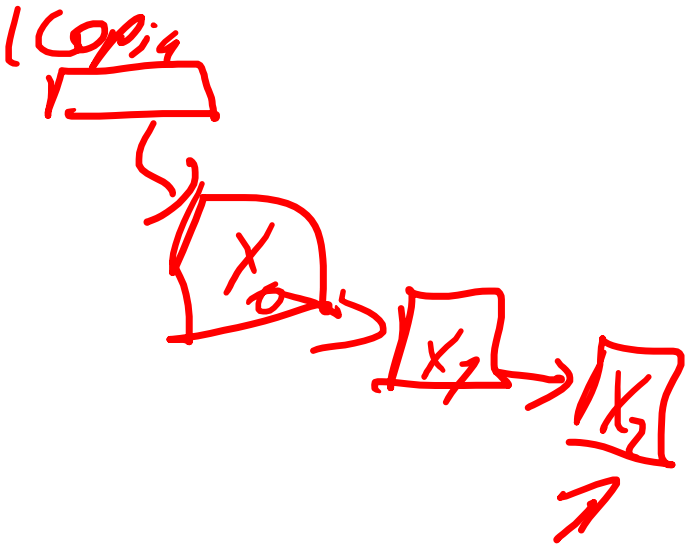
Copiar lista (recursivo)

```
Lista copia (Lista l){
    if (l == NULL) return NULL;
    else{
        Lista res = new nodoLista;
        // no comparte memoria
        res->info = l->info;
        res->sig = copia(l->sig);
        return res;
    }
}
```



Copiar lista (iterativo)

```
Lista copia (Lista l)
{
    Lista lCopia, nuevo;
    if (l==NULL) res = NULL;
    else
    {
        Lista ultimo = NULL; // inserciones al final de res
        while (l!=NULL)
        {
            nuevo = new nodoLista;
            nuevo->info = l->info;
            nuevo->sig = NULL;
            if (ultimo==NULL){lCopia = ultimo = nuevo; }
            else
            {
                ultimo->sig = nuevo;
                ultimo = nuevo;
            }
            l = l->sig;
        }
    }
return lCopia;
}
```



Inserción al final de una lista

- Vamos a simplificar un poco la copia, vamos a refactoriarla, haciendo lo que se llama extraer una función
-
```
Lista copia (Lista l)
{
    Lista lCopia, nuevo;
    if (l==NULL) lCopia = NULL;
    else
    {
        Lista ultimo = NULL; // inserciones al final de lCopia
        while (l!=NULL)
        {
            nuevo = new nodoLista;
            nuevo->info = l->info;
            nuevo->sig = NULL;
            if (ultimo==NULL){lCopia = ultimo = nuevo; }
            else
            {
                ultimo->sig = nuevo;
                ultimo = nuevo;
            }
            l = l->sig;
        }
    }
return lCopia;
}
```

Inserción al final de una lista

```
nuevo = new nodoLista;
nuevo->info = l->info;
nuevo->sig = NULL;
if (ultimo==NULL)
{
    lCopia = ultimo = nuevo;
}
else
{
    ultimo->sig = nuevo;
    ultimo = nuevo;
}
```

Inserción al final de una lista

```
void insFinal(Lista &lCopia, Lista &ultimo, T e)
{
    Lista nuevo = new nodoLista;
    nuevo->info = e;
    nuevo->sig = NULL;
    if (ultimo==NULL)
    {
        lCopia = ultimo = nuevo;
    }
    else
    {
        ultimo->sig = nuevo;
        ultimo = nuevo;
    }
}
```

- Inserta el elemento T al final de la lista, y nos devuelve el puntero al último nodo

Copia lista (iterativo v2)

```
Lista copia (Lista l)
{
    Lista lCopia, nuevo;
    if (l==NULL) lCopia = NULL;
    else
    {
        Lista ultimo = NULL; // inserciones al final de lCopia
        while (l!=NULL)
        {
            insFinal(lCopia, ultimo, l->info);
            l = l->sig;
        }
    }
return lCopia;
}
```

Concatenar dos listas (iterativo)

```
void concat(Lista &l1, Lista l2)
{
    if(l1 == NULL)
        l1 = copia(l2);
    else
    {
        Lista iter = l1;

        //buscamos el puntero al ultimo nodo
        while(iter->sig != NULL)
        {
            iter = iter->sig;
        }

        //ponemos la copia al final
        iter->sig = copia(l2);
    }
}
```

Concatenar dos listas (iterativo v2)

- Si queremos que devolver una nueva lista independiente de la primer lista
-
```
Lista concat(Lista l1, Lista l2)
{
    if(l1 == NULL) return copia(l2);
    else
    {
        Lista res = new nodoLista;
        res->info = l1->info;
        res->sig = concat(l1->sig, l2);
        return res;
    }
}
```

Invertir una lista

- Otro clásico, invertir una lista
- Vamos a retornar una nueva lista, con los mismos elementos, pero en orden inverso
- Es muy parecido a la recorrida
- ```
  Lista invertir(Lista l)
  {
      Lista aux;
      Lista inv = NULL;
      while(l!=NULL)
      {
          insPrincipio(inv, l->info);
          l = l->sig;
      }
      return inv;
  }
  ```
  ```

Invertir una lista (v2)

- Cuando hacemos `insPrincipio`, estamos creando un nuevo nodo, si la lista es suficientemente grande, puede llegar a tener un impacto visible
  - `malloc` es una operación que puede ser muy lenta
- Una solución a este problema es reutilizar los mismos punteros de la lista original
- ```
  Lista invertir(Lista l)
  {
      Lista aux;
      Lista inv = NULL;
      while(l != NULL)
      {
          aux = l->sig;
          l->sig=inv;
          inv = l;
          l = aux;
      }
  }
  ```
  ```

Inserción ordenada

- Pre: la lista está ordenada
- Pos: se agrega el elemento a la lista, y la lista sigue siendo ordenada
- Veamos primero en la notación matemática
- ```
  insOrd: a x Lista[a] -> Lista[a]
  insOrd(e, []) = ...
  insOrd(e, x.l) = ..., Si e <= x
  insOrd(e, x.l) = ..., en otro caso
  ```
  ```

Inserción ordenada

- Pre: la lista está ordenada
- Pos: se agrega el elemento a la lista, y la lista sigue siendo ordenada
- Veamos primero en la notación matemática
- ```
  insOrd: a x Lista[a] -> Lista[a]
  insOrd(e, []) = e.[]
  insOrd(e, x.l) = e.x.l, Si e <= x
  insOrd(e, x.l) = x.insOrd(e,l), en otro caso
  ```
  ```

Inserción Ordenada

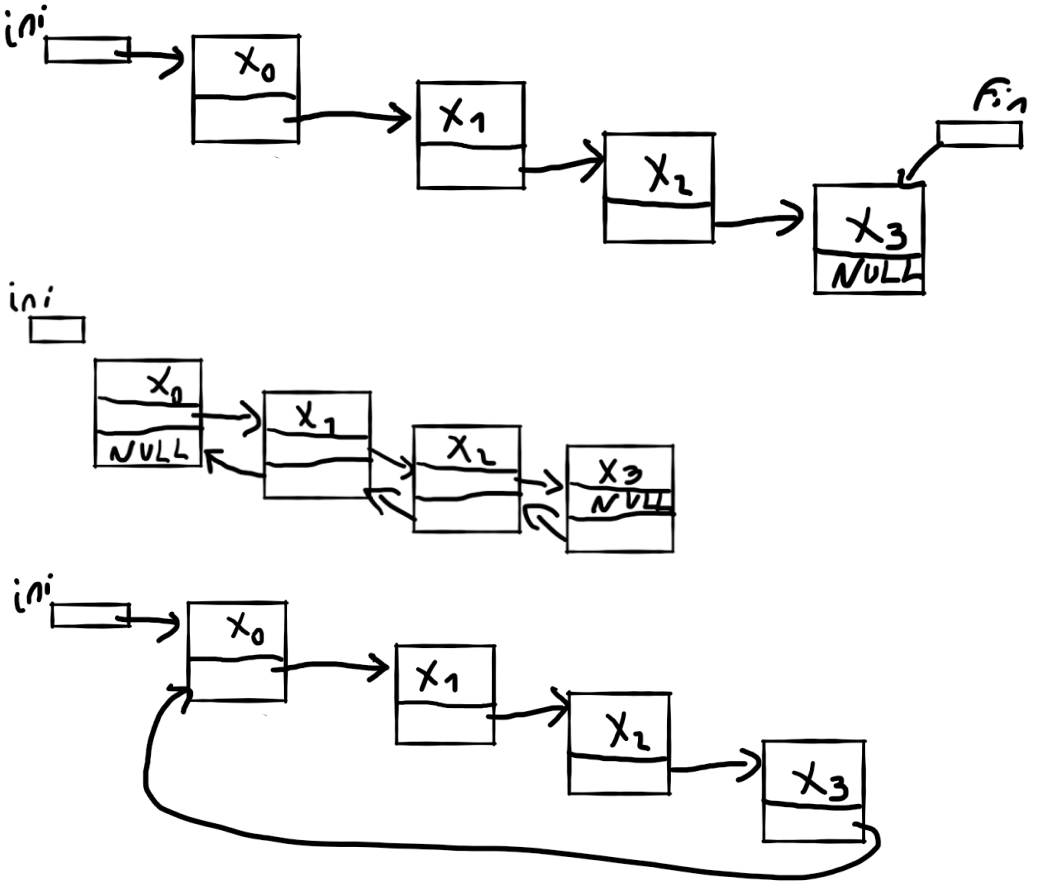
```
void insOrd(Lista &l, T e)
{
    if(l ==NULL) insPrincipio(l, e);
    else
    {
        if(e <= l->info) insPrincipio(l, e);
        else(insOrd(l->sig, e)
    }
}
```

Insert Sort

- ```
  insertSort: Lista[a]->Lista[a]
  insertSort([])=[]
  insertSort(e.l)=insOrd(e,insertSort(l))
  ```
- ```
  //iterativo
  Lista insertSort(Lista l)
  {
      Lista ord = NULL;
      while (l != NULL)
      {
          insOrd(ord, l->info);
          l = l->sig;
      }
  }
  ```

Variantes de listas

- Lista encadenada con puntero al final
- Lista doblemente encadenada
- Lista circular




AND THAT'S HOW I INVENTED
THE LINKED LIST