

Listas simplemente enlazadas

Informática II
R2004 - 2020

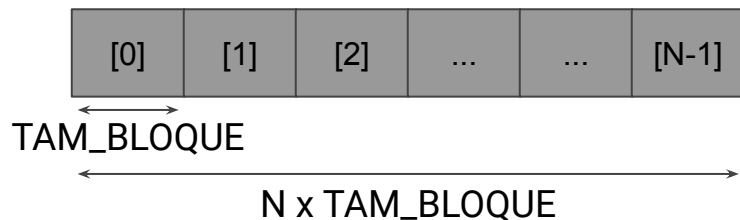
Vectores COMPACTOS

Los vectores son espacios de memoria pensados para almacenar un número determinado de datos del mismo tipo.

Se caracterizan por estar formados por bloques (posiciones) consecutivos en memoria.

Para hacer más eficiente el manejo de la memoria, estos vectores pueden ser dinámicos, creciendo o decreciendo en función de lo que se necesite.

Sin embargo, tanto los vectores dinámicos como los estáticos (de una longitud fija) ocupan un bloque consecutivo en la memoria RAM del sistema, que será tanto más grande dependiendo del tamaño de los bloques y de la cantidad de los mismos.




Se necesita un espacio de
($N \times \text{TAM_BLOQUE}$) bytes en memoria
CONSECUTIVOS para almacenar el
vector

Vectores compactos... ventajas y desventajas

Una clara ventaja de almacenar datos en vectores es la facilidad de acceso a todos los elementos del mismo. Conociendo únicamente la dirección de inicio y la cantidad de bloques que posee el vector, puedo acceder fácilmente a cualquiera de estos bloques mediante el operador **corchetes** - `[]` -, o haciendo la suma de punteros y utilizando el operador **asterisco** - `*(inicio + i)` -

Una desventaja de esta forma de almacenar datos es que el sistema debe tener disponible un gran bloque de memoria a partir de la dirección de inicio del puntero, de manera de poder guardar en forma consecutiva todos los elementos del vector. Si se tratase de un vector dinámico y decido agrandarlo, es posible que tenga que reubicar todos los bloques de datos para almacenar un nuevo sector de memoria, y esto demora tiempo.




Alternativa... listas de datos

Una lista es conceptualmente lo mismo que un vector (una concatenación de N bloques de datos del mismo tipo), con la única diferencia de que estos datos ya no se encuentran uno consecutivo al otro en memoria, sino que se encuentran dispersos por la misma. De esta manera puedo hacer un aprovechamiento más eficiente de la memoria, disponiendo de pequeños bloques libres.

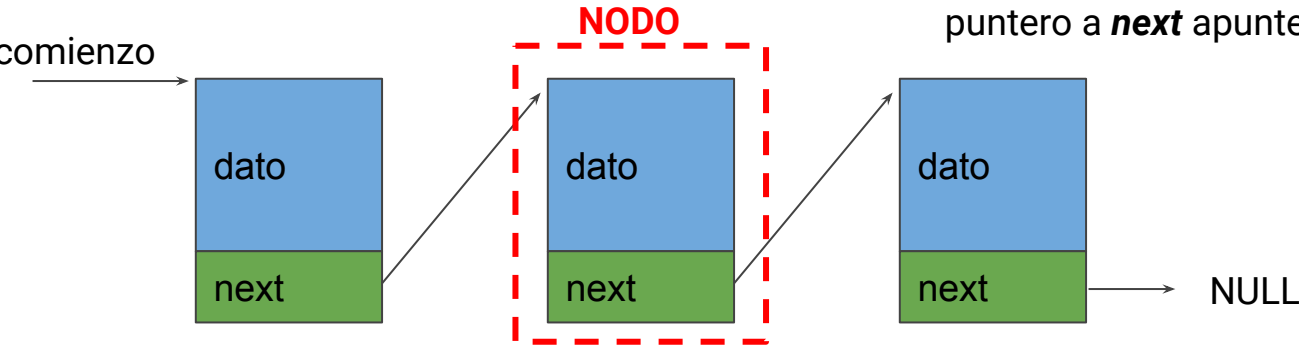
Como contrapartida, tengo que poder almacenar en algún lado la dirección de cada uno de estos bloques de memoria, de manera de no perderlos.

Teniendo en cuenta que esta metodología es de por sí dinámica, la lista de datos puede crecer o decrecer en forma indeterminada, limitada únicamente por la cantidad de espacio disponible en la memoria del sistema.



¿Cómo hago una lista?

```
struct NODO {  
    tipo_dato dato;  
    struct NODO *next;  
};
```



Cada **nodo**, o bloque de datos de mi lista, contendrá el dato que quiero almacenar (puede ser un entero, un float, un char, u otra estructura u objeto), y deberá estar seguido por la **dirección** en donde se almacena el dato siguiente, que es un puntero a otro bloque de datos del mismo tipo (**struct NODO ***).

De esta manera, el comienzo del bloque de datos, que normalmente lo señalábamos con el nombre del vector, estará señalado por un puntero a una estructura de tipo NODO (**struct NODO * comienzo**), y podré saber que estoy apuntando al último nodo de una lista cuando su puntero a **next** apunte a **NULL** (o **nullptr**, en C++).

Ejemplo: Una lista para almacenar objetos de tipo Persona

```
using namespace std;

class Persona
{
    string Nombre;
    string Apellido;
    long dni;

private:
    Persona();
    Persona(const string &,const string &,long);
    Persona(const Persona &);

    //...
};
```

DATO QUE QUIERO ALMACENAR

```
class Lista
{
    struct NODO{
        Persona dato;
        NODO * next;
    };

    NODO * comienzo;
    int tam;
```

Para armar la **lista** armo en primer lugar un **NODO**, que serán los bloques básicos de los que se compone la lista.

Cada NODO contendrá la dirección de la posición siguiente del vector (**NODO *next**)

También debo contar con una **dirección de inicio** de la lista (similar a la dirección de inicio de un vector).

Se instancia el objeto *Lista*: Comienza vacía

```
int main ( void )  
{  
    Lista L1;  
    //...  
}
```

```
Lista::Lista ( void )  
{  
    comienzo = nullptr;  
    tam = 0;  
}
```

Cuando tengo una lista vacía, solo tengo un puntero que no apunta a ningún lado (***nullptr***, para que no esté descontrolado). Sería lo mismo que un vector dinámico que todavía no guardó ningún elemento.

comienzo
→ NULL

Esto puede señalizarse con su puntero de inicio apuntando a ***nullptr***, o bien con una variable tamaño que valga 0. En este caso, y para hacer el ejemplo más sencillo, usaremos AMBOS métodos, ***aunque son redundantes***.

Se van agregando elementos...

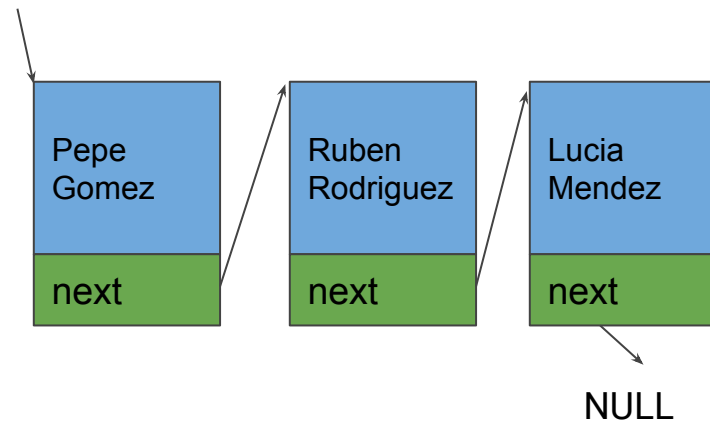
Ya sea por interacción del usuario, o por cualquier método de entrada de datos iré incorporando información a mi lista de elementos...

```
int main ( void )
{
    Lista L1;

    //Creo datos:
    Persona P1 ( "Pepe", "Gomez", 31896345);
    Persona P2 ( "Ruben", "Rodriguez", 34223445);
    Persona P3 ( "Lucia", "Mendez", 342224422);

    //Los pongo en la lista:
    L1.AgregarNodo(P1);
    L1.AgregarNodo(P2);
    L1.AgregarNodo(P3);
}
```

comienzo



VISTO DESDE EL MAIN

Se van agregando elementos...

```
Lista & Lista::AgregarNodo ( const Persona &P )
{
    //Creo un nuevo nodo con el dato que recibo:
    NODO * nuevo = new NODO;
    nuevo->next = nullptr; //Lo voy a poner al final, el siguiente sera NULL
    nuevo->dato = P; //Debe estar sobrecargado el operador =

    //Si la lista estaba vacia, pongo este nuevo nodo al principio...
    if ( tam == 0 )
        comienzo = nuevo;

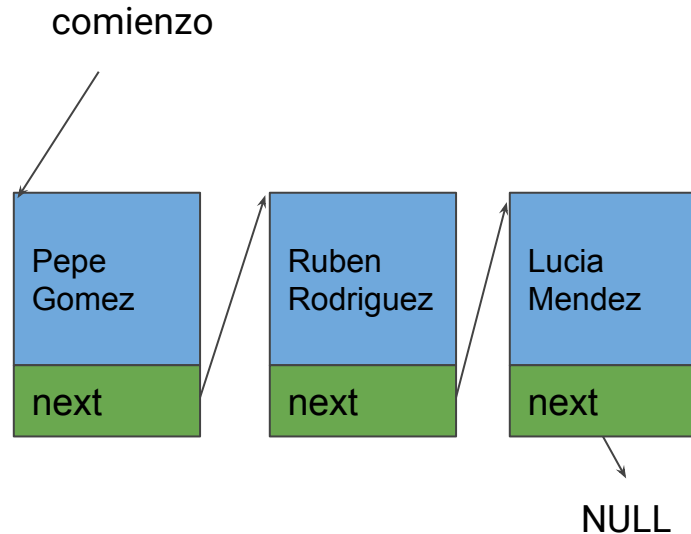
    //Sino, debo recorrer la lista hasta llegar al ultimo nodo, y ponerlo ahi:
    else
    {
        //obtengo la dirección de comienzo de la lista:
        NODO *aux = comienzo;

        //recorro la lista hasta llegar al ultimo nodo:
        for (int i = 0 ; i< tam; i++)
            aux = aux->next;

        //En aux tengo cargada la direccion del ultimo nodo, pongo el nuevo en el NEXT:
        aux->next = nuevo;
    }

    //En cualquier caso, incremento el tamaño de la lista:
    tam++;

    //Por las dudas, devuelvo la lista modificada:
    return *this;
}
```



VISTO DESDE LA CLASE

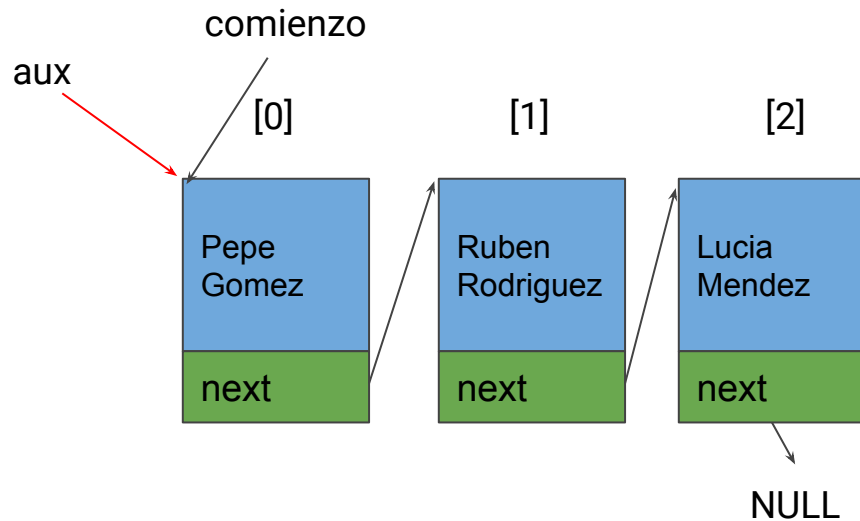
¿Cómo recorro la lista?

```
int main ( void )
{
    Lista L1;

    //Creo datos:
    Persona P1 ("Pepe","Gomez",31896345);
    Persona P2 ("Ruben","Rodriguez",34223445);
    Persona P3 ("Lucia","Mendez",342224422);

    //Los pongo en la lista:
    L1.AgregarNodo(P1);
    L1.AgregarNodo(P2);
    L1.AgregarNodo(P3);

    int posicion = L1.BuscarNodo(P1);
    cout << "La persona " << posicion << " de la lista es " << L1[posicion] << endl;
}
```



2 métodos que recorren la lista, ya sea buscando un elemento en particular, o accediendo al elemento n-ésimo de la lista...
Veamos alguno de ellos desarrollado:

¿Cómo recorro la lista?

```
Persona & Lista::operator [] ( int posicion )
```

```
{
```

```
    //Si la posicion que recibí es mayor al tamaño de la lista
```

```
    static Persona err("N","N",0);
```

```
    Persona &retorno = err;
```

```
    if ( posicion < tam)
```

```
    {
```

```
        NODO * aux = comienzo;
```

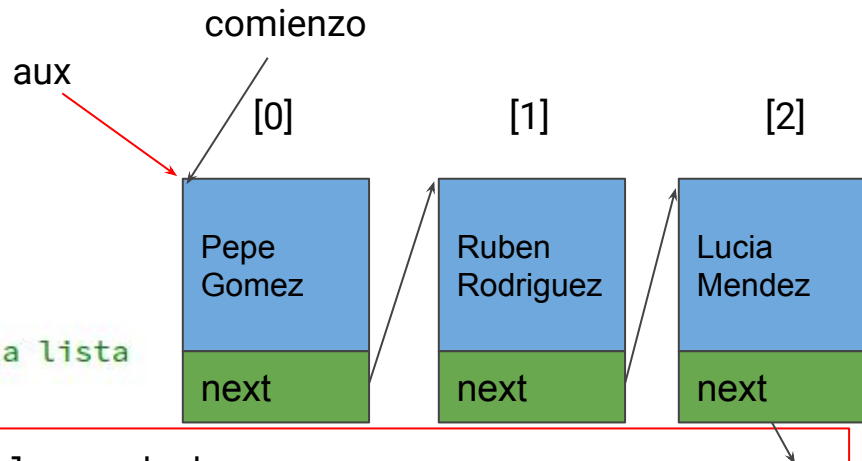
```
        for (int i = 0 ; i < posicion ; i++)
```

```
            aux = aux->next;
```

```
        retorno = aux->dato;
```

```
    }
```

```
    return retorno;|
```



En lugar de hacer:

```
retorno = vector[posicion];
```

Ahora hago:

```
for ( i = 0 ; i < posicion ; i++ )
```

```
    aux=aux->next;
```

```
retorno = aux->dato;
```

¿Cómo recorro la lista?

```
int Lista::BuscarNodo(const Persona &P)
```

```
{  
    int retorno = -1;
```

```
    //Si la lista está vacia, no tengo nada que hacer
```

```
    if ( tam != 0 )
```

```
{
```

```
    //utilizaré una variable auxiliar para recorrer la lista:
```

```
    NODO *aux = comienzo;
```

```
    //Recorro las TAM posiciones buscando si el SIGUIENTE es el nodo a eliminar:
```

```
    for (int retorno = 0 ; retorno < tam ; retorno ++ , aux = aux->next)
```

```
{
```

```
        if ( aux->dato == P ) //Tengo que tener sobrecargado el operador ==  
            break;
```

```
}
```

```
    //Si no lo encuentre, devuelvo -1:
```

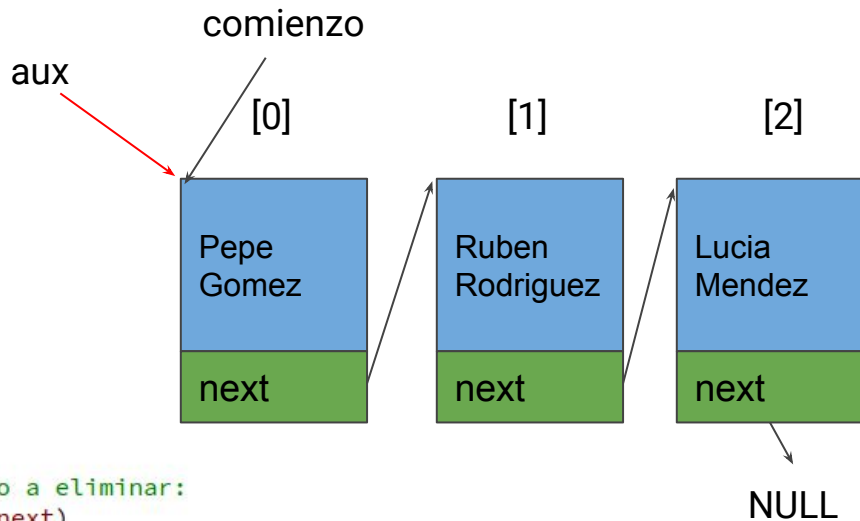
```
    if ( retorno == tam )
```

```
        retorno = -1;
```

```
}
```

```
    return retorno;
```

```
}
```



En lugar de hacer:

```
for ( i = 0 ; i < tam ; i++)  
    if ( vector[i] == P )
```

Ahora hago:

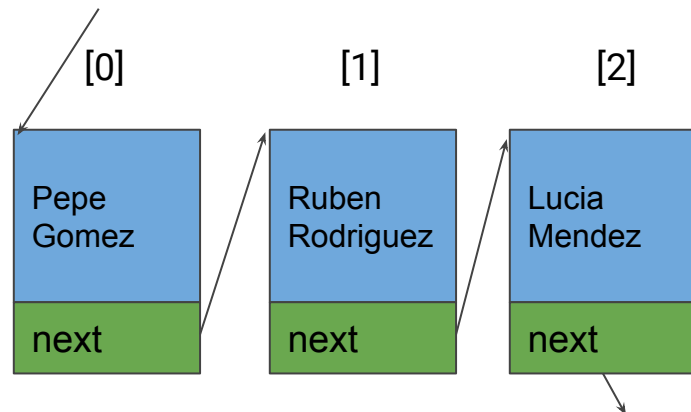
```
for ( i = 0 ; i < tam ; i++, aux=aux->next )  
    if ( aux->dato == P )
```

¿Cómo elimino un nodo?

```
Lista & Lista::EliminarNodo ( const Persona & P )
{
    //Busco el nodo que quiero eliminar:
    int nro_nodo = BuscarNodo(P);

    if ( nro_nodo != -1 ) //Si existe el nodo que quiero borrar:
    {
        NODO *aux = comienzo;
        //Si el nodo es el primero:
        if ( nro_nodo == 0 )
        {
            //modifico el puntero a comienzo de la lista:
            comienzo = aux->next;
            //borro el elemento:
            delete aux;
        }
        else
        {
            //Genero un puntero auxiliar:
            NODO *borrar;
            //Recorro la lista hasta encontrar el elemento anterior:
            for ( int i = 0 ; i < nro_nodo - 1 ; i++ , aux = aux->next )
            {
                //Me quedo con la direccion del nodo a borrar:
                borrar = aux->next;
                //Cambio la direccion del nodo anterior:
                aux->next = borrar->next;
                //Y borro el nodo "borrar"
                delete borrar;
            }
        }
        //Decremento la cantidad de nodos de la lista:
        tam--;
    }
}
```

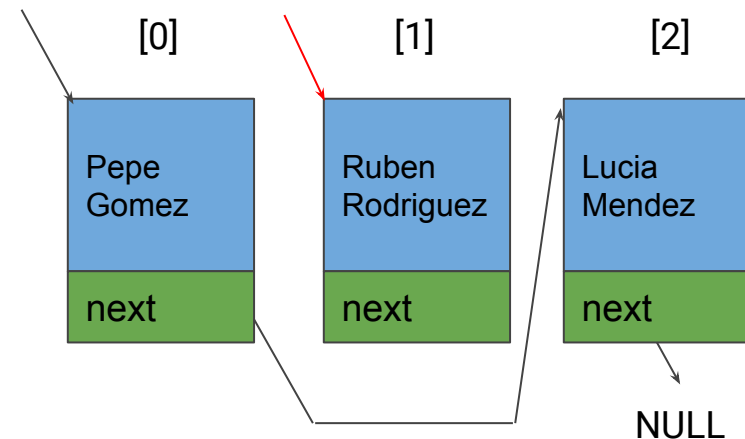
comienzo



aux

borrar

NULL



Ejercicio: completar la clase...

```
class Lista
{
    struct NODO{
        Persona dato;
        NODO * next;
    };
    NODO * comienzo;
    int tam;

public:
    //Constructor por defecto:
    Lista();
    //Destructor de la clase:
    ~Lista();
    //Agrega una Persona a la lista
    Lista & AgregarNodo(const Persona &);
    //Elimina una Persona de la lista
    Lista & EliminarNodo(const Persona &);

    //Igual que AgregaPersona (pero con un operador)
    Lista & operator +=(const Persona &);
    //Igual que EliminaPersona (pero con un operador)
    Lista & operator -=(const Persona &);
    //Concatena dos listas:
    Lista & operator +=(const Lista &);
    //Obtiene el indice de un elemento de la lista:
    int BuscarNodo(const Persona &);
    //Obtiene el tamaño de la lista:
    int Tam(void);
    //Obtiene el elemento de una posición de la lista:
    Persona & operator[] (int);
    //Imprime la lista por pantalla mediante cout
    friend ostream & operator << (ostream &, const Lista &);
};
```

De este listado nos faltarían:

~Lista() //Recorre la lista y va borrando uno por uno los nodos
L1 += L2 //Recorre la lista L2 y va agregando uno a uno los nodos a L1
cout << L1 //Recorre la lista L1 y va imprimiendo por pantalla los datos
//de los nodos de a uno (debería estar sobrecargado el
//método << en el objeto Persona)

Otros problemas (no los resolveremos aqui):

- ¿Cómo hago si quiero agregar elementos en cualquier lugar de la lista, y no solo al final?

REALIZAR MÉTODO:

Lista & Lista::AgregarElemento (const Persona & , int pos);

- ¿Cómo podría hacer para recorrer la lista tanto para un lado (aux = aux -> next), como para el otro?

VER LISTAS DOBLEMENTE ENLAZADAS

- ¿Podría hacer un método para ordenar la lista?
- ¿Se podría hacer un template, de manera de usar la lista para cualquier tipo de dato? ¿Qué particularidades debería tener el dato?;