Estructuras de datos dinámicas IV

1ºDAM IoT

Estructuras de datos dinámicas con registros

Componiendo múltiples registros (struct) del mismo tipo (con la misma estructura de datos), enlazados unos con otros y en tiempo de ejecución podemos crear:

- Listas
  - Listas simplemente enlazadas
  - Listas doblemente enlazadas
- Colas
- Pilas

Estas estructuras de datos dinámicas son colecciones de tipos de datos struct que dependiendo de su implementación y forma de insertar y borrar serán listas, colas o pilas.

Estructuras de datos dinámicas con registros

Tendremos algo como esto. Una colección de elementos de tipo struct. Esta colección asignará de forma dinámica espacio para un struct según lo vaya necesitando. Y liberará espacio cuando borremos el struct.

Los elementos de la colección no tienen que estar de forma consecutiva en la memoria.

Listas

La implementación completa de una lista no es sólo definir la estructura de los registros que representan a la colección y a cada ítem.

También hay que definir las funciones que se encargarán de realizar estas labores de mantenimiento:

- Inicializar la lista.
- Añadir un ítem nuevo (al final o como inserción entre dos ya existentes).
- Recorrer los ítems (acceder a todos ellos de forma secuencial).
- Acceder a un ítem dado, para consultar o modificar sus valores.
- Eliminar un ítem dado, manteniendo la coherencia de los enlaces entre sus vecinos.

Listas

Cada elemento podemos definirlo como una estructura.

Su contenido nos da igual, es lo de menos.

```
struct miItem
{
     tipo_campo1 campo1;
     tipo_campo1 campo1;
     ...
     tipo_campoN campoN;
};
```

Tal como hemos visto antes, las posiciones de la lista no se encuentran de forma consecutiva en la memoria como un array.

Listas

Lo primero que podemos observar es que necesitamos poder identificar cuál es el primer elemento de la lista.

Para ello utilizaremos otra estructura que enlace con el primer elemento de la colección.

```
struct miLista
{
     struct miItem *primerItem;
};
```

La estructura miLista es la que nos dice cómo es la colección.

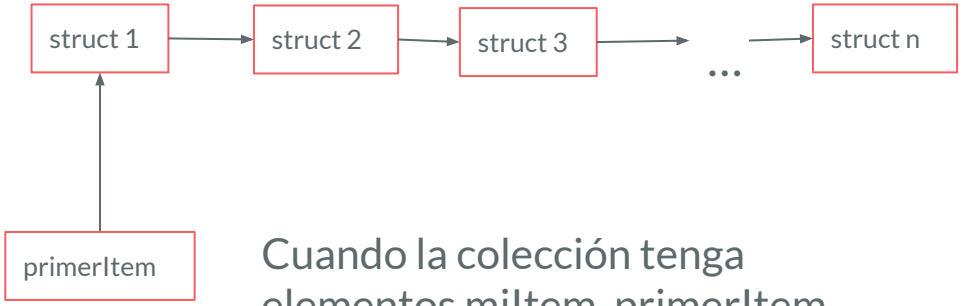
La estructura miItem es el contenido de cada uno los elementos de la colección.

Listas: Apunta al primer elemento

Cuando inicializamos la colección y no tenemos ningún elemento, al puntero que apunta al primer elemento debemos asignarle el valor de NULL.

La forma que tendremos de saber si una colección está vacía es comprobando si este puntero es igual NULL.

```c
#include <stdio.h>
#include <stdlib.h>
struct miItem //Nuestra estructura guarda unas coordenadas
{
    float x;
    float y;
};
struct miLista
{
    struct miItem *primerItem; //Apunta al primer elemento de la lista
};
int main(void)
{
    struct miLista lista; //Creo una estructura miLista
    lista.primerItem = NULL; //El primer elemento es NULL, está vacía

}
```

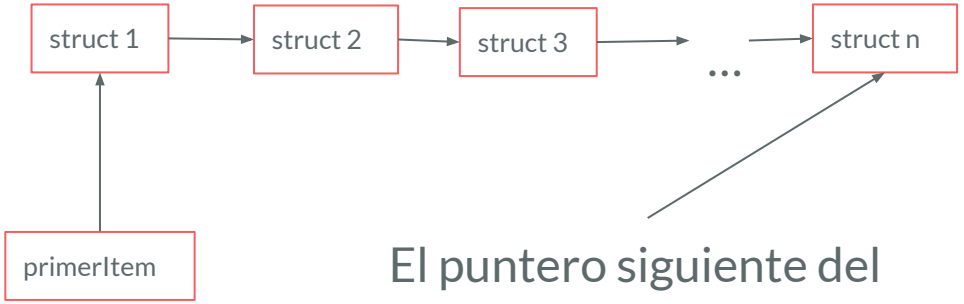


Cuando la colección tenga elementos miItem, primerItem apuntará al primer elemento, ¿pero cómo podremos recorrer el resto de elementos?

Listas

Necesitamos modificar la estructura miItem para que tenga un puntero a la siguiente estructura miItem.

```c
#include <stdio.h>
#include <stdlib.h>
struct miItem //Nuestra estructura guarda unas coordenadas
{
    float x;
    float y;
    struct miItem *siguiente; //Apunta al siguiente elemento miItem
    //Si es el último apuntará a NULL
};
struct miLista
{
    struct miItem *primerItem; //Apunta al primer elemento de la lista
};
int main(void)
{
    struct miLista lista; //Creo una estructura miLista
    lista.primerItem = NULL; //El primer elemento es NULL, está vacía

}
```

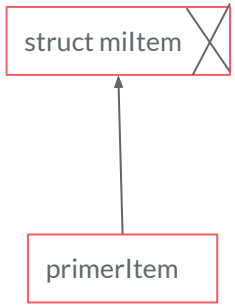


El puntero siguiente del último item apunta a NULL.

Listas: Crear el primer elemento

1. Reservamos espacio para el item nuevo.
2. Asignamos valores a los campos. El puntero de siguiente será NULL.
3. Compruebo si la lista está vacía. El primer elemento es NULL.
4. Actualizamos el valor del primer elemento de la lista.

```
struct miItem *itemNuevo; //Variable auxiliar para añadir item
itemNuevo = (struct miItem*)malloc(sizeof(struct miItem));
itemNuevo->x = 10.5;
itemNuevo->y = 12.9;
itemNuevo->siguiente = NULL; //No hay siguiente

if(lista.primerItem == NULL){ //Soy el primero
    //Indicamos en la lista que el primer elemento es el que acabamos de crear
    lista.primerItem = itemNuevo;
}
```



Listas

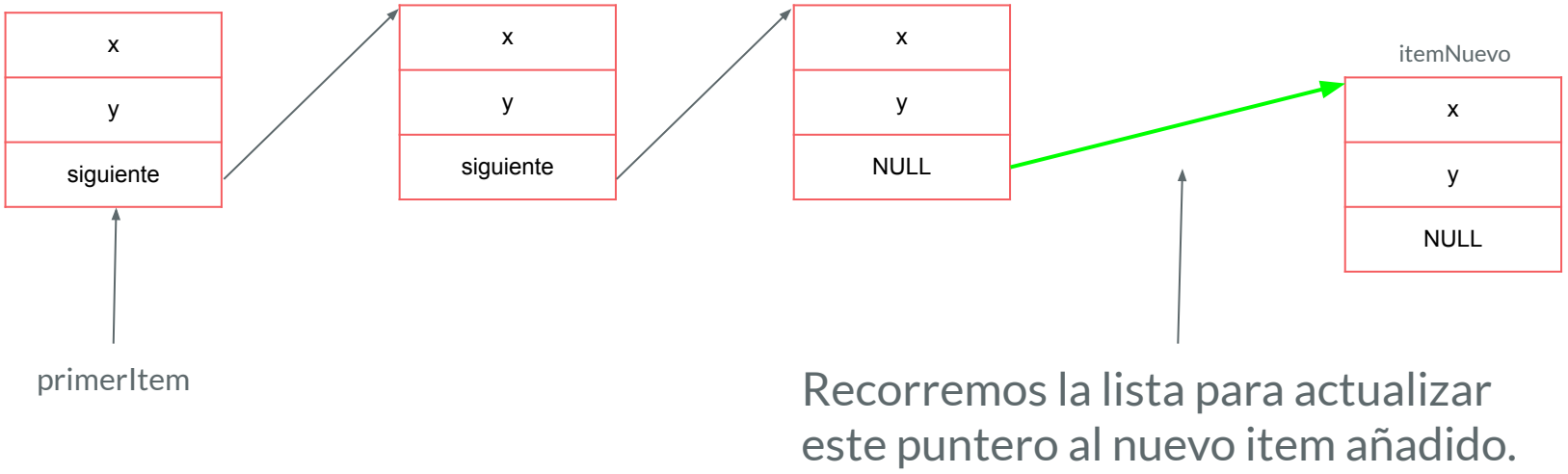
¿Y si queremos añadir más de un elemento?

Lo más lógico es que tengamos un programa con un menú con opciones de insertar, borrar, imprimir la lista. Y que el usuario desee insertar más de un elemento en la lista. En esos casos iremos reservando memoria para el nuevo item y comprobando si la lista está vacía o no.

Ya sabemos cómo insertar el primer elemento. Como estamos con una lista simple, los siguientes elementos los insertamos al final. Podríamos insertarlos también de forma ordenada según algún criterio (pero es un poco más complicado).

Listas: Insertar elementos

1. Reservamos espacio para el item nuevo.
2. Asignamos valores a los campos. El puntero de siguiente será NULL.
3. Si la lista no está vacía recorro la lista para llegar al último elemento.
4. Actualizo el puntero siguiente del último item de la lista al nuevo item creado.



Listas: Insertar elementos

Reservo espacio para el nuevo item.
Inicializo campos.
Siguiente a NULL.

Compruebo si la lista está vacía. Insertamos el primer elemento.

Si no está vacía recorro todos para llegar al último.
Utilizo un puntero auxiliar.

```
struct miItem *itemNuevo; //Variable auxiliar para añadir item
itemNuevo = (struct miItem*)malloc(sizeof(struct miItem));
itemNuevo->x = 10.5;
itemNuevo->y = 12.9;
itemNuevo->siguiente = NULL; //No hay siguiente

if(lista.primerItem == NULL){ //Soy el primero
    //Indicamos en la lista que el primer elemento es el que acabamos de crear
    lista.primerItem = itemNuevo;
}else{ //Tiene más de un elemento
    struct miItem *itemAux; //Puntero para recorrer la lista
    itemAux = lista.primerItem; //Apunta al primer elemento
    while (itemAux->siguiente != NULL) //Mientras que no sea el último
    {
        itemAux = itemAux->siguiente; //Avanzo al siguiente item
    }
    //Una vez he llegado al final, indico que el siguiente es el nuevo item
    itemAux->siguiente = itemNuevo;
}
```

Listas: Recorrer una lista

Observa que con el ejemplo anterior ya hemos aprendido a recorrer una lista para poder hacer algún cálculo con los campos de cada elemento o simplemente imprimirlos por pantalla.

Esta vez para recorrer comprobamos si el item con el que recorremos es NULL.

```
//Imprimir lista
if(lista.primerItem == NULL){ //Lista vacía
    printf("La lista está vacía");
}else{ //Tiene más de un elemento
    struct miItem *itemAux; //Puntero para recorrer la lista
    itemAux = lista.primerItem; //Apunta al primer elemento
    while (itemAux != NULL) //Mientras que no sea el último
    {
        printf("%f  %f\n", itemAux->x, itemAux->y);
        itemAux = itemAux->siguiente; //Avanzo al siguiente item
    }
}
```

Lista

Ejercicio: Crea un menú con las opciones de insertar e imprimir en la lista. Sólo tienes que copiar y pegar lo visto hasta ahora y adaptarlo a la creación del menú. Haz pruebas de insertar elementos e imprimirlos.

Lista

Ejercicio: Crea un menú con las opciones de insertar e imprimir en la lista. Sólo tienes que copiar y pegar lo visto hasta ahora y adaptarlo a la creación del menú. Haz pruebas de insertar elementos e imprimirlos.

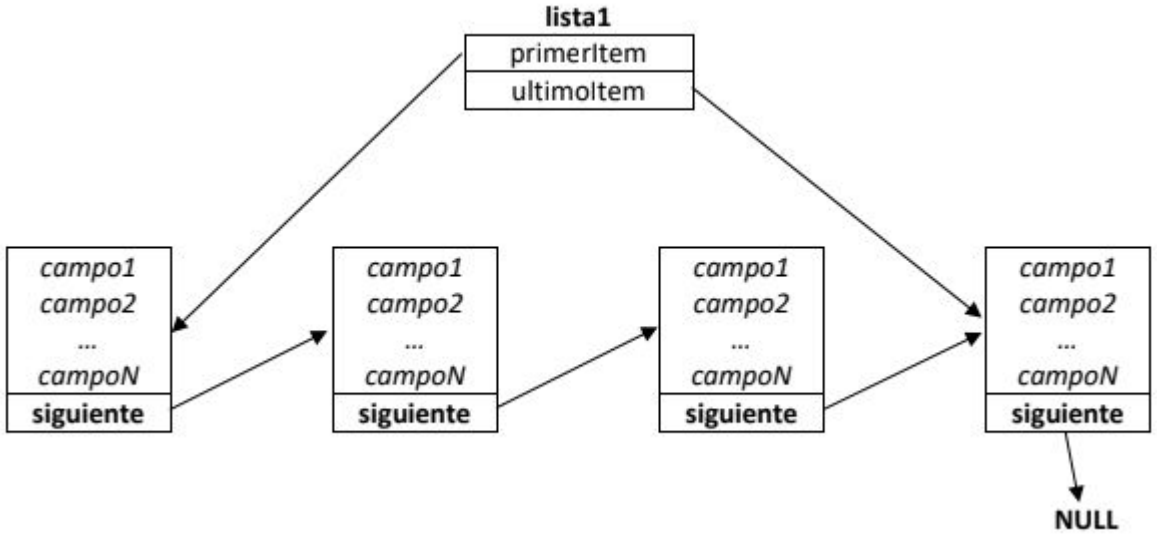
La solución la tienes en el aula virtual. El fichero es listasimple.c.

Listas

Observa que cada vez que queremos insertar un elemento nuevo debemos recorrer toda la lista. Cuando tengamos muchos elementos, por ejemplo, 1000, tendremos que hacer recorridos más largos. ¿Se te ocurre alguna forma de poder solucionar esto utilizando punteros?

Listas: Apunta al primero y último

Una solución sería que guardaramos también el puntero al último item en la estructura dónde guardamos el primer item.



Listas

Ahora la estructura miLista será algo así:

```
struct miLista
{
    struct miItem *primerItem; // Apunta al primer elemento de la lista
    struct miItem *ultimoItem; //Apunta al último elemento de la lista
};
```

Cuando inicialicemos la lista debemos asignar NULL al último elemento también.

Al insertar tendremos que modificar quién es el último elemento de la lista. No será necesario recorrer todos los elementos de la lista. Podemos ir directamente al último.

La forma de recorrer la lista no cambia.

Listas

Inicialización de la lista.

```
struct miLista lista;    // Creo una estructura miLista
lista.primerItem = NULL; // El primer elemento es NULL, está vacía
lista.ultimoItem = NULL; // El último elementoe es NULL, está vacía
```

Listas

Inserción en la lista

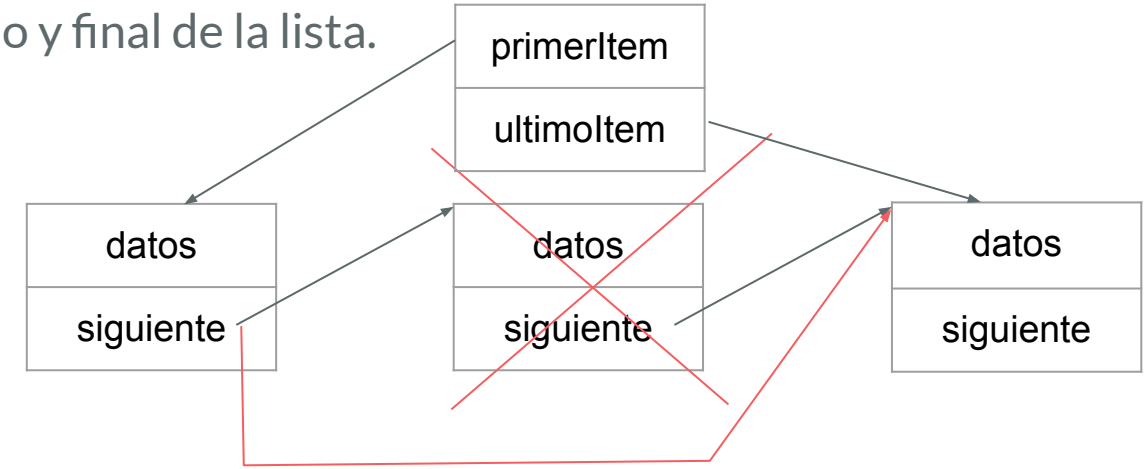
```
itemNuevo = (struct miItem *)malloc(sizeof(struct miItem));
printf("Introduce el valor de x\n");
scanf("%f", &itemNuevo->x);
printf("Introduce el valor de y\n");
scanf("%f", &itemNuevo->y);
itemNuevo->siguiente = NULL; // No hay siguiente

if (lista.primerItem == NULL)
{ // Soy el primero
    // Indicamos en la lista que el primer elemento es el que acabamos de crear
    lista.primerItem = itemNuevo;
    //Como sólo hay uno también es el último
    lista.ultimoItem = itemNuevo;
}
else
{
    // Tiene más de un elemento
    //Me situo en el último y le digo que apunte al nuevo item
    lista.ultimoItem->siguiente = itemNuevo;
    //El nuevo item se convierte en el último
    lista.ultimoItem = itemNuevo;
}
```

No es necesario recorrer la lista. Directamente me sitúo en el último.

Borrar elemento de una lista

Cuando borremos un elemento de la lista utilizaremos la función free() para liberar el espacio de ese elemento.

¿Pero qué ocurre con todos los apuntadores? Tendremos que actualizarlos y habrá que tener en cuenta la posición del elemento a borrar y si tenemos apuntadores al principio y final de la lista.



Borrar elemento de una lista

Ejemplo: Para la lista que únicamente apunta al primer elemento de la lista. Solicitamos al usuario que introduzca las coordenadas a eliminar. Tenemos que tener en cuenta todos los casos. Consideraremos que las coordenadas son únicas, por lo que sólo tendremos que eliminar un elemento.

Borrar elemento de una lista

Si el que se quiere eliminar es el primer elemento debemos actualizar el nuevo primero de la lista que sería el siguiente.

Sino, recorremos la lista buscando si es el siguiente el elemento a eliminar para no perder el puntero.

```c
if (lista.primerItem == NULL)
{
    printf("No hay elementos en la lista");
}
else
{   // Tiene más de un elemento
    printf("Introduce el valor de x\n");
    scanf("%f", &xAux); //Leo en una variable auxiliar
    printf("Introduce el valor de y\n");
    scanf("%f", &yAux); //Leo en una variable auxiliar

    struct miItem *itemAux;            // Puntero para recorrer la lista
    itemAux = lista.primerItem;        // Apunta al primer elemento
    if(itemAux->x == xAux && itemAux->y == yAux){ //El primer elemento es el que queremos borrar
        //El primero ahora es el siguiente elemento
        lista.primerItem = itemAux->siguiente;
        free(itemAux);
        itemAux = NULL;
        printf("Elemento borrado\n");
    }else{
        while (itemAux->siguiente != NULL) // Mientras que no sea el último
        {
            if(itemAux->siguiente->x == xAux && itemAux->siguiente->y == yAux){
                itemBorrar = itemAux->siguiente; //El siguiente es el que tengo que borrar
                itemAux->siguiente = itemAux->siguiente->siguiente;
                //lo he guardado en itemBorrar para no perderlo
                free(itemBorrar);
                itemBorrar = NULL;
                printf("Elemento borrado\n");
                break; //Encontré el elemento a borrar
            }
            itemAux = itemAux->siguiente; // Avanzo al siguiente item
        }
    }
}
```

Borrar elemento de una lista

Ejercicio: Modifica el código de la lista simple que apunta al último elemento para añadir la opción de borrar. Recuerda que debemos comprobar si estamos eliminando el último elemento.

Borrar elemento de una lista

Comprobaciones de si he borrado el último para poder actualizarlo.

```c
if (lista.primerItem == NULL)
{
    printf("No hay elementos en la lista");
}
else
{ // Tiene más de un elemento
    printf("Introduce el valor de x\n");
    scanf("%f", &xAux); // Leo en una variable auxiliar
    printf("Introduce el valor de y\n");
    scanf("%f", &yAux); // Leo en una variable auxiliar
    struct miItem *itemAux;     // Puntero para recorrer la lista
    itemAux = lista.primerItem; // Apunta al primer elemento
    if (itemAux->x == xAux && itemAux->y == yAux)
    { // El primer elemento es el que queremos borrar
        // El primero ahora es el siguiente elemento
        lista.primerItem = itemAux->siguiente;
        // Si el siguiente elemento del que vamos a borrar es NULL
        // es que es el último
        if (itemAux->siguiente == NULL)
        {
            lista.ultimoItem = NULL;
        }
        free(itemAux);
        itemAux = NULL;
        printf("Elemento borrado\n");
    }
    else
    {
        while (itemAux->siguiente != NULL) // Mientras que no sea el último
        {
            if (itemAux->siguiente->x == xAux && itemAux->siguiente->y == yAux)
            {
                itemBorrar = itemAux->siguiente; // El siguiente es el que tengo que borrar
                itemAux->siguiente = itemAux->siguiente->siguiente;
                // Si el siguiente es NULL es que itemBorrar era el último, actualizamos el nuevo último
                if (itemAux->siguiente == NULL)
                {
                    lista.ultimoItem = itemAux;
                }
                free(itemBorrar);
                itemBorrar = NULL;
                printf("Elemento borrado\n");
                break; // Encontré el elemento a borrar
            }
            itemAux = itemAux->siguiente; // Avanzo al siguiente item
        }
    }
}
```

Listas doblemente enlazadas

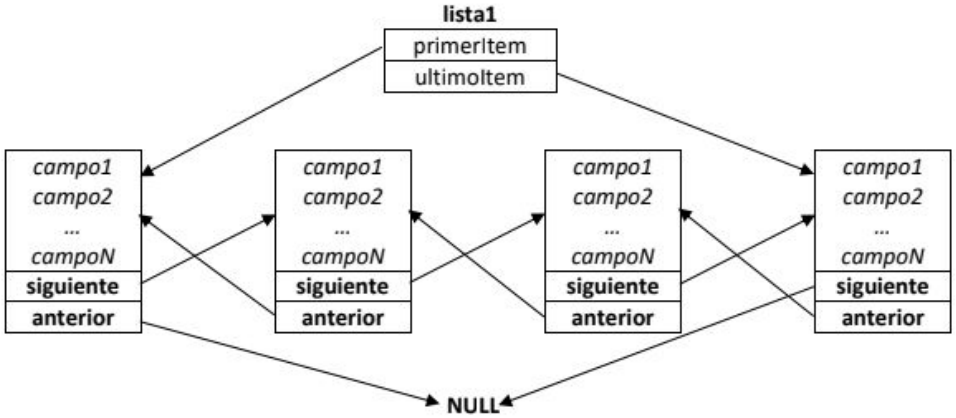
Las listas doblemente enlazadas poseen un doble enlace en sus ítems, a los que podríamos llamar “siguiente” y “anterior”.

Ventajas: Poder recorrer la lista en las dos direcciones, agilizando así muchas tareas relacionadas con la búsqueda, inserción, eliminación...

Desventajas: Tener que ampliar y complicar el código de las operaciones que mantienen la coherencia de los enlaces, además de que el enlace extra consumirá un espacio extra por cada registro.

Listas doblemente enlazadas

```
struct coordenadas
{
    float x;
    float y;
    struct coordenadas *siguiente;
    struct coordenadas *anterior;
};
struct miColeccion
{
    struct coordenadas *primerItem;
    struct coordenadas *ultimoItem;
};
```



Colas

Una cola es una implementación simplificada de una lista simplemente enlazada.

Una cola es una lista en la que:

- Se mantiene el planteamiento de tener un puntero al primer y al último ítem.
- Se inserta por el final. Todas las inserciones se hacen como ítem "siguiente" al último (se agrega al final).
- Se elimina por el principio. El único elemento que se puede eliminar es el "primero". Cuando se elimina el primero el siguiente pasa a ser el primero de cola y el único que puede eliminarse.

Las colas son útiles para cuando necesitamos atender cosas "en el orden de llegada", atendiendo primero al que primero llega. Por eso se conocen también como listas **FIFO**, acrónimo inglés de **First-In-First-Out** (el primero que entra es el primero que sale).

Pila

Es también un caso particular de la lista simple enlazada.

Una pila es como una variante de una cola en la que las inserciones se hacen siempre “por el principio”. Es decir, insertamos y eliminamos siempre el primer elemento de la colección.

El struct de la pila sólo necesita un puntero al primer ítem (el puntero al último no tiene sentido). Cada ítem que se va insertando pasa a ser el primero, desplazando al resto “una posición hacia atrás”.

Cuando se saca un registro se hace siempre sobre el que menos tiempo lleva en la pila.

Las pilas se conocen como listas **LIFO** (**Last-In-First-Out**): el último en llegar es el primero que sale.