# Licenciatura en Sistemas
# Programación de computadoras

***Equipo docente:*** Jorge Golfieri, Natalia Romero, Romina Masilla y Nicolás Perez
***Mails:*** jgolfieri@hotmail.com , nataliab_romero@yahoo.com.ar , romina.e.mansilla@gmail.com, nperez_dcao_smn@outlook.com
***Facebook:*** https://www.facebook.com/groups/171510736842353
***Git:*** ***http://***github.com/UNLASistemasProgramacion/Programacion-de-Computadoras

---

***Unidad 8 – Parte I:***

Estructuras dinámicas lineales de datos. Lista simple y doble enlazada. Pila. Cola. Doble cola. Implementación con TDA y aplicaciones. Problemas: implementación y testeo.

***Bibliografía citada:***

Luis Joyanes Aguilar , Luis Rodríguez Baena y Matilde Fernández Azuela. McGraw-Hill España – Capitulo 12.

Luis Joyanes Aguilar, Andrés Castillo Sanz, y Lucas Sánchez García (2005) - C algoritmos, programación y estructuras de datos - McGraw-Hill España. Capítulo 18 y 19.

---

# *Listas – Guía Teórica*

Para comprender el uso y la necesidad del estudio de las “listas”, primero deberíamos plantearnos lo siguiente: Supongamos que tenemos una estructura Empleado, y yo quiero guardar los datos de algunos de ellos, por ejemplo, para tener una empresa con 10 empleados. Con lo que sabemos hasta ahora hubiéramos hecho un vector de diez componentes, Empleado empleados[10], y si fuéramos un poco más conservadores hubiéramos puesto 100 o 50 o 60 en vez de 10, para asegurarnos que siempre tengamos lugar para agregar algún empleado extra.

Ahora bien, ¿qué pasa si necesitamos por algún motivo agregar 10000 empleados porque nuestra empresa creció demasiado?, claramente nuestro vector no nos va a alcanzar. Los

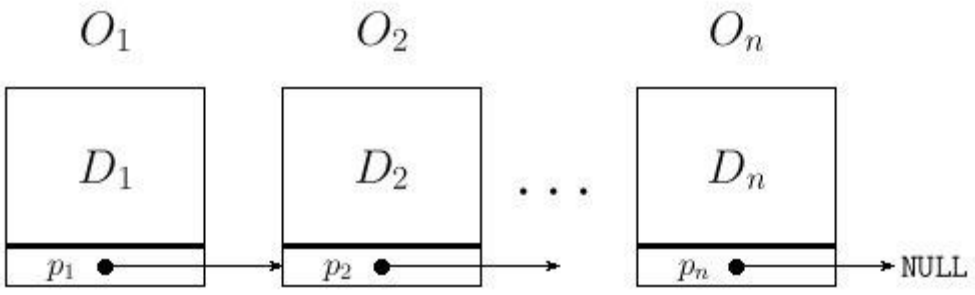
vectores necesitan tener su tamaño definido de antemano, y si no sabemos cuál será de verdad el tamaño máximo podríamos quedarnos cortos. También podría ocurrir lo contrario, hacer un vector de 100 componentes y solo necesitas 2 o 3, por lo que nos sobrarían 98 o 97, esto seria un desperdicio de espacio.

Para solucionar este problema de conocer el tamaño de antemano, surgen las Listas. Si nosotros queremos almacenar el dato de varios empleados desde ahora crearemos una LISTA de empleados.

Una lista es una estructura dinámica de datos. Cada objeto de la estructura está formado por los datos junto con un puntero al siguiente objeto. Al manejar punteros, los datos no tienen por qué estar situados en posiciones consecutivas de la memoria, y lo más normal, es que estén dispersos. Debe imaginarse una lista de la siguiente forma:

$$(O_1, O_2, \ldots, O_n)$$

$O_1$ es el primer objeto y está constituido por dos partes: los datos que pretendemos almacenar, que simbólicamente llamaremos $D_1$, y un puntero $p_1$ que apuntará al siguiente objeto, en éste caso $O_2$, es decir:

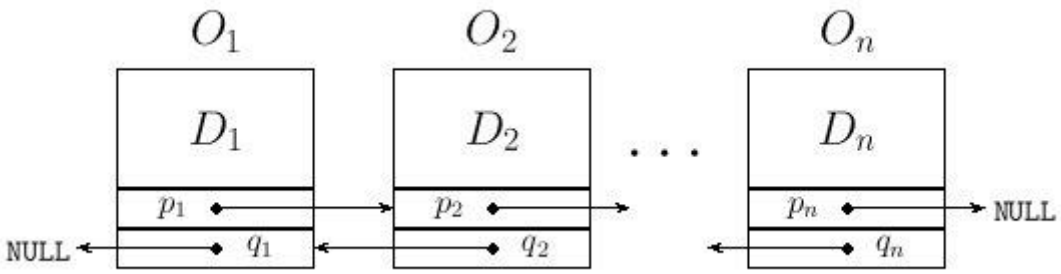



Discusión análoga para $O_2$ y así sucesivamente. El final de la lista es el objeto $O_n$, cuyo puntero $p_n$ apunta a NULL, es decir, a ningún sitio. De esta forma, podemos saber cual es el final de la lista.

Cada elemento de datos, por ejemplo, el $D_1$ de antes, será una estructura (struct).

Las listas no tienen problemas para crecer a izquierda, centro o derecha, dependiendo de lo que se pretenda conseguir.

Las listas también pueden estar doblemente enlazadas, es decir, cada objeto, contiene dos punteros, uno de ellos, al siguiente, como antes, y otro, al anterior. De esta forma, puede ir hacia adelante y hacia atrás:




Nosotros nos enfocaremos en las listas enlazadas. No tanto en las doblemente enlazadas.

## ***Armado de la lista***

Es conveniente separar los datos de los punteros, así que la definición de lista es la siguiente:

```
struct lista { /* lista simple enlazada */
  struct dato datos;
  struct lista *sig;
};
```

Las listas las trataremos con un archivo .h y un archivo .c como ya venimos trabajando.

## ***Lista.h, su estructura y sus primitivas***

Primero definimos el nodo de la lista, es decir esa “cajita” donde guardaremos el puntero al siguiente nodo y el dato que queremos guardar, en este caso un entero ítem.

```
typedef struct {
  int item;
  struct No* prox;
} No;
```

Luego creamos la lista propiamente dicha, donde tenemos su tamaño y un puntero al nodo inicial.

```
typedef struct {
  No* inicio;
  int tam;
} ListaEnc;
```

Acá tenemos una de las principales virtudes y cuestiones que debemos recordar, las listas poseen algunas primitivas que tenemos que recordar porque siempre las usaremos.

En las listas podemos, agregar un dato al inicio, al final o en alguna determinada posición,

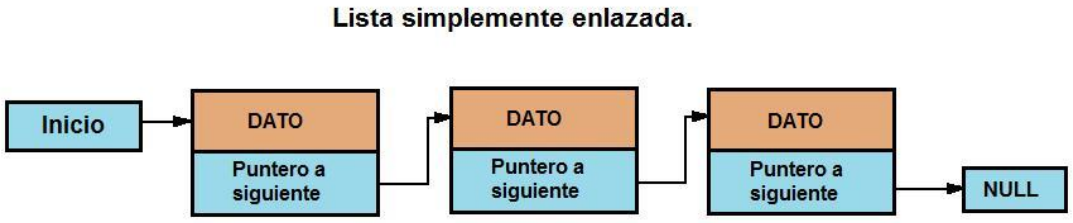
También podemos borrar los datos el inicio, del final o de alguna posición.

Por último, podemos obtener la cantidad de datos que tenemos enlistados y algún elemento en particular.

Otras funciones que nos pueden resultar útiles serian mostrar la lista, ver si está o no vacía y vaciarla.

```
ListaEnc* crearLista();

int liberarLista(ListaEnc* lista);
int estaVacia(ListaEnc* lista);

int insertarInicio(ListaEnc* lista, int item);
int insertar(ListaEnc* lista, int item, int pos);
int insertarFin(ListaEnc* lista, int item);

int removerInicio(ListaEnc* lista, int* item);
```

```
int remover(ListaEnc* lista, int* item, int pos);
int removerFin(ListaEnc* lista, int* item);

int obtenerElemento(ListaEnc* lista, int* item, int pos);
int obtenerTamanio(ListaEnc* lista, int* tam);


void imprimir(ListaEnc* lista);
```




## ***¿Como usar estas funciones?***

De esta forma podríamos usar las primitivas antes mencionadas para ir enlistando números:

```
int main() {

   //CREO LA LISTA CON LA QUE VOY A TRABAJAR
   ListaEnc* miLista = crearLista();



   //INSERTO EL 7 AL PRINCIPIO
   insertarInicio(miLista, 7);
   //INSERTO EL 5 AL FINAL
   insertarFin(miLista, 5);
   //INSERTO EN LA POSICION 1 EL 4
```

```
insertar(miLista, 4,1);
//INSERTO AL FINAL EL 3
insertarFin(miLista, 3);
//QUEDO EL 7   4   5    3


//MUESTRO LO QUE HAY EN LA POSICION 2, ES DECIR EL 5
int i;
obtenerElemento(miLista, &i, 2);
printf("El elemento en la posicion  2 es: %d\n", i);


printf("Probamos el obtener tamaño \n");
printf("Tamaño: %d", obtenerTamanio(miLista));


printf("Esta vacia? \n");
printf("Vacia: %d", estaVacia(miLista));



//MUESTRO LA LISTA ENTERA
imprimir(miLista);
//SACO EL ULTIMO
removerFin(miLista, NULL);
//MUESTRO COMO QUEDO
imprimir(miLista);
//SACO EL 2 ELEMENTO
remover(miLista, NULL, 2);
//MUESTRO
imprimir(miLista);
//SACO EL PRIMERO
removerInicio(miLista, NULL);
```

```
    //MUESTRO
    imprimir(miLista);
}
```

## ***¿Como funcionan las primitivas?***

El funcionamiento detallado de las siguientes funciones y procedimientos no será analizado en profundidad en esta materia, nos interesa mas que logremos dominar su uso y comprensión.

Por ejemplo, al crear la lista vemos que se reserva la memoria necesaria y se devuelve el tamaño cero al crearla, el inicio de la lista será NULL.

```
ListaEnc* crearLista() {
    ListaEnc* lista = malloc(sizeof(*lista));
    if (lista == NULL)
        return NULL;
    lista->inicio = NULL;
    lista->tam = 0;
    return lista;
}

//Esta función lo que hace es limpiar la lista
int liberarLista(ListaEnc* lista) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;

    // remove todos os elementos da lista
    while(!estaVacia(lista))
```

```
        remover(lista, NULL, 0);
    free(lista);
    lista = NULL;
    return OK;
}
```

Con esta función veremos si la lista que estamos usada está vacía o no.

```
int estaVacia(ListaEnc* lista) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    if (lista->inicio == NULL)
        return TRUE;
    return FALSE;
}
```

Aquí empiezan las funciones que mas nos importan, esta por ejemplo nos permite crear un nuevo nodo para almacenar información.

```
No* crearNo(int item, No* prox) {
            No *no = malloc(sizeof(*no));
            if (no == NULL)
        return NULL;
            no->item = item;
            no->prox = prox;
            return no;
}
```

Con esta insertamos un dato al inicio de la lista, en nuestro caso lo que ***insertamos es un entero ítem, si quisiéramos hacer lista de otra cosa esto es lo que deberíamos cambiar, solo esto.***

```
int insertarInicio(ListaEnc* lista, int item) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    No *novoNo = crearNo(item, lista->inicio);
    if (novoNo == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    lista->inicio = novoNo;
    lista->tam++;
    return OK;
}
```

Aquí insertamos un dato, entero ítem, en una determinada posición, nuevamente si queremos insertar otro tipo de dato hay que reemplazar ítem.

```
int insertar(ListaEnc* lista, int item, int pos) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    if (pos < 0 || pos > lista->tam)
        return INDICE_INVALIDO;

    No *novoNo;
    if (pos == 0) {
        return insertarInicio(lista, item);
    } else {
        // prepara para inserir
        No *aux;
        aux = lista->inicio;
        for(int i = 0; i < pos - 1; i++) {
            aux = aux->prox;
        }

        // adiciona o nó
```

```
        novoNo = novoNo = crearNo(item, aux->prox);
        if (novoNo == NULL)
            return ESTRUCTURA_NO_INICIALIZADA;
        aux->prox = novoNo;
    }
    lista->tam++;
    return OK;
}

int insertarFin(ListaEnc* lista, int item) {
    return insertar(lista, item, lista->tam);
}

int removerInicio(ListaEnc* lista, int* item) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    if (estaVacia(lista))
        return ESTRUCTURA_VACIA;
    No *aux = lista->inicio;
    if (item != NULL)
        *item = aux->item;
    lista->inicio = aux->prox;
    free(aux);
    aux = NULL;
    lista->tam--;
    return OK;

}

int remover(ListaEnc* lista, int* item, int pos) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    if (estaVacia(lista))
```

```
        return ESTRUCTURA_VACIA;
    if (pos < 0 || pos >= lista->tam)
        return INDICE_INVALIDO;

    No *ant, *atual;
    if (pos == 0) {
        return removerInicio(lista, item);
    } else {
        // prepara para remover
        ant = NULL;
        atual = lista->inicio;
        for(int i = 0; i < pos; i++) {
            ant = atual;
            atual = atual->prox;
        }


        ant->prox = atual->prox;
        if (item != NULL)
            *item = atual->item;
        free(atual);
        atual = NULL;
    }
    lista->tam--;
    return OK;
}

int removerFin(ListaEnc* lista, int* item) {
    return remover(lista, item, lista->tam - 1);
}

int obtenerElemento(ListaEnc* lista, int* item, int pos) {
    if (lista == NULL)
```

```
        return ESTRUCTURA_NO_INICIALIZADA;
    if (estaVacia(lista))
        return ESTRUCTURA_VACIA;
    if (pos < 0 || pos >= lista->tam)
        return INDICE_INVALIDO;
    if (item == NULL)
        return PARAMETRO_INVALIDO;
    No *aux;
    aux = lista->inicio;
    for(int i = 0; i < pos; i++) {
        aux = aux->prox;
    }
    *item = aux->item;

    return OK;
}

int obtenerTamanio(ListaEnc* lista, int* tam) {
    if (lista == NULL)
        return ESTRUCTURA_NO_INICIALIZADA;
    if (tam == NULL)
        return PARAMETRO_INVALIDO;
    *tam = lista->tam;
    return OK;
}

void imprimir(ListaEnc* lista) {
    int qtdeElementos;
    obtenerTamanio(lista, &qtdeElementos);
    printf("[");
    for(int i = 0;i < qtdeElementos; i++) {
        int el;
        obtenerElemento(lista, &el, i);
```

```
        printf(" %d ", el);
    }
    printf("]\n");
}
```

***<u>En nuestro directorio de github tendrán disponibles siempre para reutilizar la lista tanto el .h como el .c.</u>***

# ***Listas – Guía Practica***

1- Trabajar con el ejercicio propuesto en el apunte teórico, crear un menú que nos permita ingresar datos e irlos enlistando, además que nos permita ver la lista como esta compuesta, que nos permita elegir donde insertar, y que elementos remover.

2- Una vez entendido el funcionamiento de las primitivas, realizar una "nueva primitiva" que nos permita saber si un determinado numero pertenece o no a la lista. Deberá reutilizar la primitiva mostrar para realizar la búsqueda de una forma simple.

3- Realizar un menú semejante al de arriba, pero los datos a enlistar son dato de una Persona. ¿Puede conseguir una primitiva para encontrar o no a una persona en la lista?

4- Crear un TDA Concesionaria, donde la concesionaria tenga una Lista de Autos. ¿Cómo diseñaría este ejercicio? Realizar un menú que nos permita ingresar autos o sacarlo de la lista de la concesionaria.