Estructuras de datos

Tema 4. Listas

Presenta: Mtro. David Martínez Torres
Universidad Tecnológica de la Mixteca
Instituto de Computación
Oficina No. 37
dtorres@gs.utm.mx

Contenido

1. Definiciones y operaciones
2. Implementación dinámica
3. Listas circulares
4. Listas doblemente ligadas
5. Listas de listas
6. Casos de estudio
7. Conclusiones
8. Bibliografía

1. Definiciones y operaciones

La **lista** es una colección ordenada de elementos en la que se pueden insertar y eliminar elementos en el lugar que se requiera. Aunque normalmente se crean **listas ordenadas**.

La **lista ligada** completa se accede desde un apuntador externo **lista** que apunta al que contiene la dirección del primer nodo de la misma, inicialmente **lista** es igual a **NULL**.

El campo **sig** del último nodo de la lista contiene un valor especial conocido como **NULL**. Este apuntador se usa para señalar el final de la lista.

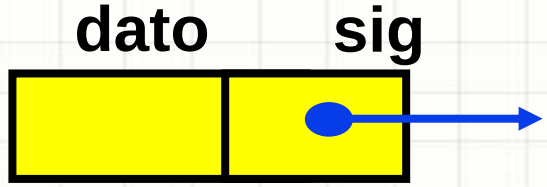
1. Definición y operaciones

Estructura básica de un nodo para una lista dinámica:

```
typedef struct nodo{
      int dato;
      struct nodo *sig;
}nodo;
typedef nodo * nodoPtr;
```

Datos reales

Puntero autoreferenciable



1. Operaciones básicas en listas

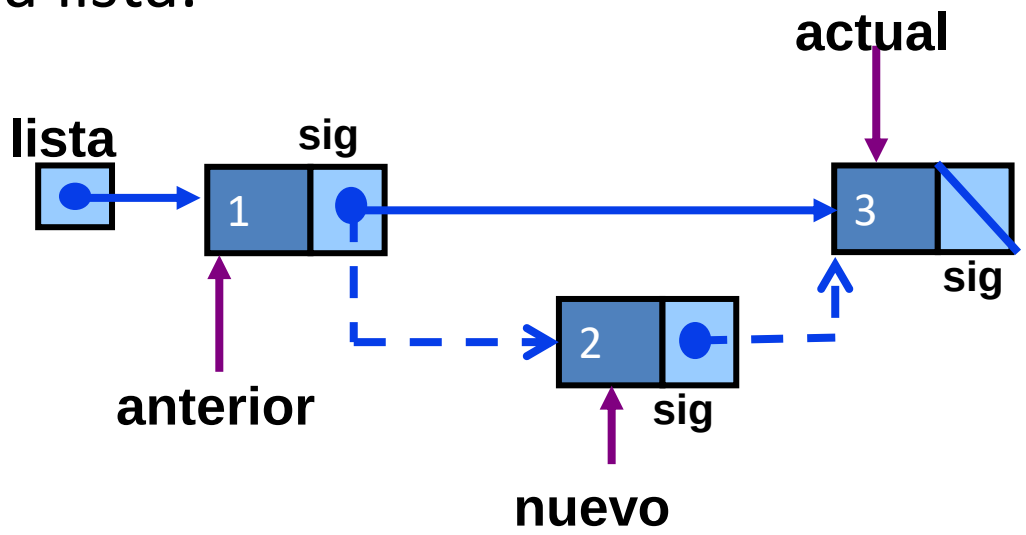
Después de haber visto Pilas y Colas, conviene trabajar una lista ordenada. A continuación se listan algunas operaciones:

- Insertar un elemento a la lista
- Buscar un elemento en la lista
- Imprimir toda la lista
- Eliminar un elemento de la lista
- Modificar los datos de un elemento de la lista
- Escribir un elemento o la lista completa a un archivo
- Leer un archivo e insertar los elementos en una lista.

2. Implementación dinámica

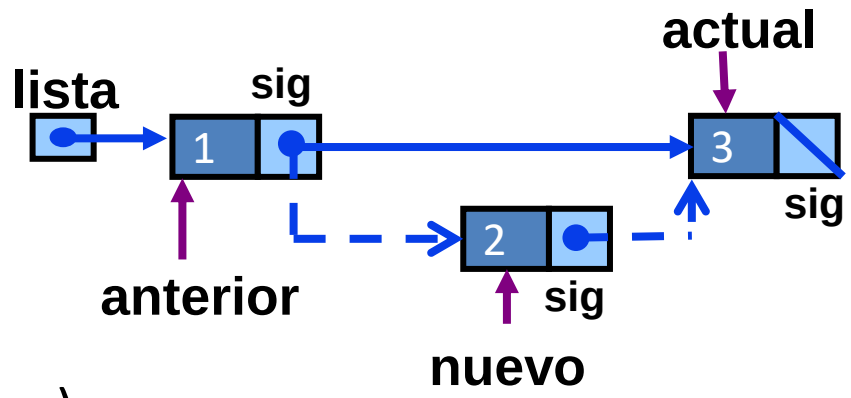
Algoritmo **Insertar** un elemento en una lista ordenada

1. Crear nodo **nuevo,** llenarlo e insertarlo
2. Si la lista está vacía, nuevo será el primer nodo de la lista.
3. En otro caso buscar la posición a insertarlo mediante apuntador anterior y actual. Si apuntador **anterior** es NULL, el nodo se inserta al inicio de la lista, en otro caso en medio o al final de la lista.



2. Implementación dinámica: Insertar

```
typedef struct nodo{
        int dato;
        struct nodo*sig;
}nodo;
typedef nodo * nodoPtr;

void insertar(nodo **listaT, int num);
void imprimir(nodo *listaT);
int eliminar(nodo **listaT, int num);

int main(){
nodo  *lista=NULL;

insertar(&lista, 3);
insertar(&lista, 1);
insertar(&lista, 2);
Insertar(&lista, 8);
imprimir(lista);
eliminar(&lista, 2);
…
}
```



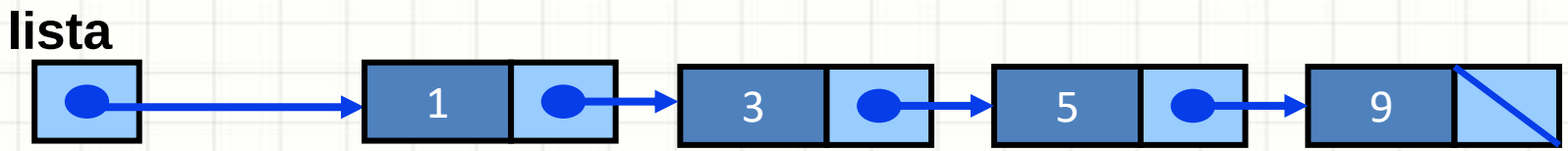
Realice una prueba de escritorio

2. Implementación dinámica: Insertar ordenada

Termine de realizar la prueba de escritorio

```
void insertar(nodo **listaT, int num){
nodo *nuevo, *anterior, *actual,
nuevo=(nodo *) malloc(sizeof(nodo));
if(nuevo==NULL)
      printf("no hay memoria\n");
else {
      nuevo->dato=num;
      nuevo->sig=NULL;
      if(*listaT==NULL)
            *listaT=nuevo;
      else {
            anterior=NULL;
            actual=*listaT;
      while(actual!=NULL && num > actual->dato){
            anterior=actual;
            actual=actual->sig;
      }
      if(anterior==NULL){
            nuevo->sig=*listaT;
            *listaT=nuevo;
      } else{
            anterior->sig=nuevo;
            nuevo->sig=actual;
      }
   }
}
}
```

2. Implementación dinámica: Imprimir lista

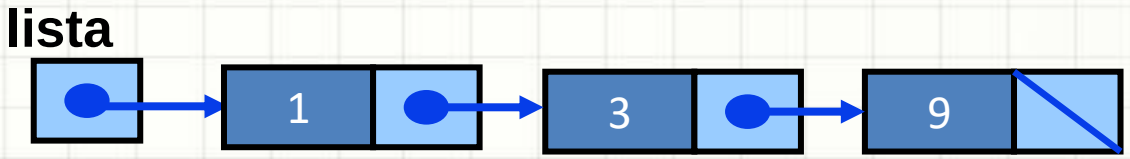



```
void imprimir(nodoPtr lista){
while(lista) {
      printf("%d ->", lista->dato);
      lista = lista->sig;
}
printf("NULL");
}
```

Realice una prueba de escritorio

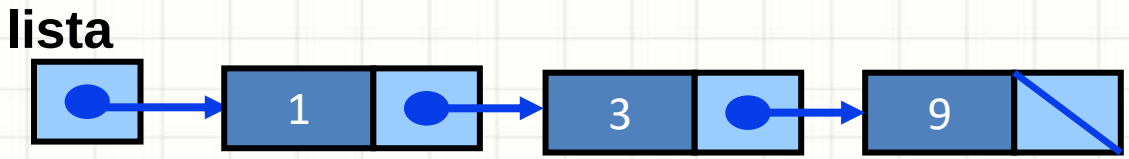
```
void eliminar(nodoPtr *listaT, int dato){
nodoPtr ant, act, temp;
if(*listaT==NULL){
        printf("La lista esta vacia\n");
        return;
}
if((*listaT)->dato==dato){
        temp=*listaT;
        *listaT=(*listaT)->sig;
        free(temp);
        printf("Elemento borrado.");
}else {
        ant=*listaT;
        act=(*listaT)->sig;
        while(act!=NULL && act->dato!=dato){
                ant=act;
                act=act->sig;
        }//termina while
```

2. Implementación dinámica: Eliminar




```
    //continua
    if(act!=NULL){
            temp=act;
            ant->sig=act->sig;
            free(temp);
            printf("Elemento borrado.");
    } else
            printf("No existe este elemento en la lista.");
    }//termina else
}
```

2. Implementación dinámica

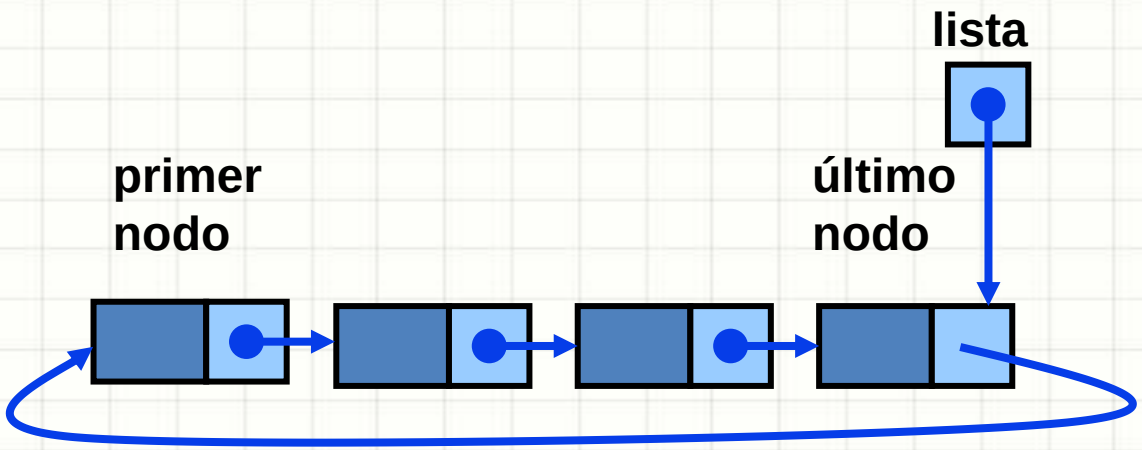



```
nodoPtr  buscar(nodoPtr  lista, int dato) {
nodoPtr  temp=NULL;
while (lista != NULL && lista->dato!=dato )
        lista = lista->sig;
if(lista!=NULL)
        temp=lista;
return temp;
}
```

Realice una prueba de escritorio e integre a un programa

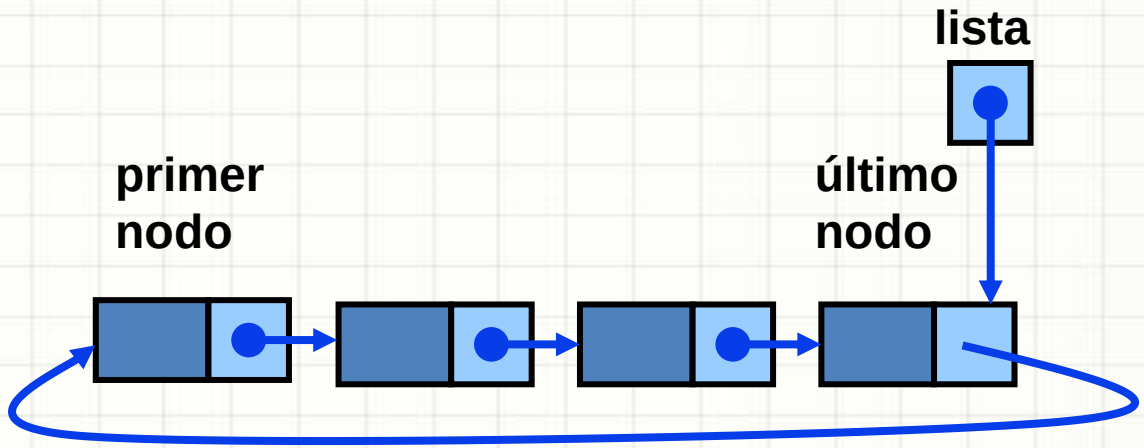
3. Listas circulares

Son listas ligadas lineales, con un ligero cambio al último nodo en su campo siguiente, en lugar de apuntar a NULL, debe apuntar al primer nodo, de ahí que se les conozca como listas circulares.



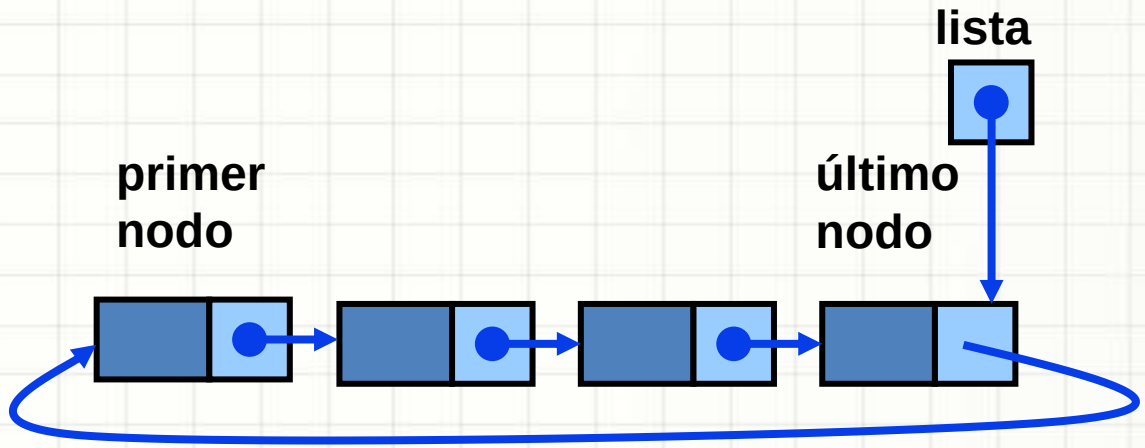
```
typedef struct nodo{
        char nombre[20];
        struct nodo *sig;
}nodo;
typedef nodo* nodoPtr;

void insertar(nodoPtr *, char []);
int main(){
nodoPtr lista=NULL;

insertar(&lista, "leche");
insertar(&lista, "azucar");
imprimir(lista);
eliminar(&lista, "leche");
...
}
```

3. Listas circulares: Insertar un elemento



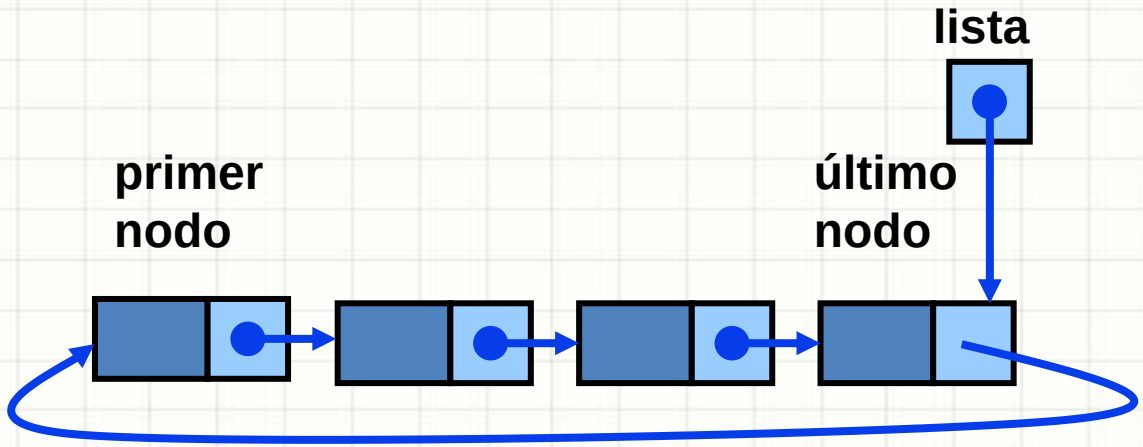
```
void insertar(nodoPtr *listaT, char dato[]){
nodoPtr nuevo;
nuevo=crearNodo();
if(nuevo!=NULL){
      strcpy(nuevo->dato,dato);
      if(*listaT==NULL){
            nuevo->sig=nuevo;
            *listaT=nuevo;
      } else {
            nuevo->sig=(*listaT)->sig;
            (*listaT)->sig=nuevo;
            *listaT=nuevo;
      }
}
}
```

3. Listas circulares: Insertar un elemento



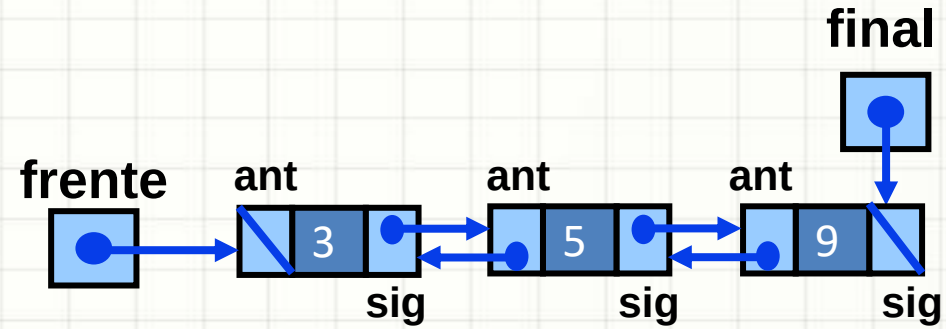
3. Listas circulares

Escribir las funciones imprimir, eliminar y buscar un elemento de una lista circular. Después, agréguelas a un programa con un menú de opciones.

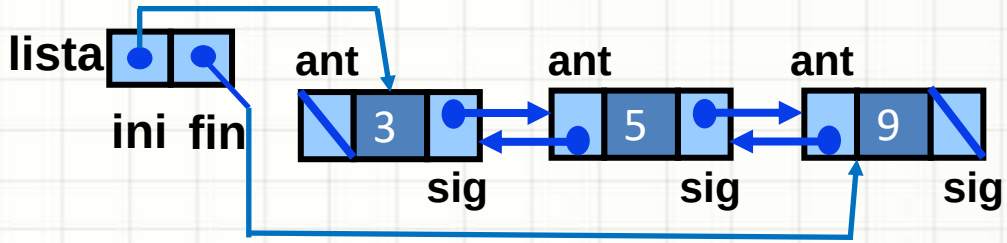


4. Listas doblemente ligadas

Son **listas ligadas abiertas**, donde cada nodo tiene otro apuntador al **nodo anterior**



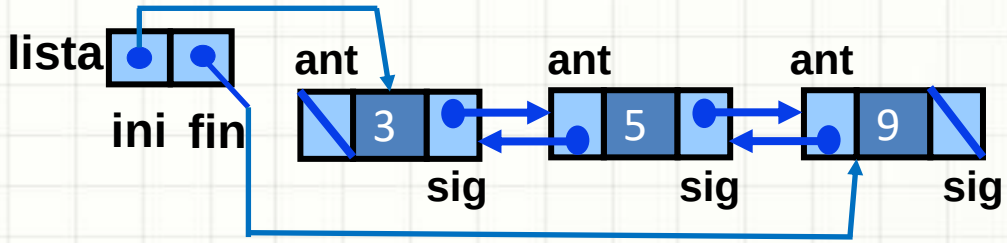
√



Se pueden **recorrer las listas en ambas direcciones** si tenemos un apuntador externo al final de la lista, como si fuera una cola.

De los dos esquemas, el segundo tiene más ventajas.

4. Listas doblemente ligadas




```
typedef struct nodo{
      char nombre[30];
      struct nodo *ant,*sig;
}nodo;
typedef struct lista{
      nodo *ini;
      nodo *fin
}Lista

typedef Lista * ListaPtr;
```

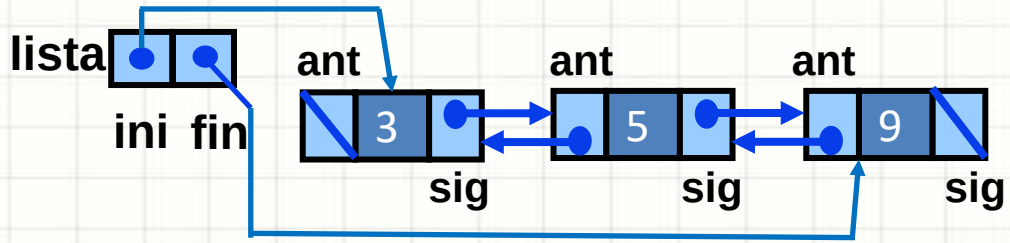
```
void insertar(ListaPtr , char dato[]);
void imprimirAsc(nodo *);
void imprimirDesc(nodo *);

void main(){
Lista lista={NULL};

insertar(&lista, “teclado”);
insertar(&lista, “bocinas”);
imprimirAsc(lista.ini);
imprimirDesc(lista.fin);
}
```

4. Listas doblemente ligadas Ordenadas: Insertar

```
void insertar(ListaPtr lista, char dato[]){
nodo *nuevo, *anterior, *act;
nuevo=crearNodo(dato);
if(nuevo!=NULL){
        anterior=NULL;
        act = lista->ini;
        while(act!=NULL y strcmp(act->dato,dato)<0){
                anterior = act;
                act = act->sig;
        }
        if (anterior == NULL){
                lista->ini = nuevo;
                if (act==NULL)
                        lista->fin = nuevo;
                else {
                        nuevo->sig = act;
                        act->ant = nuevo;
                }
        }
        else {
                anterior->sig = nuevo;
                nuevo->ant = anterior;
                if(act!=NULL){
                        nuevo->sig = act;
                        act->ant = nuevo;
                }else
                        lista->fin = nuevo;
        } //fin if anterior
        } //fin if nuevo
        } //fin insertar
```

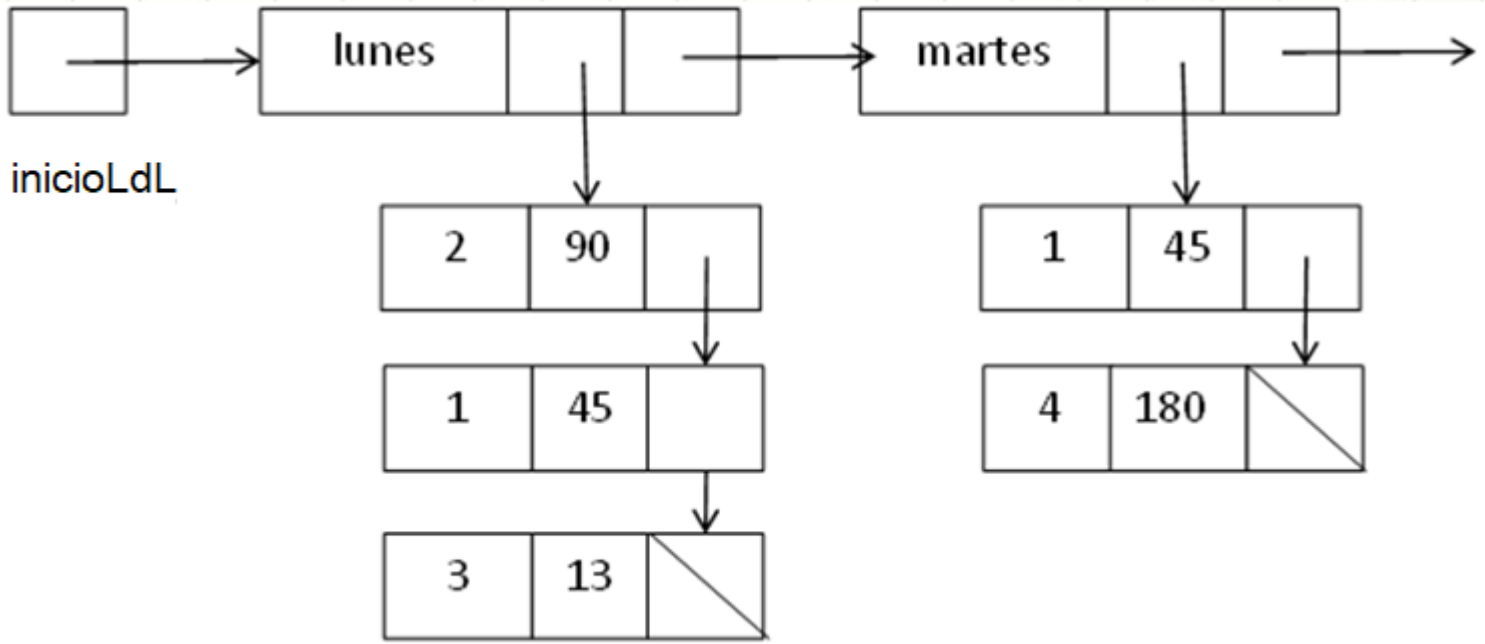
4. Listas doblemente ligadas Ordenadas

Escribir las funciones, imprimir de forma ascendente y de forma descendente, eliminar y buscar. Después, agréguelas a un programa con un menú de opciones.



5. Listas de listas ligadas

Son aquellas listas ligadas en las que uno o más **apuntadores internos** de un nodo puede ser el inicio para formar otras listas.



Ejercicio:

El Circo X, tiene funciones todos los días de la semana (lunes-domingo) y le han solicitado que implemente un programa para que conozca entre otras cosas, el número de boletos vendidos por día y las entradas totales por día; el día que hubo más entradas, el día que hubo menos entradas y las entradas totales de toda la semana.

Para esto, Ud tiene que diseñar las estructuras de datos para el uso de listas de listas, donde cada nodo de la lista contiene el nombre de un día de la semana, una lista simple (donde cada nodo contiene cantidad de boletos vendidos, importe de los mismos y un apuntador al siguiente nodo de la lista de boletos), así como el apuntador al siguiente nodo del día. A continuación se presenta un bosquejo de los tipos de datos.

```
typedef struct nodoL{
        int cantBoletos;
        float precioBoleto;
        struct nodoL *sig;
}NodoL;

typedef NodoL * NodoLPtr;
```

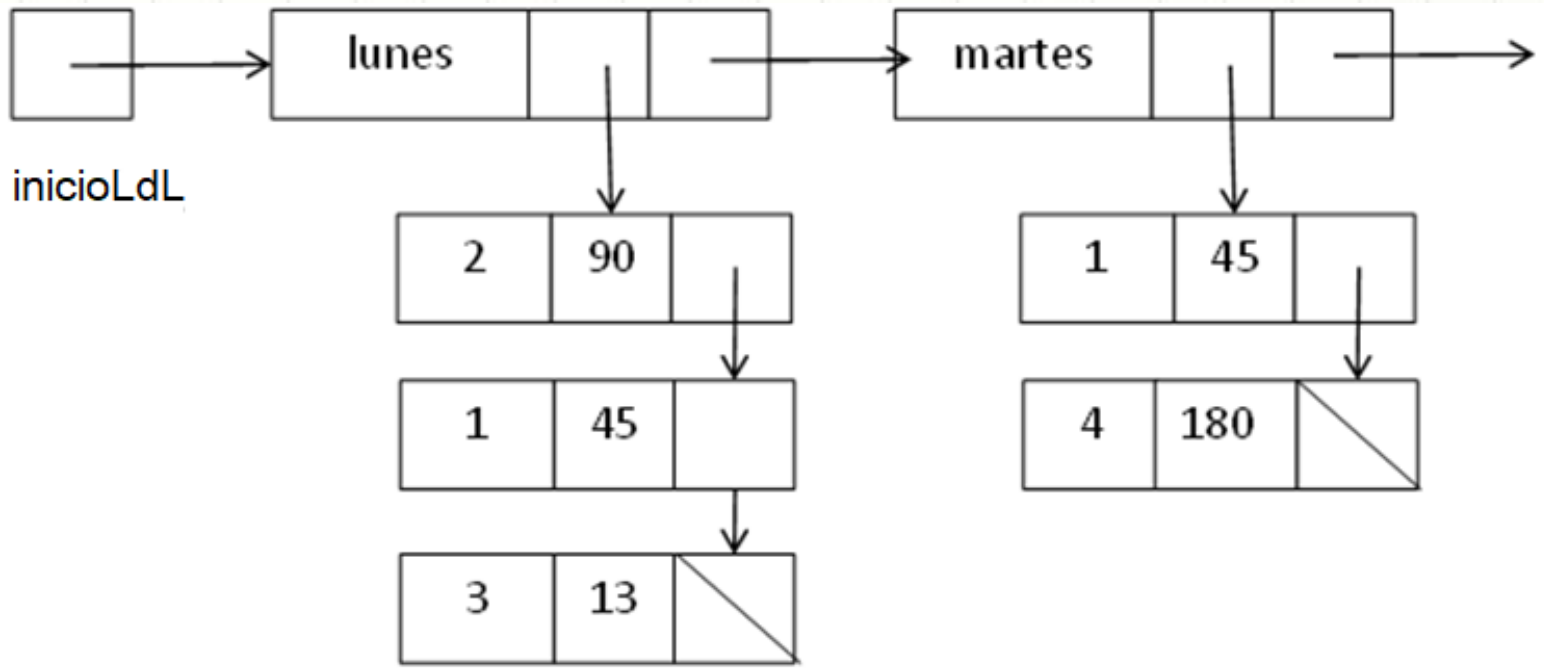
```
typedef struct nodoLdL{
        char dia[30];
        float totalEntradasDia;
        NodoL * inicioL;
        struct nodoLdL * sig;
}NodoLdL;

typedef NodoLdL * NodoLdLPtr;
```

5. Listas de listas ligadas

Se presentan posibles tipos de datos abstractos para el ejemplo.

```
typedef struct nodoL{
        int cantBoleto;
        float precioBoleto;
        struct nodoL *sig;
}NodoL;

typedef NodoL * NodoLPtr;
```

```
typedef struct nodoLdL{
        char dia[30];
        float totalEntradasDia;
        NodoL * inicioL;
        struct nodoLdL * sig;
}NodoLdL;

typedef NodoLdL * NodoLdLPtr;
```



Escriba las funciones que preparen la lista de listas:

1. Crear lista de los días de la semana. Invoca a la función **insertarDiaSemana**, que recibirá el apuntador a la lista de listas, y un nombre del día de la semana, el nodo se insertarán en el orden correspondiente a los días de la semana.
2. Insertar boletos al día que corresponda. Invoca a la función **insertarBoletosDia**, que recibirá el número de boletos a comprar y el apuntador al inicio de la lista de boletos previa búsqueda del nodo correspondiente al día de la semana. La función insertará el nodo como una pila.
3. Listar los boletos vendidos de toda la semana. Invoca a la función **imprimirListaDeLista**, esta función imprimirá cada uno de los nodos de los días de la semana con sus respectivas listas internas de boletos.

4. Para un día de la semana, elimine todos los nodos de boletos calculando y actualizando en el campo totalEntradasDia. Invocar a la función **eliminarBoletosDia**, que reciba la lista de listas y un nombre de un día de la semana.
5. Una función **calcularDiaMasEntradas** que reciba la lista de listas. La función calculará y devolverá el día que tuvo más entradas durante la semana, con su correspondiente totalEntradasDia.
6. Una función **calcularTotalEntradasSemana**, que reciba la lista de listas y que calcule y devuelva las entradas totales de toda la semana.

Por último, en la función principal con un menú de opciones invoque a las siguientes funciones que permitirán conocer lo solicitado por el cliente.

6. Casos de estudio

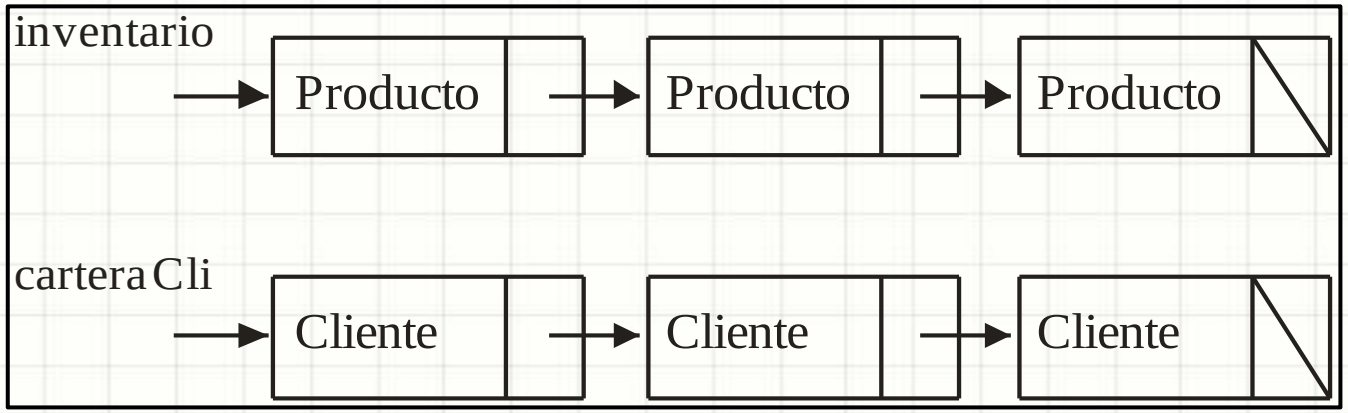
1. Proyecto de supermercado
2. Proyecto de un sistema bancario
3. Proyecto de un sistema de casetas de cobro
4. Proyecto para la gestión de un Cine
5. Proyecto para la gestión de un hospital
6. Etc.

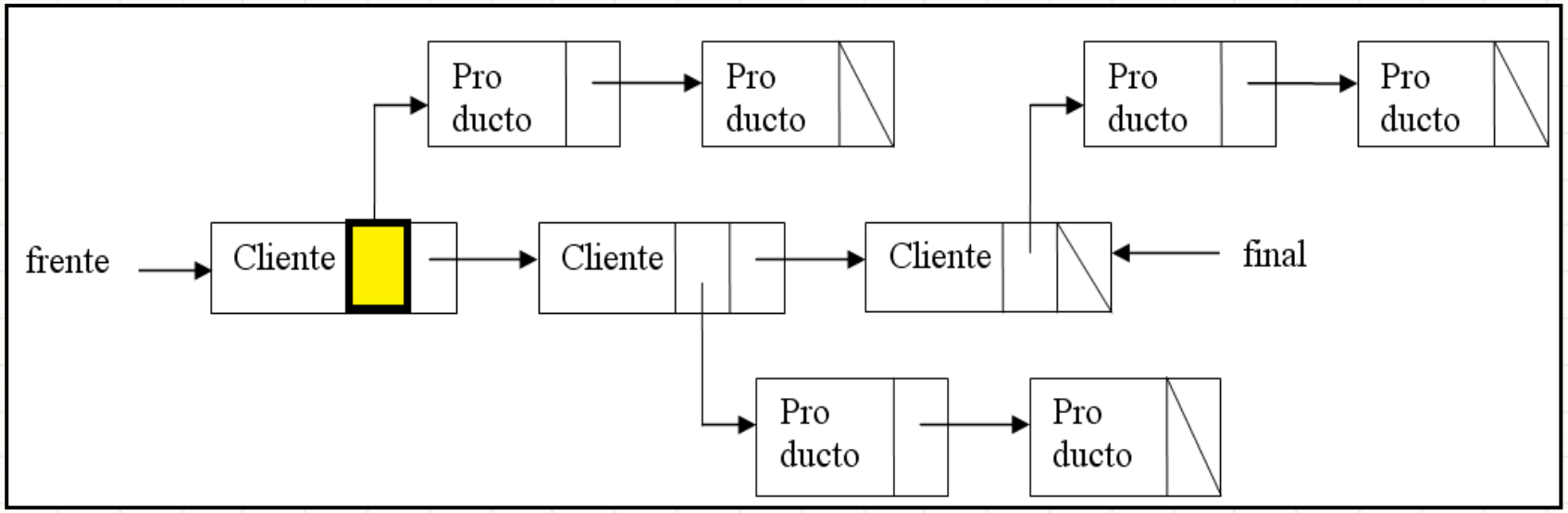
6. Casos de estudio: Supermercado

A continuación se explica las estructura de datos que se implementarán en el proyecto. La especificación completa se encuentra en el documento de requerimientos en classroom.

1. Listas para representar el inventario de productos.
2. Listas para representar la cartera de clientes.
3. Colas para representar una cola de clientes en una caja de cobro como lista de listas.
4. Pilas para representar el carrito de compras, y
5. Listas para representar las ganancias.

6. Casos de estudio: Supermercado





6. Casos de estudio: Supermercado

A continuación se presenta un posible diseño para las estructuras de datos.

```
typedef struct producto{
        int id;
        char nombre[20];
        float precioC;
        float precioV;
}Producto;

typedef struct cliente{
        int numCte;
        char nombre[30];
        char telefono[15];
}Cliente;

typedef struct nodoCola{
        Cliente cliente;
        Producto *carritoProd;
        float efectivo;
        float total;
        struct nodoCola *sig;
}NodoCola;

typedef struct{
        NodoCola * ini;
        NodoCola * fin;
}Cola;
```

```
typedef struct inventario{
        Producto producto;
        int cantidad;
        struct inventario *sig;
}Inventario;

typedef struct carteraC{
        Cliente cliente;
        struct carteraC *sig;
}CarteraC;

typedef struct ganancia{
        int idGanancia;
        char fecha[20];
        struct ganancia *sig;
}Ganancia;
```

Implementación dinámica

```
void insertarCola(Cola *cola, Inventario *inv, CarteraC *carteraC){
NodoCola *nuevo;
//Reservar memoria a nuevo
//pedir id del cliente y buscarlo
nuevo->cliente=clienteBuscado
nuevo->carritoProd=NULL
nuevo->sig=NULL
//se llena el carrito de compras
hacer
        teclear id producto a comprar idProd
        //se busca el producto, si se encuentra se
        //inserta como pila de acuerdo a la especificación
        pushCarritoProd(&nuevo->carritoProd, producto)
mientras comprarMasProductos
```

```
Si cola->ini = NULL
    cola->ini=nuevo
Sino
    cola->fin->sig=nuevo
cola->fin=nuevo
```

Se presenta un pequeño bosquejo del algoritmo de insertar un cliente con su carrito en la cola

6. Caso estudio Supermercado

La función que inserta un cliente en la cola, recibe los apuntadores a la cola, el inventario y la cartera de clientes.

```
void insertarCola(Cola *cola, Inventario *inv,
CarteraC *carteraC){
…
}
```

6. Caso estudio Supermercado

- Código que puede utilizar para obtener la fecha del sistema para la parte de ganancias.

```
time_t tiempo = time(0);
struct tm *tlocal = localtime(&tiempo);
char fecha[128];

strftime(fecha,128,"%d/%m/%y %H:%M:%S",tlocal);
```

6. Caso estudio Supermercado

- Terminar el proyecto final, cubriendo los requerimientos indicados en el documento "EspecificaciónProyectoFinal.pdf".
- El proyecto se entrega para el examen ordinario

Referencias

1. Tenenbaum, Aaron & Langsam, Yedidyah & Augenstein, Moshe “Estructuras de Datos en C”. Prentice-Hall, México 1997.
2. Deitel & Deitel “Como programar en C/C++”. Prentice-Hall, México
3. Wirth, Niklaus “Algoritmos y estructura de Datos”. Prentice-Hall, México.