

## Tipos de Datos Abstractos (TDA) - LISTA.

### Más acerca de las primitivas.

Cuando en el proyecto de prueba se utilizaron algunas (poquísimas) primitivas del TDA LISTA...

```
void probarPonerYSacarDeLista(void)
{
    tLista lista;

    crearLista(&lista);
    if(listaLlena(&lista, 1))
        puts("Que pasa? No hay lugar ni para un Byte? No hay memoria???");
    if(listaVacia(&lista))
        puts("Obviamente que la lista esta vacia! Recien creada!");
    vaciarLista(&lista);
}
```

Con las tres primeras, no hay ninguna duda, son de las vistas en la primera parte.

Pero ¿qué tal si se requiere que la primitiva que vacía la lista devuelva cuántos eliminó?

Es una nueva primitiva a resolver (no es el caso de los TDA PILA y TDA COLA que sólo admiten siete). Esta primitiva tendrá una ínfima sobrecarga en tiempo de ejecución sobre la anterior, con lo que se la podrá modificar para que haga lo que hacía y además devuelva cuántos eliminó, dejando de este modo una única primitiva.

```
int listaLlena(const tLista *p, unsigned cantBytes);

//void vaciarLista(tLista *p); <-- Reemplazo la primitiva y luego elimino su declaración
int vaciarLista(tLista *p);
```

```
//void vaciarLista(tLista *p) <-- modifico la definición
int vaciarLista(tLista *p)
{
    int cant = 0;
    while(*p)
    {
        tNodo *aux = *p;

        cant++;
        *p = aux->sig;
        free(aux->info);
        free(aux);
    }
    return cant;
}
```

En el código anterior, el mayor costo de ejecución lo tienen las dos invocaciones a free, el del incremento de una variable es ínfimo.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

---

Pero, ¿qué tal si es necesario que la primitiva que vacía la lista y devuelve cuántos eliminó, que además los muestre por pantalla?

Esta sí, es una nueva primitiva a resolver. Esta primitiva tendrá una sobrecarga mucho mayor en tiempo de ejecución sobre la anterior por el hecho de mostrar por pantalla.

Hay que tener en cuenta que para un TDA LISTA que se precie de serlo (porque con las mismas primitivas se puede hacer una lista de personas, productos, empleados e incluso de líneas de texto), para mostrar los elementos que se están eliminando, la primitiva "*recibirá*" la función que "*sabe mostrar*" esa información<sup>(1)</sup>. La otra sobrecarga en tiempo de ejecución estará dada en el uso de la función de biblioteca "**printf**" (o "**puts**"). Por lo tanto esta primitiva no debe reemplazar (aunque podría hacerlo con mucho mayor costo de ejecución) a la anterior.

Y acá, un nuevo . . .

Pero, ¿qué tal si la primitiva que vacía la lista y devuelve cuántos eliminó, además los graba en un archivo binario?, ¿y si en vez de binario es de texto?

La respuesta para las dos "Pero, ¿qué tal si . . . ?" es que con una sola primitiva se resuelven todos estos casos. El único *detalle* es que además de recibir una función que "*sabe mostrar*" la información de los elementos de la lista recibe un puntero a **FILE** (con el archivo ya abierto). Se tendrá en cuenta que si en lugar de invocarla con el puntero a **FILE** la invoca con **stdout** (o **stderr**), mostrará por pantalla (para el Lenguaje C, la pantalla, además del teclado, es un **stream**). Por supuesto que la función que "*sabe mostrar*" debe usar "**fprintf**" con el puntero a **FILE** que recibe. De este modo, mostrará por pantalla o grabará en archivo de texto.

Además, como la lista responde a un TDA en el que se ponen elementos de cualquier tipo, la declaración y la definición de las funciones que "*saben mostrar*" recibirán un puntero genérico (o sea un puntero **void**) a la información además del puntero a **FILE**).

Ejemplificando: . . .



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

En "lista.h" la declaración de **vaciarListaYMostrar**...

```
int vaciarListaYMostrar(const tLista *p, unsigned cantBytes);

int vaciarLista(tLista *p);          <-- la declaración anterior ya fue eliminada y
                                     <-- queda la nueva

int vaciarListaYMostrar(tLista *p, <-- se agrega la declaración de la nueva primitiva
                             void (*mostrar)(const void *, FILE *), FILE *fp);

int ponerAlComienzo(tLista *p, const void *d, unsigned cantBytes);
```

En "lista.c" la definición de la nueva primitiva...

```
int vaciarListaYMostrar(tLista *p,
                       void (*mostrar)(const void *, FILE *), FILE *fp)
{
    int cant = 0;
    while(*p)                mientras hay nodo
    {
        tNodo *aux = *p;     'memoriza' dónde está

        cant++;
        *p = aux->sig;        lo contabiliza
                             lo desengancha
        if(mostrar && fp)     no fallará si no recibe la función o el archivo
        {                   muestra el elemento
            mostrar(aux->info, fp);
            free(aux->info);   libera la memoria del elemento
            free(aux);         libera la memoria del nodo
        }
    }
    return cant;             fin-mientras
}
```

En "productos.c" la definición de "mostrarProducto" se transforma, por ejemplo, en "mostrarProductoTxt"...

```
30 //void mostrarProducto(const tProd *d) <-- luego se eliminará la anterior definición
31 void mostrarProductoTxt(const tProd *d, FILE *fp)
32 {
33     if(d)
34     {
35         fprintf(stdout,
36                 "%-*s %-*s ...\n",
37                 (int)sizeof(d->codProd) - 1, d->codProd,
38                 (int)sizeof(d->descrip) - 1, d->descrip);
39     }
40     else
41     {
42         fprintf(stdout,
43                 "%-*s %-*s ...\n",
44                 (int)sizeof(d->codProd) - 1, (int)sizeof(d->codProd) - 1,
45                 "Cod. Producto",
46                 (int)sizeof(d->descrip) - 1, (int)sizeof(d->descrip) - 1,
47                 "Descripcion del producto");
48     }
49 }
```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

... la definición de "mostrarProducto" queda reescrita como se ve más abajo, además de las funciones que permiten ser utilizadas desde los TDA para mostrar por pantalla / grabar en archivo de texto / grabar en archivo binario ...

```

49
50 void mostrarProducto(const tProd *d)
51 {
52     mostrarProductoTxt(d, stdout);
53 }
54
55 void mostrarProductoTxtTDA(const void *d, FILE *fp)
56 {
57     mostrarProductoTxt((const tProd *)d, fp);
58 }
59
60
61 void grabarProductoBinTDA(const void *d, FILE *fp)
62 {
63     fwrite(d, sizeof(tProd), 1, fp);
64 }
65
66
67

```

... y en "productos.h" las declaraciones quedan ...

```

void mostrarProducto(const tProd *d);

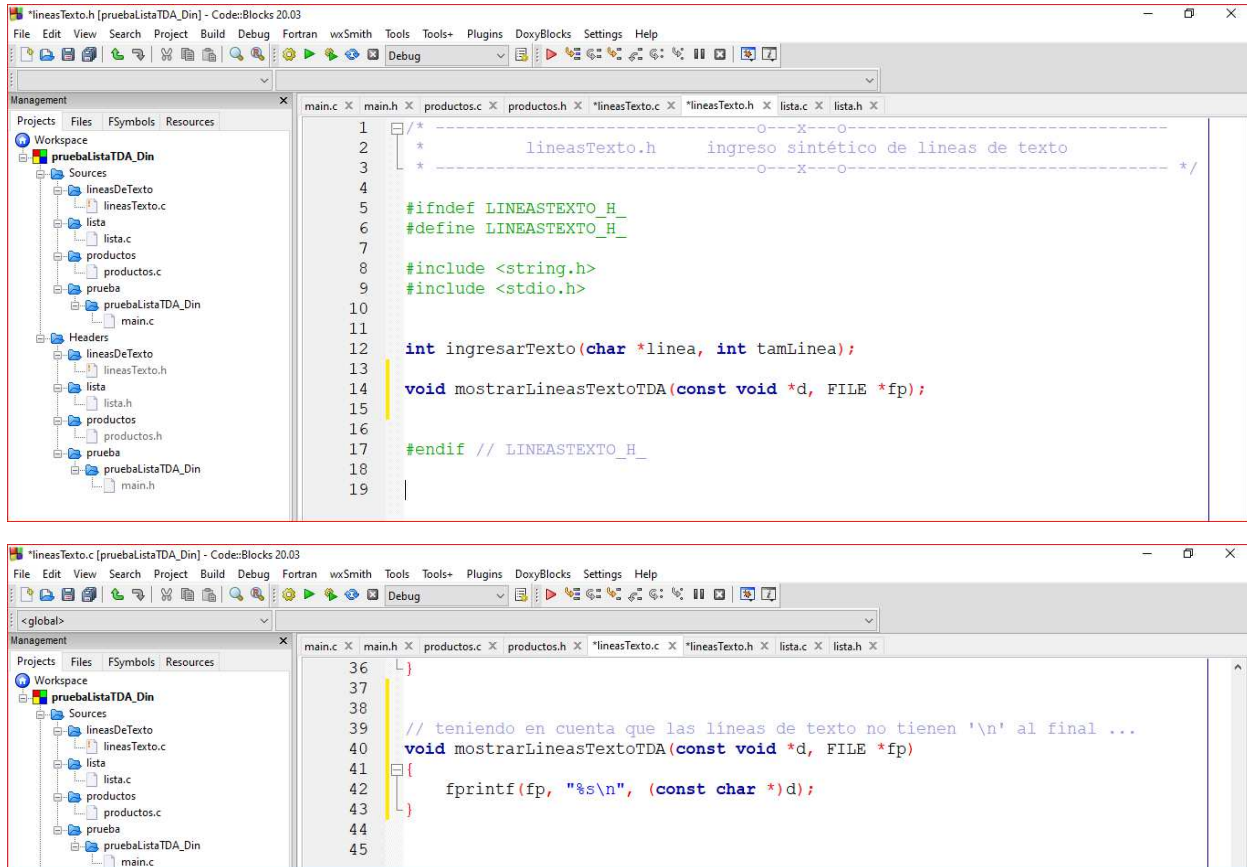
void mostrarProductoTxt(const tProd *d, FILE *fp);

void mostrarProductoTxtTDA(const void *d, FILE *fp);

void grabarProductoBinTDA(const void *d, FILE *fp);

```

Por su parte, en "lineasTexto.h" y "lineasTexto.c" ...



The image shows two screenshots of the Code::Blocks IDE. The top screenshot displays the header file "lineasTexto.h" with the following code:

```
1  /* -----X----- */
2  /*           lineasTexto.h   ingreso sintético de líneas de texto           */
3  /* -----X----- */
4
5  #ifndef LINEASTEXTO_H_
6  #define LINEASTEXTO_H_
7
8  #include <string.h>
9  #include <stdio.h>
10
11
12  int ingresarTexto(char *linea, int tamLinea);
13
14  void mostrarLineasTextoTDA(const void *d, FILE *fp);
15
16
17 #endif // LINEASTEXTO_H_
18
19
```

The bottom screenshot displays the source file "lineasTexto.c" with the following code:

```
36
37
38
39 // teniendo en cuenta que las líneas de texto no tienen '\n' al final ...
40 void mostrarLineasTextoTDA(const void *d, FILE *fp)
41 {
42     fprintf(fp, "%s\n", (const char *)d);
43 }
44
45
```

... se agrega la declaración y la definición de la función que muestra por pantalla.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Pase a probar la primitiva **"vaciarListaYMostrar"** junto con algunas más . . .

```

main.c x main.h x productos.c x productos.h x lineasTexto.c x lineasTexto.h x lista.c x lista.h x
51  /**/* -----X---O---X----- *//**/
52  /**/* FUNCIONES DE SERVICIO *//**/
53  /**/* -----X---O---X----- *//**/
54  #define CON_MSJ          1
55
56  int _abrirArchivo(FILE **fp, const char *nomArch, const char *modo,
57                  int conSinMsj)
58  {
59      *fp = fopen(nomArch, modo);
60      if(*fp == NULL)
61      {
62          if(conSinMsj == CON_MSJ)
63              fprintf(stderr,
64                  "ERROR - abriendo archivo \"%s\" en modo \"%s\".\n",
65                  nomArch, modo ? modo : "NULL");
66          return 0;
67      }
68      return 1;
69  }
70
71  void _verYCerrarArchivo(FILE *fp, const char *nomArch)
72  {
73      char comando[50];
74
75      if(fp == stdout || fp == stderr)
76          return;
77      fflush(fp);
78
79      fclose(fp);
80      sprintf(comando, "start /low notepad.exe %s", nomArch);
81      system(comando);
82  }
83  int _probarPonerAlComienzo(tLista *lista, FILE *fp) /**/
84  {
85      tProd  prod;
86      int    cant = 0;
87
88      fprintf(fp, "- Probando poner al comienzo.\n");
89      if(ingresarProducto(&prod))
90          mostrarProductoTxt(NULL, fp);
91      do
92      {
93          if(!ponerAlComienzo(lista, &prod, sizeof(prod)))
94          {
95              fprintf(stderr,
96                  "ERROR - inesperado - lista llena.\n"
97                  "Producto no cargado:\n");
98              mostrarProductoTxt(&prod, stderr);
99          }
100         else
101         {
102             cant++;
103             mostrarProductoTxt(&prod, stderr);
104         }
105     } while(ingresarProducto(&prod));

```





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

105     } while(ingresarProducto(&prod));
106     fprintf(fp, "Se pusieron al comienzo de la lista %d productos.\n", cant);
107     return cant;
108 }
109
110 int _probarPonerAlFinal(tLista *lista, FILE *fp) /**/
111 {
112     tProd prod;
113     int cant = 0;
114
115     fprintf(fp, "- Probando poner al final.\n");
116     if(ingresarProducto(&prod))
117         mostrarProductoTxt(NULL, fp);
118     do
119     {
120         if(!ponerAlFinal(lista, &prod, sizeof(prod)))
121         {
122             fprintf(stderr,
123                 "ERROR - inesperado - lista llena.\n"
124                 "Producto no cargado:\n");
125             mostrarProductoTxt(&prod, stderr);
126         }
127         else
128         {
129             cant++;
130             mostrarProductoTxt(&prod, stderr);
131         }
132     } while(ingresarProducto(&prod));
133     fprintf(fp, "Se pusieron al final de la lista %d productos.\n", cant);
134     return cant;
135 }
136
137 int _probarSacarNCom(tLista *lista, FILE *fp, int n)
138 {
139     int _probarSacarNCom(tLista* lista, FILE* fp, int n)
140     tProd prod;
141     int cant = 0;
142
143     fprintf(fp, "- Probando sacar del comienzo de la lista %d productos.\n", n);
144     if(sacarPrimeroLista(lista, &prod, sizeof(prod)))
145         mostrarProductoTxt(NULL, fp);
146     do
147     {
148         mostrarProductoTxt(&prod, fp);
149         cant++;
150     } while(--n && sacarPrimeroLista(lista, &prod, sizeof(prod)));
151     if(cant)
152         fprintf(fp,
153             "Se sacaron del comienzo de la lista %d productos.\n",
154             cant);
155     else
156         fprintf(fp, "La lista estaba vacia - no se pudo sacar del comienzo.\n");
157     fprintf(fp, "Lista %squedo vacia.\n", listaVacia(lista) ? "" : "no ");
158     return cant;
159 }
160

```



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

160
161 int _probarSacarNFin(tLista *lista, FILE *fp, int n)
162 {
163     tProd prod;
164     int cant = 0;
165
166     fprintf(fp, "- Probando sacar del final de la lista %d productos.\n", n);
167     if(sacarUltimoLista(lista, &prod, sizeof(prod)))
168         mostrarProductoTxt(NULL, fp);
169     do
170     {
171         mostrarProductoTxt(&prod, fp);
172         cant++;
173     } while(--n && sacarUltimoLista(lista, &prod, sizeof(prod)));
174     if(cant)
175         fprintf(fp,
176             "Se sacaron del final de la lista %d productos.\n",
177             cant);
178     else
179         fprintf(fp, "La lista estaba vacia - no se pudo sacar del final.\n");
180
181     fprintf(fp, "Lista %squedo vacia.\n", listaVacia(lista) ? "" : "no ");
182     return cant;
183 }
184
185
186 int _probarVaciarLista(tLista *lista, FILE *fp)
187 {
188     int cant = 0;
189
190     fprintf(fp, "- Probando vaciar y mostrar lista.\n");
191     if(!listaVacia(lista))
192         mostrarProductoTxt(NULL, fp);
193     cant = vaciarListaYMostrar(lista, mostrarProductoTxtTDA, fp);
194     fprintf(fp, "Se eliminaron y mostraron %d elementos de la lista.\n", cant);
195     return cant;
196 }
197
198
199
200
201

```

... y la invocación de lo anterior desde la función probarPonerYSacarDeLista ...

```

259 }
260
261
262 void probarPonerYSacarDeLista(void)
263 {
264     tLista lista;
265     int veces;
266     FILE *fp;
267     char nomArch[] = { "salida" };
268
269     crearLista(&lista);
270     /* se prueba con todos los productos

```





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

```

270  /* se prueba con todos los productos
271      poner al comienzo, luego
272      poner al final
273      sacar el primero 3 veces
274      sacar el ultimo 4 veces
275      ver el primero
276      ver el último
277      vaciar la lista
278  Esta secuencia:
279      la primera vez mostrando por pantalla y
280      la siguiente vez generando un archivo de texto */
281  veces = 2;
282  fp = stdout;
283  while(veces--)
284  {
285      int    cantCom,
286            cantFin,
287            cantElimCom,
288            cantElimFin,
289            cantElimTot;
290
291      cantCom = _probarPonerAlComienzo(&lista, fp);
292      printf("***Se pusieron %d elementos al comienzo***\n\n", cantCom);
293      cantFin = _probarPonerAlFinal(&lista, fp);
294      printf("***Se pusieron %d elementos al final***\n\n", cantFin);
295      cantElimCom = _probarSacarNCom(&lista, fp, 3);
296      printf("***Se sacaron %d elementos del comienzo***\n\n", cantElimCom);
297      cantElimFin = _probarSacarNFin(&lista, fp, 2);
298      printf("***Se sacaron %d elementos del final***\n\n", cantElimFin);
299      cantElimTot = _probarVaciarLista(&lista, fp);
300      printf("***Se eliminaron %d elementos al vaciar la lista***\n\n",
301             cantElimTot);
302      if(!_abrirArchivo(&fp, nomArch, "wt", CON_MSJ))
303          fp = stderr;
304  }
305  _verYCerrarArchivo(fp, nomArch);
306  if(!listaVacia(&lista))
307  {
308      printf("***ERROR - la lista deberia estar vacia***\n"
309             "    Se procede a vaciarla.\n");
310      vaciarLista(&lista);
311  }
312  }
313
314
315
316
317

```

La siguiente es la "salida" por archivo de texto de la prueba anterior, en la que haciendo uso de poner al comienzo de la lista todos los productos, luego ponerlos al final, luego sacar el primero (tres veces seguidas), luego sacar dos del final, para finalmente vaciar la lista mostrando los elementos eliminados.

Esta secuencia se repite dos veces, la primera por pantalla (fp inicializado con stdout).



Al final del ciclo se abre el archivo (si no se lo pudiera abrir la salida será por stderr -la pantalla-).

Al salir del ciclo se verifica si la lista no quedó vacía.

```

salida - Notepad
File Edit Format View Help
Cod. Produ Descripción del producto ...
Se pusieron al comienzo de la lista 6 productos.
- Probando poner al final.
Cod. Produ Descripción del producto ...
Se pusieron al final de la lista 6 productos.
- Probando sacar del comienzo de la lista 3 productos.
Cod. Produ Descripción del producto ...
limagoma17 Lima de goma de 17 pulgadas ...
plom-telgo Plomada de poliestireno expandido ...
rem-vid15 Remache de vidrio de 1,5 milímetros ...
Se sacaron del comienzo de la lista 3 productos.
Lista no quedo vacia.
- Probando sacar del final de la lista 2 productos.
Cod. Produ Descripción del producto ...
limagoma17 Lima de goma de 17 pulgadas ...
plom-telgo Plomada de poliestireno expandido ...
Se sacaron del final de la lista 2 productos.
Lista no quedo vacia.
- Probando vaciar y mostrar lista.
Cod. Produ Descripción del producto ...
alamyeso1 Alambre de yeso de un milímetro de espesor ...
martillo3K Martillo bolita con saca clavos de 3 kilos ...
clavoro3/4 Clavo de oro 24 kilates de 3/4 de pulgada ...
clavoro3/4 Clavo de oro 24 kilates de 3/4 de pulgada ...
martillo3K Martillo bolita con saca clavos de 3 kilos ...
alamyeso1 Alambre de yeso de un milímetro de espesor ...
rem-vid15 Remache de vidrio de 1,5 milímetros ...
Se eliminaron y mostraron 7 elementos de la lista.

Se ha seleccionado todo para que se aprecie mejor la "salida" por archivo de texto
Ln 30, Col 1 100% Windows (CRLF) UTF-8

```

Pero, ¿qué tal si hay que poner los elementos en forma ordenada? . . .

Para atacar este problema hay que (tal vez) razonar sobre que ya hay elementos ordenados en la lista, entonces lo que se debe hacer es buscar el lugar en que se va a insertar el nuevo elemento.

Esto se traduce en:

**mientras hay-nodo (1)**  
**avanzar por la lista**



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

**fin-mientras**

Al terminar de buscar el lugar de inserción, se pudo haber llegado al final de la lista, con lo que habrá que ponerlo ahí. Lo que puede generar algún problema es que no se haya terminado la lista.

Así que una vez encontrado el lugar de inserción,

```

si hay nodo y la comparación da que son iguales      (2)
    si hay que acumular                                (3)
        si falla acumular                              (4)
            es un error                                (5)
        si-no
            es una coincidencia                        (6)
    si-no
        es una coincidencia                            (6)
    fin-si
fin-si                                             (y no sigue adelante)
```

Acá hay que prestar atención a qué significa "*acumular*".

- Si se trata por ejemplo de productos o movimientos de cuentas bancarias, tal vez signifique dejar la mayor (o menor) fecha, dejar el precio más alto o acumular el saldo, acumular la cantidad de productos o incrementar la cantidad de movimientos, etcétera. O tal vez hay que reemplazar uno por otro, O tal vez no hay que "*acumular*", porque el hecho de darse que hay dos era algo que no debía suceder.

- Si se trata de líneas de texto, se pueden dar los casos anteriores, y en el peor caso, se modifique el tamaño de la línea de texto (haciéndose mayor). En algún uso que se le de a la primitiva, "*acumular*" puede significar concatenar ambas líneas de texto.

Es por ese motivo que "*acumular*" deberá poder modificar el elemento en la lista. Y por las líneas de texto puede fallar o ser exitosa, con lo que se produce una situación distinta del de haber encontrado una coincidencia.

Haber encontrado una coincidencia no necesariamente es un error, puede significar simplemente que no se agrega un nuevo elemento a la lista.

En el caso que no haya coincidencia:



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

- puede suceder una falla catastrófica, que no haya lugar en la lista, y en este caso se lo identifica como tal.
- que haya lugar en la lista con lo que el nuevo elemento puede quedar al comienzo de la lista (estuviera o no vacía), al final o intercalado en su posición.

```

main.c x main.h x productos.c x productos.h x lineasTexto.c x lineasTexto.h x lista.c x lista.h x
203 int ponerEnOrden(tLista *p, const void *d, unsigned cantBytes,
204                  int (*comparar)(const void *, const void *),
205                  int (*acumular)(void **, unsigned *, const void *, unsigned))
206 {
207     tNodo *nue;
208
209     while(*p && comparar((*p)->info, d) < 0)                <-- (1)
210         p = &(*p)->sig;
211     if(*p && comparar((*p)->info, d) == 0)                  <-- (2)
212     {
213         if(acumular)                                        <-- (3)
214             if(!acumular(&(*p)->info, &(*p)->tamInfo, d, cantBytes)) <-- (4)
215                 return SIN_MEM;                            <-- (5)
216         return CLA_DUP;                                     <-- (6)
217     }
218     if((nue = (tNodo *)malloc(sizeof(tNodo))) == NULL ||    <-- desde acá, ¡ya es
219         (nue->info = malloc(cantBytes)) == NULL)            "terreno" conocido!
220     {
221         free(nue);
222         return SIN_MEM;
223     }
224     memcpy(nue->info, d, cantBytes);
225     nue->tamInfo = cantBytes;
226     nue->sig = *p;
227     *p = nue;
228     return TODO_BIEN;
229 }
230

```

- (1) el ciclo avanza por la lista buscando el lugar de inserción.
- (2) cuando sale del ciclo, si hay nodo y la comparación da que coinciden, procede a acumular (o no hace nada si para la función de acumular se invoca a la primitiva con **NULL**). Hay que tener en cuenta que si acumular significa concatenar líneas de texto, si la acumulación falla es porque no hay memoria para las líneas concatenadas (o si hay que reemplazar una línea de texto por una más grande y no hay memoria). Este es el único motivo por el cual puede fallar la acumulación.

Finalmente el paso conocido, el de obtener memoria, etcétera.



UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

Esta primitiva permitirá generar la lista de menor a mayor si el criterio de comparación de la función `comparar` sigue el lineamiento tradicional (del modo que lo hace `strcmp`), devolviendo:

- algún valor negativo si su primer argumento es menor que el segundo
- algún valor positivo si su primer argumento es mayor que el segundo
- cero si son iguales

Pero si el criterio de comparación es para generar la lista de mayor a menor, devolverá

- menor que cero cuando el primero sea mayor
- mayor que cero cuando el primero sea menor
- cero cuando coinciden.

Además permitirá poner el nuevo elemento que coincide con el que ya estaba o antes o después de este. Esto lo da el criterio de la función de comparación que se emplea.

Pero, ¿qué tal si hay que ordenar una lista que ya está cargada?

```
void ordenar(tLista *p, int (*comparar)(const void *, const void *))
{
    tLista *pri = p; <-- pri siempre mantendrá la dirección del lro de la lista

    if(*p == NULL) <-- si no hay lista, no hay nada que hacer
        return;
    while((*p)->sig) <-- mientras haya siguiente ...
    {
        if(comparar((*p)->info, (*p)->sig->info) > 0) <-- si el actual es mayor
            que su siguiente
        {
            tLista *q = pri; <-- prepara comenzar desde el inicio de la lista
            tNodo *aux = (*p)->sig; <-- toma la dirección del nodo

            (*p)->sig = aux->sig; <-- lo desvincula de la lista
            while(comparar((*q)->info, aux->info) > 0) <-- avanza buscando el
                q = &(*q)->sig; lugar de inserción
            aux->sig = *q;
            *q = aux; <-- vincula el nodo en la lista
        }
        else
            p = &(*p)->sig; <-- avanza si no tuvo que hacer lo anterior
    }
}
```



Elija algún método de ordenamiento: burbuja, inserción, selección, etc. Cualquiera de ellos podrá ser resuelto. En el caso de burbuja, se podrá intercambiar el elemento entre los nodos (los punteros **info** de los nodos además del miembro **tamInfo** de los mismos). En la figura anterior se muestra uno de los métodos en que no es posible el intercambio de elementos. Consiste en que para cada nodo, si a continuación del mismo hay un elemento que debe ir antes, se lo '*desvincula*' de la lista y se busca su lugar de inserción desde el comienzo de la lista.

Atención: una primitiva que se *precie de serlo*, no utiliza otras primitivas.

Otras operaciones sobre una lista pueden ser las de eliminar los nodos con elementos cuyas claves coinciden (dos alternativas)

- se acumule en un nodo
- no queden ninguno

Cada una de estas alternativas en dos casos

- lista ordenada
- lista no ordenada

Y cada una de estas cuatro alternativas almacenando en archivo (binario / de texto / mostrando por pantalla) los elementos eliminados / acumulados.

Eliminar los que son únicos en lista ordenada / no ordenada además (o no) de *mostrar*.

Insertar al final, salvo que encuentre otro con la misma clave y acumule o inserte antes o inserte a continuación del que ya estaba.

Y cualquier otra operación que se requiera para un modelo computacional.

#### Notas adicionales sobre la función que permite acumular.

Debido al caso en que los '*elementos*' almacenados en una lista sean líneas de texto, al necesitar '*acumular*' en ella (en caso de clave duplicada) haya que concatenar la que ya estaba con la nueva o reemplazarla con una de distinto tamaño, es por eso que en las





UNLaM

Dto. Ingeniería e Investigaciones Tecnológicas

---

primitivas que acumulan, en la función **acumular** que reciben por argumento responden a la declaración (o prototipo):

```
int (*acumular)(void **, unsigned *, const void *, unsigned)
```

los dos primeros argumentos permiten modificar al '*elemento*' de la lista (caracterizado por el *elemento* en sí mismo (miembro **info**) y la cantidad de bytes que ocupa (miembro **tamInfo**). Esto es lo indicado para las primitivas de insertar en orden, eliminar duplicados acumulando, buscar y reemplazar, y cualquier otra operación (primitiva) sobre el TDA LISTA que tenga tal objetivo.

Planteo: poner líneas de texto ordenadas en forma descendente por cantidad de palabras, a igual cantidad de palabras que la línea de texto a poner debe quedar antes que la que ya estaba.

Para este caso, no habrá acumulación.

El criterio de comparación será acorde a lo pedido (no es una comparación 'habitual').

```

1  /* -----o--x--o-----
2  *          lista.h      declaración y primitivas del TDA LISTA
3  *                      implementada en lista dinámica simplemente enlazada
4  * -----o--x--o----- */
5
6  #ifndef LISTA_H_
7  #define LISTA_H_
8
9
10 #include <stdlib.h>
11 #include <string.h>
12 #include <stdio.h>
13
14
15 #define TODO_BIEN      0
16 #define SIN_MEM        1
17 #define CLA_DUP        2
18
19
20 typedef struct sNodo
21 {
22     void          *info;
23     unsigned      tamInfo;
24     struct sNodo  *sig;
25 } tNodo;
26 typedef tNodo *tLista;
27
28
29 void crearLista(tLista *p);
30
31 int  listaVacia(const tLista *p);
32
33 int  listaLlena(const tLista *p, unsigned cantBytes);
34
35 int  vaciarLista(tLista *p);
36
37 int  vaciarListaYMostrar(tLista *p,
38                          void (*mostrar)(const void *, FILE *), FILE *fp);
39
40 int  ponerAlComienzo(tLista *p, const void *d, unsigned cantBytes);
41
42 int  sacarPrimeroLista(tLista *p, void *d, unsigned cantBytes);
43
44 int  verPrimeroLista(const tLista *p, void *d, unsigned cantBytes);
45
46 int  ponerAlFinal(tLista *p, const void *d, unsigned cantBytes);
47
48 int  sacarUltimoLista(tLista *p, void *d, unsigned cantBytes);
49
50 int  verUltimoLista(const tLista *p, void *d, unsigned cantBytes);
51
52 int  mostrarLista(const tLista *p,
53                  void (*mostrar)(const void *, FILE *), FILE *fp);
54
55 int  mostrarListaAlReves(const tLista *p,
56                          void (*mostrar)(const void *, FILE *), FILE *fp);
57
58 int  mostrarListaAlRevesYVaciar(tLista *p,
59                                void (*mostrar)(const void *, FILE *),
60                                FILE *fp);
61
62 int  ponerEnOrden(tLista *p, const void *d, unsigned cantBytes,
63                   int (*comparar)(const void *, const void *),
64                   int (*acumular)(void **, unsigned *, const void *, unsigned));
65
66 void ordenar(tLista *p, int (*comparar)(const void *, const void *));

```

```

67
68 #endif // LISTA_H_
69
70 /* -----o--x--o-----
71 *          lista.c      definición y primitivas del TDA LISTA
72 *                      implementada en lista dinámica simplemente enlazada
73 * -----o--x--o----- */
74
75 #include "lista.h"
76
77
78 #define minimo( X , Y )      ( ( X ) <= ( Y ) ? ( X ) : ( Y ) )
79
80
81 void crearLista(tLista *p)
82 {
83     *p = NULL;
84 }
85
86
87 int listaVacía(const tLista *p)
88 {
89     return *p == NULL;
90 }
91
92
93 int listaLlena(const tLista *p, unsigned cantBytes)
94 {
95     tNodo *aux = (tNodo *)malloc(sizeof(tNodo));
96     void *info = malloc(cantBytes);
97
98     free(aux);
99     free(info);
100     return aux == NULL || info == NULL;
101 }
102
103
104 int vaciarLista(tLista *p)
105 {
106     int cant = 0;
107     while(*p)
108     {
109         tNodo *aux = *p;
110
111         cant++;
112         *p = aux->sig;
113         free(aux->info);
114         free(aux);
115     }
116     return cant;
117 }
118
119
120 int vaciarListaYMostrar(tLista *p,
121                        void (*mostrar)(const void *, FILE *), FILE *fp)
122 {
123     int cant = 0;
124     while(*p)
125     {
126         tNodo *aux = *p;
127
128         cant++;
129         *p = aux->sig;
130         if(mostrar && fp)
131             mostrar(aux->info, fp);
132         free(aux->info);

```

```

133         free(aux);
134     }
135     return cant;
136 }
137
138
139 int ponerAlComienzo(tLista *p, const void *d, unsigned cantBytes)
140 {
141     tNodo *nue;
142
143     if((nue = (tNodo *)malloc(sizeof(tNodo))) == NULL ||
144        (nue->info = malloc(cantBytes)) == NULL)
145     {
146         free(nue);
147         return 0;
148     }
149     memcpy(nue->info, d, cantBytes);
150     nue->tamInfo = cantBytes;
151     nue->sig = *p;
152     *p = nue;
153     return 1;
154 }
155
156
157 int sacarPrimeroLista(tLista *p, void *d, unsigned cantBytes)
158 {
159     tNodo *aux = *p;
160
161     if(aux == NULL)
162         return 0;
163     *p = aux->sig;
164     memcpy(d, aux->info, minimo(cantBytes, aux->tamInfo));
165     free(aux->info);
166     free(aux);
167     return 1;
168 }
169
170
171 int verPrimeroLista(const tLista *p, void *d, unsigned cantBytes)
172 {
173     if(*p == NULL)
174         return 0;
175     memcpy(d, (*p)->info, minimo(cantBytes, (*p)->tamInfo));
176     return 1;
177 }
178
179
180 int ponerAlFinal(tLista *p, const void *d, unsigned cantBytes)
181 {
182     tNodo *nue;
183
184     while(*p)
185         p = &(*p)->sig;
186     if((nue = (tNodo *)malloc(sizeof(tNodo))) == NULL ||
187        (nue->info = malloc(cantBytes)) == NULL)
188     {
189         free(nue);
190         return 0;
191     }
192     memcpy(nue->info, d, cantBytes);
193     nue->tamInfo = cantBytes;
194     nue->sig = NULL;
195     *p = nue;
196     return 1;
197 }
198

```

```

199
200 int  sacarUltimoLista(tLista *p, void *d, unsigned cantBytes)
201 {
202     if(*p == NULL)
203         return 0;
204     while((*p)->sig)
205         p = &(*p)->sig;
206     memcpy(d, (*p)->info, minimo(cantBytes, (*p)->tamInfo));
207     free((*p)->info);
208     free(*p);
209     *p = NULL;
210     return 1;
211 }
212
213
214 int  verUltimoLista(const tLista *p, void *d, unsigned cantBytes)
215 {
216     if(*p == NULL)
217         return 0;
218     while((*p)->sig)
219         p = &(*p)->sig;
220     memcpy(d, (*p)->info, minimo(cantBytes, (*p)->tamInfo));
221     return 1;
222 }
223
224
225 int  mostrarLista(const tLista *p,
226                 void (*mostrar)(const void *, FILE *), FILE *fp)
227 {
228     int    cant = 0;
229
230     while(*p)
231     {
232         mostrar((*p)->info, fp);
233         p = &(*p)->sig;
234         cant++;
235     }
236     return cant;
237 }
238
239
240 int  mostrarListaAlReves(const tLista *p,
241                        void (*mostrar)(const void *, FILE *), FILE *fp)
242 {
243     if(*p)
244     {
245         int n = mostrarListaAlReves(&(*p)->sig, mostrar, fp);
246
247         mostrar((*p)->info, fp);
248         return n + 1;
249     }
250     return 0;
251 }
252
253
254 int  mostrarListaAlRevesYVaciar(tLista *p,
255                               void (*mostrar)(const void *, FILE *),
256                               FILE *fp)
257 {
258     if(*p)
259     {
260         int n = mostrarListaAlReves(&(*p)->sig, mostrar, fp);
261
262         mostrar((*p)->info, fp);
263         free((*p)->info);
264         free(*p);

```

```

265     *p = NULL;
266     return n + 1;
267 }
268 return 0;
269 }
270
271
272 int ponerEnOrden(tLista *p, const void *d, unsigned cantBytes,
273                 int (*comparar)(const void *, const void *),
274                 int (*acumular)(void **, unsigned *, const void *, unsigned))
275 {
276     tNodo *nue;
277
278     while(*p && comparar((*p)->info, d) < 0)
279         p = &(*p)->sig;
280     if(*p && comparar((*p)->info, d) == 0)
281     {
282         if(acumular)
283             if(!acumular(&(*p)->info, &(*p)->tamInfo, d, cantBytes))
284                 return SIN_MEM;
285         return CLA_DUP;
286     }
287     if((nue = (tNodo *)malloc(sizeof(tNodo))) == NULL ||
288        (nue->info = malloc(cantBytes)) == NULL)
289     {
290         free(nue);
291         return SIN_MEM;
292     }
293     memcpy(nue->info, d, cantBytes);
294     nue->tamInfo = cantBytes;
295     nue->sig = *p;
296     *p = nue;
297     return TODO_BIEN;
298 }
299
300
301 void ordenar(tLista *p, int (*comparar)(const void *, const void *))
302 {
303     tLista *pri = p;
304
305     if(*p == NULL)
306         return;
307     while((*p)->sig)
308     {
309         if(comparar((*p)->info, (*p)->sig->info) > 0)
310         {
311             tLista *q = pri;
312             tNodo *aux = (*p)->sig;
313
314             (*p)->sig = aux->sig;
315             while(comparar((*q)->info, aux->info) > 0)
316                 q = &(*q)->sig;
317             aux->sig = *q;
318             *q = aux;
319         }
320         else
321             p = &(*p)->sig;
322     }
323 }
324
325
326 /* -----o--x--o-----
327 *          productos.h      ingreso sintético de productos
328 * -----o--x--o----- */
329
330 #ifndef PRODUCTOS_H_

```



```

331 #define PRODUCTOS_H_
332
333 #include <stdio.h>
334
335
336 typedef struct
337 {
338     char    codProd[11],
339           descrip[46];
340 } tProd;
341
342
343 int ingresarProducto(tProd *d);
344
345 void mostrarProducto(const tProd *d);
346
347 void mostrarProductoTxt(const tProd *d, FILE *fp);
348
349 void mostrarProductoTxtTDA(const void *d, FILE *fp);
350
351 void grabarProductoBinTDA(const void *d, FILE *fp);
352
353
354 #endif // PRODUCTOS_H_
355 /* -----o--x--o-----
356 *          productos.c      ingreso sintético de productos
357 * -----o--x--o----- */
358
359 #include "productos.h"
360
361 int ingresarProducto(tProd *d)
362 {
363     static const tProd productos[] = {
364         ///1234567890      123456789 123456789 123456789 12345
365         { "clavoro3/4", "Clavo de oro 24 kilates de 3/4 de pulgada" },
366         { "martillo3K", "Martillo bolita con saca clavos de 3 kilos" },
367         { "alamyeso1", "Alambre de yeso de un milimetro de espesor" },
368         { "rem-vid15", "Remache de vidrio de 1,5 milímetros" },
369         { "plom-telgo", "Plomada de poliestireno expandido" },
370         { "limagoma17", "Lima de goma de 17 pulgadas" } };
371     static int posi = 0;
372
373     if(posi == sizeof(productos) / sizeof(tProd))
374     {
375         posi = 0;
376         return 0;
377     }
378     *d = productos[posi];
379     posi++;
380
381     return 1;
382 }
383
384 //void mostrarProducto(const tProd *d)
385 void mostrarProductoTxt(const tProd *d, FILE *fp)
386 {
387     if(d)
388     //      fprintf(stdout,
389     //      fprintf(fp,
390     //      "%-*s %-*s ...\n",
391     //      (int) sizeof(d->codProd) - 1, d->codProd,
392     //      (int) sizeof(d->descrip) - 1, d->descrip);
393     else
394     //      fprintf(stdout,
395     //      fprintf(fp,
396     //      "%-*.*s %-*.*s ...\n",

```

```

397         (int)sizeof(d->codProd) - 1, (int)sizeof(d->codProd) - 1,
398         "Cod. Producto",
399         (int)sizeof(d->descrip) - 1, (int)sizeof(d->descrip) - 1,
400         "Descripcion del producto");
401     }
402
403
404     void mostrarProducto(const tProd *d)
405     {
406         mostrarProductoTxt(d, stdout);
407     }
408
409
410     void mostrarProductoTxtTDA(const void *d, FILE *fp)
411     {
412         mostrarProductoTxt((const tProd *)d, fp);
413     }
414
415
416     void grabarProductoBinTDA(const void *d, FILE *fp)
417     {
418         fwrite(d, sizeof(tProd), 1, fp);
419     }
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443     /* -----o--x--o-----
444     *          main.h      prueba del TDA LISTA con asignación dinámica de memoria
445     * -----o--x--o----- */
446
447     #ifndef MAIN_H_
448     #define MAIN_H_
449
450     #include <stdio.h>
451     #include <string.h>
452
453
454     #include "../productos/productos.h"
455     #include "../lineasDeTexto/lineasTexto.h"
456     #include "../lista/lista.h"
457
458
459     void probarIngresarYMostrarProd(void);
460
461     void probarIngresarYMostrarTexto(void);
462

```

```
463 void probarPonerYSacarDeLista(void);
464
465
466 #endif // MAIN_H_
467
468 /* -----o--x--o-----
469 *          main.h      prueba del TDA LISTA con asignación dinámica de memoria
470 * -----o--x--o----- */
471
472 #ifndef MAIN_H_
473 #define MAIN_H_
474
475 #include <stdio.h>
476 #include <string.h>
477
478
479 #include "../productos/productos.h"
480 #include "../lineasDeTexto/lineasTexto.h"
481 #include "../lista/lista.h"
482
483
484 void probarIngresarYMostrarProd(void);
485
486 void probarIngresarYMostrarTexto(void);
487
488 void probarPonerYSacarDeLista(void);
489
490
491 #endif // MAIN_H_
492
493
```