Reporte 04: Ordenamientos

Aclaraciones

En este programa tengo los 4 ordenamientos que se piden en las especificaciones de la tarea, más aparte dos métodos burbuja que ya tenía programados que usé como referencia para saber cómo codificar el resto del programa.

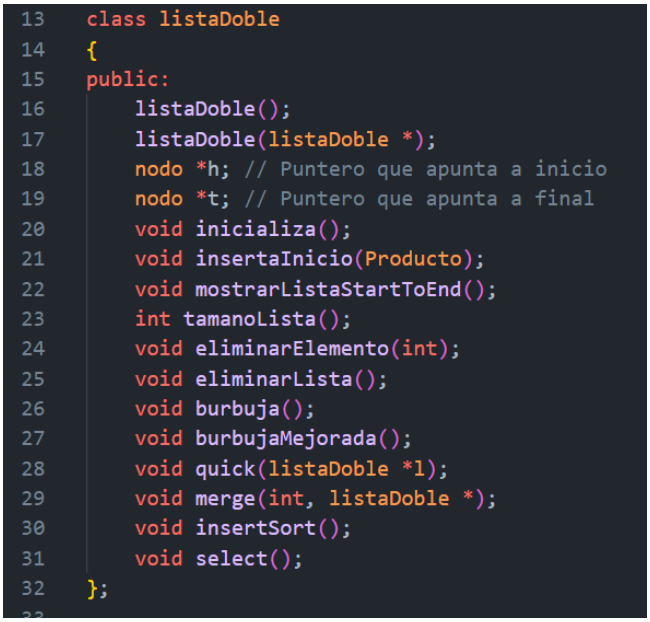
Algo más que me gustaría especificar es que si en el menú principal se escribe el numero “69” en vez del resto de opciones, aunque no aparezca en el menú, se agregarán varios productos con precios desordenados, para que poder usar el programa rápidamente sin necesidad de agregar cada producto y precio manualmente.

Menu para lista.cpp

Este archivo solo es el menú principal, con el switch para mandar a cada caso según lo quiera el usuario, y llamar a los métodos de la lista, los cuales veremos en el siguiente archivo.

ListaDoble.hpp

Esta cabecera es la más extensa y tiene casi todo el código importante del programa, si iniciamos leyendo el código desde el inicio, vemos que en la línea 9 está un contador para asignarle ID al producto que vaya agregando el usuario, y un poco más adelante está la declaración de la clase **lista doble**, la cual no tiene todas las operaciones de una lista doble pues en este caso ese no es el objetivo, si no que la mayoría de los métodos de la clase están enfocados al ordenamiento de los precios de los productos, como se ve en la imagen.



```
13  class listaDoble
14  {
15  public:
16      listaDoble();
17      listaDoble(listaDoble *);
18      nodo *h; // Puntero que apunta a inicio
19      nodo *t; // Puntero que apunta a final
20      void inicializa();
21      void insertaInicio(Producto);
22      void mostrarListaStartToEnd();
23      int tamanoLista();
24      void eliminarElemento(int);
25      void eliminarLista();
26      void burbuja();
27      void burbujaMejorada();
28      void quick(listaDoble *l);
29      void merge(int, listaDoble *);
30      void insertSort();
31      void select();
32  };
```

Ya que el propósito del programa son los ordenamientos, no me pararé a explicar las funciones que no son ordenamientos, así que solo mencionaré lo importante de los siguientes 4, quick, merge, insert y select.

Quick

Tanto quick como merge usan recursividad, es por eso que podemos ver en la declaración del método que reciben la lista misma, a diferencia del insert o el select los cuales no usan recursividad.

El ordenamiento quick, como su nombre lo dice, es de los más rápidos que hay en el programa, sin embargo, según qué caso también es posible que sea el más costoso en cuanto a recursos, por la posibilidad de que se use tanta recursividad.

```
void listaDoble::quick(listaDoble *l)
{
    nodo *piv;
    nodo *pivGuardar;
    nodo *aux;

    listaDoble *mayores = new listaDoble;
    listaDoble *menores = new listaDoble;
```

Al inicio del método podemos ver las variables que se declaran, el pivote, o piv, es de las más importantes, pues el ordenamiento quick lo que hace es ordenar los pivotes, sabemos que el programa agarra un pivote y todos los números menores o iguales a este los pone en una sublista, y los mayores o iguales en otra sublista, las cuales se puede ver que declaramos en seguida de las variables, y después de acomodar cada dato en la lista que le corresponde, volvemos a llamar a la función, aplicando la recursividad previamente mencionada, pero ahora la lista completa que vamos a dividir en menores y mayores, son ambas sublistas previamente divididas, y así seguiremos haciendo hasta que quede la unidad, donde se rompería un nivel de recursividad.

```
247             else
248             {
249                 mayores->insertaInicio(aux->dato);
250             }
251             aux = aux->siguiente;
252         }
253
254         quick(menores);
255         quick(mayores);
```

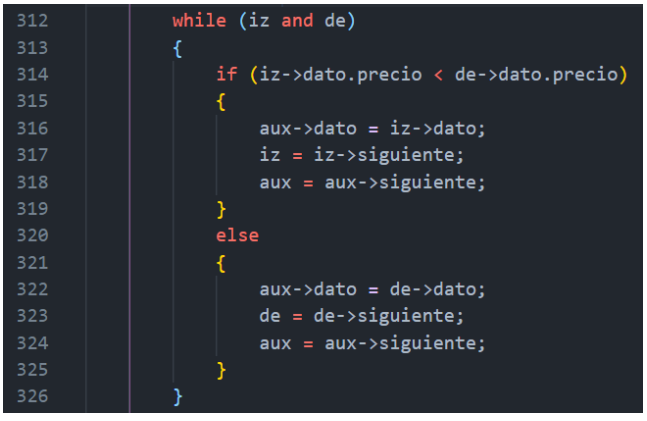
Merge

El merge también usa recursividad, por lo que también ganamos eficiencia pero sacrificando recursos de la maquina, en este caso dividimos la lista principal en 2, o si no es posible ya que no es numero par, en mitad y mitad -1.

```
280     listaDoble *derecha = new listaDoble;
281     listaDoble *izquierda = new listaDoble;
282     nodo *aux = l->h;
283     int med = tam / 2;
284     int dif = tam % 2;
285     int i, j;
286     if (l->tamanoLista() > 1)
287     {
288         i = 0;
289         while (aux and i < med)
290         {
291             izquierda->insertaInicio(aux->dato);
292             i++;
293             aux = aux->siguiente;
294         }
295 
296         j = 0;
297         while (aux and j < med + dif)
298         {
299             derecha->insertaInicio(aux->dato);
300             j++;
301             aux = aux->siguiente;
302         }
```

Una vez tenemos las dos sublistas creadas, y los auxiliares necesarios, pasamos a dividir la lista a la mitad, para esto hacemos uso de 2 indices para simular un arreglo, i y j, cuando ya hemos dividido la lista a la mitad, agarramos una de estas mitades y volvemos a llamar a la función como si esta mitad fuera la lista completa, y la volvemos a dividir, es decir, aquí es cuando la recursividad entra en juego.

Una vez tengamos divididas las listas hasta la unidad, vamos a hacer el método para combinar ambas listas de manera ordenada, en donde simplemente comparamos el precio de ambos, y ponemos primero el que tenga el precio menor.



```
312         while (iz and de)
313         {
314             if (iz->dato.precio < de->dato.precio)
315             {
316                 aux->dato = iz->dato;
317                 iz = iz->siguiente;
318                 aux = aux->siguiente;
319             }
320             else
321             {
322                 aux->dato = de->dato;
323                 de = de->siguiente;
324                 aux = aux->siguiente;
325             }
326         }
```

Así vamos haciendo con cada lista que cada vez se vuelve más grande según se vayan acabando los niveles de recursividad que se hayan llamado en el programa, hasta que la lista principal queda completamente ordenada.

Insert y select

Ya que son métodos muy sencillos y parecidos, los voy a explicar a groso modo en un solo apartado, ya que como no usan recursividad, el proceso de entenderlos es un poco más sencillo y lineal.

El select sort lo que hace es seleccionar el elemento menor, es decir, recorre toda la lista para encontrar cual es el menor que hay, y ese lo mueve al principio de la lista, y de nuevo, recorre toda la lista en busca del menor **excepto** por el elemento previamente movido, pues ya sabemos que es el menor, así que no tiene caso revisarlo, cuando ya encuentra el menor en el resto de la lista, lo mueve al lado de el previamente acomodado, y comienza de nuevo a recorrer toda la lista, hasta que todos los elementos estén ordenados.

El insert, podríamos decir que es una combinación del burbuja y del select, pues hace un cambio entre dos números adyacentes, como el burbuja, sin embargo, al detectar que hubo un cambio se regresa una posición, para ver si no tiene que mover el numero aún más, y si sí, este proceso se repite, cambia y se regresa, hasta que el numero está en su posición.

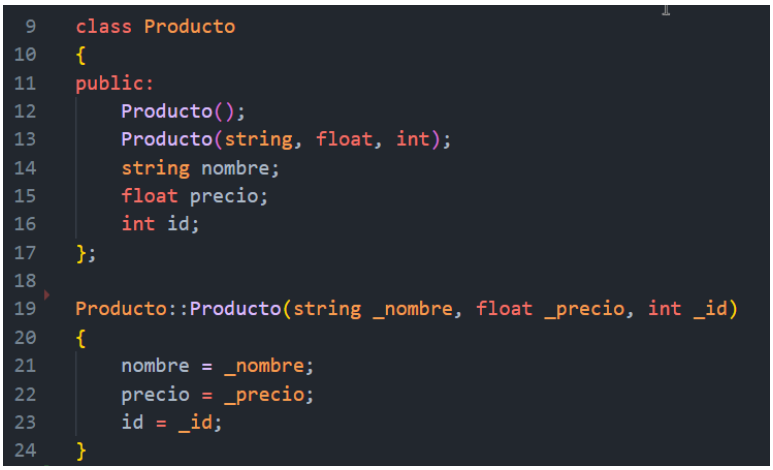
nodo.hpp y producto.hpp

Por ultimo están los archivos para la clase nodo y la clase producto, los cuales no tienen mayor complejidad pero igual creí oportuno repasarlos.

```
 7  class nodo
 8  {
 9  public:
10      Producto dato;
11      nodo *siguiente;
12      nodo *anterior;
13      nodo();
14      nodo(Producto e)
15      {
16          dato = e;
17          siguiente = nullptr;
18          anterior = nullptr;
19      };
20  };
```

La clase nodo es igual que cualquier clase nodo para una lista doblemente ligada, con ambos punteros y el dato que va a guardar, en este caso, un dato de tipo producto, además de un par de constructores para que se puedan utilizar cuando se cree un nuevo nodo.

Dentro de la clase producto simplemente tenemos los datos del producto que ingresa el usuario cuando crea un nuevo producto, que es el nombre de este, de tipo string, el precio que está en tipo float para que pueda aceptar decimales también, y la id que



```
class Producto
{
public:
    Producto();
    Producto(string, float, int);
    string nombre;
    float precio;
    int id;
};

Producto::Producto(string _nombre, float _precio, int _id)
{
    nombre = _nombre;
    precio = _precio;
    id = _id;
}
```

está en tipo int, y como se mencionó al principio del archivo, esta id no la escribe el usuario, si no que se genera automáticamente por el programa para evitar que se ingresen ids repetidas.

Conclusiones

Este programa se me hizo muy interesante por algo que el profesor mencionó en una clase, y es que no hay un algoritmo de ordenamiento perfecto, si no más bien que se usa el necesario según el caso, pues si los recursos no son un problema y solo se busca la velocidad podemos usar un algoritmo con recursividad, pero si tenemos un caso en el que no podemos malgastar recursos y debemos de usar el mínimo de recursos posibles, sin importar la velocidad del algoritmo, debemos considerar usar otro algoritmo.

Aunque se me dificulta aprenderme de memoria cada paso de cada algoritmo posible, tampoco es algo que me preocupe demasiado, pues más que saberse de memoria el algoritmo creo que es más importante entender qué hace y el por qué, para incluso poder hacer alteraciones a estos y adaptarlos a cada situación, y aunque el algoritmo que escribamos nosotros quizá ni siquiera tenga nombre, que eso no signifique que no es el adecuado para la situación en el que lo estamos aplicando.