**Instituto Tecnológico y de Estudios Superiores de Monterrey.**

**PROGRAMACIÓN DE ESTRUCTURA DE DATOS Y ALGORITMOS FUNDAMENTALES.**

****

**Actividad 2.3: Actividad Integradora .**

## Profesor: *Luis Humberto González Guerra.*

**Alumna*: A00831510*** Andrea Marisol Salcedo Vázquez

*Grupo 1*

*10/10/2021*

**Reflexión personal**

Andrea Marisol Salcedo Vázquez

Para realizar esta actividad se emplearon las listas encadenadas y al final, para hacer el conteo, usamos vectores. Lo primero que hicimos fue pedirle al usuario el archivo a leer e igual que la vez anterior lo abrimos y guardamos los datos con el struct de Registro, solo que esta vez en lugar de almacenarlo en un vector, guardamos la información de los buques en dos LinkedList, una para los buques del Mar Rojo y otra para los buques del Mediterráneo. Esta parte fue sencilla pues ya contábamos con las funciones addFirst y addLast en nuestra clase para agregar nuevos nodos a nuestra lista encadenada, decidimos usar addFirst puesto que cuenta con una complejidad O(1) mientras que addLast cuenta con complejidad O(n) por lo que se decidió usar el de menor complejidad ya que de igual forma no nos lo daban ordenado y debemos realizar una función para ordenarlo por lo que no importaba si lo añadíamos al inicio o al final.

Luego nos piden que ordenemos de forma ascendente nuestras listas encadenadas, para esto elaboramos una función llamada sort la cual usa el método de intercambio, es decir que compara el primer nodo con todos los demás y si encuentra uno menor realiza el cambio, luego se pasa al segundo nodo y realiza lo mismo y así sucesivamente hasta recorrer todos los nodos y haber realizado todos los cambios necesarios. Debido a que estamos trabajando con un struct y no con elementos de tipo entero fue necesario realizar la sobrecarga del operador >. Para esto nos basamos un poco en la función acompare que hicimos la actividad pasada para ordenar el vector, es decir, primero checamos si los ubi de los dos struct a comparar son iguales y si es así regresa la comparación de las fechas, en caso de que no sean iguales directamente compara los ubi con ayuda de la función compare() para comparar strings y regresa la comparación de si es mayor a 0. De esta forma, cuando usáramos el comparador > en nuestro sort de tipo template no habría problemas pues directamente usaría la sobrecarga que hicimos en el struct.

Sin embargo, mientras que la complejidad del sort que usamos en la actividad pasada con los vectores fue de nlogn, el ordenar las LinkedList por el método de intercambio lo volvió de complejidad n^2 por lo que ordenar una LinkedList de forma ascendente o descendente tendrá una complejidad mayor a la de ordenar un vector así que pienso que en este caso si es mejor usar vectores pues el ordenamiento tiene una menor complejidad.

Luego de ordenar las dos listas encadenadas le solicitamos al usuario que ingrese el nombre de los dos archivos con los que quiere guardar los datos de cada una de las LinkedList. Para guardar estas listas encadenadas en archivos creamos una función llamada writeData la cual recibe como argumento el nombre del archivo y recorre la lista guardando la data almacenada en cada nodo en el archivo, pero para realizar esto tuvimos que usar la sobrecarga del operador << en nuestro struct el cual hace que los datos salgan de manera ordenada de la forma:

Ubi fecha hora entrada

Después de guardar los datos en dos archivos le pedimos al usuario el ubi para buscarlo. Mientras que en la actividad anterior lo realizamos con búsqueda binaria la cual tiene complejidad log(n) esta vez no supimos como implementar búsqueda binaria en las LinkedList pues solamente se puede avanzar hacia una dirección así que decidimos usar búsqueda secuencial e implementamos la función find que recibe como parámetro un string y va comparando la data en los nodos con el string hasta encontrar el ubi en la lista encadenada y regresar su posición, en caso de que no la encuentre regresa un -1. Pero para hacer esta función adentro de nuestra de clase de tipo template <class T> necesitamos realizar la sobrecarga del operador == en nuestro struct en el cual regresa el resultado de comparar una string con los primeros 3 caracteres del ubi el cual obtenemos del método getSerie. Debido a que va comparando uno por uno el primero que encuentra es el que tiene que regresar no como en el caso de búsqueda binaria que debía comprobar si sí era el primero. Finalmente, debido a que usamos búsqueda secuencial obtuvimos una complejidad O(n) la cual es mayor que en la búsqueda binaria realizada con vectores.

Para realizar el conteo y la ultima impresión usamos vectores e hicimos un mapeo. En el caso de que el índice retornado con la función descrita anterior fuera -1 en ambas linkedlist entonces se le indica al usuario que no existe ningún registro, pero en caso contrario convertimos las linkedlist a vectores con una función que creamos llamada toVector() el cual recorre la lista poniendo la Data en un vector. Una vez que tenemos los vectores usamos map para poder recorrer ambos vectores si su índice es diferente de -1 empezando en la posición obtenida anteriormente y terminando cuando ya no coincida con el ubi solicitado e ir guardando la cantidad que tenemos de cada mes del registro de ese ubi en nuestro map contador.

Finalmente recorremos el map con un for y vamos imprimiendo:

mes año cantM cantR

Como pudimos observar las funciones que hicimos para ordenar nuestras LinkedList y para encontrar el ubi que el usuario solicitó fueron de una complejidad mayor a las que realizamos en la actividad anterior con vectores por lo que podemos decir que para esos casos es mejor utilizar vectores puesto que tienen complejidad de nlogn para el sort y de logn para la búsqueda, mientras que con LinkedList obtuvimos n^2 y n. Sin embargo, las otras acciones como agregar un elemento a la lista o imprimir las listas en archivos son de la misma complejidad que si lo hacemos con vectores pues en ambos casos tenemos que hacer el corrimiento de todo los datos. El lado bueno de la LinkedList es que podemos implementar más métodos dentro de la clase que podrían sernos útiles para otras cosas. Por lo que el lado malo de las LinkedList es que para acceder a un nodo debes pasar por todos los nodos anteriores a este mientras que en el vector no es necesario pues cuenta con un indice, sin embargo, si hiciéramos una linkedlist bidireccional es decir, que un nodo apunte tanto al siguiente como al anterior para poder ir en ambos sentidos entonces tendríamos más opciones para manejar nuestra LinkedList. No obstante, por la estructura de nuestra lista encadenada puedo decir que para realizar las funciones que hicimos en este programa es mejor usar vectores pues tienen una complejidad menor por el momento pero es bueno conocer las diferentes estructuras de datos porque cada una tiene sus beneficios.