Actividad 2.3: Reflexión individual

Estudiante: Diego Palma Rodríguez **Matrícula:** A01759772

Existe una ecuación famosa dentro de las ciencias de la computación: Estructuras de Datos + Algoritmos = Programas, la cual se basa en uno de los libros de Niklaus Wirth [1]. Esto nos indica que las estructuras de datos son parte del cimiento de cualquier programa de *software*, debido a que nos permiten almacenar variables en memoria, acceder a dichas variables de forma sencilla y realizar operaciones dentro de la misma estructura de datos. Dentro de los diferentes tipos de estructuras tenemos las estructuras de datos lineales, las cuales se caracterizan por organizar los elementos en forma lineal; es decir, cada elemento tiene solo un sucesor y un predecesor, con excepción del primer y último elemento [2]. Como ejemplos tenemos al *Array, Stack, Queue, LinkedList* y *Double LinkedList.*

Si bien tenemos a disposición diversos tipos de estructuras de datos lineales, también existen criterios para seleccionar la estructura de datos adecuada en base al problema que se quiera resolver o en la aplicación que se requiera. Uno de estos criterios es la alocación de memoria, ya sea estática o dinámica. Las estructuras de datos estáticas, como los *arrays*, tienen un tamaño fijo y permiten un acceso rápido a los elementos, pero operaciones como las de inserción y eliminación suelen ser computacionalmente costosas; mientras que en el almacenamiento dinámico, como los *LinkedList* y *Double LinkedList*, el acceso a los elementos suele ser lento, pero poseen un tamaño flexible y las operaciones anteriores se realizan de forma más rápida [2].

En esta aplicación en particular se utilizó una *Double LinkedList* sobre una *LinkedList.* Esto debido a que existen operaciones en las cuales resulta más conveniente utilizarlo. Por ejemplo, se implementó una función de ordenamiento con el algoritmo *quicksort* iterativo, y para ello se debía acceder tanto al nodo sucesor como al nodo predecesor, lo cual se puede realizar en la *Double LinkedList* sin problemas, mientras que esto no sería posible en una *LinkedList.*

Además de la función de ordenamiento se utilizaron las funciones mostradas en la Tabla 1, donde también se muestra la complejidad de cada 1. Esto es de gran importancia porque define qué tan rápido responderá nuestro programa ante los requerimientos del usuario. En base a la tabla se puede observar que la función que podría generar un tiempo de respuesta lento es la función de ordenamiento *sort*, debido a que en el peor caso tiene un comportamiento cuadrático. Además de ello, las funciones de costo lineal también generan un comportamiento relativamente lento. Sin embargo, en base a los resultados obtenidos se puede concluir que las funciones utilizadas tienen un tiempo de respuesta aceptable.

Función	Descripción	Complejidad
getNumElements	Obtiene el número de elementos total.	O(1)
addFirst	Añade un dato al inicio	O(1)
addLast	Añade un dato al final	O(1)
getData	Obtiene la data en una posición dada	O(n)
getNumSublist	Obtiene la cantidad de elementos de una sublista dada.	O(n)
sort	Ordena los datos en base al tiempo con el algoritmo *quicksort* iterativo.	Caso promedio: O(n*log(n)) Peor caso: O(n^2)
findRegistro	Encuentra un registro dado el tiempo.	O(log2(n))

REFERENCIAS

[1] N. Wirth, Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

[2] H. Kumar, “#2. importance and types of data structures,” Medium, 01-Jul-2019. [Online]. Available: https://medium.com/data-structure/importance-and-types-of-data-structures-7f90c6259826.