Reflexión

Carla Oñate Gardella A01653555

Introducción

Este trabajo a diferencia de la actividad integradora uno fue más complicado. Esta actividad te pide usar listas doblemente ligadas en todas las funciones, te pide que ordenes con un merge o un quick-sort y que uses dentro un stack o un queue. Así mimo te pide guardar buscar un rango de valores y guardar esos datos en un archivo, así como la bitácora ordenada en otro.

Lo que yo hice fue que use listas doblemente ligadas por lo que se tiene tanto el head como el tail en mis funciones. En algunos casos solo se hace el uso de head ya que no es necesario el valor de tail, pero en la mayoría se usa tanto head como tail. Para hacer la lista doblemente ligada use las funciones que se vieron en clase como addLast, findData, printList, etc. En el caso del algoritmo de ordenamiento ....use el mergeSort sin stack para poder probar las otra funciones, por lo que no pude hacer que funcionara el mergeSort con queue ni el quickSort con stack ... Para la búsqueda se pide que uses **binary search** por lo que se uso ese algoritmo pero ahora pasándolo el head de la lista y los valores que se comparaban era el data de los nodos. Así mismo esta búsqueda te regresa el nodo del primer y el segundo valor de búsqueda del usuario. Luego se busca imprimir los valores del rango de búsqueda y eso se hace con `printSearch()` que a su vez llama a `searchFile()` para crear el archivo con los resultados. Este es un funcionamiento básico de mi actividad.

Estructuras de Datos

Como ya se mencionó antes en esta actividad se usan dos estructuras de datos. Primero es la de **lista doblemente ligada** y la segunda es la de **stack**.

Lista doblemente ligada

Esta lista tiene tanto el pointer del inicio como el pointer del final, así mismo en esta lista puedes ir en dirección hacía el `tail` o hacía el `head` ya que cada nodo tiene su valor `next` y su valor `prev`. Estos dos valores nos sirven en algunas funciones como addLast, en la cual ya tienes el valor de tail por lo que es más sencillo ligarlo a que solo tenga el valor de head y tengas que ir hasta el final de la lista para ligar el nuevo nodo.

Comparada con la lista simple la lista doblemente ligada tiene un valor de dificultad mayor porque al hacer operaciones se tiene que cuidar ligar bien el `next` y el `prev`. Al igual la lista doblemente ligada ocupa mas memoria porque ahora son dos punteros en cada Nodo, mientras que lista ligada solo tiene un puntero.

Stack

El stack es una estructura de datos lineal, al igual que el queue solo que los dos tiene diferente funcionamiento. En esta actividad se usó el stack en vez del queue porque para el algoritmo de quick-sort es mas sencillo implementar stack que queue. Al igual el **merge sort** se implementa con un queue. El stack es una pila por lo que si entra un valor hasta arriba ese será el valor que salga primero. En el caso del queue el que entra es el último en salir porque se agrega hasta el final de la cola.

Algoritmos de ordenamiento

En esta actividad se podía usar el **merge sort** o el **quick sort**, en mi caso intente aplicar los dos con stack o queue pero no lo logré. Los dos algoritmos tienen que mejor caso $O(nlogn)$ pero el quick sort tiene como peor caso $O(n^2)$. Él pero caso del **merge sort** es igual que su mejor caso por lo que es $O(nlogn)$. Por el hecho de que tanto el mejor como el peor caso tienen una complejidad minima intenté hacer esta actividad con el merge, pero debido a que no logré encontrar algoritmos similares que usarán el queue o explicaciones sobre dónde se usa este queue. Intente hacerlo con otro código que existía pero no compilaba.

En el caso del quick sort sí hay un ejemplo con stacks y ese fue le que intente replicar en mi actividad pero hay unos errores en la función para intercambiar nodos por lo que o no ordena bien o se queda en un ciclo infinito. Por lo que no logre aplicar ningún algoritmo con stack pero por lo que entiende el stack o el queue van guardando la info qué sale del algoritmo, como el return que se hace dentro de estos algoritmos de forma recursiva. Solo que el código iterativo es diferente al recursivo por lo que no logré identificar este punto dónde se usan este tipo de estructuras lineales. En este algoritmo al igual tenía que saber el index de los nodos que estaba analizando por lo que tuve que llamar una función que me regresa el indice, la cual tenía que ir linealmente por toda la bitácora. Así que aunque la complejidad del algoritmo sea la mejor al agregar esta función para el indice se tiene una complejidad de $O(n)$. Esta función se tiene que correr tantas veces sea necesario en lo que ordena el algoritmo por lo que esto bajó el rendimiento total del algoritmo de ordenamiento significativamente.

Algoritmo de búsqueda

En esta actividad se pide usar **búsqueda binaria** por lo que ese algoritmo fue el que se implementó. El algoritmo de búsqueda binaria tiene como mejor caso una complejidad de $O(1)$ y en el peor caso de $O(1)$. Sin embargo el caso promedio tiene una complejidad de $O(logn)$. Este algoritmo se implemente con la lista doblemente ligada por lo que recibía el head y el nodo del valor. Dentro de esta función se comparan las fechas del nodo medio con el valor dado. En este caso para comparar igualdad se hizo checando manualmente cada parte de la fecha, mientras que para saber si era menor o mayor se usó el valor que da `mktime` el cual genera un numero desde la creación del epoch que es igual a la fecha. Dentro del algoritmo de búsqueda no se tuve que agregar alguna función extra como en el caso de quick-sort por lo que este algoritmo mantuvo su complejidad inicial.