Reflexión Actividad Integradora 2

Jacobo Soffer Levy

A01028653

Campus Santa Fe

Los algoritmos de ordenamiento y búsqueda son de los más importantes y más usados en el mundo de la programación. Hacen muchas tareas mucho más sencillas y eficientes, como por ejemplo, un algoritmo de ordenamiento puede hacer más fácil encontrar un elemento dentro de un arreglo usando un algoritmo de búsqueda como Binary Sort, que solo funciona con arreglos ordenados. Puede hacer más fácil encontrar valores únicos o duplicados en un arreglo, analizar los datos que contiene, presentar la información al usuario, entre otras cosas. Los algoritmos de búsqueda nos permiten encontrar un elemento en un mar de información de forma eficiente. Varios programas dependen de algoritmos de búsqueda, incluidos muchos que usamos día a día. Aunque existe una gran probabilidad que cualquier programa relativamente funcional haga uso de estos algoritmos, algunos ejemplos muy claros son aquellos de un motor de búsqueda o la funcionalidad de búsqueda incluida en muchos programas. Estos programas dependen de estos algoritmos para darnos la información que necesitamos, y de no tenerlos, encontrar la información sería un proceso mucho más tedioso para el usuario. Aunque a primera vista es evidente su uso de algoritmos de búsqueda, los algoritmos de ordenamiento también son esenciales para que al tener la información ordenada, pueda ser más fácil y eficiente encontrarla, así como para ordenar la lista de resultados que es presentada al usuario de una manera que es útil y hace sentido.

Pero no sólo existe un algoritmo de búsqueda o un algoritmo de ordenamiento. Sino que existen varios que abordan el problema de diferentes maneras. Esto no significa que uno sea mejor que otro, sino que son aptos para casos de uso diferentes, y serán eficientes para el adecuado. Por lo tanto, es responsabilidad del programador identificar que algoritmo será mejor para el problema que esta tratando de resolver. Esta fue una consideración importante al momento de diseñar el programa de la Actividad Integradora 2, en la cual se tienen que desplegar registros entre ciertas fechas especificadas por el usuario. Estos registros vienen dados en una bitácora con más de 16,000 registros desordenados, los cuales no vienen dados en un orden particular. Es por esto que hace sentido ordenar estos registros, ya que se tienen que mostrar de esta forma al usuario, y hace mucho más eficiente la tarea de encontrar estos registros al sólo tener que encontrar el registro inicial y final dentro de un rango de fechas.

Para esta actividad, a diferencia de la Actividad Integradora 1, se utilizó una lista doblemente ligada para almacenar la información. La lista doblemente ligada es una estructura de datos lineal muy simple permite almacenar información de una forma similar a la de una arreglo. La lista en esencia, es una colección de nodos almacenados en memoria virtual, los cuales contienen el dato almacenado en ellos, y un apuntador al nodo anterior y al siguiente. La lista contiene apuntadores al primer y ultimo nodo de la lista. Para iterar por la lista para obtener un nodo y realizar alguna operación con el es suficiente con empezar con el primer o ultimo nodo de la lista, y usar los apuntadores al nodo anterior o siguiente para llegar al nodo deseado. Para modificar el orden de la lista, agregar o eliminar elementos, basta con cambiar los apuntadores de los nodos necesarios de la lista para obtener el resultado deseado. Hacía sentido usar una lista doblemente ligada en lugar de una ligada (sus nodos solo contienen apuntadores al nodo siguiente) debido a que esta permite iterar en reversa y

obtener acceso al nodo anterior, lo que hace al programa y a algunas operaciones dentro del mismo más eficientes, lo cual es importante ya que se maneja una cantidad relativamente grande de datos en el programa. El uso de una lista doblemente ligada en lugar de un vector tiene consecuencias mayores en el funcionamiento del programa. Primero que nada, la complejidad temporal del acceso a elementos cambia. Debido a que en la lista, no esta asegurado que los nodos estén en memoria continua, para llegar a un elemento de la lista se tiene que iterar por toda la lista, nodo por nodo, hasta llegar al elemento deseado. Esto hace que el acceso a elementos de forma aleatoria no es preferible, ya que es poco eficiente, y por lo tanto es mejor accesar elementos de forma lineal cuando es posible, cosa que se tomó en cuenta durante la realización del programa. Para hacer el programa más eficiente, se crearon métodos para navegar por la lista, que dependiendo del indice del elemento a buscar, empiezan a iterar o por el final de la lista y en reversa, o desde el inicio. Esto hace que la complejidad temporal de acceso a la lista, en el mejor caso sea de $O(1)$, cuando se quiere accesar al primer o ultimo elemento de la lista, y en el caso promedio sea de $O(n)$, ya que en el peor caso se itera por la mitad de la lista por lo que: $O(\frac{n}{2}) => O(n)$. La inserción de elementos en la lista tiene una complejidad temporal de $O(1)$, ya que el programa solo permite insertar elementos al inicio o al final de la lista, y al tener apuntadores a estos nodos, su acceso es en tiempo constante. La actualización o eliminación de elementos en la lista tiene una complejidad temporal de $O(n)$, ya que hacer estas opciones requiere accesar al nodo, por lo que tienen la misma complejidad temporal que el acceso a la lista.

Otra estructura de datos usada en este programa fue un Queue. Esta también es una estructura de datos lineales muy simple, y la implementación que usa este programa esta basada en la Lista Doblemente Ligada. En esta estructura se puede almacenar cualquier tipo de elemento, sólo existen métodos para insertar y eliminar elementos del Queue, además de algunos métodos auxiliares para obtener el tamaño o si esta vacía. En un Queue no se puede elegir la posición donde se insertan o eliminan elementos, sino que funciona como una fila, donde lo primero que entra es lo primero que sale. Esto hace que sea una estructura de datos muy eficiente, en especial por que esta basada en la lista doblemente ligada. Las operaciones para insertar y eliminar elementos de la lista son de $O(1)$, ya que sólo se añaden elementos al final de la lista en la que esta basada y se eliminan del principio, y como se explico anteriormente, la complejidad temporal de estas operaciones en la lista doblemente ligada es de $O(1)$.

Para este programa, se decidió usar el algoritmo de Merge Sort (MS), en lugar de Quick Sort (QS), que se usó en la actividad integradora pasada. En papel, hace más sentido usar QS que MS, ya que tienen la misma complejidad temporal para todos sus casos, $O(nlog(n))$, pero la complejidad espacial de MS es de $O(n)$, mientras que la de QS es de $O(logn)$. Cabe notar que la complejidad temporal de QS en su peor caso es de $O(n^2)$, pero esto es irrelevante para el análisis de este programa ya que al esperar particiones algo balanceadas ya que los elementos no tienen un orden

particular, entra en el caso promedio de QS. La razón por la que se optó por usar MS en lugar de QS, es que al usar listas ligadas en lugar de vectores, las complejidades de estos algoritmos cambian. La implementación de MS usada funciona de forma iterativa, funciona metiendo todos los nodos de la lista a un Queue y haciendo que sus apuntadores al nodo previo y siguiente sean nulos. Posteriormente, se eliminan dos nodos del Queue, los cuales son pasados a un método auxiliar llamado merge, el cual une estos nodos y todos los nodos a los que llevan en un solo nodo, el cual es regresado al Queue. De esta forma, eventualmente el Queue termina teniendo un solo elemento. Cuando esto pasa, el elemento es sacado del queue y usado como el nuevo head de la lista, a partir del cual se calcula el tail. Ya sabiendo esto, podemos llegar a que la complejidad temporal de este algoritmo es de $O(nlog(n))$ para todos los casos. La n deriva de que iteramos por toda la lista para meter los nodos al queue. El $log(n)$ deriva de que cada vez que se llama merge, el número de elementos n a ordenar se divide a la mitad. Otra forma en la que cambia el funcionamiento de este algoritmo del usado en vectores es la función merge. Mientras que en el algoritmo tradicional, se tienen que copiar los arreglos que se van a unir, en esta versión no se requiere de memoria adicional, ya que solo se modifican los vínculos al nodo previo y siguiente de los nodos a unir. Para esto se comparan los nodos a unir, el menor se asigna como el nodo principal y el mayor se asigna como el nodo siguiente a este. En caso de que los nodos pasados a merge apunten a otro nodo, se itera de forma lineal hasta que uno de los nodos llegue a ser null, repitiendo la comparación realizada anteriormente hasta terminar, y finalmente añadiendo los nodos restantes a la cadena. El hecho de que no se crea memoria adicional hace que la complejidad de la implementación de este algoritmo sea de $O(1)$, haciéndolo un algoritmo muy eficiente. Al tener esta información, se puede ver porque es preferible usar Merge Sort en lugar de Quick Sort para listas ligadas. En primer lugar, el algoritmo es más eficiente ya que aunque tienen la misma complejidad temporal, MS tiene una complejidad espacial mucho mejor (O(1) en lugar de $O(logn)$). Además de esto, Quick Sort depende de acceso a memoria de forma aleatoria haciéndolo poco eficiente para listas ligadas, mientras que Merge Sort itera de forma lineal.

Quick Sort funciona particionando el arreglo, tomando un elemento como pivote (en este caso el ultimo elemento) y asegurándose de que todos los elementos más pequeños queden del lado izquierdo del pivote, y los más grandes del derecho, de esta forma generando dos particiones. Las particiones son generadas con una función auxiliar que tiene una complejidad temporal y espacial de O(n), ya que itera por todo el arreglo (o la partición que esta siendo utilizada) e intercambia cualquier elemento menor al pivote con el elemento i+1, i siendo el indice del mínimo elemento más pequeño. Cuando se termina la iteración, solo queda intercambiar el pivote con el elemento i+1. Una vez que tenemos estas particiones el algoritmo se vuelve a llamar de forma recursiva para cada partición.
Gracias a esto sabemos que el algoritmo tiene una complejidad de $O(nlogn)$, ya que la función que participan el arreglo es llamada una vez por cada elemento del arreglo y esta tiene una complejidad de $O(n)$, y al ser un caso promedio, podemos esperar

particiones balanceadas, por lo que con cada llamada a la función, el tamaño de n es dividido entre 2, lo cual es equivalente a decir $O(log(n))$. Lo que nos deja con lo siguiente: $O(n) * O(logn) = O(nlogn)$. Aunque en un caso promedio las particiones no siempre van a quedar perfectamente balanceadas (por lo que el tamaño de n no es dividido entre dos con cada partición), la complejidad del problema sigue siendo $O(nlogn)$. Esto se debe a que lo que va a cambiar con la proporción de elementos por partición es la base de $log(n)$ para representar esta nueva proporción. Sin embargo, sabemos que $log_a n = log_b n / log_b a$, por lo que llegamos a que: $log_c n = log_2 n / log_2 c$, c siendo la base del logaritmo y una constante, por lo que el resultado del logaritmo dividiendo a $log_2 n$ es una constante y es eliminado según las reglas del análisis asintótico, por lo que la complejidad temporal para el caso promedio es de $O(nlogn)$. Aunque tiene esta complejidad temporal, el algoritmo sufre en esta área por la forma en la que obtiene elementos de la lista, ya que tiene que realizar varias iteraciones por esta en cada iteración del algoritmo.

Para el algoritmo de búsqueda se decidió usar Binary Search. Este algoritmo funciona calculando la variable m, que es igual a la mitad del arreglo. Si A[m] == k, k siendo el valor que se busca, el algoritmo se detiene. Si A[m] > k, el algoritmo se repite, esta vez posicionando m dentro de la primera mitad del arreglo o en la segunda sí A[m] < k. Debido a que se eliminan mitades con cada ejecución de este algoritmo, su complejidad temporal es $O(logn)$. Existe la probabilidad de que un mejor algoritmo para realizar la búsqueda en listas ligadas sea la búsqueda secuencial, el cual consiste en iterar por la lista hasta que se encuentre el elemento buscado. Esto se debe a que cada vez que Binary Search calcula un elemento m nuevo, se tiene que acceder a ese elemento, lo que hace que se tenga que iterar por toda la lista hasta llegar a ese elemento, lo que podría hacer al algoritmo menos eficiente que su contraparte. Una posible solución a este problema podría ser la iteración de la lista usando un nodo base, del cual se pueda llegar más rápido al siguiente nodo m. Otra alternativa puede ser que cada vez que Binary Search itere por la lista para llegar al elemento m, también se realice una comparación para ver si alguno de esos elementos es el que se busca. Por cuestiones de tiempo, no se pudieron probar ninguna de las propuestas anteriores.

El hacer este programa me ayudó a darme cuenta del rol importante que juegan las estructuras de datos lineales y algoritmos de ordenamiento y búsqueda en el mundo de la programación. Me impresiona cómo estructuras de datos tan simples como un Linked List o un Queue pueden ser tan versátiles y tener un impacto grande en el desarrollo de los programas y el impacto que pueden tener en su ejecución si se usan las correctas. En este caso, el cambio de un vector a una lista doblemente ligada vino con mejoras significantes en el tiempo de ejecución del programa, lo cual probablemente se debe a que el algoritmo de ordenamiento es más eficiente. Otra área donde las estructuras de datos lineales jugaron un rol importante en el programa fue en el algoritmo de ordenamiento. Si no hubiera sido por el uso del queue, no hubiera sido posible implementar el algoritmo de forma iterativa. Y de haber usado otra

estructura com un stack o un arreglo, la complejidad de implementación hubiera aumentado considerablemente. Esto también me ayudó a darme cuenta de la importancia que tiene el saber elegir algoritmos y estructuras de datos correctamente para el caso en el que se van a usar, ya que de haber usado estructuras diferentes, la complejidad de implementación o la eficiencia del programa hubieran sufrido. O de haberme quedado con Quick Sort en lugar de usar Merge Sort, el programa hubiera sido mucho menos eficiente por la forma en la que se accesa a la lista y la complejidad espacial. Considero que gracias a esto pude desarrollar la competencia de solución de problemas con computación, ya que tuve que evaluar el problema, tomar decisiones adecuadas e implementarlo de forma adecuada.