Tecnológico de Monterrey

Act 2.3 - Actividad Integral Estructura de Datos Lineales

Programación de Estructuras de Datos y Algoritmos Fundamentales

Instituto Tecnológico y de Estudios Superiores de Monterrey

Profesor: Dr. Eduardo Arturo Rodríguez Tello

Fecha de entrega: 15 de julio de 2023

Estudiantes: Alyson Melissa Sánchez Serratos - A01771843

ACTIVIDAD INTEGRAL: REFLEXIÓN INDIVIDUAL

En el mundo actual, las estructuras de datos lineales desempeñan un papel fundamental en diversas áreas, especialmente en el ámbito de la tecnología y la gestión de información. Estas estructuras proporcionan una organización eficiente para almacenar y manipular datos de manera secuencial, lo cual es esencial en una amplia gama de aplicaciones y sistemas.

En entornos empresariales y científicos, donde la recolección y el análisis de datos son cruciales, las estructuras de datos lineales son ampliamente utilizadas para una gestión efectiva de grandes volúmenes de información. Por ejemplo, en bases de datos y sistemas de gestión de información, se emplean listas enlazadas, pilas y colas para estructurar y manipular datos de manera adecuada. Asimismo, cuando se necesita buscar y ordenar información basada en fechas, como en el caso de la situación problema actual, el uso apropiado de estas estructuras lineales puede marcar la diferencia entre un programa eficiente e ineficiente. Por lo tanto, es fundamental adquirir habilidades sólidas en la manipulación de estas estructuras y tomar decisiones inteligentes al seleccionar la estructura de datos más adecuada para el análisis y la extracción precisa de la información deseada.

En este sentido, las listas enlazadas son estructuras de datos ampliamente utilizadas en diversos contextos debido a su versatilidad y capacidad para adaptarse a la construcción de estructuras más complejas. Para ello es pertinente citar a Weiss, M. A. (2013) el cual afirma lo siguiente:

> La lista enlazada consta de una serie de nodos, que no son necesariamente adyacentes en memoria. Cada nodo contiene el elemento y un enlace a un nodo que contiene su sucesor. Nosotros llamamos a este el enlace **next**. El enlace **next** de la última celda apunta a nullptr. (p. 79)

En particular, para la presente actividad integradora se utilizó el concepto de listas doblemente ligadas donde, de acuerdo con Shaffer, C. A. (2011) “son estructuras de datos en las que cada nodo contiene referencias tanto al nodo siguiente como al nodo anterior, permitiendo así un acceso bidireccional.” (p. 115)

A continuación, en la **figura 1** se muestra una representación de una lista doblemente ligada.

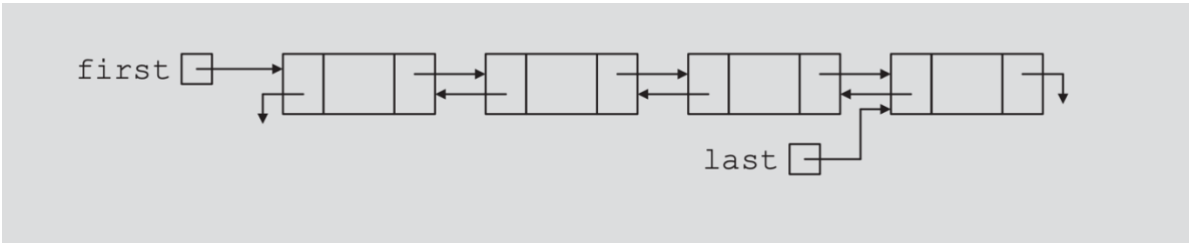

Figura 1 Lista doblemente ligada. Malik, D. (2009)

Esta característica facilita el recorrido en ambas direcciones de la lista, lo cual resulta beneficioso para las operaciones de búsqueda y acceso a los elementos. Además, las listas enlazadas dobles se destacan por su eficiencia en términos de inserción y eliminación de elementos. Al insertar un nuevo nodo, solo es necesario actualizar las referencias del nodo actual, el nodo siguiente y el nodo anterior, evitando así una reorganización completa de la estructura. Del mismo modo, al eliminar un nodo, únicamente se requiere actualizar las referencias de los nodos adyacentes. Esta característica contrasta con otras estructuras de datos, como los arrays, donde las operaciones de inserción y eliminación pueden ser más costosas debido a la necesidad de desplazar otros elementos.

En cambio, las listas enlazadas simples solo permiten el acceso a los elementos en una dirección, generalmente de principio a fin. Esto dificulta el acceso a los elementos anteriores a un nodo específico o realizar búsquedas inversas en la lista. Asimismo, la inserción de un nodo en una lista enlazada simple puede requerir recorrer la lista desde el principio hasta el nodo anterior al que se desea insertar o eliminar. Este enfoque puede ser ineficiente en términos de tiempo de ejecución, especialmente para listas de gran longitud, ya que implica un recorrido lineal completo. Por último, aunque es posible ordenar una lista enlazada simple utilizando algoritmos de ordenamiento, el acceso unidireccional puede complicar la eficiencia de estas operaciones. De acuerdo con Jain, A. (2021), algunos algoritmos de ordenamiento, como el mergesort o el quicksort, requieren acceso aleatorio constante a los elementos para lograr un rendimiento óptimo. En este sentido, las listas enlazadas dobles resultan más adecuadas debido a su acceso bidireccional.

Por lo tanto, tras haber realizado un profundo análisis, se comprueba que las listas enlazadas dobles superan a las listas enlazadas simples en términos de eficiencia en el acceso, eliminación, inserción y operaciones de ordenamiento, cumpliendo así los requisitos específicos planteados por la situación problema.

Es importante recalcar que esta conclusión no siempre será la misma. La elección de una estructura de datos adecuada dependerá del problema a solucionar y las restricciones de implementación.

Para culminar el escrito, en la **tabla 1** se describirán las operaciones básicas de la lista enlazada doble, examinando su complejidad computacional, su importancia en la solución de la situación problema, y su eficiencia.

Tabla 1 Descripción de operaciones básicas de una lista doblemente ligada

Operación	Descripción	Complejidad computacional
Inserción	**addFirst ():** este método agrega un nuevo nodo al principio de la lista doblemente ligada. Para realizar esta operación, se siguen los siguientes pasos descritos por Malik, D. (2009). - Se crea un nuevo nodo con los datos proporcionados.	La complejidad computacional de este método es constante O(1) ya que solo se realizan operaciones directas en los nodos existentes sin necesidad de recorrer la lista.

	- Se establece la referencia al nodo siguiente del nuevo nodo como el actual primer nodo de la lista. - Se establece la referencia al nodo anterior del nuevo nodo como nullptr, ya que ahora será el primer nodo. - Se actualiza la referencia al nodo anterior del antiguo primer nodo para que apunte al nuevo nodo. - Finalmente, se actualiza el primer nodo de la lista para que apunte al nuevo nodo.	
Inserción	**addLast ():** este método agrega un nuevo nodo al final de la lista doblemente ligada. Los pasos para realizar esta operación son los siguientes: - Se crea un nuevo nodo con los datos proporcionados. - Se establece la referencia al nodo anterior del nuevo nodo como el actual último nodo de la lista. - Se establece la referencia al nodo siguiente del nuevo nodo como nulllptr, ya que ahora será el último nodo. - Se actualiza la referencia al nodo siguiente del antiguo último nodo para que apunte al nuevo nodo. - Finalmente, se actualiza el último nodo de la lista para que apunte al nuevo nodo.	La complejidad computacional es constante O(1) porque solo se realizan operaciones directas en los nodos existentes sin necesidad de recorrer la lista.
Eliminación	**deleteData ():** este método elimina el primer nodo que contiene un dato específico en la lista doblemente ligada. Los pasos para realizar esta operación son los siguientes: - Se comienza desde el primer nodo y se recorre la lista hasta encontrar el nodo que contiene el dato buscado.	La complejidad computacional de deleteData () depende de la posición del nodo que contiene el dato en la lista.

	- Una vez encontrado el nodo con el dato, se actualizan las referencias de los nodos anterior y siguiente para saltar el nodo que se eliminará. - Finalmente, se libera la memoria del nodo eliminado.	En el peor de los casos, donde el dato se encuentra al final de la lista o no está presente, la complejidad sería lineal O(n) ya que se tendría que recorrer toda la lista.
Eliminación	**deleteAt ():** este método elimina un nodo en una posición específica de la lista doblemente ligada. Los pasos para realizar esta operación son los siguientes: - Se comienza desde el primer nodo y se recorre la lista hasta llegar a la posición indicada. - Una vez que se llega a la posición deseada, se actualizan las referencias de los nodos anterior y siguiente para saltar el nodo que se eliminará. - Finalmente, se libera la memoria del nodo eliminado.	La complejidad computacional de deleteAt () depende de la posición del nodo en la lista. En el peor de los casos, donde la posición es el último nodo o está más allá del tamaño de la lista, la complejidad sería lineal O(n) debido a la necesidad de recorrer toda la lista.
Búsqueda	**binarySearch ():** se utiliza para buscar un elemento específico utilizando el algoritmo de búsqueda binaria. A continuación, se describen los pasos para realizar esta operación: - Se obtiene el primer y último nodo de la lista, y se calcula el punto medio. (head y tail) - Se compara el elemento buscado con el elemento en el nodo del punto medio. Si son iguales, se ha encontrado el elemento y se retorna la posición. - Si el elemento buscado es menor que el elemento en el nodo del punto medio, se realiza la búsqueda en la mitad inferior de la lista, es decir, desde el primer nodo hasta el nodo anterior al punto medio. Se repite el proceso desde el paso 1 en esta mitad.	La complejidad computacional de binarySearch () depende del tamaño de la lista y sigue una complejidad de tiempo logarítmica O (log n), donde "n" es el número de elementos en la lista.

	- Si el elemento buscado es mayor que el elemento en el nodo del punto medio, se realiza la búsqueda en la mitad superior de la lista, es decir, desde el nodo siguiente al punto medio hasta el último nodo. Se repite el proceso desde el paso 1 en esta mitad. - Se continúa dividiendo la lista en mitades y comparando el elemento buscado con el elemento en el nodo del punto medio, hasta que se encuentre el elemento o se determine que no está presente en la lista.	

Como se puede observar, las operaciones descritas anteriormente son imprescindibles para el manejo adecuado de una lista doblemente enlazada ya que ofrecen eficiencia en la inserción y eliminación de elementos, permiten una búsqueda eficiente y brindan flexibilidad en el manejo de datos. Estas capacidades son esenciales al trabajar con estructuras de datos y operaciones en bases de datos o conjuntos de datos grandes, donde la eficiencia y el rendimiento son fundamentales. En particular, la eficiencia de las listas doblemente ligadas en inserción y borrado la hace ideal para escenarios que requieren manipulación frecuente de elementos en la lista.

Fuentes de información:

Jain, A. (2021, agosto 19). Learn easy method to Sort Doubly Linked List using Merge Sort. PrepBytes Blog. https://www.prepbytes.com/blog/linked-list/merge-sort-for-doubly-linked-list/

Malik, D. (2009). Data Structures Using C++ (2a ed.). South-Western. https://bu.edu.eg/portal/uploads/Computers%20and%20Informatics/Computer%20Science/1266/crs-10600/Files/Esam%20Halim%20Houssein%20Abd%20El-Halim_4-%20Data-Structure%20Using%20C++%20Malik.pdf

Shaffer, C. A. (2011). Data Structures and Algorithm Analysis in Java, Thi. Dover Publications. https://www.academia.edu/2817564/Data_structures_and_algorithm_analysis

Weiss, M. A. (2013). Data Structures and Algorithm Analysis in C++ (4a ed.). Pearson. https://www.uoitc.edu.iq/images/documents/informatics-institute/Competitive_exam/DataStructures.pdf