Nombre: Jefersson Jhovani Coronel Lavadenz

Practica 1

Parte 1:

1. Piensa en una situación donde hayas utilizado una estructura de datos convencional como un array, una lista enlazada o una pila. Detalla sobre los beneficios y las limitaciones de esa estructura en ese contexto específico. ¿Por qué elegiste esa estructura de datos?

R: Una situación en la que he utilizado una estructura de datos convencional como una cola fue en la que estemos desarrollando un sistema de gestión de tickets de un Banco evaluando por prioridad a los clientes. En este contexto, me plantee usar una pila, cola o un linked list. Pero me fui por una cola ya que necesitaba que el primer cliente en llegar sea el primero en salir, es decir el cliente con el ticket de menor número será atendido más rápido o primero.

Elegí utilizar una cola porque tenía varios beneficios en este contexto específico:

- **Ordenamiento:** La cola garantiza que las tareas se procesen en el orden en que se recibieron. Esto es especialmente útil cuando se requiere una secuencia de procesamiento específica y se necesita mantener la integridad de los datos.

- **Eficiencia:** Agregar elementos al final de la cola y eliminar elementos del frente de la cola es una operación eficiente, lo que permite un procesamiento rápido de las tareas entrantes. No es necesario realizar desplazamientos masivos de elementos como en otras estructuras de datos, como las listas.

- **Capacidad limitada:** Si la cola tiene un tamaño máximo definido, se puede controlar la carga del sistema y evitar que se desborde con un número excesivo de tareas. Esto ayuda a mantener un rendimiento óptimo y a evitar la saturación del sistema.

- Escalabilidad: La cola es una estructura de datos fácilmente escalable. A medida que llegan más tickets, se pueden agregar a la cola sin afectar significativamente el rendimiento del sistema. Además, si se requiere un mayor nivel de paralelismo, se pueden implementar múltiples colas para distribuir la carga de trabajo entre diferentes agentes de soporte.

Sin embargo, también hay algunas limitaciones asociadas con el uso de una cola en esta situación:

- **Acceso arbitrario:** En una cola convencional, solo es posible acceder al elemento frontal y eliminarlo. Si se necesita acceder a elementos en posiciones aleatorias, una cola no es la estructura de datos adecuada.

- **Capacidad fija**: Si bien puede ser beneficioso tener una capacidad limitada para controlar la carga del sistema, también puede ser una limitación si se espera un alto volumen de tareas entrantes. Si la cola alcanza su capacidad máxima, se deben tomar medidas adicionales, como redimensionar la cola o descartar tareas.

2. Compara y contrasta las características, ventajas y desventajas en términos de tiempo de ejecución, capacidad de almacenamiento y facilidad de implementación de un arraylist y un linkedlist.

Caracterisitica	ArrayList	LinkedList
Asignación de memoria	Los elementos en un ArrayList se almacenan en un bloque contiguo de memoria. El tamaño del ArrayList se fija al crearlo, aunque puede redimensionarse dinámicamente cuando sea necesario.	Cada elemento (nodo) en una lista enlazada contiene una referencia al siguiente nodo en la secuencia. Se asignan dinámicamente en memoria y su tamaño puede crecer o disminuir según sea necesario.
Inserción y eliminación	Las operaciones de inserción y eliminación en un ArrayList pueden ser más lentas, especialmente al agregar o eliminar elementos desde el principio o el medio de la lista. Requiere desplazar elementos para hacer espacio para el nuevo elemento o llenar el espacio dejado por el elemento eliminado. Esta operación tiene una complejidad temporal de O(n). Sin embargo, acceder a elementos por índice es rápido, con una complejidad temporal constante de O(1).	Las operaciones de inserción y eliminación pueden ser eficientes en una lista enlazada, ya que solo requieren actualizar las referencias de los nodos adyacentes. Tiene una complejidad temporal constante de O(1) para insertar/eliminar elementos al principio o al final de la lista. Sin embargo, acceder a un elemento específico requiere recorrer la lista, lo que resulta en una complejidad temporal de O(n).
Acceso a elementos	Acceder a elementos en un ArrayList es eficiente. Como los elementos se almacenan en un bloque contiguo de memoria, acceder a un elemento por índice tiene una complejidad temporal constante de O(1).	Acceder a elementos en una lista enlazada requiere recorrer la lista desde el principio o el final, lo que resulta en una complejidad temporal de O(n). El acceso aleatorio no es posible y para llegar a un elemento específico, debes partir desde la cabeza (o la cola) y seguir las referencias.
Eficiencia en memoria	Los ArrayList pueden ser menos eficientes en memoria en comparación con las listas enlazadas, ya que asignan memoria en un bloque contiguo, incluso si el número real de elementos es pequeño.	Las listas enlazadas pueden ser más eficientes en memoria en comparación con los ArrayList cuando el tamaño de la lista cambia con frecuencia. Cada nodo en una lista enlazada solo requiere memoria para los datos y una referencia al siguiente nodo, mientras que un ArrayList requiere memoria para los elementos reales independientemente de su número.
Uso	Los ArrayList se prefieren cuando se necesita un acceso aleatorio rápido y un recorrido eficiente por índice. Funcionan bien para escenarios donde	Las listas enlazadas se utilizan comúnmente cuando se requiere una inserción y eliminación eficiente al principio o al final de la lista, y el

	el tamaño de la colección se mantiene relativamente estable.	acceso aleatorio no es crucial.

ArrayList	LinkedList
ArrayList internamente utiliza un array dinámico para almacenar los elementos.	LinkedList internamente utiliza una lista doblemente enlazada para almacenar los elementos.
La manipulación de ArrayList es lenta porque internamente utiliza un array. Si se elimina cualquier elemento del array, todos los demás elementos se desplazan en la memoria.	La manipulación de LinkedList es más rápida que ArrayList porque utiliza una lista doblemente enlazada, por lo que no se requiere desplazamiento de bits en la memoria.
La clase ArrayList puede actuar solo como una lista porque implementa solo la interfaz List.	La clase LinkedList puede actuar como una lista y una cola porque implementa las interfaces List y Deque.
ArrayList es mejor para almacenar y acceder a los datos.	LinkedList es mejor para manipular los datos.
La ubicación en memoria de los elementos de un ArrayList es contigua.	La ubicación de los elementos de una linked list no es contigua.
Generalmente, cuando se inicializa un ArrayList, se asigna una capacidad predeterminada de 10 al ArrayList.	En LinkedList, se crea una lista vacía cuando se inicializa una LinkedList. No hay caso de capacidad predeterminada en LinkedList.
Para ser precisos, un ArrayList es un array redimensionable.	LinkedList implementa la lista doblemente enlazada de la interfaz list.

3. Elige dos estructuras de datos convencionales que hayas implementado en el pasado. Compara y contrasta sus características, ventajas y desventajas en términos de tiempo de ejecución, capacidad de almacenamiento y facilidad de implementación. Reflexiona sobre en qué situaciones específicas preferirías utilizar una sobre la otra y por qué.

Dos estructuras de datos convencionales que he implementado en el pasado son la lista enlazada y el árbol binario de búsqueda. A continuación, compararé y contrastaré sus características, ventajas y desventajas en términos de tiempo de ejecución, capacidad de almacenamiento y facilidad de implementación:

Lista enlazada:

Características: Una lista enlazada es una estructura de datos en la que los elementos están conectados mediante nodos que contienen referencias al siguiente elemento en la lista. Puede ser una lista enlazada simple (cada nodo tiene un enlace al siguiente nodo) o una lista enlazada doble (cada nodo tiene enlaces tanto al siguiente como al nodo anterior).

Ventajas:

- **Facilidad de inserción y eliminación**: Agregar o eliminar elementos en una lista enlazada es eficiente, ya que solo requiere modificar los enlaces de los nodos adyacentes.
- **Uso eficiente de memoria:** La lista enlazada permite un uso eficiente de la memoria, ya que los elementos pueden estar dispersos en diferentes ubicaciones y solo se requiere el espacio adicional para almacenar los enlaces.

Desventajas:

- **Acceso secuencial:** El acceso a un elemento específico en la lista enlazada requiere recorrerla desde el principio hasta el elemento deseado, lo que puede llevar un tiempo proporcional al tamaño de la lista.
- **Mayor uso de memoria en comparación con arrays:** Los nodos de la lista enlazada requieren espacio adicional para almacenar los enlaces, lo que puede aumentar la sobrecarga de memoria en comparación con los arrays.

Árbol binario de búsqueda:

Características: Un árbol binario de búsqueda es una estructura de datos jerárquica en la que cada nodo tiene dos hijos (izquierdo y derecho) y sigue la propiedad de que los valores en el subárbol izquierdo son menores o iguales que el nodo actual, y los valores en el subárbol derecho son mayores.

Ventajas:

- **Búsqueda eficiente:** La búsqueda en un árbol binario de búsqueda se realiza de manera eficiente, ya que cada comparación reduce a la mitad el número de elementos restantes a considerar.
- **Inserción y eliminación eficientes:** Agregar o eliminar elementos en un árbol binario de búsqueda se realiza de manera eficiente, ya que se pueden realizar reorganizaciones locales sin afectar a todo el árbol.

Desventajas:

- R**endimiento dependiente del equilibrio**: Si el árbol se desequilibra (es decir, tiene profundidades desiguales en los subárboles), el rendimiento puede degradarse y las operaciones pueden volverse más lentas.
- **Mayor complejidad de implementación:** La implementación de un árbol binario de búsqueda es más compleja que una lista enlazada, ya que requiere mantener el orden y el equilibrio del árbol.

Elección de la estructura de datos en situaciones específicas:

Preferiría utilizar una lista enlazada cuando necesite una estructura de datos dinámica con inserciones y eliminaciones frecuentes, y el acceso aleatorio no sea un requisito importante. Por ejemplo, podría usar una lista enlazada para implementar una cola en un sistema de procesamiento

de tareas donde las tareas se agregan al final y se eliminan del principio.

Por otro lado, elegiría un árbol binario de búsqueda cuando necesite almacenar elementos de manera ordenada y realizar búsquedas eficientes. Por ejemplo, si estoy desarrollando un sistema de diccionario en el que necesito buscar rápidamente palabras y sus definiciones, un árbol binario de búsqueda sería una elección adecuada debido a su eficiente búsqueda y capacidad de mantener los elementos ordenados.

En resumen, la elección de la estructura de datos depende de los requisitos específicos de cada situación, considerando aspectos como el tipo de operaciones que se realizarán con mayor frecuencia (inserción, eliminación, búsqueda), el rendimiento esperado y la complejidad de implementación.

Parte 2:

1. ¿Qué implicaciones tiene la elección de una estructura de datos en el diseño y la eficiencia de un programa?

R: La elección de una estructura de datos tiene implicaciones significativas en el diseño y la eficiencia de un programa. La estructura de datos adecuada puede mejorar la eficiencia de las operaciones de búsqueda, inserción, eliminación y manipulación de datos, lo que se traduce en un mejor rendimiento del programa. Además, la elección de la estructura de datos puede afectar la legibilidad, el mantenimiento y la escalabilidad del código.

2. ¿Qué implicaciones tiene la elección de una estructura de datos en el diseño y la eficiencia de un programa?

R: La elección de una estructura de datos tiene implicaciones en el diseño y la eficiencia de un programa. La estructura de datos seleccionada afecta directamente las operaciones que se pueden realizar y la eficiencia de dichas operaciones. Una estructura de datos bien elegida puede mejorar la velocidad de ejecución y el uso eficiente de los recursos, mientras que una elección incorrecta puede llevar a un rendimiento deficiente y a un código difícil de mantener.

3. ¿Cuál es la diferencia entre una estructura de datos convencional y una estructura de datos avanzada?

R: Una estructura de datos convencional se refiere a las estructuras básicas y ampliamente utilizadas, como los arrays, listas, pilas y colas. Estas estructuras suelen ser simples y fáciles de entender e implementar. Por otro lado, una estructura de datos avanzada se refiere a estructuras más complejas y especializadas, como árboles, grafos, conjuntos y mapas. Estas estructuras suelen tener características y propiedades específicas que las hacen más eficientes para ciertos tipos de operaciones o situaciones.

4. ¿Cuáles son algunas ventajas de utilizar estructuras de datos avanzadas en comparación con las convencionales?

R: Algunas ventajas de utilizar estructuras de datos avanzadas en comparación con las convencionales incluyen un mejor rendimiento en ciertas operaciones, una mayor capacidad de representación de relaciones complejas entre los datos, una mayor flexibilidad y una mejor optimización de recursos. Las estructuras de datos avanzadas están diseñadas para abordar problemas específicos y ofrecen algoritmos y operaciones especializados para lograr un mejor rendimiento y eficiencia en esas situaciones.

5. ¿Cuáles son las aplicaciones comunes de los árboles B en problemas reales?

R: Los árboles B son comúnmente utilizados en problemas que involucran operaciones de búsqueda, inserción y eliminación eficientes en grandes conjuntos de datos. Algunas aplicaciones comunes de los árboles B incluyen bases de datos, sistemas de archivos, almacenamiento de claves en memoria secundaria y estructuras de índices.

6. ¿Cuáles son las aplicaciones comunes de los heaps en problemas reales?

R: Los heaps (montículos) son ampliamente utilizados en problemas que involucran la búsqueda y extracción eficientes del elemento más pequeño o más grande. Algunas aplicaciones comunes de los heaps incluyen la implementación de colas de prioridad, algoritmos de ordenación, planificación de tareas y algoritmos de compresión de datos.

7. ¿Cómo se comparan las estructuras de datos convencionales (como arrays y listas enlazadas) con los árboles B en términos de tiempo de ejecución, capacidad de almacenamiento y facilidad de implementación?

R: En términos de tiempo de ejecución, los árboles B generalmente ofrecen una mejor eficiencia para operaciones de búsqueda, inserción y eliminación en comparación con las estructuras de datos convencionales como arrays y listas enlazadas. En cuanto a la capacidad de almacenamiento, las estructuras convencionales como los arrays y las listas pueden tener una mayor flexibilidad y capacidad en comparación con los árboles B, que tienen requerimientos de espacio adicionales para mantener el equilibrio y la estructura del árbol. En términos de implementación, las estructuras de datos convencionales suelen ser más fáciles de implementar y entender, mientras que los árboles B pueden requerir una lógica más compleja y cuidadosa para garantizar su correcto funcionamiento.

8. ¿Cómo afecta la elección de una estructura de datos al rendimiento y la eficiencia de un programa?

R: La elección de una estructura de datos puede tener un impacto significativo en el rendimiento y la eficiencia de un programa. Una estructura de datos bien adaptada al problema y las operaciones requeridas puede mejorar la velocidad de ejecución, minimizar el uso de recursos y optimizar la utilización de la memoria. Por otro lado, una elección inadecuada puede llevar a ineficiencias, cuellos de botella y un código más difícil de mantener y escalar.

9. ¿Cuáles son algunas consideraciones importantes al seleccionar una estructura de datos para manejar grandes volúmenes de datos?

R: Al seleccionar una estructura de datos para manejar grandes volúmenes de datos, es importante considerar el tiempo de ejecución de las operaciones, como búsqueda, inserción y eliminación, así como la eficiencia en el uso de memoria. Las estructuras de datos que ofrecen una complejidad de tiempo de ejecución óptima para las operaciones requeridas, como árboles balanceados o hash tables, son opciones comunes. Además, se debe tener en cuenta la capacidad de escalabilidad de la estructura de datos para manejar un crecimiento en la cantidad de datos sin una degradación significativa en el rendimiento.

10. ¿Qué implicaciones tiene la elección de una estructura de datos en la flexibilidad y escalabilidad de un programa?

R: La elección de una estructura de datos puede tener implicaciones en la flexibilidad y escalabilidad de un programa. Algunas estructuras de datos son más adecuadas para adaptarse a cambios en los requisitos y permitir una mayor flexibilidad en la manipulación de datos. Además, la elección de una

estructura de datos escalable garantiza que el programa pueda manejar grandes volúmenes de datos sin sacrificar el rendimiento y la eficiencia.

11. ¿Cuál es la relación entre la complejidad del código y la elección de una estructura de datos adecuada?

R: La elección de una estructura de datos adecuada puede tener un impacto directo en la complejidad del código. Una estructura de datos bien diseñada y seleccionada puede simplificar el código y hacerlo más legible, mantenible y eficiente. Por otro lado, una elección inapropiada de estructura de datos puede resultar en un código complejo y difícil de entender, lo que puede aumentar la complejidad general del programa. La elección de una estructura de datos adecuada puede ayudar a reducir la complejidad del código y mejorar su calidad general.