Mini Proyecto Estructura de Datos

Nombres: Nicolás Torres
Eduardo Parra

Profesor: José Sebastian Fuentes Sepulveda

Asignatura: Estructura de Datos

Fecha: 8 de Mayo de 2023

Introducción:

El siguiente proyecto tiene como objetivo implementar una estructura de datos híbrida entre un arreglo y una lista, denominada ListArr. La estructura consta de una lista ligada, donde cada nodo contiene un arreglo de tamaño fijo y otros componentes, como una variable entera que almacena la cantidad de datos almacenados y otra variable que almacena el tamaño del arreglo.

Además, se incluyen nodos resumen que sumarizan la información de dos nodos consecutivos en el ListArr, y estos nodos resumen se agrupan en pares hasta obtener un solo nodo que resume toda la información almacenada en el ListArr. Esta estructura de datos permite recorrer facilmente los datos almacenados en el ListArr y también admite la inserción de nuevos valores.

Desarrollo:

La implementación propuesta se trata de una lista enlazada con nodos que contienen arreglos de tamaño fijo, en la que cada nodo tiene un puntero al siguiente nodo. Además, se incluye un nodo especial denominado NodoResumen que almacena información general de los nodos que le preceden.

La lista enlazada se implementa mediante la clase ListArr, que cuenta con un atributo de tipo NodoDato llamado "root" que representa el primer nodo de la lista. Cada NodoDato tiene un arreglo de tamaño fijo "arr" que almacena los valores de la lista y un entero "n" que representa el número de elementos que contiene dicho arreglo. Además, tiene un puntero "nextarr" que apunta al siguiente nodo de la lista.

Para la inserción de un elemento en la lista, se emplean los métodos "insert_left" e "insert_right" que insertan un elemento al principio o al final de la lista respectivamente. Si la cantidad de elementos en el nodo actual es menor al tamaño máximo del arreglo, se agrega el elemento al final de dicho arreglo. Si el arreglo está lleno, se crea un nuevo NodoDato con un arreglo vacío, se inserta el elemento en la primera posición de dicho arreglo, y se establece el puntero "nextarr" del nuevo nodo al nodo anterior.

Se puede obtener la cantidad de elementos en la lista mediante el método "size". Para buscar un elemento en la lista, se usa el método "find" que recorre la lista en busca del elemento deseado y devuelve verdadero si se encuentra y falso en caso contrario.

Además, se incluye un método llamado "maketree" que crea un árbol binario de búsqueda a partir de la información de la lista. Este árbol se representa mediante el nodo especial NodoResumen, que tiene cuatro punteros que apuntan a otros nodos NodoResumen. Los punteros "di" y "dd" apuntan al NodoResumen que representa los elementos del lado izquierdo y derecho del primer NodoDato de la lista, respectivamente. Los punteros "ri" y "rd" apuntan al NodoResumen que representa los elementos del lado izquierdo y derecho del último NodoDato de la lista, respectivamente.

La implementación de la operación "maketree" es compleja y su análisis no es trivial. Sin embargo, se puede afirmar que tiene un tiempo de ejecución en el peor caso de O(n log n), donde n es la cantidad de elementos en la lista, debido a que se recorren todos los nodos de la lista para construir el árbol, y luego se recorren los nodos del árbol para establecer los punteros de los nodos NodoResumen.

La complejidad de insertar un elemento en ListArr depende del número de elementos que ya se encuentran en la lista y del tamaño máximo del arreglo. Si la lista ya contiene n elementos y el tamaño máximo del arreglo es b, entonces:

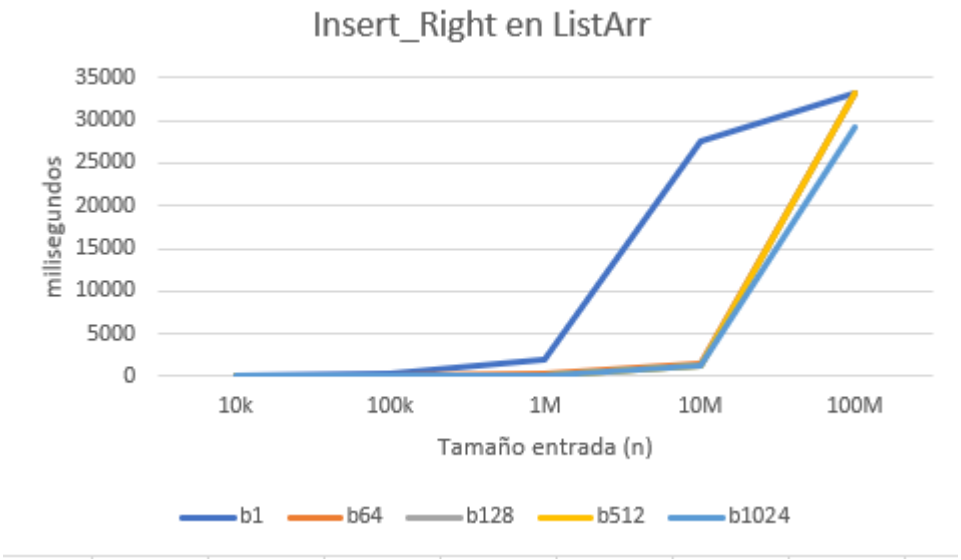
Para insert_left: si el primer NodoDato no está lleno, el tiempo de ejecución es O(b) debido al desplazamiento de los elementos del arreglo. Si el primer NodoDato está lleno, se crea un nuevo NodoDato y se establece el puntero nextarr al NodoDato anterior. Por lo tanto, el tiempo de ejecución es constante, O(1).

Para insert_right: si el último NodoDato no está lleno, el tiempo de ejecución y el espacio de almacenamiento serán menores en comparación con la implementación en la que el último NodoDato está lleno. Por lo tanto, la elección entre estas dos opciones dependerá de las necesidades específicas del programa y de los recursos disponibles en el sistema.

En cuanto a la estructura de datos para implementar una cola con prioridad, una opción común es utilizar un montículo binario, que es una estructura de datos basada en un árbol binario completo que cumple con la propiedad de ordenación, es decir, que el valor almacenado en cada nodo es mayor o menor que los valores almacenados en sus hijos.

En un montículo binario, el elemento de mayor prioridad se almacena en la raíz del árbol y se puede acceder rápidamente en O(1) tiempo. Además, cuando se inserta un nuevo elemento en el montículo, este se coloca en la posición adecuada en el árbol para mantener la propiedad de ordenación, lo que lleva a una complejidad de tiempo de O(log n) para las operaciones de inserción y eliminación.

Grafico de resultados experimentales:



Se observa que a mayor valor de b, el método Insert_Right tarda menos en realizarse y que a mayor tamaño de entrada va subiendo el tiempo necesario para realizarse.

Especificaciones utilizadas:

HP Laptop 15-dw1xxx

Nombre del dispositivo	LAPTOP-JH3T2MRU
Procesador	Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
RAM instalada	8,00 GB (7,81 GB utilizable)
Id. del dispositivo	A5379815-70ED-489C-9691-4B0881F1984B
Id. del producto	00327-30971-46824-AAOEM
Tipo de sistema	Sistema operativo de 64 bits, procesador x64
Lápiz y entrada táctil	La entrada táctil o manuscrita no está disponible para esta pantalla

Conclusión:

En conclusión, el uso de una estructura de datos como la lista enlazada puede ser beneficioso en ciertos casos donde se necesite una mayor flexibilidad en la inserción y eliminación de elementos, ya que permite ajustar el tamaño de la lista dinámicamente y no requiere la creación de un nuevo arreglo en caso de aumentar su tamaño.

En términos de complejidad temporal, tanto las listas enlazadas como los arreglos tienen sus ventajas y desventajas, dependiendo del tipo de operación que se esté realizando. En general, las listas enlazadas tienen una complejidad de O(n) para el acceso aleatorio y O(1) para la inserción y eliminación de elementos, mientras que los arreglos tienen una complejidad de O(1) para el acceso aleatorio y O(n) para la inserción y eliminación de elementos en posiciones arbitrarias. Cabe destacar que con find() no habrá mucha diferencia puesto que en todos los sistemas funciona de manera similar

Por lo tanto, la elección entre una lista enlazada y un arreglo dependerá del tipo de operaciones que se realicen con mayor frecuencia en la aplicación en cuestión.