PRÁCTICA 2

Estructura de datos

Grupo 201 GII

María Guzmán Valdezate
Guillermo López de Arechavaleta Zapatero

Contenido

Introducción

En esta práctica, hemos implementado una **estructura de datos de tipo cola**, la cual sigue el principio **FIFO (First In, First Out)**, donde el primer elemento en entrar es el primero en salir.

El objetivo de este informe es analizar la **complejidad algorítmica** de los métodos implementados, justificando su eficiencia según las operaciones realizadas. Se explicará cómo afectan al rendimiento y qué ventajas ofrece esta implementación en términos de optimización y uso eficiente de recursos.

Descripción y Análisis de Métodos

clear ()

Descripción: Elimina todos los elementos de la cola, dejando la cola vacía.

Complejidad algorítmica: No hay bucles en este método, solo modifica las referencias a los nodos y establece el tamaño en 0. Tiene O (1).

element ()

Descripción: Devuelve el primer elemento de la cola sin eliminarlo.

Complejidad algorítmica: Este método solo accede al primer nodo de la cola, tiene O (1).

contains (Object e)

Descripción: Comprueba si un elemento se encuentra en la cola.

Complejidad algorítmica: Debido al bucle do-while es de O (n).

circularIterator ()

Descripción: Devuelve un iterador circular que recorre la cola de forma continua.

Complejidad algorítmica: El iterador solo recorre los nodos de un ciclo, por lo que no tiene complejidad algorítmica.

hasNext ()

Descripción: Comprueba si el elemento tiene un siguiente elemento en la cola.

Complejidad algorítmica: Solo comprueba la primera vuelta, por lo que es O (1).

next ()

Descripción: Devuelve el siguiente elemento de la cola.

Complejidad algorítmica: Solo accede al nodo actual y avanza al siguiente, por lo que es O (1).

offer (T e)

Descripción: Inserta un elemento en la cola.

Complejidad algorítmica: Actualiza las referencias a los nodos e inserta un nuevo nodo al final de la cola. Es O (1).

poll ()

Descripción: Elimina el primer nodo de la cola y lo devuelve.1

Complejidad algorítmica: Solo modifica referencias, es O (1).

peek ()

Descripción: Devuelve el primer elemento de la cola sin eliminarlo.

Complejidad algorítmica: Solo accede al primer elemento, por lo que es O (1).

iterator ()

Descripción: Devuelve un iterador sobre los elementos de la cola.

Complejidad algorítmica: Solo crea un iterador que se inicializa con el primer nodo, por lo cual es O (1).

size ()

Descripción: Devuelve el tamaño de la cola.

Complejidad algorítmica: Utiliza el atributo size que simplemente, va almacenando el número de elementos de la cola. Es de O (1).

remove ()

Descripción: Elimina el último elemento devuelto por el iterador.

Complejidad algorítmica: Solo actualiza las referencias al eliminar un elemento, por lo cual es O (1).

Hay tres métodos (elements(), clear() y contains()) que no eran necesarios implementarlos, pero debido a las clases teóricas recibidas hemos decidido implementarlos enfocando el código a futuro, con una implementación más completa y correcta.

Conclusiones

La implementación de la cola ha demostrado ser altamente eficiente, ya que la mayoría de los métodos presentan una complejidad de O(1), lo que significa que operan en tiempo constante, independientemente del tamaño de la estructura.

El método contains (Object e) es el único con una complejidad O(n), debido a que necesita recorrer toda la estructura en el peor de los casos para verificar la existencia de un elemento. Esto implica que su rendimiento se ve afectado a medida que la cola crece en tamaño.

El uso de iteradores, incluyendo el iterador circular, permite recorrer los elementos de manera eficiente. Sin embargo, el iterador circular introduce una particularidad al reiniciar su recorrido indefinidamente, lo que puede ser beneficioso en ciertas aplicaciones donde se requiere un acceso continuo a los elementos.

Los resultados de este análisis confirman que esta implementación de la cola es altamente eficiente para operaciones básicas como inserción, eliminación y consulta del primer elemento. La correcta elección de una estructura de datos optimizada es clave para el rendimiento de los algoritmos, especialmente cuando se trabaja con grandes volúmenes de datos, donde diferencias en la complejidad pueden impactar significativamente la eficiencia y fluidez de los programas desarrollados.