Práctica 9

Estructura de datos

Grupo 201 GII

María Guzmán Valdezate

Guillermo López de Arechavaleta Zapatero

Contenido

[Introducción 3](#_Toc197358205)

[Descripción y Análisis de Métodos 4](#_Toc197358206)

[Conclusiones 6](#_Toc197358207)

## Introducción

En esta práctica se ha implementado una estructura de datos personalizada denominada ConjuntoLRU, que extiende AbstractSet e implementa la interfaz SortedSet. Su característica principal es aplicar una política de reemplazo LRU (Least Recently Used), eliminando el elemento menos recientemente accedido cuando se alcanza la capacidad máxima.

Para ello, se ha utilizado una estructura auxiliar (cacheLRUEnlazada) que almacena los elementos junto con un contador de accesos. El conjunto es capaz de actualizar internamente este contador cada vez que se accede a un elemento, bien sea por inserción o mediante iteración.

El objetivo del informe es analizar el diseño de nuestra implementación y describir el funcionamiento de esta estructura, analizar la eficiencia de sus métodos y valorar su comportamiento.

## Descripción y Análisis de Métodos

**add(E e)**

Descripción: Añade un elemento al conjunto. Si ya existe, actualiza su contador de acceso; si no existe y se ha alcanzado la capacidad, elimina el elemento menos recientemente accedido antes de insertarlo.

Complejidad algorítmica: O(n), ya que potencialmente hay que recorrer todos los elementos para identificar el menos usado.

**eliminarMenosUsado()**

*Descripción*: Método auxiliar clave para la política LRU. Se invoca automáticamente cuando el conjunto alcanza su capacidad máxima durante una inserción (add()). Identifica y elimina el elemento con el contador de acceso más bajo (menos recientemente usado) mediante un recorrido completo de la estructura.

*Complejidad algorítmica*: O(n), ya que requiere examinar todos los elementos para encontrar el mínimo contador.

**remove(Object o)**

Descripción: Elimina un elemento del conjunto si está presente.

Complejidad algorítmica: O(1), ya que se basa en una operación de eliminación sobre la estructura cacheLRUEnlazada.

**size()**

Descripción: Devuelve el número de elementos actualmente almacenados en el conjunto.

Complejidad algorítmica: O(1).

**first()**

Descripción: Devuelve el elemento menos recientemente accedido (menor contador).

Complejidad algorítmica: O(n), ya que requiere recorrer todos los elementos para encontrar el mínimo.

**last()**

Descripción: Devuelve el elemento más recientemente accedido (mayor contador).

Complejidad algorítmica: O(n), por el mismo motivo que first.

**headSet(E toElement)**

Descripción: Devuelve un subconjunto de elementos menos accedidos que el indicado.

Complejidad algorítmica: O(n), al requerir recorrer toda la estructura y comparar contadores.

**tailSet(E fromElement)**

Descripción: Devuelve un subconjunto con los elementos más accedidos que el indicado.

Complejidad algorítmica: O(n).

**subSet(E desde, E hasta)**

Descripción: Devuelve un subconjunto con los elementos cuyo número de acceso está entre los dos elementos especificados (inclusive el primero, exclusivo el segundo).

Complejidad algorítmica: O(n), por el recorrido completo y la doble condición de acceso.

**comparator()**

Descripción: Devuelve null ya que no se utiliza ningún comparador externo para el orden.

Complejidad algorítmica: O(1).

**iterator()**

Descripción: Devuelve un iterador que, al recorrer los elementos, actualiza el contador de acceso para reflejar su uso reciente.

Complejidad algorítmica: O(n), por la creación de la copia de las claves y la actualización durante la iteración.

## Conclusiones

La implementación de ConjuntoLRU demuestra cómo extender las estructuras estándar de Java permite crear conjuntos con comportamientos especializados, en este caso mediante una política LRU basada en contadores de acceso. El diseño actual ofrece una solución conceptualmente clara y funcionalmente correcta para la gestión automática de capacidad.

El análisis revela que la principal limitación reside en la operación de eliminación del menos usado (O(n)), que afecta directamente al rendimiento de las inserciones cuando el conjunto está lleno. No obstante, para conjuntos de tamaño moderado o en aplicaciones donde las operaciones de consulta predominan sobre las inserciones, esta implementación resulta perfectamente viable.

Como ventajas destacables, la solución:

1. Proporciona una gestión automática de memoria según uso
2. Mantiene una interfaz coherente con SortedSet
3. Ofrece mecanismos para consultar elementos por antigüedad de acceso

Sería recomendable, para futuras versiones, explorar optimizaciones que reduzcan la complejidad de la eliminación, posiblemente mediante el mantenimiento de referencias auxiliares a los elementos más y menos recientemente usados. Esto permitiría mantener la elegancia del diseño actual mientras se mejora el rendimiento en escenarios de alta frecuencia de inserciones.