Sergio Barbero Báscones

Estructuras de Datos

Práctica 3

UNIVERSIDAD DE BURGOS

Contenido

1. INTRODUCCION

Se solicita al alumno la implementación de un nuevo tipo de datos en java que permita implementar un conjunto de datos que deberá funcionar como una caché de datos, con un algoritmo de reemplazo LRU que tendrá en cuenta el momento de ACCESO a cada dato (no solo el momento de inserción). Para realizar esta tarea, la clase deberá implementar el interfaz java.util.SortedSet.

Se recomienda así mismo que se extienda la clase java.util.AbstractSet, para sobrescribir solo los métodos necesarios. La clase creada deberá permitir almacenar objetos de cualquier clase (será por tanto una clase genérica).

De esta forma, el conjunto tendrá un tamaño máximo que no se podrá superar. Al almacenar o acceder a un elemento se adjuntará al mismo una información de acceso que permitirá mantener un orden de acceso entre todos. Se podrá así identificar los elementos menos utilizados para que si al insertar se va a superar el tamaño máximo, se puedan eliminar éstos para mantener el tamaño. Se considerará un conjunto ordenado y el orden a respetar para los elementos contenidos será el del último acceso.

2. ESTRUCTURA GENERAL

Debemos generar dos clases anidadas, la clase padre extiende de AbstractSet e implementa de SortedSet y la hija implementará de Iterator, de esta manera reescribiremos los métodos next y hasnext, debido a que cada vez que accedemos a un elemento del conjunto modificaremos dicho conjunto, y la implementación original no nos permitiría hacer eso.

3. DESCRIPCIÓN DE LOS MÉTODOS

Clase ConjuntoLRU

- `Boolean add(T e):` Añade un nuevo elemento al conjunto, si el elemento ya está contenido entonces reemplazaremos dicho elemento asociándole el valor mas reciente, si el elemento no esta contenido entonces debemos de comprobar si añadir el nuevo elemento sobrepasaría nuestra capacidad del cojunto, si no lo sobrepasa simplemente añadiremos dicho elemento asociándole el valor mas reciente de la misma manera, si lo sobrepasa entonces debemos borrar el elemento mas antiguo del conjunto y añadir el nuevo del mismo modo que antes.

Complejidad algorítmica: O(1)

Primero nos encontramos con una estructura condicional en la que se comprueba si un elemento está contenido en la tabla de tipo HashMap, containsKey(T e) es de complejidad O(1) en los HashMap debido a que va referenciado a través de un HashCode, al tratarse de una estructura condicional seleccionaremos el camino más complejo, En este caso es de igual complejidad ambos, ambos son O(1) también.

- `Boolean remove(Object e):` Este método nos elimina un elemento del conjunto si está presente, devolviendo true, en caso contrario devuelve false.

  Complejidad algorítmica: O(1)

  Del mismo modo que antes tenemos una invocación a containsKey(T e) en una estructura condicional de igual complejidad por ambos caminos.

- `Comparator<? super T> comparator():` Este método nos instancia un comparador para poder comparar elementos del conjunto.

  Complejidad algorítmica: O(1)

- `SortedSet<T> subSet(T fromElement, T toElement): Este método nos devuelve un set comprendido entre dos elementos del conjunto.`

  Complejidad algorítmica: O(n)

  Nos encontramos con una invocación al método ordenarMapa() Que lo que hace es generar un mapa inverso al que tenemos, con Integer como clave y como T como valores. Analizaremos posteriormente este método, pero esta llamada es la que proporciona al método la categoría de O(n).

- `SortedMap<Integer, T> ordenaMapa():` Este método nos devuelve un mapa ordenado a través del Integer, de esta manera podemos observar los elementos de más reciente a más antiguo.

  Complejidad algorítmica: O(n)

  Debido a que en este método iteramos el mapa de principio a fin para ordenar los datos, por lo que la complejidad será de O(n)

- `SortedSet<T> headSet(T toElement):` Este método nos devuelve un set comprendido entre el elemento más antiguo del conjunto y el elemento proporcionado.

  Complejidad algorítmica: O(n)

  Por la misma razón que en el método subset esté método es de complejidad O(n).

- `SortedSet<T> tailSet(T fromElement):` Este método nos devuelve un set entre un elemento proporcionado y el final del conjunto.

  Complejidad algorítmica: O(n)

  Por las mismas razones que en el método subset y headset

- `T first():` Este método nos devuelve el elemento más antiguo del conjunto.

Complejidad algorítmica: O(n)

Debido a la invocación a ordenaMapa esté método es O(n)

- `T last():` Este método nos devuelve el elemento más reciente del conjunto.

  Complejidad algorítmica: O(n)

- `Iterator<T> iterator():` Este método devuelve una instancia del iterador del conjunto LRU

  Complejidad algorítmica: O(1)
  Simplemente nos devuelve una instancia.

- `void clear():` Limpia el conjunto de datos.

  Complejidad algorítmica: O(1)

  Llama a clear de HashMap, que es O(1)

- `int size():` Devuelve el tamaño del conjuntoLRU

  Complejidad algorítmica: O(1)

  Invoca a size() de hashmap.

Clase IteradorLRU

- `boolean hasNext():` Nos muestra mediante un valor booleano si el conjunto tiene elemento siguiente o no.

  Complejidad algorítmica: O(1)

- `T next():` Nos muestra el valor siguiente y lo coloca como valor mas reciente, aumentando el contador y modificando el conjunto

  Complejidad algorítmica: O(1)

  Hace un get del elemento siguiente en la lista.