PRACTICA 3 ES-DAT

[Subtítulo del documento]

VICTOR GONZALEZ DEL CAMPO- JUAN GARCIA LOPEZ

CONTENIDO

OBJETIVOS

Con esta práctica se pretende que el alumno se familiarice y comprenda:

1. Estructura de la jerarquía adicional del Java Collections Framework

Se estudia la forma de extender el Framework por medio de interfaces adicionales (Map y utilizando un nuevo interfaz Table).

2. Cómo se implementa una estructura de datos de forma completa.

3. El coste computacional que implica realizar una estructura de datos con una estructura interna determinada.

Adicionalmente se trabajará con la Genericidad de Java, que permite definir los tipos de datos a almacenar en una colección. Es una característica que mantienen todos los componentes del Collections Framework.

PROBLEMA PLANTEADO

Se pretende generar un tipo de Cola nuevo para que quede a disposición de otros programadores dentro del Java Collections Framework.

El objetivo es programar una Estructura de Datos en Tabla que permita trabajar con estructuras con dos claves (ternas de datos) de forma más natural que las que se incluyen por defecto en el Java Collections Framework. Adicionalmente se desea que esta estructura se modele como un mapa de mapas (o multimapa), para que el alumno se familiarice también con el funcionamiento de éstas. Para esta implementación se pide la creación de una clase de nombre HashMapTable, que deberá extender el interfaz es.ubu.gii.edat.Table, que se adjunta al enunciado.

Se incluirá en el paquete es.ubu.gii.edat.pr03_2324. Se pretende crear una estructura similar a un Map en Java, pero que permita almacenar valores asociados a una pareja de claves, en lugar de a una sola. Conceptualmente, se puede entender como una matriz en la que la primera clave nos da la fila que se está buscando y la segunda nos permite determinar la columna. Por último, el valor que se pretende obtener estará en la celda que se encuentra en la intersección de ambas. Se puede ver un esquema más abajo (Ilustración 1). Para implementar esta estructura, se construirá internamente como un mapa de mapas o un mapa a dos niveles.

En primer lugar, se instanciará un mapa cuyas claves serán las claves primarias a almacenar o “claves de fila” y sus valores correspondientes sean a su vez otro mapa. En este caso, el mapa estará definido incluyendo las claves secundarias o “claves de columna” como las claves de estos mapas y los valores a almacenar como los valores del mapa. De esa forma, “clave de fila” (o primaria) se empleará en el mapa principal para discriminar cual es el mapa secundario sobre el que tenemos que buscar y la “clave de columna” (o secundaria) permite acceder a la posición correspondiente al valor en el mapa secundario concreto (Ilustración 2).

CODIGO

HashMapTable

DESCRIPCIÓN

La clase HashMapTable es una implementación de la interfaz Table en Java. Esta clase utiliza una estructura de datos HashMap para almacenar los datos. La tabla asocia un par de claves, denominadas fila y columna, con un único valor. Esta tabla puede ser dispersa, teniendo solo una fracción de elementos definidos por su fila y columna ocupados por un valor. La clase HashMapTable tiene métodos para agregar, eliminar y obtener valores asociados con un par de claves de fila y columna.

También proporciona métodos para verificar si la tabla contiene un par de claves o un valor específico, y para obtener una vista de todas las asociaciones que coinciden en emplear la misma clave de fila o columna.

Además, la clase HashMapTable incluye una clase interna CellImpl que implementa la interfaz Cell. Esta clase interna encapsula los datos correspondientes a las asociaciones que se incluyen en la clase HashMapTable. Cada celda contiene una clave de fila, una clave de columna y un valor, y proporciona métodos para obtener y establecer el valor, así como para comparar celdas y calcular su hash code.

En resumen, la clase HashMapTable proporciona una implementación eficiente de una tabla bidimensional dispersa, permitiendo el acceso y la manipulación de datos asociados con un par de claves de fila y columna.

ANALISIS COMPLEJIDAD

put(R row, C column, V value): Esta función inserta un valor en el mapa. En el caso promedio, la inserción en un HashMap tiene una complejidad de tiempo de O(1) debido a su estructura de hash. Sin embargo, en el peor de los casos, cuando ocurre una colisión de hash y se debe recorrer una lista enlazada o un árbol rojo-negro asociado a la posición del hash, la complejidad puede ser O(n), donde n es el número de elementos en el mapa.

Complejidad en el peor de los casos: O(n).

remove(R row, C column): Similar a put, esta función elimina un elemento del mapa. En el caso promedio, la eliminación en un HashMap tiene una complejidad de tiempo de O(1). Sin embargo, en el peor de los casos, puede ser O(n) debido a las colisiones de hash y la necesidad de recorrer una lista enlazada o un árbol rojo-negro.

Complejidad en el peor de los casos: O(n).

get(Object row, Object column): Esta función obtiene un valor del mapa. En el caso promedio, la búsqueda en un HashMap tiene una complejidad de tiempo de O(1). En el peor de los casos, puede ser O(n) debido a colisiones de hash y la necesidad de recorrer una lista enlazada o un árbol rojo-negro.

Complejidad en el peor de los casos: O(n).

containsKeys(Object row, Object column): Esta función verifica si el mapa contiene una clave específica. En el caso promedio, tiene una complejidad de tiempo de O(1), ya que se basa en la función get. En el peor de los casos, puede ser O(n) debido a colisiones de hash y la necesidad de recorrer una lista enlazada o un árbol rojo-negro.

Complejidad en el peor de los casos: O(n).

containsValue(V value): Esta función verifica si el mapa contiene un valor específico. Tiene una complejidad de tiempo de O(n) ya que debe recorrer todos los elementos del mapa.

Complejidad en el peor de los casos: O(n).

row(R rowKey): Esta función devuelve un mapa de columnas para una fila específica. En el caso promedio, tiene una complejidad de tiempo de O(1) ya que se basa en la función get. En el peor de los casos, puede ser O(n) si la fila no existe y debe recorrer todos los elementos del mapa.

Complejidad en el peor de los casos: O(n).

column(C columnKey): Esta función devuelve un mapa de filas para una columna específica. Tiene una complejidad de tiempo de O(n) ya que debe recorrer todas las filas del mapa para construir el mapa de columnas.

Complejidad en el peor de los casos: O(n).

cellSet(): Esta función devuelve una colección de todas las celdas del mapa. Tiene una complejidad de tiempo de O(n) ya que debe recorrer todos los elementos del mapa para construir la lista de celdas.

Complejidad en el peor de los casos: O(n).

size(): Esta función devuelve el tamaño del mapa. Tiene una complejidad de tiempo de O(1) ya que simplemente devuelve el valor de la variable size.

Complejidad en el peor de los casos: O(1).

isEmpty(): Esta función verifica si el mapa está vacío. Tiene una complejidad de tiempo de O(1) ya que simplemente verifica si el tamaño del mapa es cero.

Complejidad en el peor de los casos: O(1).

clear(): Esta función elimina todos los elementos del mapa. Tiene una complejidad de tiempo de O(n) ya que debe eliminar todos los elementos del mapa.

Complejidad en el peor de los casos: O(n).

Subclase CellImpl

DESCRIPCIÓN

La clase CellImpl es una clase interna de HashMapTable que implementa la interfaz Cell. Esta clase encapsula los datos correspondientes a las asociaciones que se incluyen en la clase HashMapTable.

La clase CellImpl tiene tres campos genéricos: row, column y value, que representan la clave de fila, la clave de columna y el valor de una celda respectivamente. El constructor de la clase toma tres parámetros que corresponden a estos campos y los asigna a las variables de instancia correspondientes.

La clase proporciona métodos para obtener y establecer el valor de la celda (getValue y setValue), así como para obtener las claves de fila y columna (getRowKey y getColumnKey). El método setValue también devuelve el valor antiguo de la celda después de establecer el nuevo valor.

Además, la clase CellImpl sobrescribe los métodos equals y hashCode de la clase Object. El método equals compara las claves de fila, columna y valor de las celdas para

determinar si son iguales. El método hashCode calcula el hash code de la celda basándose en las claves de fila, columna y valor.

ANALISIS COMPLEJIDAD

La clase ***CellImpl*** es una clase interna de HashMapTable que implementa la interfaz Cell. Esta clase encapsula los datos correspondientes a las asociaciones que se incluyen en la clase HashMapTable. Aquí está el análisis de complejidad de cada función:

getRowKey(): Esta función tiene una complejidad de tiempo de O(1), ya que simplemente devuelve el valor de la variable de instancia row.

getColumnKey(): Esta función también tiene una complejidad de tiempo de O(1), ya que simplemente devuelve el valor de la variable de instancia column.

getValue(): Esta función tiene una complejidad de tiempo de O(1), ya que simplemente devuelve el valor de la variable de instancia value.

setValue(V value): Esta función tiene una complejidad de tiempo de O(1), ya que simplemente establece el valor de la variable de instancia value y devuelve el valor antiguo.

equals(Object o): Esta función tiene una complejidad de tiempo de O(1), ya que simplemente compara las claves de fila, columna y valor de las celdas.

hashCode(): Esta función tiene una complejidad de tiempo de O(1), ya que simplemente calcula y devuelve el hash code de la celda.

CONCLUSION

La práctica se centró en el estudio y la implementación de estructuras de datos adicionales en el marco de colecciones de Java, específicamente la extensión del framework con interfaces como Map y la creación de una nueva interfaz llamada Table. El objetivo principal fue comprender cómo implementar una estructura de datos completa y el costo computacional asociado a cada operación en función de la estructura interna de datos.

Para abordar este objetivo, se desarrolló la clase HashMapTable, que implementa la interfaz Table y utiliza una estructura de datos HashMap para almacenar datos asociados a pares de claves de fila y columna. Esta implementación permitió manipular eficientemente datos dispersos en una tabla bidimensional.

El análisis de complejidad reveló que la mayoría de las operaciones en la clase HashMapTable tienen una complejidad de tiempo de O(1) en el caso promedio, gracias a la eficiencia de las operaciones de inserción, eliminación y búsqueda en un HashMap. Sin embargo, en el peor de los casos, estas operaciones pueden tener una complejidad de

tiempo de O(n) debido a colisiones de hash que requieren recorrer estructuras secundarias, como listas enlazadas o árboles rojo-negro.

Además, se implementó una clase interna llamada CellImpl, que encapsula los datos de cada celda en la tabla. El análisis de complejidad de esta clase reveló que las operaciones en la clase CellImpl tienen una complejidad de tiempo de O(1), lo que significa que son eficientes y no dependen del tamaño de la tabla.

En conclusión, la práctica proporcionó una comprensión profunda de la implementación de estructuras de datos en Java, así como una apreciación del impacto del diseño interno de una estructura de datos en su eficiencia computacional. El enfoque en el análisis de complejidad permitió evaluar el rendimiento de las operaciones en diferentes escenarios, lo que es crucial para diseñar y seleccionar la estructura de datos adecuada para diferentes aplicaciones.