Universidad Politécnica de Lleida

Practica 5

Binarytree

Author:
Martin, Francisco Manuel
Martinez, Victor

DNI:
48057095k
78100640T

20 de diciembre de 2019

Índice

1. Introducción

2. Tarea 1: Implementación de los arboles binarios

2.1. Constructores

En esta sección solo vamos a comentar un pequeño detalle sobre el tercer constructor.
Tratamos **left** y **right** de manera que, si uno de ellos es null inicializamos la instancia de Node correspondiente a null.
De esta manera evitamos un NullPointerException en caso de intentar acceder al atributo root del arbol en cuestión.

2.2. método equals

2.2.1. equals de LinkedBinaryTree

En primer lugar comprobamos que el objecto de tipo *Object* pasado como parámetro apunte a una instancia de LinkedBinaryTree Si es así, realizamos un cast y ejecutamos el método estático *recEquals* de la clase Node que comprobará recursivamente si los *roots* de los arboles són iguales.

2.2.2. equals de Node

Aunque este método no lo usamos en LinkedBinaryTree ya que directamente empleamos el método *recEquals*, nunca está de más definir el método equals dentro de una classe. En este caso comprobamos que el parámetro *obj* apunte a una instancia de Node y ejecutamos el método *recEquals*

2.2.3. método recEquals

En primer lugar comprobamos si alguno de los parámetros apunta a null, si es así devolvemos el resultado de igualar los dos parámetros (si uno es null el otro también debe serlo para cumplir la igualdad). En caso contrario seguimos comprobando si los dos nodos són iguales (Contienen el mismo elemento y sus nodos derecho e izquierdo són también iguales).

3. Tarea 2: Recorridos iterativos en árboles binarios

3.1. Interfaz Traversals

¿Qué diferencias provocaría que la interfaz fuera genérica y los métodos no?
En caso de que los métodos sean genéricos y la clase no, podremos usar los métodos con cualquier BinaryTree sin importar de que tipo sean los elementos que contiene.

En cambio, si implementamos la interfaz de manera genérica y los métodos no-genericos, solo podremos usar dichos métodos con arboles binarios que contengan elementos del mismo tipo con el cual hemos inicializado la instancia de **Traversals**,
por lo cual necesitaremos crear una instancia de **Traversals** para cada arbol binario que definamos con un tipo distinto.

3.2. Clase IterativeTraversals

En esta clase implementamos los diferentes recorridos en profundidad sobre arboles binarios de manera iterativa.
Los stacks con los que trabajamos contendrán elementos de tipo BinaryTree, y así iremos comprobando si están vacíos o no para realizar las operaciones.
En la explicación de los siguientes métodos voy a referirme algunas veces a *el último elemento introducido el en stack* como el elemento que contiene el root del arbol que es el objeto con el que realmente trabajamos.

3.2.1. Preorder

Como debemos recorrer los nodos con el patrón *Parent*, *left*, *right*, en este algoritmo empezamos añadiendo al stack el arbol.
Vamos añadiendo a la lista el último elemento introducido en el stack y llenando el stack con los hijos derecho e izquierdo del elemento, tal y como lo plantearíamos en el algoritmo recursivo.

3.2.2. Inorder

La idea del algoritmo implementado para éste recorrido (*left*, *parent*, *right*) se basa en posicionarse en el elemento más a la izquierda del árbol y, en cuanto lo encontramos lo añadimos a la lista, añadimos a su padre (que estamos seguros de que será el último elemento añadido en el stack) y visitamos el hijo derecho.
Seguimos hasta que el stack esté vacío y el último arbol visitado esté vacío también.

3.2.3. Postorder

El algoritmo implementado en éste recorrido (*left*, *right*, *parent*) se basa en una idea un poco diferente a las anteriores.
Se trata en mirar el recorrido de manera inversa, por lo que recorreremos de *Parent* a *right* a *left* y iremos añadiendo los elementos al principio de la lista y no al final. De esta manera el código nos queda muy simple y prácticamente igual a la implementación de PreOrder solo que cambiando el orden en que añadimos los hijos.
Cabe destacar que es necesario que la lista sea de tipo *LinkedList* para que las inserciones sean óptimas, en caso de ser un *ArrayList* estaríamos aumentando el coste del algoritmo prácticamente de manera exponencial.

4. Tarea 3: Reconstrucción del árbol binario

La idea en la resolución de éste método se basa en los dos argumentos siguientes:

1. El último elemento de la lista que contiene los elementos en postOrden siempre será el root del árbol.
2. Debemos encontrar la posición de ese elemento en la lista de inOrden para de esta manera poder separar los elementos que se encuentran a su izquierda y a su derecha como árbol izquierdo y árbol derecho.

Siguiendo estas dos premisas implementamos el método de manera recursiva mediante sublistas.

5. Conclusiones

Durante la resolución del laboratorio hemos aprendido bastante sobre como trata Java los genéricos en tiempo de compilación (mediante el proceso de *type erasure*). Como java en tiempo de ejecución no tiene información sobre los tipos genéricos, es un error realizar un cast de la siguiente manera:

```
Node<E> node = (Node<E>) obj;
```

Y en su lugar debemos utilizar comodines:

```
Node<?> node = (Node<?>) obj;
```

De esta manera evitamos un posible error de ejecución.

En la implementación del método *postAndIn* estubimos debatiendo sobre si implementarlo utilizando índices o sublistas.
En el primer caso, la idea se basaba en utilizar 4 índices (2 índices indicando el principio y el final del segmento que queríamos tratar sobre la lista postOrder y 2 índices indicando los mismos parámetros sobre la lista inOrder).
En el segundo caso simplemente vamos reduciendo los segmentos a tratar generando sublistas, por lo que nos parecía que era más ineficiente que el primer caso.
En cambio, después de adentrarnos en el codigo fuente de la clase ArrayList vimos que el método sublist no genera una lista nueva, sino que devuelve una vista sobre la lista original, por lo que siempre trabajamos sobre el mismo objeto y de una manera muy parecida al primer caso que hemos comentado en el que trabajamos siempre sobre las mismas listas acotando segmentos mediante índices.
El método sublist de la clase LinkedList (heredado de la clase AbstractList) se basa en la misma idea, pero el coste de realizar la operación get va a ser mayor, ya que cuando lo hagamos sobre una sublista, realmente se va a realizar sobre la lista original por lo que el coste de realizar dicho get siempre va a ser O(n).