PRÁCTICA 10

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

El objetivo principal de esta práctica es profundizar en el manejo de estructuras jerárquicas, aplicando conceptos fundamentales como la recursividad y la extensión de clases abstractas. Para ello, se ha optado por emplear una representación basada en tablas hash, aprovechando la flexibilidad de la clase AbstractMap para almacenar la relación entre nodos e identificar su jerarquía.

A lo largo de este informe, se detallan las funcionalidades implementadas en el árbol, tales como la inserción y eliminación de nodos, la obtención de descendientes y ancestros, así como los recorridos en anchura. Además, se incluye un análisis de la complejidad algorítmica de cada método, evaluando su eficiencia y justificando el enfoque empleado en cada caso.

Mediante el desarrollo de esta estructura arbórea, se refuerzan los conocimientos adquiridos en el ámbito de las estructuras de datos, brindando una comprensión más profunda sobre la organización y manipulación de conjuntos de datos jerárquicos.

## Descripción y Análisis de Métodos

**put(E hijo, E padre)**

Descripción: Inserta un nodo hijo con su padre en el árbol. Si el hijo ya existe devuelve el padre anterior, si no, inserta el hijo y devuelve null.

Complejidad algorítmica: Debido a que dos operaciones son un ContainsKey(), que es O(n) ya que, en el peor de los casos, tiene que recorrer todo el mapa. Por la regla de la suma la complejidad algorítmica total es de O(n).

**remove(Object objeto)**

Descripción: Elimina un elemento del árbol, en caso de tener hijos los pasan a ser hijos del padre del elemento eliminado.

Complejidad algorítmica: Al igual que put, también tiene que recorrer el mapa con el ContainsKey. Luego tiene que recorrer toda la lista de descendientes, por lo que, por la regla de la suma el total es O(n).

**descendants(E elemento)**

Descripción: Devuelve una lista con todos los descendientes de un nodo.

Complejidad algorítmica: O(n) porque en el peor de los casos tiene que recorrer todos los descendientes ya que llama al método obtenerDescendientes.

**obtenerDescendientes(E elemento, List<E> descendientes)**

Descripción: Busca los descendientes de un elemento del árbol.

Complejidad algorítmica: En el peor de los casos, tiene que recorrer todos los nodos del árbol para encontrar los descendientes, lo que da una complejidad de O(n).

**ancestors(E elemento)**

Descripción: Devuelve una lista con todos los ancestros de un nodo.

Complejidad algorítmica: O(n) porque en el peor de los casos tiene que recorrer todos los descendientes ya que llama al método obtenerAncestros.

**obtenerAncestros(E elemento, List<E> ancestros)**

Descripción: Busca los ancestros de un elemento del árbol.

Complejidad algorítmica: O(n) donde n es la altura del nodo (en el peor caso puede ser O(n), pero se debe justificar mejor).

**depth(E elemento)**

Descripción: Calcula de profundidad de un nodo, es decir, la distancia que hay desde ese nodo hasta la raíz.

Complejidad algorítmica: O(n) porque llama al método ancestors, que en el peor de los casos tiene que recorrer todos los ancestros.

**height(E elemento)**

Descripción: Calcula la altura de un elemento, es decir, el camino más largo desde el nodo hasta una de sus hojas.

Complejidad algorítmica: O(n) porque en el peor de los casos tiene que recorrer todos los nodos.

**breadthTraverse()**

Descripción: Devuelve una lista con el recorrido en anchura del árbol.

Complejidad algorítmica: Lo primero que hace es recorrer la lista de claves, por lo que es O(n). Luego se recorre el mapa una vez por cada nivel del árbol (altura), y en cada iteración se revisan todos los nodos (n). Por tanto, **es O(n·m)**, que en el peor caso es **O(n²)** por la regla de la multiplicación. Por la regla de la suma, con el anterior podemos ver que es O(n2).

**entrySet()**

Descripción: Devuelve el conjunto de entradas del mapa.

Complejidad algorítmica: O(1) porque el acceso directo al conjunto de entradas es constante.

## Conclusiones

La implementación del ArbolTabulado ha permitido poner en práctica conceptos esenciales de las estructuras arbóreas en el ámbito de la programación en Java. Durante el desarrollo, se han abordado técnicas de recursividad, manipulación de colecciones y gestión de relaciones padre-hijo en estructuras jerárquicas.

Se han implementado métodos clave para la inserción, eliminación y consulta de elementos, garantizando que el árbol mantenga una estructura consistente. Además, se ha realizado un análisis de complejidad algorítmica para cada método, identificando los casos más costosos computacionalmente y proponiendo mejoras cuando sea posible.

Esta práctica no solo ha facilitado la comprensión teórica de los árboles en estructuras de datos, sino que también ha reforzado habilidades prácticas en el manejo del Java Collections Framework, ofreciendo un enfoque integral a la gestión de jerarquías en entornos de programación.