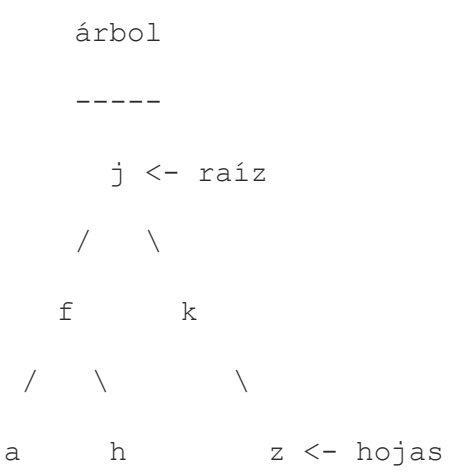
Árboles Binarios

Introducción

A diferencia de las matrices, las listas enlazadas, las pilas y las colas, que son estructuras de datos lineales, los árboles son estructuras de datos jerárquicas.

Nomenclatura

El nodo superior se llama *raíz* del árbol. Los nodos que están directamente debajo de un nodo se llaman hijos. El nodo inmediatamente superior, se llama padre. Por ejemplo, **a** es un hijo de **f** y **f** es el padre de **a**. Los nodos sin hijos se llaman hojas.



Árboles binarios

Es un tipo de árbol cuyos nodos tienen como máximo 2 hijos. Como cada nodo en un árbol binario sólo puede tener 2 hijos, generalmente los nombramos hijo izquierdo e hijo derecho.

Ejercicio 1. Implementación de BinaryOrderedTree

Vamos a implementar un árbol binario en el que los elementos se mantengan ordenados. Dado un Nodo, su hijo izquierdo contiene un valor menor y el hijo derecho contiene un valor mayor.

Vamos a utilizar genéricos para que nuestro BinaryOrderedTree<E extends Comparable> pueda funcionar con cualquier clase E que se pueda comparar.

BinaryOrderedTree tendrá una clase interna privada Node<E extends Comparable> con los atributos "E value", "Node<E> left" y "Node<E> right". No queremos que la gente tenga acceso directo a los nodos y la líe creando ciclos. BinaryOrderedTree tendrá el atributo "Node<E> root" así como "int numberOfElements".

Los métodos se van a ir implementando a lo largo de los siguientes ejercicios. Si os parece complicado trabajar con genéricos, haz el BinarySortedTree con Integer y luego ya introducirás los genéricos.

Ejercicio 2. Creación del árbol y sacar por pantalla.

Al insertar un elemento en nuestro árbol deberemos: Si no existe la raíz, la creamos y asignamos el valor. Luego, para los siguientes elementos, comenzamos con la raíz. Si el nuevo valor es mayor que del nodo actual, entonces nos movemos al subárbol de la derecha, si es menor nos movemos al subárbol de la izquierda. Si no hay tal hoja entonces creamos una y ponemos el valor.

Implementa el método **boolean add(E element)** que añade un elemento al árbol. Devuelve false si el elemento ya existía y por lo tanto no se puede añadir. Deberías hacerlo no recursivo.

Implementa el método **void print(PrintStream out)** que imprima en el PrintStream (System.out es un PrintStream) la estructura del árbol en pre-orden (raíz, hijo izquierdo, hijo derecho) añadiendo una tabulación por cada nivel del árbol. Esta función es más sencilla en recursivo. Por ejemplo:

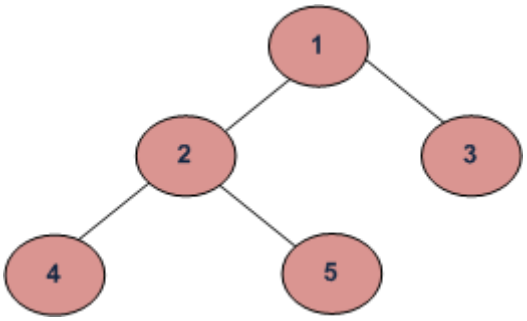
```
|Raíz
  |a
    |a
    |b
  |b
```

El árbol de la siguiente figura se ha formado a partir de la siguiente lista de palabras: "EN UN LUGAR DE LA MANCHA DE CUYO NOMBRE NO QUIERO ACORDARME".

Crea un BinaryOrderedTree<String>, añade cada palabra de la cadena, imprime el árbol por pantalla y comprueba que la estructura es la misma.

Ejercicio 3. Recorrido de árboles.

A diferencia de las estructuras de datos lineales (Arrays, listas enlazadas, colas, pilas, etc.) que tienen solo una forma lógica de recorrerse, los árboles pueden recorrerse de diferentes maneras.



Si tomamos el árbol anterior como ejemplo, los posibles recorridos serían:

a) Inorden (izquierda, raíz, derecha): 4 2 5 1 3
b) Preorden (raíz, izquierda, derecha): 1 2 4 5 3
c) Postorden (Izquierda, Derecha, Raíz): 4 5 2 3 1

Crea el método "**List<E> inorden()**" que devuelva un List con los elementos del árbol en inorden. Está función es más sencilla en recusivo.

Podemos probarlo con el ejemplo del Ejercicio 2.

Ejercicio 4. Búsqueda de elementos.

Para buscar un elemento comenzamos en la raíz. Comparamos el elemento a buscar con el valor del nodo actual. Si el elemento es mayor, continuamos a través del subárbol derecho, si es menor, continuamos a través del subárbol izquierdo, si es igual, lo hemos encontrado. Si alcanzamos una hoja sin haber encontrado el elemento, entonces ese elemento no está presente.

Implementa el método "**E getElement(E element)**" que busca element en el árbol y lo devuelve, si no está devuelve null. A partir de este método implementa también **boolean contains(E element)**.

Para probar que funciona podemos utilizar BinaryOrderedTree<Integer>, crea algunos tests para probar automáticamente si funciona. Por ejemplo, añade "5,2,3,8,6,9,11" y busca cada uno de los elementos, deberían aparecer. Busca "1,4,7,10,12" no deberían estar.

Ejercicio 5. Comparación con otras colecciones.

Sería muy fácil hacer que nuestro BinaryOrderedTree cumpliera con el interfaz Collection<E>. La forma más rápida (aunque no completa) es hacer que BinaryOrderedTree herede de AbstractCollection<E>. El método iterator déjalo en blanco, size es sencillo. add y contains lo tenemos.

Implementa la función long testContainsPerformance(Collection<Integer> col). La función añadirá en un "Set<Integer> allNumbers" números al azar (entre 0 e Integer.MAX_VALUE) hasta que el set contenga dos millones de elementos. Transforma el Set<Integer> a un ArrayList<Integer> allIntegers. Creará otro ArrayList<Integer> toInsert en el dónde se inserten elementos de allNumbers con una probabilidad del 50%. Inserta todos los elementos de toInsert en col. Cronometra cuantos milisegundos se tarda en comprobar si los elementos de allIntegers están en col (utiliza el método contains). Puedes utilizar System.currentMilllis para coger el tiempo antes y después de esta última parte.

Comprueba cuánto tardan en ejectuarse para una ArrayList<Integer>, LinkedList<Integer>, un TreeSet<Integer> y nuestro BinaryOrderedTree<Integer>. ¿Cuáles son tus conclusiones?