Esercitazione extra su alberi binari di ricerca

Esercizio 1. Date le classi

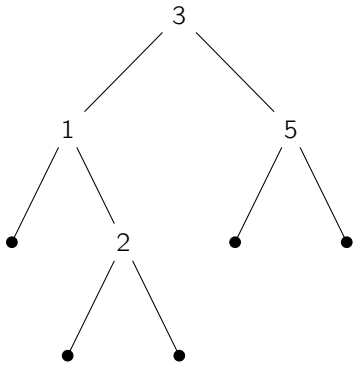
```
import java.util.*;
// classe astratta che definisce le operazioni su alberi binari
public abstract class Tree {
    public abstract boolean empty();
    public abstract int max();
    public abstract boolean contains(int x);
    public abstract Tree insert(int x);
    public abstract Tree remove(int x);
    public abstract int depth();
    // NUOVI METODI
    public abstract int size();
    public abstract int sum();
    public abstract boolean contains(int x, int n);
    public abstract boolean balanced();
    public abstract Tree filter_le(int x);
    public abstract int get(int i);
}
```

modificare le classi `Leaf` e `Branch` viste a lezione per implementare i nuovi metodi di `Tree`. In particolare, dato un oggetto `t` di tipo `Tree`, deve essere possibile eseguire le seguenti operazioni:

- `t.size()` ritorna il numero di elementi in `t`.
- `t.sum()` ritorna la somma di tutti gli elementi dell'albero `t`.
- `t.contains(x, n)` ritorna `true` se l'elemento `x` è raggiungibile dalla radice di `t` con un cammino lungo al massimo `n`. Suggerimento: è una semplice variante del metodo `contains` visto a lezione.
- `t.balanced()` ritorna `true` se `t` è bilanciato, ovvero se in ogni diramazione di `t` la differenza tra le profondità dei due sotto-alberi è al massimo 1 ed entrambi i sotto-alberi sono a loro volta bilanciati.
- `t.filter_le(x)` ritorna l'albero contenente tutti gli elementi di `t` che sono minori o uguali a `x` senza modificare `t` e assumendo che `t` sia un albero binario di ricerca. Usare l'ipotesi che `t` sia un albero binario di ricerca per limitare il numero di oggetti `Branch` creati dal metodo. $\frac{1}{2}$☠
- `t.get(i)` ritorna l'elemento con indice `i` nell'albero `t`, dove gli indici validi sono quelli compresi tra `0` e `t.size() - 1`. Realizzare il metodo `get` in modo tale che se `i < j` allora `t.get(i) < t.get(j)`. ☠

Scrivere un adeguato programma di prova per verificare il corretto funzionamento di *tutti* i metodi implementati.

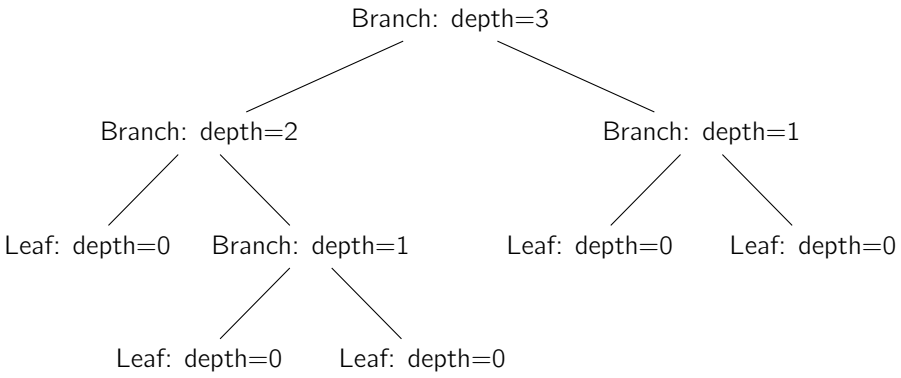
A titolo di esempio, se `t` è un riferimento alla radice dell'albero binario di ricerca



dove • sono le foglie, allora avremo:

- `t.size() == 4`
- `t.sum() == 11`
- `t.contains(3, 0) == true` e `t.contains(2, 1) == false`
- `t.filter_le(2)` è uguale al (ma non necessariamente lo stesso) sottoalbero con radice `1`
- `t.balanced() == true`
- `t.filter_le(0)` è l'albero vuoto
- `t.get(1) == 2` e `t.get(3) == 5`

NOTA: la profondità di un nodo (il metodo `depth()`)) è definita come 0 per i nodi Leaf, e come 1 più la massima profondità dei due sottoalbero per i nodi Branch. Ad esempio:



É molto importante aver capito come è definita la profondità per poter implementare correttamente il metodo `balanced`.