Exercise 1: BST height

Add to `bst.hpp` a method `height()` that computes the height of the tree. Develop two implementations: a recursive method (which takes linear time and space proportional to the height), and method like `size()` that adds a field to each node in the tree (and takes linear space and constant time per query).

Exercise 2: BST implementation and test.

All methods from slides are implemented in `bst.hpp`. Complete the BST class by adding the implementation of `max`, `ceiling`, `removeMax`, and `keys(lo, hi)` methods.

Write a test client `test_bst.cpp` for use in testing the implementations of `min()`, `max()`, `floor()`, `ceiling()`, `select()`, `rank()`, `removeMin()`, `removeMax()`, and `keys()` that are given in the text.

Exercise 3: Perfect balance.

Write a program `perfect-balance.cpp` that inserts a set of keys into an initially empty BST such that the tree produced is equivalent to binary search, in the sense that the sequence of compares done in the search for any key in the BST is the same as the sequence of compares used by binary search for the same set of keys.

Hint: Put the median at the root and recursively build the left and right subtree.

Exercise 4: Certification.

Write a method `isBST()` in `BST.hpp` that takes a `Node*` as argument and returns **`true`** if the subtree rooted at argument node is a binary search tree, **`false`** otherwise.

Exercise 5: Subtree count check.

Write a recursive method `isSizeConsistent()` in `BST.hpp` that takes a `Node*` as argument and returns **`true`** if the subtree `count` field is consistent in the data structure rooted at that node, **`false`** otherwise.

Exercise 6: Select/rank check.

Write a method `isRankConsistent()` in `BST.hpp` that checks, for all `i` from `0` to `size()- 1`, whether `i` is equal to `rank(select(i))` and, for all keys in the BST, whether key is equal to `select(rank(key))`.