Instructions: Word Prediction We want to use a tree data structure to which will store words for efficient word prediction. We should be able to query the tree by providing a prefix, e.g. "telep", and it should return a "prediction" for the full word we intend to write, e.g. "telephone". A naive algorithm, which iterates over a list of words, checking each word to see if it starts with the provided prefix, runs in linear time. Using a tree, we can get an algorithm that runs in logarithmic time. ## Procedure \* Start with a copy of the skeleton code for this task, which can be found [here](./skeleton-code). \* Modify the `DictionaryTree` class. \* YOU CAN WRITE HELPER METHODS. \* YOU MUST NOT MODIFY THE SIGNATURES OF THE METHODS PROVIDED, AS WE WILL BE USING AUTOMATED MARKING WHICH WILL ASSUME THESE METHODS HAVE BEEN IMPLEMENTED. \* IF YOU HAVE NOT IMPLEMENTED A METHOD, TESTING WILL NOT BE USED ON THIS METHOD. \* You need to add testing code to convince yourself that your code works. \* You can use the `CLI` to perform some testing by hand to quickly check if you are heading in the right direction, check for bugs, etc. \* You are highly encouraged to write unit tests for your code. There are a few sample unit tests in the `DictionaryTreeTests` class, which can be run with the `run-test-suite.sh` script. These will help to ensure you have implemented your solution correctly, and you may get credit if these are done properly. \* To avoid duplicate large files, we have used symbolic links to point to `lib/junit.jar`. If you have problems with these symbolic links, just create a copy of `lib/junit.jar` in place of the symbolic link. ## Methods to implement \* `size`: returns the number of nodes in the tree. \* `height`: returns the height of the tree. Since the tree is never empty, height should always be non-negative. \* `maximumBranching`: each node has a number of children - `maximumBranching` should return the maximum number of children in a held by any node. \* `longestWord`: returns the longest word stored in this tree. \* `numLeaves`: returns the number of leaves in this tree, i.e. the number of words in this tree which are not prefixes of any other word. \* `contains`: returns true if the given word is held in this tree, and false otherwise. \* `allWords`: returns all words held in this tree. \* `insert`: inserts the given word into this tree. \* `remove`: removes the given word from this tree. If the word is not already in the tree, nothing should change. If the word is in the tree, then `contains(word)` should return true before invoking `remove(word)` and return false after. \* `predict`: given a prefix, this method should return a word in this tree that starts with this prefix.