HW 10: Binary Search Tree

You must create a `BinarySearchTree` object, each node of which contains a single object of generic type `E`. Note that `E` *must* implement the `Comparable<E>` interface for a binary search tree to function.

Your implementation must include the following public methods:

- `BinarySearchTree()`, which creates a brand-new tree, with its initial root set to `null`.
- `void insert(E value)`, which inserts a new value into the binary search tree. (Note that if the tree was empty, the root needs to be set to the new value.)
- `boolean has(E value)`, which returns `true` if the given value is present in the tree, or `false` otherwise.
- `E findMin()`, which returns the minimum value in this tree.
- `E findMax()`, which returns the maximum value in this tree.
- `int getSize()`, which returns the number of nodes inside this `BinarySearchTree`. (You will probably want to maintain a counter.)
- `static void main(String[] args)`, which tests your object as detailed below.

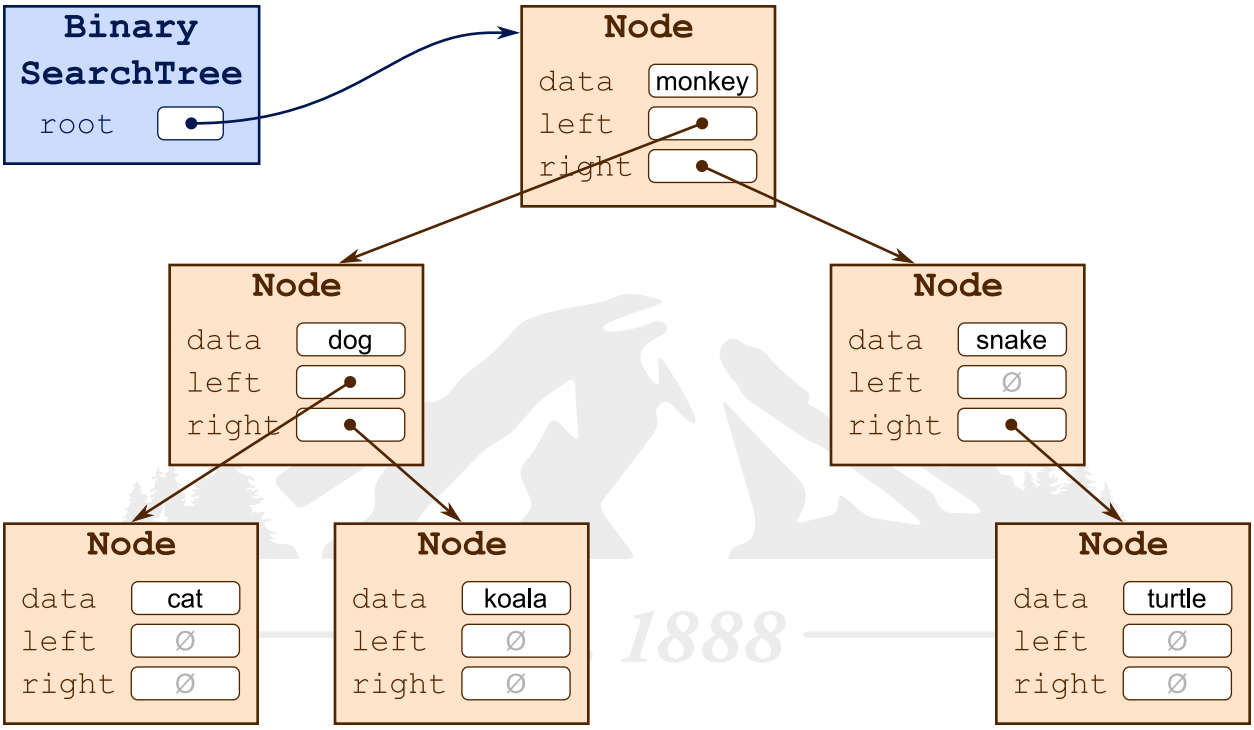
Each of these functions except `main()` and (`getSize()` should have time complexity of $O(\log n)$ on average, or $O(n)$ at worst. (`getSize()` will always be $O(1)$.)

Your `main()` function should load the file `english.lex` as an array, shuffle it, and then add each element to the `BinarySearchTree`. It will then state how many words there are, what the first and last ones are, and will enter a query loop as in the lab:

```
Testing BinarySearchTree
Loaded "english.lex" (62970 words from "a" to "zygote")
Please enter a word, or hit enter to quit:
> green
"green" is a valid word.
> sldkfj
"sldkfj" is NOT a valid word.
>
Goodbye!
```

Hints:

- While you are working on this assignment, make your own `.lex` file that contains only 10 words or so. That way you can test it with a much more human-sized data set. When you think it works, try the full file.
- You already made a shuffling function in the card homework. Copy it and modify it as needed.



- You may want to create your own private `print()` function which prints out the tree. This would be solely for your own debugging purposes.

For extra credit, you may implement the following methods:

- `E findPredecessor(E value)`, which returns the value in the tree immediately *before* the given value, alphabetically. If the value is *not* in the tree or is the first element, return *null* instead. Your query system should call this function if the user enters `<word`, which will find word's predecessor. If it has no predecessor, say so explicitly.
- `E findSuccessor(E value)`, which returns the value in the tree immediately after the given value, alphabetically, in a similar way to `findPredecessor()`. The query system will test this when the user enters `>word`. If it has no succsessor, say so explicitly.
- `boolean delete(E value)` If the value is present in the tree, delete it and return `true`. Otherwise, return `false`. The query system will test this when the user enters `-word`. It will state whether the word was successfully deleted, or if it did not exist.

Implementing these can add as much as 25% to your grade.