CSCE 2211: Applied Data Structures
Assignment #4

Date: Tuesday October 22nd
Due Date: Saturday November 2nd, 2024 11:59 CLT

Spell Checker: An Application of Binary Search Trees

Many applications such as word processors, text editors, and email clients, include a spellchecking feature. A direct way of performing spell checking is to store the set of all valid words in some data structure. For each word we need to check, we'll search the data structure to see if the word is stored. If it is, we will say that the word is spelled correctly. If not, we will say that the word is spelled incorrectly. Therefore, a spell checker's task is centered on searching for words in a potentially large word list and reporting all the misspelled words (i.e., words that are not found in the word list). An efficient search is critical to the performance of the spell checker, since it will be searching the word list not only for the words in the input, but possibly suggestions that it might like to make for each misspelled word.

Because we do not know in advance the size of the word list, the straightforward way to do this is to build a *Dictionary* of the distinct words (as keys). However, the number of words in the Dictionary may grow to be very large. The solution is to maintain the dictionary as an **array of binary search trees (BSTs)**; each BST is for words beginning with a given alphabetical letter.

Requirements

- Implement a spell checker that starts by reading a corpus file (`corpus.txt`) containing a set of words. From this file, create a Dictionary represented as an array of Binary Search Trees (BSTs), with each BST corresponding to a specific letter of the alphabet, and save this structure to a file named `dictionary.txt`. The spell checker will then use the Dictionary to check the spelling of words in an input file (`input.txt`), identifying any misspelled words and providing suggestions for corrections.
- Your spell checker is *case insensitive.*
- For each misspelling found in the input file, your program will report it, along with a list of five suggestions (i.e., the five closest words from the word list).

The Dictionary

- The dictionary is to be implemented as an **array of binary search trees (BSTs)**. Each BST stores the words in the word list beginning with a given alphabetical letter (or numeric digit).

- Your dictionary should be a *dynamic* one, i.e., it should allow for adding new words or deleting already existing ones.

Suggested Closest Words

Definition of closest in this assignment follows the **Hamming-Distance** concept. Such distance between two strings is computed using the following algorithm:

```
For a string A of length |A| = LA characters, and string B of length |B| = LB
    characters:
    A match occurs at position k if A(k) = B(k) for all k = 1 ... min (LA , LB)
    M = # of matches
    Distance = Number of mismatches D = max (LA , LB) - M
Notice that D = 0 if A and B are identical
```

For each misspelled word, starting with the same first character, find the list of *five* closest words from the word list using the above algorithm.

Required Implementations

1. **The BST ADT**: The BST ADT can be implemented with the following member functions:
    - **Constructor**: construct an empty tree
    - **Copy constructor**: copy the tree from an existing tree
    - **Destructor**: Destroy tree
    - **insert**: Insert an element into the tree
    - **empty**: Return True if tree is empty
    - **search**: Search for a key
    - **retrieve**: Retrieve data for a given key
    - **traverse**: Traverse the tree
    - **preorder**: Pre-order traversal of the tree
    - **levelorder**: level-order traversal of the tree
    - **remove**: Remove an element from the tree

    Design and implement a *template* class **BST** with a minimum of the above member functions. Use a node class that is composed of a word from the word list, a pointer to the left sub-tree and a pointer to the right sub-tree. Add to the class any required private functions.

2. **Implement a program** to build and maintain a dictionary with the following functions:
    - Generate the dictionary and the corresponding file from an input corpus file
    - Update the dictionary by adding or removing words from it
    - Save an updated dictionary to disk as a text file (See note below*)
    - Read a dictionary stored on disk into memory

- Find how many words there are in the dictionary
- Spell check an input file such that for each misspelled word, it lists five suggestions from the word list that are closest to it based on the Hamming Distance

***Note:** The BSTs generated for the first time have to be stored on disk as a text file. This is achieved by traversal of the BSTs. You should choose the traversal order that will retain the same BSTs when you read them back from disk to memory.