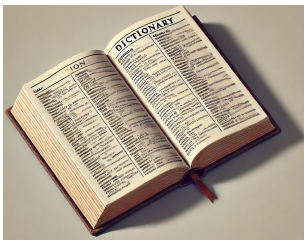
Homework 9: Building a Dictionary Application Using a Binary Search Tree (BST) and TreeSort algorithm

In this assignment, we use the term *dictionary* to mean a collection of words and their definitions. This has no relationship to the python built-in dict, which implements the Mapping ADT.



Objective

This assignment will provide a practical example of using BSTs for dictionary applications and introduce the TreeSort. algorithm as a sorting mechanism.

You'll create a dictionary application using a self-balancing Binary Search Tree (BST) in Python. Each word will be stored as a node, where the word itself is the key, and its meaning is the associated value.

Binary Search Trees (BSTs) are commonly used for searching and sorting data due to their efficient $O(\log n)$ search and insertion times. A self-balancing Tree is a type of BST that automatically balances itself. This balancing allows all operations—including insertion, deletion, and search—to maintain $O(\log n)$ time complexity.

For this homework, you'll also implement `TreeSort`, a sorting algorithm that relies on in-order traversal of a BST to sort data in $O(n \log n)$ time complexity. In TreeSort:

1. **Insertion:** Each element from the data set is inserted one by one into a balanced BST. For each insertion, the time complexity is $O(\log n)$ because a balanced binary tree maintains a height of $\log n$. As we insert n elements, the total time complexity to build the BST is $O(n \log n)$.
2. **In-order Traversal:** In-order Traversal: After all elements are inserted, we perform an in-order traversal of the BST. In-order traversal visits nodes in ascending order (left subtree, root, right subtree) and outputs the data in sorted order. This traversal has a time complexity of $O(n)$ since each node is visited exactly once.

By using a self-balancing BST, TreeSort operates efficiently in the worst case, maintaining $O(n \log n)$ time complexity for sorting the data

Requirements

1. `Node` Class: Implement a `Node` class to represent each node in the tree.
2. `DictionaryBST` Class: Implement a `DictionaryBST` class to serve as the dictionary itself, where each word and meaning is stored as a `Node` in an BST.

Required Methods:

- `insert(word, meaning)`: Insert a new word and its meaning into the dictionary. Ensure the tree remains balanced. This method should operate with $O(\log n)$ time complexity.
- `search(word)`: Search for a word in the dictionary and return its meaning if found. This method should operate with $O(\log n)$ time complexity.
- `print_alphabetical()`: This method uses the TreeSort algorithm to return all dictionary entries in alphabetical order. As described above, the TreeSort algorithm achieves $O(n \log n)$ time complexity for inserting elements into a balanced BST and $O(n)$ for the in-order traversal, resulting in an overall time complexity of $O(n \log n)$. Since the balanced BST is already created by this point, this method will maintain an $O(n)$ time complexity by performing an in-order traversal to print the entries in alphabetical order.

You may implement additional private and helper methods as needed to maintain efficiency and self-balancing properties.

Example:

```
# Sample usage
dictionary = DictionaryBST()
dictionary.insert("banana", "A yellow tropical fruit.")
dictionary.insert("apple", "A fruit that grows on trees.")
dictionary.insert("cherry", "A small, round, red fruit.")

# Search for a word
print(dictionary.search("banana"))  # Output: A yellow tropical fruit.

# Print all entries in alphabetical order
print(dictionary.print_alphabetical())
# Output:
# [
#     ("apple", "A fruit that grows on trees."),
#     ("banana", "A yellow tropical fruit."),
#     ("cherry", "A small, round, red fruit.")
# ]
```

Submission

Submission Submit the following files:

- `dictionary_bst.py`: Contains the `Node` and `DictionaryBST` classes with all required methods and any additional methods.
- `test_dictionary_bst.py`: A file with unit tests for the public interface of the dictionary application, including insertions, searches, alphabetical ordering.

Students must submit to Gradescope individually within 24 hours of the due date (homework due dates are typically Tuesday at 11:59 pm EST) to receive credit.