CISC-235 Data Structures W23
Assignment 2

February 14, 2023

General Instructions

Write your own program(s) using Python. Once you complete your assignment, place all Python files in a zip file and name it according to the same method, i.e., "235-1234-Assn2.zip". Unzip this file should get all your Python file(s).

Then upload 235-1234-Assn2.zip into Assignment 2's entry on onQ. You may upload several times if you wish. However, onQ keeps only the last uploaded file. The newly uploaded file will overwrite the old file. Please check your files after uploading. We will check the latest submission you made following the required naming.

You must ensure your code is executable and document your code to help TA mark your solution. We suggest you follow PEP8[1] style to improve the readability of your code.

All data structures involved must be implemented by yourself, except for the built-in data types, i.e., List in Python.

An "I uploaded the wrong file" excuse will result in a mark of zero.

1 Binary Search Tree (55 points)

Binary search tree (BST) is a special type of binary tree that satisfies the binary search property, i.e., the key in each node must be greater than any key stored in the left sub-tree, and less than any key stored in the right sub-tree.

Your task is to implement a BST class, satisfying the following requirements (you can create more methods/attributes if needed):

1) (5 points) Must have an **insert (self, value)** function that inserts a new node with a given value into the BST. You may assume that the values to be stored in the tree are integers.

2) (10 points) Must have a **get_total_height(self)** function that computes the sum of the heights of all nodes in the tree. Your get_total_height

[1] `https://peps.python.org/pep-0008/`

function should run in O(n) time in the worst case, where n refers to the total number of nodes in the tree.

3) (15 points) Must have a **delete(self, value)** function that could be used to delete one node from the BST by its value, **recursively**.

4) (20 points) Write a **save(self)** and a **restore(self, input_string)** function for your BST class. These two functions can transfer a BST into a string and reconstruct it back to the same tree.

5) (5 points) Write test code in the main function, covering all functions mentioned above.

2 AVLTreeMap: A Modified AVL Tree (45 points)

AVL Tree is one type of BST that ensures its balance during insertion/deletion. Your task is to implement a special AVL tree in a class named **AVLTreeMap**. This AVLTreeMap should have a *load_from_file(self, file_path)* function. This function aims to read content from a file and save word-frequency information for all words appearing in the file in the AVLTreeMap.

Specifically, *load_from_file* function takes a file_path as input, reads lines from the file, and extracts word tokens appearing in lines following their appearance order in the file. Next, it inserts extracted tokens one by one into an empty AVLTreeMap (you should empty the current AVLTreeMap each time before adding content from the file). Each AVLTreeMap node contains five attributes: leftchild, rightchild, word, frequency, and height.

For instance, if a given document contains a sentence "Binary search tree is a special binary tree.". *load_from_file* function will first get a list of meaningful words, "binary", "search", "tree", "special", "binary", "tree" using the following supporting functions (you need to modify the code and add them to your AVLTreeMap class, e.g., adding self):

```
import re
import string
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))

def parse_file(file):
    with open(file, 'r') as input:
        content = input.readlines()

    preprocessed = []
    for line in content:
        line = line.strip().lower()
        #remove punctuation
        line = line.translate(str.maketrans('', '', string.punctuation))
```

```
        #remove stop words that care no specific meaning
        line = remove_stopwords(line)
        #remove numbers
        line = re.sub('\d+','', line)
        #remove extra white space
        line = re.sub(' +', ' ', line)
        if line:
            preprocessed.extend(line.split(" "))

    print(" ".join(preprocessed))
    return preprocessed

def remove_stopwords(text):
    return " ".join([word for word in str(text).split() if word not in stop_words])
```

Then we scan the list from the first word and insert them one by one, the first node inserted would contain “binary” as the word, 1 as the frequency. When we see the second ”binary” in the list, since the node having word = ”binary” already exists, we will just update the frequency attribute of the root node to be 2. The final AVLTreeMap can be presented in Figure 1. Word ”binary” and ”tree” has a frequency = 2 because they appear twice in the input file.

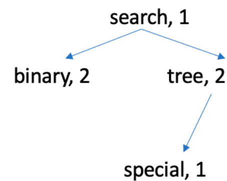

Figure 1: Sample AVLTreeMap

You should implement other supporting functions to ensure the accuracy of attribute values in each node in the AVLTreeMap. In the main function, test your AVLTreeMap.

40 points for the implementation of required AVLTreeMap class and 5 points for the testing code in the main function. Your test file should be packed together with your python code in the zipped submission.