Assignment 4
Implementing Self Balanced BSTs (AVL & Red Black)

1 Introduction

1.1 AVL Tree

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balance condition. The balance condition is easy to maintain, and it ensures that the depth of the tree is O(log n). An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. Balancing information is kept for each node (in the node structure).
All the tree operations can be performed in O(log n) time. The reason that insertions and deletions are potentially difficult is that the operation could violate the AVL tree property. The balance of an AVL tree can be maintained with a simple modification to the tree, known as a **rotation**.

1.2 Red Black Tree

A red black tree is a kind of self-balancing binary search tree in computer science. Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions. Balance is preserved by painting each node of the tree with one of two colors in a way that satisfies certain properties, which collectively constrain how unbalanced the tree can become in the worst case. When the tree is modified, the new tree is subsequently rearranged (using rotation) and repainted to restore the coloring properties. The properties are designed in such a way that this rearranging and recoloring can be performed efficiently.

2 Requirements

2.1 Trees Operations

You are required to implement both AVL and Red Black trees that can deal with keys of any comparable type (generic). You need to implement the following operations in both of them:

1. Insert: Takes a key and inserts it if it is not in the tree. Returns true if it is added and false if it already exists. Must run in O(log n)
2. Delete: Takes a key and deletes it if it is in the tree. Returns true if it is deleted and false if it is not in the tree. Must run in O(log n).
3. Search: Takes a key and searches for it returning true if it is found in the tree and false otherwise. Must run in O(log n).
4. Size: Returns the number of keys in the tree. Must run in O(1).
5. Height: Returns the height of the tree which is the longest path from the root to a leaf node. Can run up to O(n).

2.2 Application: English Dictionary

As an application based on the 2 self balanced binary search trees implementation, you are required to implement a simple English dictionary supporting the following functionalities:

1. Initialize (constructor): Takes the name of the type of the backend tree as an input and create a new empty dictionary based on it.
2. Insert: Takes a single string key and tries to insert it.
3. Delete: Takes a single string key and tries to delete it.
4. Search: Takes a single string key, searches for it and return true if it exists and false otherwise.
5. Batch insert: Takes a path to a text file containing multiple words each in a separate line. And tries to insert all that words in the dictionary.
6. Batch delete: Takes a path to a text file containing multiple words each in a separate line. And tries to delete all that words from the dictionary.
7. Size: Returns the number of string keys in the dictionary.
8. Tree Height: Returns the height of the dictionary backend tree.

2.3 Command Line Interface

You should implement a command line interface that will enable us to deal with the dictionary and apply all its implemented operations. This interface must take the type of the backend tree as an initial input then creates a dictionary based on it and allow the user to apply subsequent operations on it from the following list:

1. Insert a string and prints a confirmation message or an error one if the the string already exists in the dictionary.
2. Delete a string and prints a confirmation message or an error one if the the string doesn't exist in the dictionary.
3. Search for a string and print whether it exists in the dictionary or not.
4. Batch insert a list of strings taking the path of the file containing these strings and prints the number of newly added strings and the number of already existing ones.
5. Batch delete a list of strings taking the path of the file containing these strings and prints the number of deleted strings and the number of non existing ones.
6. Get the size of the dictionary and print it.
7. Get the height of the backend tree and print it.

2.4 Java Unit Testing

You should provide a set of 15-20 JUnit tests that tests the correctness and effeciency of the different implemented parts. Also, these tests must show a comprehensive comparison between the 2 types of binary search trees w.r.t time and tree height factors.

3 Notes

- You need to work in teams of 4 or 5.
- You need to use Java in your implementation.
- Each team should submit via teams the code and a **report** explaining the time analysis of the implemented functions in addition to a comparison between the 2 trees at different tree sizes w.r.t the tree height and the mean insert and delete time.

Good Luck