Data Structures 2 - Lab 2
Implimenting AVL Trees

1 Purpose

This lab assignment focuses on self-balancing binary search trees.

2 Backgroud

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 $O(\lg n)$. 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. (The height of an empty tree is defined to be -1.)

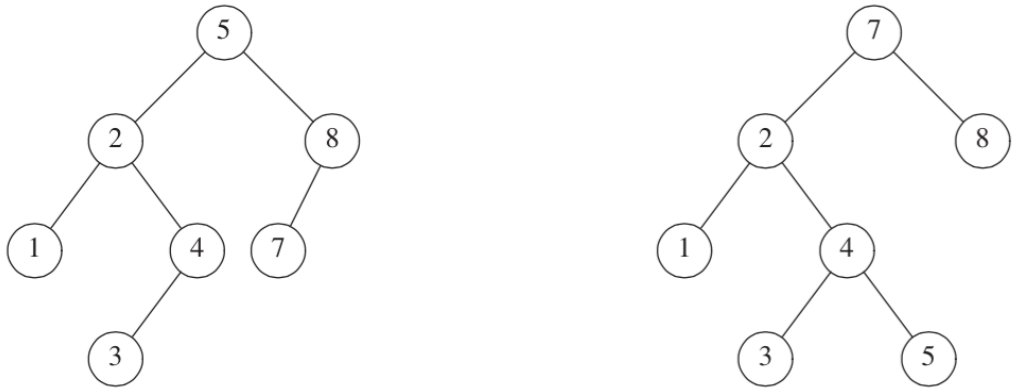

Figure 1: Two binary search trees. Only the left tree is AVL

In Figure 1 the tree on the left is an AVL tree, but the tree on the right is not. Height information is kept for each node (in the node structure).

All the tree operations can be performed in $O(\lg n)$ time, except possibly insertion and deletion. The reason that insertions and deletions are potentially difficult is that the operation could violate the AVL tree property.[1] The balance of an AVL tree can be maintained with a simple modification to the tree, known as a **rotation**.

[1]For instance, inserting 6 into the AVL tree in Figure 1 would destroy the balance condition at the node with key 8.

2.1 Rotations

Let us call the node that must be rebalanced α. Since any node has at most two children, and a height imbalance requires that α's two subtrees height differ by two, it is easy to see that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of α.
2. An insertion into the right subtree of the left child of α.
3. An insertion into the left subtree of the right child of α.
4. An insertion into the right subtree of the right child of α.

Cases 1 and 4 are mirror image symmetries with respect to α, as are cases 2 and 3. The first case, in which the insertion occurs on the "outside" (i.e., left-left or right-right), is fixed by a **single rotation** of the tree. The second case, in which the insertion occurs on the "inside" (i.e., left-right or right-left) is handled by the slightly more complex **double rotation**.

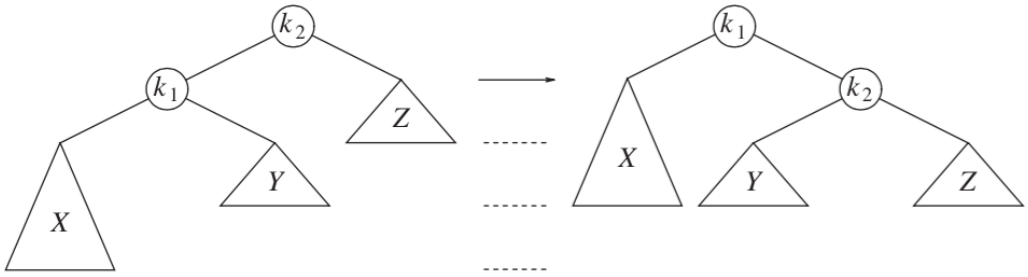

Figure 2: Single rotation to fix case 1

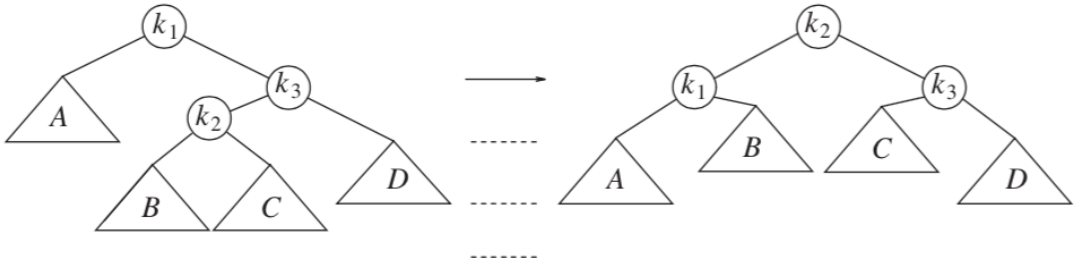

Figure 3: Rightleft double rotation to fix case 3

3 Requirements

3.1 AVL Tree Implementation

You are required to implement the AVL Tree data structure supporting the following operations:

1. **Search:** Search for a specific element in an AVL Tree.
2. **Insertion:** Insert a new node in an AVL tree. Tree balance must be maintained via the **rotation** operations.
3. **Deletions:** Delete a node from an AVL tree. Tree balance must be maintained via the **rotation** operations.
4. **Get Tree Height:** Return the height of the AVL tree. This is the longest path from the root to a leaf-node.

Please refer to the reference below for more implementation details.

3.2 Application: English Dictionary

As an application based on your AVL Tree implementation, you are required to implement a simple English dictionary, supporting the following functionalities:

- **Load Dictionary:** You will be provided with a text file, "dictionary.txt", containing a list of words. Each word will be in a separate line. You should load the dictionary into an AVL Tree data structure to support efficient insertions, deletions and search operations.
- **Get Dictionary Size:** Returns the current size of your dictionary.
- **Insert Word:** Takes a word from the user and inserts it, only if it is not already in the dictionary. Otherwise, print the appropriate error message (e.g. "ERROR: Word already in the dictionary!").
- **Look-up a Word:** Takes a word from the user and returns "True" or "False" according to whether it is found or not.
- **Remove Word:** Takes a word from the user and removes it from the dictionary. It returns true in case of successful deletion, false otherwise (e.g. if the word is not in the dictionary).

4 Interfaces

- INode: this interface will represent any node in your AVL tree.

```
package eg.edu.alexu.csd.filestructure.avl;

public interface INode<T extends Comparable<T>> {
        /**
         * Returns the left child of the current element/node in the heap tree
         * @return        INode wrapper to the left child of the current element/node
         */
        INode<T> getLeftChild();

        /**
         * Returns the right child of the current element/node in the heap tree
         * @return        INode wrapper to the right child of the current element/node
         */
        INode<T> getRightChild();

        /**
         * Set/Get the value of the current node
         * @return        Value of the current node
         */
        T getValue();
        void setValue(T value);
}
```

- IAVLTree: This interface will represent your AVL tree implementation.

```
package eg.edu.alexu.csd.filestructure.avl;

public interface IAVLTree<T extends Comparable<T>> {

        /**
         * Insert the given value using the key
         * @param key    the value to be inserted in the tree
         */
        void insert(T key);

        /**
         * Delete the key (if exists)
         * @param key    the key of the node
         * @return       true if node deleted, false if not exists
         */
        boolean delete(T key);

        /**
         * Search for a specific element using the key in the tree
         * @param key    the key of the node
         * @return       true if the key exists, false otherwise
         */
        boolean search(T key);

        /**
         * Return the height of the AVL tree. This is the longest path from
         * the root to a leaf-node
         * @return        tree height
         */
        int height();

        /**
         * Return the root of your AVL tree.
         * @return        root of the AVL tree.
         */
        INode<T> getTree();


}
```

- IDictionary: This interface will represent your dictionary.

```
package eg.edu.alexu.csd.filestructure.avl;

public interface IDictionary {

        /**
         * Load the dictionary into an AVL Tree data structure. The file is text
         * containing a list of words. Each word will be in a separate line
         * @param file   the dictionary file
         */
        void load(java.io.File file);

        /**
         * Takes a word and inserts it, only if it is not already in the dictionary
         * @param word            word to insert
         * @return    true if inserted successfully, false if already exists
         */
        boolean insert(String word);

        /**
         * Takes a word from the user and checks whether it is found or not.
         * @param word   word to lookup
         * @return       true if exists, false if not
         */
        boolean exists(String word);

        /**
         * Takes a word from the user and removes it from the dictionary.
         * @param word     word to delete
         * @return         false if the word is not in the dictionary, true otherwise
         */
        boolean delete(String word);

        /**
         * Prints the current size of your dictionary
         * @return    dictionary size
         */
        int size();

        /**
         * Print the maximum height of the current tree
         * @return   AVL tree height
         */
        int height();
}
```

5 Notes

- You **must work in groups** of two or three
- You will need to submit a report including; problem statement, algorithms and data structures used, important code snippets and sample runs.
- Try very hard to clean up your implementation. Remove all unused variables. Do not write redundant and repeated code. You may use Checkstyle with your IDE to ensure that your code style follows the JAVA coding style standards

6 References

- Weiss, Mark Allen. "Data structures and algorithm analysis in Java." Addison-Wesley Longman Publishing Co., Inc., 1998.

Good Luck