Assignment 4

General Instructions

The grading will be done automatically. To ensure a smooth process, an interface will be provided to you, which you are NOT supposed to change. Your solution classes will implement these interfaces.

Code Skeleton

You are provided with the skeleton of the code. This contains the interfaces and other relevant information. Your task is to implement these functions. Please DO NOT modify the interface. You are free to change and implement other parts in any way you like. Code is available in a .zip file named COL106A4.zip.

Question 1 - Implementation of Weak AVL trees (60 Marks)

What are Weak AVL trees?

Weak AVL (WAVL) trees are rank-balanced trees that can be seen, as the name suggests, a variant of AVL tree (and also of red-black trees!). They are theoretically shown to be more efficient than both AVL and red-black trees for updates, while keeping the same search complexity. These trees were introduced only very recently (first in 1996, and revised journal paper in 2015), and therefore there are very few implementations of these data structures.

In this assignment, you will implement WAVL trees such that they can be used as sorted map structures (i.e., a Map ADT which also allows for sorted access and range search based on the values of keys).

A weak AVL tree has two kinds of nodes: **internal** nodes and **external** nodes.

- An internal node is allowed to store a data item, whereas an external node is not. An internal node is linked to its parent node (except for a root node) and to exactly two children in the tree- the left child and the right child. An external node has a link only to its parent node in the tree. Internal nodes store data in sorted order (or in an in-order traversal order).

- Additionally, each node (either internal or external) maintains a number called the node's rank,~~ i.e., the height of the node~~, which is a kind of "stand in" for height of the node rather than having nodes have ranks exactly equal to their heights. In WAVL trees the tree balancing during insertion and deletion of nodes is done using ranks. The rank of the tree is the rank of its root. For each node *x* in the tree, there is a **rank difference**, i.e., **rank of parent(*x*) - rank of *x***. Based on the rank difference, we classify internal nodes of WAVL as:
    1. **1,1-node**: if its children each have a rank difference of 1.
    2. **2,2-node:** if its children each have rank difference 2.
    3. **1,2-node**: if it has one child with rank difference 1 and one child with rank difference 2.
- Ranks are assigned to node such that they satisfy the following properties:
    - Every external node has rank 0,
    - If an internal node has rank r, then its parent (if it exists) must have rank either r+1 or r+2.
    - The internal nodes with two external children must have rank exactly 1.

Operations on WAVL trees:

1. Insertion:

Insertion into WAVL trees is done the same way as insertion in standard BSTs, but a series of rebalancing steps must be done from the newly inserted node, possibly up to the root. Insertion of a key *k* into a WAVL tree is performed by performing a search for the external node where the key should be added, replacing that node by an internal node with data item *k* and two external-node children, and then rebalancing the tree. The rebalancing step can be performed either top-down or bottom-up, but the bottom-up version of rebalancing is the one that most closely matches AVL trees.

In this rebalancing step, one assigns rank 1 to the newly created internal node, and then follows a path upward from each node to its parent, incrementing the rank of each parent node if necessary to make it greater than the new rank of its child, until one of three stopping conditions is reached.

- If the path of incremented ranks reaches the root of the tree, then the rebalancing procedure stops, without changing the structure of the tree.
- If the path of incremented ranks reaches a node whose parent's rank previously differed by two, and (after incrementing the rank of the node) still differs by one, then again the rebalancing procedure stops without changing the structure of the tree.
- If the procedure increases the rank of a node x, so that it becomes equal to the rank of the parent y of x, but the other child of y has a rank that is smaller by two (so that

the rank of y cannot be increased) then again the rebalancing procedure stops. In this case, by performing at most two tree rotations, it is always possible to rearrange the tree nodes near x and y in such a way that the ranks obey the constraints of a WAVL tree, leaving the rank of the root of the rotated subtree unchanged.

Thus, overall, the insertion procedure consists of a search, the creation of a constant number of new nodes, a logarithmic number of rank changes, and a constant number of tree rotations.

2. Deletion

Deletion is also done as it is in standard BSTs, with a series of rebalancing steps up to the root. As with binary search trees more broadly, deletion operations on an internal node *x* that has at least one external-node child may be performed directly, by removing *x* from the tree and reconnecting the other child of *x* to the parent of *x*. If, however, both children of a node *x* are internal nodes, then we may follow a path downward in the tree from *x* to the leftmost descendant of its right child, a node *y* that immediately follows *x* in the sorted ordering of the tree nodes. Then *y* has an external-node child (its left child). We may delete *x* by performing the same reconnection procedure at node *y* (effectively, deleting *y* instead of *x*) and then replacing the data item stored at *x* with the one that had been stored at *y*.

In either case, after making this change to the tree structure, it is necessary to rebalance the tree and update its ranks. As in the case of an insertion, this may be done by following a path upwards in the tree and changing the ranks of the nodes along this path until one of three things happens: the root is reached and the tree is balanced, a node is reached whose rank does not need to be changed, and again the tree is balanced, or a node is reached whose rank cannot be changed. In this last case a constant number of tree rotations completes the rebalancing stage of the deletion process.

Overall, as with the insertion procedure, a deletion consists of a search downward through the tree (to find the node to be deleted), a continuation of the search farther downward (to find a node with an external child), the removal of a constant number of new nodes, a logarithmic number of rank changes, and a constant number of tree rotations.

3. Searching

Searching in a WAVL is the same as in an AVL tree. To search a key k, start at the root of the tree, compare k with each node’s value. If not matched, follow the path to the left child of the node when k is smaller than the value at the node, else follow the path to the right child of the node. Search will stop when a node’s value matches with k.

Implementation Specification

Write a Java program for a Sorted Map ADT with weak AVL tree having (i) **insert**, (ii) **delete**, (iii) **search** and (iv) **search-range** methods. The name of the class should be

col106.assignment4.WeakAVLMap.WeakAVLMap<K,V> and the signature of the public methods of the class should be the following:

a) **WeakAVLMap()** to create a default constructor.
b) **V put(K key, V val)** to insert value val with the specified key into the Map. If Key exists then update the value corresponding to key with value passed. Return the previous value associated with the key, or null if there was no mapping for the key.
c) **V remove(K key)** to delete the entry from the tree with specified key, and return the associated value object. If there is no node in the tree with the specified key, then it should return **null.**
d) **V get(K key)** to search for a value with the specified key in the tree. Again, if not found, this should return **null.**
e) **Vector <V> searchRange (K key1, K key2)** to search for values within the specified key range -- i.e., all values that have an associated key between key1 and key2 (both inclusive). You may assume that key1 <= key2 -- using a suitable comparator defined on class K. Note that the method returns a Vector object with value objects of class V in the sorted order of their key values. The method will return **null** if there are no entries in the specified key range.
f) **int rotateCount ()** to return the number of tree rotations performed by the WAVL tree so far -- i.e., from the moment the WeakAVLMap object was constructed. It should be updated whenever the tree is restructured through rotate left/right operations during the put/remove operations.
g) **int getHeight ()** to return the current height of the WAVL tree.
h) **Vector<K> BFS ()** to return the BFS traversal of the WAVL tree. You can use Queue for BFS.

References and Instructions:

a) More details about weak AVL can be found here.
b) You can start your implementation with writing code for AVL trees and then make it for WAVL. Your WAVL implementation must use the bottom-up approach.
c) The implantation ideas for the same can be found in the blog and associated github. You have to make sure that your code should not be copied from this link or any other resource.

Question 2 - Search, Hash, Maps (30 marks)

A. Implement the HashMap data structure.

Open addressing is one of the approaches used in hashing for collision resolution. Use this approach with linear probing and implement the hash map data structure via a Java class named **col106.assignment4.HashMap.HashMap<V>.** Each hash entry consists of a key and a value. In your implementation, consider the types of Keys as String and generic type V for the values.
You must use the following hash function : $x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0))\cdots))$ with a = 41, where x_i specifies the *i-th character of the string from the right* (NOTE: we are counting character positions from 0).

You need to implement the following public methods in your implementation.

1. **HashMap(int N)** : Constructor for the class. Your underlying hash table must have size N and it should be fixed throughout the execution. It is guaranteed that the number of <key, value> pairs inserted in the HashMap are less than N.

2. **V put(String key, V value)** : Inserts <key, value> pair in the map if the key doesn't already exist. If Key exists then update the value corresponding to key with value passed. Returns the previous value associated with key, or null if there was no mapping for key.

3. **V get(String k)** : get the value corresponding to the given key. If no key is found then return **null**.

4. **boolean remove(String key)** : Remove the mapping for the specified key from this map if present. Return true if specified mapping was ~~removed~~ present else return false.

   This following invariant may require shifting of keys after removing mapping in remove() method

   Invariant: For each key present in the hashmap, there must be no vacant slots between their natural hash position and their current position.

5. **boolean contains(String key)** : Return true if there is any mapping for the specified key.

6. **Vector<String> getKeysInOrder()** : Return the keys stored in the HashMap, in the order they are stored in the underlying ArrayList/Array.

B. Using the above implementation implement word counting problem.

In this problem you will be given a string and you need to implement a method which will return the number of times given word is repeated in the given string. For that you must implement a class **col106.assignment4.HashMap.WordCounter** Implement following methods.

1. **WordCounter ()** : Default constructor for the class.
2. **Int count (String str, String word)** : Returns frequency of specified word in the given string.

Question 3 - Performance Evaluation (10 marks)

You need to implement a common implementation for comparing the performance of WeakAVL and HashMap on a same input sequence. You will be given a sequence of commands of type **put**, and ~~**get**~~ **remove.** In this part, consider the types of Keys as String and Values as generic type V.

You must implement class **col106.assignment4.Map.Map<V>**. You need to implement the following public methods in your implementation:

1. **Map()** : Default constructor for the class.
2. **void eval (String inputFileName, String outputFileName)** : Takes input from file named "inputFileName" and generates output to the file named "outputFileName".
   a. **Input Format:** The first line contains one integer N, the number of commands. Each of the next N lines, contain two types of commands:
      i. Three space separated strings cmd, key and value, where cmd is "I" which instructs to insert the key-value pair

   ii. Two space separated strings cmd and key, where cmd is “D” which instructs to delete the key-value pair associated with the given key

b. **Output Format:** You need to report four integers, total time taken(in ms) by WeakAVL and HashMap on the insert commands and delete commands. Look at the example for the exact format.

Input Constraints: $1 \leq N \leq 10^6$

Example:

V = int

Input:

```
5
I apple 1
I mango 2
D apple
I orange 10
D orange
```

Output:

```
Operations  WAVL  HashMap
Insertions  53    30
Deletions   68    36
```

Regarding submission: You will be provided a file named “map_input” in the “Map” directory. You need to execute **eval** on it, (assuming V = int) and output the results in a file named “map_output”.

Submission Guidelines:

1. Submit your code in a *.zip* file named in the format **<EntryNo>.zip**. Make sure that when we run unzip <yourfile>.zip a folder <YourEntryNo> should be produced in the current working directory. The interface files are provided in a *zip* file exactly in the submission format.
   You will be penalized for any submissions that do not conform to this requirement.

2. The exact directory structure will be detailed on the moodle submission link. Your submission will be auto-graded. This means that it is essential to make sure that your code follows the specifications of the assignment precisely.

3. Your code must compile and run on our VMs. They run amd64 Linux version ubuntu 16.04 running Java JDK 11.0.5 (https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html).

4. You will be able to check if your submission complies with the proper format and runs as expected in the test environment by submitting to Moodle. The auto-grading in Moodle will run your code and verify if it passes the preliminary test cases.

5. Your submission will only be accepted once it passes the initial test cases. For this, you must at least complete the Minimal submission tests of the assignment (will be announced soon). Further test cases will be used for Evaluation.

What is Allowed / Not Allowed?

1. You must work on this assignment individually.
2. The programming language to be used is Java.
3. You are not allowed to use any other external libraries for this assignment.
4. Your code must be your own. You are not to take guidance from any general-purpose or problem-specific code meant to solve this or related problem.
5. We will run a plagiarism detection check. Any person found guilty will be awarded a suitable penalty as announced in the class/course website. Please read academic integrity guidelines on the course home page and follow them carefully.
6. **Allowed Imports** (Do not import the entire package like java.util.* that will be invalid. Import specific allowed class only)
    a. java.util.ArrayList
    b. java.util.Vector
    c. java.util.LinkedList
    d. java.util.Queue