Programming Assignment #2
Word Search: Binary Tree vs. Hashing

Problem Statement

Binary Search Tree (BST) and Hashing are both powerful tools for search. In order to understand and evaluate differences between the search methods, we develop an electronic dictionary with three different data structures, namely, a *standard binary search tree*, an *AVL tree*, and a *linear hash table*. The electronic dictionary stores a set of distinct words (or strings) and determines whether a search word is found in its dictionary. The terms *key* and *word* are interchangeable in this document.

Requirements & Implementation

- You are to write Java classes that implement the standard BST, the AVL tree, and the Linear Hash table. Skeleton files of the classes are provided in the eTL site. They are named `BSTskel.java`, `AVLskel.java` and `LinearHashskel.java`. You can copy them to your working directory and rename to `BST.java`, `AVL.java` and `LinearHash.java`. You should then provide implementations of all the methods defined in the class skeletons. You can add more methods as you wish, but are not allowed to remove any method defined in the class skeletons.
- The `BST` methods must meet the following requirements.
  1. The `insert(String key)` method inserts a key into the tree using the standard BST insertion algorithm. If the same key exists already in the tree, increment its *frequency* by one without inserting the key itself into the tree. Otherwise, insert the key and set the *frequency* to one.
  2. The `find(String key)` method probes the tree to find a search key. A Boolean value `true` is returned if the key is found. Otherwise, `false` is returned. In either case, the `find()` method increments the *access count* by one for all the nodes probed (*i.e.*, all the nodes on the search path from the root).
  3. The `sumFreq()` and `sumProbes()` methods return the frequency sum and the access count sum of all the keys in the tree, respectively. The `resetCounters()` method resets the frequencies and access counts of all keys in the tree to one and zero, respectively.
  4. The `size()` method returns the number of keys in the tree. The `print()` method prints the keys in the tree in the increasing order of their values. Each key should appear on a separate line in the format of `[key:frequency:access_count]`.
- The `AVL` class is a subclass of `BST`. The AVL method must meet the following requirements.
  1. The `insert(String key)` method inserts a key into the tree using the standard AVL insertion algorithm. If the same key exists already in the tree, increment its *frequency* by one without inserting the key itself into the tree. Otherwise, insert the key and set the *frequency* to one.
- The `LinearHash` methods must meet the following requirements.
  1. The `LinearHash(int HTinitSize)` constructor creates an empty hash table with `HTinitSize` slots. Assume that the value of `HTinitSize` is always a power of two (say, $M_k = 2^k$). (There is no need to check this in your program.)
  2. The `insertUnique(String word)` method inserts `word` into the hash table using a base hash function h_k and optionally an extended hash function h_{k+1} if the target hash entry has already been split. The method returns the index of the hash table entry to which the word is inserted. Return `-1` if the word exists already in the hash table. Whenever a collision occurs, a new hash entry is added to the end of the hash table and a hash table entry is split (or rehashed). Specifically, at the i-th collision

($1 \leq i \leq 2^k$), the i-th hash entry is split and all the keys in the entry are rehashed between the i-th and $(2^k + i)$-th hash entries by the extended hash function h_{k+1}. A collision is detected in the hash entry where the `word` is actually inserted. When a total of 2^k collisions have occurred, h_{k+1} becomes a new base hash function and h_{k+2} becomes a new extended hash function, and no hash entries are considered split.

3. The `lookup(String word)` method returns the number of words in the hash table entry where the `word` is found. Return the negative value of the number if the word is not found. Use the base hash function to find out if the target hash entry has been split. If it has been split, use the extended hash function to find the `word`.
4. The `wordCount()`, `emptyCount()` and `size()` methods return the number of inserted words, the number of empty entries, and the size of the hash table, respectively. The return value from the `size()` method does not include the number of overflow entries. That is, the return value must be 2^k plus the number of collisions encountered in the current round.
5. The `print()` method prints the words in the hash table in the increasing order of their hash table entry numbers. All the words in the same entry should appear in ascending order on a separate line in the format of `[entry_number: a list of words]`.

- Use a static method `ELFhash` defined in `MyUtil.class` to map a string into a long integer. For example, if you need a hash function that maps a string s into a non-negative integer less than M, then your hash function will be $h(s) =$ `MyUtil.ELFhash`(s, M). For the `LinearHash` ADT, a family of hash functions would be created by $h_k(s) =$ `MyUtil.ELFhash`$(s, 2^k)$ for $k > 0$.
- Follow the directions given in the first assignment handout regarding 'no `main()` method.'

Grading

This assignment is worth 15 percent of your final grade. In writing the code, correctness is the most important attribute, followed by efficiency. General grading guidelines are:

10 points	Program compiles without errors and is on the right track,
0-10 points	The BST class works on simple and complex test cases,
0-30 points	The AVL class works on simple and complex test cases,
0-50 points	The LinearHash class works on simple and complex test cases.

To evaluate the efficiency of your class implementation, we will measure the time spent on construction of a tree or a hash table and its search. If it is more than 10 (100 or 1000) times slower than my own implementation, you will lose 10% (25% or 50%) of the credits reserved for the correctness. The late submission and regrading policies are described in the course syllabus.

Submission

Refer to the first programming assignment for submission instructions.

Due date

The programming assignment is handed out on Tuesday Apr 07, 2020, and due by 11pm on Monday May 04, 2020.