Programming and Data Structures with Python: Problem Set 3

1. Write a Python function `leftval` which takes an `AVLTree` (as defined in the class) and a value v as input and returns the largest value in the tree that is strictly smaller than v. If there is no such value it returns `None`.

2. In any tree, we say that a node x is the ancestor of the node y if x lies on the path from y to the root. In particular, y is an ancestor of y. We say that x is a common ancestor of y and z if it is an ancestor of y and an ancestor of z. We say that x is the least common ancestor of y and z if it is a common ancestor of y and z and further if x' is any other common ancestor of y and z then x' is an ancestor of x.

   Your aim is write a Python function `lcaVal` that takes a `AVLTree` (as defined in class) and two values u and v that appear in the tree and returns the value stored at the least common ancestor of the nodes where u and v are stored.

3. Modify the code written for `AVLTree` in the class to maintain the number visits to a node in the node. This is to be done as follows: Add an attribute `opCount` in the `AVLTree` which keeps track of the number of times that node has been visited during any operation (i.e. every time a insert or a search or a delete reaches a node, increment the count stores in that node) Add an additional function `report` to `AVLTree` that returns the sum of the `opCount` values of all the nodes in the tree. This is clearly the number of times the nodes have been accessed altogether by all the operations carried out so far.

4. Modify the code written for `hashtable` in Lecture 18 by keeping a count of the total number of cells visited by the various operations carried out on the table so far. Add an additional function `report` that returns this number.

5. Carry out a comparison of AVLTree versus Universal Hashing (the version described in class) as follows: Assume that the universe is $1, 2, \ldots, 5000$. Use the (Universal) `hashtable` class described in Lecture 18 to create a table with 5000 entries.

   (a) Using the `randint` function of Python to generate a set of 3000 (in the range $1, 2, \ldots, 5000$) values to insert into the table (some of these inserts may involve the same value and hence may not insert anything). Then, generate a random sequence of 3000 inserts or searches for a value (with equal probability).

   (b) Use the extended `AVLTree` described above to carry out all these operations and report the total number of nodes accessed.

   (c) Use the extended `hashtable` described above to carry out all these operations and report the total number of cells accessed.

6. Carry out a similar comparison using open hashing with binary probing as well.

7. Write a Python function `wordSearch` which does the following: It should read a file `inputText.txt` and store the words in this file in a set. Suppose the size of this set is m. It then builds a suitable Bloomfilter with size $8m$ and with 5 hash functions. It then inserts all the words in `inputText.txt` into this filter. Finally it opens another file `words.txt` which contains a sequence of words, one per line. It reports each word in `words.txt` that is not in the set holding the words from `inputText.txt` but in the Bloomfilter. That is, it reports all the false positives it encounters among the words in `words.txt`
