Assignment 2 (7.5%, 100 marks)

Please submit by 23:55, Nov 3, 23:55PM via virtual campus as a single zipped file.
Late assignments are accepted up to 24hs late with 30% penalty.

Background

Given two strings $a_0\ a_1 \ldots a_p$ and $b_1\ b_2 \ldots b_q$ on an ordered set of characters, we say that string a is lexicographically less than string b if
1)there exists an integer j, $0 <= j <= \min(p,q)$ such that $a_i = b_i$ for $i=0,\ldots,j-1$ and $aj < b_j$ or
2) $(p<q)$ and $ai=bi$ for all $i=0,1,\ldots p$,
This definition gives the same order as the usual alphabetical order used in a language dictionary.

In this assignment we will work with binary alphabet {0,1}. Here are some examples of lexicographical order for some binary strings: 01 < 10; 00101 < 01; 101 < 1010; 100<11.

In this assignment you will implement a type of tree called **trie** (I pronounce as "try" to not confuse with tree, but there is a debate on pronunciation since this is an invented name which came from the word re**trie**val). Because we are using binary strings our tries will be binary trees, but for larger alphabets the tree would have the same arity as the alphabet. Tries are used in file compression (e.g. Huffman code), to store words in a dictionary in a compact way, among other uses.

In a trie, the edge to the left child of a node corresponds to the character 0 and the edge to the right child of a node corresponds to the character 1. Any node in the tree corresponds to the string formed by the concatenation of characters in the path from the root to the node. The white nodes in Figure 1 show the string represented

In Figure 1 you can find an example with the trie that stores the strings: 0, 01, 000, 0101, 011, 111, 0100. For illustration purposes we explicitly label the edges and label the nodes corresponding to these strings (white nodes). Note that the grey nodes are intermediate nodes but their strings are not stored in the trie.
Figure 2 shows how the tree can be compactly stored without actually storing these strings/labels. It is enough to store the tree and a flag indicating whether the node is holding a string or is simply an intermediate node.
Note that tracing the edges that lead to a node allows us to recover the string it represents. For instance, if from the root we go left,right,right we know that this represents the string 011. Another example, if we go left and stop we know this represents string 0.

From the representation in Figure 2 we should be able to carefully use an appropriate tree traversal to print these strings in lexicographical order:
0, 000, 01, 0100, 0101, 011, 111.

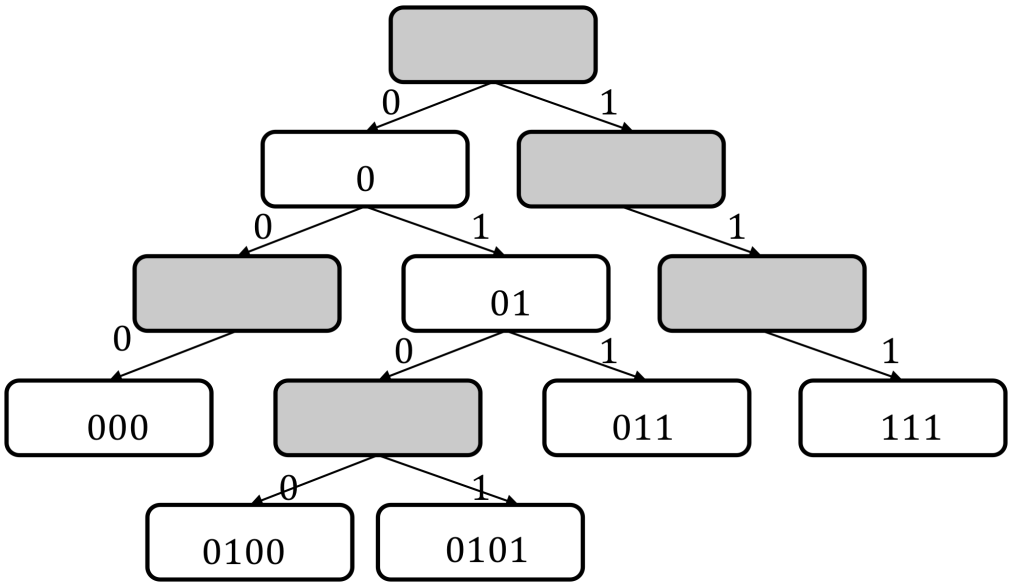

Figure 1: A trie storing strings 0, 01, 000, 011, 111, 0100, 0101 with strings/numbers used for illustration.

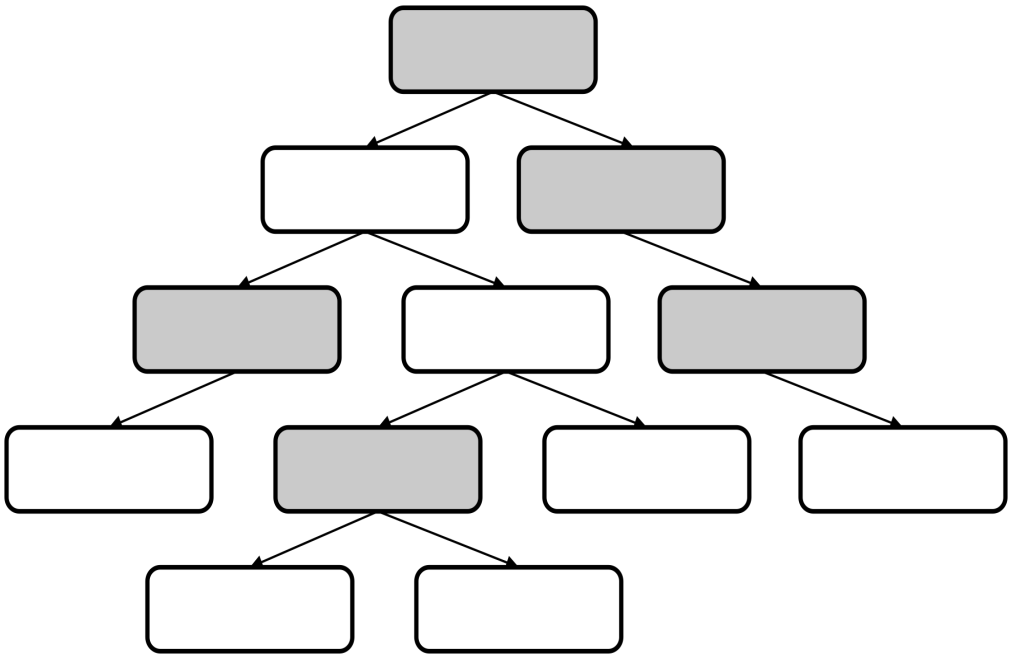

Figure 2: The data structure representing the same trie as in Figure 1 where nodes simply store a flag true or false to indicate whether the node corresponds to a strings or is an intermediate node, respectively.

Question 1) Using tries for sorting a list of binary strings in lexicographical order. You are given a startup code with classes TreeNode, MyTrie, TestTry. TreeNode is a node in the trie, MyTrie stores the trie as in **Figure 2** and TestTry is a sample code that does some tests for class MyTrie. TreeNode and TestTry must not be changed, while you can change class MyTrie. You are allowed to add extra methods or member variables, and add code to given methods, but the signature of the given methods should not be changed.
Most methods of MyTrie are not implemented and implementing them is part of your task.

Part 1A) (55 marks = 25+15+15) Implement the missing code in the following methods of class myTrie:

- `public boolean insert (String s):` This method "inserts" a string in the trie by adding appropriate nodes to the tree to represent the string. Note that the node representing the given string must have the flag "isUsed" set to true, while intermediate nodes that do not represent any of the strings inserted into the tree will have flag "isUsed" equal to false. If the string being inserted is already present in the trie, return false, otherwise insert the string and return true.
- `public boolean search(String s)` : Returns true if and only if the string is "stored" in the trie.
- `public void printStringsInLexicoOrder()` This method will print, in lexicographical order, all strings that are "stored" in the trie. This printing can be done by an appropriate tree traversal. Note that it may be useful to do some of this traversal recursively, and you are free to add another private auxiliary recursive method that will do most of this traversal/printing.
  Important: This method should be efficient and run in Theta(N) where N is the sum of the lengths of all the binary strings stored in the trie.

While you are testing and implementing, you may find useful to check that your tree is being built in the right way. For this purpose we provide the method printInOrder() that prints the tree using an inorder traversal that prints for each node the boolean value node.isUsed() .

Part 1B) (5 marks) Test the methods you have implemented.

Use the tester class testTrie to test the above methods. This class provides a method that given a set of binary strings, print them in lexicographical order.
Create another class testTrie2 which does a more comprehensive testing.
TAs may test your code with their own comprehensive testing to make sure your code is error free.

Question 2) Compressed tries.
The tree in Figure 1 could be optimized by deleting shaded nodes that have a single child (only delete nodes that do not hold a string in Fig 1) .This can be very beneficial if many of the entries have the same prefix . However, when we compress paths (several edges become one edge) we need to store a string (representing the bits in the compressed path) into the child node. This tree node that also holds a string is provided in class TreeNodeWithData should not be modified. The class with methods to be implemented is called MyCompressedTree. That stores the trie as in **Figure 4**.
See Figure 3 and 4 which are how Figure 1 and 2 are compressed.

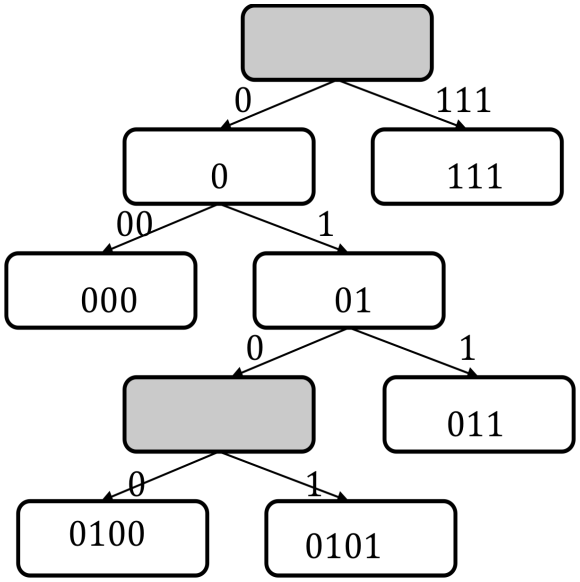

Figure 3: The compressed tree based on the picture in Figure 1.

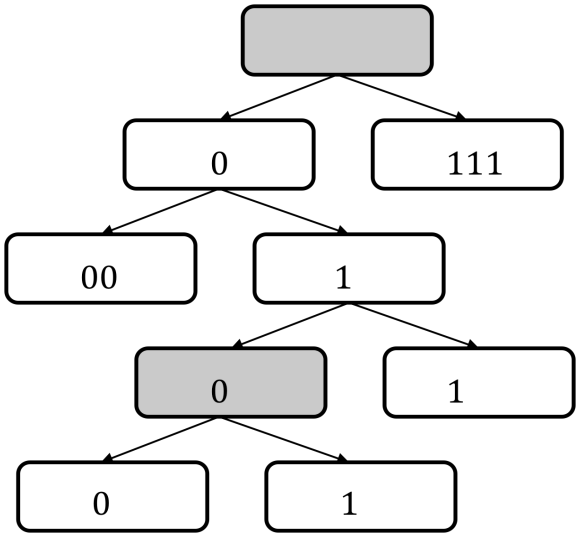

Figure 4: The data structure for the compressed trie in Figure 3 where nodes simply store the string representing the compressed edges.

Part 2A) (35 marks =25+10) Implement the following methods:

- `public MyCompressedTrie(MyTrie trie)`: this is a new constructor method that received a regular trie and constructs this compressed trie equivalent to it.
- `public void printStringsInLexicoOrder()`: prints the strings stored in the compressed trie in lexicographical order.

From the representation in Figure 4 we should be able to carefully use an appropriate tree traversal to print these strings in lexicographical order:
0, 000, 01, 0100, 0101, 011, 111.

Part 2B) (5 marks) Test the methods you have implemented, using testCompressedTrie class and another testCompresseTrie2 that you provide (similarly to 1B).

SUBMISSION INSTRUCTIONS

- All classes where you have added code must contain a comment header with your name, student number and uottawa id.
- Create a directory named jsmith033 where jsmith033 is your uottawa id/email.
- Store in this directory all the classes used in the assignment, namely: TreeNode, TreeNodeWithData, MyTrie, MyCompressedTrie, all the tester classes given or created by. You may also include a textfile called README.txt (optional) with any information that can be relevant for marking (known bugs, information on which parts are correct and which parts your know are incorrect or have not been implemented, whenever applicable). README.txt will be read at the discretion of the TA.
- Zip this directory and use jsmith033.zip as your assignment#2 submission.

Deviations from the submission instructions can have penalties of up to 20%.