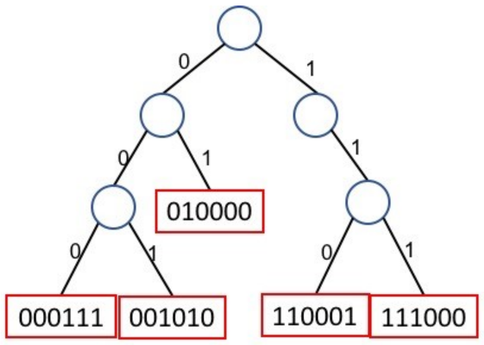
Data Structures and Algorithms CSC 172
Longest Prefix Matching Lab 5

Posted: 02/24/19
Due: 03/08/19 11:59PM

1 Problem

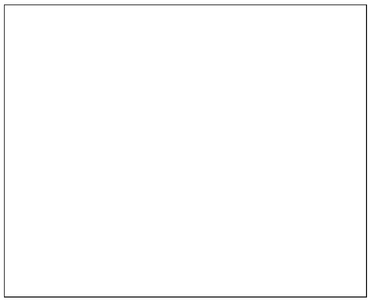
Let S be a set of strings. For a given string x, we want to find a string in S that has the longest prefix matching with x. Here we restrict strings to bit strings. As an example, if $\mathtt{x} = 001100$ is looked up in a given set $S = \{001010, 000111, 111000, 010000, 110001\}$, the string of interest is 001010. This problem is known as the longest prefix match (LPM) problem. In fact LPM has a wide range of applications in computer science including IP address lookup in routers, data compressions, coding, hashing, and so on. In this lab, we are going to implement an efficient data structure for LPM called a *trie*.



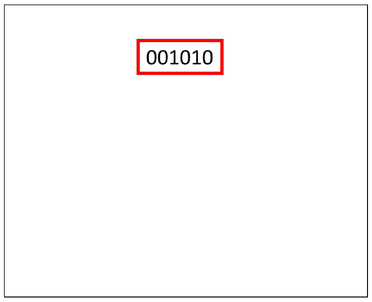
A binary trie (from re*trie*val), or simply trie, is a variant of the binary search tree that stores (binary) strings. Unlike the binary trees considered in the lecture, data will be stored in a leaf of the trie. The above diagram shows a trie containing five strings, where circles and rectangles represent internal nodes and leaf nodes, respectively. Hereafter we assume that all strings to be stored in a trie and checked against the trie are of the same length.

1.1 Construction of a Trie

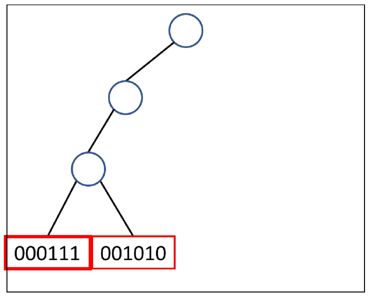
The trie over S is built as follows. For $|S| = 0$, the trie is empty. For $|S| = 1$, the trie is a single leaf. Now assume that the trie contains at least one string. To insert a new string x to the trie, we first need to locate a proper place to insert x. The search starts from the root and chooses either left or right child depending on the ith bit x_i of x, where $i = 0, 1, 2, ...$; left child if $x_i = 0$ and right if $x_i = 1$. For example, if $x = 110...$ then search chooses right, right, left, and so on. The search continues until it ends up with either a leaf or an empty node [1]. For the latter, the empty node is replaced by a new node containing new string x. In the other case where a leaf is found at the end of the search path, the path extends further down the trie. To do this, let $y = y_0 y_1 y_2 ...$ be the string stored in the leaf node. Now the leaf splits further by creating one or more internal nodes until both x and y can be inserted to the trie. i.e. until the first index i such that $x_i \neq y_i$ is found. Observe that all strings are stored in leaf nodes while internal nodes are branching nodes used to direct strings to their destinations (i.e. leaf nodes). The figure below illustrates how a trie grows as strings are inserted in the order 001010, 000111, 111000, 010000, and 110001. A newly inserted string is indicated by a rectangle with a thick outline in each figure.



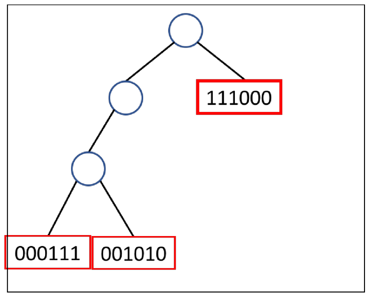
height = 0, size = 0



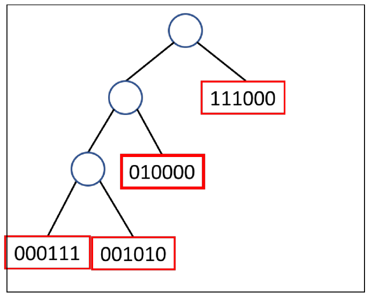
height = 1, size = 1



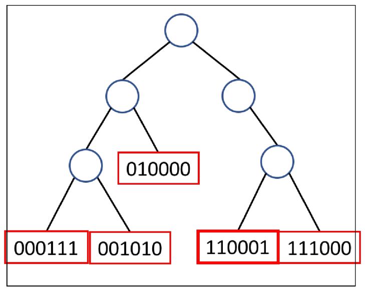
height = 4, size = 2



height = 4, size = 3



height = 4, size = 4



height = 4, size = 5

1.2 Search for a Longest Prefix Match

The search method, `search(T, x)` takes a non-empty trie `T` and string `x`, and returns a string stored in `T` that has a longest prefix match with `x`. If the search ends up with a leaf, the string in the node is the longest prefix match to be returned. On the other hand, if the search runs into an empty node, there are more than one strings in `T` that have the longest prefix match with `x`. In this case, the function must return the "closest" one to `x` among the multiple candidates. By the closest string to `x`, we mean the string that comes right before/after `x` when all candidate strings and `x` were listed in increasing lexicographic order. For example, given the trie in the figure and `x` = 100010, the search will encounter an empty node. In this case, the matching candidates are 110001 and 111000 The closest to `x` is 110001, which comes right after `x`.

1. Note that empty nodes are not internal nodes and they are not visible in the figure.

1.3 Requirements

Your program <u>must</u> implement the following methods:

1. `insert( trie, st )` inserts a string `st` into a non-empty trie `trie`. Returns either `true` or `false` indicating whether or not the insertion is successful.

2. `trieToList( trie )` creates a list of strings in `trie` in increasing lexicographic order. You are not allowed to use any kind of sort methods to sort the list. You may consider using the List ADT implemented for previous coding assignments.

3. `largest( trie )` returns the largest string in lexicographic order from the set of strings stored in `trie`. You can assume that `trie` is not empty.

4. `smallest( trie )` returns the smallest string in lexicographic order from the set of strings stored in `trie`. You may assume that `trie` is not empty.

5. `search( trie, st )` returns the string in `trie` that has the longest (and closest) prefix match with `st` as described in Section 1.2. You may assume that `trie` is not empty.

6. `size( trie )` returns the number of strings stored in the trie.

7. `height( trie )` returns the height of the trie.

1.4 Additional Requirements

For efficient implementation, the method `search()` should not be using the method `trieToList()`.

1.5 Provided Files

In order to make sure the structure of your code and the output meets the requirements, we'll provide two text files:

1. `commands` with a set of commands, one per line.

2. `01.ans` with the output you should get after execution of the commands in (1).

List of all possible commands in the commands file:

1. `insert`: takes as parameter the string to be added to the trie. Uses the method `insert()`. No output expected.
2. `search`: takes as parameter the string to search for in the trie. Uses the method `search()` to print out the longest prefix match with the given string.
3. `print`: takes no parameters. Uses the method `trieToList()` for printing out all strings in the trie in increasing lexicographic order.

4. `largest`: takes no parameters. Uses the method `largest()` to print out the largest string in the trie.
5. `smallest`: takes no parameters. Uses the method `smallest()` to print out the smallest string in the trie.
6. `height`: takes no parameters. Uses the method `height()` to print out the height of the trie.
7. `size`: takes no parameters. Uses the method `size()` to print out the number of strings stored in the trie.

Note: The *trie* doesn't appear as an argument in the list of commands as we assume all operations are performed on the unique trie you will create and maintain in your program.

2 Grading

Grading follows the general rules posted on BB for grading of coding assignments. Consider the following rules as requirements for a correct submission:

1. Your submission executes from the command line (shell) as follows:

   ```
   java Lab5 commands
   ```

   where `Lab5` is your executable and `commands` is a file with commands as described in 1.5.

2. All output of your code goes to the standard output (usual `System.out.println()`).
3. Make sure your output follows the same pattern of the provided output sample. Output for each command goes in a separate line. Separators of strings are just blank spaces, no tabs or other symbols.
4. The given files are only for you to understand our expectations on the output. Your code will be tested with our own command files for correctness.

3 Submission

You have to include a `README` file, with name(s) of authors in the first line and with a description of your work, `Lab5.java` (with your main) and all other relevant Java files. Compress all files in a single *zip* file named **[YourNetID]_Lab5.zip** and submit it to the appropriate folder on Blackboard.