
Please download the code for this exercise from http://lec.inf.ethz.ch/baug/informatik2/2016/ex/ex09/Assignment09.zip. We are not using the judge for this exercise. Please mail your TA for feedback to your solution.

9.1 Recursive Search Tree

For this part you use the code in the package `SearchTrees` which contains 3 files: SearchNode.java, SearchTree.java and TestSearchTree.java. You will work on the SearchTree.java file. Also have a look at the code in SearchNode.java which specifies a node in the tree. The code in TestSearchTree.java is used to test your implementation.

In the lecture you have seen the data structure *binary tree*. In the lecture the method `Search` was presented which iterates through the tree to find a node. The code from the lecture is contained in the downloaded exercise material. In this exercise you will implement an alternative version of the following iterative search method:

```
public SearchNode Search (int k){
    SearchNode n = root;
    while (n != null && k != n.key)
    {
        if (k < n.key) n=n.left;
        else n = n.right;
    }
    return n;
}
```

More specifically we ask you to implement the same functionality, but in a recursive fashion. Please complete the following two functions:

```
private SearchNode SearchRecursion(SearchNode current_root, int k) {
    //TODO implement this
}

public SearchNode SearchRecursive( int k ){
    SearchNode result;

    //TODO implement this

    return result;
}
```

`SearchRecursive` is the function being called instead of the orignal `Search` function. The function `SearchRecursive` must use the private method `SearchRecursion` to do the actual recursion. So `SearchRecursive` calls the function `SearchRecursion` once and then `SearchRecursion` recursively calls itself until it reached the specific node or a leaf node. In case a the node is found it returns the Node otherwise it returns null.

To verify the correctness of your implementation use the provided `TestSearchTree` class which will compare the output of the provided `Search` function with the output of your implementation of `SearchRecursive`.

9.2 T9 directory

Maybe some of you still experienced the so called T9 keyboards which used to be very popular before the raise of smart phones. Before smartphones, most phones only featured a key layout as seen in Figure 1: When typing messages a user would have to repeat a number to get to second or third

1	2 (abc)	3 (def)
4 (ghi)	5 (jkl)	6 (mno)
7 (pqrs)	8 (tuv)	9 (wxyz)
*	space	#

Figure 1: Common key layouts on older generations of phones

latter mapped on a number. So e.g to type a 'f' one would have to type '3' three times. This makes entering text a slow and cumbersome process. The problem becomes even more apparent if we consider the word 'hallo'. With this method, the user must press 4, 4, 3, 3, 5, 5, 5, then pause, then 5, 5, 5, 6, 6 and finally another 6. To speed up this process $T9$ was invented.

Predictive Text (also known as T9) is a system that aims to reduce the number of key presses necessary to enter text. Instead of pressing a number multiple times to reach some letters the user has to only press the number once. For instance, in order to obtain 'h' he would press 4 only once – instead of twice. This reduces the number of key presses in the case of 'hallo' from 12 to 5. `hallo` will be mapped to the combination 4,2,5,5,6.

The main drawback of this approach is that these mappings are not unique. Fore example 'gcjkm' would be mapped to the same number combination 4,2,5,5,6. Usually this was resolved by ordering words that map to the same combination by their likelihood of appearance and then the user could cycle through them using the * button.

For this part you use the code in the package `t9` which contains 3 files: T9Node.java, T9Tree.java and TestT9Dictionary.java. You will work on the T9Tree.java file. Also have a look at the code in T9Node.java which specifies a node in the tree. The code in TestT9Dictionary.java is used to test your implementation.

Your task in this exercise is to complete a very simple T9 implementation contained in the downloaded exercise material. Complete the following tasks:

- Study the code and understand the existing implementation. Understand that we are dealing with a tree here, where each Node has eight children (numbers 2-9 on the phone keyboard). Each node contains Vector (growing array) of Strings representing words which match the T9 sequence to this Node.
- Complete the methods `reverseNumber`, `addWord` and `findWords` in T9Tree.java which we describe in more detail below

reverseNumber

Method `reverseNumber` is used as a tool function in `findWords` and should be pretty straightforward.

```
//pre: a number
//post: the reversed of that number – eg. 12345 becomes 54321
private long reverseNumber(long number)
```

Hint: To get the last digit of a number you can use:

```
int last_digit = (number % 10);
```

addWord

```
public void addWord(String newword)
```

Within this function your task is to take a word - lets take e.g. our running example 'hello' and add it to the tree. To do so you would start at the tree root and look at the first letter of your word - in this case a 'h'. You map the 'h' to a number - you can use the `char2number` method for this purpose. This returns the number encoding for the string. Watch out that the indexing in the array will be off by two (so e.g. number encoding '2' should map to the zeroth child Node. So in our example 'h' would map to 4 - we subtract 2 to get the 2nd child Node. Watch out that this could be empty (null). If thats the case we create a new node - otherwise we follow the existing one. We now repeat this process for each letter of the word, and traveling down the tree in the process (and / or creating nodes in the process). Once we ran out of letters in our word, we reached the final node and we call the 'addString' method of this node.

findWords

```
//pre: a word encoded as a number
//post: returns all the strings contained in the tree that are encoded
//  by this number(can be more then one!)
public Vector<String> findWords(long number)
```

Finally we turn the problem around. The method `findWords` takes a number and returns all the words that would map to it. So for 42556 it could e.g. return hallo. To implement the function remember that the leading number always determines the direction of traversal in the tree. For instance, in the running example you would first look at the 4, then at the 2, 5 and so on. Therefore it might be beneficial to use the `reverseNumber` method you already implemented to make access to those digits easier. Pass through the tree guided by the digits of the number - should you encounter an empty child (null) - simply return null. Otherwise return all the strings stored in the final node.