Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Übungen zur Vorlesung Informatik II (D-BAUG) FS 2017

D. Sidler, F. Friedrich

http://lec.inf.ethz.ch/baug/informatik2/2017

Solution to exercise sheet # 9 **8.5.2017 – 16.5.2017**

Problem 9.1. Recursive Search Tree

For this task open the project on codeboard: https://codeboard.ethz.ch/inf2baugex09t01

There are three files: SearchNode.java, SearchTree.java, and Main.java. You will work on the class SearchTree, but have a look at the class SearchNode which specifies a node in the tree and the Main class which tests your implementation.

In the lecture you have seen the data structure *binary tree* and the method `Search` was presented. `Search` iterates through the tree to find a node. As you can see in the SearchTree class, the iterative version is already implemented:

```
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;
}
```

In this exercise you will implement an alternative version of the above iterative `SearchNode` method. 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 original `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 the node is found it returns the node otherwise it returns null.

The `Main` class is testing your implementation by comparing the output of provided `Search` function with the output of your implementation of `SearchRecursive`.

Once the two functions are functionally equal, submit your code.

Solution of Problem 9.1.

```
public SearchNode SearchRecursive(int k) {
    SearchNode result = SearchRecursion(root, k);
    return result;
}

private SearchNode SearchRecursion(SearchNode current_root, int k) {
    if (null != current_root && k != current_root.key) {
        if (k < current_root.key) {
            return SearchRecursion(current_root.left, k);
```

```
        } else {
            return SearchRecursion( current_root . right , k);
        }
    } else {
        return current_root ;
    }
}
```

Problem 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 latter

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

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 exercise open the template project at `https://codeboard.ethz.ch/inf2baugex09t02`. The project contains 3 files: T9Node.java, T9Tree.java and Main.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 Main.java is used to test your implementation and is explained further down.

Your task in this exercise is to complete a very simple T9 implementation by completing 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 a 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 that is 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.

You can test your program by un-commenting the annotation `@RunTests` at the beginning of the class `Main`. Once you pass the test you can submit your program.

To test your implementation, you can use the following commands which are implement in the `Main` class:

```
enter Adam
number 2326
samet9 Adam
```

The `enter` command takes a word (Adam) and adds it to the T9 dictionary (T9Tree). The `number` command takes a number (2326) and returns all words which map to this T9 number, e.g. Adam. The `samet9` command returns all words which have the same t9 mapping as the given word (Adam).

Solution of Problem 9.2.

```
//pre: it will have the tree in any state - only guarantee is that there is the root node
//     so be aware that you will have to create nodes while traversing if they do not exist
//post: changed the tree, such that it contains all the nodes necessary to encode the string
//          saves the string at the node with the right encoding - see handout for more details
public void addWord(String newWord) {
    T9Node curRoot = root;
    for (int i = 0; i < newWord.length(); i++) {
        char current_letter = newWord.charAt(i);
        int index = char2number(current_letter) - 2; //watch out for the -2 here
        T9Node child = curRoot.children[index];
        if (child == null) {
            child = new T9Node();
            curRoot.children[index] = child;
        }
        curRoot = child;
    }
    //after this loop cur_root should point at the T9 Node that responds to the number sequence
        given by our string
    curRoot.addString(newWord);
}


//pre: a number
//post: the reversed of that number - eg. 12345 becomes 54321
private long reverseNumber(long number) {
    long n = number;
    long reverse = 0;
    while (n != 0) {
        reverse = reverse * 10;
        reverse = reverse + n%10;
        n = n/10;
    }
    return reverse;
}

//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) {
    long curNumber = reverseNumber(number);
    T9Node curRoot = root;
    while (curNumber >= 1 ) {
        int digit = (int) (curNumber % 10);
        curNumber = curNumber / 10;

        curRoot = curRoot.children[digit - 2];
        if (curRoot == null)
            return null;
    }
    return curRoot.words;
}
```

Problem 9.3. Binary Trees

Consider the sequence $9, 5, 14, 7, 3, 16, 1, 4$. Insert these elements in this order into an empty

1. binary search tree (without any balancing).
2. binary Max-Heap.

Draw and give the *pre-*, *post-* and *in-order* traversal of the resulting trees.

Now insert the element 2 and draw the trees after the insertion.

Then remove the element 14 and draw the trees after the deletion.

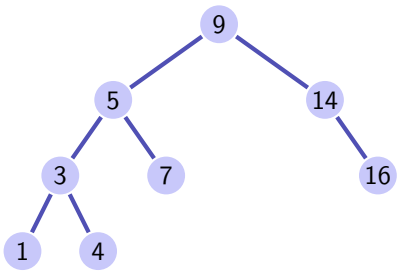
Hint: To remove an element in a heap, swap it with the last element. Then, depending on the parent element, *sink* or *bubble up* the element.

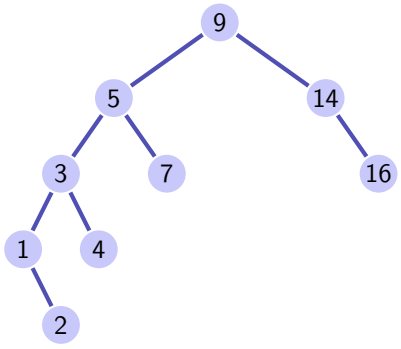
Submission link: https://codeboard.ethz.ch/inf2baugex09t03

Solution of Problem 9.3.

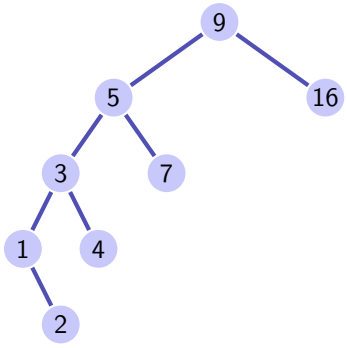
The binary tree before and after inserting 2:

1. Pre-order: 9 5 3 1 4 7 14 16
2. In-order: 1 3 4 5 7 9 14 16
3. Post-order: 1 4 3 7 5 16 14 9





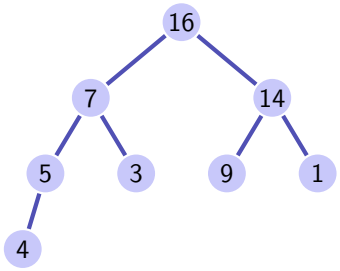
And after deleting 14.

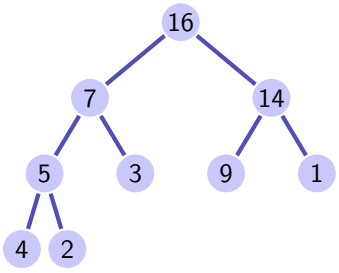


The binary Max-Heap before and after inserting 2:

1. Pre-order: 16 7 5 4 3 14 9 1

2. In-order: 4 5 7 3 16 9 14 1
3. Post-order: 4 5 3 7 9 1 14 16





And after deleting 14**:**