**Exercises: Trees Representation and Traversal (BFS and DFS) - Exercises**

This document defines the lab for ["Data Structures – Fundamentals (Java)" course @ Software University](https://softuni.bg/trainings/2812/data-structures-fundamentals-with-java-march-2020). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2199/04-Trees-Representation-and-Traversal-BFS-and-DFS-Exercises).

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards you can write and locally test your solution with the Java 13 standard, however **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after Java 10** release doing **otherwise** will result in **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the tests logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add new file add it in the same package of usage.

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions on uploading the solutions for each task. Submit as **.zip archive** the files contained inside **"...\src\main\java"** folder this should work for all tasks regardless of current DS implementation.

In order for the solution to compile the tests **successfully** the project **must** have **single** **Main.java** file containing single **public static void main(String[] args)** method even empty one within the **Main class**.

Some of the problem will have simple **Benchmark** **tests** inside the skeleton. You can try to run those with **different** **values** and **different** **implementations** in order to **observe** behaviour. However **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different** **data** **structures** **performance** on their **common** operations.

The Benchmark tool we are using is **JMH** (Java Microbenchmark Harness) and that is Java harness for building, running, and analyzing, **nano/micro/milli/macro** benchmarks written in Java and other languages targeting, the JVM.

**Additional** **information** can be found here: [JMH](https://openjdk.java.net/projects/code-tools/jmh/) and also there are other examples over the **internet**.

**Important:** when importing the skeleton **select** **import** **project** and then **select** **from** **maven** **module**, this way any following **dependencies** will be **automatically** **resolved**. The project has **NO** **default** **version** of **JDK so after the import you may (depends on some configurations) need to specify the SDK, you can download** **JDK 13** from [**HERE**](https://jdk.java.net/13/)**.**

## The Matrix

You can solve this problem by using **BFS** or **DFS**,you can even try the both approaches.

You are given a matrix (2D array) of lowercase alphanumeric characters (a-z, 0-9), a starting position – defined by a start row startRow and a start column startCol – and a filling symbol fillChar. Let’s call the symbol originally at startRow and startCol the startChar. Write a program, which, starting from the symbol at startRow and startCol, changes to fillChar every symbol in the matrix which:

* is equal to startChar AND
* can be reached from startChar by going up (row – 1), down (row + 1), left (col – 1) and right (col + 1) and “stepping” ONLY on symbols equal startChar

So, you basically start from startRow and startCol and can move either by changing the row OR column (not both at once, i.e. you can’t go diagonally) by 1, and can only go to positions which have the startChar written on them. Once you find all those positions, you change them to fillChar.

In other words, you need to implement something like the Fill tool in MS Paint, but for a 2D char array instead of a bitmap.

Study the code inside TheMatrix class and the tests. Implement the two methods: **solve ()** and **toOutputString ().**

### Input

There are two classes for this problem TheMatrix and TheMatrixTest .You have to study the code the input is passed upon creation of TheMatrix object inside the tests.

Two integers will be entered – the number R of rows and number C of columns.

On each of the next R lines, C characters separated by single spaces will be entered – the symbols of the Rth row of the matrix, starting from the 0th column and ending at the C-1 column.

On the next line, a single character – the fillChar – will be entered.

On the last line, two integers – startRow and startCol – separated by a single space, will be entered.

### Output

**Implement the toOutputString ().**

The output should consist of R lines, each consisting of exactly C characters, **NOT SEPARATED** by spaces, representing the matrix after the fill operation has been finished.

### Constraints

0 < R, C < 20   
0 <= startRow < R   
0 <= startCol < C

All symbols in the input matrix will be lowercase alphanumerics (a-z, 0-9). The fillChar will also be alphanumeric and lowercase.

The total running time of your program should be no more than 0.1s

The total memory allowed for use by your program is 5MB

### Examples

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 5 3  a a a  a a a  a b a  a b a  a b a  x  0 0 | xxx  xxx  xbx  xbx  xbx |
| 5 3  a a a  a a a  a b a  a b a  a b a  x  2 1 | aaa  aaa  axa  axa  axa |
| 5 6  o o 1 1 o o  o 1 o o 1 o  1 o o o o 1  o 1 o o 1 o  o o 1 1 o o  3  2 1 | oo11oo  o1331o  133331  o1331o  oo11oo |
| 5 6  o o o o o o  o o o 1 o o  o o 1 o 1 1  o 1 1 w 1 o  1 o o o o o  z  4 1 | oooooo  ooo1oo  oo1o11  o11w1z  1zzzzz |
| 5 6  o 1 o o 1 o  o 1 o o 1 o  o 1 1 1 1 o  o 1 o w 1 o  o o o o o o  z  4 0 | z1oo1z  z1oo1z  z1111z  z1zw1z  zzzzzz |

## Tree Problems Overview

You are given a **tree of N nodes** represented as a set of N-1 pairs of nodes (parent node, child node). For each problem you have specified definition of the implementation required to complete the task. You can use different approaches which you find suitable for the task you are solving. In general all the problems require basic tree knowledge and understanding of DFS and BFS traversal algorithms.

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Comments** | **Tree** | **Definitions** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6  27  43 | N = 9  Nodes:  7 🡪 19,  7 🡪 21,  7 🡪 14,  19 🡪 1,  19 🡪 12,  19 🡪 31,  14 🡪 23,  14 🡪 6  P = 27  S = 43 |  | Root node: 7  Leaf nodes: 1, 6, 12, 21, 23, 31  Middle nodes: 14, 19  Leftmost deepest node: 1  Longest path: 7 -> 19 -> 1 (length = 3)  Paths of sum 27: 7 -> 19 -> 1 7 -> 14 -> 6  Subtrees of sum 43: 14 + 23 + 6 |

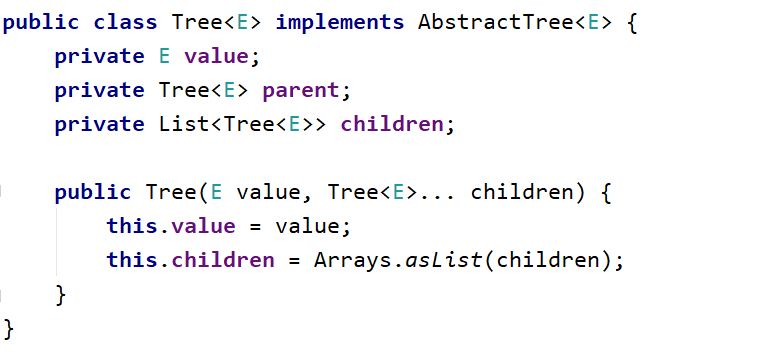
## Create Tree

Write a program to create a tree and find its **root** node then return it as an entry point to the DS:

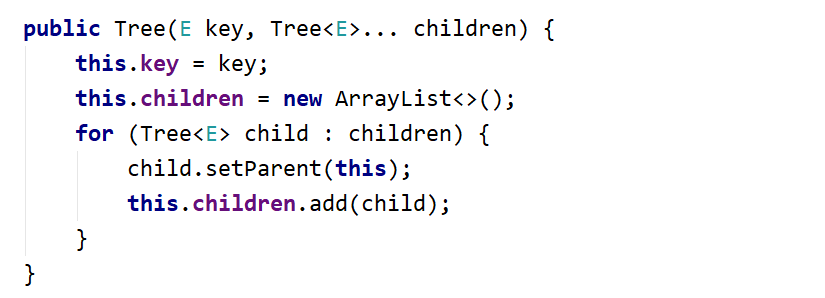
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | Root node: 7 |  |

Hints

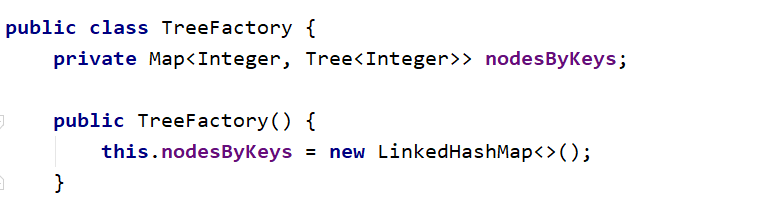
Use the recursive Tree<E> definition. Keep the **value**, **parent** and **children** for each tree node:



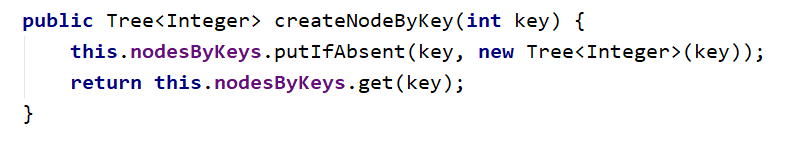
Modify the Tree<T> **constructor** to **assign a parent** for each child node:



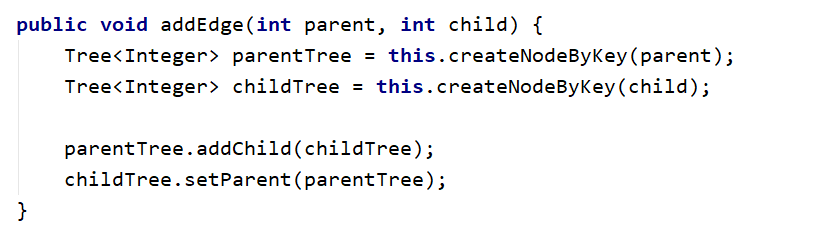
Use a **Map or Array** to map nodes by their value. This will allow you to find the tree nodes during the tree construction:



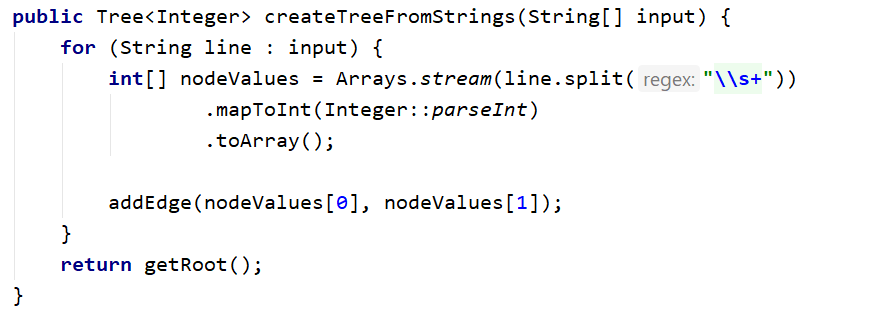
Write a method to **find the tree node by its value or create a new node** if it does not exist:



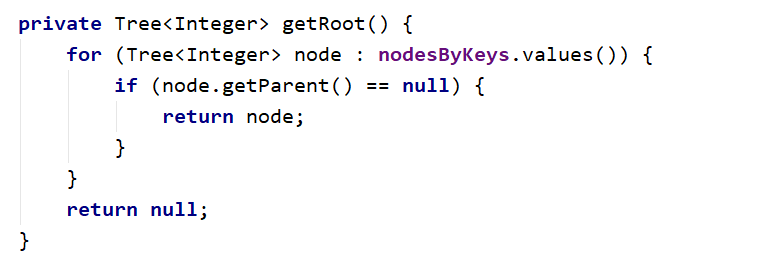
Create a method for adding an edge to the tree:



Now you are ready to **create the tree**. You are given the **tree edges** (parent + child). Use the map to lookup the parent and child nodes by their values:



Finally, you can find the root (the node that has no parent)



## Tree As String

Write a program to create tree from and print it in the following format (each level indented +2 spaces):

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | 7  19  1  12  31  21  14  23  6 |  |

Hints

Find the root and recursively print the tree

## Leaf Nodes

Write a program to read the tree and find all **leaf** nodes (in increasing order):

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | Leaf nodes: 1 6 12 21 23 31 |  |

Hints

Find the all nodes that have no children

## Middle Nodes

Write a program to read the tree and find all **middle** nodes (in increasing order):

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | Middle nodes: 14 19 |  |

Hints

We can say that **middle** **nodes** are the once that have a **parent** **and at least one child at the same time.**

## \* Deepest Node

Write a program to read the tree and find its deepest node (leftmost):

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | Deepest node: 1 |  |

## Longest Path

Find the **longest path** in the tree (the leftmost if several paths have the same longest length)

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6 | Longest path: 7 19 1 |  |

## All Paths With a Given Sum

Find all paths in the tree with **given sum** of their nodes (from the leftmost to the rightmost)

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6  27 | Paths of sum 27:  7 19 1  7 14 6 |  |

## \* All Subtrees With a Given Sum

Find all **subtrees with given sum** of their nodes (from the leftmost to the rightmost). Print subtrees in **pre-order** sequence

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Tree** |
| 9  7 19  7 21  7 14  19 1  19 12  19 31  14 23  14 6  43 | Subtrees of sum 43:  14 23 6 |  |