# CS25100: Data Structures and Algorithms, Spring 2015

## Project 1: Reading Trees

Due date: Jan 29th, 2015

---

PLEASE READ THE DESCRIPTION VERY CAREFULLY. YOU WILL STAND TO LOSE SIGNIFICANT AMOUNTS OF POINTS IF YOU DO NOT FOLLOW INSTRUCTIONS PERFECTLY IN YOUR PROJECTS, WE WILL NOT BE LENIENT!

## Conceptual Overview

We define a textual representation of binary trees whose nodes are labeled with integers. You are to read such a representation from the input and build from it an internal representation where the nodes have left and right pointers plus a data field for the integer label. You may assume that the input is correctly formatted.

Having built the internal representation, you will read several strings consisting of any combination of the letters L and R. Each such string spells a path through the tree, beginning at the root. You are to print the sequence of integers encountered along the path. If the path is shorter than the string, print an additional asterisk.

## External Tree Representation

The tree description uses the parentheses, '(' and ')', a comma ',', and integers. Blanks are ignored (except in integers). Each string is a single input line terminated by the carriage return. The textual binary tree representation is defined as follows:

- `( )` denotes the empty tree (internally a null pointer)
- `( n )` as well as `( n , , )` denote a leaf labeled with integer 'n'. Note that the forms `(n,(),())` and `(n,(),)` and `(n,,())` are excluded.
- If 'T1' and 'T2' denote two textual binary tree representations, then so does `( n , T1 , T2 )`. This represents a binary tree with node 'n' at its root, and 'T1' as its left subtree and 'T2' as its right subtree.
- Nothing else textually denotes a binary tree.

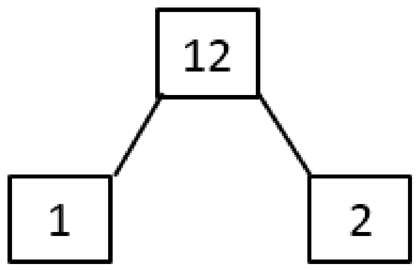
Having built the internal representation, you will read several strings consisting of any combination of the letters 'L' and 'R'. Each such string spells a path through the tree, beginning at the root. You are to print the sequence of integers encountered along the path. If the path is shorter than the string, print an additional asterisk.

## Example Textual Specifications

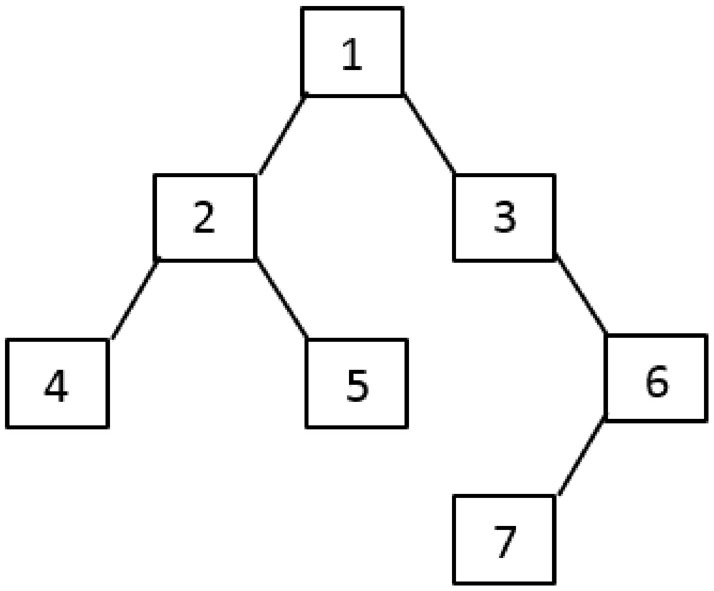
- `( 15)` denotes a tree consisting of a single node, labeled 15
- `(13,,)` denotes a tree consisting of a single node, labeled 13
- `(12,(1),(2))` denotes a balanced binary tree with root label 12, left leaf labeled 1, and right leaf labeled 2
- `(1,(2,(4),(5)),(3,,(6,(7,,),)))` denotes a tree with root label 1. The left subtree is a balanced binary tree of height 2, with root label 2. The left subtree is a balanced binary tree, with root label 2, and having 4 and 5 as its left child and right child, respectively. The right subtree is not balanced and has root 3. 6 is the right child of the root of this subtree and a node labeled 7 is the left child of the node which is labeled 6.

## Visual Examples

The image contains some examples of text representation and associated tree structures:



(12,(1),(2))



(1,(2,(4),(5,,)),(3,,(6,(7),)))

Examples of path evaluation, for the tree `x = (1,(2,(4),(5,,)),(3,,(6,(7),)))`

- L: 1 2
- LR: 1 2 5
- RRRL: 1 3 6 *
- RL: 1 3 *
- RLRLR: 1 3 *

## Tasks

There are three tasks you will need to complete.

Task 1 - Text Input (20 points)

Read in the string, delete all blanks, then return the pruned string.

You will edit the code in the file `TreeParser.java`. Specifically, you will write code under the function `getCleanedString`. The function `getCleanedString` should take a string and return the string without any spaces in it. Write code under the function

```
public static String getCleanedString(String text) {
}
```

For example, if text is " (1,(2, (4), (5,, )),(3, ,(6,(7) ,))) ", you should return "(1,(2,(4),(5,,)),(3,,(6,(7),)))".

Task 2 - Tree Construction (40 points)

Build the tree. One way to do this would be to use a recursive function that parses the string. If you want to pursue this approach, you will probably want to have your recursive procedure take a position of the input string as an argument and attempt to build a subtree from that point. Of course, your procedure will have to ensure that the subtree created will have to be placed at the right place in the final tree that you want to build.

Once again, you will edit the code in the file `TreeParser.java`. Specifically, you will write code under the function `createTree`. The function `createTree` takes in a string and will need to return an object of type `Node`. Remember, the `Node` object has a fields called `leftChild` and `rightChild` both of type `Node`. Thus, a tree can be represented just by the root node. To illustrate, if `rootNode`' is the variable which represents the root of the tree, then you would access descendants of the root as `rootNode.getLeftChild().getRightChild().getLeftChild()...` Write code under the function

```
    public static Node createTree(String treeRepresentationText) {
    }
```

Task 3 - Path Evaluations (30 points)

Finally, once again, you will edit the code in the file `TreeParser.java`. Read the path input, delete white spaces. Note that if you have already written the `getCleanedString` function, there is nothing you need to do for the input cleaning part. After this is done, you will call the function `traversePath`. The function `traversePath` takes in the root node of the tree as well as a pruned direction string and the output must be the result of the path evaluation, as described, presented as a single string. When creating the path evaluation, please make sure to separate values by just one space. For example, if the values we see on a path are 1, 3, 6, the output of your function will just be the single string "1 3 6". No extra spaces or extra text. This function could be done with a loop or recursively. A loop appears to be simpler. Write code under the function

```
    public static String traversePath(Node root, String direction) {
    }
```

## Input:

Input will be via standard input. Below is how one round of input will look like.

```
    <Number of test cases (n)>
    <Textual representation of Tree 1>
    <Number of path queries on Tree 1 (nt1)>
    <Query 1 on Tree 1>
    <Query 2 on Tree 1>
    ...
    <Query nt1 on Tree 1>
    <Textual representation of Tree 2>
    <Number of path queries on Tree 2 (nt2)>
    <Query 1 on Tree 1>
    <Query 2 on Tree 1>
    ...
    <Query nt2 on Tree 1>
    ...
```

## Output:

Output should be to standard output. Below is how one round of output should look like.

```
    Testcase 1: <Cleaned textual representation of Tree 1>
    Output for testcase 1
```

```
<Cleaned query 1 on Tree 1>: <Output of Query 1 on Tree 1>
<Cleaned query 2 on Tree 1>: <Output of Query 2 on Tree 1>
...
<Cleaned query nt1 on Tree 1>: <Output of Query nt1 on Tree 1>
Testcase 2: <Cleaned textual representation of Tree 2>
Output for testcase 2
<Cleaned query 1 on Tree 2>: <Output of Query 1 on Tree 2>
<Cleaned query 2 on Tree 2>: <Output of Query 2 on Tree 2>
...
<Cleaned query nt2 on Tree 1>: <Output of Query nt2 on Tree 2>
```

## Sample Input/Output:

Let's see an example. For the input

```
2
(101,    (212,   (434),(   545 ,,)),(323,,(656,(767),)))
8
L
LL
LL L
LR
R R    RL
RL
RRL
R LRLR
(71,(6591),(122127887))
3
L
 R
RR
```

the output must be

```
Testcase 1: (101,(212,(434),(545,,)),(323,,(656,(767),)))
Output for testcase 1
L: 101 212
LL: 101 212 434
LLL: 101 212 434 *
LR: 101 212 545
RRRL: 101 323 656 *
RL: 101 323 *
RRL: 101 323 656 767
RLRLR: 101 323 *
Testcase 2: (71,(6591),(122127887))
Output for testcase 2
L: 71 6591
R: 71 122127887
RR: 71 122127887 *
```

## General Instructions:

- We have provided skeleton code and ancillary files. Your changes will have to be on top of what has already been given to you. Writing your own code will cause you to lose points.
- Do not change file structures, file names, folder names or folder structures.
- You are free to create as many Java classes and files as you wish, and as many functions as you wish. But do not change what is already written in the skeleton. You can also create test scripts if you like.

- Always, you will need to write code only when you see "TODO" markers. Whenever there is a TODO marker, we will make sure to write some extra comments under the TODO markers and this will tell you what you really need to do. What this means is that parts of the task will already be done for you in the skeleton. Just to re-iterate, you can create extra classses/functions that you might want to use/call.

- Please carefully note the input and output formats. Also note the exact casing of the alphabets and the exact number of spaces and newlines to be added at various points in the output. Even one missing or extra character will cause you to lose points in the projects. *In this project, the skeleton code is doing this for you; you only need to focus on the data structures and algorithmic aspects.*

- Different people write/edit code in different ways, so:
    - If you are going to be using Eclipse, read this.
    - If you are going to be using Dr. Java, read this.
    - If you are going to be coding in a low-tech manner using system editors like Vim, read this.
    - If you are going to be using something else, you are on your own.

## Project Specific Instructions:

At a high level, there are going to be four things you will need to do for this project:

- Coding: Download the skeleton code which is available here. Unzip it and edit the code present in the folder (the folder should be `251Spring15P1`). Make sure you are acquainted with the process of editing code by reading the instructions in the previous section. You will fill in the empty functions in `TreeParser.java`.

- Testing: You will begin by using the sanity test script provided to you. It is present in `251Spring15P1`.

    Assuming you are in `251Spring15P1`, just running:

    ```
    $sh sanity_test.sh 1
    ```

    will tell you whether or not your program conforms to basic output requirements. Remember, we require a very specific output format and this will test if your program is passing the simplest test case. Running

    ```
    $sh sanity_test.sh 2
    ```

    will test your program more thoroughly. The file contains a few tricky test cases.

    If your program is perfect, you will see an output which says

    ```
    $Sanity Test1 Passed!
    ```

    or

    ```
    $Sanity Test2 Passed!
    ```

    correspondingly.

  You should also create your own test cases and test your program against them. The program takes input from standard input, so you should have to feed in your test cases accordingly. It is highly recommended that you use redirection to test your programs. Talk to your TA if you want to learn how to do it. You could also just look at what `sanity_test.sh` is doing to understand how to use redirection.

Move ahead only once you are convinced of the correctness of your work. PLEASE REMEMBER THAT `sanity_test.sh` IS THERE ONLY TO TEST YOUR PROGRAM AGAINST ELEMENTARY TEST CASES. It is up to you to construct all kinds of test cases and validate your work.

- Submission: Submit your solution before January 29th 11:59 pm (note that when the time becomes 12:00 am, the date timestamp changes to the next day; please read the late policy carefully). The simple program "turnin" will be used to submit assignments in this course. Please follow these instructions very carefully as any small error can make your submission invalid:

  Once you unzipped the skeleton code provided, you will have a folder which contains everything. This folder should be `251Spring15P1`. You will have edited the code inside `251Spring15P1` (without changing the structure) and will also have placed `Report.pdf` inside it. You might have also created some other files inside the folder.

  Log into (or `ssh` to) `data.cs.purdue.edu` or `lore.cs.purdue.edu` (you can also use puTTY). Then, please follow these steps:

  - cd into `251Spring15P1` and make sure all the files you wish to submit are there. WE WILL NOT BE LENIENT WITH PEOPLE WHO SUBMIT THE WRONG FOLDER OR FORGET TO SUBMIT REQUIRED FILES.
  - While in the upper level directory (e.g. if the files are in `/homes/jdoe/mycode/251Spring15P1`, cd to `/homes/jdoe/mycode/`), execute the following command:

    ```
    $turnin -c cs251 -p project1 251Spring15P1
    ```

  - Keep in mind that old submissions are completely overwritten with new ones whenever you execute this command, and the timestamp we will use to check your submission time according to our late policy is the time of last submission (turnin does not keep track of timestamps or any contents of any earlier submissions).
  - If you wish to, you can verify your submission by executing the following command:

    ```
    $turnin -v -c cs251 -p project1
    ```

    Do not forget the `-v` flag here, as otherwise your submission will be completely replaced with a blank (an empty) submission and your earlier timestamp overwritten.