Homework 2

You may **not** use java's built-in queues, stacks or trees.
You also may not use any of java's built-in lists to implement a stack, queue or tree.
Doing either of these is cheating and results in an F in the course. If in doubt, ask on the forum.

The problems are in order of difficulty. The final problem is only worth a few points and intended for A students. For all of these questions, the difficulty is the design/problem solving part, not the program. You may ask the TAs and friends for help with java syntax but not on how to solve the problems.

Permissions

1. You are allowed to add any methods or properties you want.
2. You are not allowed to remove/not implement the ones given in the class diagrams.
3. You may not modify test cases. More precisely, we will always erase your test cases when grading and use our own, so if your code only works on modified tests cases it will not work with ours.
4. If the goal of a question is to implement a data structure, you may not delegate any part of that to a built-in java data structure (this counts as cheating).
5. You may use built-in data structures if it is needed for a public interface (e.g., if you are expected to take in or give out a List, you can import and use List).
6. You may import code from one of your questions to another. This is new for this assignment. Prior to this the grading script didn't allow this. Hopefully we've fixed that.

Rules

1. Submit a zip (not rar, stuffit, etc.) file containing the java source code you wrote (i.e., we don't need your copy of the tests, we already have those).
2. If your code breaks the grading script, you will get a 0 for the assignment. Concretely, do not submit code that doesn't compile or has an infinite loop. Either don't submit those files or make them compile (just return null or -1).

---

1. Tree (5 points)

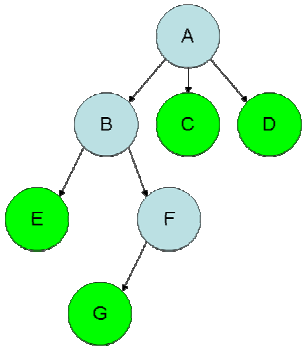
Tests: General Trees

Rank: Easy (since you already made it in lab and are just copying and pasting)

MyGeneralTree<T>
+ root : MyGeneralTreeNode<T>
+ getHeight() : int + getNumberOfBranchNodes() : int + getNumberOfLeaves() : int + getNumberOfNodes() : int + getPostOrderStructure() : String + getPreOrderStructure() : String + getStructure() : String

MyGeneralTreeNode<T>
+ data : T + children : List<MyGeneralTreeNode<T>>
...whatever you need...

This is just the normal tree you've done in lab and that we've done in class a trillion times. Feel free to use your lab code. As a reminder of what everything does…



`getHeight()`	The length of the longest path. In the example, 4 (A→B→F→G).
`getNumberOfBranchNodes()`	The number of non-leaf nodes. In the example, the 3 blue nodes.
`getNumberOfLeaves()`	The number of nodes that have no children. In the example, the 4 green nodes.
`getNumberOfNodes()`	In the example, 7.
`getPostOrderStructure()`	Post-order order. EGFBCDA
`getPreOrderStructure()`	Depth-first order. ABEFGCD
`getStructure()`	Return a string showing the tree structure.

Pre- and post-order strings are just the data put together, no spaces, commas, etc.

For tree structure, use the format we used in the lab. As a reminder, it looks like this:

```
|-- A
    |-- B
        |-- E
        |-- F
            |-- G
    |-- C
    |-- D
```

---

2. Stack Trace (10 points)

Tests: Stacks
Rank: Easy (if you already have a stack)

When your code breaks or test fails, java prints off a stack trace that shows the method that failed, the method that called it, the method that called that method, etc. all the way up to the `main()` method. That list of method names is called the call stack. You're going to make a poor man's version of that.

StackTrace
- instance : StackTrace - callStack : MyStack
- StackTrace() : ctor + getSingleton() : StackTrace + clear() : void + getCallDepth() : int + getCallStack() : List<String> + peek() : String + pop() : String + push(String methodSignature) : void

Notes:

- Uses the singleton pattern. This was explained in the class. Implications:
  - The constructor is private
  - `instance` and `getSingleton()` are static
- Since this calls for a `MyStack`, you'll need to make one. We won't test it so design it however you like. However, you must implement it – you cannot use any built-in java stacks or lists for this.

`StackTrace()`	The constructor is private. There are no other constructors. The class cannot be externally instantiated.
`getSingleton()`	Lazy loads (gets if it exists, creates if it doesn't) the static, singleton instance variable `instance`.
`clear()`	Reset the state of the stack.
`getCallDepth()`	How deep the program currently is. In other words, it's the stack size.
`getCallStack()`	Return a list representing the items in the call stack. The first string is the first (main) method, the last string is the current/most recent method.
`peek()`	Show the current method you're in.

`pop()`	The current method has been completed and is removed from the call stack.
`push(String methodSignature)`	Add the current method to the call stack

You might want/need to add a few methods to your MyStack to make this easier.

---

3. Tree Creation (20 points)

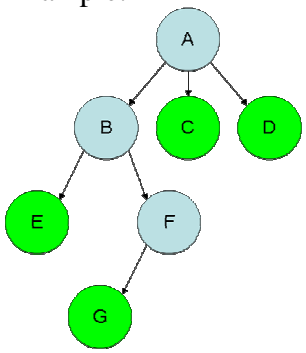
Tests: General trees, dictionaries
Rank: Medium

Did you see all that code needed to add data to trees? It's ugly and very manual. Let's automate that. TreeLoader takes a list of strings describing the nodes and returns a full populated tree.

TreeLoader
`+ createGeneralTree(List<String> nodeDescriptors) : MyGeneralTree<String>`

The input strings will be in the format "id : parentID : data" where ID refers to a meaningless but unique ID for a node. Always treat data as a string. Skip lines that start with #. The root node ALWAYS has an ID of 0. Strings are not guaranteed to be in any order. IDs are not guaranteed to be consecutive. IDs are not guaranteed to be numeric. Data can be any length. You will never see a : character except as a delimiter.

Example:




```
7 : 6 : G
0 : : A
2 : 0 : B
4 : 0 : D
  6 : 2 : F
apple : 2 : E
3 : 0 : C
```

Some help with Strings:

- `line.split(" ")` splits line into an array of Strings. In this example, the line is split on spaces (you can change the delimiter to whatever you want). See intellisense (the API that pops up while you're typing) or Google for more help.
- `line.trim()` removes white space (spaces, tabs, etc.) from the front and end of the string. " A B C " becomes "A B C".
- As mentioned in class, Strings are immutable and thus String methods are non-mutating. If you forgot what that means, you'll figure it out when you test your code.

You probably don't want to try to go directly from String list to a tree. If you use an intermediate representation, ask yourself what data / format / data structure would make it easy to piece together a tree, or maybe just associate data of any kind. Since you have three pieces of data to track for each node, you might need more than one intermediate representation. This is an easy problem if you figure out the trick to it.

---

4. File System (20 points)

Tests: Querying trees, stacks
Rank: Medium

You know what a directory tree is. You have a folder, it has folders underneath it, those might have folders, etc. For example, your movies folder might look like this:

```
|-- movies
    |-- sci fi
        |-- BSG S01E01.avi
        |-- GoT S03E09.mpg
        |-- Buffy S06E07.avi
    |-- action
        |-- Sucker Punch.mpg
        |-- martial arts
            |-- The Raid.mpg
            |-- Romeo Must Die.mov
        |-- The Killer.mov
    |-- romance
        |-- Four Funerals and a Wedding.avi
        |-- Baytown Outlaws.avi
        |-- Kill Bill.mpg
```

You know all those times you asked yourself, "i wonder how many movies i have"? We're going to find out. Build a `FileSystemTree` that contains `Directory` and `File` objects. Once you have that, we can ask it questions, change names, move things around, etc.

One way to do that is to make a set of specialized objects representing directories and files. i considered that but for reasons you'll soon understand i think you'll find it easier to implement if we just use strings. The data of every node is a single String.

Your tree will look something like this:

```
|-- directory
    |-- name
        |-- Movies and Pictures
    |-- directory
        |-- name
            |-- sci fi
        |-- file
            |-- name
                |--- GoT S03E09.mpeg
            |-- date
                |--- 20130602
            |-- size
                |--- 1400123
        |-- file
```

```
        |-- name
            |--- Buffy S06E07.avi
        |-- date
            |--- 20010906
        |-- size
            |--- 256123
```

The height of this tree is 5 and includes paths such as directory→directory→file→name→Buffy. There are two file nodes and two directory nodes. Directories have one property (name) plus an arbitrary number of files and directories under them. Each file has exactly three named properties and each property has exactly one child node. We will not give you any trees that do not follow these rules.

The above tree represents two directories ("Movies and Pictures", "sci fi") and two files in the "sci fi" directory (GoT and Buffy). The structure is similar to that used in XML (if you don't know what XML is, it's how a lot of data on The Web is stored). The XML of the above might be:

```
<directory>
    <name>
        Movies and Pictures
    </name>
    <directory>
        <name>
            sci fi
        <name>
        <file>
            <name>
                GoT S03E09.mpg
            </name>
            <date>
                20130602
            </date>
            <size>
                1400123
            </size>
        </file>
        <file>
            <name>
                Buffy S06E07.avi
            </name>
            <date>
                20010906
            </date>
            <size>
                256123
            </size>
        </file>
    </directory>
</directory>
```

Not coincidentally, for this question you need to be able to read and write the above format. Just to make sure you see it, every data field starts with an opening tag and closes with a closing tag. We colored a few tags to help you see that they can be nested inside one another.

A few rules:

1. If the current line is an open tag, the next line must be data, another opening tag or a closing tag.
2. If the current line is a close tag, the next line must be an open tag or another c lose tag.
3. If the current line is data, the next line must be a close tag.

All tags come in pairs. If you encounter a close tag, how do you know which open tag it matches. You might try to match on name. Don't, it won't work. Is there some other rule about which of a list of open tags matches a given close tag? If you can figure that out, you'll have a good hint as to what data structure you'll need to track an important subset of the information.

Since our focus is not string parsing, we'll make the file format simple:

- All tags will be lower case.
- The first tag encountered represents the highest-level item. There is only one top-level item. It is the root of the tree.
- Every element is on its own line. There will not be any lines with multiple items (e.g., not "`<name>X</name>`").
- There may be extra spaces at the start and end of each line (trim first!) but not within a tag (i.e., not "`< name >`").
- When you write out the file, don't worry about indenting – different levels don't need to be spaced over a different amount as in the above. It makes it prettier but doesn't matter to your tree or file parser.

Just in case anybody gets any funny ideas, you may not use sax, dom or any other existing XML parser. If you use someone else's library, that's cheating which means you fail the class.

FileSystemNode
+ data : String + children : List<FileSystemNode>
+ FileSystemNode() : ctor + FileSystemNode(String data) : ctor

FileSystem
+ root : FileSystemNode
+ deserialize(List<String> lines) : void + getDirectories() : List<String> + getFiles() : List<String> + getFilesLargerThan(int size) : List<String> + getFilesNewerThan(int date) : List<String>

```
+ getNestingLevel() : int
+ getNumberOfDirectories() : int
+ getNumberOfFiles() : int
+ getNumberOfFilesLargerThan(int size) : int
+ getNumberOfFilesNewerThan(int date) : int
+ getNumberOfMovies() : int
+ getNumberOfMoviesNewerThan(int date) : int
+ getNumberOfPictures() : int
+ getTreeStructure() : String
+ insertVirus(String fileName, String virusCode,
    int virusSize) : void
+ rename(String oldName, String newName) : void
+ serialize() : List<String> lines
```

`deserialize`	Given an ordered set of lines representing XML data, turn yourself into the corresponding tree.
`getDirectories`	Return the names of all directories.
`getFiles`	Return the names of all files.
`getFilesLargerThan`	Return the names of all files where size >= the specified size. The size of a file is stored in the only child of the node where data="size". Note that you'll need to convert the size from a String to an integer.
`getFilesNewerThan`	Return the names of all files where date >= the specified date. To make life easy, all dates will be specified as Julian values (i.e., they're just ints, larger numbers are more recent dates).
`getNestingLevel`	Returns the length of the deepest level of a directory. It is equivalent to the height of the tree if only directories are counted (files and properties don't count). Example, in the path "A\B\C\aFile", aFile is three directories down.
`getNumberOfDirectories`	Return the number of directories (i.e., entries where data = "directory").
`getNumberOfFiles`	Return the number of directories (i.e., entries where data = "file").
`getNumberOfFilesLargerThan`	Return the number of files where size >= the specified size.
`getNumberOfFilesNewerThan`	Return the number of files where date >= the specified date.
`getNumberOfMovies`	Return the number of files where the name ends with ".mov", ".avi" and ".mpeg". Ignore the case of the file extensions.
`getNumberOfMoviesNewerThan`	i'm just going to assume you can figure this out.
`getNumberOfPictures`	Pictures are files ending in ".png", ".jpg" or ".jpeg". Case does not matter.

`getTreeStructure`	You've already written a print method in your other trees, you can probably copy and paste it here. Use it to debug your code.
`insertVirus(String fileName, String virusCode, int virusSize)`	For this assignment, pretend that the only impact of a virus is to change the file size to the specified size. The virusCode doesn't do anything but if we pretended we were infecting something with a virus and didn't actually pass in a virus, it would look silly. Just ignore the parameter. Viruses only affect files.
`rename`	Rename the specified (case irrelevant) file name to the new file name (case does matter).
`serialize`	Serialization (sometimes called dehydrate or export) converts the contents of the tree to XML. The format you create in serialization should be usable by your deserialization method.

---

5. Reverse Polish Calculator (5 points)

Tests: Stacks
Rank: Challenging

Given a math formula in reverse Polish notation, calculate the answer. You must solve this problem using a single stack.

ReversePolishCalculator
`+ getValue(List<String> expression) : float`

There's just one method. To switch things up a bit, we'll give you list of strings. Each string is either a float (you'll need to parse it) or an operator (+, -, *, /). The expression is in postfix / reverse Polish form:

```
6 3 2 + 4 * + 5 -
```

With parentheses (which you never need in reverse Polish and we won't give you), this is:

```
((6 (((3 2 +) 4 *) +) 5 -)
```

If you can't read reverse Polish, this is the same as:

```
6 + ((2+3) * 4) - 5
```

or, if you love scheme:

```
(- (+ 6 (* 4 (+ 2 3))) 5)
```

`getValue()` is required to use a `MyStack`. The challenge here is to figure out how to use it. As always, i highly recommend you solve it on paper first – how can a list of binary operators and numbers and a stack i can alternate pushing things onto and popping off of be used to figure out the answer (or order of operations) to a math problem.

The code is pretty trivial but determining the English/real world way of solving this takes some thought. Because that's the goal here, we won't help you with this and you can't talk to your friends about it.

Modifications

- 1.1. Q3Tree Building. Explicitly state implied assumption that keys are not necessarily numeric.
- 1.2 Q4 Added a second constructor to FileSystemNode.