GIT Department of Computer Engineering
CSE 222/505 - Spring 2020
Homework 5
Due date: 28.04.2020 – 12:30

Q1:

Implement **FileSystemTree** class to handle a file system hierarchy in a general tree structure. You need to implement **FileNode** class to handle the nodes of the tree. A node can be created either for a file or a directory. You will decide how discrimination is done between files and directories.

Your **FileSystemTree** implementation must have the following:

A constructor to create a file system with a root directory. Name of the root directory will be given as a parameter to the constructor.

addDir and **addFile** methods to add directories or files to the file system. The path of the new directory (or file) will be given as a parameter to the method.

remove method to remove a directory (or a file) from the file system. The path of the directory (or the file) will be given as a parameter to the method. The method will warn the user if the path cannot be found. If the directory includes some other directories (or files), method will list the contents and ask the user whether to remove or not.

search method to search the entire file system for a directory or a file including the given search characters in its name. The search characters will be given as the parameter of the method.

printFileSystem method to print the whole tree.

Give information about your traversal methods in your javadoc file.

Here are some code lines to help you understand what we expect:

```
//Create a file system with root directory
FileSystemTree myFileSystem = new FileSystemTree("root");


//Add directories and files using paths
myFileSystem.addDir("root/first_directory");
myFileSystem.addDir("root/second_directory");
myFileSystem.addFile("root/first_directory/new_file.txt");
myFileSystem.addDir("root/second_directory/new_directory");
myFileSystem.addFile("root/second_directory/new_directory/new_file
.doc");
```

```
//Search file or directory names including "new"
myFileSystem.search("new");
//This will output:
// file - root/first_directory/new_file.txt
// dir - root/second_directory/new_directory
// file - root/second_directory/new_directory/new_file.doc


//Remove files or directories
myFileSystem.remove("root/first_directory/new_file.txt");
myFileSystem.remove("root/second_directory/new_directory");
```

Q2:

Implement **ExpressionTree** class of arithmetic operations which extends the **BinaryTree** class implementation given in your Data Structures book.

Your **ExpressionTree** implementation must have the following:

A constructor to initialize the tree structure with the given expression string. The expression string will be given as a parameter to the constructor. The expressions will include integer operands and arithmetic operators. Operands and operators will be separated by spaces. The constructor will use the overridden **readBinaryTree** method. You should override it such that it will be able to create an expression tree by reading both prefix and postfix expressions.

The binaryTree implementation of the book includes a preOrderTraverse method. You will add a **postOrderTraverse** method to traverse the tree post order.

The binaryTree implementation of the book includes a toString method which creates a string of the tree structure in preorder. You will add a **toString2** method which will create a string of the tree structure in post order. This method will use the postOrderTaverse method.

You will add **eval** method which evaluates the expression and returns the result as an integer.

Here are some sample code lines:

```
//Create a tree for preorder expression
ExpressionTree expTree = new ExpressionTree("+ + 10 * 5 15 20");
//Create a tree for postorder expression
ExpressionTree expTree2 = new ExpressionTree("10 5 15 * + 20 +");


//Evaluate expressions
result1 = expTree.eval();
```

```
result2 = expTree2.eval();
```

Q3.

Assume you need to record the number of people in each age for a population. Extend the BinarySearchTree class of your book as **AgeSearchTree** class. You will implement **AgeData** class to handle both age and number of people at that age values. You should keep instances of AgeData in your tree. Note that AgeData should be Comparable. CompareTo method of AgeData class will be used and the comparison will be done considering the age only. For the AgeSearchTree class you will override add, remove and find methods as follows:

- While adding a node, the **add** function will first check if a node with that age exists. If it exists, the number of people field of the AgeData object in that node will be increased 1. Otherwise a new node with the AgeData object will be inserted.
- While removing a node, the **remove** function will first check if a node with that age exists. If it exists and the number of people field of this node's AgeData object is greater than 1, it will decrease the number of people field 1. If the number of people field is 1, it will remove the node.
- The **find** method will get an AgeData object of any age and find the AgeData object with the same age and return it.
- Add a **youngerThan** method which returns the number of people younger than an age.
- Add an **olderThan** method which returns the number of people older than an age. Be careful! If your youngerthan and olderThan methods always traverse all nodes that you cannot get whole credit. You should traverse only the nodes needs to be traversed.

Here are some example code lines to help you:

```
//Create an empty age tree
AgeSearchTree<AgeData> ageTree = new AgeSearchTree<AgeData>();

//Add nodes for some ages, remove method works similarly
ageTree.add(new AgeData(10));
ageTree.add(new AgeData(20));
ageTree.add(new AgeData(5));
ageTree.add(new AgeData(15));
ageTree.add(new AgeData(10));

//Create a string representing the tree and print it
treeStr = ageTree.toString();
System.out.println(treeStr);
//This will print:
//10 - 2
//5 - 1
```

```
//null
//null
//20 - 1
//15 - 1
//null
//null
//null

//Print the number of people younger than 15
System.out.println(ageTree.youngerThan(15))
//output will be 3

//Find the number of people at any age
System.out.println(ageTree.find(new AgeData(10)).toString())
//It will print:
//10 - 2
```

Q4.

Solve the problem in Q3 by using a max_heap (where the maximum element is in the root node) this time. Implement your heap by using ArrayList as described in your book. Implement **MaxHeap** class to handle the ArrayList heap operations. Use the **AgeData** class you implemented in Q3 to hold age and the number of people at that age data. **The key of heap will be "number of people" this time. So, the age which the highest number of people is at, will be at the root.** Write a Comparator to compare two objects of class AgeData. You should use the compare method in your comparator to compare the elements in the heap. Implement the following methods (and any other methods you need) in MaxHeap class:

- **add** function to add a new record. It will first check if an AgeData object with that age exists in any index of the ArrayList. If it exists, the number of people field of the AgeData object in that node will be increased 1. (Check if any changes in heap is needed since the key is "number of people".) Otherwise a new heap record with the AgeData object will be inserted.
- While removing a node, the **remove** function will first check if a node with that age exists. If it exists and the number of people field of this node's AgeData object is greater than 1, it will decrease the number of people field 1. (Check if any changes needed since the key is "number of people".) If the number of people field is 1, it will remove the node.
- The **find** method will get an AgeData object of any age and find the AgeData object with the same age and return it.
- Add a **youngerThan** method which returns the number of people younger than an age.
- Add an **olderThan** method which returns the number of people older than an age.

```
//Create an empty heap
MaxHeap<AgeData> heap = new MaxHeap<AgeData>();

//Add nodes
heap.add(new AgeData(10));
heap.add(new AgeData(5));
heap.add(new AgeData(70));
heap.add(new AgeData(10));
heap.add(new AgeData(50));
heap.add(new AgeData(5));
heap.add(new AgeData(15));

//Create a string representing the heap and print it
heapStr= heap.toString();
System.out.println(heapStr);
//output:
//10 - 2
//5 - 2
//70 - 1
//50 - 1
//15 - 1

//Print the number of people younger than 15
System.out.println(heap.youngerThan(10))
//output will be 2

//Find the number of people at any age
System.out.println(heap.find(new AgeData(10)).toString())
//It will print:
//10 - 2
```

IMPORTANT: I will test your classes with my own test main, so use the class names, function names and function parameters as mentioned above.

RESTRICTIONS:

- Use only specified data types
- Can be only one main class in each question
- Don't use any other third part library

GENERAL RULES:

- For any question firstly use course news forum in Moodle, and then the contact TA.
- You can submit assignment one day late and will be evaluated over twenty percent (%40).

TECHNICAL RULES:

- Use given CSE222-VM to develop and test your Homeworks (your code must be working on CSE222-VM), CSE222-VM download link will be given on Moodle.
- Implement clean code standards in your code;
    - Classes, methods and variables names must be meaningful and related with the functionality.
    - Your functions and classes must be simple, general, reusable and focus on one topic.
    - Use standard java code name conventions.

REPORT RULES:

- Add all javadoc documentations for classes, methods, variables …etc. All explanation must be meaningful and understandable.
- You should submit your homework code, Javadoc and report to Moodle in a "studentid_hw3.tar.gz" file.
- Use the given homework format including selected parts from the table below:

Detailed system requirements	
Use case diagrams (extra points)	
Class diagrams	X
Other diagrams	
Problem solutions approach	X
Test cases	X
Running command and results	X

GRADING :

- **No OOP design: -100**
- **No interface: -95**
- **No method overriding: -95**
- **No error handling: -50**
- **No inheritance: -95**
- **No polymorphism: -95**
- No javadoc documentation: -50
- No report: -90
- Disobey restrictions: -100
- **Cheating : -200**
- Your solution is evaluated over 100 as your performance.

CONTACT :

- Teaching Assistant : Nur Banu Albayrak