**CS1027 Computer Science Fundamentals II**

**Assignment 4**

**Due date: April 5**

**Weight: 8%**

**Purpose**

To gain experience with

* The solution of problems through the use of trees and iterators
* The design of recursive algorithms and their implementation in Java.

1. **Introduction**

In this assignment you are to design and implement algorithms to print the structure of the file system in your computer. You will be given code to display on the screen a graphical user interface that shows the information of your files system. Your task is to write algorithms to collect that information. The information about your file system will be stored in a tree, as explained below.

1. **Classes that you Need to Design and Implement**

For this assignment you will have to implement only two java classes. The methods and instance variables for these classes are specified below. You can add more instance variables or methods, if you need to, but you must follow good programming practices; so, for example, all instance variables and auxiliary methods must be declared as *private*.

* 1. **Class *TreeNode***

This class represents the nodes of the tree that will store the information about the file system. The header for the class will be

*public class TreeNode<T>*

In this class you need to import *java.util.Comparator* and *java.util.Iterator*. This class will have three instance variables:

*private TreeNode<T> parent*

*private ListNodes<TreeNode<T>> children*

*private T data*

Like their names suggest, instance variable *parent* is a pointer to the parent of *this* node; *children* is a reference to a list storing the children of *this* node and *data* points to the data object stored in *this* node. This class will have two constructors:

*public TreeNode()* and

*public TreeNode (T d, TreeNode<T> p)*

The first constructor will set *parent* and *data* to *null*, while *children* will be set to an empty *ListNodes<TreeNode<T>>* object (note the nested generic type). The second constructor will also set *children* to an empty *ListNodes<TreeNode<T>>*, while *data* will be set to *d* and *parent* to *p*. You need to implement the following methods in this class:

* *public void setParent(TreeNode<T> p).* Sets the parent of *this* node to *p*.
* *public TreeNode<T> getParent().* Returns the parent of *this* node.
* *public void addChild(TreeNode<T> c).* Adds the given node *c* to the list of children of this node. Node *c* must have its parent set to *this* node.
* *public Iterator<TreeNode<T>> getChildren().* Returns an iterator containing the children nodes of *this* node.
* *public Iterator<TreeNode<T>> getChildren(Comparator<TreeNode<T>> sorter).* Returns an iterator containing the children nodes of *this* node sorted in the order specified by the parameter *sorter*. A Java *Comparator* is an interface providing method *compareTo(T obj1, T obj2)* that compares two objects of type *T.* To sort the list of children of *this* node according to the order specified by *sorter* you can use this code:

*children.sortedList(sorter)*

which will return an object of the class *ListNodes<TreeNode<T>>* whose elements are sorted according to *sorter*. Method *sortedList* is in class *ListNodes* provided to you; see Section 3 for more details on this class.

* *public T getData().* Returns the data stored in *this* node.
* *public void setData(T d).* Stores in *this* node the data object referenced by *d*.
  1. **Class FileTree**

This class represents a tree storing the information of the file objects (files and folders or directories) in your file system. To create this tree, the user will select a folder *F* of the file system. This folder will be represented by the root of the tree. The subfolders and files contained in *F* will be represented by the other nodes of the tree. The files and folders stored directly inside *F* will be the children of the root. If a child *C* of the root represents subfolder *FC*, then the files and folders directly contained in *FC* will be represented as the children of *C*, and so on.

For example, suppose that folder *Assmt4* contains a file *assmt4*.*docx* and two folders: *src* and *bin*. Inside folder *src* there are two files: *TreeNode*.*java* and *FileTree*.*java*. Inside folder *bin* there are three files: *TreeNode*.*class*, *FileTree*.*class* and .*classpath* and one folder: *version1*. Inside *version1* there are two files: *tmp1*.*txt* and *backup*.*bk*. Folder *Assmt4* and the file objects contained in it can be represented with the tree shown in the following figure.

Each node of the tree will store an object of class *FileObject*, described below. Each *FileObject* represents either a file or a folder. In this class you need to import *java.util.Iterator*. For this class you must implement the following methods:

* *public FileTree (String fileObjectName) throws FileObjectException*. This is the constructor for the class. The argument is the name of a file object that will be stored in the root of the tree. If *fileObjectName* is the name of a file, the *FileTree* will have only one node storing the *FileObject* representing the file. Otherwise, the *FileTree* will include nodes for all the file objects contained inside the folder named by *fileObjectName*, as explained above.

*Assmt4*

*bin*

*src*

*assmt4*.*docx*

*TreeNode*.*java* *FileTree*.*java*

*TreeNode*.*class* *FileTree*.*class* *version1* .*classpath*

*tmp1*.*txt* *backup*.*bk*

For this assignment you do not need to write a *main* method. The *main* method for this assignment is in class *Gui.java* described below. This class will get from the user interface the name of the file object to be used to create the *FileTree* tree. Method *main* in class *Gui.java* will invoke your constructor to build the tree.

Here are some hints about how to construct the tree. Please feel free to ignore the hints given for this assignment and design your own algorithms.

First create a new *FileObject (fileObjectName)* representing the file or folder named by *fileObjectName*. This constructor will throw a *FileObjectException* exception if there is a problem while constructing the *FileObject*. You do not need to do anything with the exception. As the signature of the constructor specifies, if a *FileObjectException* is thrown by *FileObject(fileObjectName)* this exception will simply be re-thrown by the constructor.

If *fileObjectName* is the name of a folder, you can use an auxiliary recursive algorithm to explore the folder and to create the corresponding nodes of the tree. This recursive algorithm will take as parameter a *TreeNode<FileObject>* node; let us call it *r*. In the first call this parameter is the root of the tree. The recursive algorithm will have two parts:

* + Base case. This happens when *r* represents a file. The algorithm does not need to do anything in this case, as the node *r* already represents the file and there are no additional file objects inside a file.
  + Recursive case. This happens when *r* represents a folder. Let *f* be the *FileObject* stored in *r*. Use method *f.directoryFiles()* to get an iterator containing all the objects of type *FileObject* representing the file objects contained directly inside the folder represented by *f*. For each *ObjectFile f’*contained in the iterator create a *TreeNode<FileObject>* node *n’* storing *f’*;set *n’* as the child of *r* and *r* as the parent of *n'*. Finally, invoke the algorithm recursively passing as parameter *n’* so the algorithm builds a subtree for the part of the file system corresponding to file object *f’*.
* *public TreeNode<FileObject> getRoot(*). This method just returns the root of the tree.
* *public Iterator<String> filesOfType (String type).* This method returns an iterator storing *String* objects with all the names of the files of the specified type represented by nodes of this *FileTree*; each name should include the absolute path to the file. A type is a file extension like “.java”, “.class”, or “.jpg”. For example, if we invoke this method on the tree shown in the above figure and we pass as parameter the type “.java” then the iterator returned by the method will contain two *String* objects: "C:\Assmt4\src\TreeNode.java" and "C:\Assmt4\src\FileTree.java".

As suggested for the constructor of this class, this method can be implemented with a recursive auxiliary algorithm that receives a parameter a *TreeNode<FileObject>* node *r* (in the initial call that node is the root of the tree), and the file type. The algorithm will also need some container *C* to store the names of the files with the given type. For this you can use, for example, a Java *Array*L*ist*, or any other data structure that can be converted to an iterator.

In the base case, *r* represents a file. Let *f* be the *FileObject* stored in *r*. Use method *f.getLongName()* to get the name of the file represented by *f*. If the name of the file ends with the specified type (Java class *String* has methods that you an use to test this) then add the name to the container *C*.

In the recursive case *r* represents a folder. Use method *r.getChildren()* to get an iterator storing *TreeNode<FoleObject>* objects representing the file objects directly inside the folder represented by *r*. For each one of these file objects create a node *n* storing the file object and invoke the algorithm recursively, but now passing *n* as the first parameter; this recursive call will add to *C* all those files inside *n* that are of the given type.

Once the recursive algorithm ands and container *C* stores all file names of the given type, convert *C* to an iterator and return it.

* *public String findFile(String name).* This method will look for a file with the specified name inside this tree. If the file is found, then a *String* containing the absolute path to the file must be returned; otherwise an empty *String* "" must be returned. For example, if the method is invoked with parameter "tmp1.txt" on the tree shown in the figure of page 3, the algorithm must return the string "C:\Assmt4\bin\version1\tmp1.txt" (assuming that folder "Assmt4" is inside drive "C:"). If there are several files with the same given name inside the tree, the name of the first one found is the one returned by the algorithm. The implementation of this method is vey similar to that for method *filesOfType*.

If *f* is a variable referencing a *FileObject* representing a file, method *f.getLongName()* returns a *String* containing the full name of the file that includes the absolute path to the file. For example for the file object *f* represented by the node in the figure of page 3 storing "tmp1.txt", the method *f.getLongName()* will return "C:\Assmt4\bin\version1\tmp1.txt". There is another method in class *FileObject* called *getName()* which returns only the name of the file, without the absolute path to it; so for the same file object *f* as above, *f.getName()* will return the *String* "tmp1.txt".

**Important Note**. If you are not implementing the extra credit, you must include the following code in your *FileTree* java class, as otherwise it will not compile:

*public Iterator<String> duplicatedFiles() { return null;}*

*public Iterator<String> duplicatedFolders() {return null;}*

1. **Classes Provided**

You can download from OWL the following classes.

* **Gui.java, ControlPanel.java, MyTextArea.java, SplitPanel.java**. These classes contain code to display the user interface for this assignment. The user interface is divided in two parts. On the left part the user can select a file object, and then choose either to print the directory structure represented by the file object, find files of a given type, find a particular file, or for the bonus part, find duplicated files or folders.

Class *Gui.java* contains the *main* method. To use the program, you first need to select a file or folder in the file chooser and click on "Select"; then you can choose one of the functions provided.

* **ListNodes.java.** This class stores a list of object of type *T*. You will use this class to store the list of children of a *TreeNode<FileObject>* node. This class provides the following public methods:
  + *public ListNodes().* The constructor creates an empty list.
  + *public void add(T element).* Adds the given element to the list.
  + *public Iterator<T> getList().* Returns an iterator containing all the elements in the list.
  + *public Iterator<T> sortedList(Comparator<T> sort).* Returns an iterator containing all the elements in the list sorted according to the specified *Comparator*.
* **FileObject.java.** This class represents a file or a folder. This class provides the following methods:
  + *public FileObject(String name) throws FileObjectException.* The constructor for the class. The parameter specifies the name of a file or a folder. A *FileObjectException* will be thrown if the object cannot be created.
  + *public boolean isFile().* Returns true if this *FileObject* represents a file.
  + *public boolean isDirectory().* Returns true if this *FileObject* represents a folder or directory.
  + *public String getName().* Returns the name of the file; this name does not include the absolute path to the file.
  + *public String getLongName().* Returns the name of the file; this name includes the absolute path to the file.
  + *public int numFilesInDirectory*. If this object represents a folder, then the method returns the number of file object stored directly inside this folder. Note that if the folder represented by *this* *FileObject* contains subfolders, the number of file objects in the subfolders is not included in the value returned by the method.
  + *public Iterator<FileObject> directoryFiles().* Returns an iterator of *FileObject* objects; each *FileObject* represents a file or folder directly inside the folder represented by this *FileObject*.
* **FileObjectException.java.** The exception class thrown when a *FileObject* cannot be created.
* **NameComparator.java** and **SizeComparator.java**. Java classes used to sort the list of children of a node by name or by size.
* **PrintFile.java.** Java class used to print the file structure represented by a *FileTree*.

1. **Extra Credit (10% Bonus)**

For extra credit you are required to implement two additional methods (5% bonus for each). For these methods you cannot first store the nodes or *FileObject*'s of the tree in a container (array, list, inked list, stack, queue, or anything else) and then look for the duplicated files or folders in the container.

* *public Iterator<String> duplicatedFiles().* This method returns an iterator storing the long names (including absolute paths to the files) of all duplicated files in a *FileTree*.

For this assignment two files are duplicated if they have the same name. So, for example, your method might return an iterator with the following Strings:

"C:\Assmt4\bin\version1\tmp1.txt", "C:\Assmt2\tmp\tmp1.txt", "C:\tmp\tmp1.txt", "C:\Users\control.exe", "C:\Programs\tools\control.exe".

The iterator should be sorted so that files with the same name (although not the same absolute path) appear in adjacent positions. So the iterator for the above example cannot store the file names in this order: "C:\Assmt4\bin\version1\tmp1.txt", "C:\Programs\tools\control.exe", "C:\Assmt2\tmp\tmp1.txt", "C:\Users\control.exe", "C:\tmp\tmp1.txt".

* *public Iterator<String> duplicatedFolders().* This method returns an iterator storing the long names (including absolute paths to the files) of all duplicated folders in a *FileTree*. Note that for two folders to be duplicated they not only must have the same name, but they must contain the same files and folders. So for example, in a *FileTree* there might be two folders called "tmp" each one containing two files called "file1" and "file2". These folders will be considered as duplicated. However a third folder called "tmp" and storing only file "file2" would not be considered a duplicate of the other two as it does not contain "file1". The iterator returned by your method could, for example, then contain these two Strings: "C:\tmp" and "C:\Progams\tmp".

The iterator should be sorted so that folders with the same name (although not the same absolute path) appear in adjacent positions.

**5. Non-functional Specifications**

1. Assignments are to be done individually and must be your own work. Software will be used to detect cheating.
2. Include comments in your code in javadoc format. Add javadoc comments at the beginning of your classes indicating who the author of the code is and a giving a brief description of the class. Add javadoc comments to methods and instance variables. Read information about javadoc in the second lab for this course.
3. Add comments to explain the meaning of potentially confusing parts of your code.
4. Use Java coding conventions and good programming techniques. Read the notes about comments, coding conventions and good programming techniques in the first assignment.

Submit all your .java files to OWL. **DO NOT** put the code inline in the textbox. **Do not submit your *.class* files.** If you do this, and do not attach your *.java* files, you will receive a mark of zero!

**6. What You Will Be Marked On**

1. Functional specifications:

* Does the program behave according to specifications?
* Does it run with the test input files provided?
* Are your classes created properly?
* Are you using appropriate data structures?
* Is the output according to specifications?

1. Non-functional specifications: as described above
2. Assignment submission: via OWL assignment submission.