Assignment 4

CS 1027
Computer Science Fundamentals II

Due Date: Thursday April 6 at 11:55 pm.

Learning Outcomes

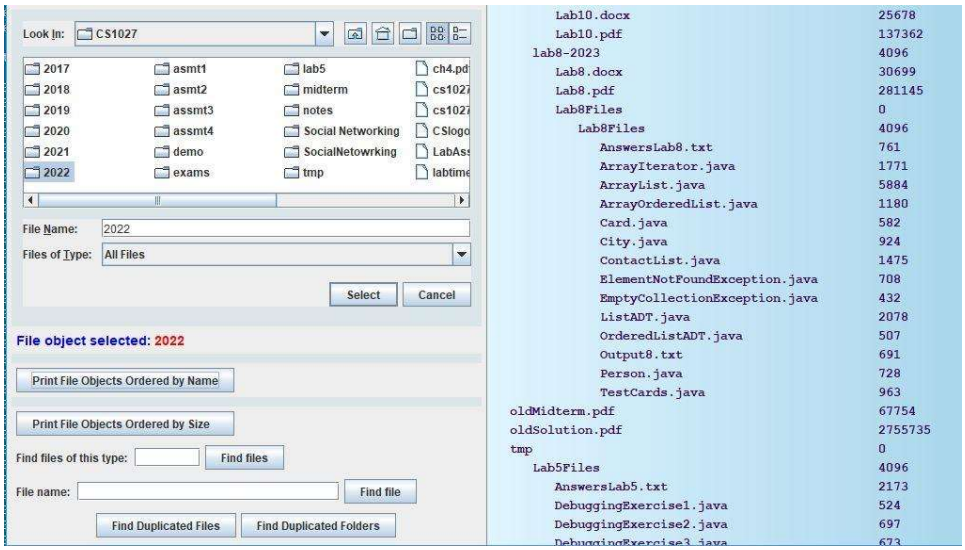
To gain experience with

- The solution of problems using non-linear data structures
- 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 information of your files system. Your task is to write algorithms to collect that information and store it in a non-linear linked data structure. A non-linear data structure is formed by nodes containing references to other nodes but, different from lists, a node could have references to more than two other nodes in the structure. We will study this kind of non-linear liked data structures in class when we learn about trees.

The following figure shows how what the graphical user interface looks like. The user interface looks a bit different on a Mac than on a Windows computer.



As mentioned above, the linked data structure that we will use is such that a node *n* can have references to many other nodes, such nodes are called the children of n. However, at most one other node p can have a reference to node n; this node p is called the parent of n. You will learn more about this terminology when we talk about trees in class.

In this assignment you will also use the ADT Iterator, which will be discussed also in class. An iterator stores a collection of data items and provides 3 operations:

- hasNext(): returns true if the iterator has more data items.
- next(): returns the next data item
- remove(): deletes the last data item returned by the next operator.

An iterator allows us to process a collection of data items one at a time. In Java you can create an iterator by storing a collection of data items in one of the Java collections of the standard library, like ArrayList, LinkedList, HashMap, and invoking the method iterator(). For example, the following code stores a set of data items in and ArrayList and then converts it into an iterator:

```
ArrayList<String> list = new ArrayList<String>();
// Add strings to list
Iterator<String> myIterator = list.iterator();
```

The data items stored in the above iterator can then be processed one by one using, for example, this code:

```
while (list.hasNext()) {
        String nextItem = list.next();
        …
}
```

Classes to Implement

1. Class NLNode

This class represents the nodes of the non-linear data structure that will store the information about the file system. This class will be declared as follows:

```
public class NLNode<T>
```

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

- private NLNode<T> parent. A reference to the parent of this node, i.e. the unique node that has a reference to this node.
- private ListNodes<NLNode<T>> children. A reference to a list storing the children of this node (note the nested generic type).
- private T data. A reference to the data object stored in this node.

This class will have two constructors:

- public NLNode(). Sets instance variables parent and data to null, while children is initialized to an empty ListNodes<NLNode<T>> object.

- public NLNode (T d, NLNode<T> p). Initializes instance variable children to an empty ListNodes<NLNode<T>>, while data is set to d and parent to p.

You need to implement the following methods in this class:

- public void setParent(NLNode<T> p). Sets the parent of this node to p.
- public NLNode<T> getParent(). Returns the parent of this node.
- public void addChild(NLNode<T> newChild). Adds the given node newChild to the list of children of this node. Node newChild must have its parent set to this node.
- public Iterator<NLNode<T>> getChildren(). Returns an iterator containing the children of this node. Read the provided class ListNodes to learn how to convert the list of children to an Iterator.
- public Iterator<NLNode<T>> getChildren(Comparator<NLNode<T>> sorter). Returns an iterator containing the children of this node sorted in the order specified by the parameter sorter. As explained in class the Java Comparator is an interface providing method compareTo(T obj1, T obj2) that compares two objects of generic 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<NLNode<T>> whose elements are sorted according to sorter. Method sortedList is in class ListNodes provided to you; read below 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.

2. Class FileStructure

This class represents the linked structure that will store the information of the file objects in your file system. A file object is either a file or a folder (or directory). To create this linked structure, the user will first select a folder *F* of the file system. This folder will be represented by the top node of the linked structure; this top node is called the *root* node. The subfolders and files stored directly in *F* will be represented by nodes in the list of children of the root. If a child *C* of the root represents subfolder F_C, then the files and folders directly contained in F_C will be stored as nodes in the list of children of *C*, and so on.

For example, suppose that folder C:*Assmt4* contains a file *assmt4.docx* and two folders: *src* and *bin*. Inside folder *src* there are two files: *NLNode.java* and *FileStructure.java*. Inside folder *bin* there are three files: *NLNode.class*, *FileStructure.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 are represented with the linked structure shown in the following figure.

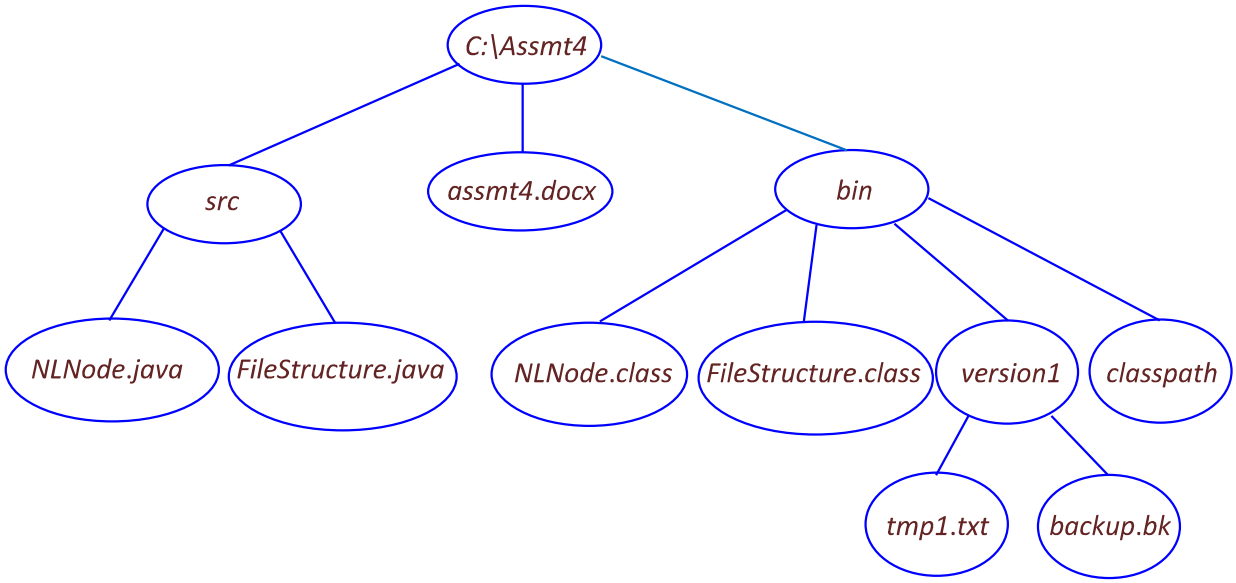
Each node of this structure stores an object of class FileObject, described below. Each FileObject represents either a file or a folder.

This class will have one instance variable:

- private NLNode<FileObject> root. A reference to root node.

In this class you need to import java.util.Iterator and you must implement the following methods:

- public FileStructure (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 node. If fileObjectName is the name of a file, this FileStructure object will have only one node storing the FileObject representing the file. Otherwise, this FileStructure object will have a list with nodes for all the file objects contained inside the folder named fileObjectName, as explained above.
  For this assignment you do not need to write a main method. The main method for this assignment is in class FileSystem.java described below. This class will get from the user interface the name of the file object to be used as the root node of the FileStructure. Method main in class FileSystem.java will invoke your constructor to build the linked structure representation for your file system.



Here are some hints about how to construct the FileStructure. You do not have to follow these hints; you can design your own algorithms. However, the algorithm that you use for creating the nodes of the file structure **must be recursive**.

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 this exception if thrown because as the signature of the constructor specifies, if a FileObjectException is thrown by FileObject(fileObjectName) this exception will be re-thrown by the constructor.

If fileObjectName is the name of a folder (read below class FileObject to learn how to determine whether fileObjectName is the name of a folder), you can use an auxiliary recursive algorithm to explore the folder and to create the nodes corresponding to the file objects contained in it. This recursive algorithm will take as parameter a NLNode<FileObject> node, let us call it *r*. In the first call to this auxiliary recursive algorithm the parameter is *r*, the root node. 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() from class FileObject 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 FileObject *f'* contained in the iterator create a NLNode<FileObject> node *n'* storing *f'*; then set *n'* as the child of *r* and *r* as the parent of *n'*; and finally, invoke the algorithm recursively passing as parameter *n'* so the algorithm builds the structure corresponding to the part of the file system represented by *f'*.

- public NLNode<FileObject> getRoot(). Returns the root node.
- public Iterator<String> filesOfType (String type). This method returns a String iterator with the names of all the files of the specified type represented by nodes of this FileStructure; each name should include the absolute path to the file. A file type is an extension like ".java", ".class", or ".jpg". For example, if we invoke this method on the structure shown in the above figure and we pass as parameter the type ".java" then the iterator returned by the method will contain two Strings: "C:\Assmt4\src\NLNode.java" and "C:\Assmt4\src\FileStructure.java". (Absolute paths on a Mac are slightly different.)

  This method **must** be implemented with a **recursive** auxiliary algorithm that receives a parameter a NLNode<FileObject> node *r* (in the initial call that node is the root node), and the file type. The algorithm will also need some container to store the names of the files with the given type. To implement this container you can use, for example, a Java ArrayList, 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() from class FileObject to get the absolute path to the file represented by *f*. If the name of the file ends with the specified type (Java class String has methods that you can use to test this) then add the name to the aforementioned container.

  In the recursive case *r* represents a folder. Use method r.getChildren() to get an iterator storing NLNode<FileObject> objects representing the file objects directly inside the folder represented by *r*. For each one of these file objects *n* invoke the algorithm recursively, but now passing *n* as the first parameter; this recursive call will add to the

container all those files inside *n* that are of the given type. Once the recursive algorithm ends, convert the container to an iterator and return it.

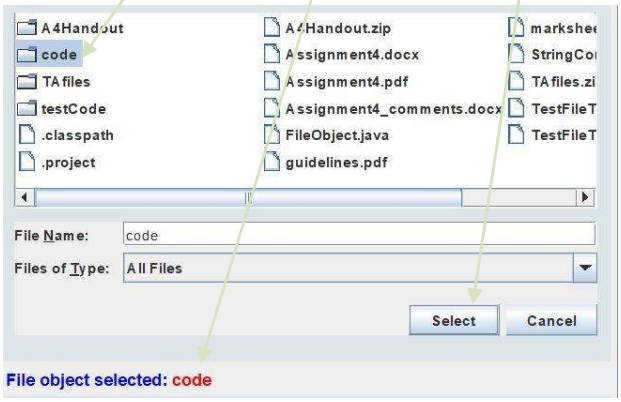
- public String findFile(String name). This method will look for a file with the specified name inside this FileStructure. If the file is found, then a String containing the absolute path to the file must be returned; otherwise, an empty String "" must be returned. For example, if the method is invoked with parameter "tmp1.txt" on the file system shown in the figure above, the algorithm must return the string "C:\Assmt4\bin\version1\tmp1.txt". If there are several files with the same given name inside the file system, 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, as mentioned above 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 above 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".

Classes Provided

- FileSystem.java, ControlPanel.java, MyTextArea.java, SplitPanel.java. These classes contain code to display the graphical 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, or find a particular file.

  As mentioned above class FileSystem.java contains the main method. After running the program, you first need to choose a file or folder in the file chooser (the window showing icons for the files and folders) and click on "Select". You will see that the message “File object selected:” will show in red the file object that you have chosen. If not file object has been selected the program will not work. After selecting a file object you can choose one of the functions provided by the graphical user interface.



- ListNodes.java**.** This class stores a list of object of generic type *T*. You will use this class to store the list of children of a NLNode<FileObject> node. This class provides the following public methods:
  - public ListNodes(). The constructor creates an empty list.
  - public void add(T dataItem). Adds the given dataItem to the list.
  - public Iterator<T> getList(). Returns an iterator containing all the data items in this list.
  - public Iterator<T> sortedList(Comparator<T> sort). Returns an iterator with all the data items in this 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. If the object cannot be created a FileObjectException will be thrown.
  - 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 object represented by this FileObject; this name does not include the absolute path to the file.
  - public String getLongName(). Returns the name of the file object represented by this FileObject; 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.
- PrintFileStructure.java**.** Java class used to print the file structure represented by a FileStructure object.

Assignment 4

CS 1027
Computer Science Fundamentals II

Marking Notes

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented as specified using recursive algorithms as stated?
- Does the code run properly on Gradescope (even if it runs on Eclipse, it is **up to you** to ensure it works on Gradescope to get the test marks)
- Does the program produce compilation or run-time errors on Gradescope?
- Does the program fail to follow the specifications stated above?

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)? Add comments to explain the meaning of potentially confusing parts of your code.
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a penalty.
- Including a "package" line at the top of a file will receive a penalty.

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. Sharing, copying code, or submitting code not designed completely by you is an academic offense. All submitted code will be run through similarity-detection software.

Submission (due Thursday, April 6 at 11:55 pm)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below.
- Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope. If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.

- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works for a variety of cases. We are providing you with some tests but we may use additional tests that you haven't seen for marking.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code as many times as you wish, however, re-submissions after the assignment deadline will receive a late penalty.

Files to submit

- NLNode.java
- FileStructure.java

Grading Criteria

Total Marks: 20

14 marks	Passing Tests
6 marks	Functional and non-functional Specifications (correct data structures, correct algorithms, code readability, comments, correct variables and functions, etc.)