CSCI-UA 102 (Data Structures)

Project 4: Disk Usage Analyzer

Project 4 due date: November 9

submission mode: individual

YOU MAY DISCUSS ANY OF THE ASSIGNMENTS WITH YOUR CLASSMATES AND TUTORS (OR ANYONE ELSE) BUT ALL WORK FOR ALL ASSIGNMENTS MUST BE ENTIRELY YOUR OWN. ANY SHARING OR COPYING OF ASSIGNMENTS WILL BE CONSIDERED CHEATING (THIS INCLUDES POSTING OF PARTIAL OR COMPLETE SOLUTIONS ON ED, GITHUB, DISCORD, GROUPME, ... OR ANY OTHER FORUM). IF YOU GET SIGNIFICANT HELP FROM ANYONE, YOU SHOULD ACKNOWLEDGE IT IN YOUR SUBMISSION (AND YOUR GRADE WILL BE PROPORTIONAL TO THE PART THAT YOU COMPLETED ON YOUR OWN). YOU ARE RESPONSIBLE FOR EVERY LINE IN YOUR PROGRAM: YOU NEED TO KNOW WHAT IT DOES AND WHY. YOU SHOULD NOT USE ANY DATA STRUCTURES AND FEATURES OF JAVA THAT HAVE NOT BEEN COVERED IN CLASS (OR THE PREREQUISITE CLASS). IF YOU HAVE DOUBTS WHETHER OR NOT YOU ARE ALLOWED TO USE CERTAIN STRUCTURES, JUST ASK YOUR INSTRUCTOR.

Introduction and objectives

You are going to write a program that uses your new expertise in recursion to explore a directory tree on a user's computer. Your program will provide a tool that given a name of a directory, explores all its sub-directories and files and does two things:

- computes the total size of all the files and sub-directories in the given directory,
- prints a list of `n` largest files (their sizes and paths).

The goal of this programming project is for you to master (or at least get practice on) the following tasks:

- developing and writing recursive algorithms,
- working with existing code,
- using classes and methods that are part of the Java API,
- using command line arguments,
- implementing classes according to provided specification.

Start early! This project may not seem like much coding, but debugging and testing always takes time, especially for recursive algorithms.

▼ BACKGROUND

Files and Directories

Files and directories form a tree-like structure on your computer. Each file/directory has a unique path name that *points to* where it is within that file system tree. For example:

- On Windows systems `C:\Users\asia\top\helloA` is a path name that tells us that a file or a directory named `helloA` is located in a directory named `top`, which is located in the direcotry named `asia` (a home directory for the user `asia`), in a directory named `Users` on drive `C`.
- On Mac systems `/Users/asia/top/helloA` is a path names that tells us that a file or a directory named `helloA` is located in a directory named `top`, which is located in the direcotry named `asia` (a home directory for the user `asia`), in a directory named `Users` in the root directory which is denoted by `/` (forward slash).
- On Linux/Unix systems `/home/asia/top/helloA` is a path names that tells us that a file or a directory named `helloA` is located in a directory named `top`, which is located in the direcotry named `asia` (a home directory for the user `asia`), in a directory named `Users` in the root directory which is denoted by `/` (forward slash).

One can also use relative paths that tell us how to *get to the file/directory* from the perspective of another directory. But we will not get into these.

Irrespective of how we navigate to a particular directory, we can look at it and ask the questions about its content. This is what this project is about.

Consider the following directory structure within a directory called `top` (this directory could be in the user's home directory, or somewhere else - this does not really matter):

```
top
├── A
│   └── another
├── B
└── C
    └── CC
        ├── alpha
        └── CCC
```

The `top` directory has three subdirectoris: `A`, `B` and `C`. Within `A` there is a subdirectory called `another`. `B` does not have any subdirectories. `C` contains one subdirectory called `CC`, which, in turn, has two subdirectories: `alpha` and `CCC`.

If we add file names to the structure, it may look something like this (some file names have been omitted for brievity):

```
top
├── A
│   ├── another
│   │   ├── csci200000
│   │   ├── csci307860
│   │   ├── csci4000
│   │   ├── csci70
│   │   └── csci789000
│   ├── foo1
│   ├── foo10
│   ├── foo100
│   ├── foo1000
```

Project 4 due date: November 9

submission mode: individual

```
│   ├── foo10000
│   └── foo100000
├── B
│   ├── tiny1
│   ├── tiny10
│   ├── tiny100
│   ├── tiny11
    ...
│   ├── tiny9
│   ├── tiny90
│   ├── tiny91
│   ├── tiny92
│   ├── tiny93
│   ├── tiny94
│   ├── tiny95
│   ├── tiny96
│   ├── tiny97
│   ├── tiny98
│   └── tiny99
├── C
│   ├── CC
│   │   ├── alpha
│   │   ├── CCC
│   │   │   ├── alpha1
│   │   │   ├── alpha100
│   │   │   ├── alpha1000
│   │   │   ├── alpha2000
│   │   │   ├── nyu1
│   │   │   ├── nyu100
│   │   │   ├── nyu1000
│   │   │   └── nyu2000
│   │   ├── nyu1
│   │   ├── nyu100
│   │   ├── nyu1000
│   │   └── nyu2000
│   ├── nyu1
│   ├── nyu100
│   ├── nyu1000
│   └── nyu2000
├── helloA
├── helloB
└── helloC
```

Notice that it may be hard to tell a difference between a name of a file and a name of a directory (especially an empty directory in the schematic above). In many cases that difference does not matter. The `File` class that we'll be using for this project can distinguish between files and directories.

Project 4 due date: November 9

submission mode: individual

The `File` class

The `File` class in Java provides a way to represent and interact with files/directories within our programs.

Before you continue with the rest of this project, you should carefully read through the documentation page for that class and learn as much

about it as possible. There are many methods in that class that will come in very handy for this project. (You will not need to work with any classes and methods of the `java.nio.file` package which the `File` class refers to in a few places.)

Project 4 due date: November 9

submission mode: individual

PROGRAM USAGE, RESULTS AND OUTPUT

The program should expect two command line arguments.

First one is a directory name. If the directory is omitted from the command line, it is an error. The program should display an error message and terminate. The error message should indicate what went wrong. If the directory name is provided, but it is not a name of a valid/existing directory, the program should display an error message and terminate. The error message should indicate what went wrong.

The second command line argument is the maximum number of largest files found in the directory provided as the first argument that the program displays. (It is the largest number, because if the directory does not have that many files, the list will need to be shorter.) If the number is missing, the default value of 20 should be used.

Your program should determine and print to standard output the total space used by the directory in question (that includes the sizes or all the files and subdirectories contained in it). The program should also print the sizes and pathnames of the largest files.

The Appendix contains a few sample runs of the program.

The class `DiskUsage` is provided as part of this project. It implements all the things mentioned in this section. You should use it as is and not make any changes (unless they are announced in class.)

DATA STORAGE AND ORGANIZATION

You need to provide an implementation of two classes that store the data and compute the results when the program is executed. You may implement additional classes and additional methods in the required classes, if you wish.

You cannot change the signatures of methods that are required.

The `DiskUsage` program is given in the Appendix. It provides usage verification and calls appropriate methods of your class to calculate the results.

`FileOnDisk` class

`FileOnDisk` class represents a file or directory in the program. Depending on your implementation choices your methods may need to declare exceptions that are not mentioned below. If your function makes a call to a method from the `File` class and such a method throws a checked exception, then your code should either declare it or handle it (that choice is part of your design decision).

This class should inherit from the `File` class. This means that your class can accomplished everything that the `File` class can. Your class adds a few extra features to the `File` class. They are described below.

Project 4 due date: November 9

submission mode: individual

The **data fields** in this class are really up to you, but here are some ideas that may be helpful:

- The `File` class can handle any type of pathname to a file. But from the point of view of this project, having a quick access to the "canonical path" will be useful.
- The `File` class has the `length()` method which returns the number of bytes in a file. It may be handy to store that value in this class for faster access.
- For most directories, the program will need to *know* the total size of all the files and subdirectories stored under it. Once this is computed, it would be a good idea to keep that value around for future use.
- For most directories, the program will need to *know* the list of all the files that are stored in it (to be able to pick the largest ones). Once that list is computed, it would be a good idea to keep that information for future use.

The **constructor** should be one-parameter constructor. Its argument is the file path.

This constructor should throw an instance of a `NullPointerException` if it is called with a `null` pathname argument.

HINT: since the `File` class does not provide a default constructor, your constructor will need to make an explicit call to the superclass' constructor.

`long getTotalSize()` method should compute and return

- (for a directory) the total size of all the files and subdirectories stored in it, or
- (for a file) the size of the actual file.

This method will need to trigger a recursive algorithm (most likely implemented in another private method) that calculates the total size of all the files and subdirectories. For efficiency reason, you should make sure that once the total size is calculated, it is not recalculated again.

`List <FileOnDisk> getLargestFiles(int numOfFiles)` method should compute and return

- (for a directory) the list of `numOfFiles` largest files stored in the directory structure of the object on which it is called; this list should be sorted from largest to smallest file, or
- (for a file) `null`.

The class should override the **`String toString()`** method. It should return a string with the following format:

```
SSSSSSSS XB    PATH
```

- `SSSSSSSS` is a sequence of 8 *spaces* reserved for the file size. This value should be printed with exactly two digits after the decimal point. The value should be right-justified within the field of 8 *spaces*. You can accomplish this by using `%8.2f` format specifier in the `String.format()` method.

- `XB` should be replaced by either `bytes`, `KB`, `MB`, or `GB` depending on the size of the file. (All file sizes should be converted to one of these units by dividing the number of bytes by appropriate powers of 1024 so that the number reported is always smaller than 1024. For example, if the number of bytes in a file is 16384, then this is equivalent to 16496/1024 = 16.11 KB and this value should be reported. If the number of bytes is 4198592, then this is equivalent to 4100.19 KB, or 4.00 MB. Depending on the length of this value, there should be either two or five spaces printed after it and before the `PATH`.
- `PATH` is the actual path name for the file. You should use the canonical path. (Note that some path names may be very long. This may force the output of your pgogram to wrap in some cases. This if fine.)

Project 4 due date: November 9

submission mode: individual

This class should implement a recursive algorithm that computes the results that are returned by the `getTotalSize()` and `getLargestFiles()` methods. Here is the pseudocode of the recursive algorithm you should use:

```
exploreDir ( potentialDirName )

    if potentialDirName is a directory that was not explo

        add its size to totalSize

        get the list of all the files and sub-directories

        for each of the files and sub-directories

            call exploreDir <-- this is the recursive cal

    otherwise potentialDirName is a file

        add file's size to totalSize
```

(You will need to add appropriate error checking and steps that allow you to capture the list of files in a list.)

WARNING: If implemented incorrectly, this algorithm results in infinite recursion when used on systems that allow shortcuts/links in the directory structure (because they may produce circular paths). To avoid this, use the `getCanonicalPath()` method of the File class rather than getAbsolutePath() to obtain the name of the directory and make sure that you never visit the same directory twice.

HINT: When implementing the above algorithm, you will need to keep track of all the files in a list of some kind. The choice of the structure for that list is up to you, but it should be be one of the ones we covered in this class (not a hash table, set, or a binary tree).

The `File` class implements `Comparable<File>` interface and your FileOnDisk inherits that implementation. Rather than overriding it, you should implement an alternative way of comparing FileOnDisk objects that uses Comparator` interface. See below for details.

`FileOnDiskComparatorBySize` class

The only purpose of this class is to implement comparison between `FileOnDisk` objects that is different than the comparison provided by the `File` class.

You should review the documentation for `Comparator<T>` interface.

This class should implement `Comparator<FileOnDisk>` interface. It should have one method:

```
int compare ( FileOnDisk o1, FileOnDisk o2)
```

that compares the two `FileOnDisk` objects by their size (number of bytes), and, if the sizes are equal by their path names (using lexicographic ordering).

Project 4 due date: November 9

submission mode: individual

`DiskUsage` class

This class is implemented for yuou. You should not modify it in any way.

PROGRAMMING RULES

- You should follow the rules outlined in the document Code conventions
- You have to use a recursive algorithm to compute the total size of the directory. You will not get any credit for an iterative algorithm.
- You may use any exception-related classes.

WORKING ON THIS ASSIGNMENT

GRADING

HOW AND WHAT TO SUBMIT

APPENDIX

Sample Program Runs

Here is a sample run of a program with the directory structure shown earlier in this specification.

```
$ java project4.DiskUsage /home/asia/top
    4.54 GB     /home/asia/top
Largest 20 files:
    1.86 GB     /home/asia/top/helloA
    1.50 GB     /home/asia/top/A/another/csci789000
  601.29 MB     /home/asia/top/A/another/csci307860
  390.63 MB     /home/asia/top/A/another/csci200000
   97.66 MB     /home/asia/top/A/foo100000
   18.89 MB     /home/asia/top/C/CC/CCC/alpha2000
   18.89 MB     /home/asia/top/C/CC/CCC/nyu2000
   15.83 MB     /home/asia/top/C/nyu2000
    9.77 MB     /home/asia/top/A/foo10000
    9.44 MB     /home/asia/top/C/CC/CCC/alpha1000
    9.44 MB     /home/asia/top/C/CC/CCC/nyu1000
    7.92 MB     /home/asia/top/C/nyu1000
    7.81 MB     /home/asia/top/A/another/csci4000
    6.68 MB     /home/asia/top/helloC
 1000.02 KB     /home/asia/top/A/foo1000
  967.01 KB     /home/asia/top/C/CC/CCC/alpha100
  967.01 KB     /home/asia/top/C/CC/CCC/nyu100
  810.76 KB     /home/asia/top/C/nyu100
  589.86 KB     /home/asia/top/C/CC/nyu2000
  294.94 KB     /home/asia/top/C/CC/nyu1000

$ java project4.DiskUsage /home/asia/top 5
    4.54 GB     /home/asia/top
Largest 5 files:
    1.86 GB     /home/asia/top/helloA
    1.50 GB     /home/asia/top/A/another/csci789000
  601.29 MB     /home/asia/top/A/another/csci307860
  390.63 MB     /home/asia/top/A/another/csci200000
   97.66 MB     /home/asia/top/A/foo100000

$ java project4.DiskUsage /home/asia/top/A
    2.59 GB     /home/asia/top/A
Largest 20 files:
    1.50 GB     /home/asia/top/A/another/csci789000
  601.29 MB     /home/asia/top/A/another/csci307860
  390.63 MB     /home/asia/top/A/another/csci200000
   97.66 MB     /home/asia/top/A/foo100000
    9.77 MB     /home/asia/top/A/foo10000
    7.81 MB     /home/asia/top/A/another/csci4000
 1000.02 KB     /home/asia/top/A/foo1000
  140.02 KB     /home/asia/top/A/another/csci70
  100.02 KB     /home/asia/top/A/foo100
   10.02 KB     /home/asia/top/A/foo10
    1.02 KB     /home/asia/top/A/foo1

$ java project4.DiskUsage /home/asia/top/B
  633.20 KB     /home/asia/top/B
Largest 20 files:
   12.52 KB     /home/asia/top/B/tiny100
   12.39 KB     /home/asia/top/B/tiny99
   12.27 KB     /home/asia/top/B/tiny98
   12.14 KB     /home/asia/top/B/tiny97
   12.02 KB     /home/asia/top/B/tiny96
   11.89 KB     /home/asia/top/B/tiny95
   11.77 KB     /home/asia/top/B/tiny94
```

Project 4 due date: November 9

submission mode: individual

```
   11.64 KB     /home/asia/top/B/tiny93
   11.52 KB     /home/asia/top/B/tiny92
   11.39 KB     /home/asia/top/B/tiny91
   11.27 KB     /home/asia/top/B/tiny90
   11.14 KB     /home/asia/top/B/tiny89
   11.02 KB     /home/asia/top/B/tiny88
   10.89 KB     /home/asia/top/B/tiny87
   10.77 KB     /home/asia/top/B/tiny86
   10.64 KB     /home/asia/top/B/tiny85
   10.52 KB     /home/asia/top/B/tiny84
   10.39 KB     /home/asia/top/B/tiny83
   10.27 KB     /home/asia/top/B/tiny82
   10.14 KB     /home/asia/top/B/tiny81

$ java project4.DiskUsage /home/asia/top/B 5
  633.20 KB     /home/asia/top/B
Largest 5 files:
   12.52 KB     /home/asia/top/B/tiny100
   12.39 KB     /home/asia/top/B/tiny99
   12.27 KB     /home/asia/top/B/tiny98
   12.14 KB     /home/asia/top/B/tiny97
   12.02 KB     /home/asia/top/B/tiny96
```

Project 4 due date: November 9

submission mode: individual

`DiskUsage` class

```
package project4;

import java.io.*;
import java.util.*;
/**
 * This class provides a simple program that provides inf
 * directory sizes (or rather combined size of all the fi
 * in a directory) along with a list of largest files.
 *
 * @author Joanna Klukowska
 * @version 10-31-2023
 */

public class DiskUsage {

        /**
     * This program expects two command line arguments.
     * @param args <code>args[0]</code> is the name of th
     *             to explore,
     *             <code>args[1]</code> is an optional ar
     *             can be used to indicate how many files
     *             displayed in the list of largest files
     *             value is 20)
     */
        public static void main(String[] args) throws IOE

        //make sure that there is at least one command li
                if (args.length == 0) {
                        System.err.println("Missing name
                        System.exit(0);
                }
```

```
                // use the directory from args[0]
                String directory = args[0];
                FileOnDisk dir = new FileOnDisk(directory
                if ( !dir.exists() ) {
                        System.out.printf("ERROR: %s does
                        System.exit(1);
                }

        int numOfFiles = 20;
        // if args[1] contains a valid positive number, u
        // as the number of files to display
        if (args.length == 2) {
            try {
                numOfFiles = Integer.parseInt(args[1]);
                numOfFiles = numOfFiles > 0 ? numOfFiles
            }
            catch (NumberFormatException ex ) {
                //ignoring the second argument, using 20
                //the number of files to display
            }
        }

        // show the total size of the directory and its p
                System.out.println( dir.toString() );


                // show the list of largest files (from l

                System.out.printf("Largest %d files: \n",

        List<FileOnDisk> list = dir.getLargestFiles(numOf

                for (FileOnDisk f : list ) {
                        System.out.println( f );
        }

        }

}
```

Project 4 due date: November 9

submission mode: individual