Joanna Klukowska
joannakl@cs.nyu.edu

Project 4: NYC Street Trees - Revisited

Due date: April 22, 11:55PM EST.

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 Piazza, GitHub or any other public 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.

In this project you get a chance to revisit your code for project 1. Once again, you will provide a tool for visualizing popularity of New York City street trees in different boroughs of the city. Your program will use the 2015 Street Tree Census data provided by Department of Parks and Recreation: `https://goo.gl/pIjCPG`. Using this data and a name of the tree specified by the user, your program will need to generate information about popularity of this type of tree in each borough of New York City.

Many parts of this project specification are exactly the same as the specification for project 1. The modified and new parts are printed in a different color. You should concentrate on the new/modified parts, but you should read through the entire specification to make sure that your program adheres to the entire specification.

Objectives

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

- working with multi-file programs
- modifying/rewriting code written previously
- reading data from input files
- using and understanding command line arguments
- working with large data sets
- writing classes and extending existing classes (inheritance)
- implementing binary search tree data structure

This is a chance for you to modify the code that you previously wrote in this class. For those of you who did not do well on the first project, it is a chance to fix the problems that the graders pointed out.

Start early! This project may not seem like much coding, but debugging always takes time. Make sure to ask questions during recitations, in class and on Piazza way ahead of the due date.

Dataset

In this project you will be working with open data. Wikipedia has a good description of open data: ”Open data is the idea that some data should be freely available to everyone to use and republish as they wish, without restrictions from copyright, patents or other mechanisms of control.”

The data set that you need can be found at the NYC OpenData website at `https://goo.gl/PAHDV1`. For your convenience, you can also download the csv[1] file from the course website. (There are several different data formats available on NYC OpenData site. Your program **has to** work with the csv format of the data.)

[1]CSV = Comma Separated Values. A csv file is a plain text file that can be opened either using a plain text editor or a spreadsheet program. Each row is stored in its own line and colum entries are separated by commas.

The file that you download contains records for over 600,000 trees. Each row has 41 columns, i.e., there are 41 different pieces of information for each tree

You can find a detailed description of the data from each column at https://goo.gl/TF5qdB (this description seems to be missing the listing of column with index 14 that appears between `user_type` and `root_stone` headers in our dataset). Your program will work with a subset of those columns.

Each valid line in the dataset should contain 41 columns. Some of these columns may be empty. The columns are determined by commas separating each entry. This means that a valid line has to contain 40 commas separating the entries (even if the entries are empty). The program should silently skip over any invalid lines.

The last section of this specification has a function that can be used to split a singe line from a CSV file into separate entries. You may use it, if you wish, or you can design your own.

User Interface

Your program has to be a **console based** program (no graphical interface, i.e. the program should not be opening any kind of windows to obtain user's input). If you have written only Java FX based programs in cs101, talk to your instructors or the tutors to make sure you know how to develop this program correctly!

Program Usage

This program should use command line arguments. When the user runs the program, he/she will provide the name of the input file containing the list of trees as a command line argument. (This way the program can be used with a similar data sets from other years or subsets of the data set containing only partial data.)

The user may start the program from the command line or run it within an IDE like Eclipse - **from the point of view of your program this does not matter**.

If the name of the input file provided as a command line argument is incorrect or the file cannot be opened for any reason, the program should display an error message and terminate. It should not prompt the user for an alternative name of the file. If the program is run without any arguments, the program should display an error message and terminate. It should not prompt the user for the name of the file. The error messages should be specific and should describe exactly what happened, for example:

```
Error: the file NYC_Street_Treeeeees.csv cannot be opened.
```

or

```
Usage Error: the program expects file name as an argument.
```

Any error messages generated by your code should be written to the `System.err` stream (not the `System.out` stream).[2]

Input and Output

The program should run in a loop that allows the user to check popularity of different tree names. On each iteration, the user should be prompted to enter either a name (for which the program computes the results) or the word "quit" (any case of letters should work) to indicate the termination of the program.

The user should not be prompted for any other response.

If the name entered by the user cannot be found in the list of trees stored in the dataset, the program should print a message

```
Tere are no records of TREE_NAME on NYC streets
```

(in which `TREE_NAME` is replaced by the name that the user entered) and continue into the next iteration.

Output format:

The name entered by the user may match names of several species (by match we mean that the name entered by the user is a substring of an actual species name - ignoring the case). The name entered by the user may contain spaces.

[2] If you are not sure what the difference is, research it or ask questions.

Joanna Klukowska
joannakl@cs.nyu.edu

The program should print the list of all the different species matching the name (without repeats) and then print the information regarding the counts of all trees with those species names for NYC and for each borough individually.

Here is how this information should be formatted:

```
Enter the tree species to learn more about it ("quit" to stop):
linden
All matching species:
  american linden
  silver linden
  littleleaf linden

Popularity in the city:
  NYC:           51,267(683,788)    7.50%
  Manhattan :      5,457(65,423)    8.34%
  Bronx :          6,719(85,203)    7.89%
  Brooklyn :     15,299(177,293)    8.63%
  Queens :       20,817(250,551)    8.31%
  Staten Island : 2,975(105,318)    2.82%

Enter the tree species to learn more about it ("quit" to stop):
quit
```

For NYC and each borough: the first value is the total number of the tree different types of linden trees in that borough; the number in parenthesis is the total number of trees in that borough; the last column contains the percentage calculated as the total number of lindens divided by the total number of trees times 100.

The program has to produce the output formatted in aligned columns, with commas grouping the tree digits in larger numbers and with two digits after the decimal point in the last column. This is a perfect place to use `String.format()` or `System.out.printf()` functions.

Here is another sample run of the program for which the user initially enters a name that provides no matches:

```
Enter the tree species to learn more about it ("quit" to stop):
frog

Tere are no records of frog on NYC streets

Enter the tree species to learn more about it ("quit" to stop):
oak
All matching species:
  pin oak
  willow oak
  white oak
  sawtooth oak
  swamp white oak
  scarlet oak
  black oak
  northern red oak
  english oak
  schumard's oak
  southern red oak
  shingle oak
  bur oak

Popularity in the city:
  NYC :           82,867(683,788)   12.12%
  Manhattan :       8,736(65,423)   13.35%
  Bronx :          11,103(85,203)   13.03%
  Brooklyn :       22,372(177,293)   12.62%
  Queens :         30,571(250,551)   12.20%
  Staten Island : 10,085(105,318)    9.58%

Enter the tree species to learn more about it ("quit" to stop):
bur oak
All matching species:
  bur oak

Popularity in the city:
  NYC :            515(683,788)    0.08%
  Manhattan :        36(65,423)    0.06%
  Bronx :            80(85,203)    0.09%
  Brooklyn :       158(177,293)    0.09%
  Queens :         170(250,551)    0.07%
  Staten Island :   71(105,318)    0.07%

Enter the tree species to learn more about it ("quit" to stop):
quit
```

Note that if the count of trees in a given borough is equal to zero, the line wit the particular borough should still be printed and it should contain zeroes for the counts and for the percentage. Your program should not print results using Nan (not a number) values that result from dividing by zero.

Data Sorage and Organization

Your need to provide an implementation of several classes that store the data and compute the results when the program is executed. In particular, your program must implement and use the following classes. You may implement additional classes as well, if you wish.

As you are working on your classes, keep in mind that they should be (and will be) tested separately from the rest of your program.

`MyBST<E extends Comparable <E>>` Class

This is a completely new class in this project. The class has to provide the implementation of a binary search tree. The implementation details can be found in the lecture notes and in our textbook. If you are running into trouble with any of the methods, use these resources to help you along.

This class should implement the following methods:

- `public boolean add(E e)`
- `public boolean remove(Object o)`
- `public boolean contains(Object o)`
- `public E first()`
- `public E last()`
- `public String toString()`
- default constructor `MyBST()`

The implementation of these methods should follow the specifications of the corresponding methods in the Java's `TreeSet<E>` class, `https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html`. Refer to that specification to determine the details about the behavior, the meaning of parameters and the return values and the exceptions being thrown (keep in mind, that some of these exceptions might not need to be thrown explicitly).

The methods `add`, `remove` and `contains` **have to be implemented recursively**.

The data fields in this class should have `protected` access specifiers to simplify the implementation of the `TreeCollection` class which inherits from the `MyBST<E>` class.

Joanna Klukowska
joannakl@cs.nyu.edu

For the implementation of this class, you need to implement a node class. Make sure that this is an external class (defined in its own class file). The class should be named `BSTNode<E extends Comparable<E>>`. Your node class should implement the `Comparable <BSTNode<E>>` interface. It should provide a one parameter constructor. All data fields in the class should be private.

`Tree` Class

The `Tree` class stores information about a particular tree that grows in New York City. The class should store only a subset of the entries from the input file, namely:

- tree_id as a non-negative integer
- tree_dbh as a non-negative integer
- status as a string, valid values: "Alive", "Dead", "Stump", or empty string or `null`
- health as a string, valid values: "Good", "Fair", "Poor", or empty string or `null`
- spc_common (or the name) as a, possibly empty, string, cannot be `null`
- zipcode as a positive five digit integer (This means that any number from 0 to 99999 is acceptable. The values that are shorter should be treated as if they had leading zeroes, i.e., 8608 represents zipcode 08608, 98 represents zip code 00098, etc.)
- boroname as a string, valid values: "Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"
- x_sp as a double
- y_sp as a double

All of the string data fields should be case insensitive - i.e., "Alive", "alive", "ALIVE" and "aLIVe" are all valid values for the status data field.

This class should provide a nine parameter constructor:

```
public Tree ( int id, int diam, String status, String health, String spc,
        int zip, String boro, double x, double y )
```

All parameters have to be validated according to the rules specified above. If the constructor is called with invalid arguments, then an instance of `IllegalArgumentException` should be thrown carrying an appropriate error message.

There should be no default constructor.[3]

This class should **override the `equals` methods** (see the documentation for the `Object` class for details). The two `Tree` objects should be considered equal if their id's and species name (ignoring the case) are the same. The other values should not be considered in the equality. If two `Tree` objects have the same id, but different species name, than an instance of `IllegalArgumentException` should be thrown carrying an error message. (The tree id's should be unique, so there should not be any possibility of creating two `Tree` objects with identical id's but different species names.)

This class should **implement a `boolean sameName ( Tree t )` method**. The method should return `true` if this tree and `t` have the same species names; it should return `false` otherwise. The method should not take the tree's id into consideration. This method should be case insensitive.

This class should **implement `Comparable<Tree>` interface**. The comparison should be done by the species name as the primary key (using alphabetical order), and by the tree id as the secondary key (i.e., when two objects that have the same species name are compared, the comparison should be performed by the id). The comparison method should be case insensitive (i.e., two `Tree` objects with the species name stored as "Baldcypress" and "BaldCypress" should be compared by their id's since their names are the same).

This class should **implement a `int compareName ( Tree t )` method**. The method should return zero if this tree and `t` have the same species names; it should return a negative value if this tree's species name is smaller than the `t` species name (based on the case sensitive comparison of the String objects representing the species name); it should return a positive value if this tree's species name is larger than the `t`'s species name (based on the case sensitive comparison of the String objects representing the species name). The method should not take the tree's id into consideration.

The class should **override the `toString` method**. The details are up to you, but you should make sure that it returns a `String` object that is a meaningful representation of the object on which it is called.

[3]A default constructor is one that can be used without passing any arguments.

`TreeCollection` Class

The `TreeCollection` class should be used to store all the `Tree` objects.

The class should inherit from `MyBST<Tree>` class. (An alternative implementation would be for this class to contain an instance of `MyBST<Tree>` as a data field. For the purpose of this project the `TreeCollection` class **must** inherit from `MyBST<Tree>`.)

The class needs to provide the default constructor that creates an empty list.

```
public TreeCollection ( )
```

The class should implement

- `public int getTotalNumberOfTrees()`

  method that returns the total number of `Tree` objects stored in this list.
  The efficient implementation of this method should run in $O(1)$.

- `public int getCountByTreeSpecies ( String speciesName )`

  method that returns the number of `Tree` objects in the list whose species matches the `speciesName` specified by the parameter. This method should be case insensitive. If the method is called with a non-existent species, the return value should be 0.
  The efficient implementation of this method should not visit every object stored in the collection. Instead, it should only traverse the smallest number of the tree branches required to discover all trees with the matching species. See example below.

- `public int getCountByBorough ( String boroName )`

  method that returns the number of `Tree` objects in the list that are located in the borough specified by the parameter. This method should be case insensitive. If the method is called with a non-existent borough name, the return value should be 0. **The efficient implementation of this method should run in $O(B)$ where B is the number of boroughs**, not the number of `Tree` objects stored in this `TreeCollection`.

- `public int getCountByTreeSpeciesBorough ( String speciesName, String boroName )`

  method that returns the number of `Tree` objects in the list whose species matches the `speciesName` specified by the first parameter and which are located in the borough specified by the second parameter. This method should be case insensitive. If the method is called with a non-existent borough name or species, the return value should be 0. **The efficient implementation of this method should not visit every object stored in the collection.** Instead, it should only traverse the smallest number of the tree branches required to discover all trees with the matching species. See example below.

- `public Collection<String> getMatchingSpecies(String speciesName)`

  method that returns a `Collection<String>` object containing a list of all the actual tree species that match a given parameter string `speciesName`. The actual species matches `speciesName` if `speciesName` is a substring of the actual name (case insensitive). The list returned by this function should not contain any duplicate names.
  The efficient implementation of this method should run in $O(S)$ where S is the number of the unique tree species, not the number of `Tree` objects stored in this `TreeCollection`.

The class should **override the `toString` method**. The details are up to you, but you should make sure that it returns a `String` object that is a meaningful representation of the object on which it is called (it may or may not contain the listing of all of the elements).

You may implement other methods, if you wish.

Examples of efficient way of traversing the collection in search for matching `Tree` objects.
Consider the bst in figure 1. In each node, the letter indicates the species name and the number indicates the tree id.

An efficient implementation should visit only the nodes shaded in gray when searching for `Tree` objects whose name matches ”F”.

Similarly, an efficient implementation should visit only the nodes shaded in blue when searching for `Tree` objects whose name matches ”A”.

`NYCStreetTrees` Class

The `NYCStreetTrees` class is the actual program. This is the class that should contain the `main` method. It is responsible for opening and reading the data files, obtaining user input, performing some data validation and handling all errors that may occur (in particular, it

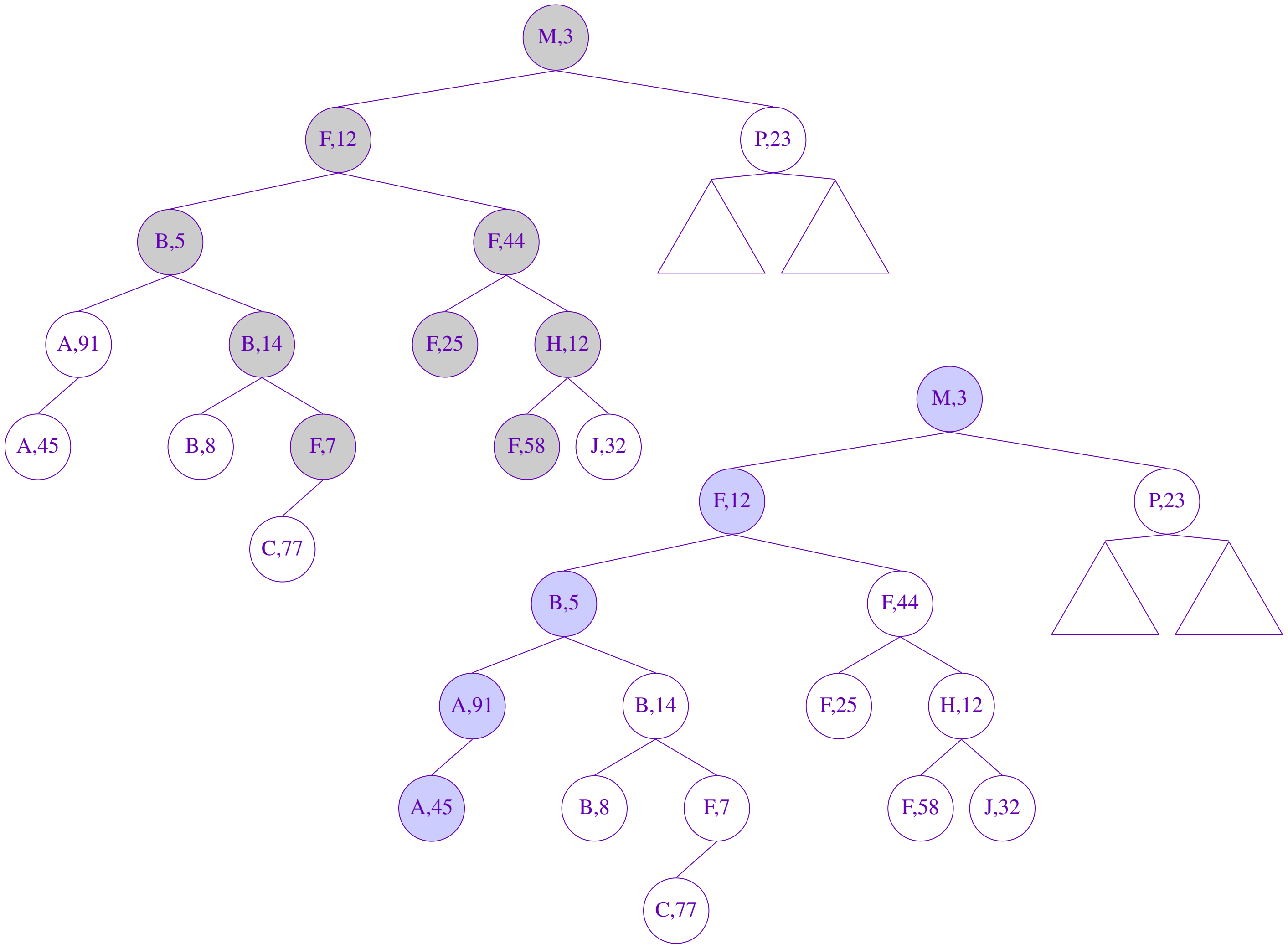

Figure 1: Efficient search for matching species.

should handle any exceptions thrown by your other classes and terminate gracefully, if need be, with a friendly error message presented to the user).

You may (and probably should) implement other methods in this class to modularize the design.

Programming Rules

You should follow the rules outlined in the document *Code conventions* posted on the course website at `http://cs.nyu.edu/~joannakl/cs102_s17/notes/CodeConventions.pdf`.

The data file should be read only once! Your program needs to store the data in memory resident data structures.

You may use any classes to handle the file I/O, but probably the simplest ones are **`File`** and **`Scanner`** classes.

Working on This Assignment

You should start right away!

You should modularize your design so that you can test it regularly. Make sure that at all times you have a working program. You can

Joanna Klukowska
joannakl@cs.nyu.edu

implement methods that perform one task at a time. This way, if you run out of time, at least parts of your program will be functioning properly.

If you have trouble deciding in what order things should be implemented here are some ideas for parts of the program that should be independent:

- fix the I/O related problems from project 1 (if any)
- fix any problems that the `Tree` class had → add the two new methods
- design tests for `MyBST<E>` class (using the TDD approach)
- implement the methods of the `MyBST<E>` class
- modify the `TreeList` to adhere to the `TreeCollection` description

You should make sure that you are testing the program on much smaller data set for which you can determine the correct output manually. You should create your own small test files for that purpose. (Feel free to share those with other students on Piazza.)

You should make sure that your program's results are consistent with what is described in this specification by running the program on carefully designed test inputs and examining the outputs produced to make sure they are correct. The goal in doing this is to try to find the mistakes you have most likely made in your code.

You should backup your code after each time you spend some time working on it. Save it to a flash drive, email it to yourself, upload it to your Google drive, do anything that gives you a second (or maybe third copy). Computers tend to break just a few days or even a few hours before the due dates - make sure that you have working code if that happens. (A second copy of the files on the same computer is a good idea to keep multiple versions, but it is NOT a good backup since you do not have access to it if there are problems with your computer.)

Grading

If your program does not compile or if it crashes (almost) every time it is run, you will get a zero on the assignment.

If the program does not adhere to the specification, the grade will be low and will depend on how easy it is to figure out what the program is doing.

20 points efficiency of the implementation (this refers mostly to the `MyBST<E>` and `TreeCollection` classes)

40 points program and class correctness: the correct values and format of output, correct behavior of methods, handling of invalid arguments

20 points design and implementation of the three required classes and any additional classes

20 points proper documentation, program style and format of submission

How and What to Submit

Your should submit all your source code files (the ones with .java extensions only) in a single **zip** file to NYU Classes.

You can produce a zip file directly from Eclipse:

- right click on the name of the package (inside the `src` folder) and select Export...
- under General pick Archive File and click Next
- in the window that opens select appropriate files and settings:
  - in the right pane pick ONLY the files that are actually part of the project, but make sure that you select all files that are needed
  - in the left pane, make sure that no other directories are selected
  - click Browse and navigate to a location that you can easily find on your system (Desktop or folder with the our course materials or ...)
  - in Options select "Save in zip format", "Compress the contents of the file" and "Create only selected directories"
- click Finish

Do not submit any data files in your zip file!