Assignment 2: New York City Street Trees

1 Overview

In 2016, New York City made public the results of *TreesCount!*, the *2015 Street Tree Census*, conducted by volunteers and staff organized by the NYC Department of Parks & Recreation as well as partner organizations. The data includes information about more than 680,000 trees on the streets of New York City. In this project, you will write a program that will allow a user to visualize certain aspects of the tree data, such as how many trees of a given species there are, borough by borough. The user will be able to specify a part of a species common name, such as "oak" and the program will display the frequency of occurrence of all types of oak trees throughout the city, such as pin oaks, sawtooth oaks, scarlet oaks, and white oaks.

The data set is part of the NYC OpenData website and can be found here:

`https://data.cityofnewyork.us/Environment/2015-Street-Tree-Census-Tree-Data/uvpi-gqnh`

You may find it interesting to take a look at an online visualization project based on an older New York City tree census data set at `http://www.cloudred.com/labprojects/nyctrees/`.

Your project will be storing the tree data in a searchable container class, specifically an AVL tree. In other words, it will be a tree containing tree data!

2 Objectives

This project is designed with a few objectives in mind:

- to give you exposure to and experience with large, open data sets. Open data sets are to data what open source software is to software. No one has proprietary rights to the data. You can download it and analyze it for free. 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."
- to require that you implement an AVL tree, some of the methods of which are slightly modified, as will be explained below.
- to require that you write implementations for two class interfaces that are provided to you and cannot be modified, as well as a main program that is a client of those classes. This gives you experience writing code that has been specified by someone else.

3 About The Data Set

The NYC OpenData website for this tree census data,

`https://data.cityofnewyork.us/Environment/2015-Street-Tree-Census-Tree-Data/uvpi-gqnh`

gives you the means to download the data in various formats. Your program has to work with the *csv* format of the data. A file in *csv* format, in case you are not familiar with it, is a ***comma-separated-values*** file. A comma-separated-values file is a plain text file in which each line represents a single record, and within the line, commas separate the individual fields of the record. (Fields can contain commas if they are within quoted strings.) Spreadsheet applications let you import csv files to view their contents by rows and columns. The tree data file that can be downloaded contains records for over 680,000 trees. Each row represents a single tree (or tree stump) and has 41 columns, which means that there are 41 different pieces of information for each tree. Although this dataset has 41 columns, your program will not store the entire record; it will store and work with a much smaller subset of those columns.

A detailed description of the meaning and form of every column[1] in that dataset can be found in the ***data dictionary*** described here:

[1] This description is missing the description of the column with index 14 that appears between the *user_type* and *root_stone* columns in the dataset.

`https://data.cityofnewyork.us/api/views/uvpi-gqnh/files/8705bfd6-993c-40c5-8620-0c81191c7e25?download=true&filename=StreetTreeCensus2015TreesDataDictionary20161102.pdf`

This data dictionary is also available on our server in the `resources` subdirectory of the `cs335_sw` directory. Each valid line in the dataset should contain 41 columns. Some of these columns may be empty. The columns are determined by the 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. Your main program will have to parse each line and split it into its 41 separate fields.

4 Program Invocation, Usage, and Behavior

The program is invoked from the command line and expects a single command line argument, which specifies the ***csv*** file to be opened for reading. If no file is specified, it is a usage error and the program must write an appropriate and meaningful error message onto the standard error stream, after which it must exit. If the file that is specified does not exist or cannot be opened for some reason, the program must write an appropriate and meaningful error message onto the standard error stream and then exit.

Assuming that the file was opened successfully, the program must read the entire file, line by line, parsing the lines and storing certain information contained in them. If the file has invalid lines, they should be skipped over. The program must declare and create a `TreeCollection` object. The `TreeCollection` object is responsible for storing the data contained in the input file and for computing properties based on that data. The main program will call on the methods of the `TreeCollection` class to do most of its work. The specification of the `TreeCollection` class and its public methods and required implementation details is contained in Section 7. The `TreeCollection` is essentially a container that stores `Tree` objects in an AVL tree, and also stores other information contained in the file. Each `Tree` object is uniquely identified in the data set by a numeric `tree_id` field, but because tree ids are not a user-friendly way to find trees, the species common name field, denoted `spc_common`, and `tree_id` field, as a pair, will be used as the primary and secondary keys of the ordering relation maintained internally by the `TreeCollection` object. The user does not need to know this though.

Once the entire file has been read and stored into `TreeCollection` object, the program will enter an interactive mode, which is explained in Section 4.2 below.

4.1 Processing the Input File

It is the task of the main program to read the input file, parse its lines, construct a `Tree` object for each line and make the calls to the `TreeCollection` class to insert that object into the collection. As each line of data is read, its 41 fields must be separated, and the proper subset of nine of them must be used to construct the `Tree` object. The `TreeCollection` will use the (`spc_common, tree_id`) pair as the unique key for inserting the `Tree` objects into its encapsulated AVL tree. The `TreeCollection` will also keep track of the names of all species that it stored, and in which boroughs the trees are located.

4.2 Interactive Mode

After all input has been processed, the program enters an interactive loop where it prompts the user to enter input and then responds to it. In each iteration, the user must be prompted to enter either the name of a species of tree, or the word "`quit`" to indicate that she wants to quit the program. The word "`quit`" should be checked case insensitively, i.e., if the user types "`quit`" in any form, the program should terminate. ***The user should not be prompted for any other type of input.***

The user is allowed to enter either the full name of a species, such as "southern white oak", or one or more whole words that are a substring of that name, such as "oak", "white", "southern", "southern white", or "white oak". It is the responsibility of the program to determine which tree species names that were stored when the input file was read ***match*** the user's input. Matching is now defined.

Let us call the user's entered string, `tree_to_find`, and let us call the complete species name against which it is compared, `tree_type`. For the purpose of matching, a hyphen character is treated like a whitespace character - it separates two distinct words. Thus, for example, "Douglas-fir" consists of two words, "Douglas" and "fir". Then `tree_to_find` ***matches*** `tree_type` if any of the following conditions are true:

- `tree_to_find` is exactly the same as `tree_type`, ignoring case.

- If `tree_to_find` has no whitespace or hyphen characters (it is one word) then if `tree_type` contains white space characters or hyphens and consists of the words w_1, w_2, ..., w_k, then `tree_to_find` is exactly one of w_1, w_2, ..., w_k.
- if `tree_to_find` has whitespace or hyphen characters, then then if `tree_type` contains white space characters or hyphens and consists of the words w_1, w_2, ..., w_k, then `tree_to_find` is some subsequence of the sequence of words w_1, w_2, ..., w_k.

Otherwise `tree_to_find` does not match `tree_type`. Examples:

`oak` matches "`white oak`"

`birch` matches "`paper birch`"

`fir` matches "`Douglas-fir`"

"`two-winged`" matches "`two-winged silverbell`"

`Japanese` matches each of "`Japanese hornbeam`", "`Japanese maple`", and "`Japanese tree lilac`"

`range` does not match "`Osage-orange`"

"`bur oak`" matches "`bur oak`"

`chest` does not match "`chestnut`"

`locust` does not match "`honeylocust`"

With matching so defined, we can state what the program does when the user enters a tree species name. The set of all matching full species names is constructed and all matching species names are obtained. Then, all occurrences of any of the matched tree types are found in the stored data, the program determines which borough each is in, and displays the number of those types of trees in the city in total, and in each borough, as well as the percentage of total trees that this represents, in the city as a whole, and in each borough. A sample session should look like this:

```
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):
```

In the above display, for NYC and for each borough:

- the first value is the total number of the different types of linden trees in that borough;
- the number in parenthesis is the total number of trees in that borough;
- the last number is 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.

To be clear, although there may be multiple, distinct species, when multiple species match the user's input tree name, the counts for all species that match are summed and the totals are used in the output display.

5 Program Organization and Implementation

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 may be) tested separately from the rest of your program.

6 The Tree Class

As noted above, the csv file has 41 fields, but your program will use and store a subset of them. The `Tree` class represents a single tree on some street in New York City. The header file for the `Tree` class shall be stored in the file `tree.h`, which will be accessible in the `cs335_sw/resources` directory on the server. The `Tree` class encapsulates the following fields of the data set:

- `int tree_id`; a non-negative integer that uniquely identifies the tree
- `tree_dbh`; a non-negative integer specifying tree diameter
- `string status`; a string, valid values: "`Alive`", "`Dead`", "`Stump`", or the empty string
- `string health`; a string, valid values: "`Good`", "`Fair`", "`Poor`", or the empty string
- `string spc_common`; the common name of the tree, such as "`white oak`" or a possibly empty string
- `int zipcode`; 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.)
- `string boroname`; valid values: "`Manhattan`", "`Bronx`", "`Brooklyn`", "`Queens`", "`Staten Island`"
- `double x_sp`; x spatial coordinate specifying position in New York State *state plane coordinates*
- `double y_sp`; y spatial coordinate specifying position in New York State *state plane coordinates*

These are the private data members of the `Tree` class. 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. The spatial coordinates can be converted for use by GPS and other systems for actually mapping the trees onto a map.

The `Tree` class provides the following public methods. You may add other methods but these must be implemented in the project in a `tree.cpp` implementation file. All methods must be case insensitive when comparing string data. The fact that some of these methods may not be used by your program does not mean you do not need to implement them.

The class must also provide the following public methods. Your program must use the public interface described below. It cannot modify it in any way. Your implementation file must implement exactly what this interface file defines. You are free to define the private part in any way that you like, but if the public part deviates in any way and my client cannot make calls to it, it will be considered incorrect.

Method Syntax	**Description**
`Tree ( int id, int diam, string status, string health, string spc, int zip, string boro, double x, double y )`	A constructor for the class.
`friend bool operator==(const Tree & t1, const Tree & t2)`	Given two `Tree` objects, it returns true if and only if they have the same species common name (`spc_common`) and `tree_id`, (case insensitive).

Method Syntax	Description
`friend bool operator<(const Tree & t1, const Tree & t2)`	This compares the two trees using `spc_common` as the primary key and `tree_id` as the secondary key and returns true if `t1` is less than `t2` in this ordering and false otherwise. (case insensitive)
`friend ostream& operator<< (ostream & os, const Tree & t )`	This prints a `Tree` object onto the given ostream. Each of the members of the `Tree` object should be printed, with the tree `spc_common` name first, then the `tree_id`. After that the order is not specified - it can be whatever the programmer decides as long as it is clear.
`friend bool samename(const Tree & t1, const Tree & t2)`	This returns true if and only if the two trees passed to it have identical `spc_common` members. This differs from `operator==` because it ignores the `tree_id`. (case insensitive)
`friend bool islessname(const Tree & t1, const Tree & t2)`	This returns true if and only the `spc_common` member of the first `Tree` object is smaller than that of the second as strings. This differs from `operator<` because it ignores the `tree_id`. (case insensitive) For example, `mytree.follows("mimosa")` is true if `mytree.spc_common == "pine"`
`string common_name()`	This returns the `spc_common` name of the `Tree`.
`string borough_name()`	This returns the name of the borough in which the `Tree` is located.
`double diameter()`	This returns the value of the `tree_dbh` member.

7 The TreeCollection Class

The `TreeCollection` class provides the functionality to the main program for storing and accessing tree data and its properties. It must encapsulate three containers:

- an AVL tree that stores the tree objects;
- a `TreeSpecies` container that stores the set of all `spc_common` tree species names that were found in the input data set, and
- a container that stores the names of each New York City borough and how many trees from the data set are in each borough.

These containers must be private or protected members of the `TreeCollection` class. The `TreeCollection` object has two other tasks to perform when it is given a `Tree` object to insert into its AVL tree:

1. It must determine within which borough the tree is located and update the count of the total number of `Tree` objects located in that borough, even if the tree is dead or just a stump. If an object is inserted into the AVL tree, then it is part of the "census."
2. For each `Tree` object, if the `spc_common` member is not an empty string, it should be stored into the `TreeSpecies` container, so that, when the entire file has been read and its trees stored, the `TreeCollection` will have a list of all of the species names of trees that have been stored into the AVL tree. For example, if the input file has ten lines consisting of three magnolia trees, two mimosas,

one white oak, and four mulberry trees, then the `TreeSpecies` container would contain just these four names: magnolia, mimosa, mulberry, white oak. ***It cannot contain duplicates of these names.***

There are several different containers that the `TreeCollection` can use:

- a simple, unsorted vector. The advantage is that it is easy to maintain, but will require linear searches of the entire vector each time a record is read. On the other hand, there will not be more than about 130 different species names, so the time to search will not be a severe impediment.
- a simple, unsorted list. Like the vector, it will also require full linear searches for each record that is read but will be even slower in searching.
- a sorted list. A lot more work to implement since there is no class template for it in C++, but it reduces insertion time by about half on average.
- a C++ `set` class template. This is a built-in container class that does all of the work for you. A `set` is a class template that allows insertion and automatically prevents duplicates. The advantage is speed and simplicity. The disadvantage is that you need to learn something new. (But you are a student - isn't it a good thing to learn soemthing new?)

No matter which container you decide to use, let us now suppose that it is named `TreeSpecies` in `TreeCollection` class. When the program is in interactive mode and the user enters a partial species name, such as `oak`, instead of "`white oak`" or "`pin oak`", the `TreeSpecies` container will be checked to see which trees match the words that the user entered according to the matching rules described in Section 4.2 above. The set of matching tree names in `TreeSpecies` will be used for searching the AVL tree.

The class must also provide the following public methods. In the descriptions, whenever they refer to "matching," it is by the rules described in Section 4.2 above. Your program must use the public interface described below. It cannot modify it in any way. Your implementation file must implement exactly what this interface file defines. You are free to define the private part in any way that you like, but if the public part deviates in any way and my client cannot make calls to it, it will be considered incorrect.

Method Syntax	**Description**
`TreeCollection ( )`	A default constructor for the class that creates an empty AVL tree, and empty `TreeSpecies` container, and an empty `BoroughNames` container.
`~TreeCollection ( )`	A destructor for the class.
`int total_tree_count()`	This returns the total number of `Tree` objects stored in the collection.
`int count_of_tree_species ( const string & species_name )`	This returns the number of `Tree` objects in the collection whose `spc_common` species name matches the species_name 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.
`int count_of_trees_in_boro( const string & boro_name )`	This returns the number of `Tree` objects in the collection 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.

Method Syntax	Description
`list<string> get_matching_species(const string & species_name)`	This returns a `list<string>` object containing a list of all of the actual tree species that match a given parameter string `species_name`. This method should be case insensitive. The list returned by this function should not contain any duplicate names and may be empty.

8 The AVL Tree Interface

Your program must use the public AVL tree class interface contained below. It cannot modify it in any way. Your AVL tree implementation file must implement exactly what this interface file defines. You are free to define the private part in any way that you like, but if the public part deviates in any way and my client cannot make calls to it, it will be considered incorrect.

The AVL tree class will use the `operator<` and `operator==` methods of the `Tree` class to implement the `insert()`, `remove()`, `find()`, `findMin()`, and `findMax()` methods. Thus, in AVL tree implementation code such as

```
if ( current_tree < current_node->tree )
```

where `current_tree` and `current_node->tree` are both `Tree` objects, the `<` operator being invoked is really the overloaded `operator<` from the `Tree` class. In short, your AVL tree does not need to "know" how the trees are compared during insertion because the `Tree` class has its own "compares" method. But to implement the `findallmatches()` method, it must use the `samename()` and `islessname()` methods, because they rely on only the primary key.

```
class AVL_Tree
{
public:
    AVL_Tree  ( );                          // default
    AVL_Tree  ( const AVL_Tree & tree);  // copy constructor
    ~AVL_Tree ( );                          // destructor

    // Search methods:
    const Tree& find    ( const Tree & x ) const;
    const Tree& findMin () const;
    const Tree& findMax () const;
    list<Tree>& findallmatches ( const Tree & x ) const;

    // Displaying the tree contents:
    void   print   ( ostream& out ) const;

    // Tree modifiers:
    void   clear();                          // empty the tree
    void   insert( const Tree& x);  // insert element x
    void   remove( const Tree& x);  // remove element x
};
```

The `findallmatches()` method must search the entire tree for all occurrences of `Tree` objects that match its `Tree` argument. I have not provided code for you to implement this. Your task is to figure out how to do it without having to examine every single node in the tree every time it is called and without missing any matching objects.

Testing Your Program

You should make sure that you are testing the program on a 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.

Be warned - do not try to use a large data set when writing and debugging the code. If you do, you will discover that it can be hours before you see results on typical laptops and desktop computers.

Programming Rules

Your program must conform to the programming rules described in the Programming Rules document on the course website. It is to be your own work alone.

Grading Rubric

The program will be graded based on the following rubric, based on 100 points.

- A program that cannot run because it fails to compile or link on a `cslab` host receives only 20%. This 20% will be assessed using the rest of the rubric below.
- Meeting the requirements of the assignment, including performance requirements: 60%
- Design (modularity and organization): 15%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that does not compile on a `cslab` host cannot receive more than 20 points.

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on March 30, 2017. Create a directory named named *`username`*`_project2`. where *username* is to be replaced by your CS Department network login name. Put all project-related source-code files into that directory. ***Do not place any executable files, data files, or object files into this directory.*** You will lose 1% for each file that does not belong there, and you will lose 2% if you do not name the directory correctly[2].

Next, create a zip archive for this directory by running the zip command

```
zip -r username_project2.zip ./username_project2
```

This will compress all of your files into the file named `username_project2.zip`. Do not use the tar compress utility.

The command requires two arguments: the number of the project and the pathname of your file. Thus, if your file is named `username_project2.zip` and it is in your current working directory you would type

```
submit_cs335_assignment 1  username_project2.zip
```

The program will copy your file into the `project2` subdirectory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project2/
```

[2] I have scripts that process your submissions automatically and misnamed files force me to manually override them.

and if it is successful, it will display the message, "`File ... successfully submitted.`"

You will not be able to read this file, nor will anyone else except for me. But you can double-check that the command succeeded by typing the command

```
ls -l /data/biocs/b/student.accounts/cs335_sw/projects/project2
```

and making sure you see a non-empty file there.

If you put a solution there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.