# Programming Project 2: Processing 2015 New York City Street Tree Census Data

## 1 Overview

This project extends what you did in Project 1 and is a much larger project. In Project 1, you wrote a program that read the NYC open data set, *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 again read the data from that data set, but this time around, you will do some processing with that data. You will write a program that will allow a user to summarize certain aspects of the data, such as how many trees of a given species are growing, borough by borough, or which trees are within a given distance of a given GPS location. The user will be able to specify a fragment 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 an abstract tree containing real tree data! As with the first project, you will be given portions of the code, which I have written, to reduce your programming effort.

## 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 give you experience writing an AVL tree class, some of the methods of which are slightly modified, as will be explained below.
- to get you to 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.
- to give you practice writing code that uses a class whose implementation is hidden.

## 3 About The Data Set

It is the same data set that was the basis of Project 1. 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/`.

The NYC OpenData website for this tree census data 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 also contain commas if they are within quoted strings, e.g., "Brooklyn, New York" is a single field.) 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. *The data set that is downloaded will have as its first row, the labels of its columns. For this assignment, you should delete that row, so that the program can assume all rows are actual data rows. (The version of the data set that I provide on the server has that first row deleted.)*

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

`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 contains 41 columns. Some of these columns may be empty. An empty column is represented by two commas with no intervening characters. The columns are determined by the commas separating each entry. This means that a valid line has to contain at least 40 commas separating the entries (even if the entries are empty), and maybe more, if the fields contain embedded commas. While there should not be invalid lines in the file, if any are found, the program should handle them by ignoring them.

## 4 Program Invocation, Usage, and Behavior

The program is invoked from the command line and expects two command line arguments, which specify respectively (1) the ***csv*** file to be opened for reading and (2) the sequence of user commands to be processed. If two files are not 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 a 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 both files are opened successfully, the program must read the entire csv file, line by line, parsing the lines and storing certain information contained in them in a `TreeCollection` object. If the file has invalid lines, they should be skipped over. The `TreeCollection` object is responsible for storing the data contained in the input file and for computing various properties of 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 5.3. 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 level of detail, but you as a programmer, do.

Once the entire csv file has been read and stored into `TreeCollection` object, the program will process the commands from the command file. These commands are described and 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 ten of them must be used to construct the `Tree` object[2]. 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 common names of all species that it stored, and in which boroughs the trees are located. I provide a class

[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.

[2]If you did Project 1 successfully, you already have the various code modules that can do this task

named `TreeSpecies` that encapsulates the set of all species common names found in the data set, and which provides various methods for interacting with this set of names. Again, the details are below.

## 4.2 Command Processing

After all input has been processed, the program enters a command processing loop, in which it reads commands from the second file specified on the command line. The syntax of the possible commands from that file is as follows. The `bold` text is the command and the ***italicized*** text is its parameter list.

| **Command** | **Description** |
|---|---|
| `tree_info` ***`tree_to_find`*** | where ***`tree_to_find`*** is a string that might contain white space. This command lists certain information about the trees whose common name matches ***`tree_to_find`***. Details about output and the definition of "matches" are below. |
| `listall_names` | Lists all tree common names found in the `TreeCollection`. |
| `listall_inzip` ***`zipcode`*** | where ***`zipcode`*** is an `int` type. Lists the common names of all trees found in the given zipcode, together with their frequencies. For example, if three red oaks and two spruce are in the given zipcode it lists<br>`red oak:  3`<br>`spruce:  2` |
| `list_near` ***`latitude longitude dist`*** | Lists the common names and frequencies of all trees within ***`dist`*** kilometers of the given point (***`latitude,longitude`***). |

### 4.2.1 Details About the Commands

**The `tree_info` Command** The argument to the `tree_info` command is a string consisting of a single word or one or more words with intervening whitespace or hyphens. The sequence can be 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". The program must determine which tree species names that were stored when the input file was read ***match*** the argument string. Fortunately for you, I have implemented the matching algorithm and functions needed to use it, and made all of those functions part of the `TreeSpecies` class. This class will be provided to you in two files, `tree_species.h` and `tree_species.o` (a binary). The interface will be an appendix to this document as well. Even though I provide it, you need to know how matching is defined.

Let us call the argument 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 string 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 the words $w_1$, $w_2$, ..., $w_k$. For example if tree_to_find is Japanese, and tree_type is Japanese tree lilac, then tree_to_find matches tree_type.
- 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 $w_i w_{i+1} ... w_j$ of the sequence of words $w_1$, $w_2$, ..., $w_k$. So tree lilac matches Japanese tree lilac, but tree lilac does not match lilac.

Otherwise `tree_to_find` does not match `tree_type`. Examples both positive and negative:

`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`"

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

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

"`Japanese horn`" does not match "`Japanese hornbeam`"

`chest` does not match "`chestnut`"

`locust` does not match "`honeylocust`"

With matching so defined, we can state what the program does when the `tree_info` command is processed. The program uses the given words to try to match one or more species common names. The set of all such matching common names is constructed. Then, all occurrences of any of the matched tree types are searched for 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. For example, the output for the command

```
tree_info linden
```

should look like this:

```
linden

All matching species:

    american linden
    silver linden
    littleleaf linden

Frequency by borough:

     Total in 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%
```

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.

**The `listall_names` Command** This command lists all tree common names for all trees found in the `TreeCollection` object (i.e., all those stored in the `TreeSpecies` object.)

**The `listall_inzip` Command** This command lists the common names for all trees in the given zipcode together with their number of occurrences in that zipcode, such as

```
white oak: 12
red oak:   15
sycamore:  32
```

Note that the `TreeSpecies` object does not store locations, so it is of no use in implementing this command.

**The `list_near` Command** This command lists the common names and the frequencies of all trees whose distance from the given point (*`latitude,longitude`*) is at most *`dist`* kilometers. In order to compute this, you need to compute the distance between two points given by decimal values of latitude and longitude (i.e., GPS coordinates). If the distance between the given point and a tree is at most *dist*, then that tree is counted in the result of this command. The distance between two such points can be computed using the *Haversine* formula, which follows.

#### 4.2.2 Distance Between Two Points on Sphere (The Haversine Formula)

The *Haversine* formula (see https: //en.wikipedia.org/wiki/Haversine_formula) can be used to compute the approximate distance between two points when they are each defined by their latitude and longitude in degrees. The distance is approximate because (1) the earth is not really a sphere, and (2) numerical round-off errors occur. Nonetheless, for points that are no more than ten kilometers apart, the formula is accurate enough. Given the following notation

$d$ : the distance between the two points (along a great circle of the sphere,

$r$ : the radius of the sphere,

$\varphi_1$, $\varphi_2$: latitude of point 1 and latitude of point 2, in radians

$\lambda_1$, $\lambda_2$: longitude of point 1 and longitude of point 2, in radians

the formula is

$$2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\left(\varphi_1\right)\cos\left(\varphi_2\right)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

A C++ function to compute this formula in a numerically efficient way is given in Listing 1.

Listing 1: Haversine Function (corrected version)

```
#include <cmath>
# To build , link to the math library using -lm

const double R = 6372.8              // radius of earth in km
const double TO_RAD= M_PI / 180.0; // conversion of degrees to rads

double haversine( double lat1 , double lon1 , double lat2 , double lon2)
{
    lat1        = TO_RAD * lat1;
    lat2        = TO_RAD * lat2;
    lon1        = TO_RAD * lon1;
    lon2        = TO_RAD * lon2;
    double dLat = (lat2 - lat1)/2;
    double dLon = (lon2 - lon1)/2;
    double a    = sin(dLat);
    double b    = sin(dLon);

    return 2 * R * asin(sqrt(a*a + cos(lat1)*cos(lat2)*b*b));
}
```

## 5 Program Organization and Files

You 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/or use the classes listed in this section. 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.

### 5.1 The `Tree` Class

As stated above, the csv file has 41 fields, but your `Tree` object will 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 should be stored in a file named `tree.h`. The `Tree` class must encapsulate the following fields of the data set:

- `string spc_common`; the common name of the tree, such as “`white oak`” or a possibly empty string
- `int tree_id`; a non-negative integer that uniquely identifies the tree
- `integer 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 address`: nearest estimated address to tree
- `string boroname`; valid values: ”`Manhattan`”, ”`Bronx`”, ”`Brooklyn`”, ”`Queens`”, ”`Staten Island`”
- `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.)
- `double latitude`; specifies GPS latitude of the tree point, in decimal degrees
- `double longitude`; specifies GPS longitude of the tree point, in decimal degrees

These must be private data members of the `Tree` class. All of the string data fields should store the data in the exact case (upper or lower) as it is in the original input file. The spatial coordinates are GPS coordinates that can be used to locate the trees on a map.

The `Tree` class must provide ***at least*** the following public methods. You may add other methods if you think they are necessary. In any case, the `Tree` class implementation file must be in a file named `tree.cpp` implementation file. All methods must be case insensitive when comparing string data. Your program should not modify these methods in any way.

| **Method Syntax** | **Description** |
|---|---|
| `Tree(const string & treedata);` | A constructor for the class that takes a string from a csv file. |
| `Tree ( int id, int diam, string status, string health, string spc, int zip, string addr, string boro, double latitude, double longitude );` | 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). |

| | |
|---|---|
| `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, in the exact same order as they are described in the table above, e.g., with the tree `spc_common` name first, then the `tree_id`. Fields should be separated by commas in the output stream. |
| `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() const;` | This returns the `spc_common` name of the `Tree`. |
| `string borough_name() const;` | This returns the name of the borough in which the `Tree` is located. |
| `string address() const;` | This returns the street address nearest to which the `Tree` is located. |
| `int diameter() const;` | This returns the value of the `tree_dbh` member. |
| `int zipcode() const;` | This returns the value of the `zipcode` member of the tree. |
| `void get_position(double & latitude, double & longitude) const;` | This stores into its two paremeters the `latitude` and `longitude` of the Tree. |

## 5.2 The `TreeSpecies` Class (Provided to You)

The `TreeSpecies` class encapsulates the set of all common names of trees found in the input data set. For example, if the data set has trees with common names *pin oak*, *red oak*, *red pine*, and *spruce*, then the `TreeSpecies` class would contain each of these names exactly once, regardless of how many individual trees of each type exist. This class also provides the functionality to test if a word matches a particular tree common name, and to list all common names that are matched by a word. The class interface is defined as follows.

| **Method Syntax** | **Description** |
|---|---|
| `TreeSpecies( );` | A default constructor for the class that creates an empty `TreeSpecies` container. |
| `~TreeSpecies( );` | A destructor for the class. |

| Method Syntax | Description |
|---|---|
| `void print_all_species(ostream & out) const;` | This writes the set of all common names found in the data set to the output stream `out`, one per line. There is no particular order in which they are written. |
| `int number_of_species() const;` | This returns the total number of distinct species common names found in the data set. |
| `int add_species( const string & species);` | This adds the species to the `TreeSpecies` container. It returns a 0 if the species was already in the `TreeSpecies` container and a 1 if it was not. |
| `list<string> get_matching_species(const string & partial_name) const;` | This returns a `list<string>` object containing a list of all of the actual tree species that match a given parameter string `partial_name`. This method should be case insensitive. The list returned by this function should not contain any duplicate names and may be empty. |

### 5.3 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 that were found in the input data set,
- 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 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 inserted into the `TreeSpecies` object, 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 does not contain duplicates of these names.*** The `TreeSpecies` container provided to you ensures that this is the case; you do not need to do anything other than call its `add_species()` method.

When the program is processing commands and the `tree_info` command is supplied with a partial 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 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. 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, and you may add additional public methods, but if your public part changes anything described below in any way, it will be considered incorrect. If you wish to make changes, please request permission and justify the change.

| **Method Syntax** | **Description** |
|---|---|
| `TreeCollection ( );` | A default constructor for the class that creates an empty AVL tree, an empty `TreeSpecies` container, and an empty `BoroughNames` container. |
| `~TreeCollection ( );` | A destructor for the class. |
| `int total_tree_count() const;` | This returns the total number of `Tree` objects stored in the collection. |
| `int count_of_tree_species ( const string & species_name ) const;` | 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 ) const;` | 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. |
| `list<string> get_matching_species(const string & species_name) const;` | 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. |
| `list<string> get_all_in_zipcode(int zipcode) const;` | This returns a `list<string>` object containing a list of all of the actual tree species that are located in the given zipcode. |
| `list<string> get_all_near(double latitude, double longitude, double distance) const;` | This returns a `list<string>` object containing a list of all of the actual tree species that are located within `distance` kilometers from the GPS position (`latitude, longitude`). |

### 5.4 The `AVL_Tree` Interface

Your program must use the public AVL tree class interface contained below. You cannot modify the member functions, but you may add others. Your AVL tree implementation file must implement at least what this interface file defines. You are free to define the private part in any way that you like.

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;

    // Traversals:
    // prints the Tree objects onto the ostream using inorder traversal.
    // Each of the members of the Tree object is printed, in the exact
    // same order as they are above, e.g., with the tree spc_common name
    // first, then the tree_id. Fields should be separated by commas
    // in the output stream.
    void   print   ( ostream& out ) const;  // prints the Tree objects

    // Methods to consider adding- getting all trees in a zipcode,
    // or near a given point

    // 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.

Even though this project never deletes trees from the collection, you must have that code!

### 5.5 Required Files

Your program must contain at least the following files:

```
main.cpp
tree.h
tree.cpp
tree_collection.h
tree_collection.cpp
avl.h
avl.cpp
tree_species.h  (provided to you)
tree_species.o  (provided to you)
README
Makefile        (provided to you)
```

The `README` file must contain a running log of your progress and changes and thoughts and possibly frustrations during this project, or your "eureka" moments. It can also contain documentation of the program. There is no hard rule about it. The Makefile I will supply to you, and you can modify it as needed for your code.

## 6 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. Suggestions:

- all trees of a single type, or a single borough, or within a single zipcode
- trees in a file in sorted order and reverse sorted order ( to make sure the trees are constructed in extremem cases correctly)
- empty data set
- data set with one tree
- commands that result in predictable set of outputs, such as points and distances that produce just one tree type, or two tree types.

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.

## 7 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 work and your work alone.

## 8 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 functional requirements of the assignment: 60%
- Design (choices of algorithms, data structures, modularity, 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.

## 9 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on March 29, 2018. 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 and README and Makefile into that directory. ***Do not place any executable files, data files, or object files other than tree_species.o 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[3].

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 submit command that you will use is `submit_cs335_project`. It 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_project 2  username_project2.zip
```

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

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

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.

## 10 The `TreeSpecies` Header File

```
#include <string>
#include <iostream>
#include <list>
#include <set>
#include <vector>
#include "tree.h"
#define  MAXWORDS  10

using namespace std;

/**
   This class is useful for matching. It will be documented more soon...
 */
class SpeciesName {
public:
```

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

```
    /** SpeciesName(s) constructs the list of all words in s, to faciliate
     *  matching.
     */
    SpeciesName(string s);

    /** matches(s) returns true if the string s matches the Species_Name
     * according to the rules in the assignment spec.
     */
    bool matches( string s );

    void print( ostream & out );

private:
    int num_subwords;
    string subwords[MAXWORDS];
    string name;
};


class TreeSpecies
{
public:
    /*
       A default constructor for the class that creates an empty TreeSpecies container.
    */
    TreeSpecies( );

    /* A destructor for the class. */
    ~TreeSpecies( );

        /**
     * This writes the set of all common names found in the data set to the
     * output stream out, one per line. There is no particular order in which
     * they are written.
     */
    void print_all_species(ostream & out) const;

    /**
     * This returns the total number of distinct species common names found in
     * the data set.
     */
    int number_of_species() const;

    /**
     * This adds the species to the TreeSpecies container. It returns a 0 if the
     *  species was already in the TreeSpecies container and a 1 if it was not.
     */
    int add_species( const string & species);

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

private:
    set<string>      treenames;
    int              tree_species_count;
};
```