Programming Project 1: Processing 2015 New York City Street Tree Census Data

1 Overview

In 2016, as part of its *Open Data* initiative, 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. This is a large dataset, with over 683,000 lines of text totaling more than 193 MB of data. Although it will fit into your computer's memory, it will make many computers behave sluggishly, depending on what you do with the data.

In this project, your program will process the data from that data set. It will allow a user to query this data set to obtain information about selected 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 program will be storing the tree data in a searchable, modifiable container class, specifically an AVL tree. In other words, the program will use an abstract tree to store representations of real trees. You will not write the main program for this assignment. I will supply the main program, but only in binary form. You will not be able to see its source code. You will have the `main.o` object file, compiled and built on the cslab machine architecture. The details follow.

The implications are significant:

1. You can only build your executable on a cslab machine or on another machine running Ubuntu 16 with the same C++ libraries.
2. You will not have to write code to parse the command line.
3. Because the main program will parse the commands, you are freed from having to do so.
4. You must write code that implements the interface files that are given to you, exactly.
5. Although you will not write the main program, you need to know the form of the inputs that it expects in order to write tests of your code.

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 several class interfaces that are provided to you as abstract classes and hence cannot be modified. This gives you experience writing code that has been specified by someone else.
- to give you practice writing code for a client (the main program) whose implementation is hidden.

3 About the Input Data Set

The NYC OpenData website for this tree census data gives everyone 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. Because of this, parsing the data is not simply looking for the commas! 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. *If you download the dataset yourself, it will have as its first row, the column labels. 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 Usage and Error Handling

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 will write an appropriate and meaningful error message onto the ***standard error stream***, after which it will exit. If a file that is specified does not exist or cannot be opened for some reason, the program will write an appropriate and meaningful error message onto the standard error stream and then exit.

Usage Examples Suppose the program is named `nyctreeviewer`. If `nyctrees.csv` is a file containing a subset of the trees from the NYC data set and `queryfile` is a file containing queries to be processed, then correct usage includes

```
$ nyctreeviewer nyctrees.csv  queryfile
```

On the other hand,

```
$ nyctreeviewer
```

and

```
$ nyctreeviewer nyctrees.csv
```

[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.

and

```
$ nyctreeviewer queryfile
```

all result in the error message

```
usage: nyctreeviewer input-file query-file
```

being displayed on the standard error stream (your terminal by default).

Fortunately for you, the main program handles parsing the command line and checking that the program is called correctly and that the files exist and that you have permission to open and read them. Assuming that both files are opened successfully, the program reads the tree data file and the command file and applies the commands in the command file to the data it read from the tree data file. The commands are described and explained in Section 6 below.

5 Division of Work

The main program does the work of reading the two files, but it depends upon on two classes that you must implement in order to store and process this data: the `Tree` class and the `TreeCollection` class. The main program invokes methods of these two classes to parse, store, query, and retrieve the data. In order for your program to run correctly, you must implement the `Tree` and `TreeCollection` classes exactly as they are specified later in this assignment specification.

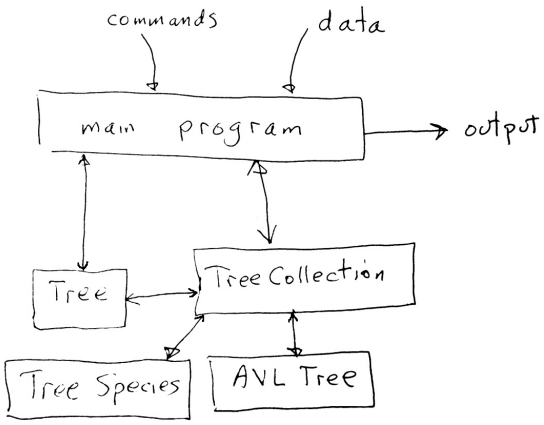

Figure 1: The architecture of the program.

Roughly, the work is divided as follows. The main program reads the *csv* file, line by line, and passes each line to a `Tree` object to parse it and construct a `Tree` from it. The `Tree` object detects whether the data is valid or not. If it is not valid, the `Tree` object that is constructed is a null tree and the main program can detect this. If the `Tree` is valid, the main program then hands this `Tree` to the `TreeCollection::add_tree()` method, which inserts it into its hidden container. The `TreeCollection` object is responsible for storing the data and for computing various properties of that data. The main program calls on the methods of the `TreeCollection` class to do most of its work. The specifications of the `Tree` class and the `TreeCollection` class and their public methods and required implementation details are contained in Sections 7.1 and 7.3 respectively. 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 this is a pretty useless way to identify a tree because it would require remembering these numeric ids. Instead, 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.

The `TreeCollection` will also keep track of the common names of all species that it stored, and in which boroughs the trees are located. To do this, it will rely on the methods of a class 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. Figure 1 illustrates the relationships between the various classes of the project.

6 Command Specification

Following is a description of the commands that the main program is able to process, with their meaning and syntax. It reads and carries out these commands from the second file specified on the command line. The `bold` text is the command name and the ***italicized*** text is its parameter list. Unless stated otherwise, all output is to the standard output stream.

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 which information is output by the command and the definition of "matches" are in Section 6.1 below.
`listall_names`	Prints all tree common names found in the `TreeCollection`, one name per line.
`print_all`	Prints the entire stored set of tree objects in sorted order, one tree per line. The tree objects are printed as a comma-separated-values file, with the ten stored fields of each tree printed one after the other on each line, in the order specified in Section 7.1 below.

`listall_inzip` ***zipcode***	where ***zipcode*** is an `int` type. Lists the common names of all trees found in the given zipcode, together with how many of each occur in that zipcode, in the form
	common_name: count
	one per line. For example, if three red oaks, four hemlocks, and two spruce are in the given zipcode its output is
	`red oak: 3`
	`hemlock: 4`
	`spruce: 2`

`list_near` ***latitude longitude dist***	where ***latitude***, ***longitude***, and ***dist*** are fixed point decimal numbers, lists the common names and frequencies of all trees within ***dist*** kilometers of the given GPS point (***latitude,longitude***). Distance is defined the *Haversine* formula, which is given in Section 6.1 below.

6.1 Matching and the `tree_info` Command

6.1.1 Matching

The argument to the `tree_info` command is a string consisting of a single word or one or more words with intervening white-space 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 determines which tree species names that were stored when the input file was read ***match*** the argument string.

As part of this project, you are given the interface file `__tree_species.h` for an abstract class named `__TreeSpecies`. This `__TreeSpecies` class contains the functionality to solve this problem. Although we define matching here, the definition of it is also contained in the documentation within the `__tree_species.h`

file.

The member function of the `__TreeSpecies` class that performs the matching is

```
virtual list<string> get_matching_species(const string & partial_name) const = 0;
```

Let us call the complete species name against which `partial_name` is compared, `tree_type`. For the purpose of matching, a hyphen character is treated like a white-space character - it separates two distinct words. Thus, for example, "`Douglas-fir`" is treated as if it were "`Douglas fir`" consists of two words, "`Douglas`" and "`fir`". Then `partial_name` ***matches*** `tree_type` if any of the following conditions are true:

- `partial_name` is exactly the same string as `tree_type`, ignoring case.
- If `partial_name` has no white-space 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 `partial_name` is exactly one of the words w_1, w_2, ..., w_k. For example if `partial_name` is "`Japanese`", and `tree_type` is "`Japanese tree lilac`", then `partial_name` matches `tree_type`.
- if `partial_name` has white-space 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 `partial_name` 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 `partial_name` does not match `tree_type`.

Positive Examples:

`oak` matches "`white oak`" and "`oak`"

`birch` matches "`paper birch`"

`paper` matches "`paper birch`"

`fir` matches "`Douglas-fir`"

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

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

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

Negative Examples:

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

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

`chest` does not match "`chestnut`"

`locust` does not match "`honeylocust`"

6.1.2 `tree_info` Command Output

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 species common names 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
```

might 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 produces 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.

6.1.3 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 decimal 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,

φ_1, φ_2: latitude of point 1 and latitude of point 2, in radians

λ_1, λ_2: longitude of point 1 and longitude of point 2, in radians

the formula is

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

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));
}
```

7 Project Organization and Structure

Your project 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.

7.1 The `Tree` Class

The interface for the `Tree` class is given to you in a file named `tree.h`. This class represents an individual tree on a public street in New York City. As stated above, the *csv* file has 41 fields, but your `Tree` class will represent trees using just a subset of them. Specifically, the `Tree` class encapsulates the following fields of the data set (listed in the order in which they appear in the file, except for `spc_common`):

Data Dictionary

- `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
- `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 latitude`; specifies GPS latitude of the tree point, in decimal degrees
- `double longitude`; specifies GPS longitude of the tree point, in decimal degrees

These are 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 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`. All methods must be case insensitive when comparing string data. Your program should not modify any of the public methods in `tree.h` 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 life_status() const;`	This returns the status of the `Tree` .
`string tree_health() const;`	This returns the health of the `Tree.`
`int id() const;`	This returns the `Tree` id value.
`string nearest_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 zip() 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 parameters the `latitude` and `longitude` of the Tree.

7.2 The `TreeSpecies` Class

You are given an abstract class named `__TreeSpecies` from which you must derive a class named `TreeSpecies`, overriding all of its virtual functions. This class encapsulates the set of all common names of trees found in the input data set. 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. An efficient implementation of this class will require no more storage than the number of distinct names in the data set. Anything else is not efficient. The class interface is defined as follows. Note that the abstract class does not have a constructor or a destructor, but your concrete class must have them.

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. They are written in lexicographic order.
`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.

7.3 The `TreeCollection` Class

You are given an abstract class named `__TreeCollection` from which you must derive a class named `TreeCollection`, overriding all of its virtual functions. The `TreeCollection` class provides the functionality to the main program for storing and accessing tree data and its properties. When you derive it from the abstract class, it must encapsulate at least two objects:

- an AVL tree that stores the tree objects that were found in the input data set, and
- a `TreeSpecies` object that stores the set of all `spc_common` tree species names that were found in the input data set.

You might find it useful to provide a third object for it, one that stores the names of each New York City borough and how many trees from the data set are in each borough.

These objects, which are essentially containers, must be private or protected members of the `TreeCollection` class. No other component of the program needs access to them.

The class must provide the following public methods. In the descriptions, whenever they refer to "matching," it is by the rules described in Section 6 above. Your subclass 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 and the program will not compile when I try to build it using the main program supplied to you. If you wish to make changes, you do so at great risk. The methods described below are in the interface file `__tree_collection.h`. and are described there in more detail. Note that the abstract class does not have a constructor or a destructor, but your concrete class must have them.

Method Syntax	**Description**
`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_tree_species_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.
`int get_counts_of_trees_by_boro ( const string & species_name, boro tree_count[5] );`	This fills the array `tree_count` of `boro` structures with the number of `Tree` objects in each borough whose `spc_common` species name matches the species_name. This method should be case insensitive. The return value is the total number of trees of given species in all boroughs.
`int add_tree( Tree & new_tree );`	This inserts `Tree new_tree` into the collection, updates the list of species names and counts of the number of trees and types of trees in the borough.
`void print_all_species(ostream & out ) const;`	This writes the set of all species common names found in the data set to the output stream `out`, one per line. The species common names are printed in lexicographic order, using the default string comparison ordering.
`void print(ostream & out ) const;`	This writes the entire data set to the output stream `out`, one tree per line, in sorted order by species common name as primary key, and then by `tree_id` as a secondary key. The members of the `Tree` object are printed in the same order as they are described in the `Tree` Data Dictionary (Section 7.1), EXCEPT THAT the species common name is printed FIRST.
`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 common names that match a given parameter string `species_name`. This method is 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 common names that are located in the given zipcode. The list returned by this function should not contain any duplicate names and may be empty.

Method Syntax	Description
`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 common names that are located within `distance` kilometers from the GPS position (`latitude, longitude`). The list returned by this function should not contain any duplicate names and may be empty.

Adding Trees to the Tree Collection 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 these four names: magnolia, mimosa, mulberry, white oak.

Finding All Matches: Performance 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 6 above. The set of matching tree names in `TreeSpecies` will be used for searching the set of stored `Tree` objects.

The `get_matching_species()` method of the `TreeCollection` must be efficient. 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.

Deleting Trees Although there are no visible commands that cause trees to be deleted form the TreeCollection, your AVL tree class must still provide a method to remove a `Tree` object, given its species common name and `tree_id`.

7.4 Required Files

I provide the abstract class interfaces and an object file for the main program. Your project must contain all of these files:

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

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.

8 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 extreme 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.

9 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.

10 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% is the maximum it can receive. It 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.

11 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on March 12, 2019. Create a directory named named *`username`*`_project1`. 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 main.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[2].

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

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

```
zip -r username_project1.zip ./username_project1
```

This will compress all of your files into the file named `username_project1.zip`. ***Do not use the tar compress utility.***

Assuming this file is in your current working directory you submit by entering the command

```
submit_cs335_hwk 3  username_project1.zip
```

because it is the third homework. The program will copy your file into the `hwk3` subdirectory

```
/data/biocs/b/student.accounts/cs335_sw/hwks/hwk3/
```

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/hwks/hwk3
```

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.