Assignment 5:
Processing New York City Open Data Using Binary Search Trees

Due date: December 7, 11:59PM EST.

1 Summary

This assignment serves a few different purposes. In a nutshell, your task is to re-implement the `HotspotList` class using a ***binary search tree*** instead of a list, and to modify the main program, whose user interface will be modified slightly. This project should enforce the following principles:

- that one can implement the same interface using completely different data structures;
- that the choice of data structure can radically affect the running time of the various methods;
- that, although some of an interface's methods are not used by one client program, another client might use them; and
- that well-designed programs with low-coupled classes and highly modular design make code reuse easier.

If your second programming project was less than perfect, and there were things that needed fixing outside of the linked list implementation of the `HotspotList` class, you should fix them before tackling this assignment. In any case, you should try to reuse as much code as possible.

The details of the `Hotspot` and `HotspotList` classes are repeated here. Things that are different from the second programming project are *italicized*.

2 Program Inputs

Your program will obtain its hotspot list data from a file whose pathname is given as the first command line argument. It will also obtain the sequence of commands it must carry out from a file given as the second command line argument. There is no other input provided. In particular, there is no user interaction. To repeat, the program must get its input from command line arguments. It does not prompt for the name of a file, it does not assume the file has some fixed name hard-coded into it. It must process its command line!

2.1 Input File Details

The program is given the pathnames of two text files as command line arguments. The first is the pathname of a file containing a data set in CSV (comma separated values) format. The second is the pathname of a file containing a list of commands that your program needs to perform. For example, if the program is named `proj4_sweiss`, and the data is stored in a file in my working directory named `hotspots.csv` and the commands are in a file named `hotspot_commands`, then I would type

```
proj4_sweiss hotspots.csv hotspot_commands
```

to run my program on the `hotspots.csv` dataset using the commands in the file `hotspot_commands`. It is an error if someone attempts to run the program without two file names. The program must check that two arguments were given on the command line and that each file can be opened for reading by the program. If there are any errors, such as missing arguments, arguments that do not exist or cannot be opened, the program must display a short but descriptive error message that indicates the nature of the error (such as "`the file hotspot_commands cannot be opened.`") It is not enough to write something like "`could not open file`." The error message should be written to the standard error stream (`cerr` in C++, `stderr` in C). Your program will not be able to distinguish whether the file does not exist or whether the user does not have permission to open it. It is enough to report that it could not be opened, for whatever reason.

2.1.1 Hotspot Files

Every hotspot file that the program reads or write is a *CSV* file. Recall from the previous assignment that a CSV file is a file in which each line represents a record of some type, and within each line, the fields or members of that record are separated by commas. Commas separate the fields; if they appear within any field, the field itself is enclosed in double-quote characters. The CSV files in this assignment may contain commas within fields, and so they must be parsed with that in mind. ***Hotspot files can not contain headers or blank lines; the number of lines in the file equals the number of records exactly.***

The data files for this assignment were obtained from a single file downloaded from the webpage
`https://goo.gl/RG6QBX`

as a CSV file. They were then edited to suit the parameters of this assignment. Each line in a hotspot file contains nine comma-separated fields. The following table gives their names, descriptions, and data types.

Field Name	Description	Value Type
`ObjectId`	Identification number automatically generated by *ArcMap* map software.	Number
`Boro`	Borough of New York City. MN = Manhattan BX = Bronx BK = Brooklyn QU = Queens SI = Staten Island	Text
`Type`	Type of WiFi provided by franchise.	Text
`Provider`	Franchise that is providing the Wifi connection.	Text
`Name`	The name of the location where the WiFi is located.	Text
`Location`	A brief description of where the WiFi point is.	Text
`Latitude`	Latitude: Points that fall north or south of the Equator, in degrees. (North is positive)	Number
`Longitude`	Longitude: Points that fall east or west of the Prime Meridian, in degrees. (East is positive)	Number
`SSID`	The name of the WiFi seen on people's devices.	Text

No white space is allowed before or after the commas that separate fields. If white space is found, this is an error. White space will be found in various text fields however. There may be double-quote characters in fields as well. If a filed contains a double-quote, it will be a sequence of two double quotes with no intervening space. To illustrate, the following are both valid records. They are wrapped because they are too long to fit across the width of this page:

```
1340,QU,Limited Free,SPECTRUM,"Phil ""Scooter"" Rizzuto Park",South side of Park,
40.694095,-73.821334,GuestWiFi

864,MN,Free,Manhattan Down Alliance,8,"182-188 Front Street, New York, NY 10038, USA",
40.7065010001,-74.0045012998,#DwtwnAllianceFreeWiFi
```

Notice the double quotes in the first line and the commas and spaces embedded in the 6^{th}field, the `LOCATION` data of the second line. Fields that contain embedded commas will always be enclosed in double quotes in a valid file. The program is required to validate the hotspot file; it may not assume that it is in the proper form.

2.1.2 Commands File

The set of commands that the application must support, together with their descriptions, is listed below. The commands are all in lower case, but their arguments can be mixed case. Words in *italic* are placeholders for user-supplied data. Any file argument that is described as a *CSV* file must be in the format described in Section 2.1.1 above. ***Files that are called `ObjectId` files are text files that contain one `ObjectId` per line, and nothing else***. Remember that an `ObjectId` is just a positive integer.

Command	Description
`save_by_id` *`fromfile tofile`*	The given *`fromfile`* is an `ObjectId` file. For each `ObjectId` in the *`fromfile`*, this command searches through the current hotspot list for a record with that `ObjectId`. If there is one, it writes it to the *`tofile`*, which is a hotspot (CSV) file. If not, no action is taken. If `tofile` already exists, it is overwritten, and if not, it is created. If it cannot be opened for writing, an error message must be written to the standard error stream. If *`fromfile`* cannot be opened, an error message must be written to the standard error stream.
`save_by_loc` *`lat lon dist tofile`*	This finds all records in the current hotspot list that are within *`dist`* kilometers of the point defined by latitude *`lat`* and longitude *`lon`* in degrees. For each such record, it writes its `ObjectId` to the given `tofile`, which must be an `ObjectId` file. The formula for the distance in kilometers between two points on a sphere (our planet) is given below. If `tofile` already exists, it is overwritten, and if not, it is created. If it cannot be opened for writing, an error message must be written to the standard error stream.
`save_by_boro` *`borocode tofile`*	This finds all records in the current hotspot list that are in the given borough, where a two-character borocode is used as the argument (as defined in the table above). For each such record, it writes its `ObjectId` to the given `tofile`, which must be an `ObjectId` file. If `tofile` already exists, it is overwritten, and if not, it is created. If it cannot be opened for writing, an error message must be written to the standard error stream.
`insert` *`fromfile`*	This inserts the contents of the given *`fromfile`*, which must be a valid hotspot (CSV) file, into the current hotspot list. The *`fromfile`* is not in any particular order. If *`fromfile`* cannot be opened, an error message must be written to the standard error stream.
`delete_by_id` *`fromfile`*	This deletes all records from the current hotspot list whose `ObjectIds` are contained in the *`fromfile`*, which is an `ObjectId` file. If an `ObjectId` contained in *`fromfile`* is not the `ObjectId` of any record in the hotspot list, no action is taken for it. If *`fromfile`* cannot be opened, an error message must be written to the standard error stream.
`write` *`tofile`*	This writes all records in the current hotspot list, in order of increasing `ObjectId`, to *`tofile`* as a CSV file. If `tofile` already exists, it is overwritten, and if not, it is created. If it cannot be opened for writing, an error message must be written to the standard error stream.

2.1.3 Performance Considerations

You will be graded on performance of your implementation. Think about some of these commands. For some, should you repeatedly search for each item in a list, over and over, or would a tree traversal be more

efficient? Does it depend on how big the list is? How would your BST implementation make various solutions easier or harder? There are good and not-so-good solutions to the implementation of these commands.

2.1.4 Examples of Valid Commands

A sequence of commands that would collect the records of hotspots within 0.5 kilometers of the Columbus Circle entrance of Central Park would be

```
save_by_loc 40.767985 -73.981285 0.5 cp_hotspot_ids
save_by_id cp_hotspot_ids cp_hotspots
```

A command that would then delete these records from the hotspot file would be

```
delete_by_id cp_hotspot_ids
```

and write the changed file to a new hotspot file:

```
write hotspots_no_cp
```

2.1.5 Parsing the Command File

I have written a `Command` class that contains methods to open and parse the command file. The class interface file and the object file will be posted in the appropriate directory on the server. The class interface is included at the end of this assignment specification as an appendix.

2.2 Distance Between Two Points on Sphere

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,

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>
# Link to 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));
}
```

3 The Hotspot Class

A hotspot record must be represented as an instance of a `Hotspot` class. This class must encapsulate all of the fields described in Section 2.1.1 above, and must have a public interface with at least the methods contained in Listing 2 below. These methods are not described in as much detail as you must provide in your interface file.

Listing 2: Minimal Hotspot Class Public Interface

```
class Hotspot
{
public:
    // Default constructor; fills fields with zeros or null strings
    Hotspot();

    // Constructor that creates Hotspot object from a hotspot file text line
    Hotspot(const string);

    // Copy constructor
    Hotspot(const Hotspot &);

    // Constructor to create a Hotspot object from nine separate data values
    Hotspot(int, string, string, string, string,
            string, double, double, string);

    // Sets all nine data members
    void set(int, string, string, string, string,
             string, double, double, string);

    // Gets all nine data members
    void get(int&, string&, string&, string&,
             string&, string&, double&, double&, string&);

    /** print() - prints the hotspot data onto the given ostream
     * @param ostream s   [inout]  ostream opened for  writing
     * @pre   the object has valid data
     * @post  if the object has valid data, then it is written to ostream
     *        in CSV format and the ostream is updated
     */
    void print ( ostream & s );
```

```
    /** Two friend comparison operators:
     * bool operator< ( lt,  rt) is true if and only if:
     *      lt.ObjectId < rt.ObjectId
     * bool operator== ( lt, rt) is true if and only if:
     *      lt.ObjectId == rt.ObjectId
     */
    friend  bool operator<  (const Hotspot & lt, const Hotspot & rt );
    friend  bool operator== (const Hotspot & lt, const Hotspot & rt );
};
```

If you are wondering what else might belong in this interface, consider methods to compute the distance between two `Hotspots`, or between a `Hotspot` and another point given as a (*latitude,longitude*) in decimal degrees, or that return just the `ObjectId` of a `Hotspot`. You might choose to implement your own assignment operator (`operator=`) as well. These are design considerations. Lastly, the private data that this class must encapsulate should be fairly obvious.

4 The HotspotList Class

The program must include a `HotspotList` class, which encapsulates the data and methods of an easily searchable list of `Hotspot` objects. *The underlying implementation must be a binary search tree whose nodes contain `Hotspot` objects, ordered by `ObjectId`.* You are relatively free to design your own interface, but you are constrained in that it must support *at the very least the following methods.* They are described informally below. It is up to you to refine them and create a suitable interface. But you must create a binary search tree to implement these methods. Asymptotic worse case running times are provided, with n being the number of nodes in the list and h the height of the tree. Your algorithms must run within these running times.

Operation and Arguments	**Description**	**Worst Case Running Time**
`int write(ostream & )`	Given an `ostream` that has been opened for writing, the hotspot list is written to the `ostream` in order of increasing `ObjectId`. This operation should return the number of records written.	$O(n)$
`int insert(const Hotspot & )`	A single record is added to the hotspot list, preserving the search tree order, provided that its `ObjectId` is not the same as that of an existing record. This should return 1 if successful and 0 if not.	$O(h)$
`int insert(const HotspotList & hlist)`	If `hlist` is in sorted order of `ObjectIds`, then this inserts every `Hotspot` record from the list into the existing `Hotspot` list, provided that none are exact duplicates. Those that are duplicates are not inserted. This should return the number of records successfully inserted into the list.	$O(mh)$ where m is the size of the `hlist`.
`int remove(ObjectId)`	This removes the `Hotspot` object from the hotspot list whose `ObjectId` matches the given `ObjectId`. If none match, it has no effect. This returns the number of records deleted, i.e., 1 or 0.	$O(h)$
`int size()`	This returns the number of objects in the list.	$O(1)$

Operation and Arguments	**Description**	**Worst Case Running Time**
`int find(ObjectId, Hotspot & )`	Searches the hotspot list for the record whose `ObjectId` matches the given `ObjectId`. If it finds one, it is copied into the supplied parameter. If not, the parameter is unchanged. It returns 1 if it found the record and 0 if it did not.	$O(h)$
`int make_empty()`	This deletes all of the records in the list and returns the number of records deleted.	$O(n)$

5 The Binary Search Tree Template Class

The binary search tree must be implemented as a template class. It must contain at the very least, methods to:

- print the tree contents in sorted order onto a given output stream
- insert a single object
- delete a single object
- find a given object and return a copy of it
- find the smallest object in the tree return a copy of it
- make the entire tree empty

For example, the following is a possible abstract interface:

```
template <typename T>
class BST
{
public:
    BST  ( );                  // default
    BST  ( const BST & tree);  // copy constructor
    ~BST ( );                  // destructor

    // Search methods:
    virtual T find    ( const T& x) const = 0;
    virtual T findMin (  ) const = 0;

    // Displaying the tree contents:
    virtual void   print   ( ostream& out ) const = 0;

    // Tree modifiers:
    virtual void   clear() = 0;              // empty the tree
    virtual void   insert( const T& x) = 0;  // insert element x
    virtual void   remove( const T& x) = 0;  // remove element x
};
```

6 Project Organization

Your project must contain, at the very least, the following files, *named exactly as follows*, case-sensitively:

`hotspot.h` The `Hotspot` class interface, with at least the public methods specified in Listing 2 above.

`hotspot.cpp` The implementation of the `Hotspot` class.

`hotspotlist.h` The `HotspotList` class interface, with at least the set of public methods specified in Section 4. You are free to add more, but use your judgment in deciding whether to add public or private methods.

`hotspotlist.cpp` The implementation of the `HotspotList` class.

`bst.h` The binary search tree template class interface

`bst.cpp` The binary search tree template class implementation file

`command.h` The `Command` class interface file that I provide for you.

`command.o` The object file for the `Command` class implementation file.

`main.cpp` This file should contain the main program and all functions that it uses other than the member functions of the above classes.

You must create a separate template class for a binary search tree, and then implement the `HotspotList` class by embedding an instance of that class as a private member. The template class must work for nodes with data of any underlying type that has comparison methods.

7 Grading Rubric

The program will be graded on a 100 point scale based on the following rubric.

- A program that fails to build, i.e., to compile and link on a `cslab` host loses 80%. The remaining 20% will be assessed using the rest of the rubric below.
- Meeting the functional requirements of the assignment: 50%
- Meeting the performance requirements of the assignment: 10%
- Design (modularity and organization) 15%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that cannot be built on a lab computer cannot receive more than 20 points. The performance component of 10% applies to the performance of all aspects of the program, but with emphasis on the binary tree methods. As a reminder, you must include proper documentation, including build instructions. ***If I cannot build it because there are no instructions, it will lose 80 points.***

8 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on December 7, 2017. There is a directory in the CSci Department network whose full path name is

`/data/biocs/b/student.accounts/cs235_sw/projects/project4`.

That is where your submission will go. In order to put it there you must follow these steps. First you will create a zip file containing your source code, including the header file `command.h` and the object file `command.o`. To do this, create a directory named `proj4_`*`username`* where username is replaced by your login name, and put all of your files into that directory. Do not place anything else into this directory. ***You will lose 1 point for each file that does not belong there***. With all files in your directory, change directory so that `proj4_`*`username`* is ***within*** the current directory and run the command

```
zip -r proj2_username.zip ./proj4_username
```

This will compress the directory and all of the files within that directory into the file named `proj4_`*`username`*`.zip`. You must use the `-r` option so that when this is unzipped, all files will be contained in a directory named `proj4_`*`username`*. You will lose 2 points if the `unzip` command does not create this directory. Then you will use the program `submit235project` to deposit the zip file into the submission directory. The program requires two arguments: the number of the assignment and the pathname of your zip file. For example, if your username on our system is `Bugs.Bunny` and your zip file is named `proj4_Bugs.Bunny.zip` and it is in your current working directory (e.g., your home directory) then you would type

```
/data/biocs/b/student.accounts/cs235_sw/bin/submit235project 4 proj4_Bugs.Bunny.zip
```

The program will create the file

`/data/biocs/b/student.accounts/cs235_sw/projects/project4/proj4_Bugs.Bunny.zip`.

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

```
ls -l /data/biocs/b/student.accounts/cs235_sw/projects/project4
```

and making sure you see your file and that its size is the same as the size of the original `proj4_Bugs.Bunny.zip`.

If you decide to make any changes and resubmit, just run the command again and it will replace the old file with the new one. Do not try to put your file into this directory in any other way - you will be unable to do this.

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.

9 Command Class Interface

```
#ifndef __COMMAND_H__
#define __COMMAND_H__

#include <iostream>
using namespace std;

/*****************************************************************************
                                Exported Types
*****************************************************************************/
```

```
/** Command_type:
    An enumerated type to represent the different types of commands. This is
    more efficient than storing command types as strings. Notice that the last
    value of the type is num_Command_types, which is simply a count of how many
    values the type contains. It is a useful method of counting, because as long
    as you insert new values before it, it remains valid.
*/
typedef enum
{
    write_cmmd = 0,
    save_by_id_cmmd,
    save_by_loc_cmmd,
    save_by_boro_cmmd,
    insert_cmmd,
    delete_by_id_cmmd,
    bad_cmmd,
    null_cmmd,
    num_Command_types
} Command_type;

typedef enum
{
    MN = 0,
    BX,
    BK,
    QU,
    SI,
    BAD_BORO,
    num_Boros
} Boro_type;

/*****************************************************************************
                              Command Class Interface
*****************************************************************************/

class Command
{
public:
    /** Command()  A default constructor for the class
     */
    Command ();

    /** get_next(istream & in) resets the values of the command object on
     * which it is called to the values found at the current read pointer of
     * the istream in, provided in.eof() is false.
     * @param istream in [inout]  an istream already opened for reading
     * @pre   istream in is open for reading and in.eof() is false
     * @post  If in.eof() is false on entry to this constructor, then
     *        the command is re-initialized to the values found in the input
     *        stream in, and the istream pointer is advanced to the next line.
     *        If the command is invalid, then when typeof() is called on it,
     *        it will return bad_command.
     *        If in.eof() is true on entry, then the Command_type is set
```

```
     *        to null and the remaining values are undefined.
     * @return true if the command was initialized to something other than a
     *        bad_command, and false otherwise.
     */
    bool get_next (istream & in );

    /** typeof() returns the type of the Command on which it is called.
     * @pre  None
     * @post None, as this is a const method
     * @return A value of Command_type, depending on the type of the command
           object.
     */
    Command_type type_of () const;

    /** args() sets the values of its parameters to the argument values of
     * the Command object on which it is called. If the Command object is a
     * bad_command or null command then the result is set to false and the
     * remaining parameter values are undefined. Otherwise, the Command_type
     * should be one of print_cmmd, save_by_id_cmmd, save_by_loc_cmmd,
     * insert_cmmd, or delete_by_id_cmmd, and the appropriate values are
     * copied from the current values in the Command object,  meaning:
     * if write, then the tofile,
     * if insert, then the fromfile
     * if delete_by_id, then the fromfile
     * if save_by_id, then the tofile and fromfile,
     * if save_by_boro, then the borocode and the tofile
     * if save_by_loc, then the latitude, longitude, distance, and tofile
     * @pre  Command_type is initialized to a valid value
     * @post Either result == false or all members are
     *       set to the values in the object.
     */
    void  get_args (
                string    & fromfile,
                string    & tofile,
                double    & latitude,
                double    & longitude,
                double    & distance,
                Boro_type & borocode,
                bool      & result
                ) const;

private:
    Command_type type;       // The type of the Command object
    string       fromfile;
    string       tofile;
    double       latitude;
    double       longitude;
    Boro_type    borocode;
    double       distance;
};

#endif /* __COMMAND_H__ */
```