Programming Project 4: Binary Search Trees

Summary

This assignment serves a few different purposes. In a nutshell, your task is to re-implement the `ContactList` class using a binary search tree instead of a list, and to modify the main program, whose user interface will be augmented. 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 third project was less than perfect, and there were things that needed fixing outside of the linked list implementation of the `ContactList` 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 `ContactList` class are repeated below. Things that are different from the third assignment are *italicized*.

Requirements

The program must implement a ***contact list***. Specifically, when the program starts up, it looks for a file named `contactlist` in the current working directory[1]. If the program finds the file and opens it successfully, *it displays on the console a message that it opened the file and read in however many records it found*, after which it enters an interactive mode, which is described below; otherwise, it displays a message informing the user that it could not open the contact list file, after which it exits.

Contact List

A ***contact list*** consists of an unlimited number of records, each of which has three members:

- *name*, which consists in turn of two members:
  - *first name*
  - *last name*
- *telephone number*,
- *email address*.

Names are not necessarily unique; there may be multiple records with the same name. In this way multiple telephone numbers and/or email addresses can be stored for the same person. Names and email addresses are represented case-sensitively. In other words, Hunter and hunter are different strings as far as this application is concerned. First names and last names may each contain up to 32 characters, which may be letters, hyphens (-), or apostrophes ('). Telephone numbers are strings that may contain up to 16 digits. They

[1] It is not named contactlist.csv, nor contactlist.txt, nor anything else!

cannot contain anything else. The application may display them with hyphens and parentheses, but these are not stored in the record. Email addresses may be up to 127 characters, and must contain exactly one '@' character. All characters to the right of the '@' must be letters, digits, or periods. The characters to the left may be anything except a comma. The telephone number and/or email address field can be a null string (a string with nothing in it.)

Contact List File

The `contactlist` file must be in a specific format in order for the program to open it correctly. It must be a comma-separated-values file, with one record per line and individual members within the records separated by commas. A record consists of the strings that are values of the *first name, last name, telephone, email address*. For example, a record might look like

```
anthony,rocco,2121234567,rocco.cannon@blasting.com
```

No white space is allowed before or after the commas. If white space is found, this is an error. *However, the program is not required to validate the contactlist file; it may assume it is in the proper form.*

The Contact List ADT

The operations supported by the contact list ADT are stated informally below.

Operation and Arguments	**Description**
display(ostream)	The contact list is written to the given ostream in sorted order, by last name as primary key and by first name as a secondary key. If there are two records with identical primary and secondary key, the telephone number is used as a tertiary key, and if need be, the email address is the quaternary key. This operation returns the number of records written.
insert(record)	The given record is added to the contact list, provided that it is not an exact duplicate of an existing record (same first and last name, same telephone and same email address. This returns the number of records inserted. Again, the keys are last name, first name, telephone number, then email address.
insert(contact_list)	Every record in the given contact list should be inserted into the existing contact list, provided that none are exact duplicates. Those that are duplicates are not inserted; records are inserted using the same ordering relation as described for the insert operation above. This returns the number of records successfully inserted into the list.
remove(record)	The supplied record must contain at least a non-null last name and first name. If the supplied record has either a null last name or a null first name, this operation does nothing. Every record in the contact list whose non-null members match the corresponding non-null members of the given record is deleted from the list. For example, if the given record is `smith,john,2127725000,""` then all records with the name `john smith` with telephone number `2127725000` are deleted. In other words, the email address is ignored in this case. This returns the number of records deleted.
size()	This returns the number of records in the contact list.
save()	This saves a copy of the current state of the contact list by writing it to a file named `contactlist.bkp` in the current working directory, overwriting any such file if it already exists. It returns the number of records written to the `contactlist.bkp` file. If it does not have permission to create the file, or if the file cannot be written for any other reason, it returns -1.

Operation and Arguments	**Description**
find(ostream, lastname, firstname)	This searches the contact list for all records whose last and first names match the given names and displays them on the given (open) stream in ***sorted order*** using the same ordering relation as the insert method. It returns the number of records written to the stream.
find(ostream, record)	This writes all records whose non-null members match the non-null members of the given record, in the order in which they occur in the contact list, on the open stream. It returns the number of records written to the stream. If all fields of the record are null, this writes nothing on the stream and returns zero.
make_empty()	This deletes all of the records in the contact list and returns the number of records deleted.

Implementation Requirements

ContactList Class

The `ContactList` class must be implemented using a binary search tree, which is allowed to be an unbalanced tree. The ordering relation of the tree is, by increasing last name as primary key, increasing first name as secondary key, increasing telephone number as tertiary key, and increasing email address as quaternary key. No member function shall visit any node in the tree more than once. The `size()` method should be *O(1)*. The `display()`, `save`(), and `make_empty()` methods should be *O(n)*, where n is the size of the contact list. The implementations of (single record) insertions, deletions, and finds should be *O(h)*, where h is the height of the tree. The insertion of a contact list containing m records may take *O(mh)* time in the worst case.

Note. Your implementation must not use any public functions other than those specified in the ADT, and those functions must have the exact signatures specified in the ADT. You are free to add private methods to the ContactList class and are free to create auxiliary classes if you think they make the program better.

Main Program Requirements

After the contact list has been loaded into memory, the main program enters interactive mode, repeatedly displaying a prompt and waiting for the user to enter a command. ***The prompt must display a menu so that the user knows what commands are allowed***. After a command is entered and the program responds to it, the prompt is displayed again, unless the command was the `quit` command. The set of commands that the application must support is listed below. The commands are case-sensitive; they must be entered in lower case only, exactly as shown.

Command	**Description**
`list`	List the entire contact list on standard output (the terminal window) followed by the number of entries listed.
`insert`	Prompt the user to enter contact info: last name, first name, telephone number, and email address. The user should be allowed to omit the telephone number and/or the email address. It will display a simple message indicating whether the contact was inserted or not. (E.g. record inserted/record not inserted.)
`merge`	*Prompt the user for the pathname of a file containing a contact list in the same format as the contactlist file described above. If the file exists and can be opened and is in the correct format, this file's records are inserted into the existing contact list, and a message will be displayed stating how many records were inserted. If it fails for any reason, a message will be displayed that it did not succeed.*

Command	Description
`delete`	Prompt the user to enter information to delete a record. The user must supply a last name and a first name, and optionally, a telephone number and/or email address. It will search for entries that match all of the non-null data supplied by the user and delete them. It displays how many were deleted.
`clear`	*Delete all records from the contact list. It displays how many were deleted.*
`find`	Prompt the user to enter information to find a record. The user can leave any member blank. This will display on the standard output all records whose non-null members match the non-null data items entered by the user. *If all fields are null, it displays nothing.*
`save`	Save the current state of the contact list to a file named `contactlist.bkp` in the current working directory, replacing that file if it already exists.
`quit`	Terminate the application.

You are free to decide how to allow the user to enter the various pieces of data for the commands, but you must document your method well. For example, you can prompt for each item or use something like "lastname=smith firstname=john" etc. Your program is expected to ensure that only valid data is stored in the contact list. You may assume that the user enters at least one correct character, so that a name is never an empty string. The program should use the longest valid prefix of the entered text as the name. For example, all of these entered strings should be stored as the name "`john`": `john653`, `john`, `john$%^`, and `john___`. *All telephone numbers and email addresses must be validated as well.*

Project Organization

Your project must consist of three separate files, *named exactly as follows*, case-sensitively:

`contactlist.h` The class interface alone, with the exact set of public member functions specified in the ADT I provided, unmodified. If you choose to create other classes that your contact list class will use, then their interfaces should be placed into this file.

`contactlist.cpp` The implementation of the class interface. If you define any other classes in the `contactlist.h` file, then their implementations should be placed into this file.

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

Implementation 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 loses 80%. The remaining 20% will be assessed using the rest of the rubric below.
- Meeting the requirements of the assignment, including performance requirements: 60%
- Design (modularity and organization): 10%
- Design of the user interface: 5%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that does not compile on a lab computer cannot receive more than 20 points. Some implementations might be more efficient in terms of running time than others. The performance requirements stated above will be the basis for evaluating your program's running time performance.

Submitting the Project

The ADT you are to use is below. This assignment is due by the end of the day (i.e. 11:59PM, EST) on December 12, 2016. Once your program is finished, you are to create a directory named `project4_username`, where `username` is to be replaced by your username on our system. Put all source code files into that directory. Do not put executables, data files, or test files in it. If I find any, you will lose three points for each file that does not belong there. Zip up this directory using the UNIX zip command, i.e.,

```
zip -r project4_username.zip ./project4_username
```

This will compress all of your files into the file named `project4_username.zip`. Make sure that, when this file is unzipped, the files are extracted into the directory `project4_username`.

Before you submit the assignment, make sure that it compiles and runs correctly on one of the `cslab` machines. Do not enhance your program beyond this specification. Do not make it do anything except what is written above.

You are to submit the zip file by running the program `submit235project`, which requires two arguments: the number of the project and the pathname of your zip file:

```
/data/biocs/b/student.accounts/cs235_sw/bin/submit235project 4 project4_username.zip
```

where `project4_username.zip` is replaced by the name of your zip file. 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. 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.

UML Formatted ContactList ADT

```
/** \mainpage CSci 235 Fall 2016 Project 4
 *  \author Stewart Weiss
 *  \date   Nov. 22, 2016
 *
 *  This is the Contact List ADT for Project 4.
 *  In order to specify a Contact List ADT definitively, the underlying type
 *  must be defined precisely. The first part of this file contains the
 *  definition of the Contact class, which depends upon a structure called Name.
 */

The following class is used by the Contact ADT. It is not documented
because it is self-explanatory.
/*****************************************************************************/
class Name
    +Name ( )
    +first(): string
    +last():  string
    +set_first( in fname: string ): void
    +set_last( in lname: string): void
/* Data members: */
    -fname: string
    -lname: string
/*****************************************************************************/
```

The Contact class defined below is needed by the ContactList ADT. This class is not documented either because the methods are simple accessors and mutators.

```
class Contact
{
public:
    +Contact ()
    +Contact (in person: Name, in tel_num: string, in email_addr: string)
    +set (in fname: string, in lname: string, in tel_num: string,
          in email_addr: string):          void
    +get_name  (out fullname: Name&):      void
    +get_tel   (out tel_num: string&):       void
    +get_email (out email_addr: string&): void
    +set_name  (in fullname: Name):      void
    +set_tel   (in tel_num: string):       void
    +set_email (in email_addr: string):    void
private:
    -name:       Name
    -telephone: string
    -email:      string
};
/*****************************************************************************
```

CONTACTLIST CLASS PUBLIC MEMBERS

```
*****************************************************************************/
/** Constructor:
 * Creates an empty contact list.
 * @pre  None
 * @post The object is empty.
*/
+ContactList()

/*****************************************************************************/
/** Destructor
 * Deletes all memory used by the contact list.
 * @pre None
 * @post The list is empty
 * Note that this is not called by any code.
 */
+~ContactList();
/*****************************************************************************/

/*****************************************************************************/
/** display(output)
 * Outputs the contact list in sorted order by last name, with the first name
 * as the secondary key. The data is spaced on the line so that each data field
 * is  aligned with the one above. The implementation is free to choose the
 * specific field widths.
 *
 * @pre      The ostream has been opened.
 * @post     The contacts in the contact list are appended to the ostream in
```

```
 *           sorted order, by last name, and then by first name in case last
 *           names are identical. If there are two records with identical primary
 *           and secondary key, the telephone number is used as a tertiary
 *           key, and if need be the email address is the quaternary key.
 * @param [inout] ostream output The stream for outputting the contact list.
 * @returns  int The number of records written
 */
+display ( inout output: ostream&): integer
/*****************************************************************************/

/*****************************************************************************/
/** insert(record_to_insert)
 * Inserts a given record into the contact list. If the record is an exact
 * duplicate of an existing record, it will not be added.
 *
 * @pre     record_to_insert is a valid Contact. If there is not an exact copy
 *          of record_to_insert already in contact list, then record_to_insert
 *          is inserted into the list at an unspecified position.
 * @returns int The total number of contacts successfully inserted into the list.
 */
+insert  (in record_to_insert: Contact): integer
/*****************************************************************************/

/*****************************************************************************/
/** insert( contact_list)
 * Inserts all contacts in contact_list into the current contact list.
 * If any of the contacts in contact_list are duplicates of an existing contact,
 * they will not be inserted. The contacts are inserted at unspecified positions.
 *
 * @pre    A contact list consisting of only valid Contacts.
 * @post   The contact list contains all previously existing contacts plus all
 *         contacts from contact_list that are not exact copies of records
 *         already in the existing contact list.
 * @returns int The total number of contacts sucessfully inserted into the list.
 */
+insert  (in contact_list: ContactList): integer
/*****************************************************************************/

/*****************************************************************************/
/** remove(record_to_delete)
 * Removes all contacts which match the non-null fields of record_to_delete.
 * Every contact in the contact list whose members match every non-null member
 * of record_to_delete is removed from the list.
 *
 * @pre     record_to_delete is a contact containing at least a non-null last
 *          name and a non-null first name.
 * @post    The contact list will contain no contacts which match the non-null
 *          fields of record_to_delete.
 * @returns int The total number of contacts successfully removed from the list.
 */
+remove   ( in record_to_delete: Contact): integer
/*****************************************************************************/

/*****************************************************************************/
```

```
/** size()
 * Returns the total number of contacts in the contact list.
 *
 * @pre          None.
 * @post         None.
 * @returns int The total number of contacts in the contact list.
 */
+size  (): integer
/*****************************************************************************/

/*****************************************************************************/
/** save()
 * This saves a copy of the current contactlist by writing it to a file named
 * contactlist.bkp in the current working directory, overwriting any such file
 * if it already exists. It must have write permission in the working directory.
 *
 * @pre      None.
 * @post     The contactlist.bkp file in the current working directory contains
 *           the contents of the current copy of the in-memory contact list. If
 *           the file existed before, it is replaced.
 * @returns int The number of contacts written to the file, or -1 if the write
 *           was not allowed.
 */
+save  (): integer
/*****************************************************************************/

/*****************************************************************************/
/** find(output, lastname, firstname)
 * Writes onto the output stream all contacts whose last and first names match
 * the given names.
 *
 * @pre      lastname is a valid name and firstname is a valid name and output
 *           is an open iostream.
 * @post     Any contacts whose first and last names match the given first and
 *           last names are appended to the ostream.
 * @returns int The number of records written to the stream.
 */
+find (inout output: ostream&, in lname: string, in fname: string): integer
/*****************************************************************************/

/*****************************************************************************/
/** find(output, record_to_find)
 * Writes onto the output stream all contacts that match the non-null fields
 * of record_to_find.
 *
 * @pre      record_to_find is a valid contact and output is an open ostream.
 * @post     Any contacts whose non-null members match the non-null members of
 *           the record_to_find are appended to the ostream, sorted by last name,
 *           and then first name in case of ties.
 * @returns  int The number of records written to the stream.
 */
+find (inout output: ostream&, in record_to_find: Contact ): integer
/*****************************************************************************/
```

```
/*****************************************************************************/
/** make_empty()
 * Deletes all of the contacts in the contact list.
 *
 * @pre      None.
 * @post     The contact list becomes an empty list.
 * @returns int The number of records deleted.
 */
+make_empty   (): integer
/*****************************************************************************/
```