Asgn 1 – Fall 2015 World Data Storage/Lookup
CS3310 – Kaminski Binary Search Tree

********** **OVERVIEW** **********

PROJECT SUMMARY: This project builds a "data store" for all the countries of the world using an internal data structure, a binary search tree (BST). *[We'll use a better storage structure in a future assignment].* It also provides "the user" with the ability to enter a country code to look up the other data values for that PARTICULAR country, as well as to display the data values for ALL countries (in sequence by country code). Batch processing rather than interactive processing will be used to facilitate development, testing and grading.

PROJECT STRUCTURE: The project consists of 3 separate programs: `Setup`, `UserApp` and `PrettyPrint`. *[Each program has a `main` method, and is thus independently executable].*

`Setup` and `UserApp` are OOP (Object Oriented Programming) programs which both use the `DataStorage` class and the `UIoutput` class. *[These 2 classes are in 2 physically separate files and are both shared/used by the 2 OOP programs].* Other classes include: `RawData` (for `Setup`) and `UIinput` (for `UserApp`). As OOP programs, `Setup` and `UserApp` themselves (i.e., their `main`'s and any of their local private helper methods) are **JUST CONTROLLERS which call** appropriate methods (i.e., constructors, getters, setters, public methods) in `RawData/DataStorage/ UIoutput` classes (for `Setup`) and in `UIinput/DataStorage/UIoutput` classes (for `UserApp`) to carry out the necessary tasks – these 2 main's **DO NOT DO any real WORK themselves.**

`PrettyPrint`, is just a developer's utility program, and so is just a simple PP (Procedural Programming) program rather than using OOP – so it does NOT use separate classes. *[I'll get a volunteer (extra credit for payment) to write this program which I'll share for everyone to use].*

You MAY use additional classes used by the above classes (e.g., BSTnode class, BackupFile class) – but the 2 OOP programs do NOT have access to these!

7 PHYSICALLY SEPARATE CODE FILES: 3 programs (2 OOP and 1 PP), 2 shared OOP classes, 2 non-shared OOP classes.

4 DATA FILES: `RawData.csv, Backup.csv, Log.txt, TransData.txt.` Each of these data files is handled completely inside its appropriate OOP class (when the 2 OOP programs are executed, though NOT for PrettyPrint program) and the fact that these are data files (rather than some GUI- I/O or a database or interactive screen/keyboard UI or. . .) is **completely hidden** from `Setup`'s and `UserApp`'s `main`'s (and their local helper methods).

PrettyPrint, however, just opens, reads/writes and closes `Backup.csv` and `Log.txt` files itself since it's PP not OOP.

The BST: This structure is **completely handled** within DataStorage class (and perhaps a local additional BSTnode class). None of the programs themselves are aware that the storage structure used for A1 is a BST. *[DataStorage implementation will be changed to some other type of storage structure in a future assignment].*

********** **THE 3 PROGRAMS** **********

1. **Setup** (and the 3 classes it uses) creates DataStorage based on data in the RawData file. Since DataStorage is built as an **INTERNAL** storage structure, it needs to be saved to an **EXTERNAL** storage structure (the Backup file) after it's completely built (as the last step in Setup) in order for UserApp to be able to use it – it calls finishUp method in DataStorage class which handles this. Status messages are sent to displayThis method in UIoutput class, as needed. This controller program uses the "sequential file processing" design pattern (see note below).

2. **UserApp** (and the 3 classes it uses) processes the user requests (transactions) which are in TransData file (obtained by calling appropriate methods in UIinput class), sending the answers to displayThis method in UIoutput class. To get the answers for the requests, UserApp calls appropriate public methods in DataStorage class. Again, appropriate status messages are sent to displayThis method in UIoutput class, as needed.

   However, prior to any transaction processing, the actual data has to be loaded from the EXTERNAL Backup file into the INTERNAL BST. This is all handled inside DataStorage class (i.e., its 2nd overloaded constructor only used by UserApp and NOT by Setup). Also, in case any transactions were Inserts or Deletes which change the data in the BST, at the end of UserApp the altered BST has to be saved to the Backup file – which is handled by the finishUp method in DataStorage class.

3. **PrettyPrint** is a developer's utility which reads/prints the Backup file, showing it (nicely) in the Log file. NOTE: It does NOT display the internal BST, per se! The program has no idea that the data in the Backup file is actually a BST – the program just thinks it's a series of data records.

NOTE 1: Setup's main and UserApp's main (nor their private helper methods, if any) do **NOT ACTUALLY DEAL WITH** the internal BST or Backup file directly. They only:

a) declare a dataStorage object (which runs one of the overloaded constructors),
b) call appropriate DataStorage public service methods, as needed,
c) call DataStorage's public finishUp method to tidy up at the end (to, in effect, "destruct" the object) – which saves the BST to the Backup file
[You MAY use an additional local class to handle the Backup file – though it's unknown to the actual Setup and UserApp programs].

NOTE 2: Setup's main and UserApp's main are controllers which process a stream of input data items (RawData and TransData files, respectively). As such, they use the "input stream processing" design pattern (algorithm structure) – i.e.,

a) prepare the stream (i.e., open the file)
b) loop til NoMoreInput (i.e., til EOF)
{ (1) input 1 item (i.e., read 1 record from the file),
(2) process that record completely }
c) close down the stream (i.e., close the file)

Again, the 2 programs themselves do NOT DO any of these file operations. Instead, they call constructors/methods in RawData class and UIinput class to do the file handling itself.

********** **THE 4 DATA FILES** **********

RawData.csv *(I'll provide this file for the demo)*

- Input file for Setup program
- **NOTE**: EVERYTHING to do with this file is handled completely in RawData class - i.e.,
  - open file in public constructor, close file in public finishUp method, a single record read in by input1Country public method
  - record split into individual fields by private cleanup and setter methods
  - individual field data available to Setup by public getter methods
  - only a SINGLE record (and a single set of fields) is needed within the class since a NEW record can over-write the prior record's data, since the prior record is already completely handled
- .csv file = Comma-Separated-Values text file (readable by NotePad). More in class.
- [RawData class only needs getters and setters for the field used in A1]
- Record format:

NOTE: Char fields are enclosed in single quotes – **your program code REMOVES THEM**

- ~~Extra characters for SQL compatibility: INSERT INTO `Country` VALUES (~~
  - ~~NOT USED IN THIS PROJECT~~
- code - 3 capital letters [uniquely identifies a country]
- id - 1- 3 digits [uniquely identifies a country]
- name - all chars (may contain spaces or special characters) [uniquely identifies a country]
- continent - one of: Africa, Antarctica, Asia, Europe, North America, Oceania, South America
- ~~region – NOT USED IN THIS PROJECT~~
- area (physical size of the country) - a positive integer
- ~~yearOfIndep – NOT USED IN THIS PROJECT~~
- population - a positive integer or 0 [which could be a very large integer]
- lifeExpectancy - a positive float with 1 decimal place
- ~~Rest of fields – NOT USED IN THIS PROJECT~~
- ~~Extra characters for SQL compatibility:); NOT USED IN THIS PROJECT~~
- <CR><LF>

TransData.txt *(I'll provide this file for the demo)*

- Input file for UserApp program
- EVERYTHING to do with this file is done within UIinput class– see above RawData NOTE
- transCode is the 1st char in the record: `I, D, S, A` (for Insert, Delete, Select, showAll)
- sample records: `I FRA ... D CHN S USA A` (see actual file for format)

Log.txt

- Output file for Setup, UserApp and PrettyPrint
- For Setup and UserApp (since they're OOP programs) EVERYTHING to do with this file is done within UIoutput class – see above RawData NOTE
- Since this is a **SINGLE** file used by all 3 programs, consider how it needs to be opened: append mode for UserApp and PrettyPrint (or their private methods) and truncate mode for Setup.
- See below for what needs to be written to this file and the exact format

Backup.csv

- Output file from finishUp method in DataStorage class which **OVERWRITES** any previously existing file with this name (so open it in truncate mode)
- Input file for constructor (#2, called by UserApp) in DataStorage class
- File opened/closed by constructor/finishUp methods in DataStorage class, NOT in Setup or UserApp themselves.
- You MAY use a separate local class to handle this, accessible through DataStorage – which would NOT be accessible to Setup or UserApp main's themselves, per se.
- What gets written to the file:
  - ONE HeaderRecord containingthese fields (in this order): rootPtr,n,nextEmptyPtr<CR><LF>
  - ALL the nextEmptyPtr number of BST nodes (both good nodes & tombstones) with these fields (in this order): leftChPt,rightChPtr,code,id,name,continent,area,population,lifeExp<CR><LF>
- It's a .csv file with commas for separators
- Include <CR><LF> for record-separators (since these are variable-length records because there are variable-length fields)

********** **BST (in DataStorage class)** **********

MUCH MORE ON THIS IN CLASS

********** **LOG FILE FORMAT** **********

NOTES:

- *Use my **exact format/wording/spacing/alignment** as shown below ! ! !*
- *Data below is NOT necessarily accurate based on the actual RawData file. I'm just demonstrating the appropriate display format.*
- *The . . . portion of the A transaction responses and the PrettyPrint results must be filled in*
- *Should the STORAGE LOCATIONS (SUBSCRIPTS/COUNTERS) BE DISPLAYED?*
  - *NO when responding to A transaction requests since the USER does NOT care about such things*
  - *YES when showing PrettyPrint results since the DEVELOPER DOES care to know such things.*
- *Should the reassurance message be displayed when dataStorage.insert is called?*
  - *NO when Setup calls it (multiple times)*
  - *YES when UserApp calls it for an I transaction*
- *Should TOMBSTONES be displayed?*
  - *NO when responding to an A or S transaction since the USER does NOT care about such things*
  - *YES when showing PrettyPrint results since the DEVELOPER DOES care*
- *PrettyPrint must use appropriate formatters to display the data so it aligns when printed out in **Courier (fixed-width) font:***
- *3 kinds of things written to the Log file:*
  1. *Status messages - these are used to help the DEVELOPER during testing/debugging*
  2. *Transaction processing REQUESTS and RESULTS (i.e., transaction requests are echo'd, then the responses are written)*
  3. *Output from PrettyPrint*

^^^^^^^^^^^^^^^^^^^^ Status messages ^^^^^^^^^^^^^^^^^^^^

NOTE: Status messages are generated (and appear in the Log file) at the time when the even actually happens. Put the call to uiOutput.displayThis (or for PrettyPrint, next to the line which does the action) in appropriate spot in the code - that is:

- file OPENED messages generate JUST AFTER opening file
- file CLOSED messages generate JUST BEFORE closing file
- PROGRAM started messages generate AT TOP of appropriate main method
- PROGRAM finished messages generate AT BOTTOM of appropriate main method

```
-->> SETUP started
-->> SETUP finished - inserted 26 countries into DataStorage
-->> USERAPP started
-->> USERAPP finished - processed 14 transactions
-->> PRETTYPRINT started
-->> PRETTYPRINT finished

-->> OPENED RawData file
-->> CLOSED RawData file
-->> OPENED TransData file
-->> CLOSED TransData file
-->> OPENED Log file
-->> CLOSED Log file
-->> OPENED Backup file
-->> CLOSED Backup file
```

^^^^^^^^^^^^^^^^^^ Transaction requests/results ^^^^^^^^^^^^ ^^^^^^

NOTE: For S & A transactions, truncate/pad fields ,"pretty-up" numeric fields , left-justify alphabetic fields, right-justify numeric fields, … as shown below
NOTE: I've used the same format string for A transactions as for S transactions
NOTE: Check for tombstoned records – A transactions do NOT show tombstoned records, And S and D transactions check for them and treat them as invalid codes

```
S WMU >> invalid country code [visited 5 nodes]

D MEX >> OK, Mexico deleted [visited 6 nodes]

D WMU >> invalid country code [visited 5 nodes]

S CHN >> [visited 3 nodes]
CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
ZWE 204 Zimbabwe           Africa           390,757    11,669,000 37.8

I AFG,1,Afganistan,Asia,652090,22720000,45.9 >>
OK, Afganistan inserted [visited 3 nodes]

A
CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
AFG 001 Afganistan         Asia             652,090    22,720,000 45.9
. . .
ZWE 204 Zimbabwe           Africa           390,757    11,669,000 37.8
============================
```

^^^^^^^^^^^^^^^^^^^^ PrettyPrint results ^^^^^^^^^^^^^^^^^^^^

NOTE: I've used the same format string for printing the country data as above for A and S transactions (beyond the the LOC,LCH, RCH fields). YES, use actual data rather than the . . . . .
NOTE: 000 is ID means it's a TOMBSTONE
NOTE: YES, PrettyPrint SHOWS tombstoned records

```
RootPtr is 00,  N is 25,  NextEmptyPtr is 27

LOC> LCH RCH CDE ID- NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
000> 001 003 MEX 000 . . . . .
001> 005 002 CHN 123 . . . . .
. . .
025> -01 -01 AFG 001 . . . . .
026> -01 -01 CAN 015 . . . . .
============================
```