CS3310 – Kaminski - Spring 2014
Asgn 1 Project Specs

CountriesOfTheWorld App 1.0

PROJECT OVERVIEW

Functionality: This app builds and provides user-access to a lookup table for data for the countries of the world – e.g., population, life expectancy, size, etc. – with country name as primary key (PK). The countryDataTable is implemented as an internal binary search tree (BST) using array storage. Batch processing is used for the UserApp to facilitate testing and submission (i.e., TransData file for input & TheLog file for out). The initial raw data is contained in a file.

For testing, the main program will call Setup and UserApp multiple times, each time supplying the fileNameSuffix (e.g., "Sample" or "All" for RawData - "1" or "2" or… for TransData).

Programming Style & Structure Requirements:

- OOP paradigm
- modular programming:
  - 7 physically separate code files:
    the main controller, 2 procedural classes, 4 object classes
  - any method > 1 page/screen (ish) is further modularized
- self-documenting code:
  - class/method/variable names match these specs
  - method names describe WHAT it does NOT HOW it's implemented
- information hiding:
  - object data is private and is only accessible outside the class with getters/setters
  - public service methods are public, local methods are private
  - implementation details are in the body of methods, not in the header
  - objects used only in one module are declared in that module
- sequential stream processing is used for raw data and transaction data [see note below]

CODE FILES

The **CountriesOfTheWorld** program's Main is overall controller/tester/driver which calls various functions (methods) - perhaps multiple times, in various orders (to be specified later in the DemoSpecs) including these 2 functions:

1. **Setup** – builds countryDataTable from RawData using sequential stream processing
2. **UserApp** – processes TransData (using sequential stream processing) using transCode to determine which countryDataTable public service method to use from:
   - SN → SelectByName gets one country (which uses BST search, not Linear search) and shows its data (in TheLog)
   - SA → SelectAll countries **in country name order** (which uses BST inorder traversal) and show their data (in TheLog)
   - IN → Insert one new country (and all its data) in countryDataTable
   - DE → Delete one country based on name specified from the countryDataTable

Project includes 4 objects (sharable OOP classes), each in its own code file (named as shown):

1. **RawData** – handles everything to do with **`RawData.csv`** file, its records and fields, providing public getters for individual fields, as needed [only used by Setup]
2. **TransData** - handles everything to do with **`TransData.txt`** file, its records and fields, providing public getters for fields, as needed [only used by UserApp]
3. **TheLog** - handles everything to do with **`TheLog.txt`** file and its lines of text [used by Setup & UserApp]
4. **CountryDataTable** – handles everything to do with the **internal** lookup table *(this is NOT A FILE)*, implemented as a BST [used by Setup & UserApp]

SEE NOTES AT END OF SPECS REGARDING FILES, TABLES, OOP

DATA FILES

These are all ASCII text files, viewable in NotePad (see note on .csv files below).

1. `RawData.csv` – provided on course website *[Original data from MySQL website]*
2. `TransData.txt` – provided on course website (soon)
3. `TheLog.txt` – created by program

^ ^

RawData.csv - record description *(see CountryDataDefinition.txt on website)*

NOTE: Char fields are enclosed in single quotes – **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]
- `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` - 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>` [Linux people beware !!!]~~

NOTES about .csv files

- *Comma Separated Values file → variable-length fields → variable-length records*
- *.csv files are viewable in Excel or Notepad (or…) - which one is determined by your computer's default option for .csv type files (which you can change). Double-click the file to use the default program. To use the OTHER software to open it, right-click the file, select Open With... and select either Excel or Notepad.*

^ ^

TransData.txt description

One transaction per line (end with <CR><LF>), starting with 2-char transCode, for example:

`SN United States` (i.e., Select by name)
`SA` (i.e., Select all by name)

```
        IN WMU,West Mich Uni,Europe,123,4567,88.9       (i.e., Insert)
            (i.e., code,name,continent,area, population, lifeExpectancy)
        DN United Kingdom                            (i.e.,Delete by name)
```

NOTE: transCode used in a switch in UserApp to determine CountryDataTable public service method to call

^ ^

<u>TheLog.txt – description (3 kinds of entries)</u>

NOTES:

- *You **MUST** use my **exact format/wording/spacing/alignment** as shown below ! ! ! !*
- *File opened in truncate mode to overwrite any previously existing versions of this file.*
- *Data shown below are not necessarily accurate based on the actual RawData files. I'm just showing you the appropriate display format.*
- *The . . . portion of the SA transaction response and the Snapshot is filled in, of course.*
- *Transaction responses do NOT show SUB or LCh & RCh pointersNOR TOMBSTONES vs. Snapshot DOES display*
    - *SUB & LCh & RCh*
    - *The header data: N & NextEmpty & RootPtr*
    - *"empty" TOMBSTONED nodes*
- *Use appropriate formatters to display the data so it aligns:*
    - *RIGHT-justify numeric fields with embedded commas*
    - *LEFT-justify char fields and truncated/right-padded as follows:*
      code: 3 columns, name 18 columns, continent: 13 columns

<u>1 - TheLog - **Status messages**</u> appear AT THE APPROPRIATE TIMES

NOTE: you MUST place code appropriately, that is:

- FILE OPENED messages generate in the line of code JUST AFTER opening the file (in the constructor)
- FILE CLOSED messages generate in the line of code JUST BEFORE closing the file (in the FinishUp method)
- CODE STARTED messages generate AT THE TOP of the appropriate method
- CODE FINISHED messages generate AT THE BOTTOM of the appropriate method

```
CODE STATUS > Setup started
CODE STATUS > Setup finished - 25 countries processed
CODE STATUS > UserApp started
CODE STATUS > UserApp finished - 14 transactions processed
CODE STATUS > Snapshot started
CODE STATUS > Snapshot finished - 25 nodes displayed
FILE STATUS > RawData FILE opened
FILE STATUS > RawData FILE closed
FILE STATUS > TransData FILE opened
FILE STATUS > TransData FILE closed
FILE STATUS > TheLog FILE opened
FILE STATUS > TheLog FILE closed
```

<u>2 - TheLog - **Transaction processing**</u> (transaction request echoed before data is shown)

```
SN China
   CHN China              Asia            9,572,900 1,277,558,000 71.4
      >> 2 nodes visited
SN CHINA
   CHN China              Asia            9,572,900 1,277,558,000 71.4
      >> 2 nodes visited
SN United States of America
   SORRY, invalid country name
      >> 6 nodes visited
SN United
   SORRY, invalid country name
      >> 5 nodes visited
DN Belgium
   OK, country deleted
DN Western Michigan
   SORRY, invalid country name
IN GBR,United Kingdom,12345,98765,75.6
   OK, country inserted
      >> 7 nodes visited
SA
   CDE NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE
   CHN China              Asia            9,572,900 1,277,558,000 71.4
   . . .
   ZWE Zimbabwe           Africa            390,757    11,669,000 37.8
   ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

<u>3 - TheLog – **Snapshot utility**</u> results look like this:

```
N: 25, NextEmpty: 26, RootPtr: 000

[SUB] CDE NAME-------------- CONTINENT---- ------AREA ---POPULATION LIFE LCh RCh
[000] USA United States      North America  9,363,520   278,357,000 77.1 001 004
[001] CHN China              Asia            9,572,900 1,277,558,000 71.4 003 002
[002] ZWE Zimbabwe           Africa            390,757    11,669,000 37.8 006 005
[003] TOMBSTONE
. . .
[024] RUS Russian Federation Europe        17,075,400   146,934,000 67.2 -01 -01
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

CountryDataTable

- BST's and their algorithms will be discussed in class & see readings on course website
- Static delete will be used (i.e., using TOMBSTONES) rather than the normal dynamic BST delete algorithm.
- This is internal table. It's built entirely in memory in Setup, and stored in memory all during UserApp's run.
- This uses **<u>array storage</u>** for nodes.
  [More on this in class. CAUTION: This is not the conventional way to store BST's (found in DataStructure books or online). Nor is this the conventional Array Storage for binary trees as used for heaps which uses implicit pointers.]
- This uses <u>explicit "pointers"</u> (i.e., array subscripts) rather than C-style pointers or Java/C# references.
- **IMPORTANT: Use -1 for "points nowhere". 0 won't work since that's a valid storage location in arrays.**

<u>Space management</u>

- Manual space management: nextEmpty is initialized (in the constructor?) and incremented by the program in Insert.

- Static space management: Deleted nodes are not removed from the BST, merely TOMBSTONED (i.e., marked as deleted). Insert does NOT reuse tombstoned locations – it always uses the nextEmpty location.

Header data

A BST data structure includes additional fields besides the node storage *(just as a stack needs a topPtr, a linked list needs a headPtr, . . .).*

  - **`rootPtr`** (an array subscript which will start out at 0 and stay there because of using static delete).
  - **`n`** is the number of nodes with good data
    – Insert increments it, Delete decrements it.
  - nextEmpty (an array subscript – Insert increments it, Delete does NOT decrement it. nextEmpty and n are the same during Setup – but after any successful delete during UserApp, nextEmpty will be > n)

A bstNode (a separate BstNode class) contains:

The Key:	name	(the PK for comparisons)
The Data :	code	
	continent	
	area	
	population	
	lifeExpectancy	
The Child Ptrs:	leftChPtr	(subscript where left child node is stored)
	rightChPtr	(subscript where right child node is stored)

A Tombstone

The key and child ptrs must keep their current values to allow subsequent searching.
To mark the node as "deleted", put XXX in code
Spaces in continent
0's in area, population and lifeExpectancy

Notes on Comparisons

- *A match on a tombstone is not considered "a match" – keep searching!!*
- *Ignore case when comparing – so "mExICO" successfully finds "Mexico"*
- *Ignore trailing spaces when comparing – so "France   " from the TransData file matches "France" in the table*
- *Only treat full-matches as successful, so "United" must NOT match "United States"*
- *For C# use CompareOrdinal method rather than Compare or CompareTo so that special characters follow strict ASCII-order. Java's uses ASCII-order by default.*

Use proper BST algorithms:

- SelectByName uses BST search algorithm
  Doing a LINEAR search will result in losing LOTS OR POINTS
- Delete uses BST search algorithm to locate the target node
  Then static delete (rather than the normal dynamic BST delete algorithm)
  Doing a LINEAR search will result in losing LOTS OR POINTS
- SelectByName and Delete both use the same Search method
- SelectAll uses binary tree's inorder traversal
  (skipping tombstones, of course)
  Doing a SORT will result in losing LOTS OR POINTS
- Insert uses the BST insert algorithm

NOTES on OOP

OOP - Information hiding (WHAT vs. HOW)

Class NAMES and PUBLIC METHODS describe WHAT the object is and its functionality to the "outside world" (other parts of the project). The code BODY handles HOW the underlying storage works and HOW interaction will be implemented.

Users (Main, Setup, UserApp) of the object classes (RawData, TheLog, TransData, CountryDataTable), only know what the object classes' public service method names are (including getters/setters), but NOT what's inside the methods NOR what the data is. They are NOT at all aware of:

- *WHERE the RawData field values come from (A data file? Interactive users? A database? A bar-code scanner? QR code scanner on your iPhone?) nor HOW it was derived (Any transformations? Record-splitting into fields? Field editing after reading from text-boxes? Floats changed to integers? Metric changed to imperial measures? Field-values calculated or read-in from storage?)*
- *HOW the table is stored & accessed (a BST? An ordered list? A hash table?) nor whether it's an internal or external structure, or whether it's in memory or a file or a database or the cloud*
- *HOW the user interface is implemented other than*
  - *TransData comes in a transaction at a time, which might be a file, a database, data entered in a textbox in a web app on a tablet, a QR code scanned in, an interactive user typing at the console, etc.*
  - *output is sent to TheLog which might be a file, or a database or the console or the screen on a mobile device, etc.*

This makes OOP programs easier to change since all code changes are done within a specific class, with no (few) changes to the main procedural/control parts of the program code.

OOP – Public vs. Private

What's public - and thus describes WHAT's going on and what's KNOWNABLE to the "outside world" (i.e., the main program and procedural class code themselves)?

- *Class names*
- *Public service method names (including getters/setters and constructors) and their parameters*

What's private - and thus describes HOW things are stored and IMPLEMENTED and knowable ONLY to other code within this class, but NOT to the "outside world"?

- *The bodies of the public service methods*
- *Private methods - their names, parameters, code bodies*
- *data storage within the class (public getters/setters make it accessible to the outside world)*
- *the actual FILE handling:*
  - *data file name declarations*
  - *opening the file (in the constructor)*
  - *closing the file (in a public FinishUp method, named as such so the outside world doesn't know there's a file involved)*
  - *the actual reading/writing of records (and setting/checking the "EOF switch" (called DoneWithInput so the outside world won't know it's actually a file).*

Object declaration:

- *Declare an object as locally as possible – if it's only used in one procedural class, then declare it there and FinishUp with it in there.*
- *If an object is declared in an outer callING module, and a callED method needs to use it, then the object would have to be passed in as a parameter*

Data FILE classes:

- *File is opened in constructor – fileNameSuffix must be passed in as a parameter.*

- *File is closed in FinishUp method since program can't control when a deconstructor method would actually execute.*
- *Classes for input files need to handling reading from the file and EOF-checking:*
    - *inputARecord method (e.g., input1Country, input1Trans) with no mention of "read" since that sounds like the object is definitely implemented as a FILE*
    - *a boolean doneWithInput (and doneWithTrans) method with no mention of hitEOF since that sounds like the object is definitely implemented as a FILE.*
- *Classes for output files must open a file appropriately for the situation – in truncate mode or append mode.*
- *Classes for output files need a displayThis method, with the caller supplying what needs to be written out (with no mention of "write" since that sounds like the object is definitely implemented as a FILE). This method is overloaded since status messages, IN/DE reassurance messages, Error messages calls supply a single pre-formatted string, while SN/SA and Snapshot calls supply individual fields from which a string is built here with a common (ish) formatter since these 3 produce similar output lines.*
- *Actual data is provided to the caller via getters, and not directly from variable.*

<u>TABLE class:</u>
Since Setup will/may be run multiple times, a completely new table needs to be set up for the new run (e.g., initializing N and NextEmpty).

####################   SEQUENTIAL Stream PROCESSING   ###################

Setup and UserApp both do basic sequential processing of their respective input stream. The proper approach is to get a SINGLE input data set, then handle it completely – then loop to do that again until done with the input. This allows for the input stream to be coming from a file, from a database result set, an interactive user, a series of users, repeated use of a barCode scanner, etc. The basic algorithm:

```
Get stream ready (file opened in class's constructor for OOP)
Loop til nothing more arriving from stream
          (i.e., a boolean method in the class which indicates "done")
{   1.  call a method in the input stream class to input
              a SINGLE data set (record)
              which READs & splits the record/line into fields
              And does whatever other cleanup is needed)
    2.  call a method in the output handler class to
              PROCESS that SINGLE data set
              (using public GETTERs in the input class as parameters)

}
FinishUp with stream (file closed in class's FinishUp method, for OOP)
```

<u>Implementation NOTES</u>:

- *Just because the human algorithm uses a "READ/PROCESS" loop structure doesn't mean that the implementation (in a programming language) necessarily uses that structure. It MAY instead need a "PROCESS/READ (with a priming read)" loop structure – depending on which "read" method is used and what "EOF-detection" approach is used in a particular language.*
- *There is never more than a single RawData record in memory at once. Only a single object is needed for storing a RawData record. New records are stored in the same storage space, replacing the prior record since you only ever need 1 record (and its fields) at once. Similarly for TransData..*