CS3310 – Kaminski - Fall 2014 CountriesOfTheWorld App 1.0

Asgn 2 Project Specs – PART A

[Part B discusses the BST aspects - available later]
Also see "further notes" on asgn website.

OVERVIEW

A2 is a modification of A1 [Make a copy of A1 and make changes to the COPY]. If you had mistakes in A1 (whether grader caught them or not) or now find there's a better way to handle things, change things for A2 before proceeding with the new parts of A2.

Assume A2 requirements are the same as A's, unless changes are specified here or in class. Two main changes:

- dataTable is now an **external** table (file) rather than an internal table (in memory)
  which still uses a **direct address** structure on id as the key
- nameIndex is still an **internal** index
  which now uses a **binary search tree** structure on name as the key
  rather than a sequential list

WHAT NEEDS CHANGING ?

Programs

- TestDriver – minor changes (to be described in the A2DemoSpecs)
- PrettyPrintUtility – reads/prints CountryData.txt file besides Backup.txt file
  (and Backup.txt is now different, so. . .)
- Setup and UserApp - no changes

Instantiable Classes

- RawData and UI – no changes
- DataTable and NameIndex
  - no changes to public interface (i.e., headers for public service methods)
  - lots of changes to:
    - private data storage
    - implementations (the body) of public methods/constructor/…
    - private methods (header & body)

2 NEW Instantiable Classes now required

- DataRecord class and BSTNode class

Data Files

- A2RawData.csv and A2TransData?.txt – same format,
  but different fileNameSuffix & different data contents
- CountryData.txt - NEW FILE – it's the dataTable
- Backup.txt – contains headerRec & dump of nameIndex, but NOT the dataTable
  - headerRec contains different fields than A1
  - nameIndex records contain 2 additional fields than A1
- Log.txt -
  - 2 additional status messages: STATUS > CountryData FILE opened
    STATUS > CountryData FILE closed
  - PrettyPrintUtility's output looks different for headerRec & nameIndex and the LOC's for dataTable are now RRN's (not subscripts)

DataTable & DataRecord classes

Who's responsible?

- RawData class's methods are responsible for:
  - cleaning a RawData file's LINE (remove 1st 30 char's & the last 2 char's & the single quote chars)
  - splitting(on commas) the data line into individual fields
  - discarding (i.e., not storing) all but the 7 required fields
  - perhaps converting the numeric fields into int/long/double's
  - BUT, it is NOT RESPONSIBLE for the truncating/padding of the 7 fields to the specified sizes that dataRecord needs
- DataTable class does all handling of the countryData.txt FILE
- DataRecord class's methods are responsible for dealing with individual records/fields:
  - Truncating/padding the 7 fields as required for the DataTable file

There will only be a single dataRecord object since there is never more than a single data record in memory at once. DO NOT SET UP AN ARRAY OF dataRecord objects – this is an EXTERNAL STORAGE STRUCTURE.

Setup & UserApp (& RawData & UI & NameIndex) don't know that dataTable is: a FILE or that it uses a DirectAddress structure. PrettyPrintUtility program knows it's a file. TestDriver knows it's a file since it has to delete it.

Record Description

1st) ONE header Record containing n and maxId

2nd) rest of the records (except empty locations) contain (with fields in this order)

code	– 3 char string or charArray (truncated or space-filled on right)
id	– 3 digits (0-filled on left)
name	– 15 char string or charArray (truncated or space-filled on right)
continent	– 13 char string or charArray (truncated or space-filled on right)
size	– 8 digits (0-filled on left)
population	– 10 digits (0-filled on left)
lifeExp	– 4 char (3 digits plus decimal point) (0-filled on left)

[NOTE: truncating and/or zero/space-filling is done to ensure all records are of the exact same size (i.e., fixed-length records, necessary for direct address files).

OTHER NOTES:

- DeleteById is still a dummy stub
- SelectById and Insert MUST use direct address "search" –
  Linear search → lose LOTS OF POINTS
- Empty locations and empty-location-detection will be discussed in class
  - Allow for BOTH the "all-0-bits" case and the "read-failed" case
  - Do NOT initialize the fileSpace by writing "empty records" to locations 1 through ? – this shouldn't be necessary since the OS handles this when you open a file.

- Allow for duplicate Id's (in Insert1Country) – (e.g., 2 RawData records with same id, a RawData record &later an Insert transaction, 2 Insert transactions in the same or subsequent runs of UserApp)
  - Handling:
    - The 1st country with the id is stored in the file
    - Any subsequent countries with that ID are rejected, with an error message to Log file, and the country is NOT put into the CountryData file

CS3310 – Kaminski - Fall 2014 CountriesOfTheWorld App 1.0
Asgn 2 Project Specs – PART B

[Part A discusses A2 overview & DA dataTable aspects]

##########################   NameIndex OVERVIEW   ##########################

In A2, NameIndex is still an **<u>internal</u>** index (with name as the key) as it was in A1.
However, for A2:

1. It MUST be implemented as a **<u>binary search tree</u>** data structure (vs. a sequential list)
    - *[FAQ: No, you may NOT use the built-in tree structure]*

2. It MUST use an <u>array of BST nodes</u> (objects) rather than parallel arrays (if that's what you did in A1)
    - where bstNodes are objects of the BSTNode class type
    - *[FAQ: Yes, there are MANY bstNode objects (in memory at the same time) since this is an INTERNAL data structure, not an EXTERNAL one]*
    - *[FAQ: This is NOT the conventional way to implement BST's (found in Data Structure books or online). NOR is this the conventional Array Storage for binary trees used for heaps which uses implicit pointers. We're using this ARRAY of objects approach because we need to dump & load it to port it between program-runs. More on this in class.]*
    - You MUST use explicit "pointers" (i.e., array subscripts) rather than C-style points or Java/C# references.
    - You MUST use -1 for "points nowhere".
      [FAQ: NULL won't work since these "pointers" are int subscripts].
      [FAQ: 0 won't work since that's a valid storage location in arrays].

3. It uses <u>BST algorithms</u> for the code-bodies for NameIndex methods including:
    a. insertIntoNameIndex(name, drp)
    b. selectByName(name)
    c. selectAllByName()
    d. deleteByName(name) - which is still a dummy stub

    [FAQ: No, you may NOT use the built-in "magic" tree handler methods – they must be explicitly written, "manual labor" method bodies].

#############################   Other BST Issues   ##############################

<u>Node Storage</u>

A bstNode object is of BSTNode class type
(where the class is a sub-class or separate class from NameIndex class).

BSTs contain these fields:	LeftChildPtr, KeyData, RightChildPtr
Indexes contain these fields:	KeyData, DataRecordPtr
So a single bstNode contains these 4 fields:	leftChPtr, name, drp, rightChPtr

*[FAQ: Use a static array of bstNode objects of size MAX_N_LOCATIONS = 40 for now.
However, given our RawData test file, we'll only really need 25 locations,
plus a few more potentially during UserApp doing some Inserts.
But we're NOT implementing delete, so. . .]*

*[FAQ: Do NOT use 40 or MAX_N_LOCATIONS to control such things as:
dumping to the Backup file (use n instead)
loading from the Backup file (use "watch for EOF" instead)
pretty-printing (use "watch for EOF" instead)].*

<u>Header data</u>

A BST data structure needs an additional field (besides node storage): rootPtr
Manual space management needs an additional field: n
(which can be used for nextEmptyPtr since we're starting with storage location 0, not 1, and we're not implementing DELETE)
[FAQ: insertIntoNameIndex increments n]

<u>"Pointers":</u> leftChPtr, rightChPtr, rootPtr, n (when used as nextEmptyPtr)
These are actually integer subscript values, which "point to" some location in the array.
[FAQ: Yes, rootPtr starts out at 0 and stays there because we're not doing any DELETEs].

<u>Space management (of array storage)</u>

- Use manual "static space management" using n (which is really nextEmptyPtr)
    - n is initialized (in the constructor?)
    - n is incremented in insertIntoNameIndex
- We're not implementing deleteByName, so we don't have to worry about returning a deleted node to the available node pool. So n is always the next empty location

<u>Use proper BST algorithms</u>

[FAQ: You don't have to use "MY" algorithms, specifically. You may use iterative or recursive algorithms. But you MUST tailor ANY algorithms to deal with MY SPECIFIC REQUIREMENTS FOR HOW THE BST IS IMPLEMENTED].

- selectByName uses the BST search algorithm
  Doing a LINEAR search will result in TAKING LOTS OR POINTS OFF
- deleteByName is still a dummy stub (i.e., there's a proper callable method interface)
- selectAllByName uses binary tree's inorder traversal
  Doing any kind of SORT will result in TAKING LOTS OR POINTS OFF
  [FAQ: Note that a BST is <u>in logical sequence</u> (i.e., logically sorted), but it's <u>NOT in physical sequence</u> (i.e., physically sorted). So if you looked at the array (or the Backup file), you would NOT see the names in alphabetical order. HOWEVER, if you use the <u>BT InOrder Traversal algorithm</u>, then you can visit the nodes in alphabetical order, and so use the DRPs in the order needed to print out the DATA in country-name order]
- insertIntoNameIndex uses the BST insert algorithm