CS3310 – Kaminski - Fall 2012 World Data App 1.1
Asgn 2 Project Specs *add a Code Index part (as a BST)*

OVERVIEW

Asgn 2 adds a Country Code Index to Asgn1 project to allow QueryByCode (QC), DeleteByCode (DC), ListAllByCode (LC) and InsertCode (IN) functionality. The index uses internal storage using a binary search tree (BST) data structure. An array-based implementation is used for the BST – as demonstrated in class.

Since the index uses internal storage (memory) when it's first constructed (during Setup) and during the actual transaction processing (during UserApp), the internal index has to be saved and loaded to external storage (a file) between runs of the programs (and between subsequent re-runs of UserApp program). Between run's it's stored on a serial, binary file, **`?IndexBackup.bin`** (where ? is the fileNamePrefix).

All handling of the index is done inside the ***CodeIndex*** class, shared by Setup and UserApp. ShowFilesUtility is not an OO program, so the IndexBackup file is directly accessed by the program itself.

None of the programs themselves know that the index is implemented as a BST – that's all hidden within the CodeIndex class. Initialization of the index is done in the constructor, which Setup uses. UserApp needs a different constructor which loads IndexBackup file copy of the index back into memory again [More on this below]. UserApp will need public methods to InsertCode, QueryByCode, DeleteByCode, ListAllByCode. *[Notice that I named these classes based on **WHAT** their functionality is to the outside program/caller which uses them. The names don't mention BST storage or algorithms. Inside the class, the storage and method code itself knows **HOW** the index is implemented - it's a BST and uses the BSTSearch, BSTInsert, BSTInOrderTraversal. . . algorithms. This is the goal of "information hiding" and maintainability. For example, a future version of the project could require changing the CodeIndex to use a hash table instead of a BST, without Setup and UserApp caring at all about such details].*

FinishUp method will automatically save the internal index (the BST) to the IndexBackup file. [More on this below].

Setup "controller"

Setup program still uses the Sequential Processing algorithm, as in A1. However, the "process" step of the read/process algorithm must now include:

```
1) Write1Country(a DataStorage method)
       - returns the RRN where the record was actually stored
              (= id for A1/A2, but not for future asgn's, so…)
              OR 0 for error (duplicates not stored)
2) If data record was actually stored then call InsertCode
       (a CodeIndex method), sending in code & RRN (as DRP)
```

UserApp "controller"

UserApp program still uses the Sequential Processing algorithm, but with the big switch statement expanded to accommodate the new transaction types (QC, DC, LC). Also, an IN transCode now requires that TWO calls are made, to Write1Country (in DataStorage class) and InsertCode (in CodeIndex class).

LogSession FILE

ADDITIONAL Status messages (besides what was required in asgn 1)

```
**** Saving CodeIndex to ?IndexBackup FILE
**** Loading CodeIndex from ?IndexBackup FILE
```

NOTE: DEVELOPER INFO ALSO SHOWN – i.e., the # of nodes visited in the search

```
QC FRA
   003 FRA France           Europe      Western Eu    551,500  0843    59,225,700 78.8
  >>> 4 NODES VISITED
QC WMU
  ERROR – no country with that code
  >>> 7 NODES VISITED
IN . . .
  OK, country inserted in DataStorage
  OK, code inserted in CodeIndex
  >>> 4 NODES VISITED
IN . . .
  ERROR – duplicate id for Germany          (not inserted) – id 3 is France
DC FRA
  OK, country deleted – France
  >>> 4 NODES VISITED
DC WMU
  ERROR – no country with that code
  >>> 7 NODES VISITED
LC
ID CODE NAME              CONTINENT   REGION        AREA    INDEP    POPULATION L.EXP
039 ATA . . . . . . . . . . . . . . . . . . . . . (yes, the rest of the fields print)
027 BEL . . . . . . . . . . . . . . . . . . . . . (yes, the rest of the fields print)
. . .
025 ZWE . . . . . . . . . . . . . . . . . . . . . (yes, the rest of the fields print)
+ + + + + + + + + + THE END OF DATA + + + + + + + + + +
```

ShowFilesUtility's results look like this (with the . . . part fully filled in, of course):

```
MAIN DATA STORAGE – N is 26, MaxID is 39
[RRN] ID CODE NAME              CONTINENT   REGION        AREA    INDEP    POPULATION L.EXP
[001] 001 KEN Kenya             Africa      Eastern Af    580,367  1963    30,080,000 48.0
[002] EMPTY
[003] 003 FRA France            Europe      Western Eu    551,500  0843    59,225,700 78.8
[004] EMPTY
[005] EMPTY
[006] 006 ZWE Zimbabwe          Africa      Eastern Af    390,757  1980    11,669,000 37.8
. . .
* * * * * * * * * * THE END OF DATA * * * * * * * * * * * * * * * * * * * * *

THE CODE INDEX – N is 26, RootPtr is 0
SUBS   LCH  KEY  DRP  RCH
[000]  001  MEX  012  003
```

```
[001]  005  CHN  003  002
. . .
[025]  -01  ATA  039  -01
```

################ ?IndexBackup.bin FILE DESCRIPTION ###################

A BINARY file (no field-separators, no <CR><LF>'s) with fixed-length records.

3 parts to the file (in the following order)

1) **One header record** containing: `n, rootPtr` (other fields added in Asgn3)
   - Both short int's
2) **dump of CodeIndex (i.e, n BST nodes)**
   - **NOT as a BST, per se, but as an array from 0 to n-1**
   - Each record contains 1 BST node with these fields (in this order):
     `leftChPtr, code, dataRecPtr, rightChPtr`
     where the 3 Ptr's are short int's & code is a 3-byte char array (not a string)
3) **NOT USED IN ASGN 2** (WILL BE USED IN ASGN 3)

########################## THE INDEX (stored as a BST) ###################

- BST's and their algorithms will be discussed in class. See readings on course website.
- This is INTERNAL index. That means it's built entirely IN MEMORY during Setup's run) and stored IN MEMORY all during UserApp's run.
- It is only stored on a file as a way to PORT it from Setup program to UserApp program (so it doesn't "die" when either program stops executing). It is NOT considered as an EXTERNAL index because it is NOT being PROCESSED from the FILE itself.
- This uses **array storage** for node storage. *[More on this in class. This is not the conventional way to store BST's. Nor is this the conventional Array Storage for binary trees like heaps. This uses EXPLICIT pointers, not implicit ones. Don't just use what's described in a/the book or on the internet) for "Binary Tree storage using an array" !!!]*
- The internal storage structure uses array storage with explicit "pointers" (i.e., array subscripts) rather than C-style pointers or C# references.
- **Use -1 for "points nowhere". 0 won't work since that's a valid storage location in arrays.**
- A BST data structures includes additional fields besides what's needed for node storage (just as a stack needs a topPtr, a linked list needs a headPtr, . . .).
  - A BST needs a `rootPtr`.
  - And because YOU'RE doing the space management for this (because we're using array storage for nodes), we also need N. [So, since arrays start with 0, not 1, N indicates the nextEmpty location in the array]. N is a counter which needs initializing and incrementing at the appropriate times.

########################## Save/Load Index to/from ?IndexBackup File #########

FinishUp method in CodeIndex class saves the internal index to the external backup file. The constructor used by UserApp needs to load the external backup file data back into the internal index structure in memory.

However, these procedures do NOT treat the data (in the file or in the internal array) as a BST, per se. Because of the way the BST is implemented, the internal storage structure is treated as JUST AN ARRAY in the saving/loading. Since N is available, a simple for loop suffices for control.

########################## NOTES ON COMPARISONS ######################

Comparing CODE fields

These are stored as 3-char char arrays rather than strings. However, most programming languages have built-in string comparison methods. So you'll want to use the string-comparison methods after converting both operands from char-arrays to strings. Do NOT manually compare 2 code fields char-by-char.

Comparing for LessThan (or GreaterThan) for the BST

C#'s String Compare method (and CompareTo) do not use the ASCII-order for comparisons when dealing with "special characters" (+, ', %, -, etc.), although they work correctly for capital letters. Because of the data being used here (i.e., all letters and all caps), you won't notice a problem. However, for a more robust program (which would work correctly for BST's dealing with name fields, for example, (which might include O'Leary or Smith-Jones), when comparing char fields in general, consider whether ALL modules (Insert, Search) consistently use ***Compare*** *(and/or CompareTo) vs.* ***CompareOrdinal.*** *It's important that the same data value ordering is used when building the tree AND when searching the tree.* ***FOR THIS ASGN, USE THE CompareOrdinal METHOD*** *(in C#) which follows the ASCII-code order. (Java must have a similar comparison method which uses the ASCII-code order – use that).*

########################## CAUTIONS ##################################

QC Processing

The BST is stored in an array, so linear search is possible for QC transactions – but **DON'T USE LINEAR SEARCH. YOU'LL LOSE LOTS OF POINTS IF YOU DON'T DO A PROPER BST SEARCH.**

LC Processing

ListAllByCode **MUST use InOrderTraversal algorithm. YOU'LL LOSE LOTS OF POINTS IF YOU DO A SORT.**

########################## OTHER NOTES ###############################

Assume No Duplicate Code values in the RawData files.