CS3310 – Kaminski - Spring 2016 external CodeIndex (B Tree) (.csv)
World Data App Index Tester external CountryData (.txt)

################################ OVERVIEW ##################################

A5 is only part of a larger project (building on A1-A3) which has yet to be done. But YOUR TASK for A5 is **ONLY** to do:

- UserApp, not Setup nor PrettyPrint
- queryByCode (QC) functionality, which uses codeIndex
  - the index is an external BTree
  - the index MUST be used to access the CountryData file by code
  - *NOTE: Since an INDEX is used (i.e., there's a DRP for each code (the KV (KeyValue), the project does NOT care what type of file structure (nor what key) is used for the main data storage's organization – as long as that physical file allows random access.*
- CodeIndex class contains queryByCode method & all handling of the index
  - *the index is an EXTERNAL structure (BTree file) and must be handled as such – it MUST NOT be all read into memory and searched as an INTERNAL structure*
- CountryData class – modifications from A1-A3 must be done to accommodate the format of the test data for A5

DATA FILES FOR TESTING: Because Setup hasn't been completed yet, the data files UserApp needs for testing (CodeIndex, CountryData, TransData) must somehow be created. The "test support team" (me) has done this manually with a text editor, using artificial data. Once Setup's done in the future (NOT THIS SEMESTER), only minor changes will be needed in a couple modules in this project. (SEE MORE BELOW)

########################## PROGRAM STRUCTURE ###########################

Code blocks needed (each in its own physical file):

1. UserApp PROGRAM:
   - contains `main` and perhaps other private method,s if `main` becomes > 1 page/screen-ish
   - overall controller including a switch on transCode (but QC is only case for A5)
   - program asks interactive user for NUMBER for whichTestSet (i.e., the fileNameSuffix for CodeIndex file, CountryData file & TransDataA5 file

2. **OPTIONAL**: UI OOP class (as done in prior asgn's): handles EVERYTHING to do with
   - The input: TransDataA5_?.csv
   - The output: Log.txt

   OR ALTERNATIVE: UserApp program itself handles both of these files directly

3. CodeIndex (OOP) class: handles EVERYTHING to do with CodeIndex_?.csv file/records
   [Optional: Perhaps also a separate BTreeNode class ???].

4. CountryData (OOP) class: handles EVERYTHING to do with CountryData_?.txt
   - NOTE: if you're re-using A1-A3, remove any old functionality, e.g. methods and data storage which dealt with QI, LI, DI, QN, LN, DN, IN, etc.

NOTE: "EVERYTHING" with respect to a FILE means:
Opening file in public constructor (fileNameSuffix is passed in as a parameter)
Closing file in public finishUp method
Reading a single record/node/line in a method (for input files) – clearly named:
read1Node, readHeaderRec, readDataRecord
EOF detection and setting public noMoreInput switch on (for input files)
Writing a single record/node/line method (for output files)

NOTE: whichTestSet will be a small integer indicating what the '?' is for
CodeIndex_?.csv file, for CountryData_?.txt file, for TransDataA5_?.csv file

############################### DATA FILES ################################

TransDataA5_?.csv (INPUT) - several versions of this file for testing - SEE BELOW

- example of a record: `QC,USA` (followed by <CR><LF>)

CodeIndex_?.csv (INPUT) - several versions of this file for testing – SEE BELOW

- ASCII text file (created with NotePad) with <CR><LF> after each record
- csv records for headerRec and bTreeNodeRec's
- FIXED-LENGTH records (for bTreeNodeRec's) to allow RANDOM ACCESS,
  (and headerRec is a known length – see below)
  so all numbers in the file are 2-digits (e.g., 05 rather than 5)
- a RANDOM ACCESS file – so `seek-`ing is needed (based on a given TP or RootPtr)
- headerRec contains: M,RootPtr,N (all 2-digit numbers)
- bTreeNodeRec's start with RRN 1 not 0
- bTreeNodeRec's contain: array of M-1 KV's, array of M-1 DRP's, array of M TP's
  - KV: keyValue is the 3-char country code
    - KV's are left-justified, with "empty's" (i.e., "___") on the right
  - DRP: dataRecordPtr is a small number "pointing to" actual data records (= their RRN) in the main data storage (CountryData file)
    - RRN's start at 1 not 0 for CountryData file
  - TP: treePtr is a small number "pointing to" another node in the, BTree

CountryData_?.txt (INPUT) - several versions of this file for testing – SEE BELOW

- ASCII text file (created with NotePad) with <CR><LF> after each record
- FIXED-LENGTH records to allow RANDOM ACCESS
- a RANDOM ACCESS file – so `seek-`ing is needed (based on a given DRP from the index)
- NO headerRec
- FYI: It's a direct address file on id, but that's irrelevant to your program.

NOTE: It could've been a hash file on name or a sequential file on population or a serial file on continent or…. None of that matters when an index is being used to access the data . "Searching" for a target which is the index's key field (i.e., country code) is based on using the DRP to provide the access path to the data. So only a single seek (i.e., 1 I/O for this part of the overall search path) is needed to access the correct record

- Real country data wasn't used – but that doesn't affect your program.
- Records only contain: id – 2 digit's followed by a space
  code – 3 char's followed by a space
  restOfData – 16 char' s (may include spaces)

`Log.txt` (OUTPUT) – a SINGLE file containing output from ALL of test-runs (as specified in the DemoSpecs). So the demo (to be turned in) has to have you MANUALLY DELETE the Log file at the start of the demo. Then the A5 program appends to the file on each test-run.

- NO file open/close or program starting/ending STATUS MESSAGES
- NO PrettyPrint output
- JUST the transaction handling:
  - echo of the request
  - simple printout of the DATA record (from CountryData file)
  - I/O stats:, the number of file READ's for THIS search, including:
    - number of BTree nodes read in
    - plus number of CountryData records read in

The printout format:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROCESSING TransDataA5_1.csv

QC,IMP >>>> 02 BUG in My program 24    [NODES: 1, DATA RECORDS: 1]
QC,CMU >>>> CODE NOT FOUND             [NODES: 3, DATA RECORDS: 0]
. . .
```

################################# TEST DATA #################################

ARTIFICIAL DATA & SIMPLIFIED FORMAT: Since there's NO Setup program to create CodeIndex and CountryData files, they were created manually with a text editor. To make this easier, artificial data and fewer fields and smaller values were used, including:

- Country codes are just 3-character words rather than valid country codes
- CountryData contains just a few data fields (with fake data) rather than the 6-8 actual fields of valid country data values from A1-A3
- CodeIndex file is a text file rather than a binary file, and it's a .csv file
- M's of 5 and 7 and 8 were used, rather than a more realistic M of 43 or . . .

3 DATA SETS FOR TESTING:

- `main` controller asks interactive user whichTestSet to use – i.e., the fileNameSuffix (1, 2, 3, …) for that program execution. This indicates which set of INPUT data files to use: `TransDataA5_?.csv,  CodeIndex_?.bin,  CountryData_?.txt`
- `main` supplies just the fileNameSuffix to the method/constructors so they can concatenate it appropriately before open their file
- There is NEVER more than a single data set (i.e., the 3 input files) used for a single run of the program.
- The major reason for having 3 test sets is to test how your program works for various M's and various Btree-heights for the CodeIndex. (TransData and CountryData vary just to accommodate the BTree data)

###################### CodeIndex CLASS - SOME NOTES ######################

- Internal storage:
  - headerRec 3 fields (though N's never used since we're not INSERTING)
  - 3 arrays to store a SINGLE BTree node's data:
    - array of M-1 KV's [0] to [M-2] BUT SEE "BIG NODE"NOTE BELOW so, M KV's, really, [0] to [M-1]
    - array of M-1 DRP's [0] to [M-2]
    - array of M TP's [0] to [M-1]

- *NOTE ON BIG NODE (internal storage): To simplify searching (in searchANode), reduce the 3 loop-STOPping conditions to 2:*
  1. *Found targetKV's matching KV in the node – so return corresponding DRP*
  2. *Found that targetKV is < KV[i] – so return appropriate TP*
  3. *~~Found that you've fallen off the right end of the node in your search – so return appropriate TP~~ which will automatically be caught with condition 2 by doing the following:*
  - *Define KV array to be of size M rather than M-1 (even though there are only really M-1 KV's actually in the FILE's node).*
  - *Initialize that extra KV (on the right end) as "___".*
  - *(NOTE: This initialization only has to be done ONCE at the start by the constructor, before ever reading in any node, since that extra "___" in KV[M-1] will remain there and never be over-written).*
  - *(NOTE: "___" was used to indicate "empty" because '_' is > 'Z' based on ASCII-value order. CAUTION TO C# PEOPLE: Use appropriate string comparison method*
  - *which uses ASCII-order).*

- Contains ALL handling of the external codeIndex (BTree) file – opening, reading, closing

- headerRec data is read into memory ONCE (for a particular CodeIndex file) in the constructor just after opening the file – so that all other uses of M and RootPtr in the searching part is done from MEMORY rather than having to read those values from the file each time.

- DO NOT read in the entire FILE into MEMORY. This is an EXTERNAL storage structure, not an internal data structure – so processing (searching) is done on the FILE.

- This class contains storage for a SINGLE node (well, actually, for "big node" – see above). There is NEVER more than ONE node in memory at once – so re-use the same storage space each time you read in a node into memory. (If you're using a separate node class, don't keep declaring new objects for every node you read – just keep re-using the same storage space to avoid the overhead of space-reclamation).

- Except for the headerRec data, DO NOT read in just a PART of a node (e.g., just the KV array). If you need to access any part of the node, you have to call readANode which reads in the WHOLE NODE.

- The actual BTree node handling methods are PRIVATE, including (with these names):
    - readANode
    - searchANode

  which each do EXACTLY WHAT'S ADVERTISED
    - read does not search, nor does it call search
    - search does not read, nor does it call read
    - this is a RANDOM ACCESS file, so read needs to `seek`

- queryByCode is a PUBLIC method which is the tree-search-controller
  - it calls readANode and searchANode, as needed, and decides when to stop

- DO NOT use a pseudo-goto (or recursive) loop by having
  readANode call searchANode,
  and searchANode call readANode.

- You MUST make this method GENERIC in terms of defining things as functions of M (e.g., [0] to [M-1] or. . . DO NOT HARCODE code 5 or 8 or 9 (which happen to be the M's for the 3 test files) anywhere in the program. Because realistically, as we calculated in class, M should be more like 43 or . . . depending on . . . So if M were 43 for some future CodeIndex file, there'd be no changes needed in your program.

- We want to have a SEPARATE METHOD called readANode because the project will eventually (NOT part of of A5) have to deal with a BINARY CodeIndex. So we would (pretty much) only have to change readANode's BODY when that time comes.

- readANode's body (for now) could contain:
    - a single read of the whole node (line), then. . .
    - OR a couple of for loops controlling reading of individual fields/arrays
    - OR . . .
    - But the METHOD MUST result in bringing AN ENTIRE NODE INTO MEMORY, nothing more, nothing less

- readANode can also do line-splitting, data-conversion to int's, etc. – besides the actual "reading" of a node

- searchANode MUST contain a loop based on a function of M, rather than if/else's
  since it has to be able handle any value for M

- allow for successful searches and unsuccessful searches

- SAVE TIME: only search as far into the node as needed, rather than ALL the KV's in a node (i.e., til a match is found or when targetKV < KV[i])

- SAVE TIME: don't double-search, 1st for a match in that node, then 2nd for < situation

----

WARNING: HUGE POINT_LOSS for A5 if program
- **does a linear search of CountryData file OR of CodeIndex file**
- **does NOT do a proper BTree search of the index to find target KV**
- **does NOT use the index's DRP to do directAddress of CountryData**
- **stores the whole BTree in memory**
- **has more than a SINGLE node in memory at once**
- **does NOT use some function M to control searching/reading a node (but instead uses hardcoded 5 or 8 or 9, say based on fileNameSuffix)**

But linear searching of the KV's in a single node (in memory) is FINE. It doesn't have to be binary search (which would probably be used if M were larger than say 30).

THOUGHT QUESTIONS:
What should M be using 512-byte blocks for the INDEX if:

A. 3-char CountryCode was the KV and Setup used 8-bit ASCII char's
   1. CountryData was DirectAddress on ID,
      there were relatively few countries,
      IDs were all < 32,767 (i.e., 2^16 – 1
         – the max positive value for a SHORT)
   2. So if there were <= 32,767 DATA records, there would certainly be <= 32,767 INDEX nodes

B. If countryCode used 16-bit Unicode char's rather than ASCII codes
   (and kept assumptions A1 and A2 above, using short's)?

*C. If the data was for CITIES rather than COUNTRIES, and the KV could include captital letters AND/OR digits (so that's 36 * 36 * 36 = 46,656 possible cityCodes).*
   1. ASCII char's were used for KVs
   2. The number of data records was thus possibly 46,656, so short's wouldn't work for DRPs or TPs?