CSCI 331 - Software Systems — Fall 2025

Zip Code Group Project 3.0

Objectives and Requirements

1. Generate a CSV (comma separated) file from this XLSX file.
2. Convert the CSV file to a file structure format (beginning with a header record) where the fields are still comma separated, but the records are length indicated.
3. Generate a *blocked* **sequence set** file from the data file you created in Group Project 2.0
   - Your blocked sequence set generation program's command line options should include:
     - the name of the blocked sequence set data file
     - all other information necessary for the header file
   - All blocks are the same size. (See the **Header Record Architecture** section below for the default size
   - Each block will contain a set of complete records (some blocks may have different counts of records) and a metadata architecture as shown in the **Block Architecture** section below
   - Unused or deleted blocks are *avail* list blocks (See Folk 6.2.2 & 10.1 – 10.3)
4. Process sequentially a blocked **sequence set** file using buffer classes. {functionality from Group Projects 1 & 2}
5. Use both a *block* buffer class and a *record* buffer class to read and unpack Zip Code Records from a sequence set **block** into a sorted container of record objects.
   - The *block* buffer unpacks a *record* from a block into a record buffer.
   - The *record* buffer unpacks *fields* from the record buffer into a record object.
6. *Modify* your data file *header record* buffer class to read and write the blocked sequence set data file header record
7. *Repeat* Group Project 1.0 with this new blocked sequence set file.
8. *Create* and use two blocked sequence set dump method that visibly aggregates Zip Codes into blocks including the respective predecessor & successor R(elative)B(lock)N(umber) links.
   One dump method will list the blocks sequentially by their physical ordering; the other dump method will list the blocks sequentially by their logical ordering.
   (after initial creation, both dumps will generate identical output, but use of a non-appending avail block will make them different)

```
List Head:  RBN
Avail Head: RBN
RBN  key_a key_b … key_i RBN
RBN  *available*     RBN
RBN  key_a key_b … key_j RBN
⋮
RBN  key_a key_b … key_k RBN
```

   This dump format makes it rather easy to check the results of insertions and deletions for appropriate changes — you could even use the `diff` program.
   It helps to use the smallest possible non-trivial sub-set of the data initially, so as to generate a dump which fits on a single page/window.
9. Create a simple index file which contains ordered pairs of keys (highest key in each block) & block numbers. (See Folk Figure 10.3)
10. Create a readable dump of the simple index
11. Generate (in RAM), write (as a file), and read (back into RAM), a simple primary key index [Folk Section 10.3] that can be used to display the Zip Code data for all Zip Codes listed on the command line.
    This index will store the ordered pairs: {<*highest* ***key*** *in block*>, <RBN>}

- Your blocked sequence set search program's command line options should include the name of the blocked sequence set data file
- Use a command line flag (e.g. `-Z56301`) to indicate each Zip Code record to search for.
- If the Zip Code record is not in the file, display a message to that effect.
  - Note that to determine that a record is not in the file, the indexed **block** must be read, unpacked, and searched
- Test Run Demonstration: for the blocked sequence set Zip Code data and simple index file pair
  1. Create and run a search test program - include searches (on the command line) for several valid Zip Codes and at least one invalid Zip Code.
     - the program will load the simple primary key index file into an sorted container object in RAM
     - the program will **never** load the blocked sequence set Zip Code data file into RAM
  2. Create and run a record addition and deletion test program
     - **record addition**: use the command line to indicate a file of records to add
       1. When a block is split, log the event.
       2. Optionally, also run the two dumps.
       3. If the index has to be modified, log the event.
       4. Optionally, run a dump of the index
     - **record deletion**: use the command line to indicate a file of keys for records to delete
       1. When two blocks are merged, or participants of a redistribution, log the event.
       2. Optionally, also run the two dumps.
       3. If the index has to be modified, log the event.
       4. Optionally, run a dump of the index

12. All program variables and values that can vary should be initialized either by command line parameters (or their defaults) or meta-data in the the data file or index (e.g. header record info.)
13. Document (*extensively*) your C++ source code with comments and Doxygen tags.
14. Create a Doxygen PDF of your class and application program code.
15. Create a user guide showing how to use your program (including how to use the command line options, and how the output should appear)

Header Record Architecture:

- file structure type {blocked sequence set with comma separated fields, and length-indicated records}
- version of your file structure type (we may add features in subsequent versions)
- header record size
- number of bytes for each record size integer (if fixed-size)
- size format type {ASCII or binary}
- block size {default to (512 Bytes / block)}
- minimum block capacity {default to 50%}, except for, possibly, the last block
- index file name
- index file schema information {how to read the index file format}
- record count
- block count
- count of fields per record
- for each field:
  - name or ID
  - type schema
    - (*format to read or write*)
- indicate (ordinally) which field serves as the primary key
- RBN link to the block avail-list {RBN ↔ Relative Block Number}
- RBN link to the active sequence set list
- stale flag

Block Architecture:

Each active block should include the following components:

- count of records (> 0)
- links to preceding & succeeding active blocks
- set of records ordered by key

Each *avail* list block should include the following components:

- count of records (== 0)
- link to succeeding avail block
- *all other bytes should be overwritten with blanks*

WHAT TO TURN IN:

Prior to coding (*submitted at least one week prior to submitting the following files*):

- the (*preliminary*) design document
- the (*preliminary*) test document

Final version:

- the user guide (`.txt`)
- the *length-indicated* data files files (`.txt`)
- the simple index file (`.txt`)
- the Doxygen PDF
- the `.cpp` and `.h` source code files
- the script file that was generated to demonstrate the running of the application program.
- the (*final*) design document
- the (*final*) test document demonstrating the operation of all of your programs (and classes)
  - generate and use small (*minimum useful size*) data files to demonstrate adding and deleting records to the blocked sequence set and its simple index.
    (consider testing with both the default block size and a block size that has a capacity of approximately six records) Show:
    - the adding of a record requiring no block split
    - the adding of a record requiring a block split, and use and updating of the avail list
    - the deletion of a record requiring no block deletion or redistribution
    - the deletion of a record requiring a block redistribution with no merge
    - the deletion of a record requiring a block merge with the logically rightmost block cleared and added to the avail list

(zip the `.cpp` source files: {see https://support.microsoft.com/en-us/help/14200/windows-compress-uncompress-zip-files})