# CS3114 (Fall 2017)
# PROGRAMMING ASSIGNMENT #4

**Due Thursday, December 7th @ 11:00 PM for 100 points**
**Due Tuesday, December 5th @ 11:00 PM for 10 point bonus**
Last updated: 11/13/2017

## Assignment:

**Update: In this project all parts that are designed for 2-3+ trees can be done with a corresponding BST. People using BST will still get the full credit, but those who implement 2-3+ tree (if problem coverage is above 45% out of the 60% in the correctness check) will receive extra bonus points.**

As part of a memory management package for storing variable-length records in a large memory space, in this project we will build some necessary data structures for doing search and analysis on a large song database. The records that you will store for this project are artist names and song track names from a subset of the Million Song database.

To store data for the song records, we will simplify the memory management. You should create a large array of bytes to store all songs and artist names. The artist names and song titles will be stored separately. For each record, the first byte (called flag) will be a sign to mark whether the data is active (0 means deleted and 1 means active). The next two bytes will be the (unsigned) length of the record, in (encoded) characters. Thus, the total length of a record may not be more than $2^{16} = 65536$ characters or bytes. Following that will be the string itself. Access to all records will be controlled by "handles". For each handle, it contains the start location (within the array) of the record.

To simplify the implementation, for insertion, append the new record to the last record of your array and mark it as active; For deletion, mark the corresponding data as deleted, but you don't have to physically remove the record from the array. Whenever the array size is not enough, you will create a new array with larger size and copy the old data to the new array. Access to the song records will be through several index files: two closed hash tables for accessing artist names and accessing song titles, and two 2-3+ (or BST) trees for range query.

For the hash tables, you will use the second string hash function described in the book, (the hash function will be provided by your instructors), and you will use simple quadratic probing for your collision resolution method (the **i**'th probe step will be $i^2$ slots from the home slot). The key difference from what the book describes is that your hash tables must be extensible. That is, you will start with a hash table of a certain size, (defined when the program starts). If the hash table exceeds 50% full, then you will replace the array with another that is twice the size, and rehash all of the records from the old array. For example, say that the hash table has 100 slots. Inserting 50 records is OK. When you try to insert the 51st record, you would first re-hash all of the original 50 records into a table of 200 slots. Likewise, if the hash table started with 101 slots, you would also double it (to 202) just before inserting the 51st record. The hash tables will actually store "handles" to the relevant data records that are currently stored in the memory. This handle is used to recover the record. For this project, it will be just the index of the data in the array.

You will also implement a 2-3+ (or BST) tree to support the search query. The data records to be stored in the 2-3+ tree are objects of the class KVPair, where the key is a handle, and the value is another handle. Your Handle class should be made to support the Comparable interface. To simplify implementation, we want to avoid storing records with duplicate keys. So while we use the "key" field when comparing records, if they have the same value in the key

(because they refer to the same artist, or to the same song), we will then use the value of the handle (location) in the value position of the KVPair to break the tie. For example, if one record has handle 10 in the key field and handle 30 in the value field, while another record has handle 10 in the key field and handle 40 in the value field, then the first one would appear in list of leaf nodes before the second. Whenever an insert command is called with a song/artist pair, the handles for these two strings will then be used to enter two new entries into the 2-3+ (or BST) tree. If the artist's name handle is named artistHandle and the song's name handle is named songHandle, then you will create and insert a KVPair object with the artistHandle as the key and the songHandle as the value field in the "artist" tree. You will then create and insert a second KVPair object with the songHandle as the key and the artistHandle as the value field in the "song" tree. However, be sure not to insert a duplicate artist/song record into the database, (i.e., array). If there already exists a KVPair object in the 2-3+ (or BST) tree with those same handles, then you will not add the song pair again. When a list command is processed, you will first get the handle for the associated string from the hash table. You will then search for that handle in the 2-3+ (or BST) tree and print out all records that have been found with that key value. When a remove command is processed, you will be removing all records associated with a given artist or song name. To (greatly!) simplify removing the records from the 2-3+ (or BST) tree, you will remove them one at a time. So, you will search for the first record matching the corresponding handle for the name that you want to delete, then call the (single record) delete operation on it, and repeat this process for as long as there is a record matching that handle in the tree. Also, whenever you remove a particular record (say, an artist/song pair), you will immediately remove the corresponding song/artist record. Finally, whenever you remove the last instance of either an artist or a song, you should always remove the associated record from the hash table and mark the string deleted (value 0) in the database memory, (i.e., the array).

When implementing the 2-3+ (or BST) tree, all major functions (insert, delete, search) must be implemented recursively. You may not store a parent pointer for the nodes. Your nodes must be implemented as a class hierarchy with a node base class along with leaf and internal node subclasses. For a 2-3+ tree implementation, internal nodes store two values (of type KVPair) and three pointers to the node base class. Leaf nodes store two KVPair records.

## Invocation and I/O Files:

The program would be invoked from the command-line as:

java SongSearch {initial-hash-size} {block-size} {command-file}

The name of the program is SongSearch. Parameter {initial-hash-size} is the initial size of the hash table (in terms of slots). Parameter {block-size} is the initial size of the array in bytes to store the song records. Whenever the array has insufficient space to insert the next request, it will be replaced by a new array that adds additional {block-size} bytes. All data from the old array will be copied over to the new array, and then the new string will be added.

Your program will read from text file {command-file} a series of commands, with one command per line. The program should terminate after reading the end of the file. The formats for the commands are as follows. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All commands should generate a suitable output message (some have specific requirements defined below). All output should be written to standard output. Every command that is processed should generate some sort of output message to indicate whether the command was successful or not.

***insert {artist-name}<SEP>{song-name}***

Note that the characters <SEP> are literally a part of the string (this is how the raw data actually comes to us, and we are preserving this to minimize inconsistencies in other projects), and are used to separate the artist name from the song name. Check if {artist-name} appears in the artist hash table, and if it does not, add that artist name to the memory, and store the resulting

handle in the appropriate slot of the artist hash table. Likewise, check if {song-name} appears in the song hash table, and if it does not, add that song name to the memory, and store the resulting handle in the appropriate slot of the song hash table. You should print a message if the insert causes a hash table or the memory pool to expand in size.

***remove {artist|song} {name}***

Remove the specified artist or song name from the appropriate hash table, 2-3+ (or BST) tree and make corresponding marks in the memory. Report the outcome (whether the name appears, and whether it was successfully removed).

***print {artist|song}***

Depending on the parameter value, you will print out either a complete listing of the artists contained in the database, or the songs. For artists or songs, simply move sequentially through the associated hash table, retrieving the strings and printing them in the order encountered (along with the slot number where it appears in the hash table). Then print the total number of artists or total number of songs.

***list {artist|song} {name}***

If the first parameter is artist, then all songs by the artist with name {name} are listed. If the first parameter is song, then all artists who have recorded that song are listed. They should be listed in the order that they appear in the 2-3+ (or BST) tree.

**delete {artist-name}<SEP>{song-name}**

Delete the specific record for this particular song by this particular artist. This means removing two records from the 2-3+ (or BST) tree, undoing the insert. If this is the last instance of that artist or of that song, then it needs to be removed from its hash table and the memory pool. In contrast, the remove command removes all instances of a given artist or song from the 2-3+ (or BST) tree as well as the hash table and memory pool.

**print tree**

You will print out an in-order traversal of the 2-3+ tree (or BST) as follows. First, before printing the tree, you should print out on its own line: “Printing 2-3 tree:” or “Printing BST:”. Each node is printed on its own line. Each node is indented by its depth times two spaces. So the root is not indented, its children are indented 2 spaces, its grandchildren 4 spaces, and so on. Internal nodes list the (one or two) placeholder records (both the key and the value for each one). That is, if an internal node stores a key (a handle) whose corresponding string is at position 10 in the memory, and a value handle with memory position 20, then you would print 10 20. Leaf nodes print, for each record it stores, the two handle positions. So they will print either 2 or 4 numbers. The numbers should be separated by a space.

## Programming Standards:

You must conform to good programming/documentation standards. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.

- Your header comment must describe what your program does; don't just plagiarize language from this spec.

- You must include a comment explaining the purpose of every variable or named constant you use in your program.

- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.

- Always use named constants or enumerated types instead of literal constants in the code.

- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.

- You must use indentation and blank lines to make control structures more readable.

- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.

- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

## Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

## Deliverables:

When structuring the source files of your project (be it in Eclipse as a “Managed Java Project”, or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored.

You will submit your project through the WebCat website. If you make multiple submissions, only your last submission will be evaluated.

You are permitted (and encouraged) to work with a partner on this project. When you work with a partner, then only one member of the pair will make a submission. Be sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

## Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another student,
// or any other unauthorized source, either modified or
// unmodified.
//
// - All source code and documentation used in my program is
// either my original work, or was derived by me from the
// source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
// anyone other than my partner (in the case of a joint
// submission), instructor, ACM/UPE tutors or the TAs assigned
// to this course. I understand that I may discuss the concepts
// of this program with other students, and that another student
// may help me debug my program so long as neither of us writes
// anything during the discussion or modifies any computer file
// during the discussion. I have violated neither the spirit nor
// letter of this restriction.
```

Programs that do not contain this pledge will not be graded.