CS3114 (Fall 2014)
PROGRAMMING ASSIGNMENT #1

Due Wednesday, September 17 @ 11:00 PM for 100 points

Due Tuesday, September 16 @ 11:00 PM for 10 point bonus

Updated: Tuesday 9/02

Assignment:

You will write a memory management package for storing **variable-length records** in a large memory space. For background on this project, view the modules on sequential fit memory managers in the OpenDSA class textbook.

The records that you will store for this project are artist names and track names from a subset of the Million Song database. This project will be the first in a series over the course of the semestter that will gradually build up the necessary data structures for doing search and analysis on a large song database.

Your **memory pool** will consist of a large array of bytes. You will use a doubly linked list to keep track of the free blocks in the memory pool. This list will be referred to as the **freeblock list**. You will use the **best fit** rule for selecting which free block to use for a memory request. That is, the smallest free block in the linked list that is large enough to store the requested space will be used to service the request (if any such block exists). If not all space of this block is needed, then the remaining space will make up a new free block and be returned to the free list. If there is no free block large enough to service the request, then you will grow the memory pool, as explained below.

Be sure to merge adjacent free blocks whenever a block is released. To do the merge, whenever a block is released it will be necessary to search through the freeblock list, looking for blocks that are adjacent to either the beginning or the end of the block being released. Do **not** consider the first and last memory positions of the memory pool to be adjacent. That is, the memory pool itself is not considered to be circular.

Aside from the memory manager's memory pool and freeblock list, the other major data structure for your project will be two **closed hash tables**, one for accessing artist names and the other for accessing song titles. For information on hash tables, see the chapter on Hashing in the OpenDSA textbook. You will use the second string hash function described in the book, 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 table will actually store "handles" to the relevant data records that are currently stored in the memory pool. A handle is the value returned by the memory manager when a request is made to insert a new record into the memory pool. This handle is used to recover the record.

Invocation and I/O Files:

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

```
java Memman {initial-hash-size} {block-size} {command-file}
```

The name of the program is `memman`. 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 memory pool (in bytes). Whenever the memory pool has insufficient space to insert the next request, it will be replaced by a new array that adds an additional `{block-size}` bytes. All data from the old array will be copied over to the new array, the freeblock list will be updated appropriately, 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 later 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 it to the memory pool. Check if `{song-name}` appears in the artist hash table, and if it does not, add it to the memory pool. You should print a message if the insert causes the hash table or memory pool to expand in size.

remove `{artist|song} {name}`

Remove the specified artist or song name from the appropriate hash table and the memory pool. Report the outcome (whether the name appears, and whether it was successfully removed).

print `{artist|song|blocks}`

Depending on the parameter value, you will print out either a complete listing of the artists contained in the database, or the songs, or else the free block list for the memory manager. 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. If the parameter is `blocks`, then print a listing of the freeblocks, in order of their occurrence in the freeblock list. For each block, print its start position and its length.

Design Considerations:

Your main design concern for this project will be how to construct the interface for the memory manager class. While you are not required to do it exactly this way, we recommend that your memory manager class include something equivalent to the following methods.

```
// Constructor. poolsize defines the size of the memory pool in bytes
MemManager(int poolsize);

// Insert a record and return its position handle.
// space contains the record to be inserted, of length size.
Handle insert(byte[] space, int size);
```

```
// Free a block at the position specified by theHandle.
// Merge adjacent free blocks.
void remove(Handle theHandle);

// Return the record with handle posHandle, up to size bytes, by
// copying it into space.
// Return the number of bytes actually copied into space.
int get(byte[] space, Handle theHandle, int size);

// Dump a printout of the freeblock list
void dump();
```

Another design consideration is how to deal with the fact that the records are variable length. One option is to encode the length in the record's handle. An alternative is to store the record's length in the memory pool along with the record. Both implementations have advantages and disadvantages. **We will adopt the second approach.**

The records stored in the memory pool **must** have the following format. The first 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 $2^{16} = 65,536$ characters or bytes. Following that will be the string itself.

Programming Standards:

You must conform to good programming/documentation standards. Note that Web-CAT will provide feedback on its evaluation of your coding style. While Web-CAT will not be used to define your coding style grade, the grader will take note of Web-CAT's style grade when evaluating your style. Some specific advice on a good standard to use:

- You should include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment should describe what your program does; don't just plagiarize language from this spec.
- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- 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 the codebase provided by the instructor. 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.

Deliverables:

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will **not** be given a copy of grader's test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

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. If you used a makefile to compile your code, or otherwise did something that won't automatically compile in Eclipse, be sure to include any necessary files or instructions so that the TAs can compile it.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix `zip` command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain all of the source code for the project, along with any files or instructions necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional "readme" file in your submitted archive.

You are permitted (and ecouraged) 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.