CS3114 (Spring 2016)
PROGRAMMING ASSIGNMENT #4

Due Tuesday, May 3 @ 11:00 PM for 150 points

Due Monday, May 2 @ 11:00 PM for 15 point bonus

Updated 4/26 at 9pm

Assignment:

This project serves as a capstone for the semester, in that it builds on the earlier projects that you completed for CS3114. You will re-implement Project 1 (with the same input commands), except that the entire database (rectangle objects and skip list nodes) will be stored in a file on disk. As with Project 3, all accesses to the disk file will be mediated by the buffer pool. Unlike Project 3, where the Mergesort determined where on file all data would go, there is no simple mechanism for determining where any given piece of data (such as a rectangle object or skip list node) should go in the disk file. Thus, you will add a memory manager to determine where each piece of the database should be placed when it is stored.

Input and Command Line Parameters:

The name of the program is `RectangleDisk` (this is the name where Web-CAT expects the main class to be).

Your program will take four command-line parameters. The first parameter is the name of the input command file The input commands will be identical to Project 1. The second parameter is the name of the file to be used to store the database while the program is running. Note that this will be your binary file (mediated by the buffer pool) and it will be initialized to be empty/zero length at the beginning of the program. The third parameter is the number of buffers in the buffer pool. The fourth parameter is the size (in bytes) for a buffer. So, the program would be invoked from the command-line as:

```
java RectDisk {commandFile} {diskFile} {numBuffs} {buffSize}
```

Implementation:

Just as in Project 3 you had to modify a normal memory-based Mergesort to request that data be read from or written to the file via the buffer pool, you will now need to modify your implementations for the skip list to request data access to the buffer pool rather than de-referencing pointers or allocating space for records and nodes. (I will use the term "database" to refer to the skip list and its collection of rectangle records.) Only a fixed (constant) amount of space should be needed by the database for (temporarily) holding the contents of various nodes or records. The database itself is maintained on disk. For the remainder of this discussion, I will refer to the data exchanged between the database and the buffer pool as *messages*. A given message might be KVPair (including the rectangle record), or a skip list node. For each message type that you will be exchanging with the buffer pool (i.e., KVPair or a skip list node), you will need access functions to transfer the data to/from the buffer pool. This is one of the two major changes from what you implemented in Project 1.

The second major change is that you need another entity to manage *where* in the file that a given message should be stored. When implementing the Mergesort, the sort knew where each data record belonged within the logical record array. For this project you will implement a *memory manager*, whose job is to keep track of what bytes in the disk file are available to write information, and which bytes already store messages. Before beginning this project, you should read Chapter

11 in OpenDSA on memory management. Note that the database will not actually make calls to the buffer pool. It will actually make all of its access requests to the memory manager, which in turn will interact with the buffer pool to get the message transfered to/from disk.

Your memory manager will maintain a linked list (in memory) of the free blocks, and you will use a **best fit** allocation policy, as described in OpenDSA Module 11.4.

A key consideration for your implementation is how the memory manager's client (the database in this project) interacts with the memory manager to store and retrieve its messages. Your memory manager should take a message and its size as input, and give back a **handle**. When the client wants the contents of a message, it hands the handle to the memory manager, which uses the handle to recover the contents of the message from the memory pool. To make your project easier to implement, the handle should just be a 4-byte `int` variable. The handle's value will actually be the byte position in the file for the message (although only the memory manager actually knows this). Your revised skip list implementation will replace what originally were references to KVPairs/rectangle objects, or other skip list nodes, with handles. To recover a message, the memory manager actually needs to know the starting position of the message and the length of the message. The memory manager will get the starting position from the handle. The memory manager should put the length of the message at the beginning of the message, storing this as a two-byte quantity. For example, if the message was 20 bytes long, then the memory manager will locate the smallest free block that is at least 22 bytes long. It will store the message length in the first two bytes, and then the message in the remaining 20 bytes. Any additional space in the free block beyond 22 bytes will be retained as a free block. For example, if the smallest free block is 40 bytes long, then 18 bytes will be returned to the free list.

If a request is made for a free block that cannot be serviced by the existing pool of free blocks, then new space is allocated at the end of the file, causing the file size to grow as necessary. It is acceptable to allocate an entire block (equal to a buffer pool's buffer size) or multiple blocks (as many as required) to service the request. Any additional space allocated beyond that needed to store the message would be added to the free block list. For example, if the block (buffer) size is 256 bytes and the message is 100 bytes long, then the remaining $256 - 102 = 154$ bytes should make up a new free block. Be careful that any existing free block at the end of the file is used as part of this allocation. For example, if the last 50 bytes of the file are free, the message is 100 bytes long, and blocks (buffers) are 256 bytes, then a new block will be allocated, yielding 306 total free bytes at the end of the file. 102 of those bytes are allocated to store the message, leaving a free block of 204 bytes.

Your memory manager must merge adjacent free blocks together whenever space for a message is freed by the client. You will need to work out some suitable approach for doing so, by examining the current free blocks in the linked list to determine if the newly freed block is adjacent to one or two existing free blocks.

There is one change to the output requirements from Project 1. For Project 4, the "dump" command will also print out a listing of the free blocks currently in the memory manager, indicating for each its position in the file and size of the free block.

Programming Standards:

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Beyond meeting Web-CAT's checkstyle requirements, here are some additional requirements regarding programming standards.

- 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.
- Source files should be under 600 lines.
- There should be a single class in each source file. You can make an exception for small inner classes (less than 100 lines including comments) if the total file length is less than 600 lines.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the GTAs or the instructors for debugging help 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 provided by the instructor. Note that the OpenDSA 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 implement your project using Eclipse, and you will submit your project using the Eclipse plugin to Web-CAT. 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 these 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, use a flat directory structure; that is, your source files will all be contained in the project "src" directory. Any subdirectories in the project will be ignored.

You are permitted 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.