CS3114 (Fall 2012)
PROGRAMMING ASSIGNMENT #1

Due Tuesday, September 18 @ 11:00 PM for 100 points
Due Monday, September 17 @ 11:00 PM for 10 point bonus
Initial Schedule due Tuesday, September 4 @ 11:00 PM
Second Schedule due Tuesday, September 11 @ 11:00 PM

[Updated 9/6/2012]

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 tutorial on sequential fit memory managers available at `http://research.cs.vt.edu/AVresearch/MMtutorial`.

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 **circular first fit** rule for selecting which free block to use for a memory request. That is, the first free block (starting from the last free block looked at) 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.

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 the **record array**, an array that stores the "handles" to the 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. (Note that the record array is something of an artificial construct that is being used to simplify testing the memory manager for this project. It will be replaced by something more appropriate in later projects. The idea is that the record array gives us an easy way to identify the records independent of their placement in the memory pool.)

A key element in many bioinformatics problems is the biological sequence. A biological sequence is just a list of characters chosen from some alphabet. Two of the common biological sequences are DNA (composed of the four characters A, C, G, and T) and RNA (composed of the four characters A, C, G, and U). In this project, you will be storing and searching for DNA sequences, defined to be strings on the alphabet A, C, G, and T.

The records that your memory manager will store are encoded forms of DNA strings. Instead of storing the strings as ASCII characters, you will code each letter as a two bit code. Use 00 for A, 01 for C, 10 for G and 11 for T. Thus, in encoded form, you will be able to store four letters in each byte of the memory pool array. If the sequence length is not a multiple of four characters, then the last byte will have some unused bits.

Invocation and I/O Files:

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

```
java memman <pool-size> <num-recs> <command-file>
```

The name of the program is `memman`. Parameter `<pool-size>` is the size of the memory pool (in bytes) that is to be allocated. Parameter `<num-recs>` is the size of the record array that holds the handles to the records stored in the memory pool. 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 `<recnum> <DNAstring>`

Parameter `<recnum>` specifies which slot in the record array will hold the handle for this record. An error should be reported if the value of `<recnum>` is outside of the range 0 to `<num-recs>` − 1. The `<DNAstring>` will be characters in the DNA alphabet. There is no upper limit to the string length.

If there is already a record stored at position `<recnum>` in the record array, then that earlier record should first be removed from the memory pool. If there is no room in the memory pool to handle the request, print a suitable message and do not modify the memory pool in any way. If the insert command is to a `<recnum>` that is already used, then the first step will be to delete the old record, and the second step will be to attempt to insert the new record. Should this attempt to insert fail, then the old record will remain deleted.

remove `<recnum>`

Remove the record whose handle is stored in position `<recnum>` of the record array. If there is no record there, print a suitable message. An error should be reported if the value of `<recnum>` is outside of the range 0 to `<num-recs>` − 1.

print `<recnum>`

Print out the record (the DNA sequence) whose handle is stored in position `<recnum>` of the record array. If there is no record there, print a suitable message. An error should be reported if the value of `<recnum>` is outside of the range 0 to `<num-recs>` − 1.

print

Dump out a complete listing of the contents of the memory pool. This listing should contain two parts. The first part is the listing of DNA sequences currently stored in the memory pool, in order of the record number. Print the value of the position handle along with the record. The second part is a listing of the free blocks, in order of their occurrence in the freeblock list.

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 store the record's handle and length in the record array. 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 65535 characters, for a total of 16384 bytes. Following that will be the string itself in encoded format.

Finally, we must deal with the issue that the memory pool stores the length of the physical message (the number of bytes that it stores), but this is not enough information to tell exactly how many characters are in the DNA sequence since the byte length has to correspond to a multiple of four characters. Somewhere we have to store the actual sequence length. This should be done in the record array along with the record handle.

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. 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 will submit your project through the automated Web-CAT server. Links to the Web-CAT client and instructions for those students who are not developing in Eclipse are posted at the class website. If you make multiple submissions, only your last submission will be evaluated.

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.

Scheduling:

In addition to the project submission, you are also required to submit an initial project schedule, an intermediate schedule, and a final schedule with your project submission. The schedule sheet template for this project is posted at the course website. You won't receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose 5 points from the project grade.

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.