CS3114 (Spring 2011)
PROGRAMMING ASSIGNMENT #4

Due Tuesday, May 3 @ 11:00 PM for 150 points
Early bonus date: Monday, May 2 @ 11:00 PM for a 15 point bonus
Initial Schedule due Thursday, April 14 @ 11:00 PM
Intermediate Schedule due Tuesday, April 26 @ 11:00 PM

In this project you will re-implement the functionality of Project 3. However, you will replace the in-memory search tree with a disk-based hash table using a simple bucket hash. The memory manager will operate much as it did in Project 3, however it will be storing both sequences and sequenceIDs.

As with the earlier projects, define DNA sequences to be strings on the alphabet A, C, G, and T. In this project, you will store data records consisting of two parts. The first part will be the SequenceID. The SequenceID is a relatively short string of characters from the A, C, G, T alphabet. The second part is the sequence. The sequence is a relatively long string (could be thousands of characters) from the A, C, G, T alphabet.

Input and Output:

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

`P4 <command-file> <hash-file> <hash-table-size> <memory-file>`

The name of the program is `P4`. Parameter `command-file` is the name of the input file that holds the commands to be processed by the program. Parameter `hash-file` is the name of the file that holds the hash table. Parameter `hash-table-size` defines the size of the hash table. This number must be a multiple of 32. The hash table never changes in size once the program starts. Parameter `memory-file` is the name of the file used by the memory manager to store strings.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), with no more than one command on a line. A blank line may appear anywhere in the command file (except within the **insert** command, see below), and any number of spaces may separate parameters. You need not worry about checking for syntactic errors. That is, only the specified commands will appear in the file, and the specified parameters will always appear. However, you must check for logical errors. The commands will be read from a file, and the output from the commands will be written to standard output. The program should terminate after reading the EOF mark.

The commands will be as follows (these are generally identical to Project 3, with minor changes to what gets output):

The **insert** command consists of two lines (no blank lines will come between these two lines). The first line will have the format:

insert *sequenceId length*

The *sequenceId* is a string (from A, C, G, T) that is used as the sequence identifier. The *length* field indicates how long the sequence itself will be. This seqence appears on the second line. The sequence line will contain no spaces (that is, the sequence is not preceeded or followed by spaces any space on that line, and no spaces appear within the sequence). The sequence consists only of the letters A, C, G, T. The sequence can (and often will) be thousands of characters long.

remove *sequenceID*
Remove the sequence associated with *sequenceID* from the hash table and from the memory manager, if it exists. Print a suitable message if *sequenceID* is not in the database. If a sequence is removed, then print the complete sequence.

print
Print out a list of all *sequenceIDs* in the database. For each one, indicate which slot of the hash table it is stored in. Also, print out a listing of the free blocks currently in the file. For each such free block, indicate its starting byte position and its size. Such blocks should be listed from lowest to highest in order of byte position (the same order that they are stored on the freelist).

search *sequenceID*
Print out the sequence associated with *sequenceID* if it there is one stored in the database.

Implementation:

Instead of a search tree, you will use a hash table to store and retrieve sequence records. Each slot of the hash table will store two memory handles. One memory handle will be for the sequenceID, the other memory handle will be for the sequence. Unlike Project 3, sequenceIDs will also be stored in the memory manager, so that the hash table can store fixed-length records.

Memory handles are defined to be two 4-byte integers. The first is the byte position in the file for the associated string. The second is the length (in characters) of the associated string. As with Project 3, all strings will actually be converted to 2-bit codes when stored on disk, with 4 codes per byte.

The hash table will use a modified bucket hash compatible with disk-based hashing. The first step will be to hash a sequenceID using a version of the `sfold` hash function that will be posted with this assignment. Each bucket will be 512 bytes, or 32 table slots since each slot stores two memory handles, each of which is two 4-byte integers in length. Collision resolution will use simple linear probing, with wrap-around at the bottom of the current bucket. For example, if a string hashes to slot 60 in the table, the probe sequence will be slots 61, then 62, then 63, which is the bottom slot of that bucket. The next probe will wrap to the top of the bucket, or slot 32, then to slot 33, and so on. If the bucket is completely full, then the insert request will be rejected. Note that if the insert fails, the corresponding sequence and sequenceID strings must be removed from the memory manager's memory pool as well.

Sourcecode:

For this project you may not use any Java library classes for lists, file-based memory management, or hashing. You may use sourcecode distributed with the textbook, or any code that you wrote yourself for previous projects (so long as such code does not rely on library classes for lists or otherwise forbidden sourcecode).

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 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.

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 10 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.