CS3114 (Spring 2011)
PROGRAMMING ASSIGNMENT #3

Due Thursday, April 7 @ 11:00 PM for 100 points
Early bonus date: Wednesday, April 6 @ 11:00 PM for a 10 point bonus
Initial Schedule due Thursday, March 24 @ 11:00 PM
Intermediate Schedule due Thursday, March 31 @ 11:00 PM

[Updated 3/24/2011]

In this project you will implement a simple database system for DNA sequences. Your database system will include an in-memory index structure to support searches by sequence identifier. The bulk of the database will be stored in a binary file on disk, with a simple memory manager used to keep track of where to put the DNA sequences.

As with Project 2, 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 identifier, called 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. The *sequenceID*'s will be stored in the in-memory index structure (this will be either a DNA tree from Project 2 or a BST tree). The main DNA sequences themselves are stored in a disk file, with placement of the strings within the file controlled by a memory manager.

Input and Output:

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

`P3 <command-file>`

The name of the program is `P3`. Parameter `command-file` is the name of the input file that holds the commands to be processed by the program. Output from the commands will be written to standard output. The program should terminate after reading the EOF mark.

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. **Every command that is processed should generate some sort of output message to indicate whether the command was successful or not.**

The commands will be as follows (in general, they are similar in spirit to those of Project 2).

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. The sequence itself appears on the line immediately following the **insert** command. The sequence line will contain no spaces, nor will the sequence be preceded or followed by spaces. The sequence consists only of the letters A, C, G, T. The sequence can (and often will) be thousands of characters long. The *sequenceID*s will be stored in a search tree (DNA tree or BST). The long sequences are stored in the disk file. It is an error to insert strings with duplicate *sequenceID* values. Such an error should be reported in the output, and no changes to the tree structure, memory pool, or file contents should take place.

For each insert command, you should output a message that includes the *sequenceID* and which indicates whether the insert operation was successful or not.

remove *sequenceID*

Remove the sequence associated with *sequenceID* from the DNA tree and from the memory manager, if it exists. (You do not actually need to modify the disk file when removing the sequence.) Print a suitable message if *sequenceID* is not in the tree. If a sequence is removed, then print the complete sequence. Removing the sequence makes that space in the file available for reuse later.

print

Print out a list of all *sequenceIDs* in the database. 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 *sequenceDescriptor*

Print both the *sequenceID* and the complete sequence for each record whose *sequenceID* matches *sequenceDescriptor*. The *sequenceDescriptor* can come in two forms. The first form is simply as a sequence containing letters from the alphabet A, C, G, and T. If this form is given, then print all sequences stored in the database whose *sequenceID* has *sequenceDescriptor* as a prefix (including exact matches). The second form is a sequence from the letters A, C, G, and T, followed by a $ symbol. If this form is given, then only an exact match of a *sequenceID* (without the $ symbol) would be printed if it exists in the database, along with the associated string.

Implementation:

Your database system will consist of three main parts: An indexing structure to support the searches, etc., the binary file that stores the large sequences, and a linked list used by the memory manager to track the free blocks within the disk file.

For the indexing structure, you have two choices: the DNA tree from Project 2, or a standard BST. In general, if you successfully completed implementation of the DNA tree, you should reuse it for this project. No changes to the DNA tree should be necessary, all that should change for this project is the contents of the data records. However, if you had trouble completing Project 2, then you may use a BST instead for the index.

The main implementation component for this project is support for storing the large sequences. You will store sequences on disk in a binary file. The name of this binary file will be `biofile.out`. A major consideration is deciding where to store a given sequence in the binary file. This will be controlled by a memory manager implementing First Fit. See Section 12.3 of the textbook for a description of how this works. You will use a linked list to keep track of the free sections of the binary file. Initially the binary file will be empty (have no size). Whenever a new sequence is to be inserted, it should be inserted into a free section within the existing bounds of the file if possible. When this is not possible, the size of the file should grow as necessary to accommodate the new sequence. Whenever a sequence is removed from the file, the memory manager should merge together adjacent free sections into a single larger section if possible.

Within the binary file you will **not** store the sequences as ASCII characters. Instead, you will store each letter of the sequence as two-bit code values, where A has code 00, C has code 01, G has code 10, and T has code 11. Four letters of the sequence will be packed into a single byte of the file. Space for sequences will always be allocated as full bytes. If the length of a sequence is not a multiple of 4, then the last few bits of the last byte storing the sequence will be unused.

Your memory manager should operate by receiving a sequence to store, and returning a "handle" to its caller. The caller stores the handle so that it can later retrieve the sequence. You may

implement handles as you wish, but typically they will store the starting byte position of that sequence in the file and the length of the sequence. Your indexing structure (the DNA tree or BST) will store data records consisting of the *sequenceId* and the handle for the sequence.

Be careful about allocating space for storing sequences. When reading in the sequence for the **insert** command, you are given the sequence length. You can use this to allocate a buffer into which the sequence can be read. Likewise, when requesting a sequence from the memory manager, a buffer will be created in which to place the sequence.

Sourcecode:

For this project you may not use any Java library classes for lists or file-based memory management. 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.