CS3114 (Fall 2011)
PROGRAMMING ASSIGNMENT #4

Due Tuesday, December 11 @ 11:00 PM for 150 points
Due Monday, December 10 @ 11:00 PM for 15 point bonus
Initial Schedule due Thursday, November 15 @ 11:00 PM
Second Schedule due Tuesday, December 4 @ 11:00 PM

Please note that this assignment is worth more than the others. It is not intended that this assignment be more difficult than the others. Rather, this should be an opportunity to make up for any past problems. However, the more defects that were in your prior projects, the more work you will have now.

In this project, you will re-implement the DNA tree for storing DNA sequences from Project 2. While the functionality is the same, this time the DNA Tree and the sequences themselves will reside on disk. A buffer pool (using the LRU replacement strategy) will mediate access to the disk file, and a memory manager (similar to the one implemented for Project 1) will decide where to store the DNA Tree nodes as well as the sequences. Another way to look at this project: You will take the memory pool from Project 2, move it to disk, store the DNA tree nodes in the memory pool, and use a buffer pool to manage the disk I/O.

Input and Output:

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

```
java DNAfile <command-file-name> <numb-buffers> <buffersize>
```

The `<command-file-name>` parameter is the name of the command input file, as in Project 1. The `<numb-buffers>` parameter is the number of buffers in the buffer pool, and will be in the range 1–20. Parameter `<buffersize>` is the size of a buffer in the buffer pool (and therefore determines the amount of information read/written on each disk I/O operation).

You will need to create and maintain a disk file which replaces the memory array from Project 2. The buffer pool acts as the intermediary for this file. The name of this disk file must be "p4bin.dat". After completing all commands in the input file, all unwritten blocks in the buffer pool should be written to disk, and "p4bin.dat" should be closed, **not deleted**.

Your buffer pool should need only two changes from Project 3. First, the size for the buffers is determined by the command line parameter. Second, since the data entities stored on disk are of variable size (differerent tree node types, and sequence strings), your buffer pool must be able to handle storing messages that span block boundaries, or that might even span multiple blocks.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. The commands will be read from the file given in command line parameter 1, and the output from the command will be written to standard output. The format and interpretation for the commands will be identical to Project 2, with the following exception. In the "print" command, you must include the block ID's of the blocks currently contained in the bufferpool in order from most recently to least least recently used. Note that "Block ID" simply refers to the block number, starting with 0. Thus, if the block size is 1024 bytes, then bytes 0-1023 are in Block 0, bytes 1024-2047 are in Block 1, and so on.

Implementation:

The implementation rules for the DNA Tree from Project 2 are still in place. That is, all operations that traverse or descend the DNA Tree structure **MUST** be implemented recursively, and the DNA Tree nodes **MUST** be implemented with separate classes for the internal nodes and the leaf nodes, both of which inherit from some base node type. A flyweight **MUST** be used for empty leaf notes.

The DNA Tree itself will have its nodes stored in the memory manager's space on disk, and not in main memory. This is the primary difference from Project 2. The nodes will be of variable length, and where a node is stored on disk will be determined by the memory manager. When implementing the DNA Tree nodes, access to a node (perhaps you did this by using a "node.getchild()" method) will now mean a request to the memory manager to return the node contents from the memory pool, and creation or alteration of a node will require writing to the memory pool. From the point of view of the memory manager and the buffer pool, communications are in the form of variable-length "messages" that must be stored.

The memory manager should follow the same definition as in Project 1. In particular, the location for placing the next message within the memory pool should be determined using circular first fit. The list of the free blocks may be maintained in main memory, and adjacent free blocks should be merged together.

Initially, the memory pool (and the "p4bin.dat" file) should have length 0. Whenever a request is made to the memory manager that it cannot fulfill with existing free blocks, the size of the memory pool should grow by one (or more if necessary) disk blocks (of size `<buffersize>`) to meet the request.

The memory manager will be managing data residing in the memory pool, and this memory pool will reside on disk, but the memory manager does not actually have direct access to the disk. All disk access is through the buffer pool. Thus, the flow of control for a node access is as follows: The DNA Tree will request a "message" from the memory manager via a handle, the memory manager will ask the buffer pool for the data at the physical location stored in the handle, the buffer pool will give the contents of the "message" to the memory manager, which will in turn give the contents of the "message" back to the DNA Tree.

The layout of the DNA Tree node messages sent to the memory manager **MUST** be as described in the following paragraphs. Note that, as in Project 1, the memory manager will actually add to each message the length of the message before storint it in the memory pool. Handles will always be a 4-byte quantity that indicates the starting byte position of the message in the disk file.

Internal nodes will store 21 bytes: a one-byte field used to distinguish internal from leaf nodes, followed by five 4-byte fields to store handles for the five children.

Empty nodes will be represented by storing in the empty node's parent a handle value that is recognized as representing an empty leaf node (the flyweight). The flyweight may be actually represented as a physical node on disk, or you may use a special handle value that is simply recognized as the flywight.

Leaf nodes that contain a DNA sequence will require 7 bytes, storing the following fields: a one-byte field used to distinguish internal from leaf nodes, a two-byte field to store the actual string length for the DNA sequence, and a 4-byte field that stores the handle to the DNA sequence.

Each DNA sequence will be stored as a separate message in the memory pool. Its format is idential to that of Project 1. That is, the letters are encoded by two-bit codes.

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, along with instructions on how to fill out the schedule sheet and what the requirements are for a satisfactory schedule. The scoring scheme will be similar to that of Project 1, where you will receive a penalty of 5 points for each schedule sheet that you fail to turn in on time.

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. A copy of the text of this pledge is also be posted online at the course assignments page so to make it easy for you to copy into your project file.

```
// 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.