Project #3

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 implement a simple database system for DNA sequences. Your database system will include a disk-based hash table using a simple bucket hash, to support searches by sequence identifier. You will store data records consisting of two parts. The first part will be the identifier (sequenceIDs), which is a relatively short string of characters from the A, C, G, T alphabet. The second part is the sequence, which 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:

java DNAdbase <command-file> <hash-table-size> <memory-file>

The name of the program is **DNAdbase**. Parameter command-file is the name of the input file that holds the commands to be processed by the program. 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.

An example of one command-line:

java DNAdbase sampleInput.txt 32 memory.bin

The commands will be as follows:

The **insert** command consists of two lines (no blank lines will come between these two lines).

The first line will have the format:

Insert AGCGA 10

The second line will have the format:

AAAAGGGGTT

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 sequence appears on the second line.** The sequence line will contain no spaces (that is, the sequence is not preceded or followed by 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 list 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 there is one stored in the database.

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. The part of code for memory manager will be provided as **MemoryManager.java and MemoryHandleHolder.java.**

For the indexing structure, 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. In this project, sequenceIDs will be stored in the memory manager, so that the hash table can store fixed-length records.

One 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 **bin file**. 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 16.4 of the OpenDSA for a description of how this works. You can 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. 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. All strings will actually be converted to 2-bit codes when stored on disk, with 4 codes per byte.

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.

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.

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 postconditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the TAs 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 codes 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:

Sample data files will be provided 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:

You will implement your project using IntelliJ. You are not required to submit your own test cases with your program. Of course, your program must pass the sample test cases. Your grade will be fully determined by test cases that are provided by the grader (TA). TA 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 directory structure; that is, your source files will all be contained in the project “src" directory. Compress all files into one zip file named “src.zip” and **only submit the zip file**. Any subdirectories in the project will be ignored.

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. Both names and emails must be included in the documentation.

Please strictly follow the description of the java file names and deliverables!

Tips:

Since Memory Manager is provided, you need to consider implementing other classes and objects to help you build the Hash Table.

Rubric:

Commenting: 10 points
Readability/Style: 10 points
Implementation: 40 points
Correctness: 40 points

// 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, 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.