DNA Strings in Hash Table on Disk

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. 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, hash table, or file contents should take place.

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).