CS3114 (Fall 2012)
PROGRAMMING ASSIGNMENT #2

Due Tuesday, October 16 @ 11:00 PM for 100 points

Early bonus date: Monday, October 15 @ 11:00 PM for a 10 point bonus

Initial Schedule due Wednesday, September 26 @ 11:00 PM

Intermediate Schedule due Tuesday, October 9 @ 11:00 PM

[Updated 9/20/2012]

Continuing the theme of bioinformatics applications, we turn to the problem of searching for matching sequences in a sequence database. A key element in many bioinformatics problems is the biological sequence. A 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 be storing and searching for DNA sequences, defined to be strings on the alphabet A, C, G, and T.

Given a sequence, simply searching a list of sequence strings for one that matches would take a long time for a large collection of sequences. Instead, we will use a tree structure to store sequences in a way that allows for efficient search. Not only can we determine whether a specified sequence is in the database, but we can also find any sequence in the database that matches a search prefix.

DNA Trees:

We will define a new tree data structure to store DNA sequences, that we call a DNA tree. You should read Section 13.3.2 in the textbook on PR quadtrees, since DNA trees are quite similar. DNA trees store information about sequences in their leaf nodes. Internal nodes serve only as placeholders to help direct search, they store no data. A leaf node is either empty, or stores a single sequence. Whenever you attempt to insert a new sequence and the insert process reaches a leaf node containing a sequence, that leaf node must split (just as in PR quadtree insertion). Whenever you remove a sequence from a leaf node, if possible that node will merge with its siblings.

The DNA tree is a 5-way branching tree, with a branch for each possible letter. In addition to the letters A, C, G, and T, we must augment the alphabet for DNA sequences to contain $ to indicate the termination of a sequence. This permits us to store sequences that are prefixes of other sequences already stored (without the $ symbol, a prefix would end up at an internal node of the tree, which is forbidden). Thus, the five branches correspond to the five letters of the augmented DNA alphabet: A, C, G, T, and $.

When traversing through the tree structure to perform an operation, the first branch from an internal node corresponds to the letter A, the second branch to the letter C, the third to G, the fourth to T, and the fifth branch corresponding to $. Thus, all sequences stored that begin with A will be in the first subtree, all sequences stored that begin with C will be in the second subtree, and so on. If there are no sequences stored in the tree, the tree consists of a single empty leaf node. If there is one sequence stored in the tree, the tree consists of a single leaf node containing the sequence.

Note that you will not literally store the sequence in the leaf nodes of the tree. You will actually store the string in your memory manager that you implemented for Project 1, and you will store the handle returned by the memory manager in your DNA tree leaf nodes.

There will be no command-line parameter to determine the size of the memory manager. Instead, you will initially allocate a byte array of 100 bytes for the memory pool. If a request comes in for a string that cannot fit in the available memory, then you will grow the memory pool by a multiple of 100 bytes until you have enough space. To grow the memory pool, simply allocate a new array, copy the contents of the old array over to the new one, update the freeblock list, and continue with the insert operation.

Input and Output:

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

`DNAtree <command-file>`

The name of the program is `DNAtree`. Parameter `<command-file>` is the name of the input file that holds the commands to be processed by the program.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. A blank line may appear anywhere in the command file, 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 the command file, and the output from processing the commands will be written to standard output. All commands should generate a suitable output message (some have specific requirements defined below). The program should terminate after reading the end-of-file mark. The commands are as follows:

insert `<sequence>`

Insert `<sequence>` into the DNA tree. Print a message indicating if the insertion was successful, and if so, indicate the level of the leaf node inserted. It is an error to insert a duplicate sequence. Such an error should be reported in the output, and no changes to the tree structure should take place.

remove `<sequence>`

Remove `<sequence>` from the DNA tree if it exists. Print a suitable message if `<sequence>` is not in the tree.

print

Print out the DNA tree and the freeblock list from the memory manager.

Printing the DNA tree will include displaying both the node structure and the sequences it contains. You should perform a preorder traversal of the tree, and print each node on a separate line in the order that it is visited by the traversal. If the node is internal, just print the letter I. If the node is an empty leaf node, just print the letter E. If the node contains a sequence, print the sequence. All nodes should be printed so that the line is indented by 2 spaces for each level in the tree. That is, the root node is printed with no indentation, immediate children of the root are indented 2 spaces, grandchildren of the root are indented 4 spaces, and so on.

You should print the freeblock list in a format similar to what you used for the first project.

print lengths

Output is identical to that of the **print** command, except that the length of the sequence is printed after the sequence for all sequences stored in the database.

print stats

Output is identical to that of the **print** command, except that the letter breakdown (by percentage) of the sequence is printed after the sequence for all sequences stored in the database. That is, for each letter A, C, G, and T, the percentage of A's in that sequence is printed, the percentage of C's, and so on. Percentages should be printed with two decimal places. For example, AAAAG should show A's as 80.00% and G's as 20.00%.

search `<sequenceDescriptor>`
Find all sequences that match `<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 tree for which `<sequenceDescriptor>` is 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 the sequence (without the $ symbol) is to be printed. Print the number of nodes visited in the tree during the search.

Implementation:

You must use class inheritance to design your DNA Tree nodes. You must have a DNA Tree node base class, with subclasses for the internal nodes and the leaf nodes.

Note that many leaf nodes of the DNA tree will contain no data. Storing many distinct "empty" leaf nodes is quite wasteful. One design option might be to use a NULL pointer to represent an empty leaf node. However, this requires the parent node to understand this convention, and explicitly check the value of its child pointers before proceeding, with special action taken if the pointer is NULL. Among other problems, this eliminates the possibility of using a composite design. A better approach is to use a "flyweight" object to represent the empty leaf nodes. A flyweight is a single empty leaf node that is created one time at the beginning of the program, and pointed to whenever an empty child node is needed. Your project should use a flyweight design to implement empty leaf nodes.

Internal nodes may not store data of any type (other than the pointers to children). Neither leaf nodes nor internal nodes may store a pointer to their parent. No operation should look at more nodes than necessary (especially the **search** operation).

All DNA tree operations must be implemented recursively.

As mentioned above, the actual DNA sequences are stored in the memory manager that you implemented for Project 1. The DNA tree will in effect replace the record array from Project 1. Non-empty leaf nodes will store a handle to the string as returned by the memory manager, along with the true length of the string. Whenever you need to look at the value of the string (say to print or to compare for a search), you will retrieve the string from the memory manager.

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. Unlike Project 1, points from the total score will be allocated as credit for completing the schedule sheets.

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.