DNA Tree Project

The project addresses the problem of searching for matching sequence strings in a sequence database. 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 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 on PR quadtrees in the textbook, since DNA trees are quite similar. DNA trees store 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, and so on, with 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.

Input and Output:

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

```
P2 <command-file>
```

The name of the program is `P2`. 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. The program should terminate after reading the EOF 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, including 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.

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 is to store a NULL pointer to an empty leaf node in its parent. 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. A better design is to use a “flyweight” object. 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). No operation should look at more nodes than necessary (especially the **search** operation).

All DNA tree operations must be implemented recursively.

The DNA sequences stored in the leaf nodes may be implemented using a linked list, or you may store them in a character array of the appropriate size.