

## CS2604 (Spring 2003)

### PROGRAMMING ASSIGNMENT #2

Due Wednesday, February 26 @ 11:00 PM for 100 points

Early bonus date: Tuesday, February 25 @ 11:00 PM for a 10 point bonus

Late penalty: 10 points per day (due by 11:00 PM for that day's credit)

Final late date: Saturday, March 1 @ 11:00 PM for a 30 point penalty.

Last Revised: 2/06/2003

Continuing the theme of bioinformatics applications, we turn to the problem of searching for matching sequences in a sequence database. Searching through the sequence array from Project 1 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 that matches a sequence prefix.

As with Project 1, we define DNA sequences to be strings on the alphabet A, C, G, and T.

#### **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 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 named `bio2` and be invoked from the command line with no arguments. 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 standard input, and the output from 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.

**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 A's in that sequence is printed, the percentage of C's, and so on.

**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.

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 search operation).

All DNA tree operations must be implemented recursively.

The DNA sequences stored in the leaf nodes may be implemented using your linked list class from Project 1, or you may store them in a character array of the appropriate size.

You must implement a single traversal method to support the **search** and **print** commands. The variations on these commands (two forms of search and three forms of print) will be controlled either by passing in one or more functions as arguments to the traversal method, or else by using a templated method to pass in one or more classes or methods that do the decision making.

### **Programming Standards:**

You must conform to good programming/documentation standards, as described in the Elements of Programming Style. 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 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. You may not use code from STL, MFC, or a similar library in your program.

### **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. 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 submit a tarred and gzipped file containing all the source code for the project, a makefile and an optional readme file. The file should end in a `tar.gz` extension. The makefile will allow the TA to simply type `make` at the command line and create your executable. The optional readme file will explain any pertinent information needed by the TA to aid them in the grading of your submission.

Once you have assembled the archive for submission, for your own protection, please move it to a location other than your development directory, unzip the contents, build an executable, and

test that executable on at least one input. Failure to do this may result in delayed evaluation of your program and a loss of points.

You will submit your project to the automated Curator server. The instructions and necessary software are available at: <http://ei.cs.vt.edu/~eags/CuratorGuides.html>. If you make multiple submissions, only your last submission will be evaluated.

### **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 in the header comment preceding the function `main()` in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used C++ language code obtained from another student,
//   or any other unauthorized source, either modified or unmodified.
//
// - All C++ language 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 instructor, ACM/UPE tutors or the GTAs 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.