CS2984 (Spring 2009)
PROGRAMMING ASSIGNMENT #1

Due Tuesday, February 10 @ 11:00 PM for 100 points
Early bonus date: Monday, February 9 @ 11:00 PM for a 10 point bonus
Initial Schedule due Tuesday, January 27 @ 11:00 PM
Intermediate Schedule due Tuesday, February 3 @ 11:00 PM

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 some basic functionality for manipulating DNA and RNA sequences.

Implementation:

You will implement sequences using linked lists, storing one letter of the sequence per linked list node. You may implement either a singly linked list or a doubly linked list, whichever you prefer. You may implement your linked list using the code from the textbook (source is available from the CS2604 website) or you may write your list code from scratch.

In addition to the linked lists of sequences, you will maintain a "sequence array" which stores the various sequences. Commands that manipulate sequences will refer directly to entries in the sequence array. The sequence array will store the sequence type (RNA or DNA) and a pointer or other form of access to the linked list that stores the sequence itself. The type field should use an enumerated type variable, and you should also have an enumeration value to recognize that a given position in the sequence array is unused.

You are required to implement a freelist for this project (see p. 114-117 in the textbook). All allocation and deallocation of linked-list nodes will go through the freelist functions. Many of the commands that you will process require you to create and delete linked-list nodes. You must be careful that you do not "lose" any of these nodes during this process. Losing access to a linked list node is a form of memory leak, and will be considered a major error in your program for grading purposes. You must also maintain counters for the number of times that a new linked list item has ever been created, and the number of links currently on the freelist.

All indexing (both for the sequence array and for positions in a sequence) will begin with position zero.

The primary design consideration for this project will be the interface between the list class and its client. If you reuse the book code, it is acceptable to alter or add some list methods. Regardless of whether you reuse the book code or write your own, it is mandatory that the list class remain application independent. No bioinformatics-related code should be part of the list class. All such code should be in some class above the level of the list class.

Input and Output:

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

`bio1 <array-size> <command-file>`

The name of the program is `biol`. Parameter `array-size` is the size of the sequence array, and `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. These include attempts to access out-of-bounds positions in the sequence array or in a sequence. 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 *pos type sequence*

Insert *sequence* to position *pos* in the sequence array. *type* will be either DNA or RNA. You must check that *sequence* contains only appropriate letters for its type, if not the insert operation is in error and no change should be made to the sequence array. If there is already a sequence at *pos* and if *sequence* is syntactically correct, then the new sequence replaces the old one at that position. It is acceptable that *sequence* be null (contain no characters) in which case a null sequence will be stored at *pos*. Note that a null sequence in a sequence array slot is different from an empty slot.

remove *pos*

Remove the sequence at position *pos* in the sequence array (returning the linked-list nodes to the free list). Be sure to set the type field to indicate that this position is now empty. If there is no sequence at *pos*, output a suitable message.

print

Print out all sequences in the sequence array. Indicate for each sequence its position within the sequence array and the type of that sequence (RNA or DNA). Don't print anything for slots in the sequence array that are empty. Finally, print out the number of times that a linked list element has been created so far by your program (this is done when the freelist is exhausted), and the number of linked list elements that are currently on the freelist.

print *pos*

Print the sequence and type at position *pos* in the sequence array. If there is no sequence in that position, print a suitable message.

clip *pos start end*

Replace the sequence at position *pos* with a clipped version of the sequence. The clipped version is that part of the sequence beginning at character *start* and ending with character *end*. It is an error if *start* has a value less than zero, or if *start* or *end* are beyond the end of the sequence. A clip command with such an error should make no alteration to the sequence. If there is no sequence at this slot, output a suitable message. If the value for *end* is less than the value for *start* then the result should a sequence containing no characters.

clip *pos start*

Replace the sequence at position *pos* with a clipped version of the sequence. The clipped version is that part of the sequence beginning at character *start* and continuing to the end of the original sequence. It is an error if *start* is less than zero or beyond the end of the sequence, and a clip command with an error should make no alteration to the sequence. If there is no sequence at this slot, output a suitable message.

copy *pos*1 *pos*2

Copy the sequence in position *pos*1 to *pos*2. If there is no sequence at *pos*1, output a suitable message and do not modify the sequence at *pos*2.

swap *pos*1 *start*1 *pos*2 *start*2

Swap the tails of the sequences at positions *pos*1 and *pos*2. The tail for *pos*1 begins at character *start*1 and the tail for *pos*2 begins at character *start*2. It is an error if the value of the start position is greater than the length of the sequence or less than zero. If the length of a sequence is n, the start position may be n, meaning that the tail of the other sequence is appended (i.e., a tail of null length is being swapped). The swap operation should be reported as an error if the two sequences are not of the same type, or if one of the slots does not contain a sequence. In either case, no change should be made to the sequences.

overlap *pos*1 *pos*2

Determine the position of maximum overlap between the sequences at positions *pos*1 and *pos*2. The maximum overlap is the position such that the maximum number of characters match in the two sequences, where the overlapping characters must be a suffix of the sequence at *pos*1 and a prefix of the sequence at *pos*2. Print out the position of overlap and the overlapping subsequence. If there is no such overlap, print a suitable message. If the sequences are not of the same type, then do not check for overlap, simply report that there is no overlap due to mixed types being compared. If either slot does not contain a sequence, then print a suitable message and do not check for overlap.

transcribe *pos*1

Transcription converts a DNA sequence to an RNA sequence. It is an error to perform the transcribe operation on an RNA sequence. To transcribe a DNA sequence, change its type field to RNA, convert any occurrences of T to U, complement all the letters in the sequence, and reverse the sequence. Letters A and U are complements of each other, and letters C and G are complements of each other. If the slot is empty, then print a suitable message.

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 should 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. You won't receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose 10 points from the project grade.

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. The text of the pledge will also be posted online.

```
//  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 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.