Computer Science I Linked Lists — CSCI-141 Homework

1 Problem

A single strand of DNA contains a sequence of nucleobases: cytosine (C), guanine (G), adenine (A), and thymine (T). DNA mutations manifest in many forms, including:

- substitution: a single base is replaced with another
- insertion: a sequence of bases is inserted consecutively within an existing strand
- deletion: a portion of an existing strand is removed
- duplication: a portion of an existing strand is repeated immediately following itself

After taking a course in genetics and learning about different DNA mutations, you decide that it might be advantageous when modeling these mutations to represent a single DNA strand as a linked list. DNA mutations may involve substantial insertions and deletions at various locations throughout the DNA strand, modifications that are handled well by a linked list data structure.

Your task is twofold: first, you will augment the singly-linked list from lecture to provide additional functionality to handle insertion of one list into another, as well as deletion or copying of a segment of a list. Secondly, you will write a program that provides some helper functions for a DNA application. Specifically, it can convert back and forth between a string of DNA bases (e.g. "ATCTTAGC"), and a linked list containing those same bases. It can also test whether two single strands of DNA form a base pairing (following the rule that A must pair with T, and G must pair with C).

You are provided with a complete test program that you can use to confirm that you have *correctly* and *efficiently* implemented the required functionality.

1.1 Task 1: Augmenting Linked List Functionality

Download the `slList_stu.py` file that can be found via the same webpage link as this assignment write-up. This file provides a partial implementation of the three functions you are responsible for implementing. Complete the following three functions in the `slList_stu.py` file. Add your name in the file as an author. Pay close attention to how each function must behave for various input scenarios. Also think carefully about how each function can be implemented *efficiently*. There are many ways to implement these functions. Some are *much* slower than others.

1. `insertListAt( lst1, idx, lst2 )`

   Parameters:
   - `lst1`: the first list into which the second list is inserted.

- `idx`: the index before which the insertion occurs (similar to `insertAt` function).
- `lst2`: the second list, which is inserted into the first.

Return value: there is no explicit return value (it returns `None`). However, `lst1` is modified to contain the elements from `lst2`. Also, `lst2` is cleared, as it should no longer exist as an independent list.

Behavior:
- if `idx` is out of range, the function should `raise` an `IndexError`.
- The entirety of `lst2` is inserted just before the index specified. For example, suppose `lst1` represents the DNA sequence "ATCG", and `lst2` represents the DNA sequence "AGCCA". If we call `insertListAt( lst1, 2, lst2 )`, upon completion `lst1` will represent the sequence "ATAGCCACG".
- Either or both of the input lists may be empty. If `lst2` is empty, `lst1` is unchanged. If `lst1` is empty, the only valid insertion index is 0.
- After insertion, `lst2` must be cleared.

2. `deleteSegment( lst, idx, segSize )`

Parameters:
- `lst`: the list from which a segment of elements will be deleted.
- `idx`: the index at which deletion begins.
- `segSize`: the number of elements to be deleted.

Return value: there is no explicit return value (it returns `None`). However, `lst` is modified to reflect the deletion of the requested elements.

Behavior:
- if `idx` is out of range, the function should `raise` an `IndexError`. An empty list generates this error, because there is no valid index from which to delete.
- if `segSize` is less than 0, the function should `raise` a `ValueError`. A `segSize` equal to 0 is valid, and results in `lst` unchanged.
- if the requested segment to delete extends beyond the end of the list, the function should `raise` an `IndexError`.
- The list cursor should be set to `None`.

3. `copySegment( lst, idx, segSize )`

Parameters:
- `lst`: the list from which a segment of elements will be copied.
- `idx`: the index at which the segment to be duplicated begins.
- `segSize`: the number of elements to be duplicated.

Return value: there is no explicit return value (it returns `None`). However, `lst` is modified to reflect the duplication of the requested elements.

Behavior:

- if `idx` is out of range, the function should `raise` an `IndexError`. An empty list generates this error, because there is no valid index from which to begin copying.
- if `segSize` is less than 0, the function should `raise` a `ValueError`. A `segSize` equal to 0 is valid, and results in `lst` unchanged.
- if the requested segment to copy extends beyond the end of the list, the function should `raise` an `IndexError`.
- The copied segment should be placed in the sequence immediately following its original occurrence. For example, suppose `lst` represents the DNA sequence "TGTCAGCG". If we call `copySegment( lst, 2, 3 )`, upon completion `lst` will represent the sequence "TGTCATCAGCG".

1.2 Task 2: DNA Helper Functionality

Download the `dna_stu.py` file that can be found via the same webpage link as this assignment write-up. This file provides a partial implementation of the three functions you are responsible for implementing. Complete the following three functions in the `dna_stu.py` file. Add your name in the file as an author.

1. `constructDnaList( dnaString )`

   Parameters:
   - `dnaString`: a string of characters corresponding to DNA bases.

   Return value: an `SlList` object representing the input DNA sequence. Each character of the input string is represented as a node in the list.

   Behavior:
   - In the case that the input string is empty, an `SlList` object is still created and returned. The underlying list is empty.
   - The k^{th} character in the input string must be represented by the k^{th} node in the list.

2. `convertDnaListToString( dnaList )`

   Parameters:
   - `dnaList`: a linked list in which each node contains as data a character representing a base of a DNA sequence.

   Return value: a string.

   Behavior:
   - In the case that the input list is empty, the empty string is returned.
   - The data from the k^{th} node in the input list must correspond to the k^{th} character of the returned string.

3. `isPairing( lst1, lst2 )`

   Parameters:
   - `lst1`: a first linked list with data nodes representing a DNA sequence.

- `lst2`: a second linked list with data nodes representing a DNA sequence.

Return value: boolean `True` or `False`.

Behavior:

- This function does an element by element comparison of the two lists. To be a pairing (and return `True`), the two lists must be equal length, and the elements at each index must be a valid DNA base pairing. If the lists are different length, or if there is any index at which the corresponding elements are not a valid pair, the function returns `False`.
  Valid base pairings are A with T and G with C. Note that the valid pairings are symmetric. That is, an A in the first list can be paired with a T in the second list. Equally valid is a T in the first list paired with an A in the second list.

2 Testing

Download the provided tester file, `dna_tester.py`, that can be found via the same webpage link as this assignment write-up. **This file should not be modified.** Use this test file to confirm that you have correctly implemented the functions, and that you have implemented the linked list functions efficiently. Your grade will be based on your performance against these tests (we will use this exact test file).

Note that you'll need to complete the `dna_stu.py` functions before you'll be able to use the main tests for the `slList_stu.py` functions, as these tests call the `dna_stu.py` functions.

The final tests are *stress* tests that evaluate the performance of your functions for large data sizes. In order to receive full credit for your `slList_stu.py` functions, you must implement them efficiently. Think carefully (and perhaps visually) about what each function needs to do.

As an example, deleting 100 nodes from the middle of a list by making 100 independent calls to `pop` one node at a time is not efficient. Make sure you understand why such an approach is inefficient.

3 Submission

Take your two files, `slList_stu.py` and `dna_stu.py`, and zip them into a file named `dna.zip`. Use zip format only. Do not use another compression mechanism (such as .rar or .7z). Submit the zip file to the MyCourses dropbox before the deadline.

3.1 Grading

- 30% implementation of `dna.py` functionality.
- 70% implementation of `myList.py` functionality (40% correctness, 30% efficiency).

You do not need to add any documentation beyond what is already provided.