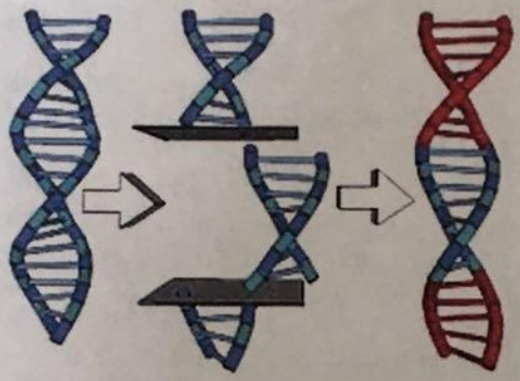
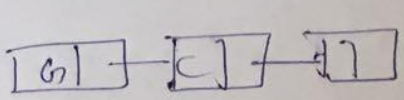
Parvathi Nair.

Computational Problem Solving CSCI-603
You are a Genetic Engineer Lab 6



In this lab, we are consider the operation of gene splicing, snipping and joining. Specifically, we will represent a strand of DNA as a linked list. Each node in the list represents one nucleotide by storing a character `A`, `C`, `G`, or `T`. Since we are interested in making efficient changes to potentially very long lists, it is important to consider the time complexity of all the operations on our DNA strands.



Problem Solving

1. Consider the following two DNA strands: `GCA` and `CTT`. Draw the list data structures that you would use to represent these strands, including labeling any additional data members of the list itself, such as the head.

2. Now, we want to join the two strands to create the gene `GCACTT`. In general, a join involves placing one gene directly after the other. Write code or pseudocode for the join operation, based on the structure from question 1. Your code should operate in $O(1)$ time!

3. Now, we would like to cut out various pieces of a gene and replace them with new genetic material (as in the picture at the beginning of this writeup). As a first step toward this process, consider trying to find whether a given DNA string is contained within another.
   (a) Write pseudocode for a function `find(self,s)` that is a member of your list class and takes in a string `s`, and returns whether the string is represented within that list.
   (b) As an example, explain what happens when you call `find("CT")` on the list created in the previous question (containing `GCACTT`).
   (c) What is the time complexity of your `find` function, with respect to the length of the list n and/or the length of the string k?

4. Finally, thorough testing is critical for any data structure development, as there will be many special cases that your code will have to consider. One of the functions that you will have to write for this lab will be the `splice` function. The signature of this function is `splice(self,ind,other)`, and inserts the list passed in as `other` into this list, at the position given by `ind`. **Without writing this code, or even necessarily worrying about how it might be done**, write a set of test cases for the `splice` function that you think will exercise all special cases of the function.

   Make a table of your test cases — for each test case, write the contents of the list that you are calling `splice` on, the index and other list that you are passing in, and the expected result.

Implementation

For the implementation, you will write a class called `DNAList`. You may use the official course version of a linked list for reference, but for the best learning experience, you should code your list class from scratch!

Your list should implement the following functions:

- `__init__(self,gene='')` This function creates a new list. The `gene` argument is an optional argument for which a default (empty string) value is provided. The list should be created such that it represents the DNA string provided as an argument. This function should run in time $O(k)$ where k is the length of the `gene` string.
- `append(self,item)` This function takes in a single character and extends the list with a node that represents this character. This function should run in $O(1)$ time.
- `join(self,other)` This function takes in another `DNAList` and adds it to the end of the list. This function should run in $O(1)$ time.
- `splice(self,ind,other)` This function takes in an integer `ind` representing an index into the list, and another `DNAList`. It should then insert the `other` list into the list immediately after the `ind`'th character of this list. This function should run in $O(n)$ time, where n is the length of the list and k is the length of the other list (that's correct, k should not appear in the time complexity of this function).
- `snip(self,i1,i2)` This function removes a portion of the gene (list) as specified by the integers `i1` and `i2`. Specifically, counting from the beginning of the list as 0, the list should no longer contain all nodes from the node at position `i1` (inclusive) up to but not including position `i2`. This function should run in $O(n)$ time, where n is the length of the list, and for full credit, should visit each node in the list at most once.
- `replace(self,repstr,other)` This function should find the *string* `repstr` as a subsequence of the list and replace it with the *list* given by `other`. This function should run in $O(n)$ time, where n is the length of the list, and should visit each node in the list at most `len(repstr)` times.
- `copy(self)` This function returns a new list with the same contents as the list called upon. It should run in $O(n)$ time, where n is the length of the list.
- `__str__(self)` This should simply return a string with the contents of the nodes all together, such as `GCACTT`. This function should run in $O(n)$ time, where n is the length of the list.
- You may implement any additional 'helper' functions that you find useful for providing the above functionality. Make sure you clearly document the purpose and behavior of these functions.