IT-309 - Assignment 4 – Linked List Operations and Maintenance

Assignment Given: 09/29/2020
Assignment Due: 10/13/2020, 11:59 pm

In this assignment you are to modify the SinglyLinkedList ADT class provided to you by (1) adding two methods to the class and then (2) by reading and processing a stream of list transactions that add, insert, and remove from a linked list created as an object of the SinglyLinkedList (with your added methods), hereafter referred to using the abbreviation SLL.

The methods to be added:

1. **insertAfter** takes two parameters, an integer index (0 and higher) and a string value, creates a new node object with the string value as the new node's element, and inserts that node object after the ith element of the SLL. For the purpose of this method the items of the SLL are numbered starting with 1, not 0. If the value of the index is greater than the number of items in the SLL the new node is to be inserted after the last item (after the rear or tail) and becomes the new rear or tail.

2. **insertElt** takes three parameters: (1) a string value used as the element for a newly created node (NN) object that is to be inserted into the SLL, (2) an indicator ("B" or "A") specifying whether the NN is to be inserted before or after a specified SLL node, and (3) a string value of the element of a node (EN) already in the SLL (maybe) before or after which the NN will be inserted. If the string value in parameter (3) is not found in the SLL, the NN is added to the end (tail) of the list.

Both methods will take in the string parameter and create a new node (NN). They will then execute their code to insert the NN into the SLL. Cases that need to be handled include (1) inserting into an empty SLL, (2) inserting as the first or head element (index 0 for insertAfter, and 'B'efore the named head element in insertElt), (3) inserting into the interior of the list (between two existing nodes), and (4) inserting as the last/tail node (when the index exceeds the size for insertAfter, and when the NN element is not found in the SLL for insertElt).

After the methods are created they are to be tested using a transaction stream to be provided as a list of strings for input, similar to the way the expressions were provided in a previous assignment. These could also easily be provided in a text file that could be read and processed in the same way. The transactions consist of comma-separated elements: (1) a transaction code indicating which SLL ADT operation is to be executed, and (2) a value(s) to use when executing an add or insert, and no value when the operation is 'remove'.

The transactions are listed in this table:

Transaction code	Parameter(s)	Description	Example
A	<string value>	Adds <string value to rear/tail> of the SLL	A,Washington
R	- no parameter -	Removes the head of the SLL	R
IA	1. <string value> 2. integer index ***i***	Creates and inserts a NN with <string value> as element immediately after SLL element # ***i***	IA, 4, Jackson
IE	1. <string1> 2. "B" or "A" 3. <string2>	Creates and inserts a NN with element <string1> immediately before or after an existing SLL node whose element is <string2>	IE,Jefferson,A,Adams
SL	- no parameter -	Displays all SLL elements from head to tail	SL

Here's a sample transaction stream, assuming an SLL object called "SLL1" has been created and the above methods added:

Transaction…	**Your code executes…**	**Effect**
A,Washington	SLL1.add('Washington')	Adds 'Washington' to an empty SLL
A,Adams	SLL1.add('Adams')	Adds 'Adams' imm. after 'Washington'
A,Madison	SLL1.add('Madison')	Adds 'Madison' imm. After 'Adams'
IE,Jefferson,B,Madison	SLL1.insertElt('Jefferson,'B','Madison')	Inserts 'Jefferson' before 'Madison'
IE,Monroe,A,Madison	SLL1.insertElt('Monroe','A','Madison')	Inserts 'Monroe' after 'Madison'
SL	SLL1.showList()	***Displays:*** Washington Adams Jefferson Madison Monroe
R	SLL1.remove()	Removes 'Washington' from list
IA,4,'Quincy Adams'	SLL1.insertAfter(4,'Quincy Adams')	Adds 'Quincy Adams' after Monroe
SL	SLL1.showList()	***Displays:*** Adams Jefferson Madison Monroe QuincyAdams

I will provide a transaction stream to be used to test your code and methods. However, correctly executing all the transactions on that list does not guarantee your code is 100% correct since the provided list may not have all cases. Be sure to try out your own transaction stream(s) as well and test that all the other SLL methods continue working as expected after you add the two methods above. The transaction stream will be a list of strings in .py format, each string consisting of a transaction as shown in the left column of the table above.

You may use the SinglyLinkedList code provided with the assignment, but must add your own code for the **insertAfter** and **insertEFT** methods and the transaction stream processing code. **The most difficult part of the assignment is making sure the methods correctly manage the links when adding/inserting nodes.**

What and where to submit:

The SLL class with added methods, transaction stream processing code (with transaction code copied in), and output produced by the code is to be submitted through Blackboard as a Jupyter notebook. Include at least one Markdown cell with identifying information at the top – at minimum the assignment number and subject, your name, and date.

In another Markdown cell near the end, include any observations you have after creating and testing the code in this assignment. Was handling the linked structures particularly difficult or tricky? Would you prefer to implement the linked list using an array as the underlying structure, and why or why not? Was one of the methods more difficult to code than the other? Is the SLL class suitable for implementing a Queue ADT?

How the assignment will be assessed

The Jupyter notebook will be visually inspected and executed using the test transaction stream and

additional test transactions.

Item	Assessment Description	Max Value
Python Code in a Jupyter notebook	All code submitted in a Jupyter notebook with some annotation/documentation in a Markdown cell and some relevant observations in another Markdown cell.	5
insertAfter method	Method is included and for full credit correctly processes all inserts. GTA will allocate partial credit where some but not all insertions are correctly handled.	15
insertElt method	Method is included and for full credit correctly processes all inserts. GTA will allocate partial credit where some but not all insertions are correctly handled.	15
Transaction stream processing	Code correctly handles all transactions in the stream. GTA allocates partial credit for less than 100% correct processing	5
Total		**40**