NYU SHANGHAI 上海纽约大学

CSCI-SHU 210 Data Structures - 100 Points

Homework Assignment 5 - Linked Lists

SinglyLinkedList - Complete Until October 20, 11:59 PM

Problem 1 – Tail Swap - 10 Points

Implement the member function *tailswap(self),* that swaps the LinkedList's last node with the list's second last node in the given SinglyLinkedList implementation. Throw the exception $Exception("Short")$ if the LinkedList is too short for the operation. You have to maintain the LinkedList integrity when swapping the nodes.

Requirements

- You are not allowed to create a new node.
- You are not allowed just to swap the node values (the node data). **(-10 Points)**
- The time complexity requirement of this method is O(n). **(5 Points)**
- The space complexity requirement of this method is O(1). **(5 Points)**

Example

```
ls1 = SingleLinkedList()
ls1.insertAtFirst(5)
ls1.insertAtFirst(10)
ls1.insertAtFirst(22)
ls1.insertAtFirst(35)

print(ls1)  # Should print: 35-->22-->10-->5-->None
ls1.tailswap()
print(ls1)  # Should print: 35-->22-->5-->10-->None
```

Problem 2 - Swap Kth-Item - 10 Points

Write the function *swapWithKth(self, k)* to swap the kth-node with the first node of the SinglyLinkedList. You are not allowed just to swap node elements. You can assume that you always have at least one node in the SinglyLinkedList.

Example

```
ls1 = SingleLinkedList()
ls1.insertAtFirst(3)
ls1.insertAtFirst(2)
ls1.insertAtFirst(1)

print(ls1)  # Should print: 1-->2-->3-->None
ls1.swapWithKth(2)
print(ls1)  # Should print: 2-->1-->3-->None
```

Problem 3 - FavSinglyLinkedList - 20 Points

A bookmark tracker records webpages associated with the number of clicks on each webpage in a SinglyLinkedList. The idea is to keep the most frequently accessed items towards the head of the list, using a move-to-front heuristic. (Textbook 7.6.2)

Implement the class FavSinglyLinkedList, using a SinglyLinkedList as a data storage. The elements stored in each list node are Python lists in the format of [url, count], where *url* is a webpage string of, and *count* is an integer of how many times a website has been clicked.

You have to implement the function *access(self, url)*. When accessing an item, we first look for the url in the existing SinglyLinkedList. If it does not exist, we add it to the tail of the SinglyLinkedList, and set the count to 1. If it exists, we increase the count by 1. The general idea is that we use the most recently accessed item as a heuristic for the most frequently used item.

Requirement

- You can write additional methods in the class FavSinglyLinkedList, if required.

Problem 4 - K-Least Item Generator - 10 Points

This question extends the FavSinglyLinkedList of Question 3.

In order to find the k-least favorite items, we iterate through the singly linked list for k times.
Write the generator *leastk(self, k)* that yields the URLs of the k-least items one by one (in the same order of which in the Linked list for those with the same URL). Notice, the FavSinglyLinkedList you created should not be altered.

NYU SHANGHAI 上海纽约大学

DoublyLinkedList - Complete Until October 27, 11:59 PM

Problem 5 - Shuffle - 25 Points

Implement the method *shuffle(self)* for a DoublyLinkedList containing 2n elements (where n > 1). This method alters the sequence of elements according to a specific shuffling process.

The shuffling process involves dividing a list L into two parts: L1 (comprising the first half of L) and L2 (comprising the second half of L). These two lists are then recombined into one by taking elements in an alternating pattern: starting with the first element in L1, followed by the last element in L2, then the second element in L1, followed by the second last element in L2, and so forth.

Requirements

- The operation has to be performed in place.
- You are not allowed to use any additional data structures like Python lists.
- You are not allowed to alter the values stored in any nodes.
- You can not delete original nodes or insert new nodes.
- The original nodes should be rearranged.

Example

```
ls1 = DoublyLinkedList()
ls1.insertAtFirst(4)
ls1.insertAtFirst(3)
ls1.insertAtFirst(2)
ls1.insertAtFirst(1)

print(ls1)  # Should print:
Header<-->1<-->2<-->3<-->4<-->Trailer
ls1.shuffle()
print(ls1)  # Should print:
Header<-->1<-->4<-->2<-->3<-->Trailer
```

Problem 6 – Pivot Linked List - 25 Points

Write a function partition_list(self, pivot) that takes a pivot value as input and modifies the list so that all nodes with values less than the pivot come before all nodes with values greater than or equal to the pivot. The function should preserve the relative order of nodes with values less than the pivot and the relative order of nodes with values greater than or equal to the pivot.

Requirements

- Your function has to be at worst in **O(n)** time complexity.
- Your function has to be at worst in **O(n)** space complexity.
- Your function has to work in place.

Example

Given the following doubly linked list:

```
Header<-->5<-->1<-->6<-->2<-->7<-->3<-->8<-->4<-->Trailer
```

The result for *partition_list(self, 5)* is:

```
Header<-->1<-->2<-->3<-->4<-->5<-->6<-->7<-->8<-->Trailer
```