HackerRank

Data Structures

Author	: Avinash Sorab
Date Created	: 25th Aug 2019
Language Used	: Python 3
Platform	: HackerRank

Contents

1. Print the elements of a linked list

The task is to print all the elements of a linked list.

Only one call to this function is required and the head of the linked list is passed.

Keep printing the data of the node and keep iterating till the next node is null.

```
def printLinkedList(head):
    node = head
    while(node != None):
        print(node.data)
        node = node.next
```

2. Insert a node at the head of a linked list

The task is to insert a node at the head of a linked list.

A loop needs to be run fetching inputs one by one and calling the insert function.

Every time the function is called, the head of the node is passed. The head after the insert is obviously updated to the new data and this needs to be returned to the main (calling) function so that the function gets the updated head everytime it's being called.

Inside the insert function, a node is created for the data it receives.

If the head is null, which is for the first call (empty linked list), The created node is simply returned, indicating that it is the first element.

Else, the created node's next is set to the head. This way new head will be the newly created node and old head continues to link the other elements.

The created node is returned in the end.

```
def insertNodeAtHead(head, data):
    node = SinglyLinkedListNode(data)
    if(head):
        node.next = head
    return node
```

3. Insert a node at a specific position in a linked list

The task is to insert a node in any specified position in a linked list.

The function is given the data and the position at which the data node needs to be inserted.

The function needs to return the head node everytime for the calling function. This helps the calling function to call the insert function by sending it an updated linked list everytime.

The function does the following functionality, it creates the node out of the data given to it.

If the head is None, which means the linked list is empty, the node just created is sent as a head, since that's the first element of the linked list.

Else, The existing linked list is iterated using a new inode variable(iterable node)

We need to keep track of how many nodes are we jumping, using a count variable, when this count equals position, there we need to insert the node created earlier.

To insert, first we need to store the next node in a temporary variable, replace this node as the next node and update the next node's next as temp.

Finally return the head immediately to come out of the while loop.

```
def insertNodeAtPosition(head, data, position):
    node = SinglyLinkedListNode(data)
    if(head == None):
        return node
    else:
        count = 0
        inode = head
        while(inode != None):
            count+=1
            if(count == position):
                temp = inode.next
                inode.next = node
                node.next = temp
                return head
            inode = inode.next
```

4. Insert a node at the tail of a linked list

The task is to insert a node at the tail of a linked list.

The function receives head and data every time it is called, so it is important to return the head of the linked list from the function so that the next time function is called, the function is going to receive an updated linked list (head).

Create a node out of the data sent.

If the head is None, return the node created since it is the only element in the linked list.

Else, using the head node as the iterative node, loop through the linked list to reach the tail and the way to ensure the reach of tail node is to check if the next node is null from being at the current node.

If the next node is null, replace the next node with the node created earlier.

Immediately returning the head from there is very important other wise it keeps adding the same node over and over and your code will never actually terminate/come out of while loop.

```
def insertNodeAtTail(head, data):
    node = SinglyLinkedListNode(data)

    if(head == None):
        return node
    else:
        inode = head
        while(inode != None):
            if(inode.next != None):
                inode = inode.next
            else:
                inode.next = node
                return head
```

5. Delete a Node

The task is to delete a node whose position is given.

The function accepts the head node and the position or index of the node to be deleted.

First, if the linked list is empty/ head is None, head is returned as is since there's nothing to delete in that anyway.

Next, we have to check if the position is 0, which means the node to be deleted is the head of the linked list. In that case simply assign the head of the node as the next node (head.next).

Else, while loop is written to iterate through the linked list till the position is reached. This is made sure using a counter.

After the position is reached, we have to also check whether the position is the last element or not. This can be verified by check if the next element from the last element is None.

If it's the last element to be deleted, it's node value is assigned to None.

If not, the next of current inode is assigned with the next of the next of the current node. This is actually how you delete a node, by ignoring it in the linked list, in other words by de linking the node from the linked list.

Return the head of the node in the if statement itself to come out of the while loop, since the work required is done.

```
def deleteNode(head, position):
    if(head == None):
        return head
    else:
        if(position == 0):
            head = head.next
            return head
        else:
            inode = head
            count = 0
            while(inode != None):
                count+=1
                if(count == position):
                    if(inode.next != None):
                        inode.next = inode.next.next
                    else:
                        inode.next = None
                    return head
                inode = inode.next
```

6. Reverse a linked list

The task is to reverse the elements of a linked list.

This can be achieved in 2 ways.

- Recursive
- Iterative

Here iterative method is followed for the ease of understanding.

$$A- > B- > C- > D- > None$$

needs to be

$$None < -A < -B < -C < -D$$

Initially all we care about achieving is to reverse the linkages. i.e., $A->B$ should be $A<-B$

The process involves, iterating through each nodes.

While being on each node,

- store the pointer to next node in a temporary variable
- assign the pointer to the next node as previous node (None for head node)
- hop on to the next node and update your previous and current nodes respectively.
- next node hopping couldn't have been done if we didn't store the pointer to next node in a temporary variable. Because the pointer gets updated in the second step.

```
def reverse(head):
    prev = None
    cur = head
    while(cur):
        nxt = cur.next
        cur.next = prev
        prev = cur
        cur = nxt
    head = prev
    return head
```

7. Compare two linked lists

The task is to compare two singly linked lists and they should be equal lengthwise and also all the elements should be equal.

Start your iteration in a while loop, enter the while loop only if both the iterative nodes are not null.

Return 0 immediately inside the loop if the data is unequal.

Return 0 if only one of them nodes have next element as None. This indicates unequal length of linked lists.

Increment both the nodes to its next nodes.

Return 1 outside the while loop, since it indicates that the elements are equal in the linked lists if the control reaches out of while loop.

```
def compare_lists(head1, head2):
```

```
    inode1 = head1
    inode2 = head2
    while(inode1 and inode2):
        if(inode1.data != inode2.data):
            return 0
        if((inode1.next == None and inode2.next != None) or (inode1.next != None and inode2.
            return 0
        inode1 = inode1.next
        inode2 = inode2.next
    return 1
```

8. Print in reverse

The task is to print a singly linked list in reverse order.

This can be very easily done using Recursion technique.

The function is called and is given the head node of the linked list.

Inside this function, make a recursive call (calling the same function inside the function) till you reach the end of the linked list.

At the end of the linked list when the node is none, print the node value and come out of the innermost recursive function.

You will actually come to the second innermost recursive function where head is now pointing to the last but 1 element. Print it and come out of the function.

This way you keep printing the elements in the reverse order using Recursion.

```
def reversePrint(head):
    if(head != None):
        reversePrint(head.next)
        print(head.data)
```

9. Get node value

The task is to get the value of a node whose position, let's call it t, from the tail is known.

The trick here is to have two iterable linked list nodes, one that goes till the end of the node and the second one that starts its increment after the first one hops t number of times.

By the time the first node reaches the end, the second node will then be lagging by t. That way we will get the data of the node which is t nodes back from the tail of the linked list.

```
def getNode(head, positionFromTail):
    count = 0
    node = head
    while(head):
        if(count > positionFromTail):
            node = node.next
        count+=1
        head = head.next
    return node.data
```

10. Delete duplicate-value nodes from a sorted linked list

The task is to delete repeated numbers from a sorted linked list.

It is so tempting to use 2 different nodes to keep track and check duplicates, but it can be achieved using only one iterable node.

Start with the head and loop till the end of the linked list.

While doing so, check if the adjacent nodes data are equal.

If so, delete the leading repeated number, you can do so by unlinking it from the chain.

Otherwise increment the node and check the same condition for the next node and so on.

Finally return the head of the linked list to the calling function.

```
def removeDuplicates(head):
    inode = head
    while(inode.next):
        if(inode.data == inode.next.data):
            inode.next = inode.next.next
        else:
            inode = inode.next
    return head
```

11. Cycle Detection

The task is to detect if the linked list is cycled, i.e., if the linked list loop never ends. Return 0 if the linked list is open chained, 1 if it has cycle or close chained.

You'll have to run 2 iterators here, once goes fast (2 hops at a time) and one goes slow (1 hop at a time).

If at some point, both the iterators are pointing to the same node (not the data), then we are guaranteed to be in a cycle. Because the slower iterator will never be able to meet a faster iterator in an open chain linked list.

Important thing to remember while doing this problem is to compare the actual nodes and not the data

```
def has_cycle(head):
    if(head == None):
        return 0
    else:
        slow = head
        fast = head
        while(fast != None and fast.next != None):
            slow = slow.next
            fast = fast.next.next
            if(slow== fast):
                return 1
        return 0
```