
Merging Linked-Lists:

A brief explanation of the functionality of each of the functions and methods in the code for problem 1 is provided in this document.

Class *Node*:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

This class essentially defines the data container in the linked list. It stores the data value and by default points to *null* as its successor.

Class *Linked-List:*

```
class LinkedList:

    def __init__(self):
        self.head = Node('head')
        self.tail = self.head
```

This class defines a linked-list constructed by chaining the nodes defined by class *Node*. It has a *dummy* head which is created along with linked-list instantiation. Later, I delete this dummy head before I print the final output. An additional pointer is declared in this class (*tail*), which is used for fast insertion of new nodes.

Three methods are implemented for this class, as required by the question.

Method 1:

```
def insertNode(self, node):
    self.tail.next = node
    self.tail = node
```

This method uses the *tail* pointer and inserts a new node to the end of the current linked-list. *self.tail.next* is only useful for the insertion of the first element because *head* and *tail* pointers point to the same element. After that, the first statement is redundant because when *tail* is overwritten, its *next* will be overwritten as well.

Method 2:

```
def printList(self):
```

Simply traverses the linked-list and prints its contents until it reaches the last element where *tail.next = None*.

Method3:

```
def skipHead(self):
```

This is function mentioned in *Class Linked-List* section. It just skips the *dummy* head. This is done for clarity of demonstration. If not done, the output list printed in the terminal/file will be preceded by “dummy.”

Some auxiliary function:

```
def convertArrToLinkedList( arr ):

    llist = []
    for i in arr:
        temp = LinkedList()
        for j in i:
            temp.insertNode(Node(j))
        llist.append(temp)

    return llist
```

This function is just for my convenience since I had my lists hard coded in a 2D array. It converts the 2D array into an array of linked-lists. Again, the choice of array of linked-lists is for my convenience. It doesn't affect the correctness or generality of the solution. This only helps to create the binary tree (discussed later) easier.

Merging:

```
def merge(lists):
```

In this function, the sorted linked-lists are merged together to create a one universal sorted linked-list.

The following lines create a binary tree from the linked-lists. This is where putting the heads of the linked-lists in an array comes handy. As can be seen, by easily indexing the array to n/2, where n is the number of sorted linked-lists, I can break them into binary groups. Remember, *lists* is the array which stores the *head* node of the linked-lists.

```
    left = merge(lists[:int(n/2)])
   right= merge(lists[int(n/2):])
```

While merging the lists, *l_ptr* and *r_ptr* keep track of the last element of each of the tree branches that has been merged into the final sorted list.

```
    l_ptr = left.head.next    #keeps track of the last index inserted into the final list
    r_ptr = right.head.next   #keeps track of the last index inserted into the final list
```

The merging happens within the following lines:

```
        if(l_ptr.data < r_ptr.data):
            res.insertNode(l_ptr)
            l_ptr = l_ptr.next
        else:
            res.insertNode(r_ptr)
            r_ptr = r_ptr.next
```

The *if-else* statements simply check to find the smallest element out of the the two elements which *l_ptr* and *r_ptr* point to. The fact that the original linked-lists are sorted is the major benefit here. By just simply checking the head of each list, we are sure that element is the smallest in its corresponding list. Once we choose an element and add it to the final sorted linked-list, we move the corresponding pointer to the next element in its list.

To make sure we check all the element while not going out of bound, the following line is performed:

```
while (l_ptr != None and r_ptr != None): #checks to not go out of bound
```

In case we checked all the elements in one list but there still exists some elements in the other array, the following pieces of code are executed.
It is good to note that the fact that the original lists are sorted comes handy in this case as well. Since all the elements are sorted, and considering that we only move forward if an element is the smallest in the remaining lists, then if all the elements in one array are completely exhausted and there exists elements in the other array, those remaining elements are definitely greater than or equal to the last element in the final sorted array. Thus, we can easily amend the remaining set to the final list.

```
    if (l_ptr != None):        #inserts the residual elements of the list to
the final list
        res.insertNode(l_ptr)

    if (r_ptr != None):        #inserts the residual elements of the list to
the final list
        res.insertNode(r_ptr)
```

Password Check Problem:

def strongPasswordChecker (s):

This function takes a string as input and performs the following operations to see if the input password is strong.

def checkLength(s):
Simply checks the length of the input and prints suggestions to add or delete a number of characters in order to meed the constraints.

def checkCharacters(s):
This function checks to see if the input password contains at least one lowercase, one uppercase, and one digit. If not, it will print a suggestion for each field that is missing.

def checkPattern(s):
This function checks to see if there is at least three same characters in a row. If yes, it will prompt the user to delete one of the repeating characters. If there are more than three the same characters in a row, the function does not differentiate. However, this is irrelevant because if there are more than three the same characters in a row, the function will prompt the user multiple time until the repeating character falls below three times in a row.