8.2 - Week 8 - Applied Practical

Easy and (not so) simple questions about Link Lists

Consider the following implementation of Nodes:

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

And answer the questions in the following slides!

LinkedLists Quiz

Question 1

What does the following piece of code do?

```
def fun1(head):
    if(head == None):
        return
    fun1(head.next)
    print(head.data, end = " ")
```

- ○ Prints the linked list
- ○ prints the linked list but in reverse
- ○ Prints alternate nodes of the linked list
- ○ prints the head's data

Question 2

What does the following piece of code do to a given linked list?

```
def fun2(head):

    if(head == None):
        return
    print(head.data, end = " ")

    if(head.next != None ):
        fun2(head.next.next)
```

- ○ Prints the linked list
- ○ Prints the linked list but in reverse
- ○ Prints alternate nodes of the linked list

- Prints the head's data

Question 3

Are you ready to see the implementation of the entire class with these functions now?

No response

Moving Elements in a Linked List

Write a function that moves the last node to the front in a given Singly Linked List.

Examples:

```
Input: 1->2->3->4->5
Output: 5->1->2->3->4

Input: 3->8->1->5->7->12
Output: 12->3->8->1->5->7
```

SPOILER -> Check for hints ahead ONLY if you are stuck

▶ Expand

Intersection of linked lists

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed.

Examples:

```
Input:
First linked list: 1->2->3->4->6
Second linked list be 2->4->6->8,
Output: 2->4->6.
The elements 2, 4, 6 are common in
both the list so they appear in the
intersection list.

Input:
First linked list: 1->2->3->4->5
Second linked list be 2->3->4,
Output: 2->3->4
The elements 2, 3, 4 are common in
both the list so they appear in the
intersection list.
```

SPOILER -> Check for hints ahead ONLY if you are stuck

▶ Expand

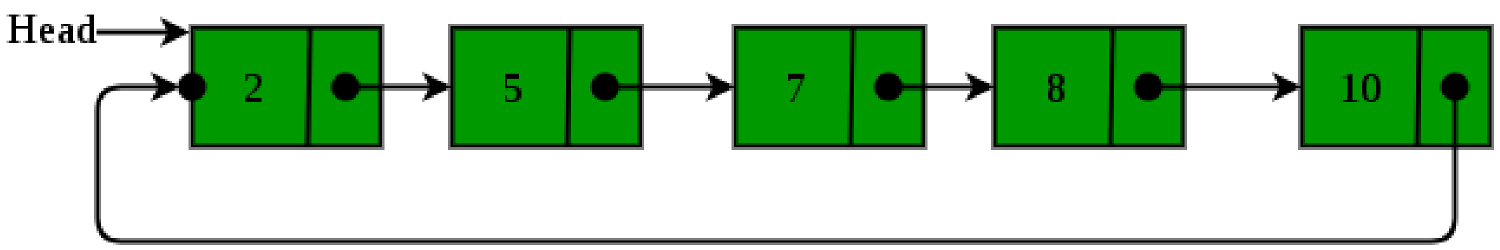
Danger Ahead (Not for the weak-hearted)

WARNING! DANGER! WARNING! DANGER! WARNING! DANGER! WARNING! DANGER!

Okay, so we are going to discuss a new ADT (which is not going to be taught or assessed) in the next slide. Proceed only if you have completed the Applied sessions and have no questions about them.

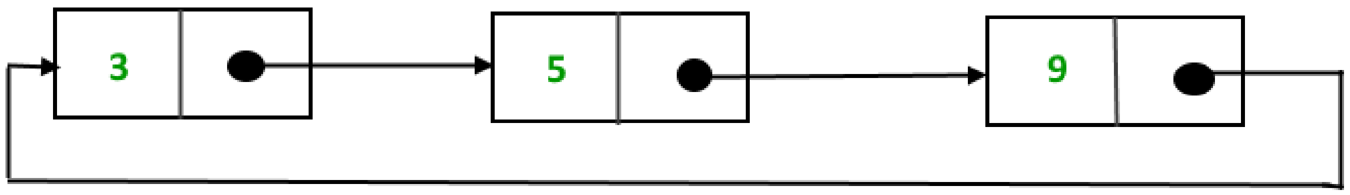
Introduction to Circular Linked Lists

*The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.*

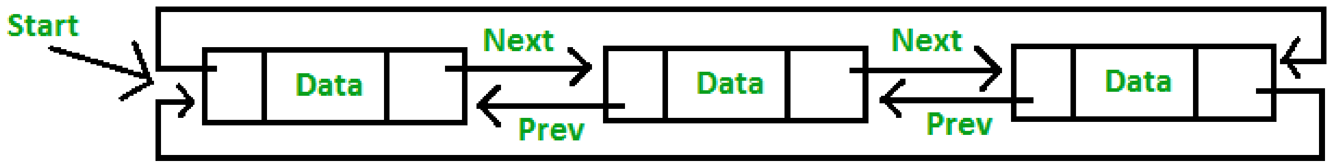


There are generally two types of circular linked lists:

- **Circular singly linked list:** In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



- **Circular Doubly linked list:** Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.



Note: We will be using the singly circular linked list to represent the working of the circular linked list.

Here is a basic implementation of this data type:

```
"""
This file contains a simple implementation of Circular Linked List.
The file also contains an implementation of the node object

Author: John Smith
Last Modified: 12/09/2022
"""

from typing import TypeVar, Generic

T = TypeVar('T')

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

class CircularLinkedList:
   def __init__(self):
      self.head = None
   def add_data(self, my_data):
      ptr_1 = Node(my_data)
      temp = self.head
      ptr_1.next = self.head

      if self.head is not None:
         while(temp.next != self.head):
            temp = temp.next
         temp.next = ptr_1
      else:
         ptr_1.next = ptr_1
      self.head = ptr_1

   def print_it(self):
      temp = self.head
      if self.head is not None:
         while(True):
            print("%d" %(temp.data)),
            temp = temp.next
            if (temp == self.head):
               break
my_list = CircularLinkedList()
print("Elements are added to the list ")
my_list.add_data (56)
my_list.add_data (78)
my_list.add_data (12)
print("The data is : ")
my_list.print_it()
```

Circular Linked Lists - Exercises

Now that you know how a CircularLinkedList works, copy the implementation given and add the following functionality:

- Add to empty
  Initially, when the list is empty, the *last* pointer will be NULL. After insertion, T is the last node, so the pointer *last* points to node T. And Node T is the first and the last node, so T points to itself.
- Add to beginning
  To insert a node at the beginning of the list, follow these steps:
  - Create a node, say T
  - Make T -> next = last -> next
  - last -> next = T
- Add to the end of the list
  To insert a node at the end of the list, follow these steps:
  - Create a node, say T
  - Make T -> next = last -> next
  - last -> next = T
  - last = T
- Add after a particular node
  To insert a node in between the two nodes, follow these steps:
  - Create a node, say T.
  - Search for the node after which T needs to be inserted, say that node is P.
  - Make T -> next = P -> next;
  - P -> next = T.
- Traversal of the list