DAY-7

Linked List in python

→ An ordered collection of objects.

→ differs from list in the way that they store elements in memory.

* While lists use contiguous memory block to store references to their data, *linked lists store references as part of their own elements.

Implementation of Singly Linked List.

The Node:-

→ where data is stored

→ each node holds a pointer, which is a reference to the next node in the linked list.

→ initialized with single datum and "pointer is set to "None" by default.

```
Class Node(object):
    def __init__(self, data=None, next_node=None):
        self.data = data
        self.next_node = next_node
    def get_data(self):
        return self.data
    def get_next(self):
        return self.next_node
    def set_next(self, new_next):
        self.next_node = new_next
```

Implementations :-

The head of the list :-

→ The first architectural piece of the linked list & the top node.

→ It has no nodes & head is set to "None"

*NOTE : Not neccessary.

→ The head argument will default to "None", if a node is not provided.

```
class Linkedlist(object):
    def __init__(self, head=None):
        self.head = head.
```

Insert :-

=> takes data → initializes a new node with given data → adds it to the list.

→ Can be inserted anywhere in the list.

* A simple way:

→ placing it at the head of the list and point the new node at the old head.

* Time complexity :- O(1) [constant]

```
def insert(self, data):
    new_node = Node(data)
    new_node.setnext(self.head)
    self.head = new_node
```

Size :-

→ Counts the node until reach the end, and returns the number of nodes found.

Time Complexity : O(n)

→ 'coz it has to visit every node in the list.

```
def size(self):
    current = self.head , count = 0
    while current:
        count += 1
        current = current.get_next()
    return count .
```

Search :-

→ Similar to size, but instead of traversing the whole list of nodes, it checks at each stop to see whether the current node has the requested data and returns the node holding that data.

* if not found, raises Value error.

Time complexity :- O(n) in the worst case.

```
def search(self, data):
    current = self.head
    found = false
    while current and found is False:
        if current.get_data() == data:
            found = True
        else:
            current = current.get_next()
    if current is None:
        raise ValueError("Data not found")
    return current
```

Delete :-

-> traverses the list in the same way that search does, but in addition to keeping track of current node, it also remembers the last node visited.

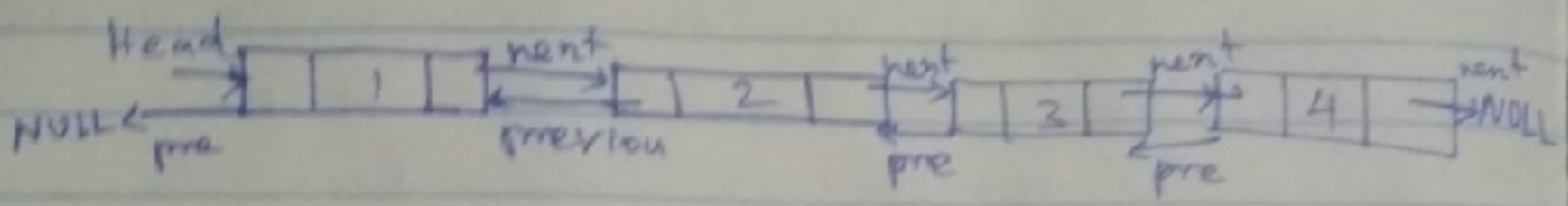
-> removes the node from chain by "leap frogging".

* Leap frogging :- when the delete method reaches the node to be deleted, it looks the last node visited (the 'previous' node), and resets its pointer to the next node which is next to that tobe deleted.

```
def delete(self,data):
    current = self.head
    previous = None
    found = False
while current and found is False:
    If current.get_data() == data:
        found = True
    else:
        previous = current
        current = current.get_next()
if current is None:
    raise ValueError("Not found")
if previous is None:
    self.head = current.get_next()
else:
    previous.set_next(current.get_next())
```

Doubly Linked List :-

-> contains an extra pointer, called previous pointer.




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

(Notes from Geeks for Geeks)

Advantages over Singly linked list :-

-> A DLL can be traversed in both forward and backward direction.

-> Delete operation in DLL is more efficient if pointer to the node to be deleted is given.

-> We can quickly insert a new node before a given node.

* In SLL, pointer to the previous node is needed, to delete.

Disadvantages Over SLL :-

→ Every node of DLL require extra space for privious pointer.

→ All operations require an extra pointer previous to be maintained.

Insertion :-

A node can be added in four ways :-

(i) at the front of the DLL.
(ii) after given node.
(iii) at the end of the DLL.
(iv) before a given node.