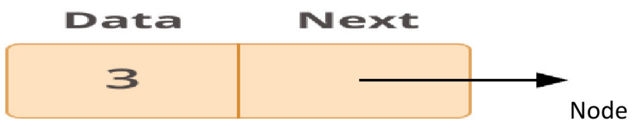
UNIT VI

Singly Linked List:

→Generally "linked list" means a singly linked list.
Each element of a linked list is called a **node**, and every node has two different fields:

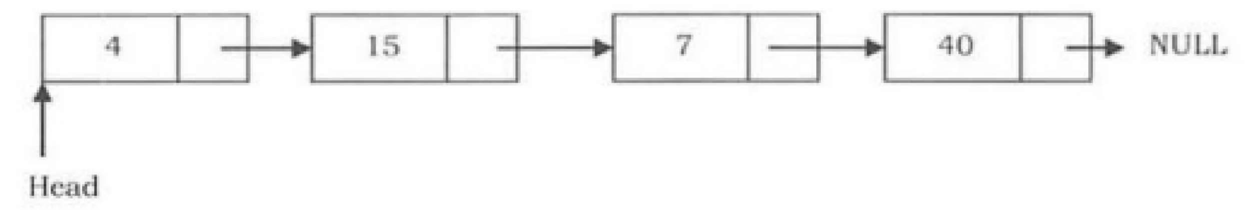
1. **Data** contains the value to be stored in the node.
2. **Next** contains a reference to the next node on the list.

Here's what a typical node looks like:



→A linked list is a collection of nodes. The first node is called the **head**, and it's used as the starting point for any iteration through the list. The last node must have its next reference pointing to None to determine the end of the list.

→ This list consists of a number of nodes in which each node has o next pointer to the following element. The link of the last node in the list is NULL, which indicates the end of the list.



In this linked list has 4 nodes. Each node has data and next fields. Data field has value and next field holds the address of the next node. Here head pointer holds the address of the first node. Last node next pointer holds the NULL value.

Creating Node:

```
class Node:
#constructor
   def __init__(self,data):
         self.data=data
         self.next=None #address of next node
class SLL:
#constructor
   def __init__(self):
         self.head=None
#creating object
s=SLL()
n1=Node(4)        #creates the node n1 with value 4
s.head=n1         #set node n1 as head
n2=Node(15)       # creates the node n2 with value 15
n1.next=n2        #linking first node and second node
n3=Node(7)        # creates the node n3 with value 7
n2.next=n3        #linking second node and third node
n4=Node(40)        # creates the node n4 with value 40
n3.next=n4        #linking third node and fourth node
```

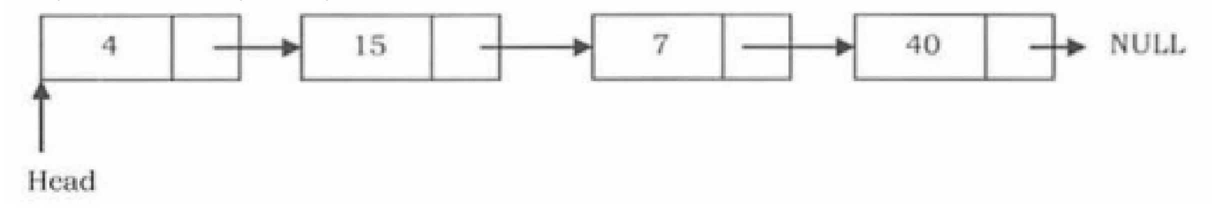
Basic operations on Singly linked list

- Traversing the Nodes

- Searching for a Node
- Prepending Nodes
- Removing Nodes.

Traversing the nodes:

Let us assume that the head points to the first node of the list. To traverse the list we do the following.
• Follow the pointers.
• Display the contents of the nodes as they are traversed.
• Stop when the next pointer points to NULL.



Algorithm:

#Purpose: This algorithm traverses from first node to last node of linked list if it is not empty and display the values of linked list.

Step1:

```
        Set current pointer to the first node of linked list.
        current=self.head
```

Step2: Check linked list is empty or not

Step3: If list is not empty travel from first node to last node means till current!=NULL

```
        while current!=None:
                print(current.data,"-->",end=" ")
                current=current.next
```

Program:

```
def traverse(self):
    current=self.head
    if current is None:
      print("linked list is EMPTY")
    while current!=None:
      print(current.data,"-->",end=" ")
      current=current.next
```

Time and space complexity:

Time Complexity: O(n) for scanning the list of size n
Space Complexity: 0(1), for creating a temporary variable.

Searching for a node:

In this operation, we have to check whether particular data value present in a linked list or not

Algorithm:

Purpose: This algorithm searches for the key in the linked list.
Input: key->the item to be searched
Output: i)the function returns 1 if key is found
ii)otherwise returns -1 indicating searching unsuccessful

```
        while current!=None:
                if current.data==key:
                        return 1
                current=current.next
        return -1
```

Program:

```
def search(self,key):
    current=self.head
    if current is None:
      print("linked list is eMPTY")
    while current!=None:
      if current.data==key:
        return 1
      current=current.next
    return -1
```

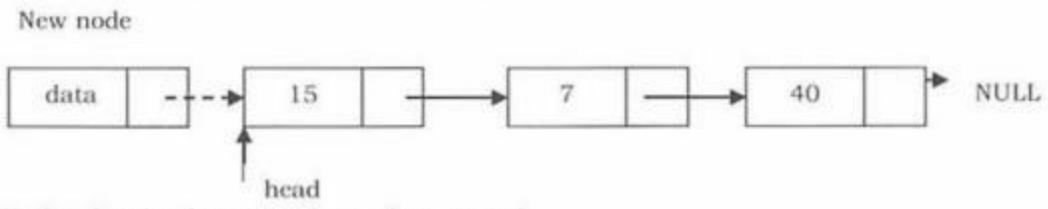
Time and space complexity:

Time Complexity: O(n) for scanning the list of size n
Space Complexity: 0(1), for creating a temporary variable.

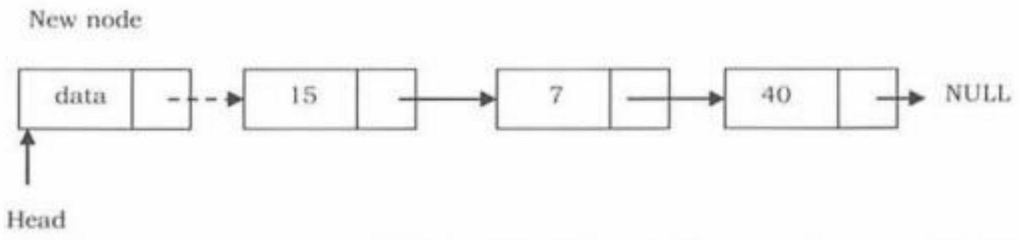
Prepending Nodes:

- Adding new node in the beginning of the linked list is considered as prepending nodes.
- In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.



- Update head pointer to point to the new node.



Algorithm:

Purpose: This algorithm inserts the new node in the beginning of the linked list

Step1: Create a newnode
    newnode=Node(data)

Step2: Link new node next pointer to the head node.
newnode.next=self.head

Step3: Update the head pointer to new node
self.head=newnode

Program:

```
def insert_begin(self,data):
    newnode=Node(data)
    newnode.next=self.head
    self.head=newnode
```

Time and space complexity:

Time Complexity: O(n), since, in the worst case, we may need to insert the node at the end of the list.
Space Complexity: O(1), for creating one temporary variable.

Removing the nodes:

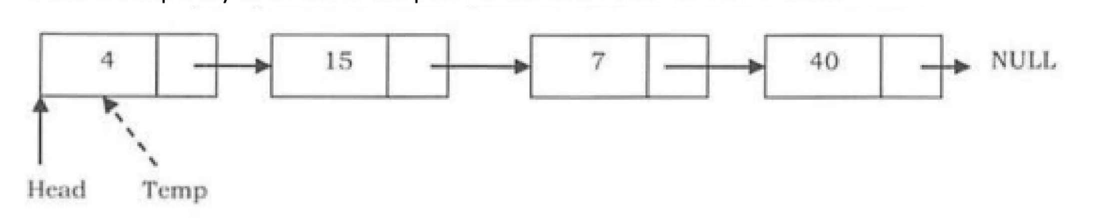
We have three cases in deletion of nodes:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

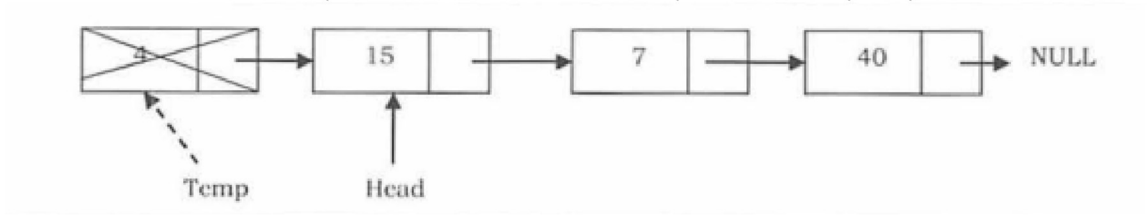
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

• Create a temporary node which will point to the same node as that of head.



• Now, move the head nodes pointer to the next node and dispose of the temporary node.



Algorithm:

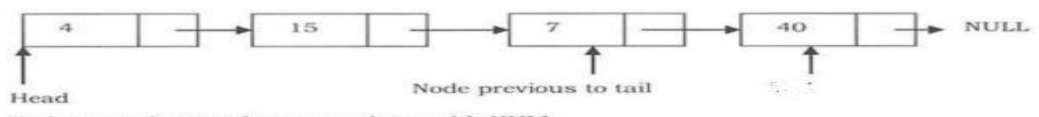
Purpose: This algorithm deletes the node in the beginning
Step1: Set temp pointer to the first node
temp=self.head
Step2: Update the head pointer to the second node
self.head=temp.next
Step3: Dispose the temporary node by setting temp.next=None

Program:

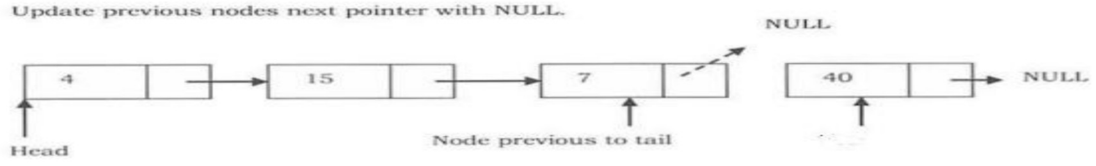
```
def delete_begin(self):
    temp=self.head
    self.head=temp.next
    temp.next=None
```

Deleting the Last Node in Singly Linked List

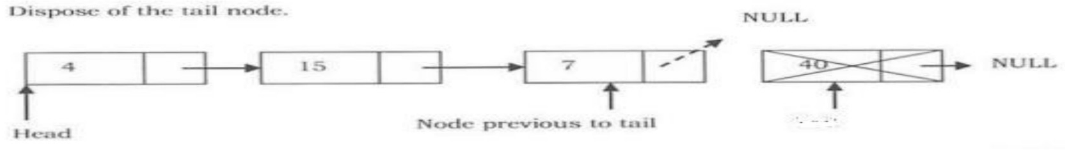
In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the last.
It can be done in three steps:
• Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the last node and the other pointing to the node before the last node.



- Update previous nodes next pointer with NULL.



- Dispose of the tail node.



Algorithm:

Purpose: This algorithm deletes the node from the end of the linked list

Step1: Set temp pointer to the second node and prev pointer to the first node

```
temp=self.head.next
prev=self.head
```

Step2: Traverse till end. While traversing move temp and prev pointer

```
while temp.next!=None:
   temp=temp.next
   prev=prev.next
```

Step3: Update prev.next=None, it disposes the last node.

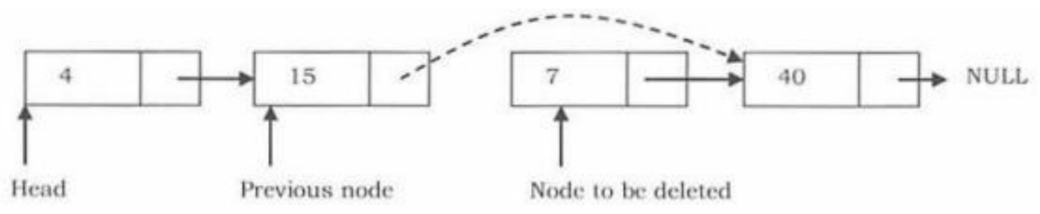
Program:

```
def delete_end(self):
      temp=self.head.next
      prev=self.head
      while temp.next!=None:
         temp=temp.next
         prev=prev.next
      prev.next=None
```

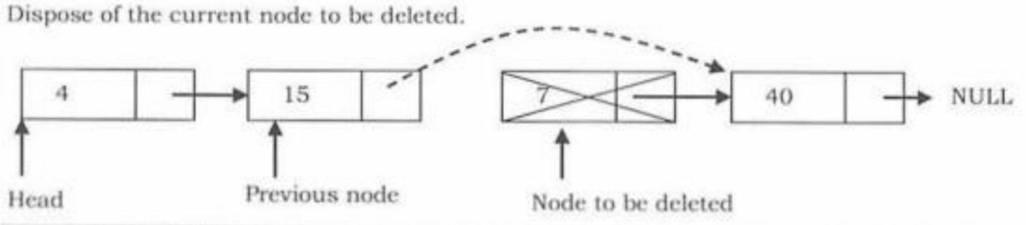
Deleting an intermediate node:

In this case, the node to be removed is always located between two nodes. Head and end links are not updated in this case. Such n removal can be done in two steps:

• Similar to the previous case, maintain the previous node while traversing the list. Once we find the node lo be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of the current node to be deleted.



Algorithm:

Purpose: This algorithm deletes the intermediate node based on position

Step1: Set temp pointer to the second node and prev pointer to the first node

```
    temp=self.head.next
    prev=self.head
```

Step2: Traverse till position

```
    for i in range(pos-1):
       temp=temp.next
       prev=prev.next
```

Step3: Update prev.next and temp.next. Dispose temp node

```
prev.next=temp.next
    temp.next=None
```

Program:

```
def delete(self,pos):
    temp=self.head.next
    prev=self.head
    for i in range(pos-1):
       temp=temp.next
       prev=prev.next
    prev.next=temp.next
    temp.next=None
```

Time and space complexity

Time Complexity: 0(n). In the worst case, we may need to delete the node at the end of the list.
Space Complexity: 0(1), for one temporary variable.

Linked List Iterators:

- Since the custom-created linked list is not iterable(we cannot print linked list values directly), we have to add an "__iter__" function so as to traverse through the list.
- __iter__ function is written inside the Singly linked list class.
- **yield** is a keyword in Python that is used **to return from a function without destroying the states of its local variable** and when the function is called, the execution starts from the last yield statement.

Algorithm:

This algorithm iterates through each and every node of singly linked list

Step1: Set n to the first node of singly linked list

```
   n= self.head
```

Step2: Iterate through each and every node till it reaches the end

```
while n!=None:
     yield n
     n = n.next
```

Program:

```
def __iter__(self):
   n= self.head
   while n!=None:
     yield n
     n = n.next
```

#Program to display the singly linked list values by using __iter__function

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

#Create a class to initialize head and tail references
class SinglyLinkedList:
    def __init__(self):
        self.head = None

     #Iterator Function
    def __iter__(self):
     n= self.head
     while n!=None:
        yield n
        n = n.next

s = SinglyLinkedList()    #Initializing object of singly linked list
n1 = Node(170)
s.head=n1
n2 = Node(210)
n1.next=n2
print("Linked list data values are:")
print([n.data for n in s])
```

Output:

```
Linked list data values are:
[170, 210]
```

Time and Space Complexity:

The time complexity for initializing a singly linked list is **O(1).** The space complexity is **O(1)** as no additional memory is required to initialize the linked list.