Week 6	Singly Linked List – Creating Nodes, Traversing the Nodes, searching for a Node, Prepending Nodes, Removing Nodes. Linked List Iterators.

Linked List

- **A Linked List data structure contains a collection of nodes, each of which contains data and at least one link to another node.**
- A linked list is a linked structure in which the nodes are connected in sequence.
- The last node in the list, commonly called the **tail node**, is indicated by a null link reference.
- The First node in the list commonly called as the **head node, or head reference**.
- A linked list can also be empty, which is indicated when the head reference is null.
- **There are two types of Linked List:**
    1. **Singly Linked List**
    2. **Doubly Linked List**

Examples of Linked List:

1. **Image Viewer:** Previous and next images can be accesses by implementing image viewer using Linked List Data Structure.
2. **Music Player:** Songs in music player are linked to previous and next song so music player can have implemented using Linked List Data Structure.
3. **Previous and Next Pages in web browser:** We can access previous and next pages of website by clicking on Prev and Next button, which can be implemented using Linked List Data Structure.

Applications of Linked List

1. Implementation of Stack and Queue Data Structures
2. Implementation of Graphs
3. Dynamic Memory Allocation
4. Maintaining of directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of Polynomials
7. To represent sparse matrices

Singly Linked List

- **A singly linked list is a linked list in which each node contains data field and a single link field.**
- **There are several operations that are commonly performed on a singly linked list:**
  - **Creating Nodes**
  - **Traversal of Nodes**
  - **Searching for a Node**
  - **Prepending Nodes**
  - **Removing Nodes**

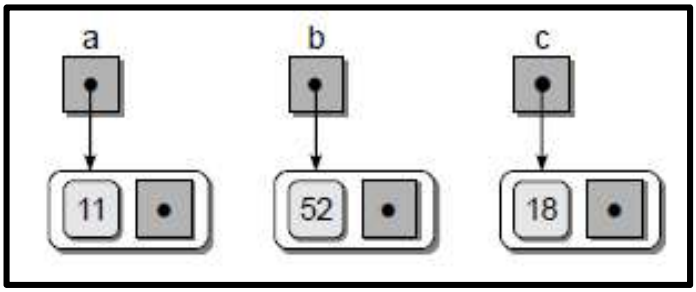
Creating Nodes

- We can create a Node by defining basic class containing data field and single link field.
- Data field contains some value and link field contains reference to some other object or node.

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

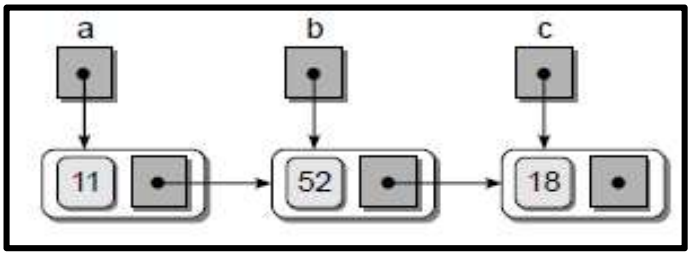
- We can create several instances of this class, each storing some data.
- In the following example, we create 3 instances, each storing an integer value:

```
a = Node(10)
b = Node(20)
c = Node(30)
```



- Since the next field contains reference to other node, we can assign node b to the next filed of node a and node c to the next field of node b as follows:

```
a.next=b
b.next=c
```



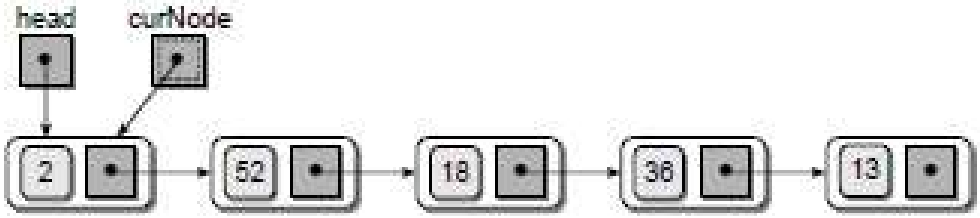
- For example, suppose we wanted to print the values of the three nodes. We can access the other two nodes through the next field of the first node:

```
print( a.data )
print( a.next.data )
print( a.next.next.data )
```

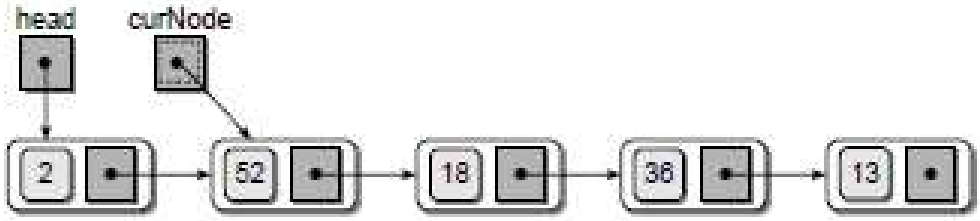
Traversing the Nodes

- It means accessing each and every node of the Linked List in sequence.
- The process starts by assigning a temporary variable **curNode** to point to the first/Head node of the list, as shown in Figure (a).
- After entering the loop, the value stored in the first node is printed by accessing the data field the node using the **curNode** variable.
- The **curNode** variable is then advanced to the next node by assigning the current node's link field, as shown in Figure (b).
- The loop continues until every node in the linked list has been accessed (when **CurNode** variable becomes None).
- The end of the Linked List/Traversal is identified when **curNode** becomes None/Null, as shown in Figure (f).

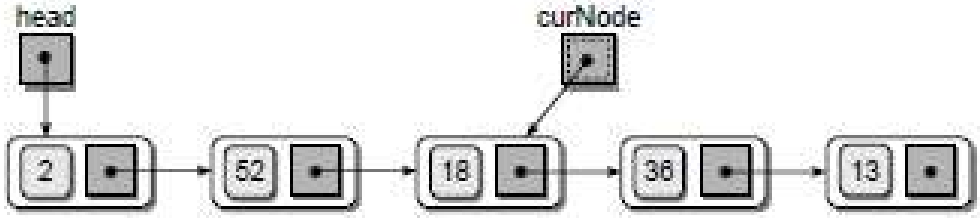
```
def traversal(self):
        curNode = self.head
        while curNode is not None:
                print(curNode.data)
                curNode = curNode.next
```



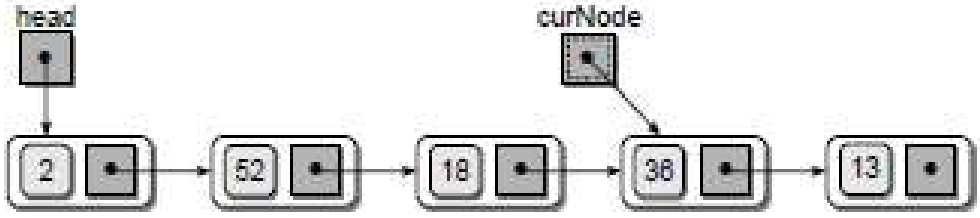
(b) Advancing the external reference after printing value 2.



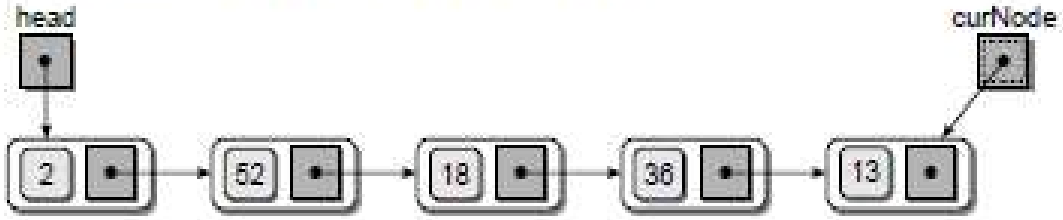
(c) Advancing the external reference after printing value 52.



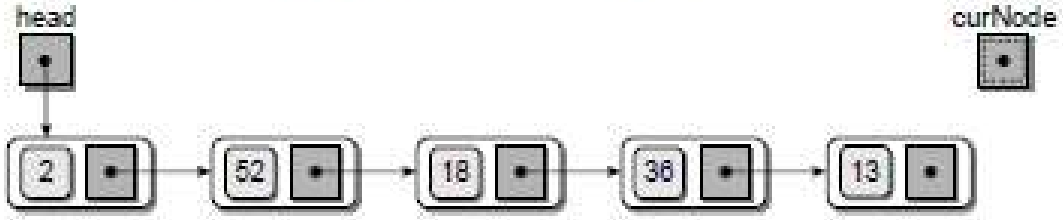
(d) Advancing the external reference after printing value 18.



(e) Advancing the external reference after printing value 36.



(f) The external reference is set to None after printing value 13.



Searching for a Node

- A linear search operation can be performed on a linked list.
- It is very similar to the traversal operation. The only difference is that the loop ends early if we found the key value within the list.

```
def  serach(self, key):
        curNode = self.head
        while curNode is not None and curNode.data != key:
            curNode = curNode.next
        return curNode is not None
```

- There are two conditions in the while loop.
- First, check whether curNode is None/Null or not. If it is not None, then check data field of curNode with the search key.
- If the Key is found in the list, curNode will not be None
- If the Key is not found in the list, curNode will be None since the end of the list is reached.

Prepending Nodes

- In Linked List. We can insert/add nodes at any location/position:
  - We can insert node at the beginning of the linked list (Prepending Node)
  - We can insert node at the end of the linked list (Appending Node)
  - We can insert node at the specific location/position of the linked list.
- Prepending a node can be done as follows:
  - First, we must create a new node to store the new value.
  - Then set its next field to point to the node currently at the front/head node of the list.
  - We then adjust head reference to point to the new node since it is now the first node in the list.

```
def prepend(self,value):
        newNode=Node(value)
        newNode.next=self.head
        self.head = newNode
```

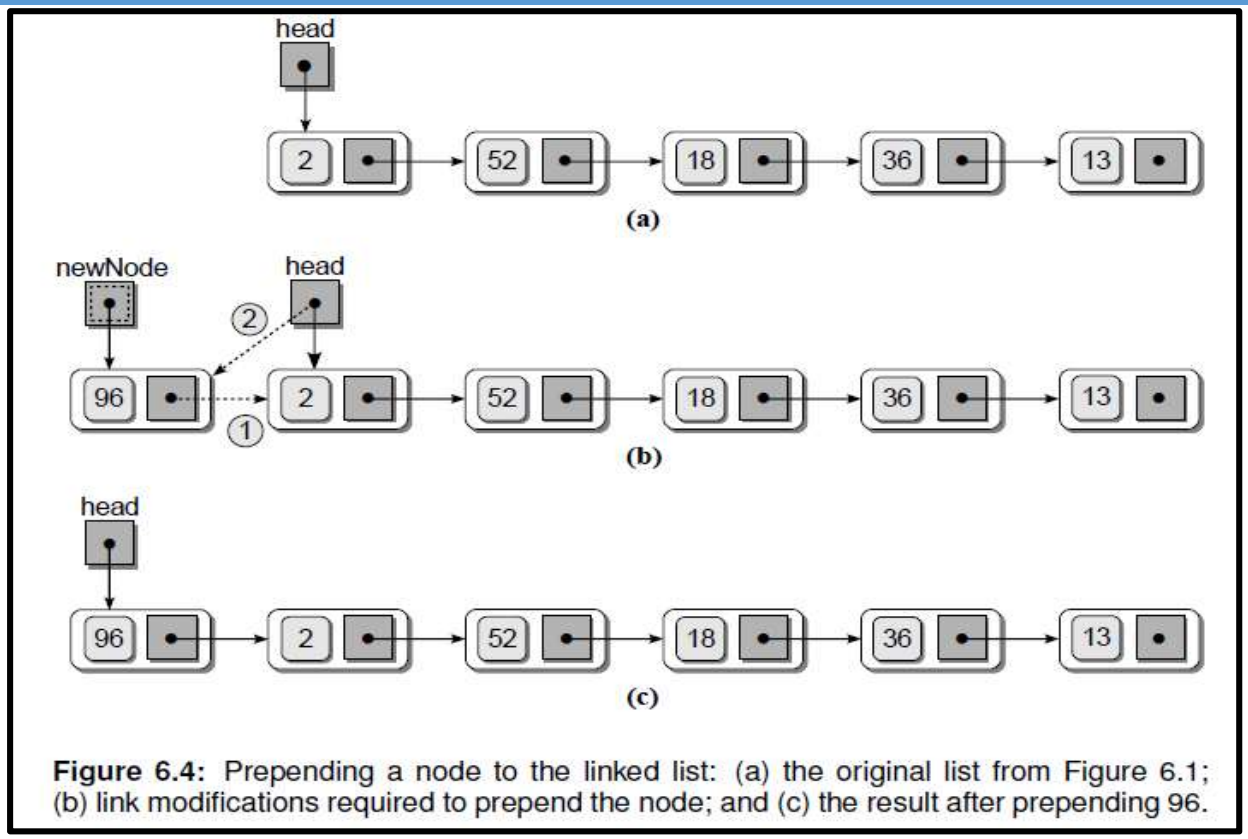
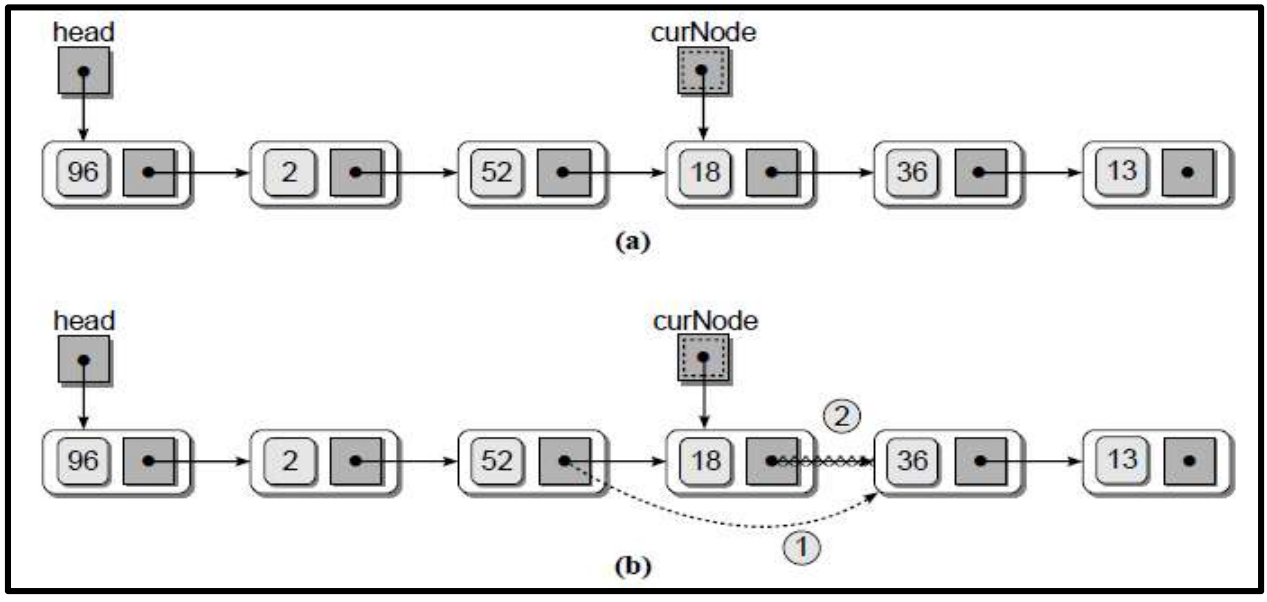


Figure 6.4: Prepending a node to the linked list: (a) the original list from Figure 6.1; (b) link modifications required to prepend the node; and (c) the result after prepending 96.

Removing/Deleting Nodes

- An item can be removed from a linked list by removing or unlinking the node containing that item.
- Consider the linked list from Figure (a) and assume we want to remove the node containing 18.
- First, we must find the node containing the target, as shown in Figure (a).
- After finding the node, it has to be unlinked from the list, which means changing the link field of the node's predecessor to point to its successor as shown in Figure (b).



```
def remove(self, target):
        predNode = None
        curNode = self.head
        while curNode is not None and curNode.data !=target:
            predNode = curNode
            curNode = curNode.next

        if curNode is not None:
            if curNode is self.head:
                self.head=curNode.next
            else:
                predNode.next = curNode.next
        else:
            print("\n",target,"not found in the list")
```

Linked List Iterators

- We can develop an iterator for Linked List by defining our iterator class.
- our iterator class would have to keep track of the current node in the linked list and thus we use a single data field, curNode in the iterator.
- This reference will be advanced through the linked list as the for loop moves over the nodes.

```
class LLIterator:
    def __init__(self,head):
        self.curNode=head

    def __iter__(self):
        return self

    def __next__(self):
         if self.curNode is None:
             raise StopIteration
         else:
             item = self.curNode.data
             self.curNode = self.curNode.next
             return item
```

Program #1: Python Program to implement Singly Linked List

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

class LinkedList:
    def __init__(self,value):
        self.head = Node(value)

    def append(self,value):
        newNode = Node(value)
        curNode = self.head
        while curNode.next is not None:
            curNode = curNode.next
        curNode.next= newNode

    def insert(self,value,index):
        newNode = Node(value)
        predNode = None
        curNode = self.head
        i=0
        while curNode is not None and i !=index:
            predNode = curNode
            curNode = curNode.next
            i=i+1
        predNode.next=newNode
        newNode.next=curNode

    def prepend(self,value):
        newNode=Node(value)
        newNode.next=self.head
        self.head = newNode

    def traversal(self):
        curNode = self.head
        while curNode is not None:
            print(curNode.data)
            curNode = curNode.next
```

```
    def remove(self,target):
        predNode = None
        curNode = self.head
        while curNode is not None and curNode.data !=target:
            predNode = curNode
            curNode = curNode.next
        if curNode is not None:
            if curNode is self.head:
                self.head=curNode.next
            else:
                predNode.next = curNode.next
        else:
            print("\n",target,"not found in the list")

    def search(self,target):
        curNode = self.head
        while curNode is not None:
            if curNode.Data == Target:
                return True
            else:
                curNode = curNode.next
        return False

    def isEmpty(self):
        return self.head is None

    def length(self):
        curNode = self.head
        count = 0
        while curNode is not None:
            curNode = curNode.next
            count = count + 1
        return count

LL = LinkedList(10)
LL.append(20)
LL.append(30)
LL.prepend(5)
```

```
LL.insert(15,2)
LL.traversal()
print("\nlength: ",LL.length())

LL.remove(20)
LL.traversal()
print("\nList Empty??  :",LL.isEmpty())
LL.remove(10)
LL.remove(100)
print(LL.search(5))
print(LL.search(20))
LL.traversal()
print("\nlength: ",LL.length())
```

Output #1:

```
5
10
15
20
30
length:  5
5
10
15
30
List Empty??  : False
 100 not found in the list
True
False
5
15
30
length:  3
```

Program #2: Python Program to implement Linked List Iterators.

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

class LinkedList:
    def __init__(self,value):
        self.head = Node(value)

    def append(self,value):
        newNode = Node(value)
        curNode = self.head
        while curNode.next is not None:
            curNode = curNode.next
        curNode.next= newNode

class LLIterator:
    def __init__(self,head):
        self.curNode=head

    def __iter__(self):
        return self

    def __next__(self):
        if self.curNode is None:
            raise StopIteration
        else:
            item=self.curNode.data
            self.curNode = self.curNode.next
            return item

LL = LinkedList(10)
LL.append(20)
LL.append(30)
L = LLIterator(LL.head)
for i in L:
    print(i)
```

Output #1:

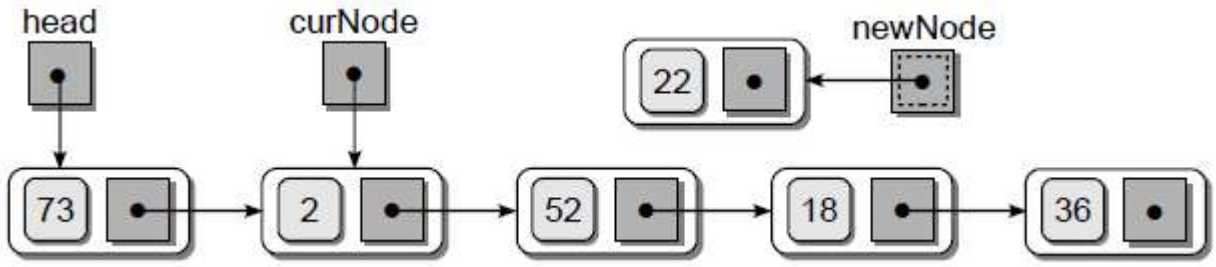
```
10
20
30
```

Activity #6

1. Develop Python function to append nodes.
2. Develop Python Program to implement Singly Linked List and perform the following operations on it:
    a. Add items 10 and 20 to the end of the list.
    b. Add item 5 at the beginning of the list.
    c. Display all nodes of the list
    d. Check whether list is empty or not
    e. Display the length of the List
    f. Check whether 40 exists in list or not
    g. Remove item 10 from the list
    h. Remove item 5 from the list
3. Evaluate the following code segment which creates a singly linked list. Draw the resulting list.

```
box = None
temp = None
for i in range( 4 ) :
if i % 3 != 0 :
temp = ListNode( i )
temp.next = box
box = temp
```

4. Consider the following singly linked list. Provide the instructions to insert the new node immediately following the node containing 52.



5. Consider the following singly linked list. Provide the instructions to remove the node containing 18.