Evaluation Scheme

B. TECH (DS)
EVALUATION SCHEME
SEMESTER-III

Sl. No.	Subject Codes	Subject Name	Periods			Evaluation Scheme				End Semester		Total	Credit
			L	T	P	CT	TA	TOTAL	PS	TE	PE		
WEEKS COMPULSORY INDUCTION PROGRAM													
1	AAS0303	Statistics and Probability	3	1	0	30	20	50		100		150	4
2	ACSE0306	Discrete Structures	3	0	0	30	20	50		100		150	3
3	ACSE0305	Computer Organization & Architecture	3	0	0	30	20	50		100		150	3
4	ACSE0302	Object Oriented Techniques using Java	3	0	0	30	20	50		100		150	3
5	ACSE0301	Data Structures	3	1	0	30	20	50		100		150	4
6	ACSDS0301	Foundations of Data Science	3	0	0	30	20	50		100		150	3
7	ACSE0352	Object Oriented Techniques using Java Lab	0	0	2				25		25	50	1
8	ACSE0351	Data Structures Lab	0	0	2				25		25	50	1
9	ACSDS0351	Data Analysis Lab	0	0	2				25		25	50	1
10	ACSE0359	Internship Assessment-I	0	0	2				50			50	1
11	ANC0301 / ANC0302	Cyber Security* / Environmental Science*(Non Credit)	2	0	0	30	20	50		50		100	0
12		MOOCs** (For B.Tech. Hons. Degree)											
		GRAND TOTAL										**1100**	**24**

****List of MOOCs (Coursera) Based Recommended Courses for Second Year (Semester-III) B. Tech Students**

S. No.	Subject Code	Course Name	University / Industry Partner Name	No of Hours	Credits
1	AMC0027	Basic Data Descriptors, Statistical Distributions, and Application to Business Decisions	Rice University	21	1.5
2	AMC0022	Data Analysis with Python	IBM	13	1

Unit III Syllabus

- Advantages of linked list over array,
- Self-referential structure,
- Singly Linked List, Doubly Linked List, Circular Linked List.
- **Operations on a Linked List:** Insertion, Deletion, Traversal, Reversal, Searching, Polynomial Representation and Addition of Polynomials.
- Implementation of Stack and Queue using Linked lists.

Branch wise Application

Unit Content

- Advantages of Linked List over Array
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Operation on Linked List
  - Insertion
  - Deletion
  - Traversal
  - Reversal
  - Searching Polynomial Representation
  - Addition, Subtraction and Multiplication of Polynomials
- Implementation of Stack and Queue using Linked List

Unit Objective

- To learn about linked lists.
- To understand different types of Linked list.
- Basic operations of linked list.

Course Objective

- Introduction to basic data structures.
- To know about the basic properties of different data structures.
- Classification and operations on data structure
- Understand algorithms and their efficiency
- Study logical and mathematical description of array and link list.
- Implementation of array and link list on computer.
- Differentiate the usage of array and link list in different scenarios.

NIET Greater Noida GET FUTURE READY Estd.2001 AN AUTONOMOUS INSTITUTE

Course Outcome

CO	CO Description	Bloom's Knowledge Level (KL)
CO 1	Describe the need of data structure and algorithms in problem solving and analyze Time space trade-off.	K2, K4
CO 2	Describe how arrays are represented in memory and how to use them for implementation of matrix operations, searching and sorting along with their computational efficiency.	K2, K6
CO 3	Design, implement and evaluate the real-world applications using stacks, queues and non-linear data structures.	K5, K6
CO 4	Compare and contrast the advantages and disadvantages of linked lists over arrays and implement operations on different types of linked list.	K4, K6
CO 5	Identify and develop the alternative implementations of data structures with respect to its performance to solve a real-world problem.	K1, K3, K5, K6

Program Outcomes (POs)

1. Engineering knowledge
2. Problem analysis
3. Design/development of solutions
4. Conduct investigations of complex problems
5. Modern tool usage
6. The engineer and society
7. Environment and sustainability
8. Ethics
9. Individual and team work
10. Communication
11. Project management and finance
12. Life-long learning

CO-PO Mapping

CO-PO correlation matrix of Data Structure (KCS 301)

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
ACSE0301.1	3	3	3	2	-	1	-	1	2	2	2	2
ACSE0301.2	3	3	2	2	-	1	-	1	2	2	1	2
ACSE0301.3	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.4	3	3	2	2	-	1	-	1	2	2	2	2
ACSE0301.5	3	3	3	3	2	2	2	2	3	3	3	3
Average	3	3	2.4	2.2	0.4	1.2	0.4	1.2	2.2	2.2	2	2.2

Program Specific Outcomes (PSOs)

On successful completion of graduation degree the Engineering graduates will be able to:

PSO1: The ability to design and develop the hardware sensor device and related interfacing software system for solving complex engineering problem.

PSO2: The ability to understanding of Inter disciplinary computing techniques and to apply them in the design of advanced computing .

PSO 3: The ability to conduct investigation of complex problem with the help of technical, managerial, leadership qualities, and modern engineering tools provided by industry sponsored laboratories.

PSO 4: The ability to identify, analyze real world problem and design their solution using artificial intelligence ,robotics, virtual. Augmented reality ,data analytics, block chain technology and cloud computing.

CO-PSO Mapping

Mapping of Program Specific Outcomes and Course Outcomes

	PSO1	PSO2	PSO3	PSO4
ACSE0301.1	3	3	2	2
ACSE0301.2	3	3	2	3
ACSE0301.3	3	3	2	2
ACSE0301.4	3	3	3	3
ACSE0301.5	3	3	3	3
Average	3	3	2.4	2.6

Prerequisite and Recap

- Interest
- Get Familiar with any programming language. C, C++ and Python.
- Start learn Data Structure and Algorithm daily.
- Practice ! Because practice makes you perfect.

Faculty Video Links, Youtube & NPTEL Video Links and Online Courses Details

- Youtube/other Video Links
- Implementation of link list
  - https://www.youtube.com/watch?v=6wXZ_m3SbEs
- Polynomial addition using link list
  - https://www.youtube.com/watch?v=V_ZNKu_pUPQ

Basic Terminology(CO1)

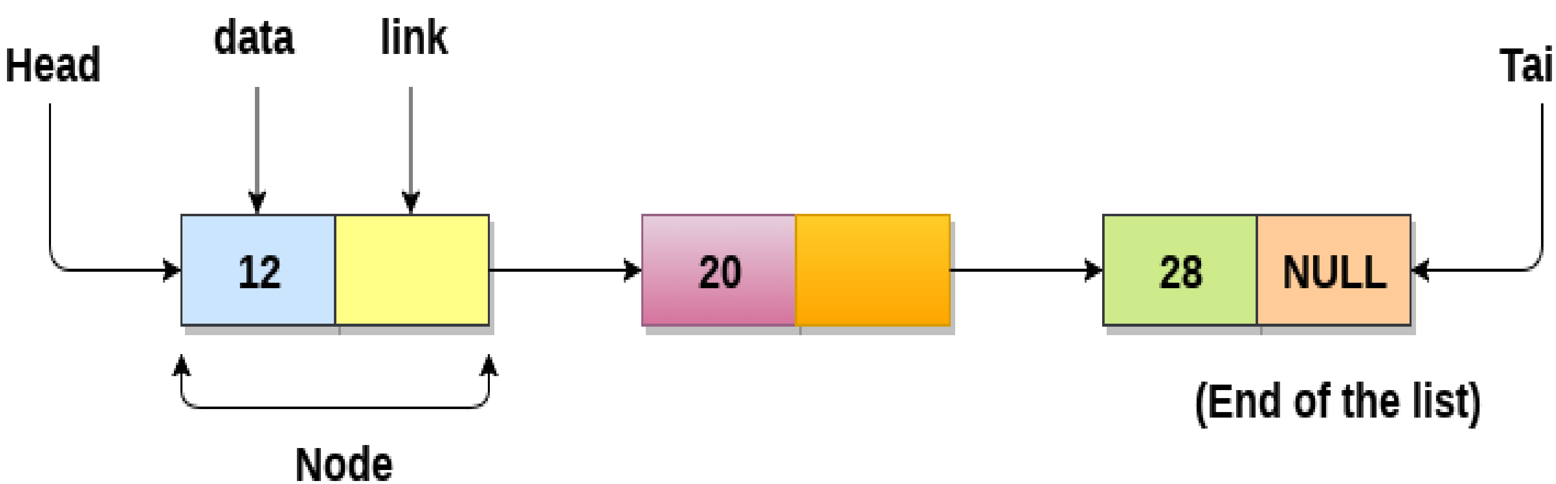
- **Linked List**
- **Doubly Linked List**
- **Circularly Linked List**
- **Circularly Doubly Linked List**

Topic Objective

- To understand linked list and the operations of linked list.
- To implement Linked list program using Python

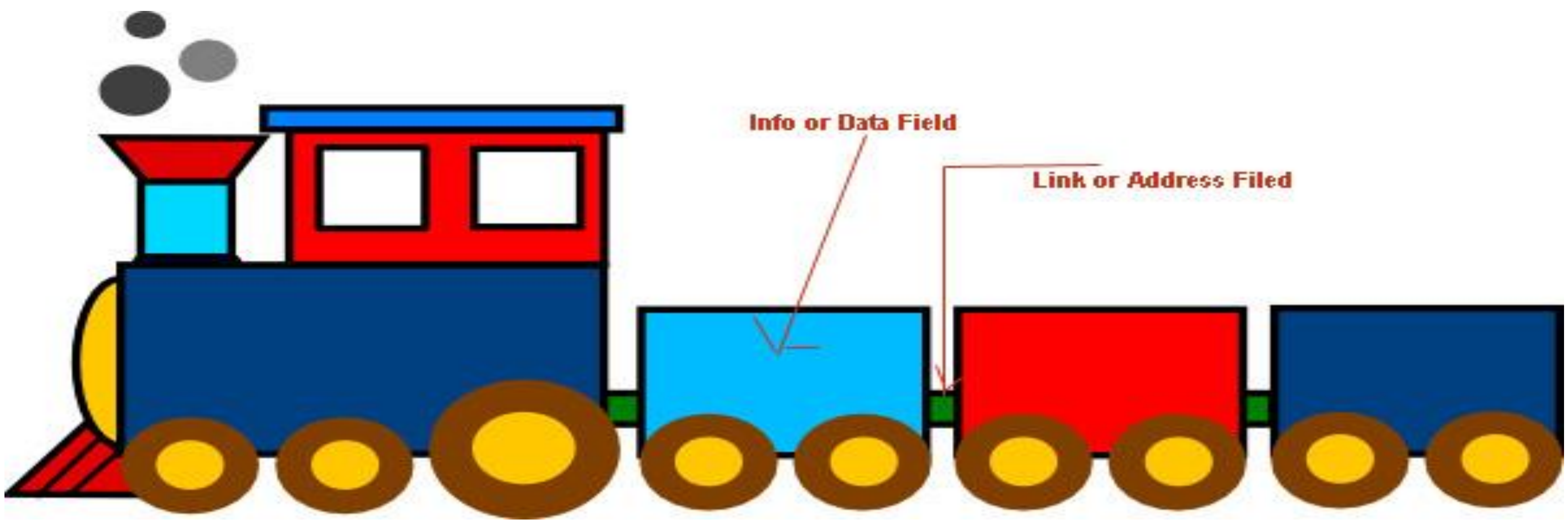
Linked List

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.



Linked List

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



Introduction to Linked List

Linked List

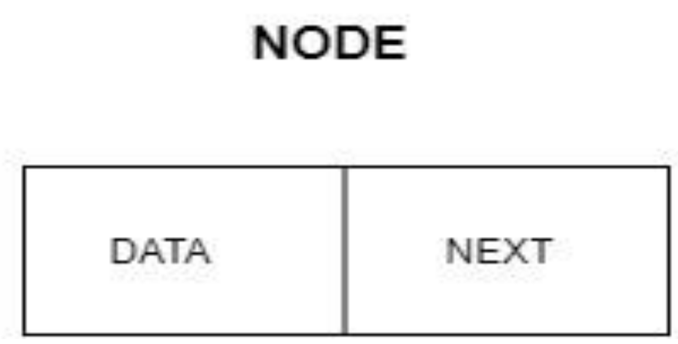
- The elements of a linked list are not stored in adjacent memory locations as in arrays.
- It is a linear collection of data elements, called nodes, where the linear order is implemented by means of pointers.

Continued....

Linked List

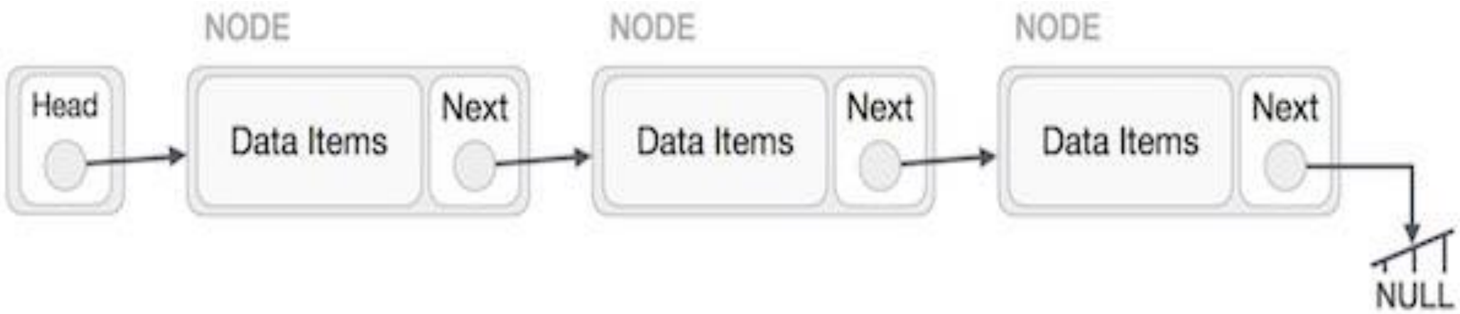
- In a linear or single-linked list, a node is connected to the next node by a single link.

- A node in this type of linked list contains two types of fields
  - data: which holds a list element
  - next: which stores a link (i.e. pointer) to the next node in the list.



Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node points to the next node.



- As per the above illustration, following are the important points to be considered.
  - Linked List contains a link element called first.
  - Each link carries a data field(s) and a link field called next.
  - Each link is linked with its next link using its next link.
  - Last link carries a link as null to mark the end of the list.

Properties of linked list

- The nodes in a linked list are not stored contiguously in the memory
- You don't have to shift any element in the list
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically

Basic Operations on Linked List

- Following are the basic operations supported by a list.
  - **Insertion** − Adds an element at the beginning of the list.
  - **Deletion** − Deletes an element at the beginning of the list.
  - **Display** − Displays the complete list.
  - **Search** − Searches an element using the given key.
  - **Delete** − Deletes an element using the given key.

Arrays & Linked list

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Arrays & Linked list

	Arrays	Linked list
INDEXING	**O(1)**	**O(n)**
Insert/Delete at the start	**O(n)**	**O(1)**
Insert/Delete at the end	**O(1)**	**O(n)**
Insert in the middle	**O(n)**	**O(n)**

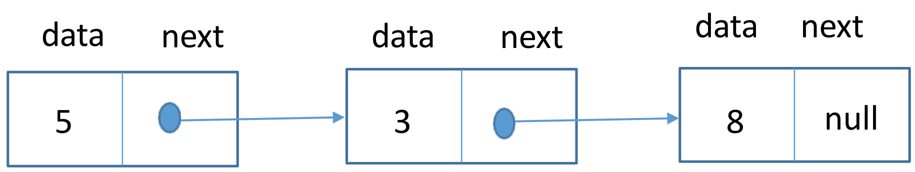
Types of Link List

- Following are the various types of linked list.
  - **Singly Linked List** − Item navigation is forward only.
  - **Doubly Linked List** − Items can be navigated forward and backward.
  - **Circular Linked List** − Last item contains link of the first element as next
  - **Circular Doubly Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous. Items can be navigated forward and backward.

Singly Linked list

- A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made.

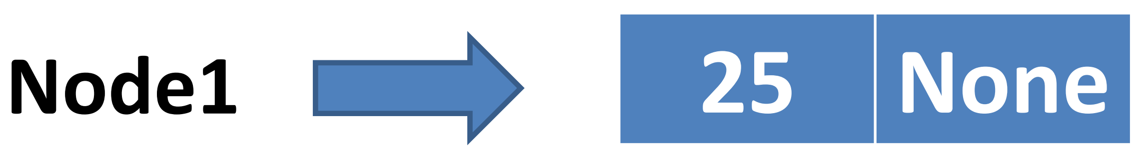
- In this type of linked list each node contains two fields one is data field which is used to store the data items and another is next field that is used to point the next node in the list.



Creating a node of linked list

```
# Node class (Creating a node of linked list)
class Node:
    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null
```

Node1=Node(25)



Creating an empty linked list

```
# Node class (Creating a node of linked list)
class Node:
      # Function to initialize the node object
   def __init__(self, data):
      self.data = data  # Assign data
      self.next = None  # Initialize next as null

# Linked List class (Linking the nodes of linked list)
class LinkedList:
      # Function to initialize the Linked List object
   def __init__(self):
      self.head = None
```

Creating a linked list with single node

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

class LinkedList:
    def __init__(self):
    self.head = None

LL = LinkedList()
LL.head = Node(3)
print(LL.head.data)
```

Creating a linked list with multiple node

```
# Linked list implementation in Python

class Node:
    # Creating a node
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:

    def __init__(self):
        self.head = None

if __name__ == '__main__':

    linked_list = LinkedList()

    # Assign item values
    linked_list.head = Node(1)
    second = Node(2)
    third = Node(3)

    # Connect nodes
    linked_list.head.next = second
    second.next = third

    # Print the linked list item
    while linked_list.head != None:
        print(linked_list.head.item, end=" ")
        linked_list.head = linked_list.head.next
```

Creation and Traversal of single linked list

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# insertion method for the linked list
 def insert(self, data):
   newNode = Node(data)
   if(self.head):
     current = self.head
     while(current.next):
       current = current.next
     current.next = newNode
   else:
     self.head = newNode
```

Creation and Traversal of single linked list (contd..)

```
# print method for the linked list
 def printLL(self):
  current = self.head
  while(current):
   print(current.data)
   current = current.next

# Singly Linked List with insertion and print methods
LL = LinkedList()
LL.insert(3)
LL.insert(4)
LL.insert(5)
LL.printLL()
```

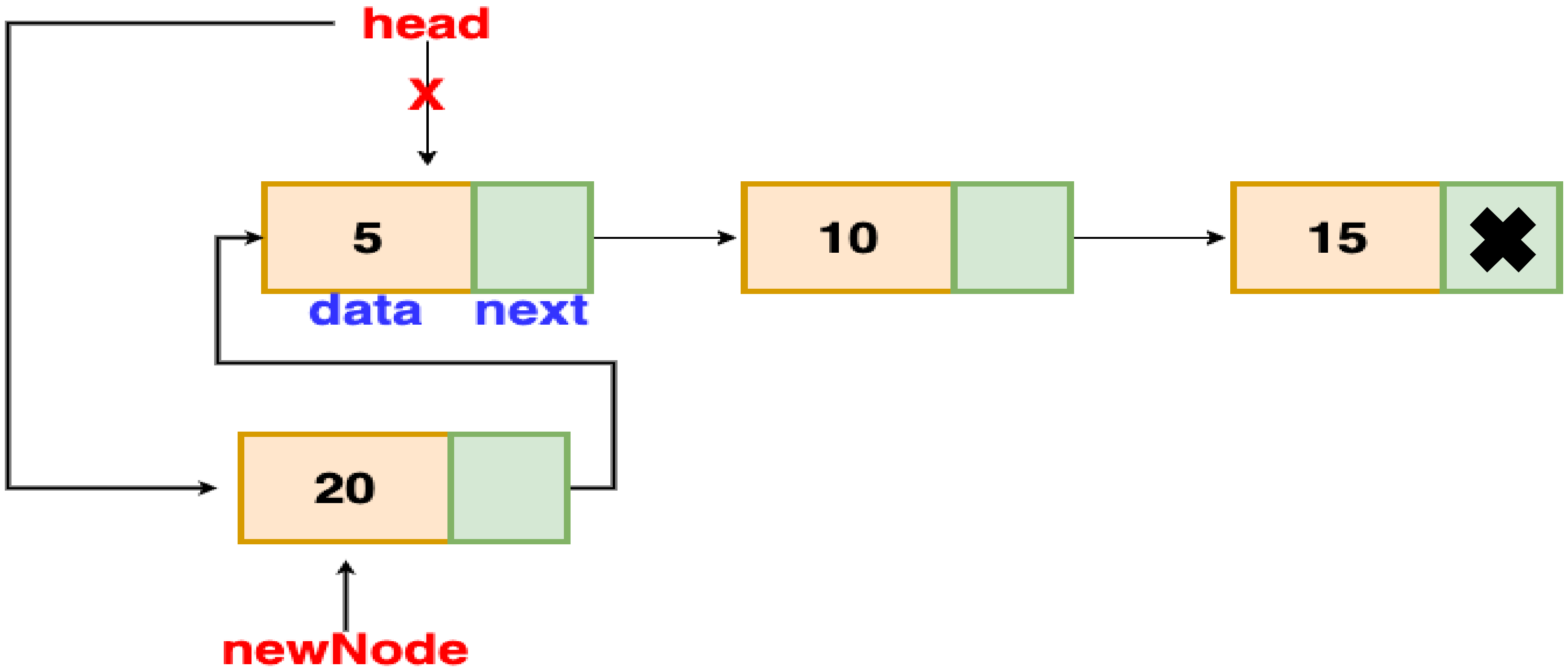
Insertion in a Single Linked List

- There are three possible positions where we can enter a new node in a linked list –
  - **Insertion at beginning**
  - **Insertion at end**
  - **Insertion at given position**

- Adding a new node in linked list is a more than one step activity.

Insertion in a Single Linked List (at beginning)

- **Insertion at beginning**



Insertion at the beginning

Insertion in single linked list (at beginning)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# insertion method for the linked list at
      beginning

def insert_beg(self, data):
    newNode = Node(data)
    if(self.head):
      newNode.next=self.head
      self.head=newNode
    else:
      self.head = newNode
```

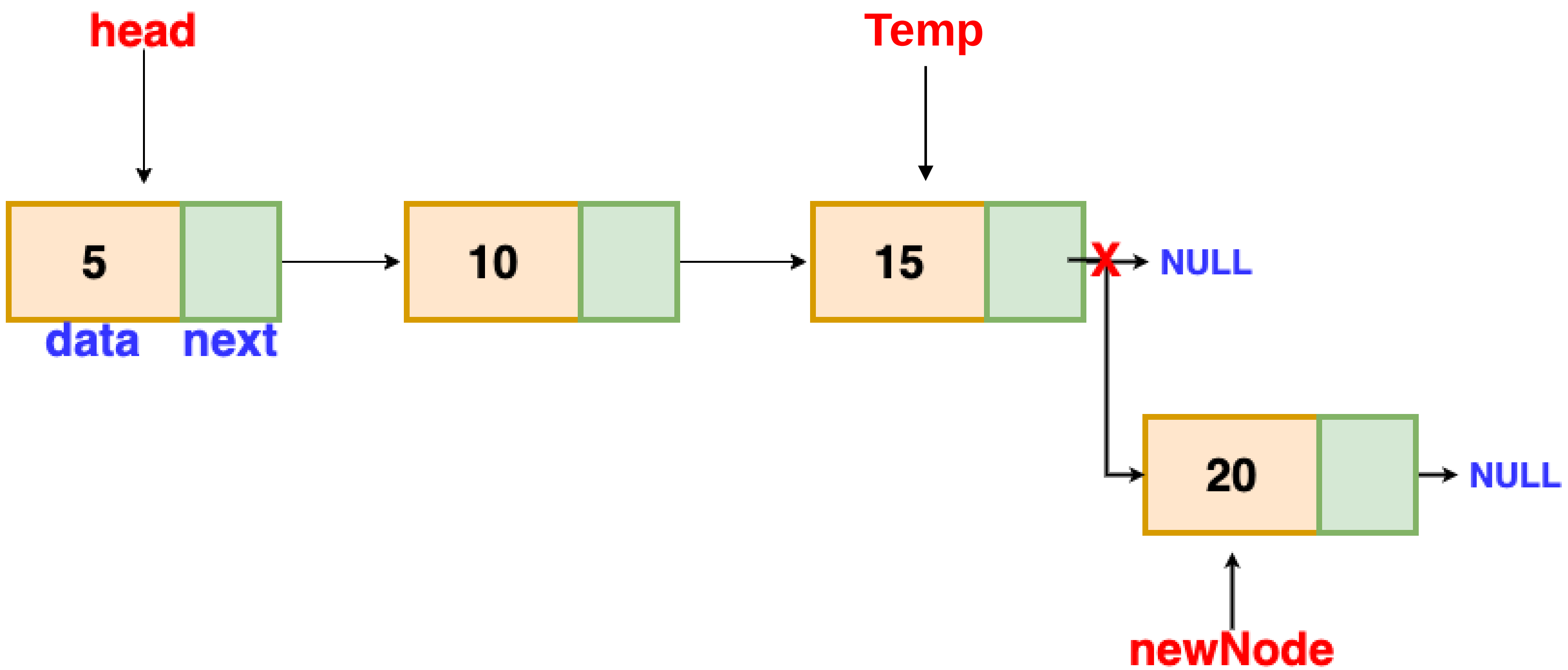
Insertion in single linked list (at beginning) (contd..)

```
# print method for the linked list
 def printLL(self):
  current = self.head
  if(current!=None):
    print("The List Contains:",end="\n")
    while(current):
      print(current.data)
      current = current.next
  else:
    print("List is Empty.")

# Singly Linked List with insertion and print methods
LL = LinkedList()
LL.insert_beg(3)
LL.insert_beg(4)
LL.insert_beg(5)
LL.printLL()
```

Insertion in a Single Linked List (at end)

- **Insertion at end**



Insertion at the end

Insertion in single linked list (at end)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

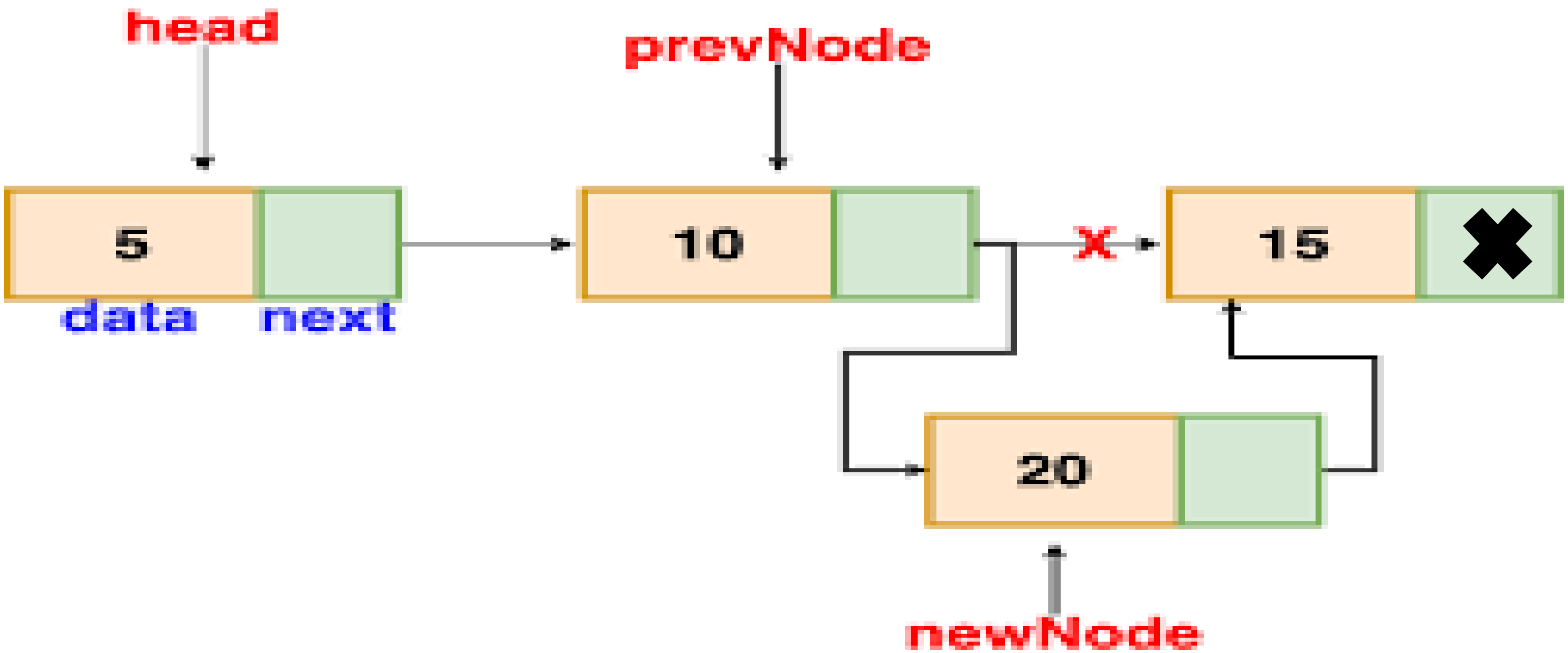
```
# insertion method for the linked list
      at end
 def insert_end(self, data):
   newNode = Node(data)
   if(self.head):
     current = self.head
     while(current.next):
       current = current.next
     current.next = newNode
   else:
     self.head = newNode
```

Insertion in single linked list (at end) (contd..)

```
# print method for the linked list
 def printLL(self):
  current = self.head
  if(current!=None):
    print("The List Contains:",end="\n")
    while(current):
      print(current.data)
      current = current.next
  else:
    print("List is Empty.")
# Singly Linked List with insertion and print methods
LL = LinkedList()
LL.insert_end(3)
LL.insert_end(4)
LL.insert_end(5)
LL.printLL()
```

Insertion in a Single Linked List (at given position)

- **Insertion at given position**



Insertion after a given node

Insertion in single linked list (at position)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# creation method for the linked list
 def create(self, data):
   newNode = Node(data)
   if(self.head):
     current = self.head
     while(current.next):
       current = current.next
     current.next = newNode
   else:
     self.head = newNode
```

Insertion in single linked list (at position)

```
# insertion method for the linked list at
    given position

 def insert_position(self, data, pos):
  newNode = Node(data)
  if(pos<1):
     print("\nPosition should be >=1.")

  elif(pos==1):
     newNode.next=self.head
     self.head=newNode
else:
     current=self.head
     for i in range(1, pos-1):
        if(current!=None):
           current=current.next
     if(current!=None):
        newNode.next=current.next
        current.next=newNode
     else:
        print("\nThe previous node is null.")
```

Insertion in single linked list (at position)

```
# print method for the linked list
def printLL(self):
  current = self.head
  if(current!=None):
    print("The List
   Contains:",end="\n")
    while(current):
      print(current.data)
      current = current.next
  else:
    print("List is Empty.")
```

```
# Singly Linked List with insertion and
   print methods
LL = LinkedList()
LL.create(2)
LL.create(3)
LL.create(4)
LL.create(5)
LL.create(6)
LL.insert_position(9, 4)
LL.printLL()
```

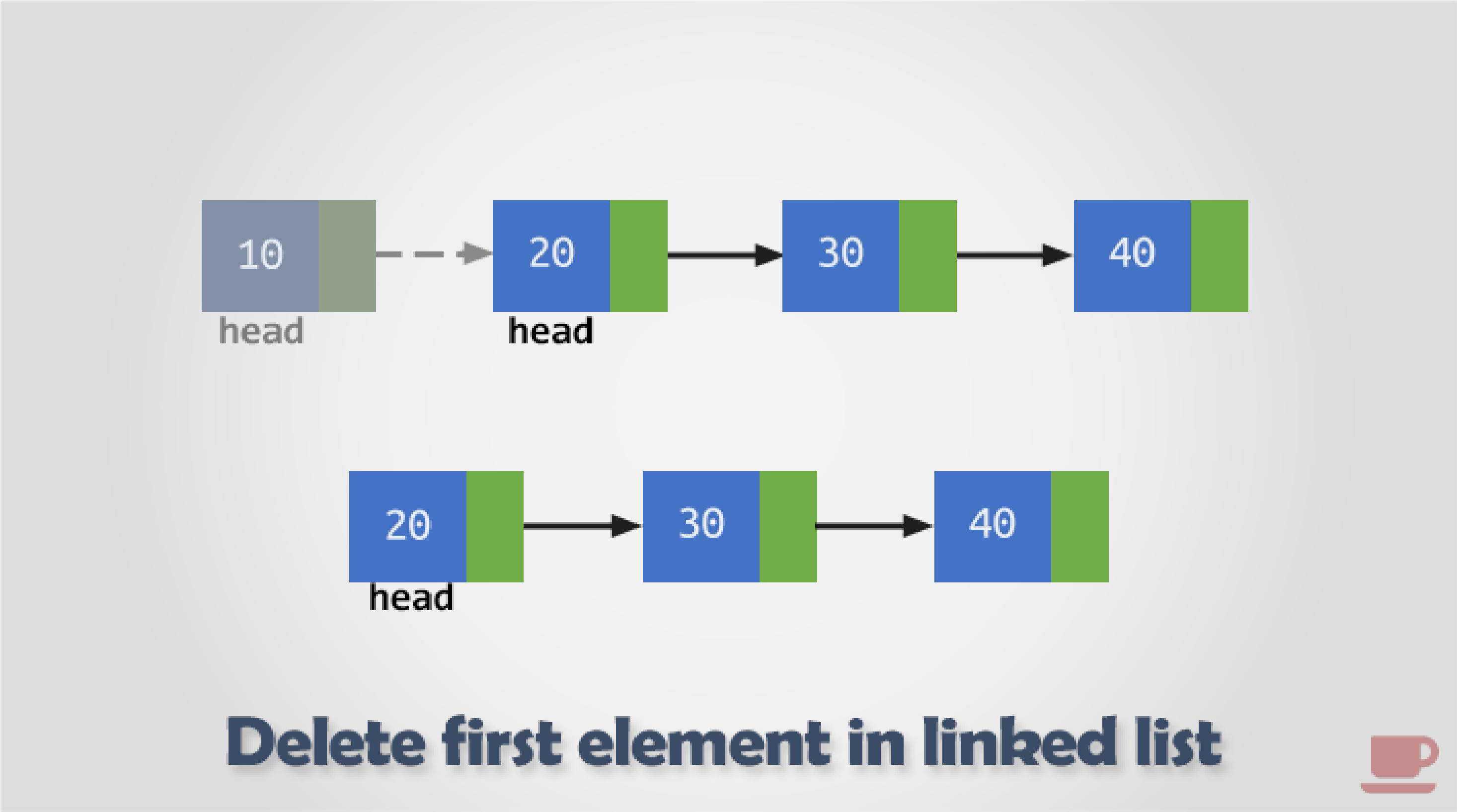
Deletion in a Single Linked List

- There are three possible positions where we can enter a new node in a linked list –
  - **Deletion at beginning**
  - **Deletion at end**
  - **Deletion from given position**

- Deleting new node in linked list is a more than one step activity.

Deletion in Single Linked List (from beginning)

- **Deletion from beginning**



Deletion in Single Linked List (from beginning)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# create method for the linked list
 def create(self, data):
   newNode = Node(data)
   if(self.head):
     current = self.head
     while(current.next):
       current = current.next
     current.next = newNode
   else:
     self.head = newNode
```

Deletion in Single Linked List (from beginning)

```
#Delete first node of the list
def del_beg(self):
  if(self.head == None):
      print("Underflow-Link List is
    empty")

  else:
    temp = self.head
    self.head = self.head.next
    print("the deleted element is",
    temp.data)
    temp = None
```

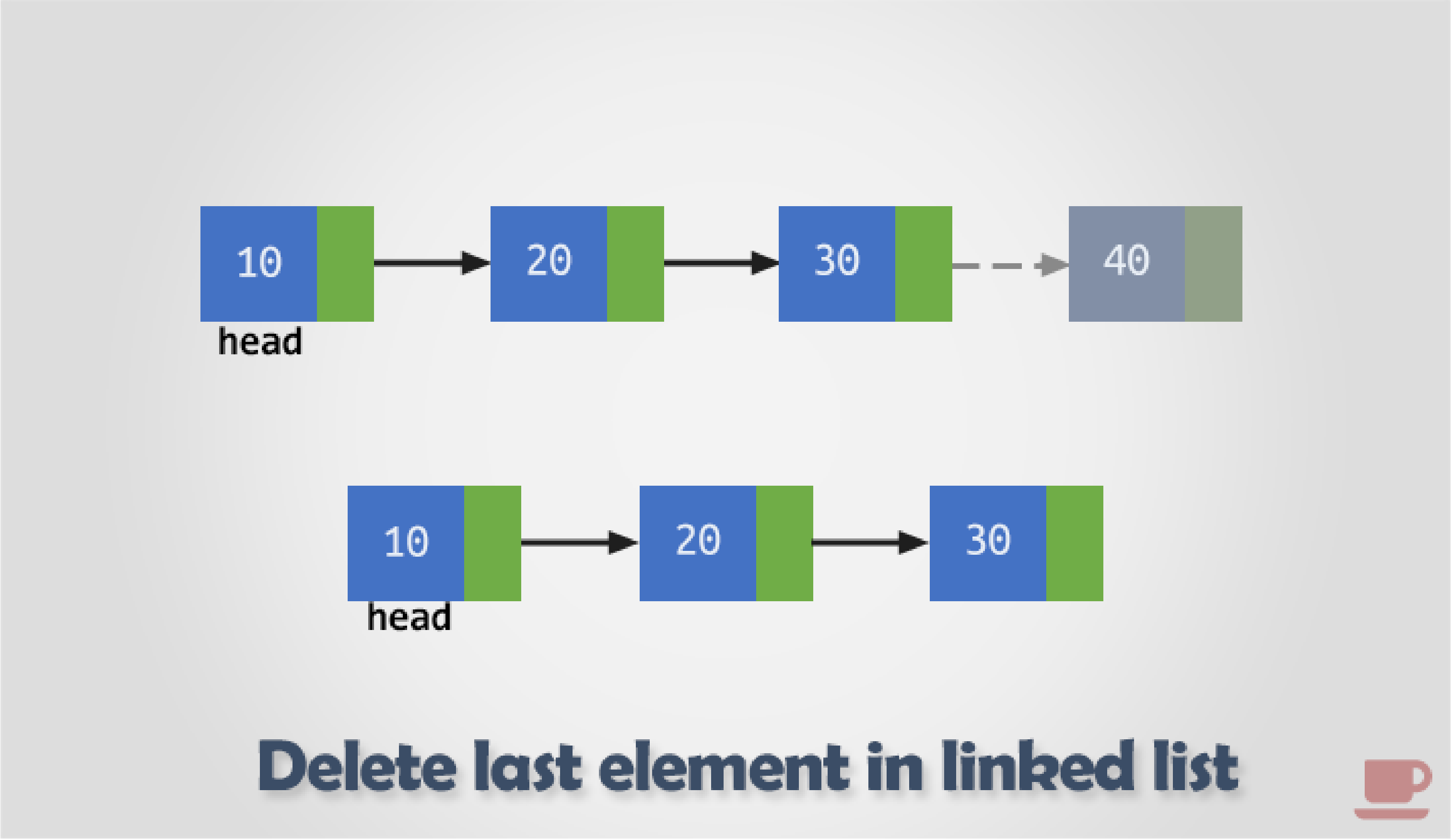
```
# print method for the linked list
def printLL(self):
  current = self.head
  if(current!=None):
    print("The List Contains:",end="\n")
    while(current):
      print(current.data)
      current = current.next
  else:
    print("List is Empty.")
```

Deletion in Single Linked List (from beginning)

```
# Singly Linked List with deletion and print methods
LL = LinkedList()
LL.create(3)
LL.create(4)
LL.create(5)
LL.printLL()
LL.del_beg()
LL.printLL()
```

Deletion in Single Linked List (from end)

- **Deletion from end**



Deletion in Single Linked List (from end)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
    head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# create method for the linked list
  def create(self, data):
    newNode = Node(data)
    if(self.head):
      current = self.head
      while(current.next):
        current = current.next
      current.next = newNode
    else:
      self.head = newNode
```

Deletion in Single Linked List (from end)

#Delete last node of the list

```
def del_end(self):
 if(self.head == None):
   print("Underflow-Link List is
  empty")

 else:
  temp = self.head
  while(temp.next!=None):
    prev=temp
    temp=temp.next
  prev.next=None
  print("The deleted element is",
  temp.data)
  temp = None
```

print method for the linked list

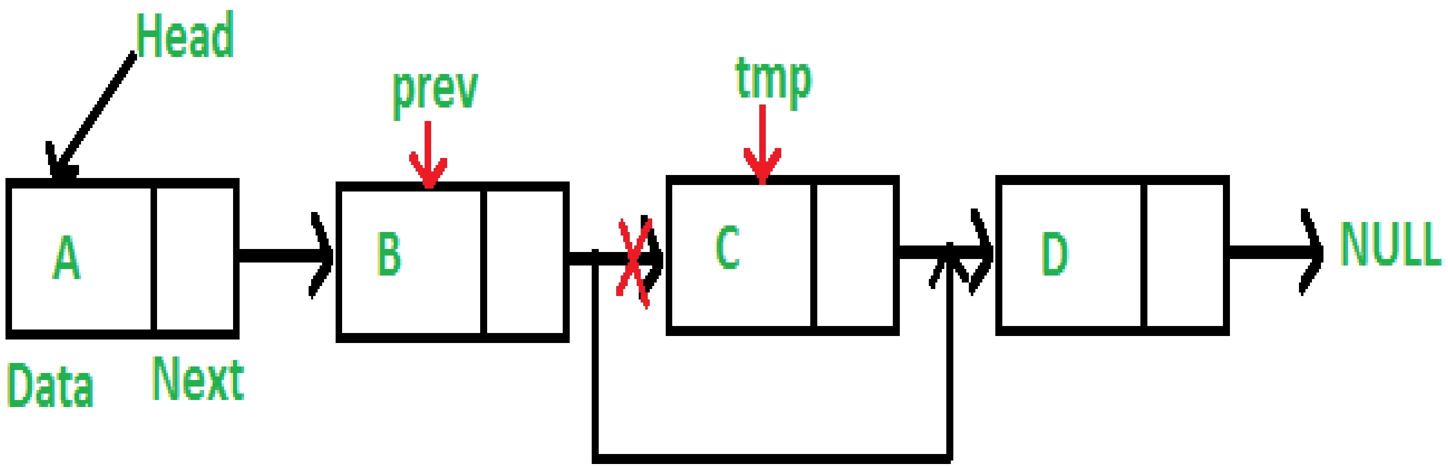
```
def printLL(self):
 current = self.head
 if(current!=None):
   print("The List Contains:",end="\n")
   while(current):
     print(current.data)
     current = current.next
 else:
   print("List is Empty.")
```

Deletion in Single Linked List (from end)

```
# Singly Linked List with deletion and print methods
LL = LinkedList()
LL.create(3)
LL.create(4)
LL.create(5)
LL.printLL()
LL.del_end()
LL.printLL()
```

Deletion in Single Linked List (from position)

- **Deletion from position**



Deletion in Single Linked List (from position)

```
# A single node of a singly linked list
class Node:
def __init__(self, data):
    self.data = data
    self.next = None


# A Linked List class with a single
      head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
# create method for the linked list
 def create(self, data):
   newNode = Node(data)
   if(self.head):
     current = self.head
     while(current.next):
       current = current.next
     current.next = newNode
   else:
     self.head = newNode
```

Deletion in Single Linked List (from position)

```
# Deletion method from the linked list at
    given position
 def del_position(self, pos):
  if(pos<1):
    print("\nPosition should be >=1.")

  elif(pos==1):
    temp = self.head
    self.head = self.head.next
    print("the deleted element is",
   temp.data)
    temp = None
else:
  temp=self.head
  for i in range(1, pos):
    if(temp!=None):
      prev=temp
      temp=temp.next

  if(temp!=None):
      prev.next=temp.next
      print("the deleted element
      is", temp.data)
      temp=None
  else:
    print("\nThe position does not
      exist in link list.")
```

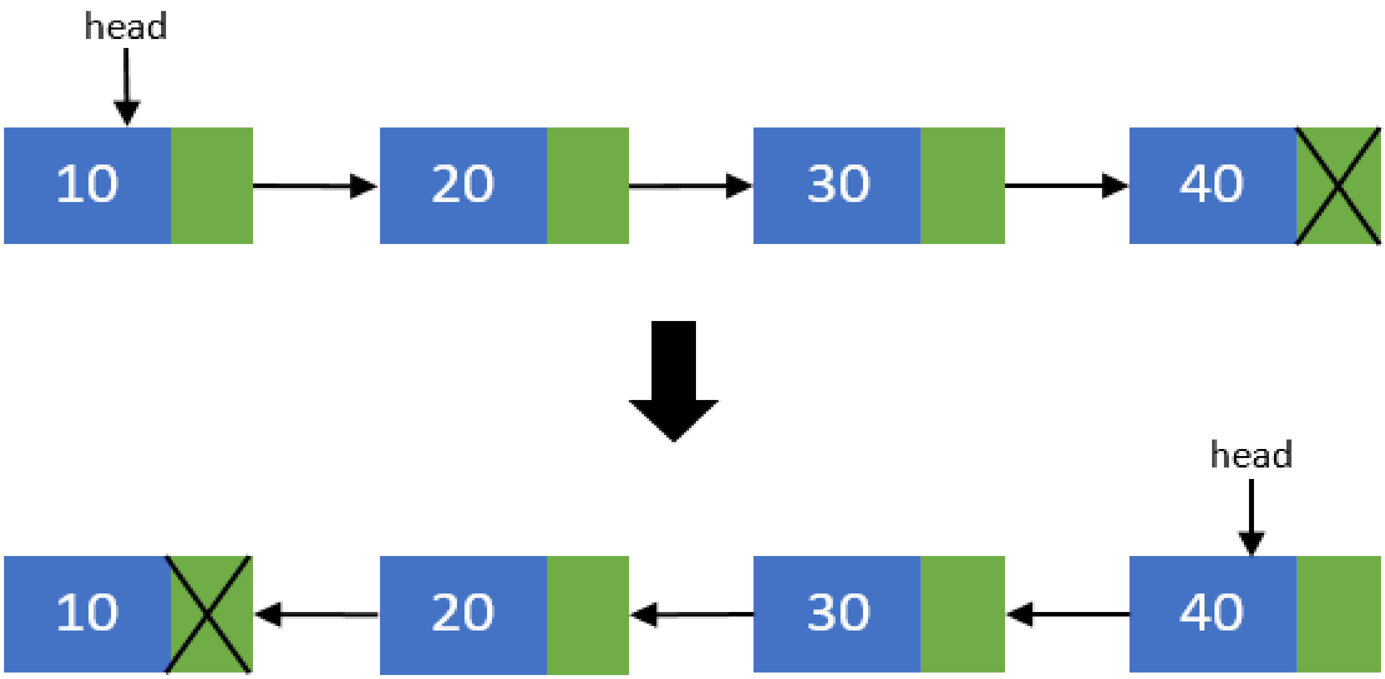
Deletion in Single Linked List (from position)

```
# print method for the linked list
def printLL(self):
  current = self.head
  if(current!=None):
    print("The List
   Contains:",end="\n")
    while(current):
      print(current.data)
      current = current.next
  else:
    print("List is Empty.")
```

```
# Singly Linked List with deletion and
      print methods
LL = LinkedList()
LL.create(3)
LL.create(4)
LL.create(5)
LL.create(6)
LL.create(7)
LL.create(8)
LL.printLL()
LL.del_position(4)
LL.printLL()
```

Reverse of a Single Linked List

If the linked list has two or more elements, we can use three pointers to implement an iterative solution..



Reverse of a Single Linked List

Method to Reverse the linked list

```
def reverse(self):
    if(self.head==None):
        print("List is Empty.")

    elif(self.head.next==None):
        print("Only one node is present in list")

    else:
        temp1 = self.head
        temp2=temp1.next
        temp3=temp2.next
        temp1.next=None
        while(temp3!=None):
            temp2.next=temp1
            temp1=temp2
            temp2=temp3
            temp3=temp3.next

        temp2.next=temp1
        self.head=temp2
```

Single Linked List operations in Python

```
# Linked list operations in Python

# Create a node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:

    def __init__(self):
        self.head = None

    # Insert at the beginning
    def insertAtBeginning(self, new_data):
        new_node = Node(new_data)

        new_node.next = self.head
        self.head = new_node
```

```
# Insert after a node
    def insertAfter(self, prev_node, new_data):

        if prev_node is None:
            print("The given previous node must inLinkedList.")
            return

        new_node = Node(new_data)
        new_node.next = prev_node.next
        prev_node.next = new_node

    # Insert at the end
    def insertAtEnd(self, new_data):
        new_node = Node(new_data)

        if self.head is None:
            self.head = new_node
            return

        last = self.head
        while (last.next):
            last = last.next

        last.next = new_node
```

Single Linked List

```
# Deleting a node
    def deleteNode(self, position):

        if self.head is None:
            return

        temp = self.head

        if position == 0:
            self.head = temp.next
            temp = None
            return
        # Find the key to be deleted
        for i in range(position - 1):
            temp = temp.next
            if temp is None:
                break

        # If the key is not present
        if temp is None:
            return

        if temp.next is None:
            return

        next = temp.next.next

        temp.next = None

        temp.next = next

    # Search an element
    def search(self, key):

        current = self.head

        while current is not None:
            if current.data == key:
                return True

            current = current.next

        return False
```

Single Linked List

```
# Sort the linked list
  def sortLinkedList(self, head):
      current = head
      index = Node(None)

      if head is None:
          return
      else:
          while current is not None:
              # index points to the node next to current
              index = current.next

              while index is not None:
                  if current.data > index.data:
                      current.data, index.data = index.data, current.data

                  index = index.next
              current = current.next

  # Print the linked list
  def printList(self):
      temp = self.head
      while (temp):
          print(str(temp.data) + " ", end="")
          temp = temp.next
```

```
if __name__ == '__main__':

    llist = LinkedList()
    llist.insertAtEnd(1)
    llist.insertAtBeginning(2)
    llist.insertAtBeginning(3)
    llist.insertAtEnd(4)
    llist.insertAfter(llist.head.next, 5)

    print('linked list:')
    llist.printList()

    print("\nAfter deleting an element:")
    llist.deleteNode(3)
    llist.printList()

    print()
    item_to_find = 3
    if llist.search(item_to_find):
        print(str(item_to_find) + " is found")
    else:
        print(str(item_to_find) + " is not found")

    llist.sortLinkedList(llist.head)
    print("Sorted List: ")
    llist.printList()
```