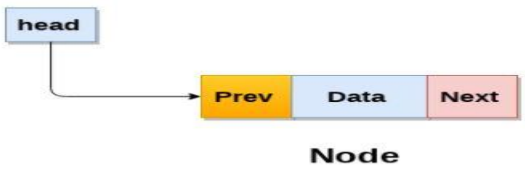
➢ Double linked list: -

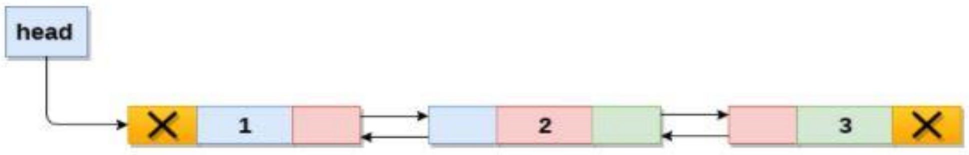
In a single linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a double linked list. A double linked list can be defined as follows...

“Double linked list is a sequence of elements in which every element has links to its previous element and next element in the sequence.”

In a double linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



Example: -



Operation on doubly linked list: -

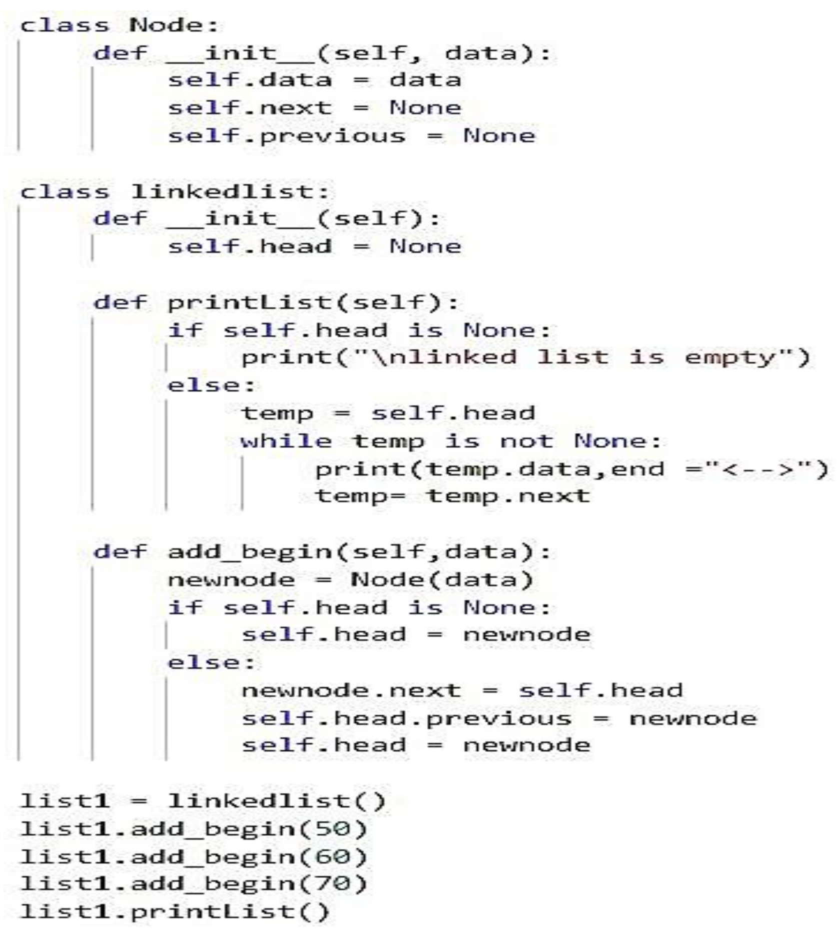
1. **Insertion**: - The insertion is categorized into the following categories.

   I. **Insertion at beginning: -**

   Adding the node into the linked list at beginning.

   - Step 1 - Create a **newNode** with given value and **newNode → previous** as **None**.
   - Step 2 - Check whether list is **Empty (head == None)**
   - Step 3 - If it is **Empty** then **newNode** to **head**.
   - Step 4 - If it is **not Empty** then, assign **head** to **newNode → next**, **newNode** to **head→ previous** and **newNode** to **head**.

   Program for insertion at beginning: -



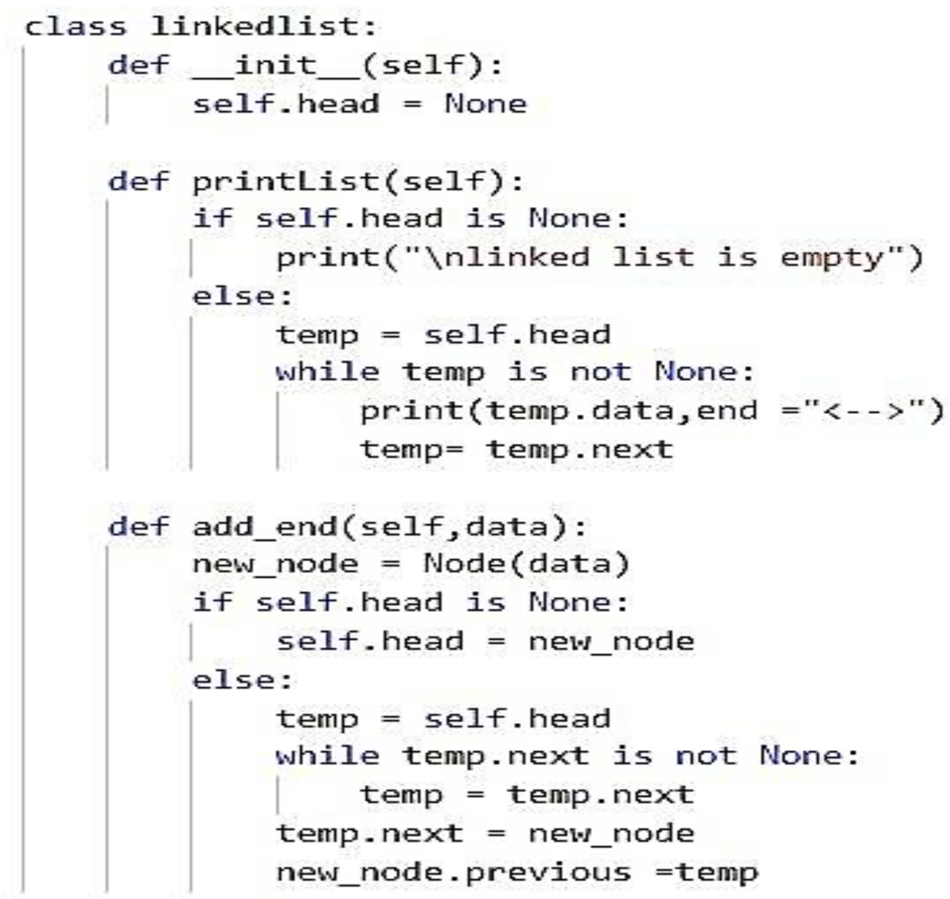
```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def add_begin(self,data):
        newnode = Node(data)
        if self.head is None:
            self.head = newnode
        else:
            newnode.next = self.head
            self.head.previous = newnode
            self.head = newnode

list1 = linkedlist()
list1.add_begin(50)
list1.add_begin(60)
list1.add_begin(70)
list1.printList()
```

I. **Insertion at the end: -**

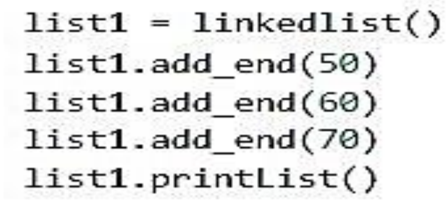
Adding the node into the linked list to the end

- Step 1 - Create a **newNode** with given value and **newNode →** **next** as **None**.
- Step 2 - Check whether list is **Empty** (**head** == **None**)
- Step 3 - If it is **Empty**, then **newNode** to **head**.
- Step 4 - If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** is equal to **None**).
- Step 6 - Assign **newNode** to **temp** → **next** and **temp** to **newNode** → **previous**.

Program for insertion at end: -



```python
class linkedlist:
    def __init__(self):
        self.head = None

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def add_end(self,data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = new_node
            new_node.previous =temp
```



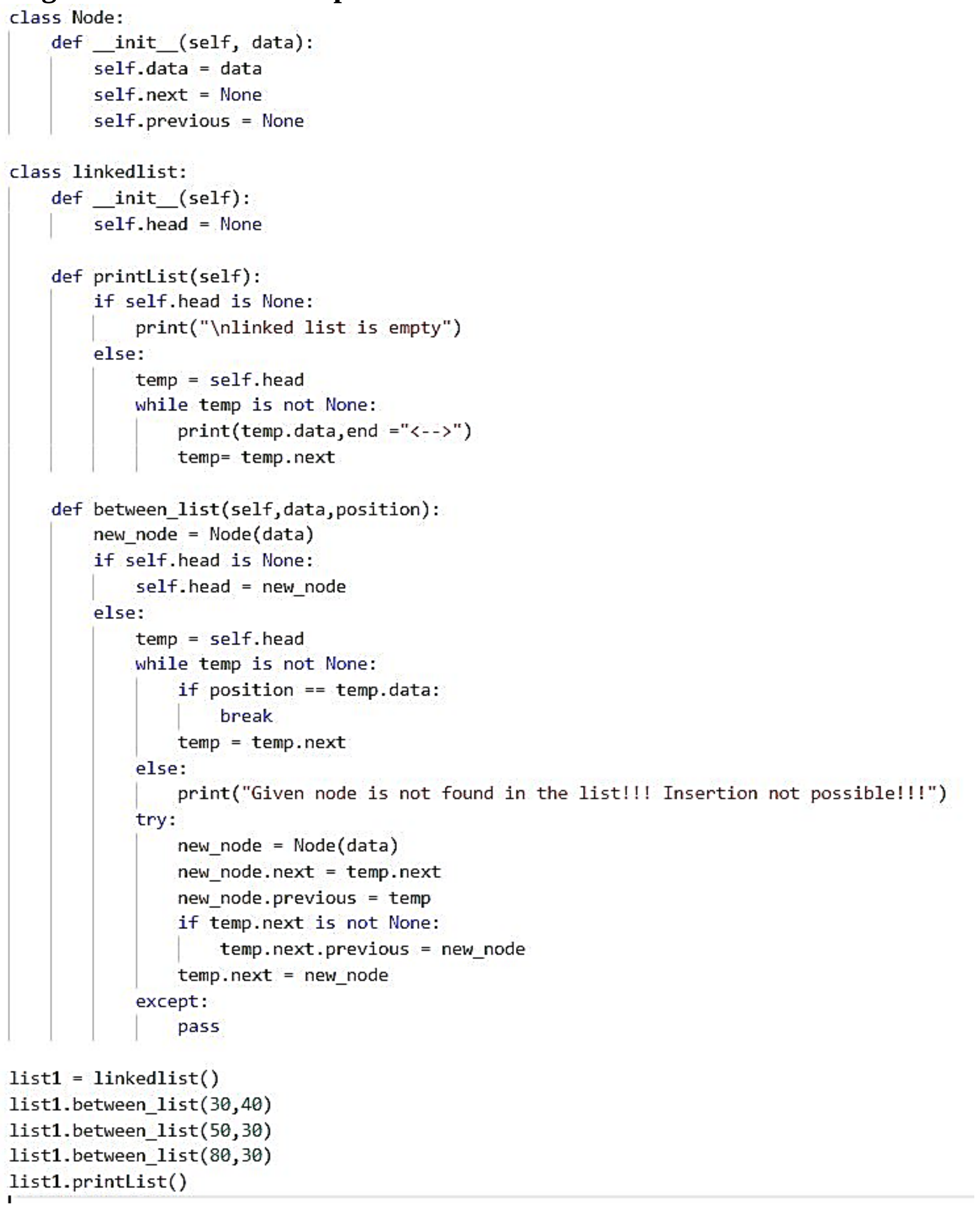
```python
list1 = linkedlist()
list1.add_end(50)
list1.add_end(60)
list1.add_end(70)
list1.printList()
```

II. Insertion after a specified node: -

Adding the node into the linked list after the specified node.

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty (head == None)**
- Step 3 - If it is Empty then, **head** = **newNode**
- Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- Step 7 - Finally, Set **newNode → next** to **temp → next, newNode→ previous** to **temp**, **temp→ next→ previous** to **newNode (**only if **temp→ next** is not **None)** and **temp → next = newNode**

Program for insertion at specified node: -



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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def between_list(self,data,position):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp is not None:
                if position == temp.data:
                    break
                temp = temp.next
            else:
                print("Given node is not found in the list!!! Insertion not possible!!!")
            try:
                new_node = Node(data)
                new_node.next = temp.next
                new_node.previous = temp
                if temp.next is not None:
                    temp.next.previous = new_node
                temp.next = new_node
            except:
                pass

list1 = linkedlist()
list1.between_list(30,40)
list1.between_list(50,30)
list1.between_list(80,30)
list1.printList()
```

2. **Deletion**: - The deletion is categorized into the following categories.

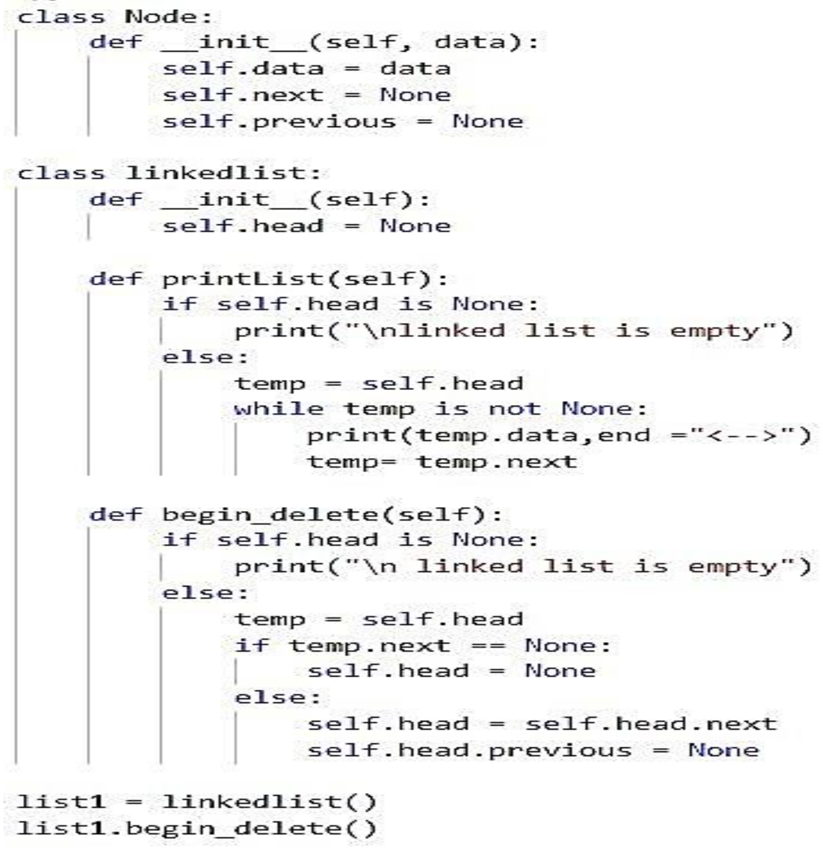
   I. Deletion at beginning: -

   Removing the node from beginning of the list

   - Step 1 - Check whether list is **Empty (head == None)**
   - Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
   - Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
   - Step 4 - Check whether list is having only one node **(temp → next == None)**

- Step 5 - If it is **TRUE** then set **head** = **None** and delete **temp** (Setting **Empty** list Conditions)
- Step 6 - If it is **FALSE** then set **head = temp → next, head→ previous** to **None** and delete **temp**.

Program for deletion at beginning: -



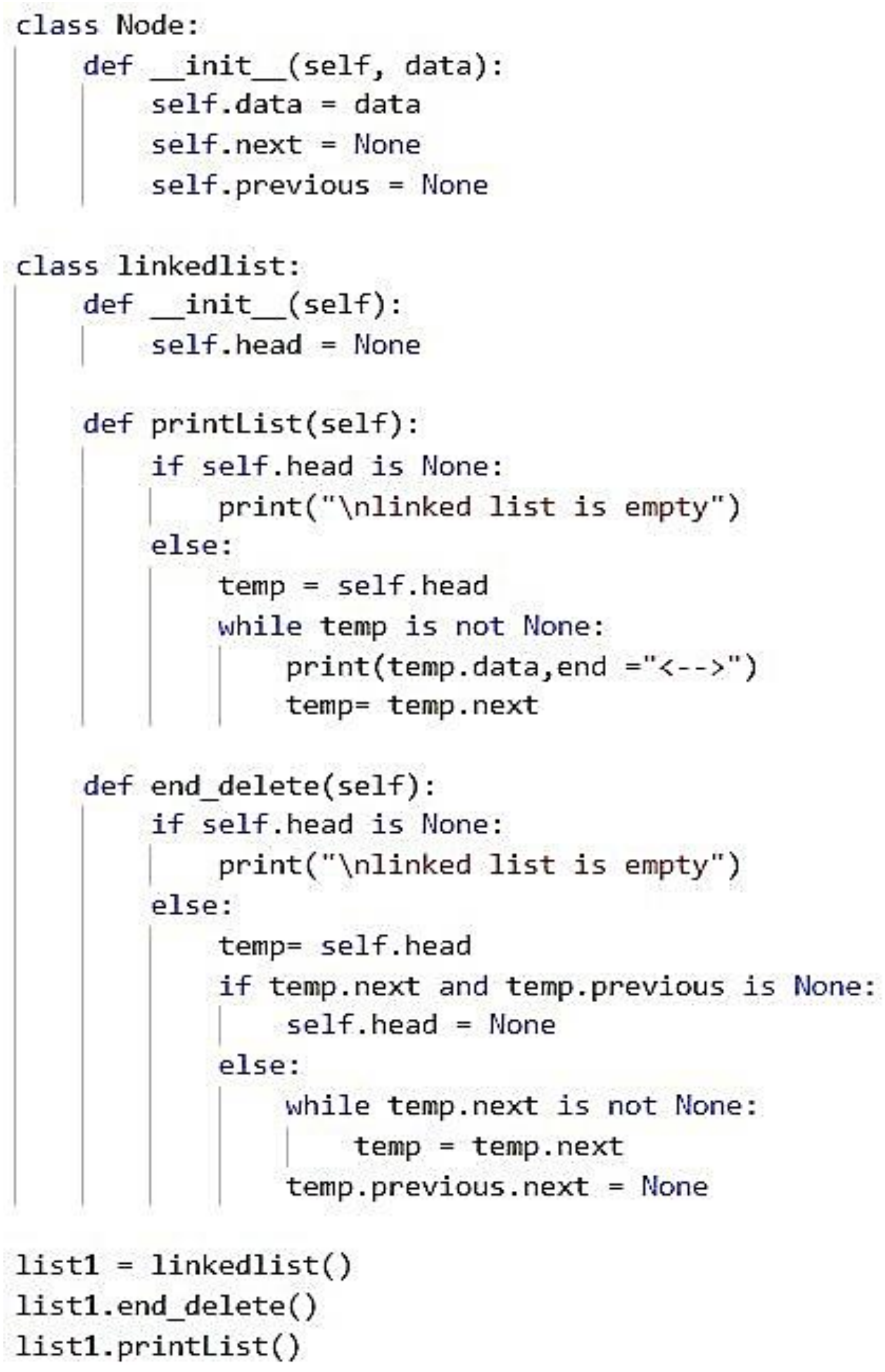
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def begin_delete(self):
        if self.head is None:
            print("\n linked list is empty")
        else:
            temp = self.head
            if temp.next == None:
                self.head = None
            else:
                self.head = self.head.next
                self.head.previous = None

list1 = linkedlist()
list1.begin_delete()
```

II. Deletion at the end: -

Removing the node from end of the list.

- Step 1 - Check whether list is **Empty (head == None)**
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is **Not Empty** then, define pointers **'temp'** and initialize **'temp'** with **head**.
- Step 4 - Check whether list has only one Node (**temp→ next and temp→ previous == None)**
- Step 5 - If it is **TRUE**. Then, set **head = None** and delete **temp**. And terminate the function. (Setting **Empty** list condition)
- Step 6 - If it is **FALSE**. Then, Keep moving the **temp** to its **next** node **→next** until it reaches to **temp → next** is equal to **None**).
- Step 7 - Finally, set **temp →previous→ next** = **None** and delete **temp**

Program for deletion at the end: -



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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def end_delete(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp= self.head
            if temp.next and temp.previous is None:
                self.head = None
            else:
                while temp.next is not None:
                    temp = temp.next
                temp.previous.next = None

list1 = linkedlist()
list1.end_delete()
list1.printList()
```

III. **Deletion after specified nodes: -**

Removing the node which is present just after the node containing the given data.

- Step 1 - Check whether list is **Empty (head == None**)

- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.

- Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.

- Step 4 - Keep moving the **temp** until it reaches to the exact node to be deleted or to the last node.

- Step 5 - If it is reached to the last node, then display **'Given node not found in the list! Deletion not possible!!!'** and terminate the function.

- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node or not

- Step 7 - If list has only one node and that is the node which is to be deleted then set **head** to **None** and delete **temp**

- Step 8 - If list contains multiple nodes, then check whether **temp** is the first node in the list **(temp == head).**

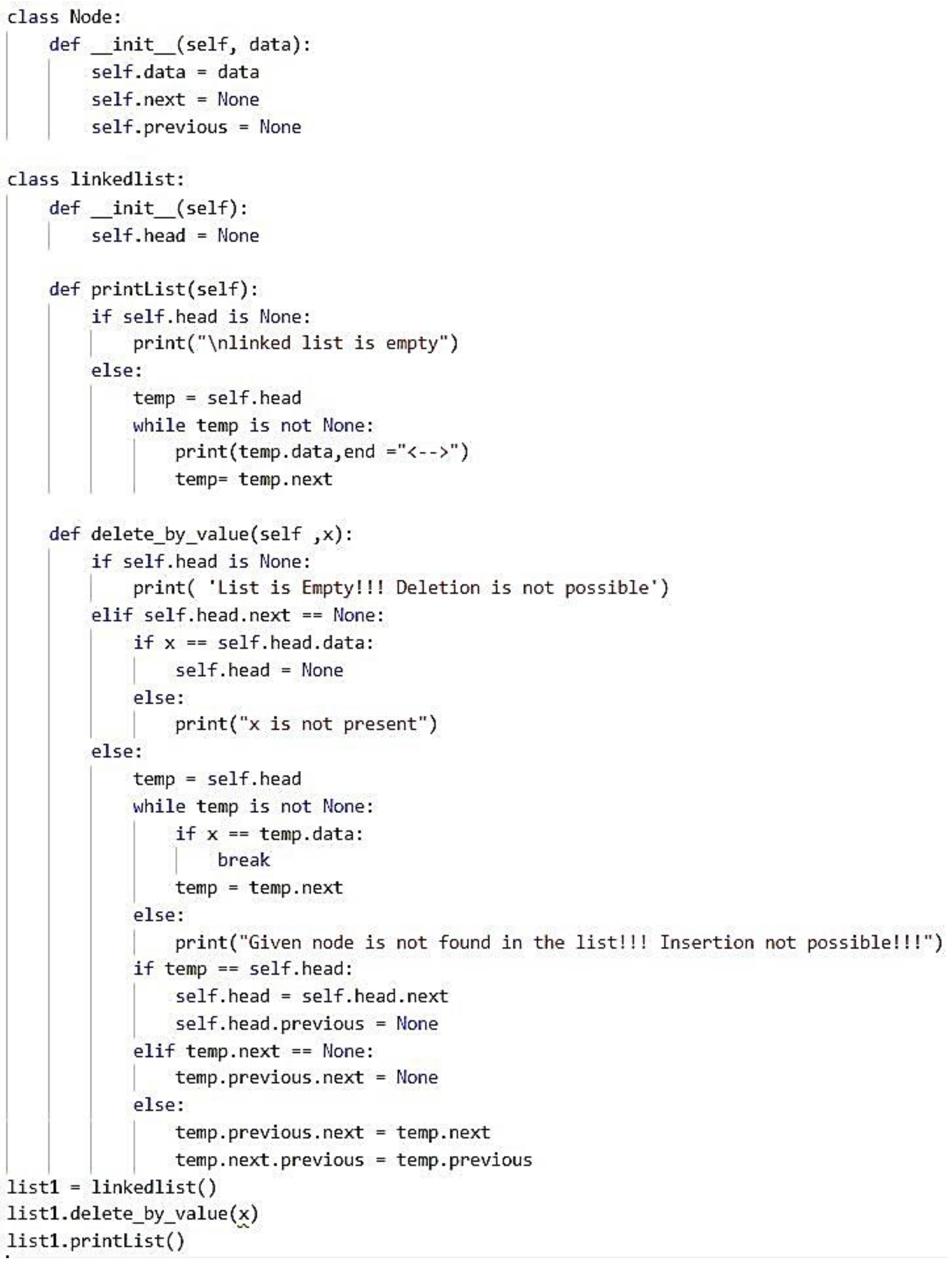
- Step 9 - If **temp** is the first node, then move the **head** to the next node (**head = head → next**), set **head** of **previous** to **None** (**head → previous = None)** and delete **temp**.

- Step 10 - If **temp** is not the first node, then check whether it is the last node in the list **(temp → next == None).**

- Step11- If **temp** is the last node, then set **temp** of **previous** of **next** to **None (temp→ previous→ next = None**) and delete **temp**

- Step 12 - If **temp** is not the first node and not the last node, then set **temp** of **previous** of **next** to **temp** of **next (temp → previous → next = temp → next), temp** of **next** of **previous** to **temp** of **previous** (**temp → next → previous = temp → previous**) and delete **temp**

Program for deletion of specific value:

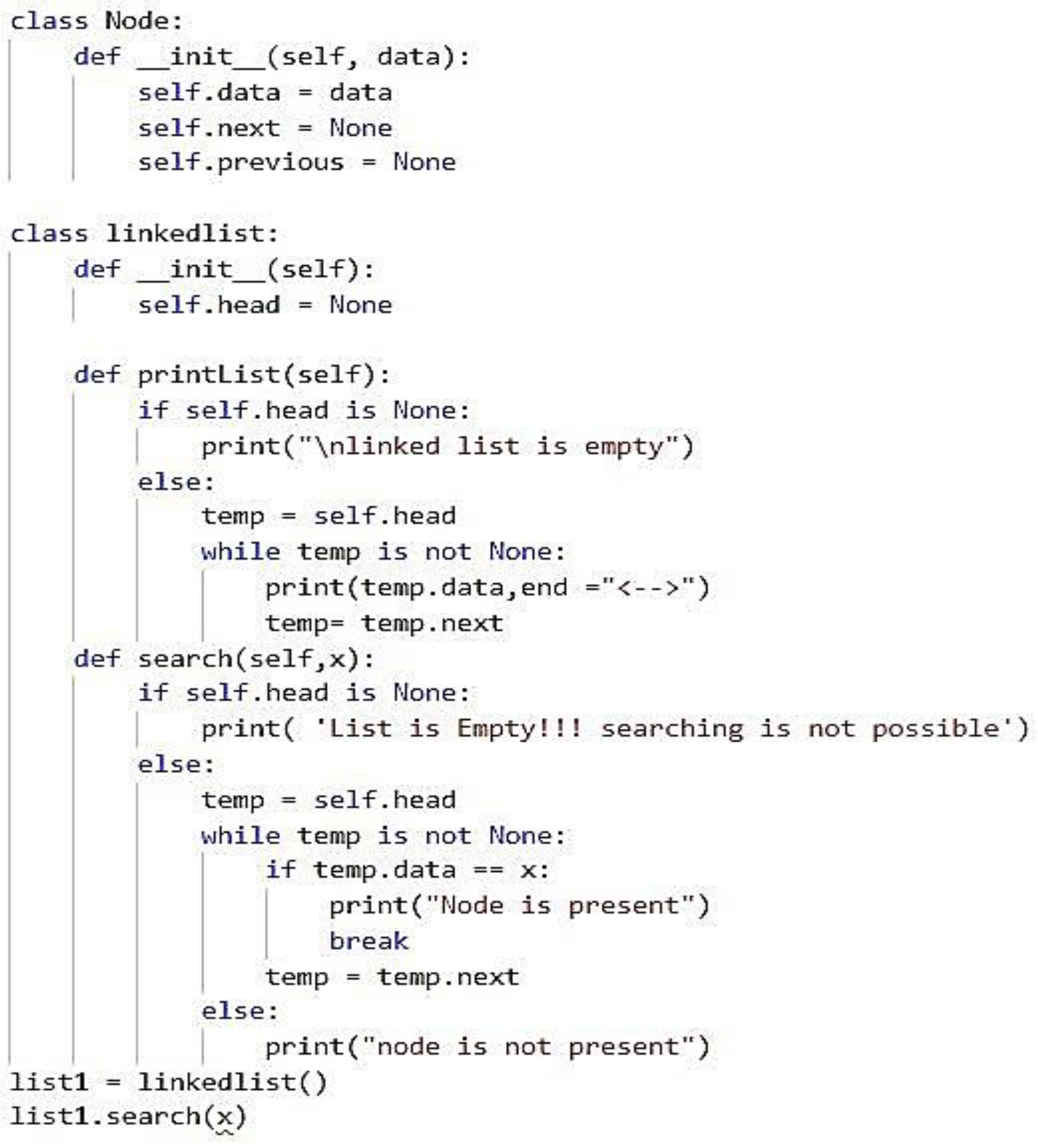


```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next

    def delete_by_value(self ,x):
        if self.head is None:
            print( 'List is Empty!!! Deletion is not possible')
        elif self.head.next == None:
            if x == self.head.data:
                self.head = None
            else:
                print("x is not present")
        else:
            temp = self.head
            while temp is not None:
                if x == temp.data:
                    break
                temp = temp.next
            else:
                print("Given node is not found in the list!!! Insertion not possible!!!")
            if temp == self.head:
                self.head = self.head.next
                self.head.previous = None
            elif temp.next == None:
                temp.previous.next = None
            else:
                temp.previous.next = temp.next
                temp.next.previous = temp.previous
list1 = linkedlist()
list1.delete_by_value(x)
list1.printList()
```

3. **Searching: -**

   Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

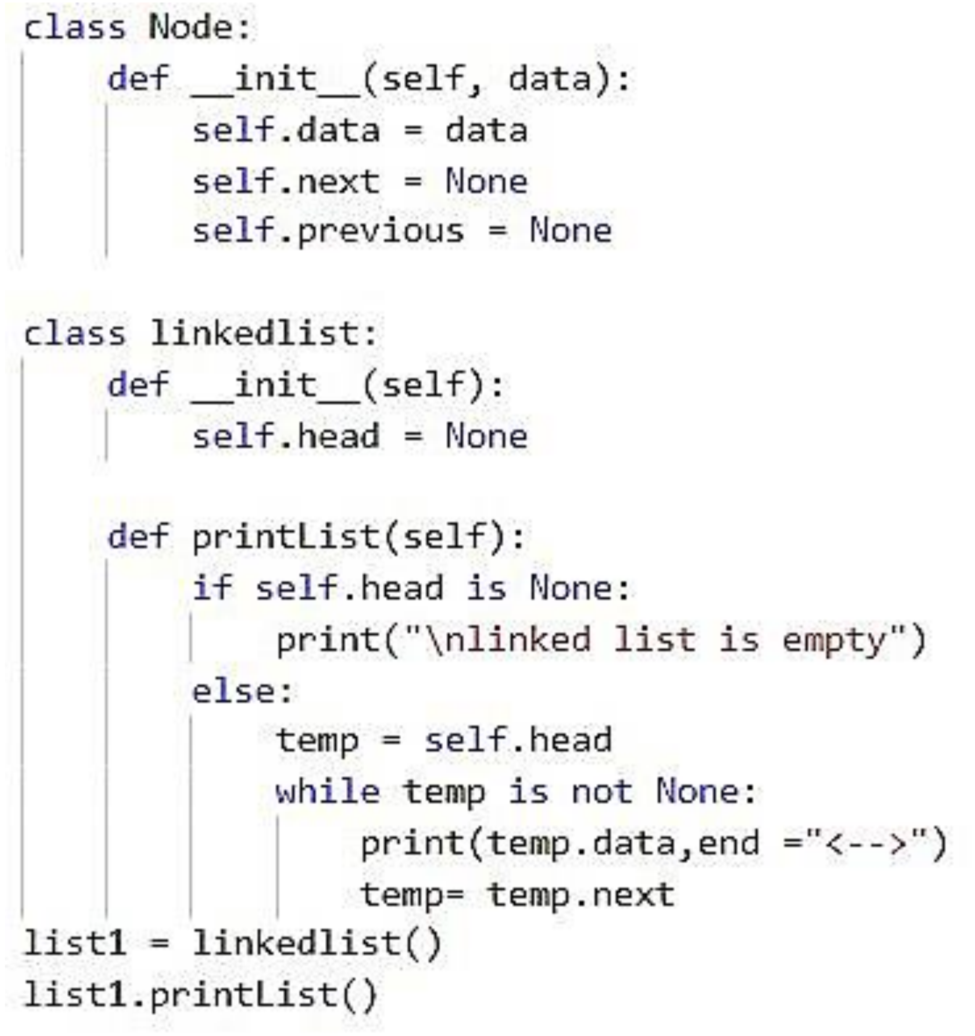
   - Step 1 - Check whether list is **Empty (head == None)**
   - Step 2 - If it is **Empty** then, display **'List is Empty!!! Searching is not possible'** and terminate the function.
   - Step 3 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
   - Step 5 - Keep moving the **temp** to its next node until it reaches to exact node to be search (until **temp→ data** is equal to **location**, here location is the node value which we want to search) and if present display **'Node is present'** And every time set **temp** to **temp→ next** before moving the 'temp ' to its next node.
   - Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!!'** and terminate the function. Otherwise move the **temp** to next node.



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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next
    def search(self,x):
        if self.head is None:
            print( 'List is Empty!!! searching is not possible')
        else:
            temp = self.head
            while temp is not None:
                if temp.data == x:
                    print("Node is present")
                    break
                temp = temp.next
            else:
                print("node is not present")
list1 = linkedlist()
list1.search(x)
```

4. **Traversing: -**
   Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.
   - Step 1 - Check whether list is **Empty (head == None)**
   - Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
   - Step 3 - If it is not Empty, then define a Node pointer **'temp'** and initialize with **head**.
   - Step 4 - Keep displaying **temp** → **data** with an arrow **(<===>)** until **temp** reaches to the last node

Program for traversing: -



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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            temp = self.head
            while temp is not None:
                print(temp.data,end ="<-->")
                temp= temp.next
list1 = linkedlist()
list1.printList()
```

➢ Application: -

- Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
- In the browser when we want to use the back or next function to change the tab, it used the concept of a doubly-linked list here.
- Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.

- It is used in music playing system where you can easily play the previous one or next one song as many times one person wants to. Basically, it provides full flexibility to perform functions and make the system user-friendly.
- In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).
- It is used in a famous game concept which is a deck of cards.