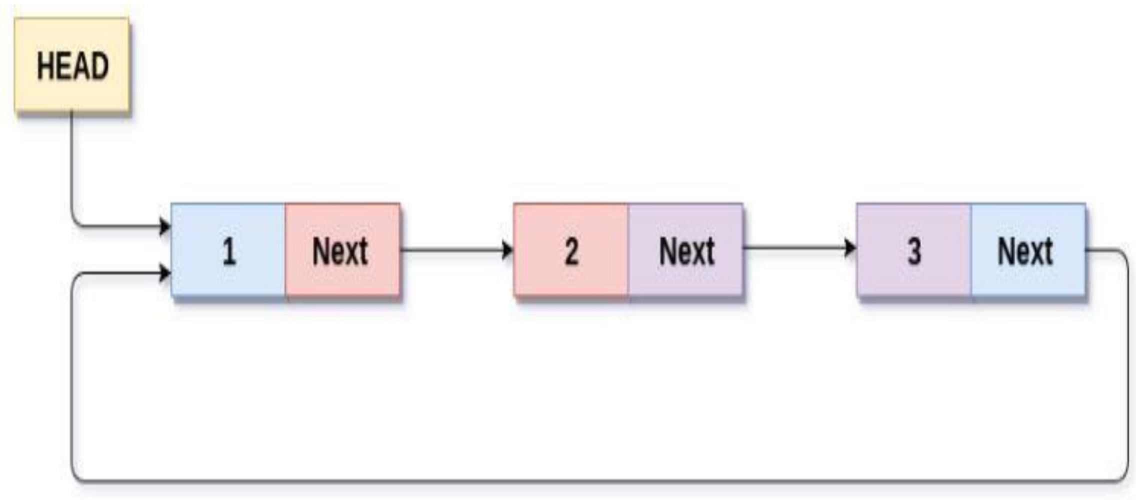
➢ Circular Linked List: -

A circular linked list is a sequence of elements in which every element has a link to its next element in the sequence and the last element has a link to the first element.

That means circular linked list is similar to the single linked list except that the last node points to the first node in the list



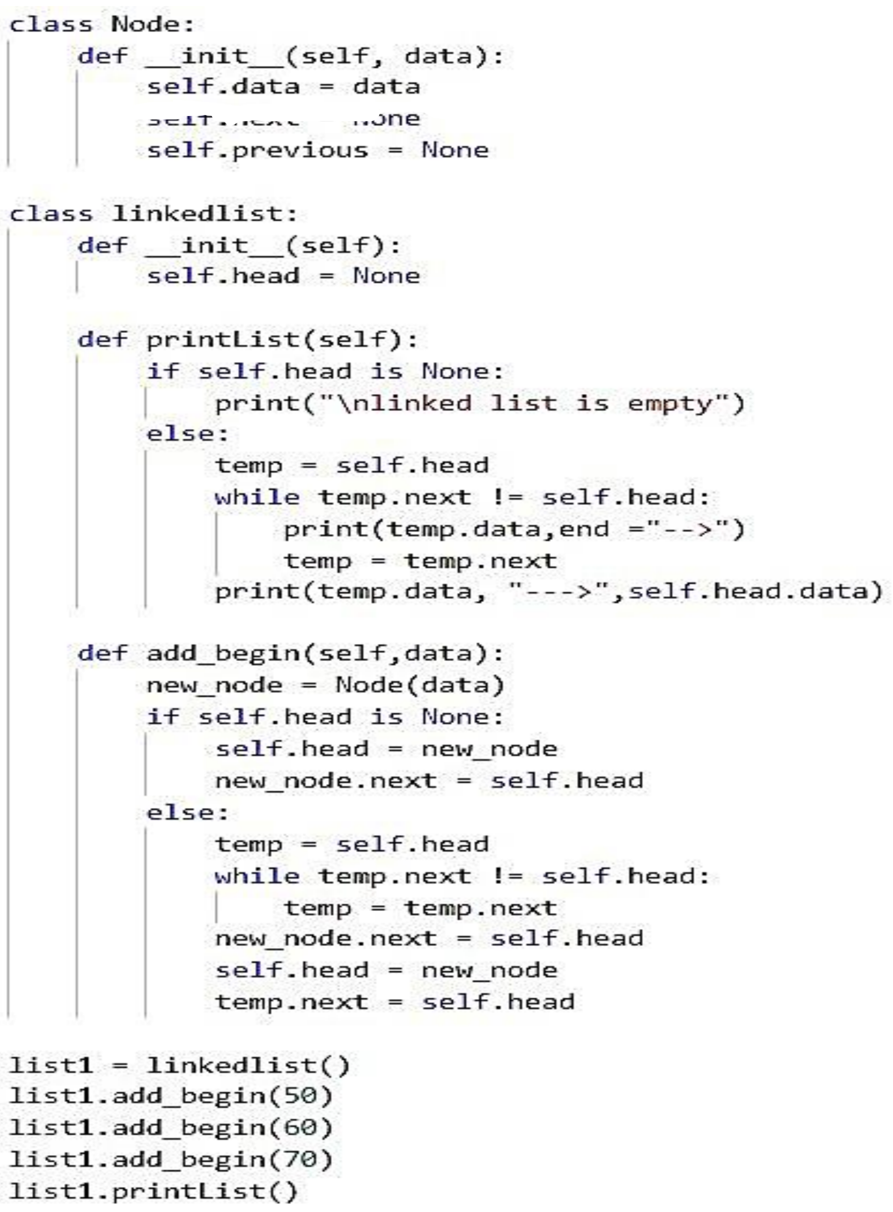
1. **Insertion**: - The insertion is categorized into the following categories.

   I. **Insertion at beginning: -**

   Adding a node into circular singly linked list at the beginning.

   - Step 1 - Create a **newNode** with given value.
   - Step 2 - Check whether list is **Empty (head == NULL**)
   - Step 3 - If it is **Empty** then, set **head** = **newNode** and newNode→ **next** = **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 (until **'temp → next == head'**).
   - Step 6 - Set **'newNode → next =head'**, **'head = newNode'** and **'temp → next = head'.**

   Program for insertion at beginning: -



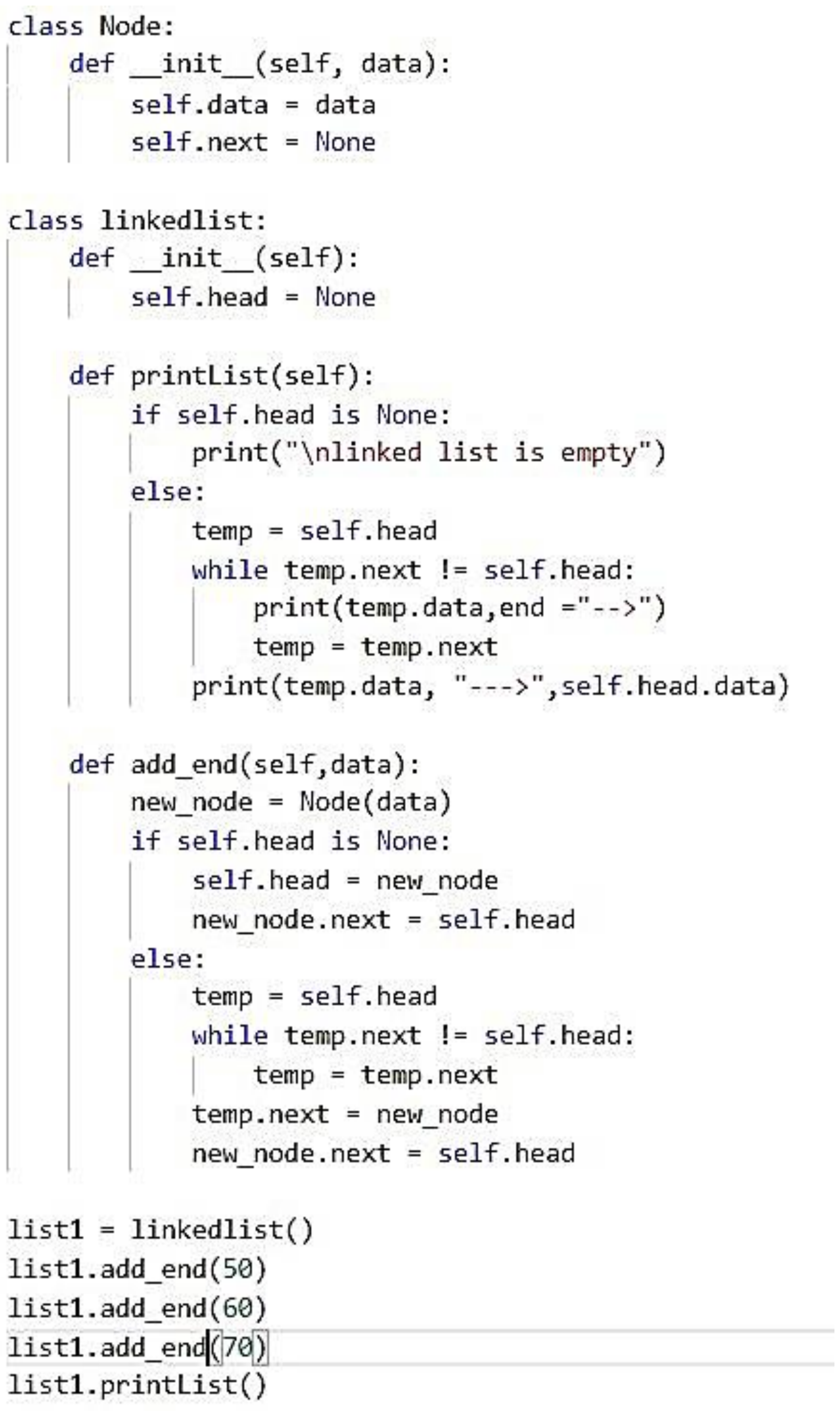
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.previous = None

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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

    def add_begin(self,data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            new_node.next = self.head
            self.head = new_node
            temp.next = self.head

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

II. Insertion at the end: -

Adding a node into circular singly linked list at the end.

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty (head == None).**
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **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 == head**).
- Step 6 - Set **temp** → **next = newNode** and **newNode** → **next = head.**

Program for insertion at the end: -

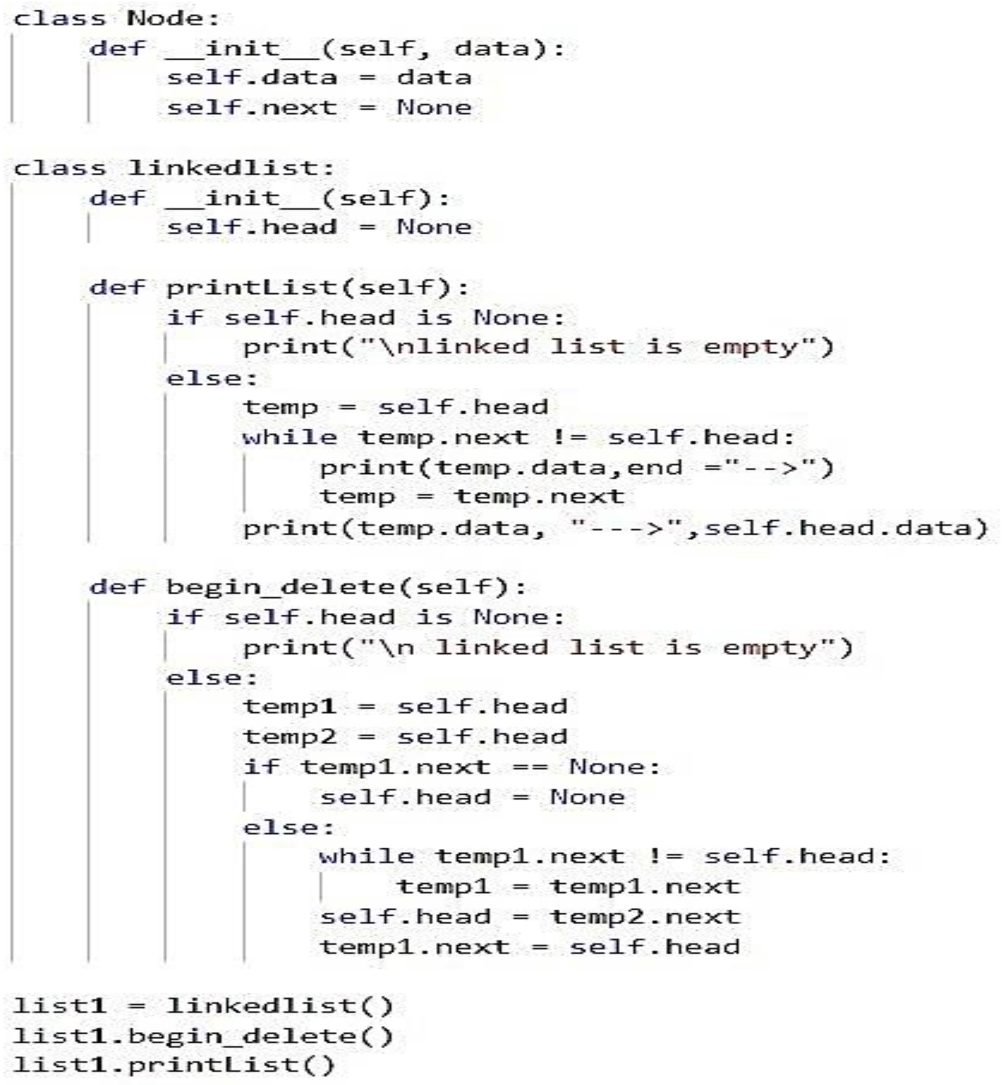


```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

    def add_end(self,data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head

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

2. **Deletion**: - The deletion is categorized into the following categories.
    I. **Deletion at beginning: -**
    Removing the node from circular singly linked list at the beginning.
    - 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 two Node pointers **'temp1'** and **'temp2'** and initialize both 'temp1' and 'temp2' with **head**.

- Step 4 - Check whether list is having only one node (**temp1 → next == head)**
- Step 5 - If it is **TRUE** then set **head = None** and delete **temp1** (Setting **Empty** list conditions)
- Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (Until **temp1 → next == head**)
- Step 7 - Then set **head = temp2 → next, temp1 → next = head** and delete **temp2**.

Program for deletion at beginning: -



```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = 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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

    def begin_delete(self):
        if self.head is None:
            print("\n linked list is empty")
        else:
            temp1 = self.head
            temp2 = self.head
            if temp1.next == None:
                self.head = None
            else:
                while temp1.next != self.head:
                    temp1 = temp1.next
                self.head = temp2.next
                temp1.next = self.head

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

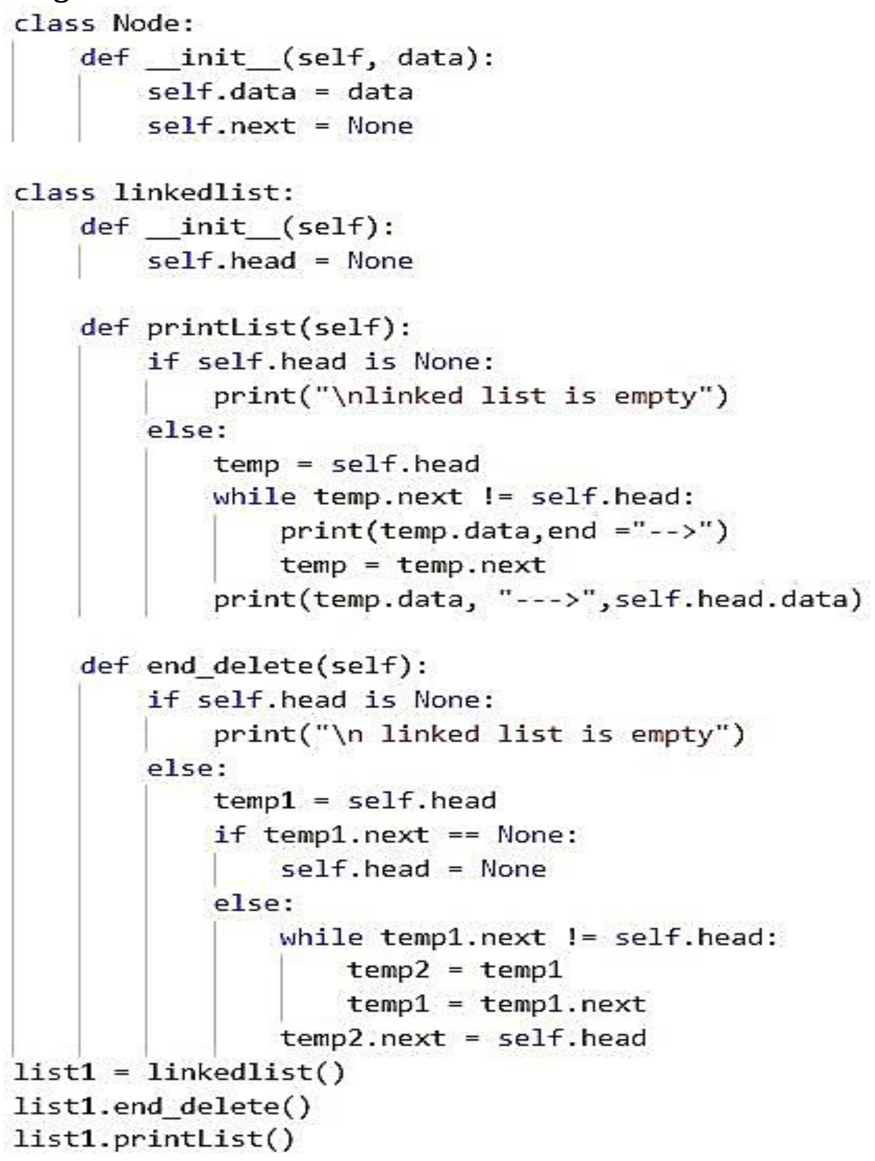
II. Deletion at the end: -

Removing the node from circular singly linked list at the end.

- 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 two Node pointers **'temp1'** and **'temp2'** and initialize **'temp1'** with head.

- Step 4 - Check whether list has only one Node (**temp1 → next == head**)
- Step 5 - If it is **TRUE**. Then, set **head = None** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- Step 6 - If it is **FALSE**. Then, set **'temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (Until **temp1 → next == head**)
- Step 7 - Set **temp2 → next = head** and delete **temp1.**

Program for deletion at the end: -

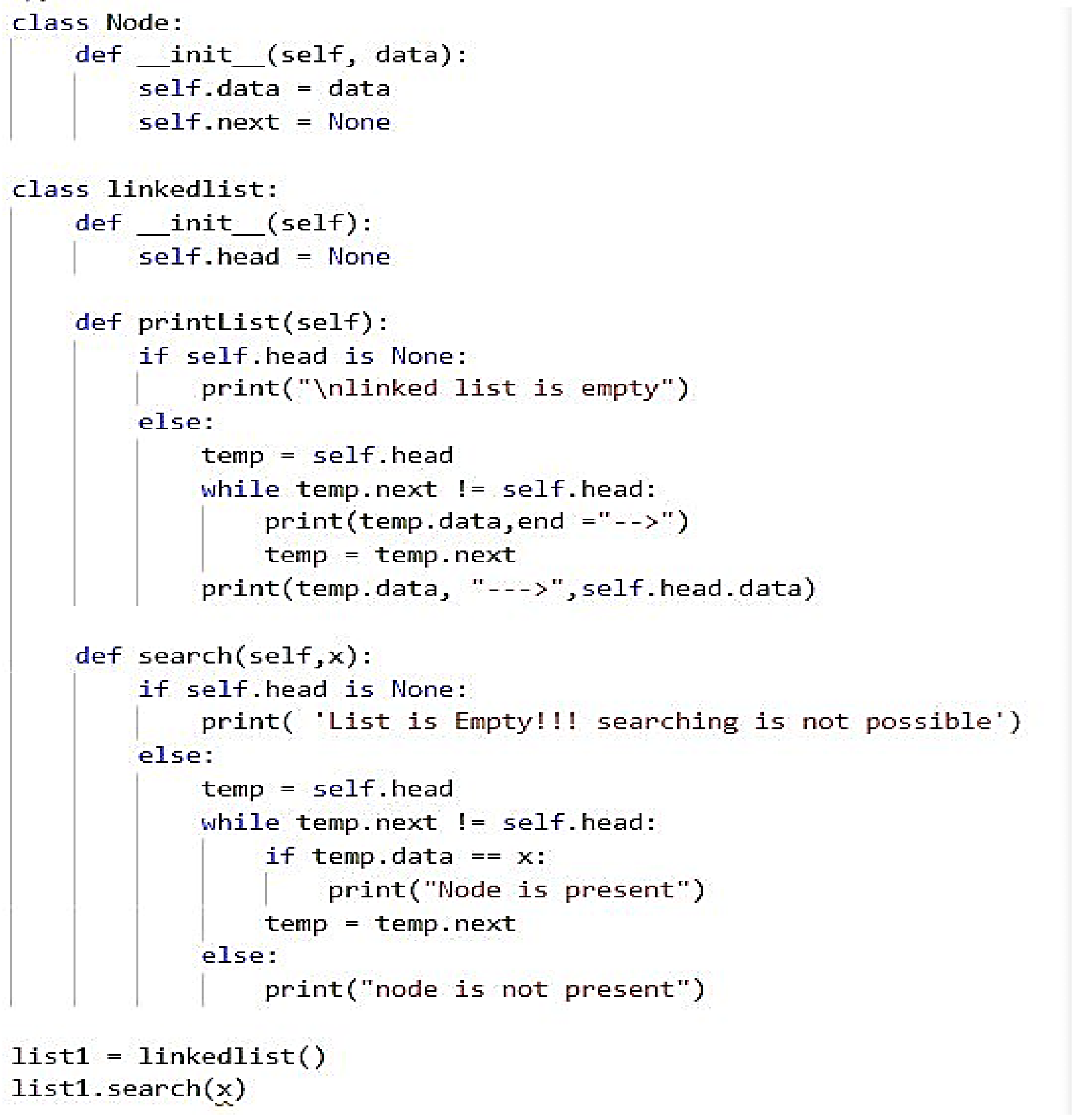


```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

    def end_delete(self):
        if self.head is None:
            print("\n linked list is empty")
        else:
            temp1 = self.head
            if temp1.next == None:
                self.head = None
            else:
                while temp1.next != self.head:
                    temp2 = temp1
                    temp1 = temp1.next
                temp2.next = self.head
list1 = linkedlist()
list1.end_delete()
list1.printList()
```

3. **Searching: -**
   Compare each element of the node with the given item and return the location at which the item is present in the list otherwise 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 (**temp→ next == head**) 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.

Program for Searching: -

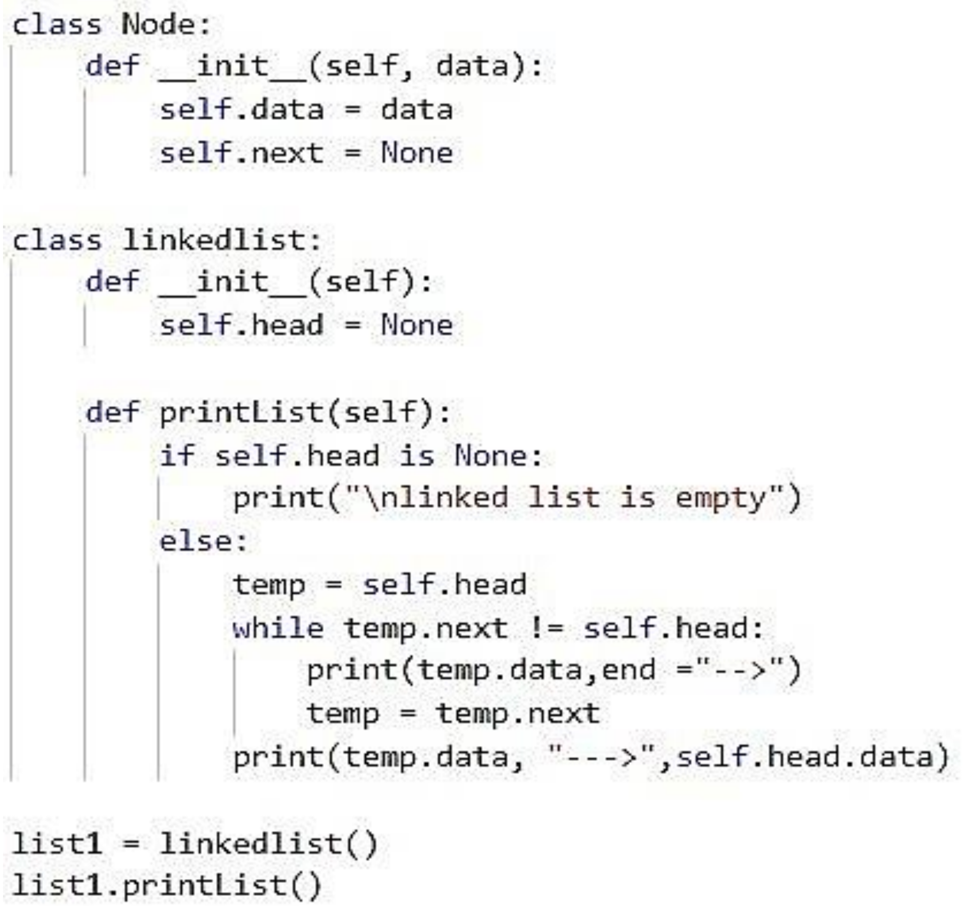


```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

    def search(self,x):
        if self.head is None:
            print( 'List is Empty!!! searching is not possible')
        else:
            temp = self.head
            while temp.next != self.head:
                if temp.data == x:
                    print("Node is present")
                temp = temp.next
            else:
                print("node is not present")

list1 = linkedlist()
list1.search(x)
```

4. **Traversing: -**
   Visiting each element of the list at least once in order to perform some specific operation.
   - 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
- Step 5 - Finally display **temp** → **data** with arrow pointing to **head** → **data.**

Program for traversing: -



```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = 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.next != self.head:
                print(temp.data,end ="-->")
                temp = temp.next
            print(temp.data, "--->",self.head.data)

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

➢ Application: -

- It can also be used to implement queues by maintaining a pointer to the last inserted node and the front can always be obtained as next of last.
- Circular Doubly Linked Lists are used for the implementation of advanced data structures like Fibonacci Heap. Also reference to the previous node can easily be found in this.
- It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism (this algorithm comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking).
- Multiplayer games use a circular list to swap between players in a loop.
- In photoshop, word, or any paint we use this concept in undo function.