Doubly Linked List

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes.

Each node contains three fields: two link fields (references to the previous and to the next node in the sequence of nodes) and one data field.

Advantages of Doubly Linked List

1. Traversal can be done on either side means both in forward as well as backward.

2. Deletion Operation is more efficient if the pointer to delete node is given.

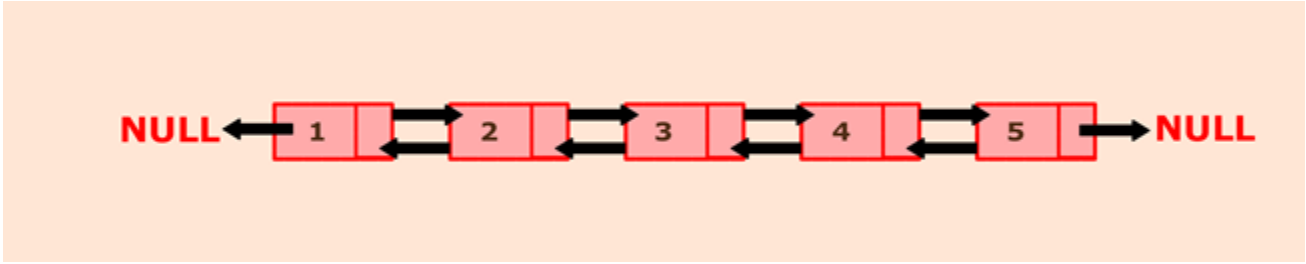
Disadvantages of Linked List

1. Since it requires extra pointer that is the previous pointer to store previous node reference.

2. After each operation like insertion-deletion, it requires an extra pointer that is a previous pointer which needs to be maintained.

So, a typical node in the doubly linked list consists of three fields:

- **Data** represents the data value stored in the node.
- **Previous** represents a pointer that points to the previous node.
- **Next** represents a pointer that points to the next node in the list.



The above picture represents a doubly linked list in which each node has two pointers to point to previous and next node respectively. Here, node 1 represents the head of the list. The previous pointer of the head node will always point to NULL. Next pointer of node one will point to node 2. Node 5 represents the tail of the list whose previous pointer will point to node 4, and the next will point to NULL.

ALGORITHM:

1. Define a Node class which represents a node in the list. It will have three properties: data, previous which will point to the previous node and next which will point to the next node.

2. Define another class for creating a doubly linked list, and it has two nodes: head and tail. Initially, head and tail will point to null.
3. addNode() will add node to the list:

- It first checks whether the head is null, then it will insert the node as the head.
- Both head and tail will point to a newly added node.
- Head's previous pointer will point to null and tail's next pointer will point to null.
- If the head is not null, the new node will be inserted at the end of the list such that new node's previous pointer will point to tail.
- The new node will become the new tail. Tail's next pointer will point to null.

a. display() will show all the nodes present in the list.

- Define a new node 'current' that will point to the head.
- Print current.data till current points to null.
- Current will point to the next node in the list in each iteration.

PROGRAM:

```
#Represent a node of doubly linked list
class Node:
    def __init__(self,data):
        self.data = data;
        self.previous = None;
        self.next = None;

class DoublyLinkedList:
    #Represent the head and tail of the doubly linked list
    def __init__(self):
        self.head = None;
        self.tail = None;

    #addNode() will add a node to the list
```

```
15.    def addNode(self, data):
16.        #Create a new node
17.        newNode = Node(data);
18.
19.        #If list is empty
20.        if(self.head == None):
21.            #Both head and tail will point to newNode
22.            self.head = self.tail = newNode;
23.            #head's previous will point to None
24.            self.head.previous = None;
25.            #tail's next will point to None, as it is the last node of the list
26.            self.tail.next = None;
27.        else:
28.            #newNode will be added after tail such that tail's next will point to newNode
29.            self.tail.next = newNode;
30.            #newNode's previous will point to tail
31.            newNode.previous = self.tail;
32.            #newNode will become new tail
33.            self.tail = newNode;
34.            #As it is last node, tail's next will point to None
35.            self.tail.next = None;
36.
37.    #display() will print out the nodes of the list
38.    def display(self):
39.        #Node current will point to head
40.        current = self.head;
41.        if(self.head == None):
42.            print("List is empty");
43.            return;
44.        print("Nodes of doubly linked list: ");
45.        while(current != None):
46.            #Prints each node by incrementing pointer.
47.            print(current.data),;
```

```
48.             current = current.next;
49.
50. dList = DoublyLinkedList();
51. #Add nodes to the list
52. dList.addNode(1);
53. dList.addNode(2);
54. dList.addNode(3);
55. dList.addNode(4);
56. dList.addNode(5);
57.
58. #Displays the nodes present in the list
59. dList.display();
```

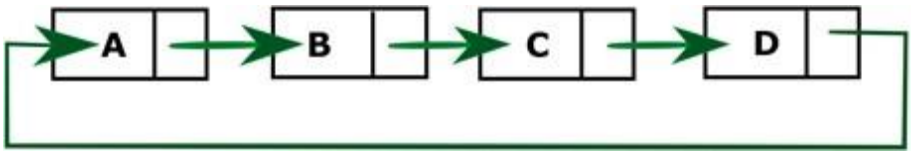
Output:

```
Nodes of doubly linked list:
1 2 3 4 5
```

Circular Linked List

The circular linked list is a kind of linked list. First thing first, the node is an element of the list, and it has two parts that are, data and next. Data represents the data stored in the node, and next is the pointer that will point to the next node. Head will point to the first element of the list, and tail will point to the last element in the list. In the simple linked list, all the nodes will point to their next element and tail will point to null.

The circular linked list is the collection of nodes in which tail node also point back to head node. The diagram shown below depicts a circular linked list. Node A represents head and node D represents tail. So, in this list, A is pointing to B, B is pointing to C and C is pointing to D but what makes it circular is that node D is pointing back to node A.



ALGORITHM:

1. Define a Node class which represents a node in the list. It has two properties data and next which will point to the next node.
2. Define another class for creating the circular linked list, and it has two nodes: head and tail. It has two methods: add() and display() .
3. add() will add the node to the list:

- It first checks whether the head is null, then it will insert the node as the head.
- Both head and tail will point to the newly added node.
- If the head is not null, the new node will be the new tail, and the new tail will point to the head as it is a circular linked list.

a. display() will show all the nodes present in the list.

- Define a new node 'current' that will point to the head.
- Print current.data till current will points to head
- Current will point to the next node in the list in each iteration.

PROGRAM:

```
#Represents the node of list.
class Node:
  def__init__(self,data):
    self.data = data;
    self.next = None;

class CreateList:
  #Declaring head and tail pointer as null.
  def__init__(self):
    self.head = Node(None);
    self.tail = Node(None);
    self.head.next = self.tail;
    self.tail.next = self.head;

```

```
15.  #This function will add the new node at the end of the list.
16.  def add(self,data):
17.    newNode = Node(data);
18.    #Checks if the list is empty.
19.    if self.head.data is None:
20.      #If list is empty, both head and tail would point to new node.
21.      self.head = newNode;
22.      self.tail = newNode;
23.      newNode.next = self.head;
24.    else:
25.      #tail will point to new node.
26.      self.tail.next = newNode;
27.      #New node will become new tail.
28.      self.tail = newNode;
29.      #Since, it is circular linked list tail will point to head.
30.      self.tail.next = self.head;
31.
32.  #Displays all the nodes in the list
33.  def display(self):
34.    current = self.head;
35.    if self.head is None:
36.      print("List is empty");
37.      return;
38.    else:
39.        print("Nodes of the circular linked list: ");
40.        #Prints each node by incrementing pointer.
41.        print(current.data),
42.        while(current.next != self.head):
43.            current = current.next;
44.            print(current.data),
45.
46.
47. class CircularLinkedList:
```

```
48. cl = CreateList();
49. #Adds data to the list
50. cl.add(1);
51. cl.add(2);
52. cl.add(3);
53. cl.add(4);
54. #Displays all the nodes present in the list
55. cl.display();
```

Output:

```
Nodes of the circular linked list:
1 2 3 4
```