Doubly Linked List

Nakul V Sunil
AM.SC.U4AIE23048
AIE - A

Explanation :

Constructors for class Node and Doubly Linked List

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

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

Insertion at the Front

Below is the method for doing insertion at the front first the function checks whether the head of the LL is None or not if it is then the head is made the newnode. The else part handles the normal case, that is a newnode is inserted at the front and head is updated to the newnode.

```
# Insertion at the front
   def insertFront(self, newData):
       newNode = Node(newData)
       if self.head is None:
           self.head = newNode
       else:
           newNode.next = self.head
           self.head.prev = newNode
           self.head = newNode
```

Insertion at the End

Below is the method for doing insertion at the end first the function checks whether the head of the LL is None or not if it is then the head is made the newnode. A variable last is created and traversed to the last node of the LL for inserting the newnode at the end.

```
def insertEnd(self, newData):
       newNode = Node(newData)
       if self.head is None:
           self.head = newNode
           return
       last = self.head
       while last.next:
           last = last.next
       last.next = newNode
       newNode.prev = last
```

Insertion at a given Position

Below is the method for insertion at a given position: if the given position is 0 then the node is inserted before the head then the newnode is made the head.Using a counter we iterate to the previous position to which the node is to be inserted then the next part of the current.next is made newnnode then necessary connections between current and newnode are made.

```
def insertPos(self, position, newData):
   if position == 0:
       newNode = Node(newData)
       newNode.next = self.head
       if self.head:
           self.head.prev = newNode
       self.head = newNode
       return

   newNode = Node(newData)
   current = self.head
   count = 0
   while current is not None and count < position - 1:
       current = current.next
       count += 1
```

```
if current is None:
      print("Position is beyond the length of the list")
      return
     newNode.next = current.next
  if current.next is not None:
      current.next.prev = newNode
  current.next = newNode
  newNode.prev = current
```

Deletion at the Front

Below is the method for doing Deletion at the front. First the function checks whether the head of the LL is None or not if it is then returns nothing. Then checks whether it is a single node if it is then head is made None.The else part handles normal deletion at the front by making head node as the head.next node.

```
def deleteFront(self):
       if self.head is None:  # If the list is empty
           print("The list is empty.")
           return
       if self.head.next is None:  # If there's only one element
           self.head = None
       else:
           self.head = self.head.next
           self.head.prev = None
```

Deletion at the End

Below is the method for doing Deletion at the end. First the function checks whether the head of the LL is None or not if it is then returns nothing. Then checks whether it is a single node if it is then head is made None.The else part handles normal deletion at the end by creating a pointer which traverses through the LL and points to the second last node then next part of the second last node is made none

```
def deleteEnd(self):
       if self.head is None:  # If the list is empty
           print("The list is empty.")
           return
       if self.head.next is None:  # If there's only one element
           self.head = None
       else:
           last = self.head
```

```
        while last.next:
            last = last.next
        last.prev.next = None
```

Deletion from a given Position

Below is the method for deletion from a given position: Checks if head is None or not then return nothing .Then checks if the given position id 0 then we delete the front node.Using a counter we iterate to the previous position to which the node is to be inserted checks whether the current position is None or not . The prev part of the current.next node is made current.prev then the next part of the previous node is made the next of the current node.

```
# Deletion at a given position
   def deleteAtPos(self, pos):
       if self.head is None:
           print("The list is empty.")
           return
       temp = self.head
       if pos == 0:  # If the position is the head
           self.deleteFront()
           return
       for _ in range(pos):
           if temp is None:  # If position is greater than the number of nodes
               print("Position out of bounds.")
               return
           temp = temp.next
       if temp.next:
           temp.next.prev = temp.prev
       if temp.prev:
           temp.prev.next = temp.next
```

Display

Below is the method for displaying both forward and backward a pointer is made which points to the head using this pointer we traverse through the LL and another pointer last is made to store the last node to print backwards.

```
def displayFront(self):
       current = self.head
       last = None
       print("Traversal in forward direction:")
       while current:
           print(current.data, end=" ")
           last = current  # Store the last node for reverse traversal
           current = current.next




        print("\nTraversal in reverse direction:")
       while last:
           print(last.data, end=" ")
           last = last.prev
       print()
```