Section A4 Linked Lists

The linked list data structure is made from a chain of nodes. Each node consists of two components, a *value* and a *pointer* to the next node in the chain.

a single node

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

Join nodes to get a linked list

```
#Linked List class contains a Node object
class LinkedList:

    #initialise head
    def __init__(self):
        self.head = None

#Start with the empty list
llist = LinkedList()
```

Traversing a Linked List

```
def traverse(self):
    cur_node = self.head
    while cur_node != None:
        print(cur_node.data)
        cur_node = cur_node.next
```

Inserting an Element to a Linked List

There are three different ways that you could insert an element into a linked list.

- Insert at the first position (*prepend*)
- Insert at the last position (*append*)
- Insert on anywhere in between (*insert*)

Prepend
To insert an element at the beginning of a linked list, you first need to create a node with the given data. Then, you assign its next reference to the first node (i.e., where the head is pointing). Next, you point the head reference to the newly inserted node.

```
def prepend(self, data):

    new_node = Node(data) #creating the new node

    if self.head == None: #linked list is empty
        self.head = new_node #give the head reference to the new node
    else:
        new_node.next = self.head #connect the new node to the linked list
        self.head = new_node #give the head reference to the new node
```

Append

To insert an element at the end of a linked list, you have to find the node where the next element (or pointer) refers to None (i.e., the last node in the linked list). Then, you create a new node with the given data and point the next element of the last node to the newly created node.

```
def append(self,data):
    new_node = Node(data) #creating the new node

    if self.head == None: #linked list is empty
        self.head = new_node #give the head reference to the new node
    else:
        #traversing the linked list to find the last node
        curr = self.head
        while curr.next != None:
            curr = curr.next
        curr.next = new_node #point the next element of the last node to the newly created node
```

Insert

first, we can count the nodes till we reach that given position. Then, we point the next element of the new node to the next node of the current node and the next reference of the current node to the new node.

```
def insert(self, data, position):
        new_node = Node(data)

        cur_node = self.head
        prev = None
        count = 0
        while cur_node != None and count < position:
            prev = cur_node
            cur_node = cur_node.next
            count += 1
        if cur_node != self.head:
            prev.next = new_node
            new_node.next = cur_node
        else:
            new_node.next = cur_node
            self.head = new_node
```

Deleting an Element of a Linked List

Same as you saw in the previous section, there are three different ways that you could delete an element of a linked list.

- Delete from the start
- Delete from the end
- Delete between two nodes

Delete First Node
To delete the first node, first, you need to check if the head of a linked list is pointing to None. If it is pointing to None, then you could mention that the linked list is empty. Otherwise (i.e., if the linked list is not empty), you need to delete the current node referred by head and move the head pointer to the next node.

```
def delete_first(self):
    if self.head == None:
        print("Empty Linked List")
    else:
        cur_node = self.head
        self.head = self.head.next
        del cur_node
```

Delete Last Node
To delete the last node, you need to traverse the nodes in a linked list and check if the next pointer of the next node from the current node points is None. If yes, this means that the next node of the current node is the last node of the given linked list and needs to be deleted.

```
def delete_last(self):
    if self.head == None:
        print("Empty Linked List")
    else:
        cur_node = self.head
        prev = None
        while cur_node.next != None:
            prev = cur_node
            cur_node = cur_node.next
        prev.next = None
        del cur_node
```

Delete by Position
To delete a node in between two nodes in a given linked list, you need to check at every node whether the position of the next node is the node to be deleted. If yes, you need to delete the next node and assign the next reference to the next node of the node being deleted.

```
def delete_position(self, position):
    if self.head == None:
        print("Empty Linked List")
    else:
        cur_node = self.head
        prev = None
        count = 0
        while cur_node != None and count < position:
            prev = cur_node
            cur_node = cur_node.next
            count += 1
        if cur_node == self.head:
            self.head = cur_node.next
            del cur_node
        else:
            prev.next = cur_node.next
            del cur_node
```

Doubly Linked Lists

the doubly linked list can be traversed in both directions, making insertion and deletion operations easier to perform. Therefore, a node in a doubly linked list comprises of three components.

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

Node & Joining Nodes

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

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

Traverse

```
def traverse(self):
    #Node current will point to head
    current = self.head
    if self.head == None:
        print("List is empty")
        return
    print("Nodes of doubly linked list: ")
    while current != None:
        #prints each node by incrementing pointer
        print(current.data)
        current = current.next
```

Append

When you need to append a new node (i.e., insert at the end of a doubly linked list), you need to first check whether the head is None. If yes, then, both head and tail will point to this newly added node and head's previous pointer will point to None and the tail's next pointer will point to None. If the head is not None, the new node will be inserted at the end of the given doubly linked list such that the new node's previous pointer will point to the tail. The new node will become the new tail and the tail's next pointer will point to None.

```
def append(self, data):
    #create a new node
    new_node = Node(data)

    #if linked list is empty
    if self.head == None:
        #both head and tail will point to new_node
        self.head = self.tail = new_node
        #head's previous will point to None
        self.head.previous = None
        #tail's next will point to None, as it is the last node of the list
        self.tail.next = None
    else:
        #new_node will be added after tail such that tail's next will point to new_node
        self.tail.next = new_node
        #new_node's previous will point to tail
        new_node.previous = self.tail
        #new_node will become new tail
        self.tail = new_node
        #since it is last node, tail's next will point to None
        self.tail.next = None
```

Delete

how to delete the first node in a doubly linked list. First, you need to check whether the head is None. If it is not empty, you need to check whether the linked list has only one node. If the list has only one node, you need to set both head and tail to None. If the linked list has more than one node, then, the head will point to the next node in the list and delete the old head node.

```
def delete_first(self):
    #checks whether list is empty
    if self.head == None:
        return
    else:
        #checks whether the list contains only one element
        if self.head != self.tail:
            #head will point to next node in the list
            self.head = self.head.next
            #Previous node to current head will be made None
            self.head.previous = None

        #If the list contains only one element
        #then, it will remove the node, and now both head and tail will point to None
        else:
            self.head = self.tail = None
```