# Doubly Linked List in Python

## Lesson Plan

PW SKILLS

# Doubly Linked List in Python

A Doubly Linked List in Python is similar to a singly linked list, except that each node also contains a pointer to the previous node. This means that in a doubly linked list we can traverse not only in the forward direction but also in the backward direction, which is not possible with a plain singly linked list.

## Inserting a new node in a Doubly Linked List

Inserting a new node in a doubly linked list is very similar to inserting a new node in a singly linked list, with the additional requirement to maintain the link of the previous node. A node can be inserted in a Doubly Linked List in several ways:

1. At the front of the DLL.
2. In between two nodes.
3. After a given node.
4. Before a given node.
5. At the end of the DLL.

## Add a node at the front in a Doubly Linked List

The new node is always added before the head of the given Linked List. The task can be performed by using the following steps:

1. Allocate a new node.
2. Put the required data in the new node.
3. Make the next of the new node point to the current head of the doubly linked list.
4. Make the prev of the current head point to the new node.
5. Lastly, point head to the new node.

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

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

    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        if self.head is not None:
self.head.prev = new_node
        self.head = new_node

    def print_list(self, node):
        while node is not None:
            print(node.data, end=" ")
            node = node.next
        print()
```

**Time Complexity: O(1)**

**Auxiliary Space: O(1)**

## Add a node after a given node in a Doubly Linked List

We are given a pointer to a node as prev_node, and the new node is inserted after the given node. This can be done using the following steps:

1. Allocate a new node.
2. Put the data in the new node.
3. Point the next of the new node to the next of prev_node.
4. Point the next of prev_node to the new node.
5. Point the prev of the new node to prev_node.
6. Change the prev of the new node's next node.

```
def insert_after(self, prev_node, new_data):
        if prev_node is None:
            print("The given previous node cannot be NULL")
            return
        new_node = Node(new_data)
        new_node.next = prev_node.next
        prev_node.next = new_node
        new_node.prev = prev_node
        if new_node.next is not None:
            new_node.next.prev = new_node
```

**Time Complexity: O(1)**

**Auxiliary Space: O(1)**

## Add a node before a given node in a Doubly Linked List

Let the pointer to this given node be next_node. This can be done using the following steps:

1. Allocate a new node.
2. Put the data in the new node.
3. Set the prev pointer of this new node to the prev of next_node.
4. Set the prev pointer of the next_node to the new node.
5. Set the next pointer of this new node to the next_node.
6. Change the next of the new node's previous node.

```
def insert_before(self, next_node, new_data):
        if next_node is None:
            print("The given next node cannot be NULL")
            return
        new_node = Node(new_data)
        new_node.prev = next_node.prev
        next_node.prev = new_node
        new_node.next = next_node
        if new_node.prev is not None:
            new_node.prev.next = new_node
        else:
            self.head = new_node
```

**Time Complexity: O(1)**

**Auxiliary Space: O(1)**

## Add a node at the end in a Doubly Linked List

The new node is always added after the last node of the given Linked List. This can be done using the following steps:

1. Create a new node.
2. Put the value in the new node.
3. Make the next pointer of the new node as None.
4. If the list is empty, make the new node as the head.
5. Otherwise, travel to the end of the linked list.
6. Now make the next pointer of the last node point to the new node.

Change the prev pointer of the new node to the last node of the list.

```
def append(self, new_data):
        new_node = Node(new_data)
        new_node.next = None
        if self.head is None:
            new_node.prev = None
            self.head = new_node
            return
        last = self.head
while last.next is not None:
            last = last.next
        last.next = new_node
        new_node.prev = last
```

**Time Complexity: O(n)**

**Auxiliary Space: O(1)**

**Delete a node in a Doubly Linked List**

The deletion of a node in a doubly linked list can be divided into three main categories:

1. Deletion of the head node.
2. Deletion of a middle node.
3. Deletion of the last node.

All three mentioned cases can be handled if the pointer to the node to be deleted and the head pointer are known.

1. If the node to be deleted is the head node, make the next node as head.
2. If a node is deleted, connect the next and previous node of the deleted node.

```
def delete_node(self, del_node):
        if self.head is None or del_node is None:
            return
        if self.head == del_node:
            self.head = del_node.next
        if del_node.next is not None:
            del_node.next.prev = del_node.prev
        if del_node.prev is not None:
            del_node.prev.next = del_node.next

# Driver Code
if __name__ == "__main__":
    dll = DoublyLinkedList()
    dll.push(2)
dll.push(4)
    dll.push(8)
    dll.push(10)
    print("Original Linked list:")
    dll.print_list(dll.head)
    dll.delete_node(dll.head)  # Delete first node
    dll.delete_node(dll.head.next)  # Delete middle node
    dll.delete_node(dll.head.next)  # Delete last node
    print("\nModified Linked list:")
    dll.print_list(dll.head)
```

**Output:**

**Original Linked list:**
10 8 4 2

**Modified Linked list:**
8

**Complexity Analysis:**

**Time Complexity:** O(1).
Since traversal of the linked list is not required, the time complexity is constant.

**Auxiliary Space:** O(1).
As no extra space is required, the space complexity is constant.

## Q1. Split linked list in parts [Leetcode 725]

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

```
class Solution:
    def splitListToParts(self, head: ListNode, k: int):
        cur = head
        N = 0
        while cur:
            cur = cur.next
            N += 1

        width, rem = divmod(N, k)
        ans = [None] * k
        cur = head
        for i in range(k):
            head1 = ListNode(0)
            write = head1
            for j in range(width + (i < rem)):
                write.next = ListNode(cur.val)
                write = write.next
                if cur:
                    cur = cur.next
 ans[i] = head1.next
        return ans
```

**Q2. Nodes Between Critical Points** **(Leetcode 2058)**

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def nodesBetweenCriticalPoints(self, head: ListNode):
        ans = [-1, -1]
        if not head or not head.next or not head.next.next:
            return ans

arr = []
        t = head.next
        prev = head
        idx = 1

        while t.next:
            if t.val > prev.val and t.val > t.next.val:
                arr.append(idx)
            if t.val < prev.val and t.val < t.next.val:
                arr.append(idx)
            idx += 1
            prev = t
            t = t.next
```

```
 if len(arr) < 2:
            return ans

        ans[1] = arr[-1] - arr[0]
        min_diff = float('inf')

        for i in range(1, len(arr)):
            min_diff = min(arr[i] - arr[i - 1], min_diff)
        ans[0] = min_diff
        return ans
```

### Q3.Reverse Even Length Groups (Leetcode 2074)

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverse(self, arr, i, j):
        while i < j:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
            j -= 1

    def reverseEvenLengthGroups(self, head: ListNode):
        if not head.next:
            return head

        arr = []
        t = head

        while t:
            arr.append(t.val)
            t = t.next

 k = 1
        n = len(arr)
        i = 0

        while i < n:
            j = min(n - 1, i + k - 1)
            if (j - i + 1) % 2 == 0:
                self.reverse(arr, i, j)
            i = j + 1
            k += 1

        t = head

        for num in arr:
            t.val = num
            t = t.next
        return head
```

## Q4. Leetcode 138 - Copy List with Random Pointer

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverse(self, arr, i, j):
        while i < j:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
            j -= 1

    def reverseEvenLengthGroups(self, head: ListNode):
        if not head.next:
            return head

        arr = []
        t = head

        while t:
            arr.append(t.val)
            t = t.next

 k = 1
        n = len(arr)
        i = 0

        while i < n:
            j = min(n - 1, i + k - 1)
            if (j - i + 1) % 2 == 0:
                self.reverse(arr, i, j)
            i = j + 1
            k += 1

        t = head

        for num in arr:
            t.val = num
            t = t.next
        return head
```

## Q5. Leetcode 430 - Flatten a Multilevel Doubly Linked List

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

class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        if not head:
            return None

        current = head
        stack = []

        while current:
            if current.child:
                if current.next:
                    stack.append(current.next)
                current.next = current.child
                current.child.prev = current
                current.child = None
if not current.next and stack:
                next_node = stack.pop()
                current.next = next_node
                next_node.prev = current

            current = current.next

        return head
```