A Doubly Linked List is a type of linked list where the nodes not only reference the next node but also the previous one. This implementation allows us to traverse a list in both directions, so operations such as appending and deleting can be much easier and faster to perform than a singly linked list.

<-- Previous | [Data] | Next -->

Its implementation is similar to the singly linked list, with only small changes to the parameters. Using the singly linked list created on the previous lecture, we can adapt its code to implement a doubly linked list.

In [14]:

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


class DoublyLinkedList():
    def __init__(self):
        self.head = None
        self.tail = self.head
        self.length = 0


    def print_list(self):
        if self.head == None:
            print("Empty")
        else:
            current_node = self.head
            while current_node!= None:
                print(current_node.data, end= ' ')
                current_node = current_node.next
        print()


    def append(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
```

```python
            self.tail = self.head
            self.length += 1
            return
        else:
            new_node.previous = self.tail
            self.tail.next = new_node
            self.tail = new_node
            self.length += 1
            return
# If linked list is empty, we make head and tail both equal to the new node.
# Otherwise, we make the previous pointer of the new node point to the current tail.
# Then, the next pointer of the current tail will refer to the new node.
# This will build a two-way link between the current tail and the new node.
# Finally, the tail is updated so that it is equal to the new node.


    def prepend(self, data):
        new_node = Node(data)
        if self.head == None:
            self.head = new_node
            self.tail = self.head
            self.length += 1
            return
        else:
            new_node.next = self.head
            self.head.previous = new_node
            self.head = new_node
            self.length += 1
            return
# The node next to the new node will point to the current head.
# We connect the node behind the current head to the new node, and then update the head.


    def insert(self, position, data):
        if position == 0:
            self.prepend(data)
            return
        if position >= self.length:
            if position > self.length:
                print('Position unavailable. Appending at the end of the list.')
            self.append(data)
```

```python
            return
        else:
            new_node = Node(data)
            current_node = self.head
            for i in range(position - 1):
                current_node = current_node.next
            new_node.previous = current_node
            new_node.next = current_node.next
            current_node.next = new_node
            new_node.next.previous = new_node
            self.length += 1
            return
# Inserting at position 0 is equivalent to prepending.
# So instead of repeating code, we call the prepend method.
# Similarly, inserting data position >= the length of the list is equivalent to appending.
# So we call the append method instead.
# We traverse up to one position before the position we want to insert the new node in.
# We make the new node's previous pointer refer to the current node, and the next pointer to the next of the curre
# Then we break the link between the current node and the next node and make the next pointer of the current node
# Finally the next node's previous pointer will be update, so that it refers to the new node instead of the curren
# This way, the new node gets inserted in betwwen the current and the next nodes.

    def delete_by_value(self, data):
        if self.head == None:
            print("Nothing to delete. Linked List is empty.")
            return

        current_node = self.head
        if current_node.data == data:
            self.head = self.head.next
            if self.head == None or self.head.next==None:
                self.tail = self.head
            if self.head != None:
                self.head.previous = None
            self.length -= 1
            return
        try:
            while current_node!= None and current_node.next.data != data:
                current_node = current_node.next
            if current_node!=None:
                current_node.next = current_node.next.next
```

```python
                if current_node.next != None:
                    current_node.next.previous = current_node
                else:
                    self.tail = current_node
                self.length -= 1
                return
        except AttributeError:
            print("Given value not found.")
            return
# If upon deleting the first node the list becomes empty or has only one node, we set the tail equal to the head.
# The new head's previous pointer is set to be 'None'.
# We add a try block in case the value is not found. The current_node.next will be 'None', and there is no data pa
# If the node deleted is not the last node(i.e., the node 2 positions after the current node is != 'None'), the pr
# This way, a connection is established.
# If the deleted node is the last node, then we update the tail to be the current node.

    def delete_by_position(self, position):
        if self.head == None:
            print("Linked List is empty. Nothing to delete.")
            return

        if position == 0:
            self.head = self.head.next
            #print(self.head)
            if self.head == None or self.head.next == None:
                self.tail = self.head
            if self.head != None:
                self.head.previous = None
            self.length -= 1
            return

        if position>=self.length:
            position = self.length-1

        current_node = self.head
        for i in range(position - 1):
            current_node = current_node.next
        current_node.next = current_node.next.next
        if current_node.next != None:
            current_node.next.previous = current_node
        else:
```

```python
            self.tail = current_node
        self.length -= 1
        return

# We update the new head's previous pointer to be equal to 'None'
# Similar logic to the delete_by_value method
```

In [15]:

```python
# Now, it is time to test the doubly linked list blueprint by creating our own.

test_list = DoublyLinkedList()
test_list.print_list()
```

```
Empty
```

In [16]:

```python
test_list.append(5)
test_list.append(2)
test_list.append(9)
test_list.print_list()
```

```
5 2 9
```

In [17]:

```python
test_list.prepend(4)
test_list.print_list()
```

```
4 5 2 9
```

In [18]:

```python
test_list.insert(2,7)
test_list.print_list()
```

```
4 5 7 2 9
```

In [19]:

```python
test_list.insert(0,0)
test_list.insert(6,0)
test_list.insert(9,3)
test_list.print_list()
```

```
Position unavailable. Appending at the end of the list.
0 4 5 7 2 9 0 3
```

In [20]:
```
test_list.delete_by_value(3)
test_list.print_list()
```

```
0 4 5 7 2 9 0
```

In [21]:
```
test_list.delete_by_value(0)
test_list.print_list()
```

```
4 5 7 2 9 0
```

In [22]:
```
test_list.delete_by_position(3)
test_list.print_list()
```

```
4 5 7 9 0
```

In [23]:
```
test_list.delete_by_position(0)
test_list.print_list()
```

```
5 7 9 0
```

In [24]:
```
test_list.delete_by_position(8)
test_list.print_list()
```

```
5 7 9
```

In [25]:
```
test_list.delete_by_value(3)
test_list.print_list()
```

```
Given value not found.
5 7 9
```

In [13]:
```
print(test_list.length)
```

```
3
```