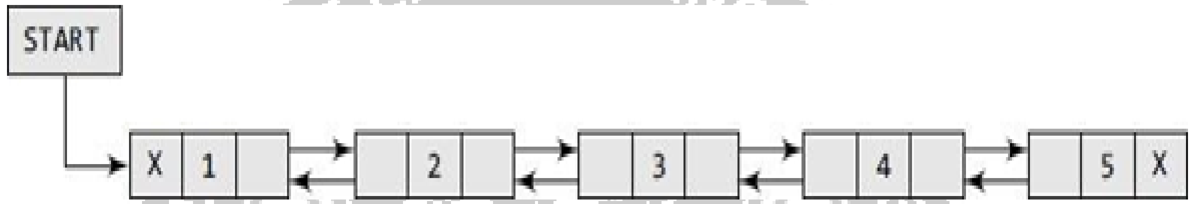
# DOUBLY LINKED LISTS

**Definition**

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Figure




The structure of a doubly linked list can be given as, struct node

```
{
struct node *prev;
int data;
struct node *next;
};
```

- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- Doubly linked list calls for more space per node and more expensive basic operations.
- However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a doubly linked list is that it makes searching twice as efficient.

**Memory representation of a doubly linked list**

START
1

| | DATA | PREV | NEXT |
|---|---|---|---|
| 1 | H | –1 | 3 |
| 2 | | | |
| 3 | E | 1 | 6 |
| 4 | | | |
| 5 | | | |
| 6 | L | 3 | 7 |
| 7 | L | 6 | 9 |
| 8 | | | |
| 9 | O | 7 | –1 |

- Variable START is used to store the address of the first node.
- In this example, START = 1, so the first data is stored at address 1, which is H.
- Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field.
- We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list.
- When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in the above example stores characters that when put together form the word HELLO.

**Inserting a New Node in a Doubly Linked List**

Case 1: The new node is inserted at the beginning.

Case 2: The new node is inserted at the end.

Case 3: The new node is inserted after a given node.

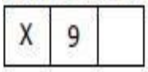
Case 4: The new node is inserted before a given node.

**Case 1: The new node is inserted at the beginning.**

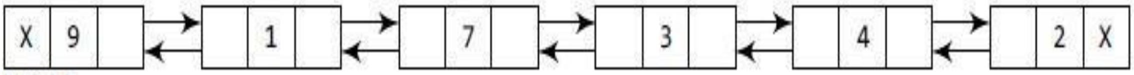
We want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.


START

Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.




Add the new node before the START node. Now the new node becomes the first node of the list.




START

**Algorithm to insert a new node at the beginning**

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

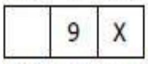
- In Step 1, we first check whether memory is available for the new node.
- If the free memory has exhausted, then an OVERFLOW message is printed.
- Otherwise, if free memory cell is available, then we allocate space for the new node.
- Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START.
- Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

**Case 2: The new node is inserted at the end.**

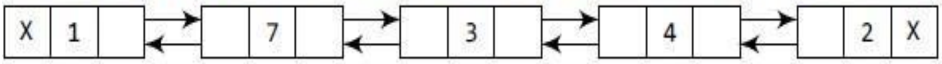
We want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

START

Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

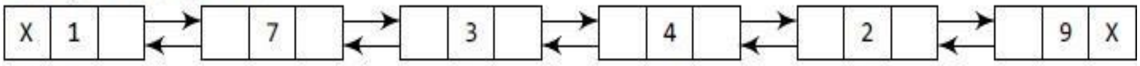


Take a pointer variable PTR and make it point to the first node of the list.



START,PTR

Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.



START PTR

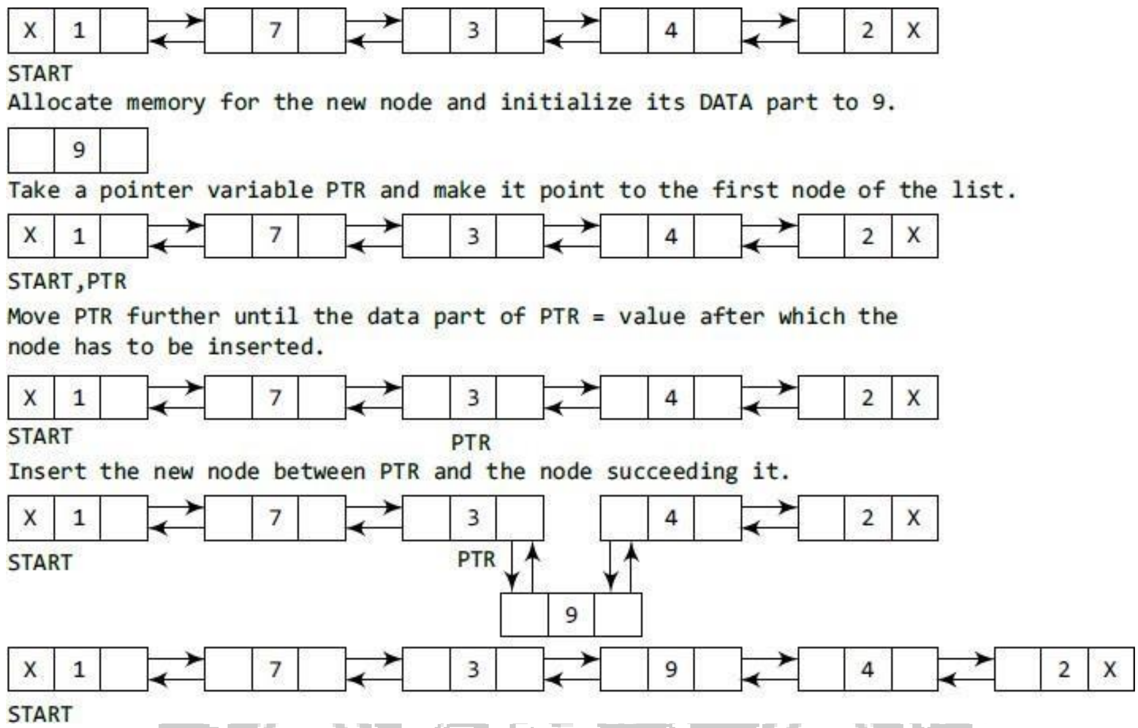
**Algorithm to insert a new node at the end**

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

- In Step 6, we take a pointer variable PTR and initialize it with START.
- In the while loop, we traverse through the linked list to reach the last node.
- Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node.
- Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list.
- The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

Case 3: The new node is inserted after a given node.

We want to add a new node with value 9 after the node containing 3.



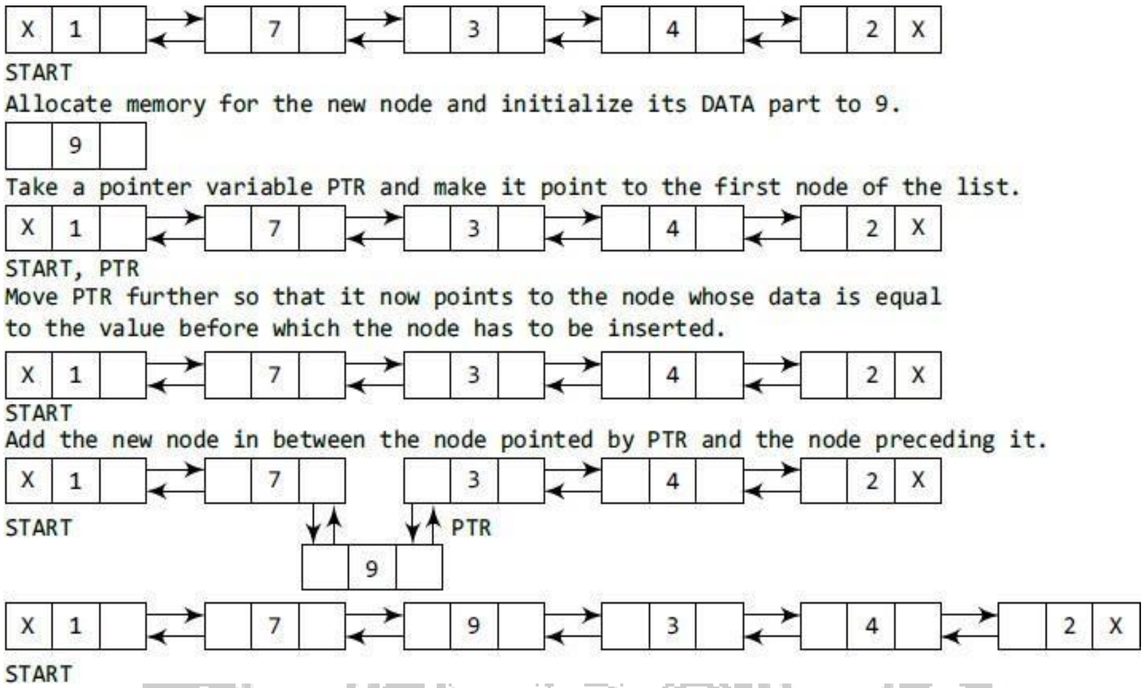

**Algorithm to insert a new node after a given node**

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

- In Step 5, we take a pointer PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.
- We need to reach this node because the new node will be inserted after this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

Case 4: The new node is inserted before a given node.

We want to add a new node with value 9 before the node containing 3.




**Algorithm to insert a new node before a given node**

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- In Step 5, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM.

- We need to reach this node because the new node will be inserted before this node.
- Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

**Deleting a Node from a Doubly Linked List**

In this section, we will see how a node is deleted from an already existing doubly linked list

Case 1: The first node is deleted.

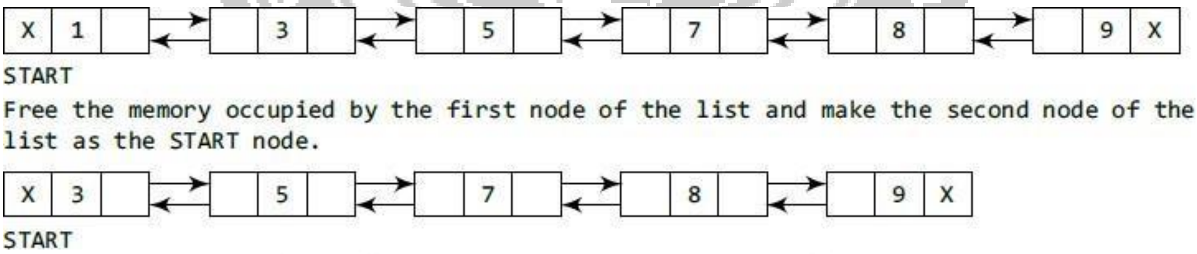
Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 4: The node before a given node is deleted.

**Case 1: The first node is deleted.**

When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list



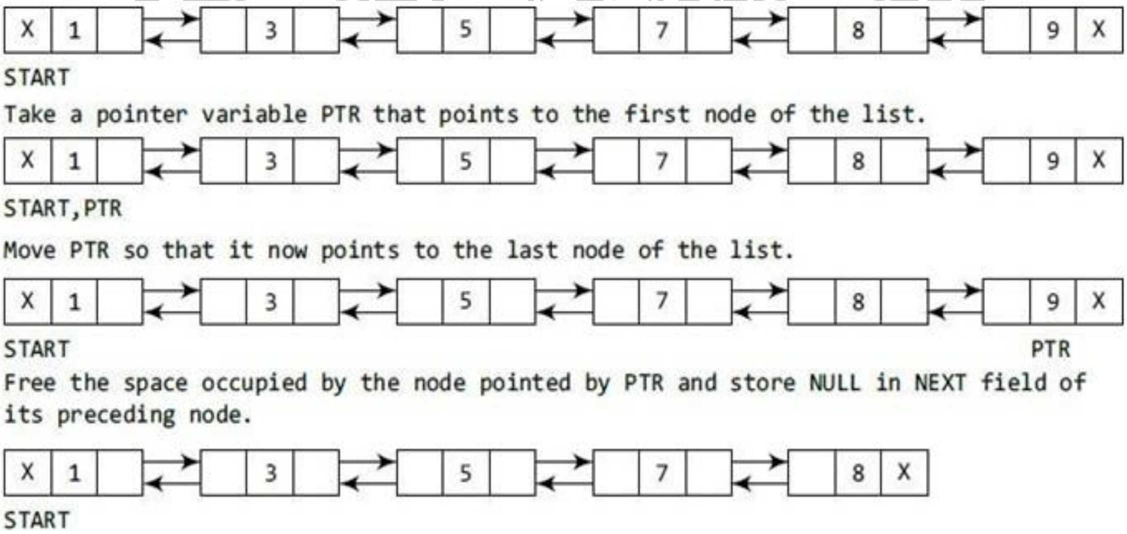

**Algorithm to delete the first node**

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 6
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

- In Step 1 of the algorithm, we check if the linked list exists or not.
- If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list.
- In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

**Case 2: The last node is deleted.**

We want to delete the last node from the linked list, then the following changes will be done in the linked list.



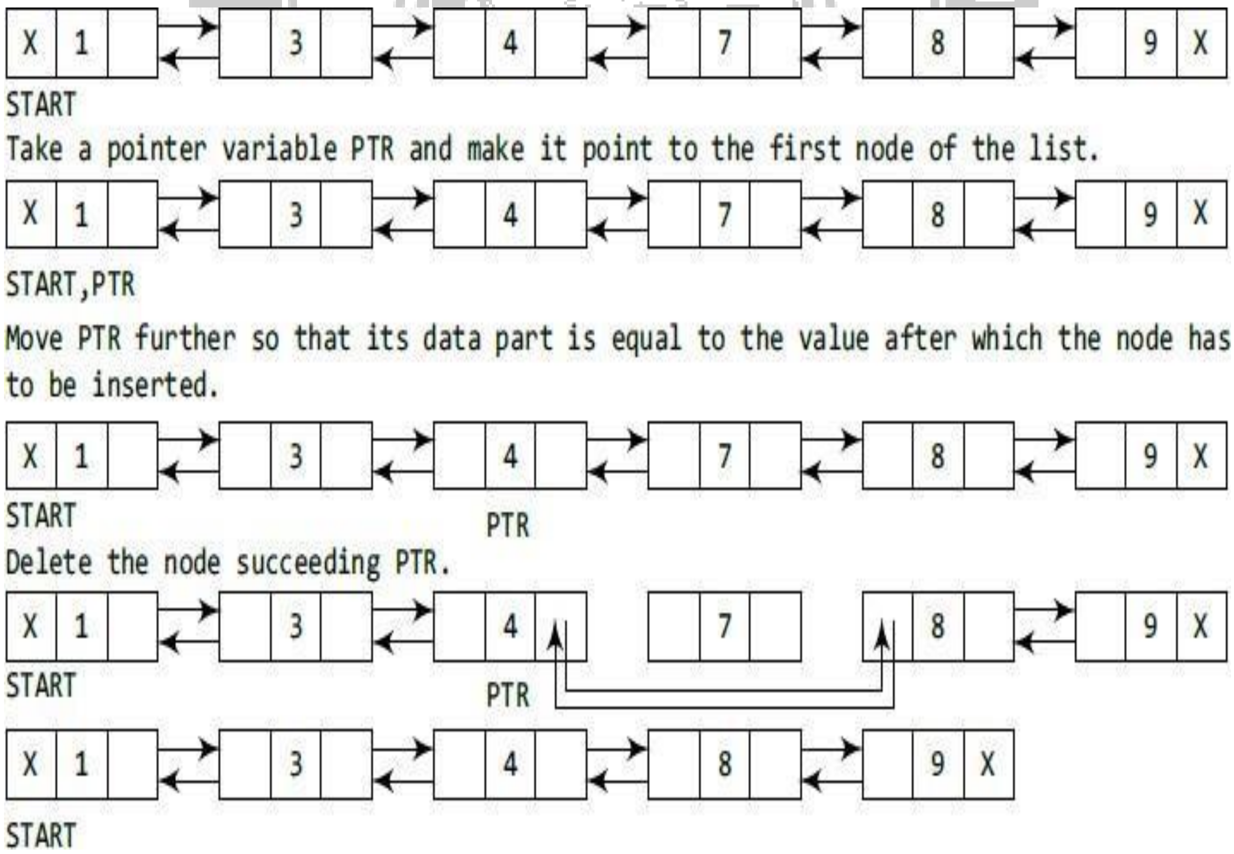

**Algorithm to delete the last node**

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 7
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != NULL
Step 4:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node.
- Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node.
- To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

**Case 3: The node after a given node is deleted.**

We want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



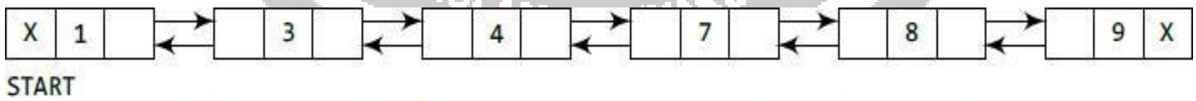
## Algorithm to delete a node after a given node

```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

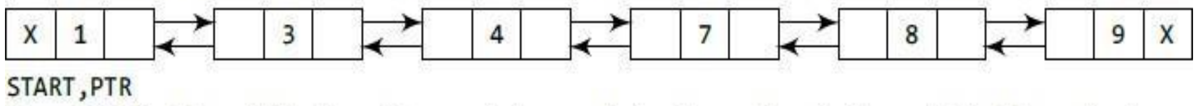
In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

**Case 4: The node before a given node is deleted.**

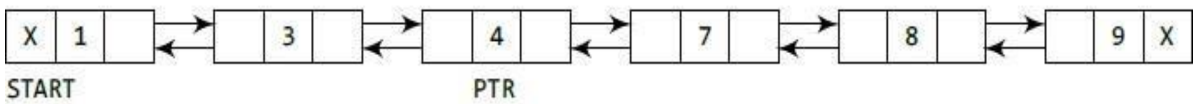
Suppose we want to delete the node preceding the node with value 4



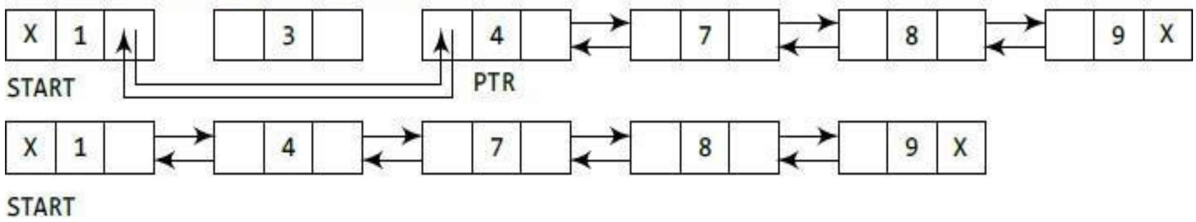

Take a pointer variable PTR that points to the first node of the list.




Move PTR further till its data part is equal to the value before which the node has to be deleted.




Delete the node preceding PTR.



**Algorithm to delete a node before a given node**

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

- In Step 2, we take a pointer variable PTR and initialize it with START.
- That is, PTR now points to the first node of the linked list.
- The while loop traverses through the linked list to reach the desired node.
- Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR.
- The memory of the node preceding PTR is freed and returned to the free pool.