Parul® University

Unit 3
Linked List

Prof. Keval Mehta,

Sr. Cyber Security Trainer

Computer Science & Engineering

TOPIC – 1

Definition

What is linked list

- A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

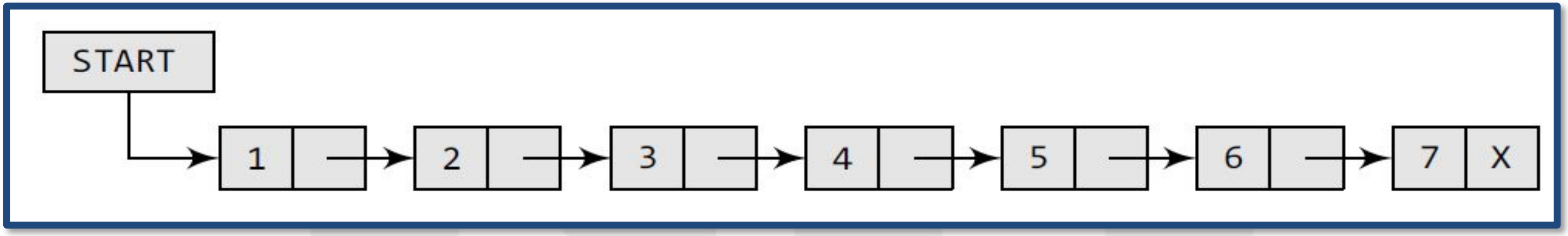
- However, unlike an array, a linked list does not allow random access of data.
- Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

- A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes.

TOPIC – 2

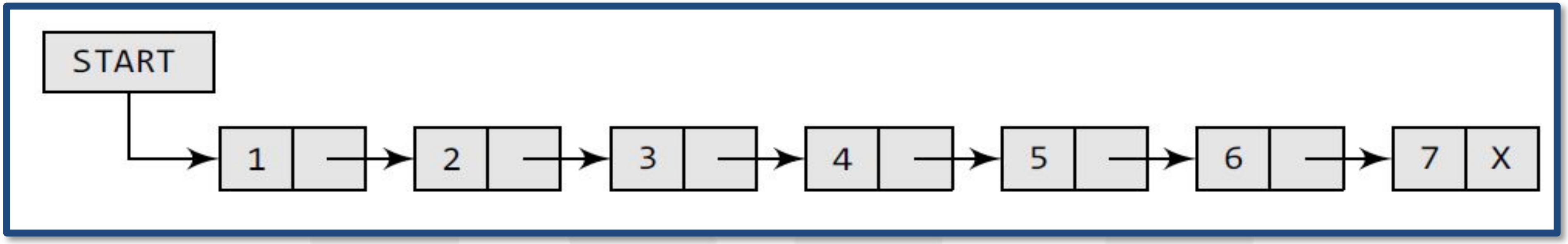
Representation of linked lists in Memory

Structure of Link List



1. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.
2. In Fig. 6.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.
3. The left part of the node which contains data may include a simple data type, an array, or a structure.

Structure of Link List



4. The right part of the node contains a pointer to the next node (or address of the next node in sequence).

5.The last node will have no next node connected to it, so it will store a special value called NULL.

6. In Fig. 6.1, the NULL pointer is represented by X.

Linked List in C Program

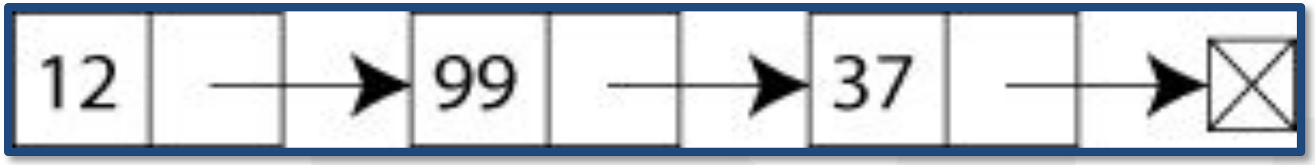
In C, we can implement a linked list using the following code:

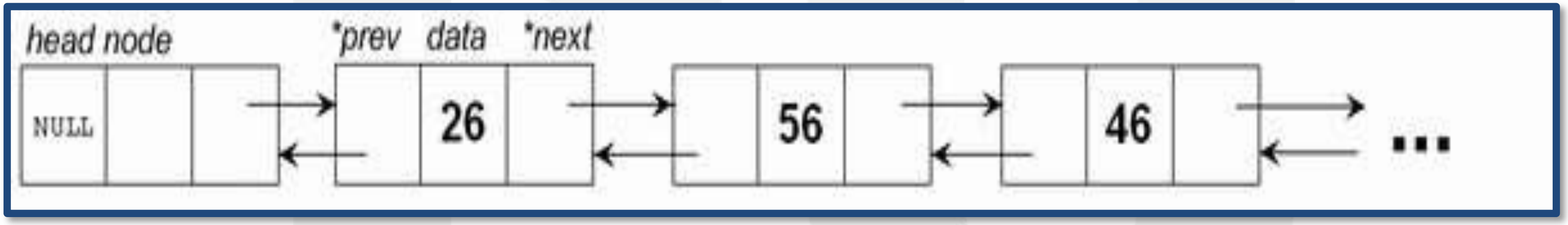
```
struct node
{
int data;
struct node *next;
};
```

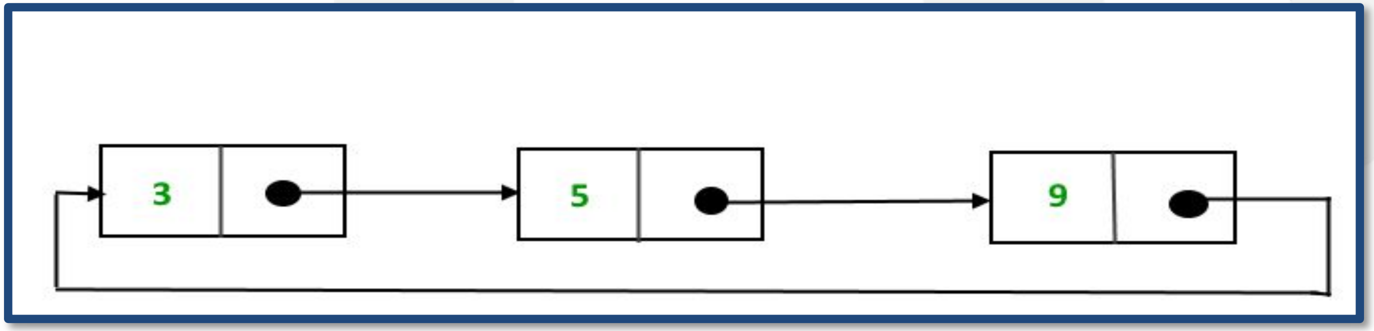
Types of Link List

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Types of Link List







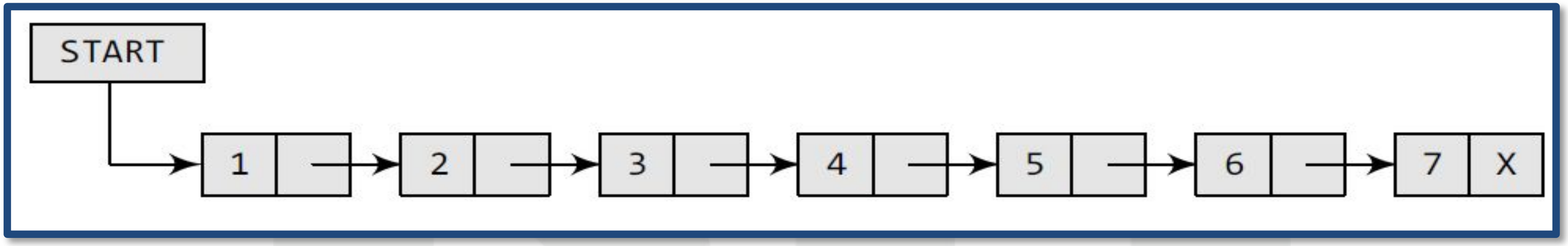
TOPIC – 3

Linked list operations

Operations in Link list

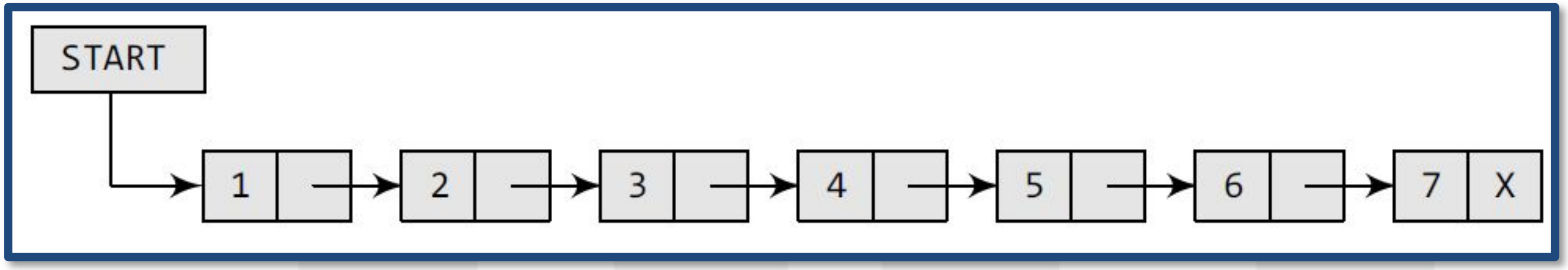
- Traversing list : accessing the nodes of the list in order to perform some
  processing on them.
- Insert element in link list
- Delete element from link list
- Search for element in link list

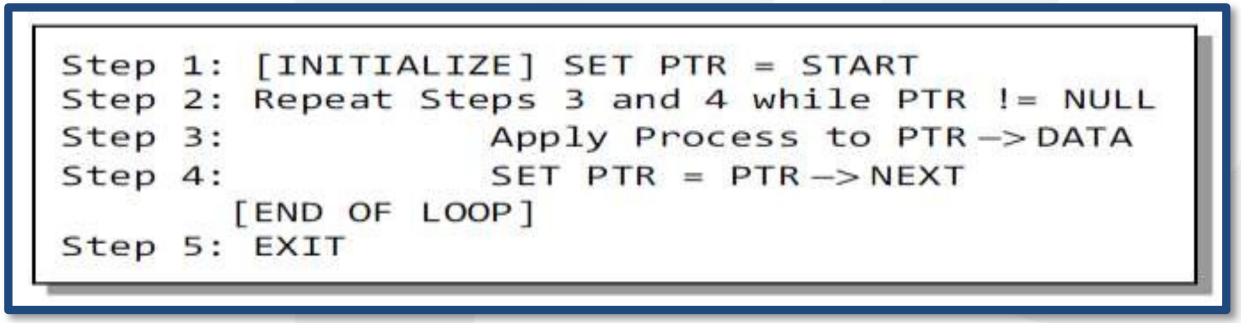
Traversing Single Link List



- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.
- A linked list always contains a pointer variable START which stores the address of the first node of the list.
- We also make use of another pointer variable PTR which points to the node that is currently being accessed.

Traversing Single Link List



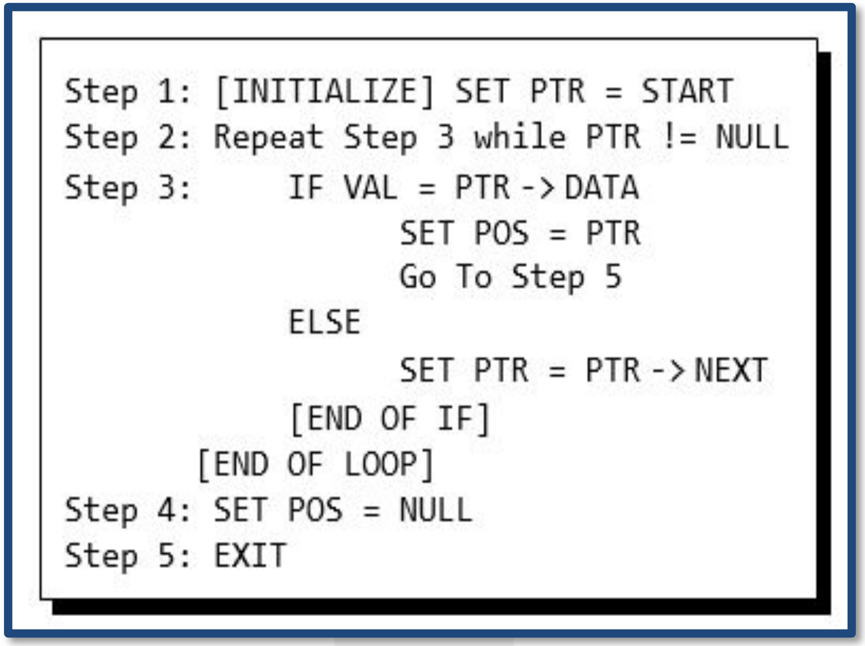


```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:             Apply Process to PTR -> DATA
Step 4:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: EXIT
```

Traversing Single Link List

- In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list.
- Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL.
- In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR.
- In Step 4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

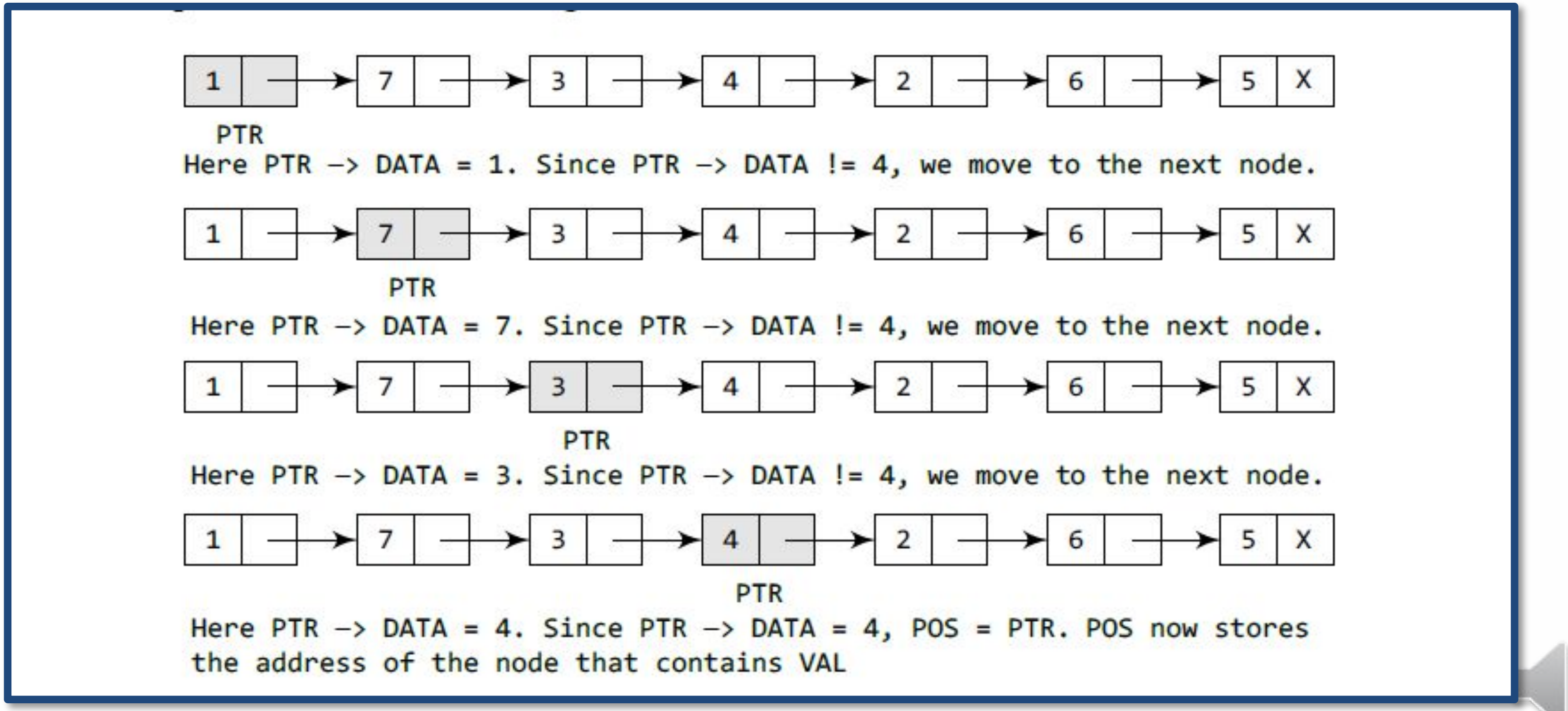
Searching : Single Link List



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:       IF VAL = PTR -> DATA
                   SET POS = PTR
                   Go To Step 5
              ELSE
                   SET PTR = PTR -> NEXT
              [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

- In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node.
- In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made. If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm.
- However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Searching : Single Link List



Here PTR -> DATA = 1. Since PTR -> DATA != 4, we move to the next node.

Here PTR -> DATA = 7. Since PTR -> DATA != 4, we move to the next node.

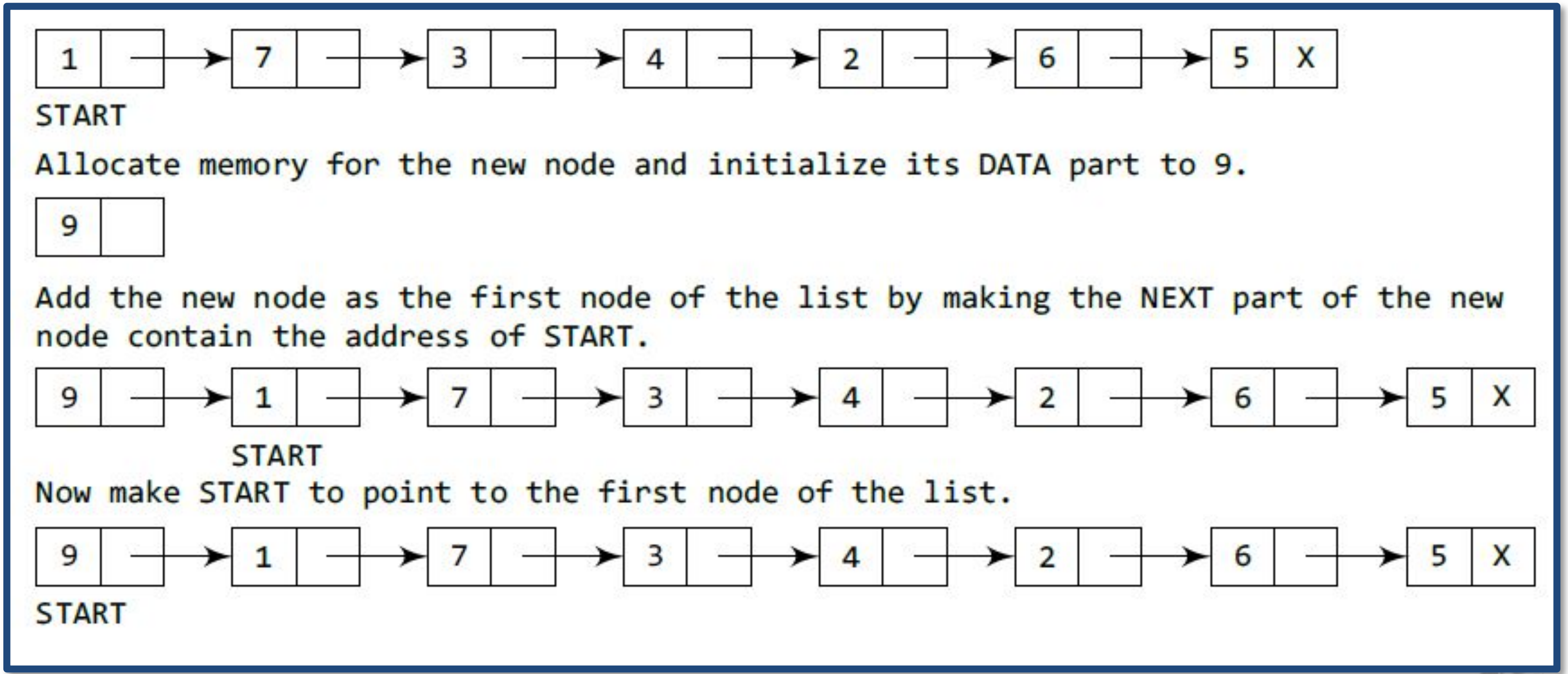
Here PTR -> DATA = 3. Since PTR -> DATA != 4, we move to the next node.

Here PTR -> DATA = 4. Since PTR -> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

Insertion : Single Link 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 1 :The new node is inserted at the beginning.

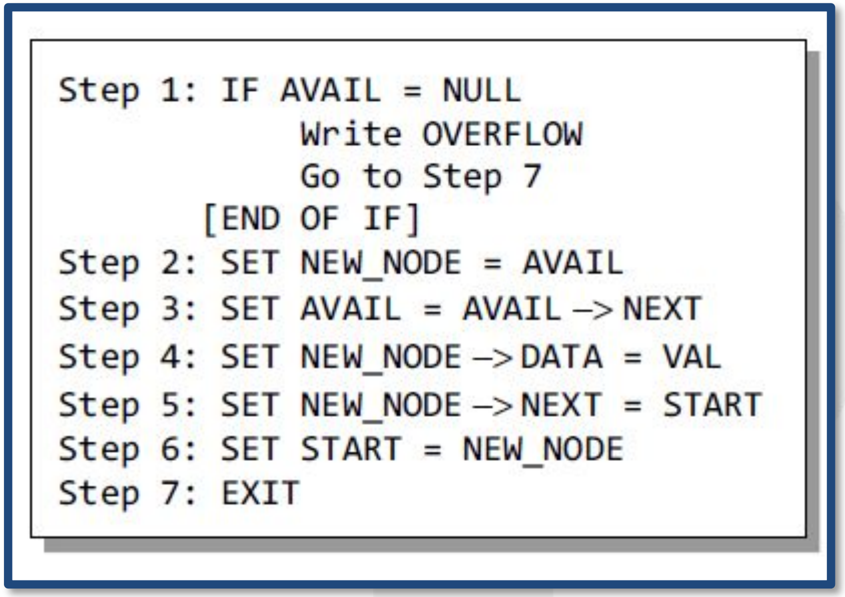


Allocate memory for the new node and initialize its DATA part to 9.

Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.

Now make START to point to the first node of the list.

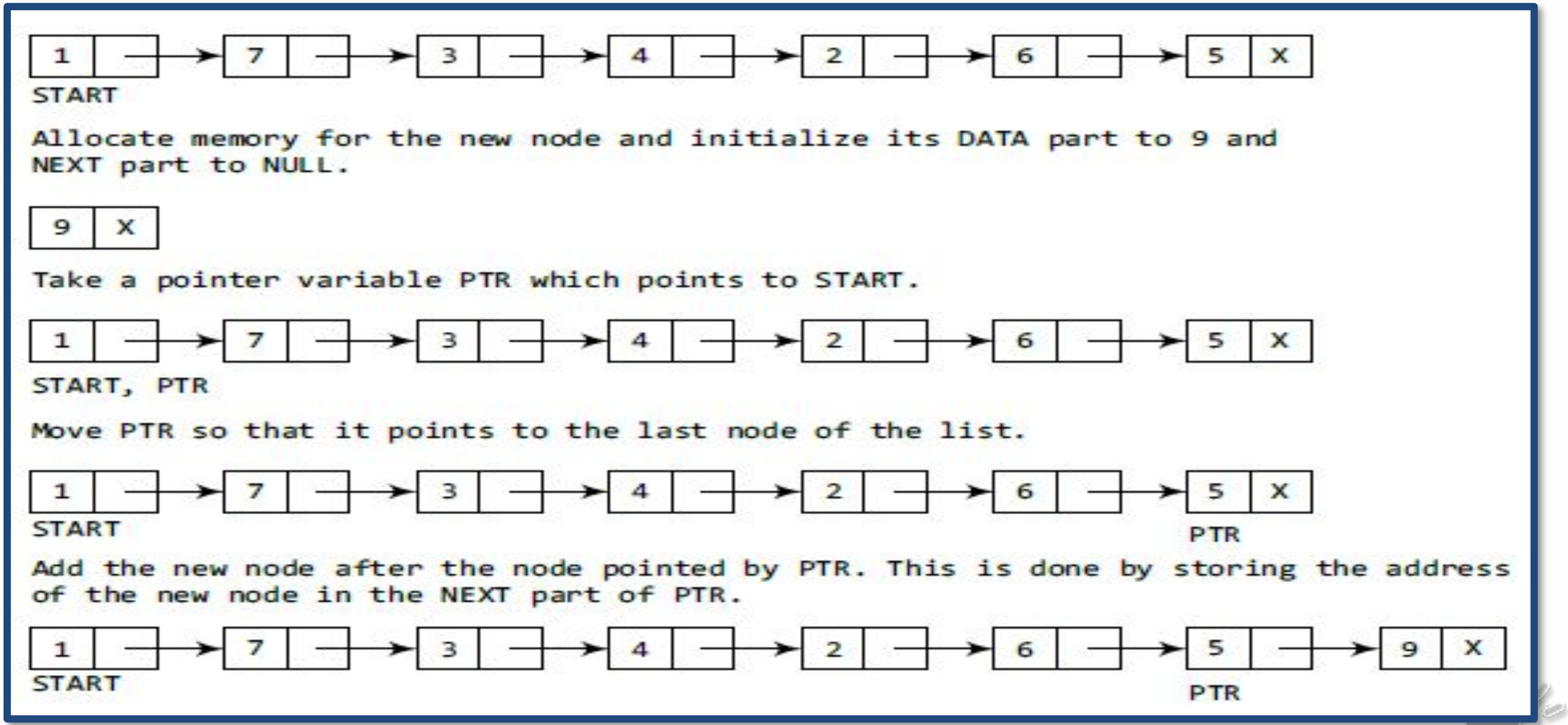
Case 1 :The new node is inserted at the beginning.



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 7
       [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 = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

- In Step 1, we first check whether memory is available for the new node.
- If a 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 the NEW_NODE.

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



1 | → 7 | → 3 | → 4 | → 2 | → 6 | → 5 | X

START

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

9 | X

Take a pointer variable PTR which points to START.

1 | → 7 | → 3 | → 4 | → 2 | → 6 | → 5 | X

START, PTR

Move PTR so that it points to the last node of the list.

1 | → 7 | → 3 | → 4 | → 2 | → 6 | → 5 | X

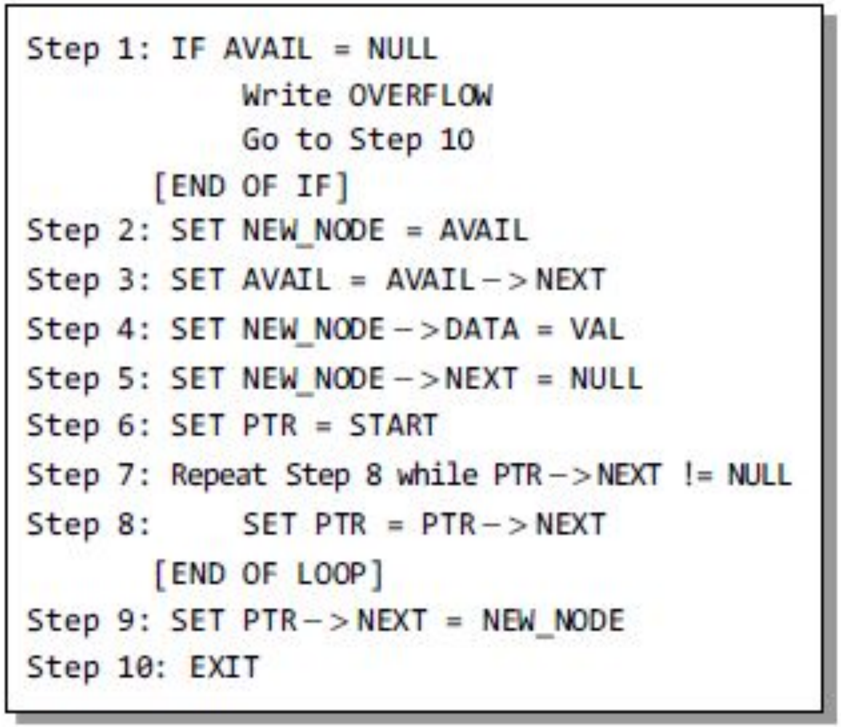
START PTR

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

1 | → 7 | → 3 | → 4 | → 2 | → 6 | → 5 | → 9 | X

START PTR

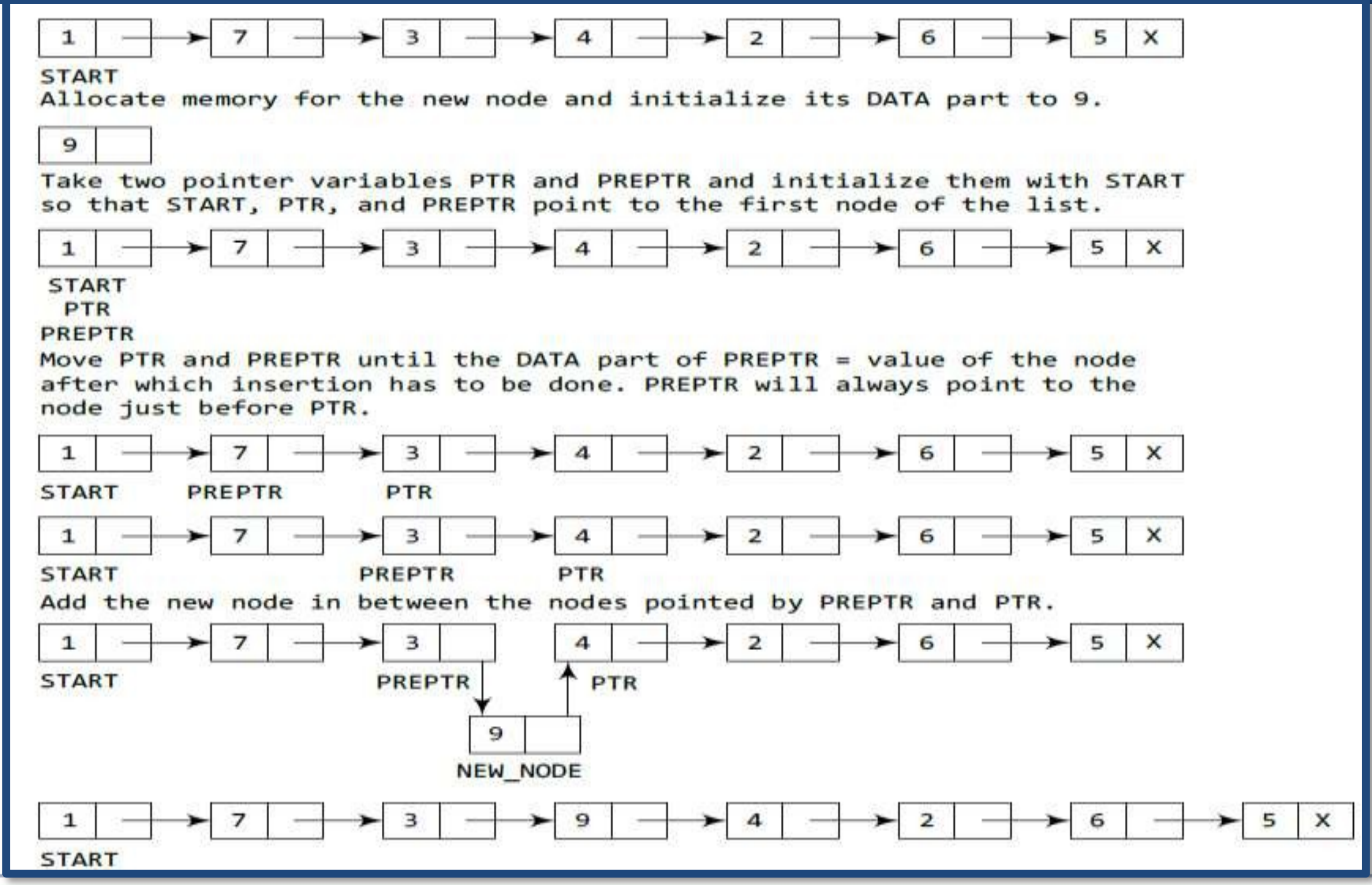
Case 2 :The new node is inserted at the end.



```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
       [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: EXIT
```

- In Step 6, 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 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.

Case 3 :The new node is inserted in between



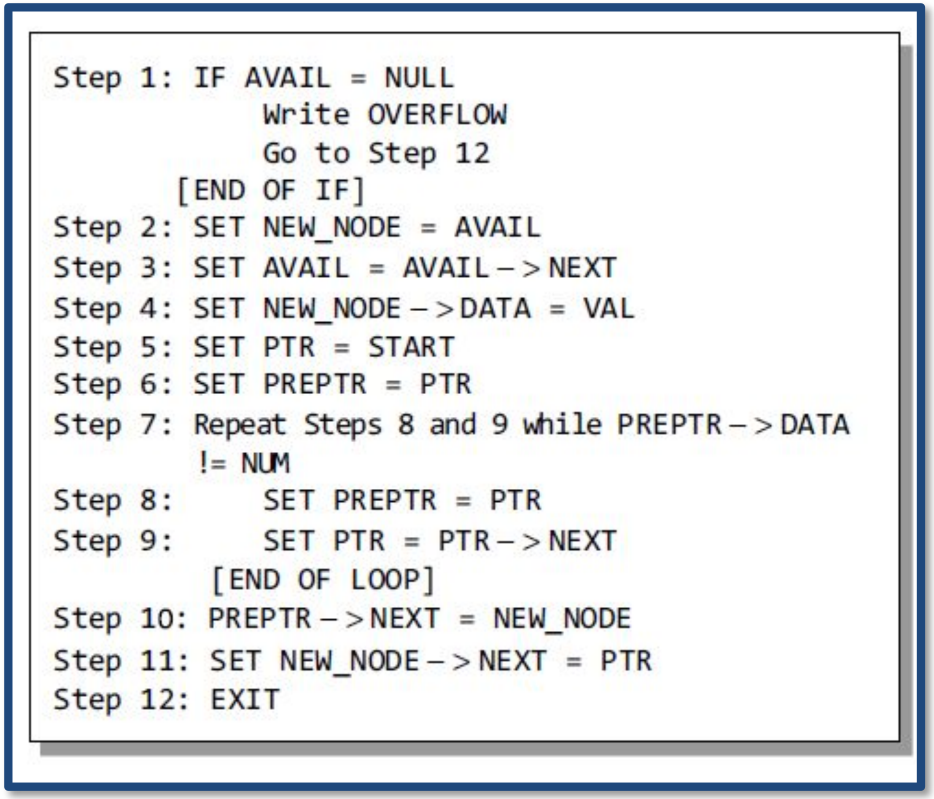
Allocate memory for the new node and initialize its DATA part to 9.

Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.

Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.

Add the new node in between the nodes pointed by PREPTR and PTR.

Case 3 :The new node is inserted in between



```
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: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR->DATA
         != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR->NEXT
         [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

- 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.
- Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR.
- Initially, PREPTR is initialized to PTR.
- So now, PTR, PREPTR, and START are all pointing 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, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

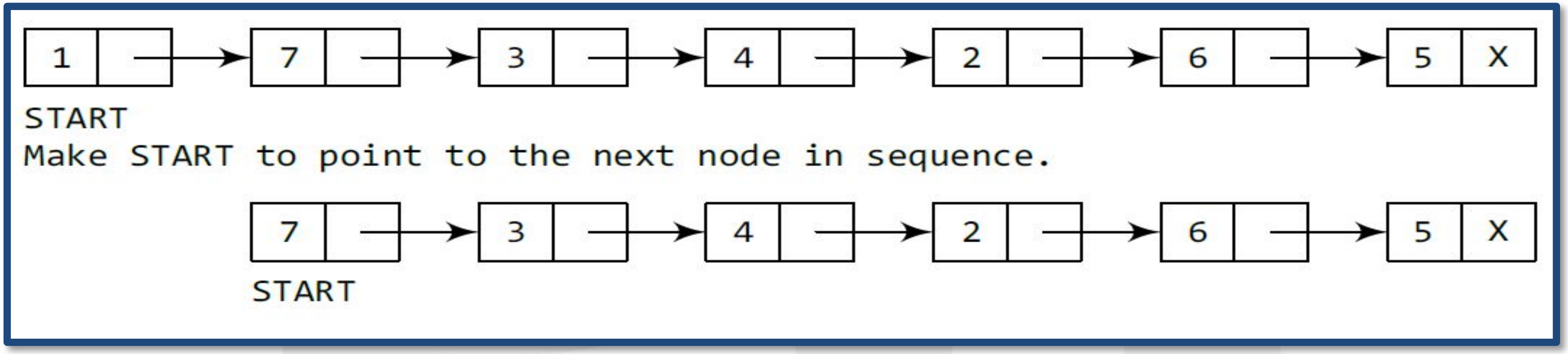
Single Link List : Deletion

Case 1: Delete first node

Case 2: Delete last node

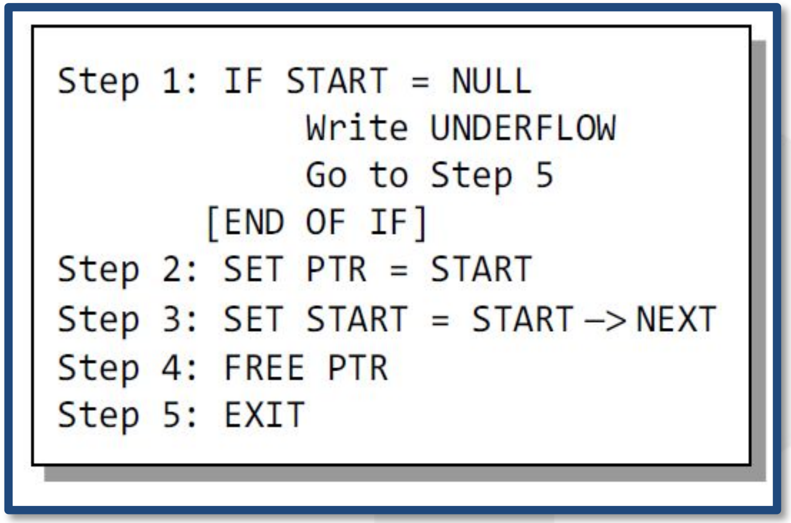
Case 3: Delete from between

Case 1 : Delete first node



Make START to point to the next node in sequence.

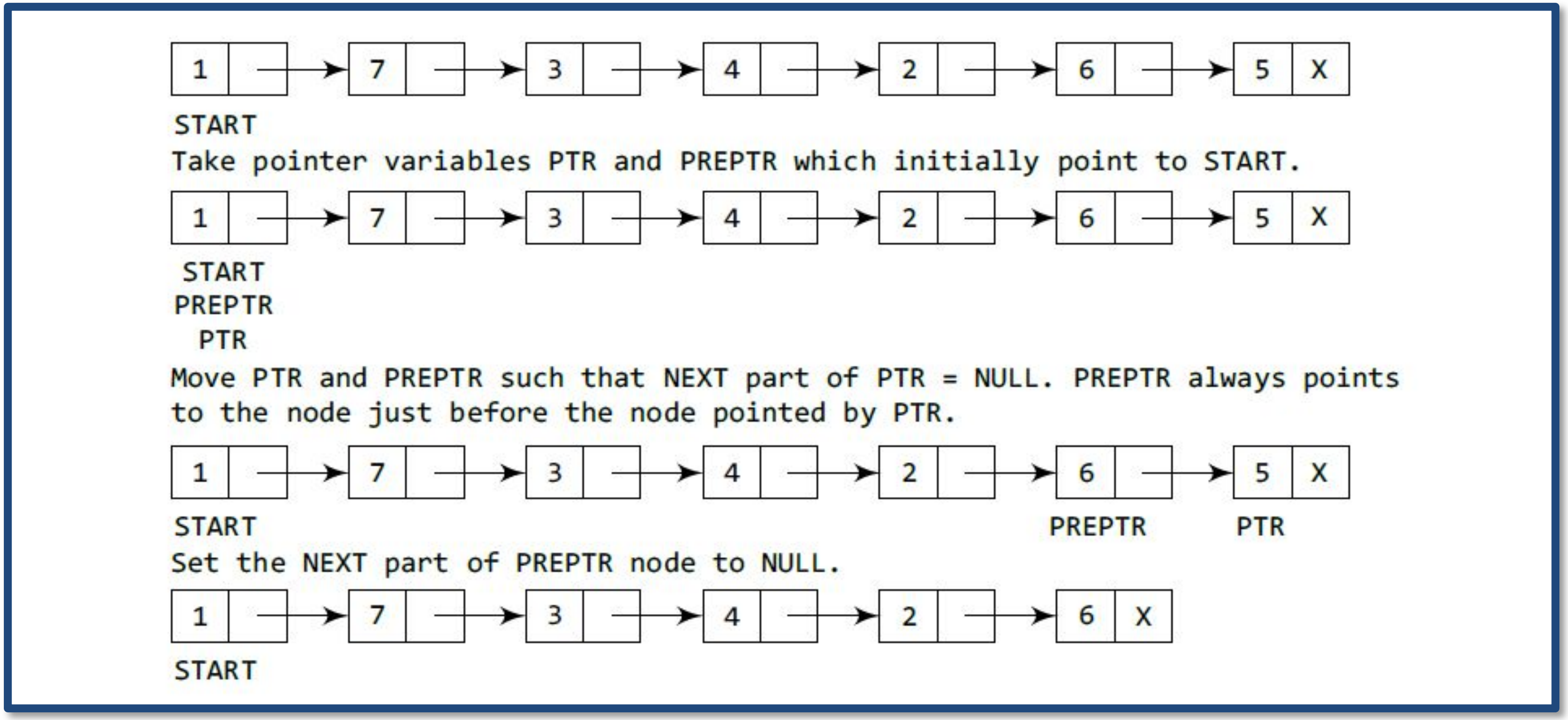
Case 1 : Delete first node



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

- In Step 1, 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 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 the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

Case 2 : Deleting Last Node



1 → 7 → 3 → 4 → 2 → 6 → 5 X
START

Take pointer variables PTR and PREPTR which initially point to START.

1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR

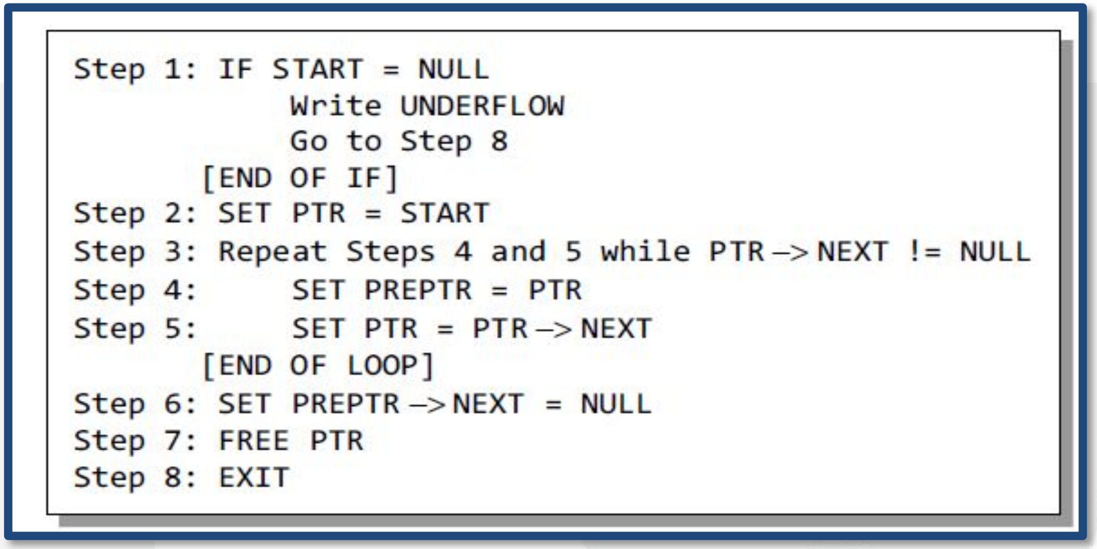
Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR

Set the NEXT part of PREPTR node to NULL.

1 → 7 → 3 → 4 → 2 → 6 X
START

Case 2 : Deleting Last Node

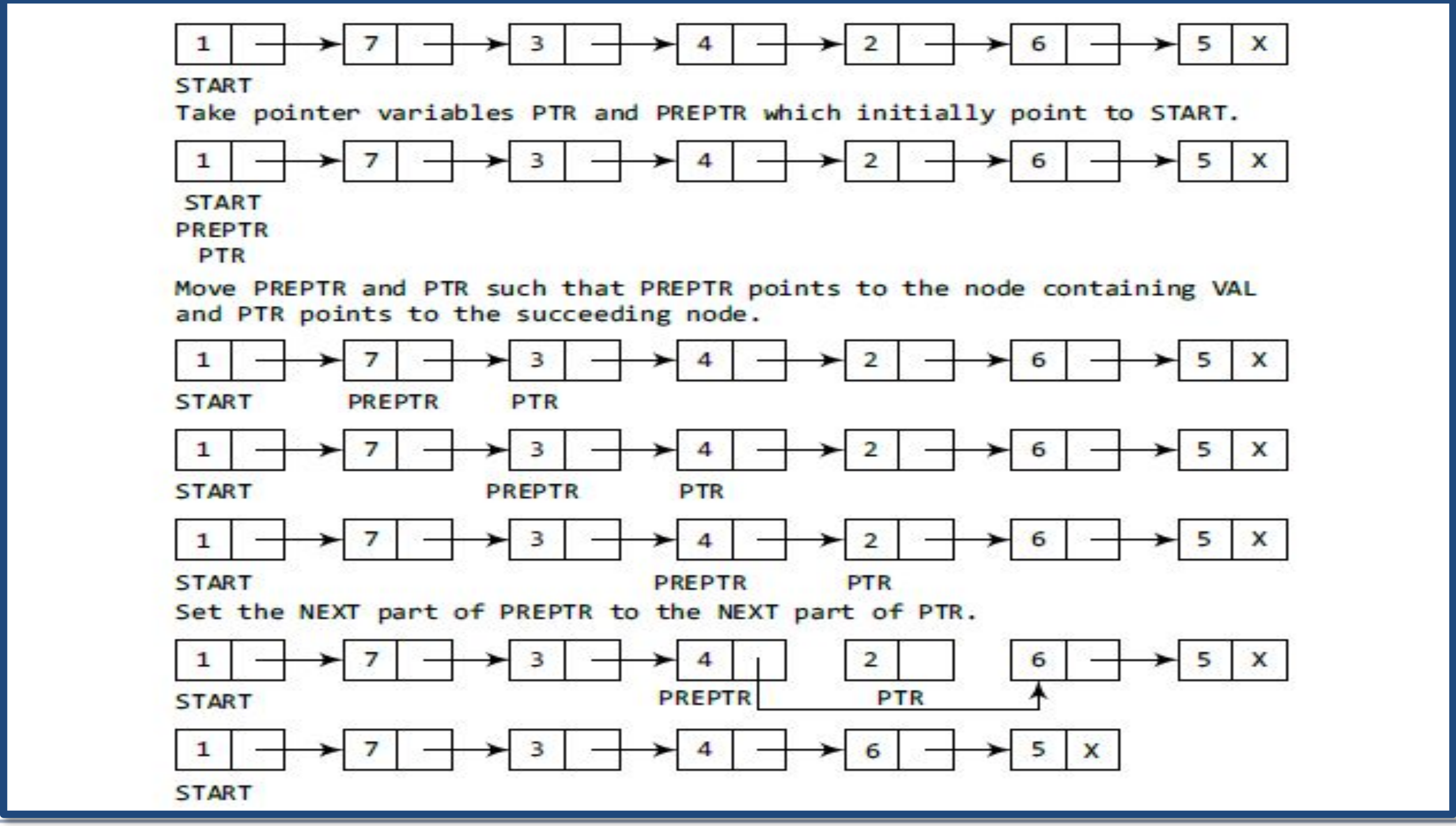


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:       SET PREPTR = PTR
Step 5:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT =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

Case 2 : Deleting Last Node

- Figure shows the algorithm to delete the last node from a linked list.
- 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.
- In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the last node and the second last node, we set the NEXT pointer of the 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 back to the free pool.

Case 3 :Deleting Node after a given position

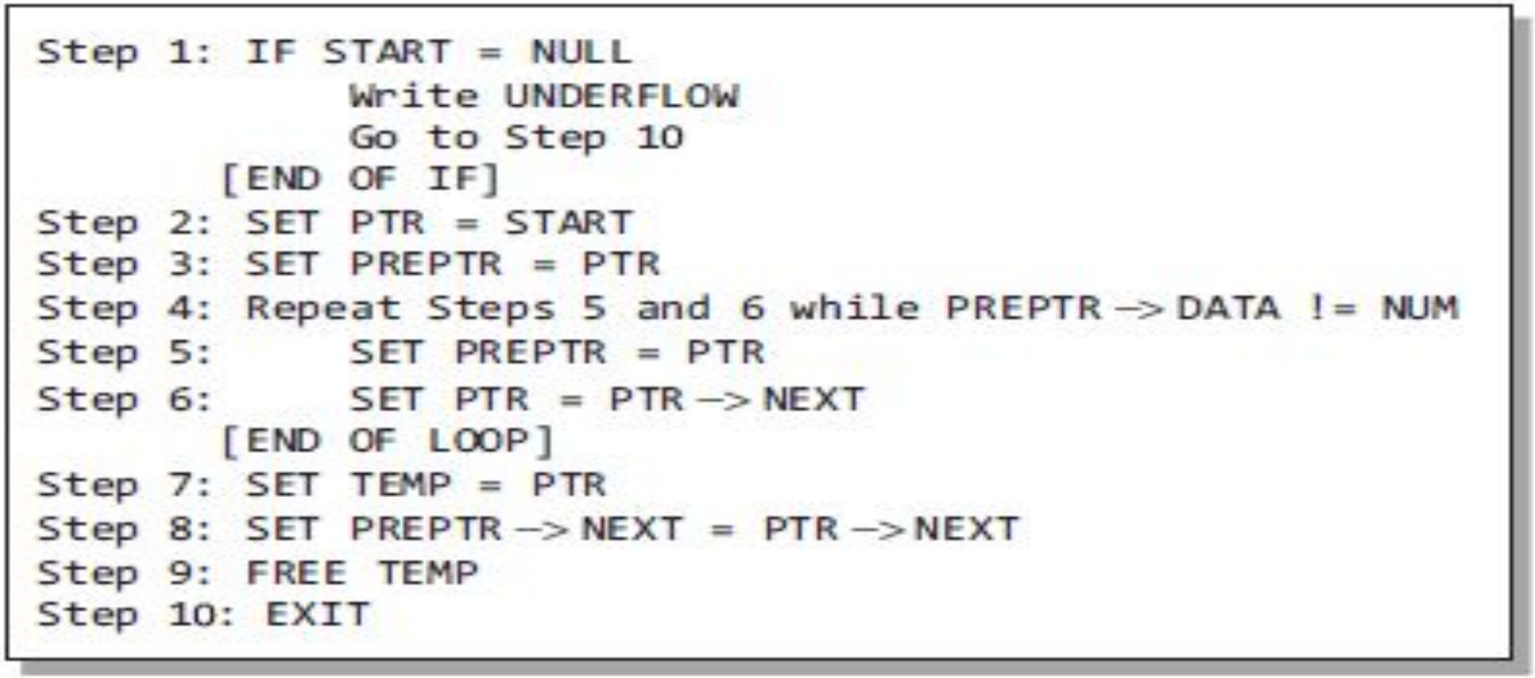


Take pointer variables PTR and PREPTR which initially point to START.

Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.

Set the NEXT part of PREPTR to the NEXT part of PTR.

Case 3 :Deleting Node after a given position

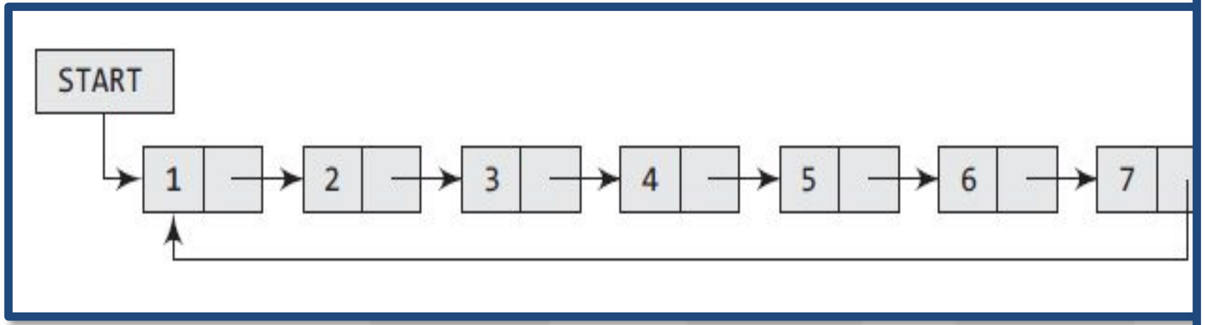


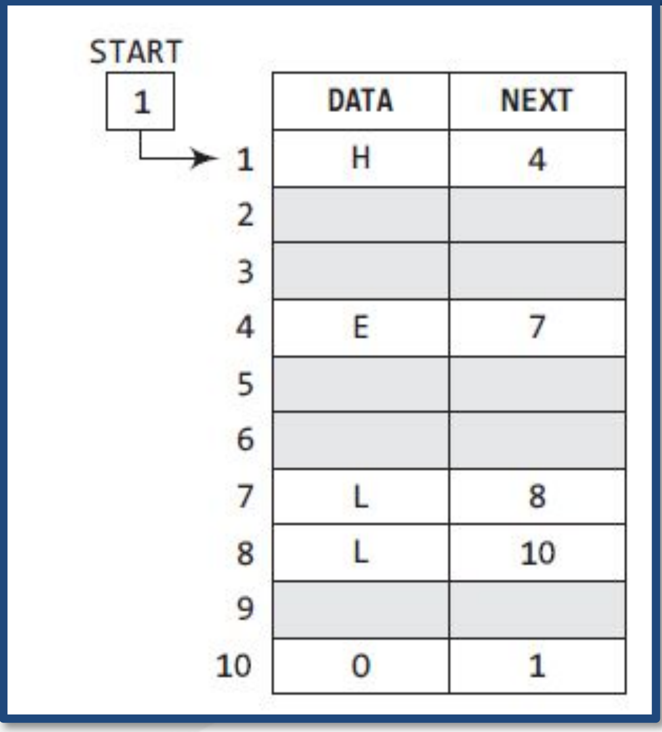
```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 10
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5:         SET PREPTR = PTR
Step 6:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Case 3 :Deleting Node after a given position

- Figure shows the algorithm to delete the node after a given node from a linked list.
- 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. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR.
- Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.

Circular Linked List




START

1

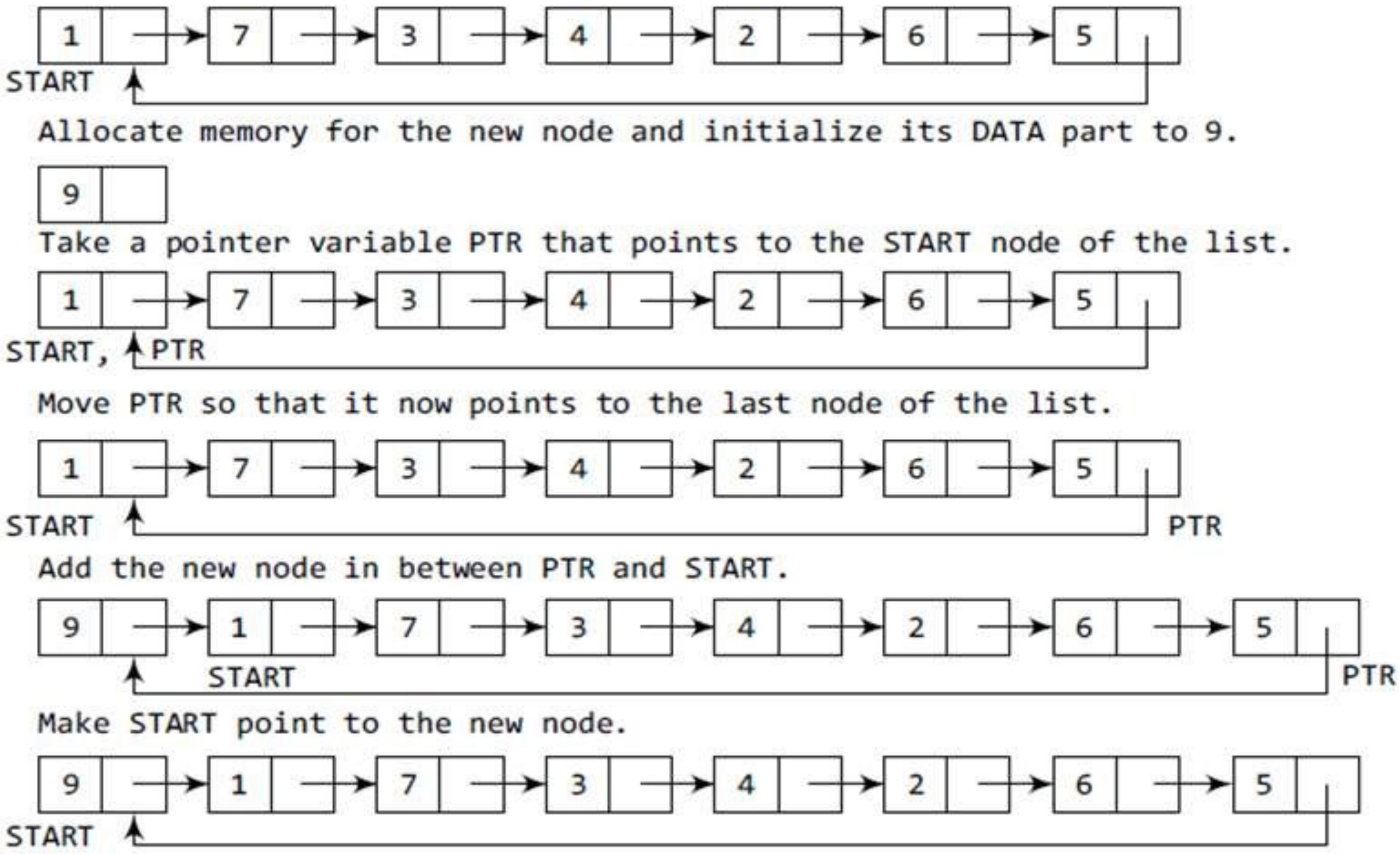
	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

In a circular linked list, the last node contains a pointer to the first node of the list.

Circular Link List-Insertion

- Case 1: The new node is inserted at the beginning of the circular linked list.
- Case 2: The new node is inserted at the end of the circular linked list.

Case 1: The new node is inserted at the beginning



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

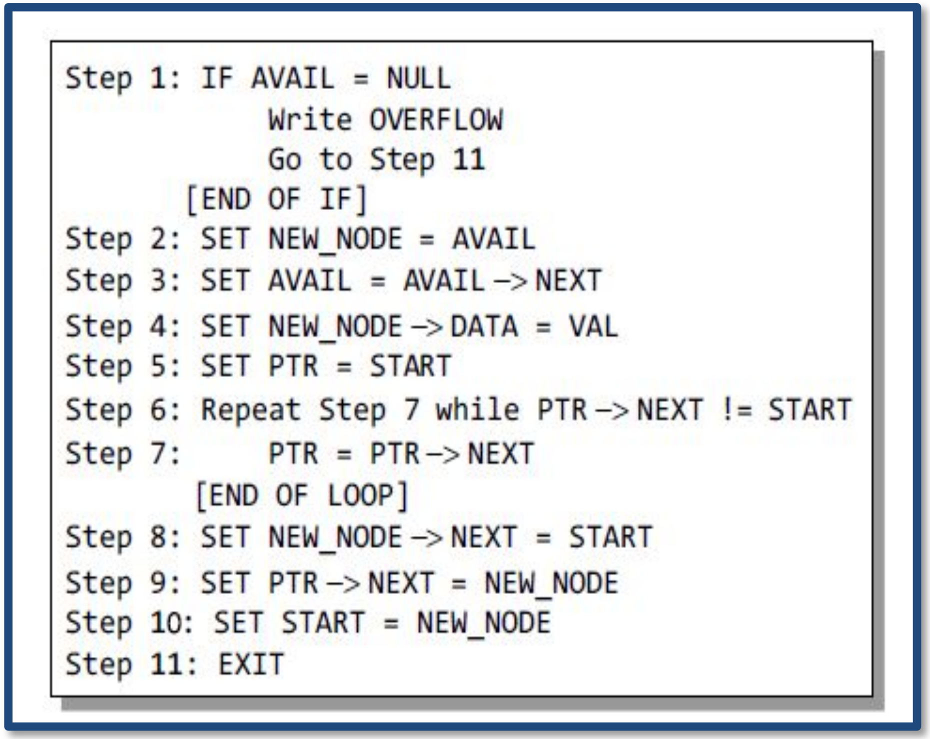
Take a pointer variable PTR that points to the START node of the list.

Move PTR so that it now points to the last node of the list.

Add the new node in between PTR and START.

Make START point to the new node.

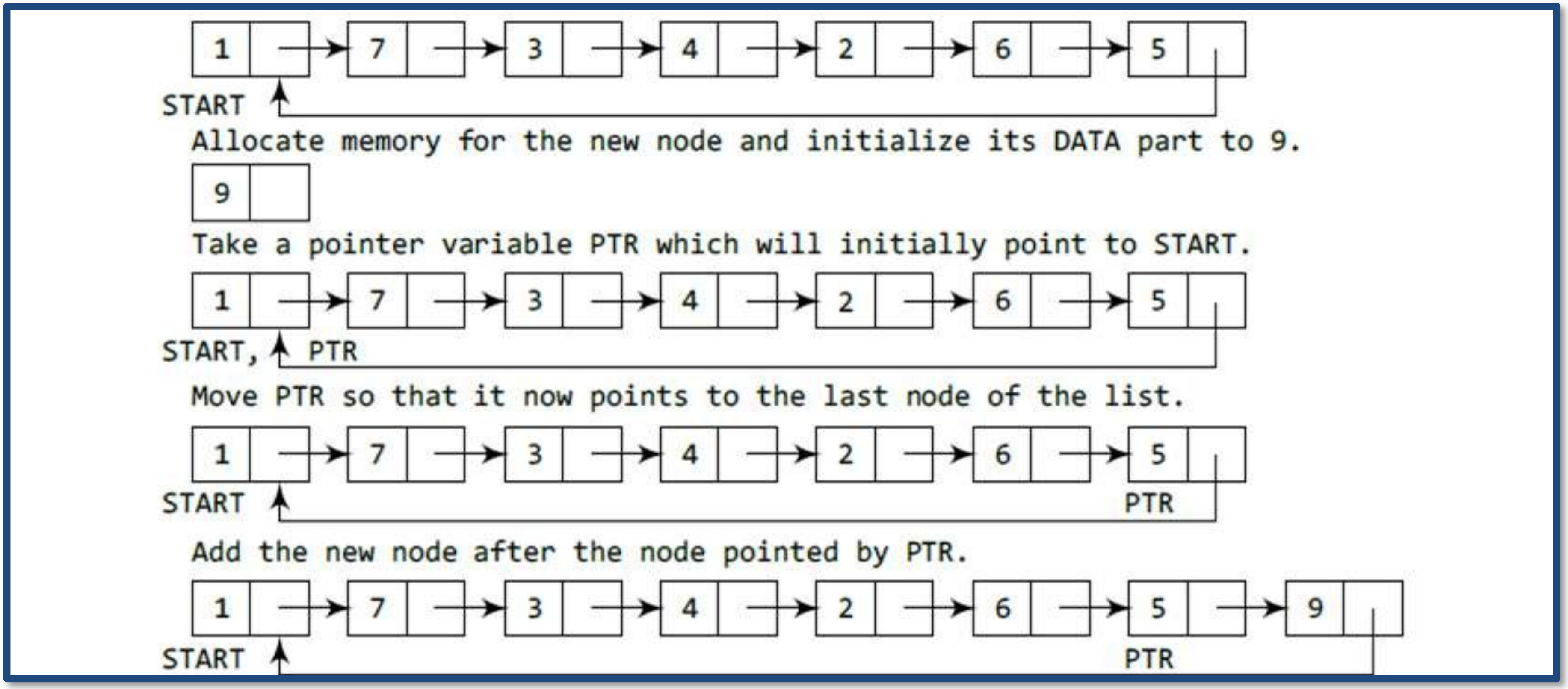
Case 1: The new node is inserted at the beginning



```
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 PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:       PTR = PTR -> NEXT
         [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

- Step 1, we first check whether memory is available for the new node
- 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.
- The START pointer variable will now hold the address of the NEW_NODE.
- While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

Insert node at end of circular link list



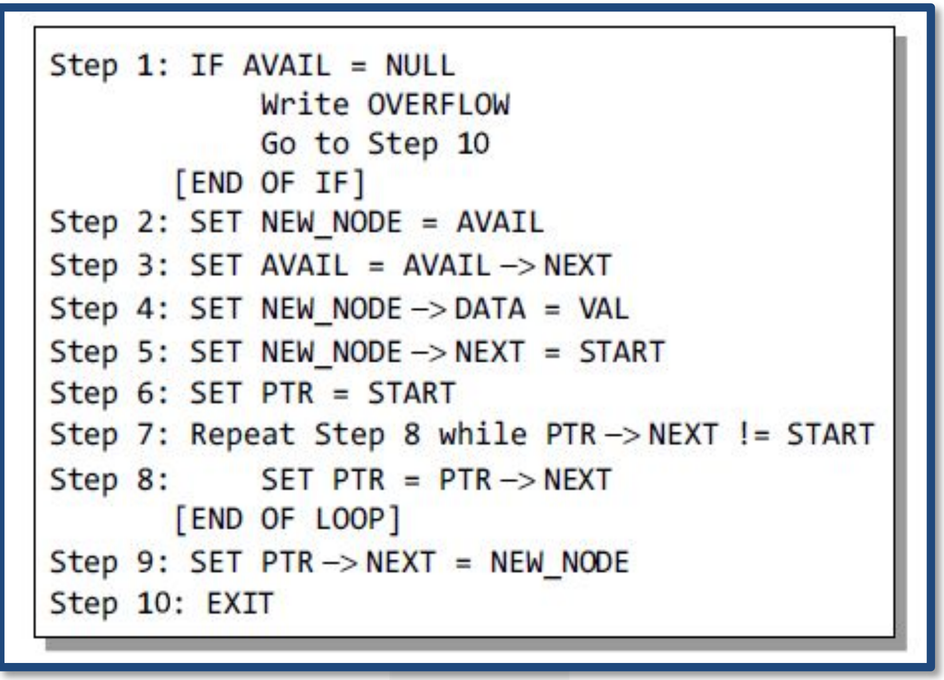
Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR which will initially point to START.

Move PTR so that it now points to the last node of the list.

Add the new node after the node pointed by PTR.

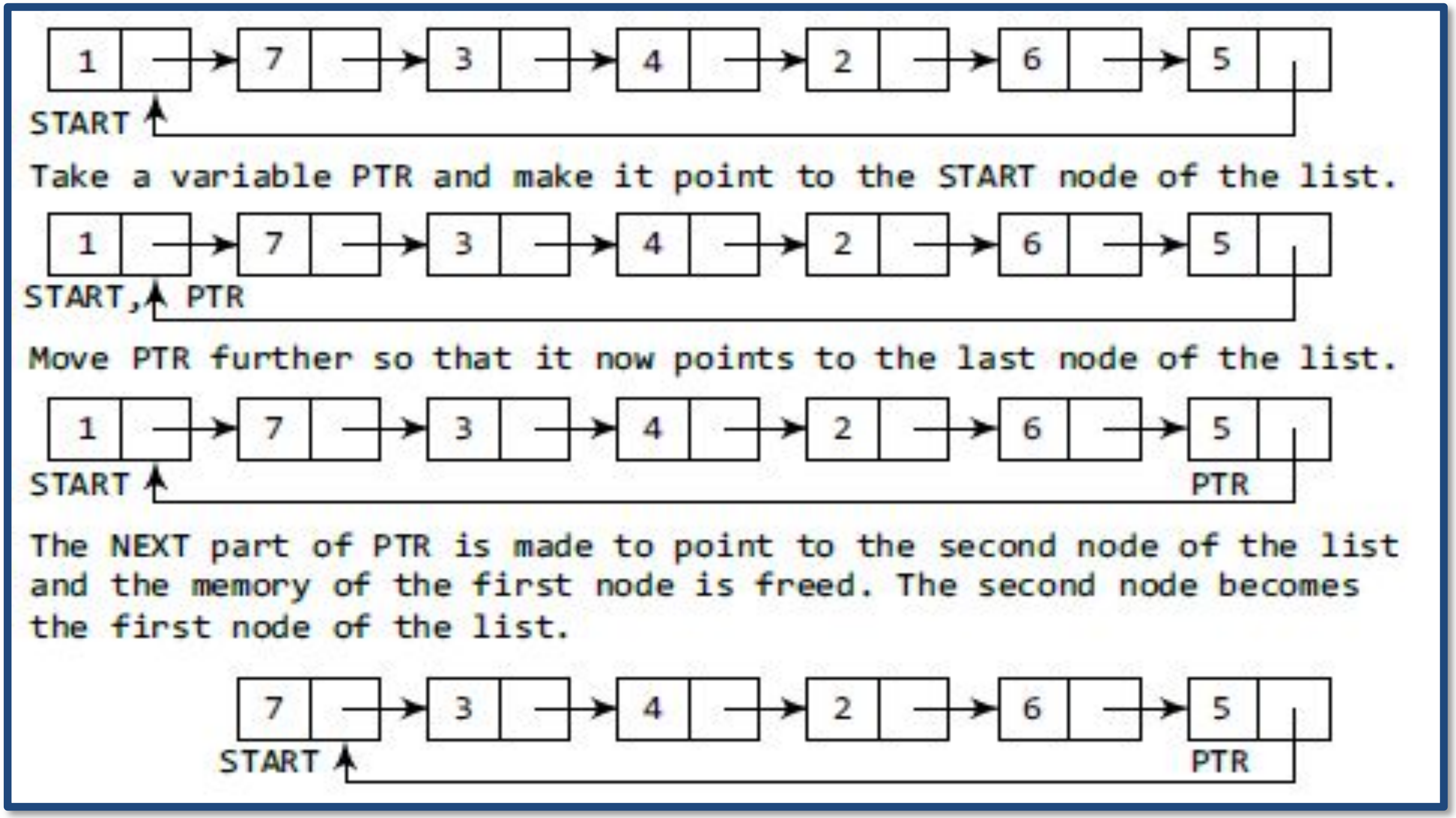
Insert node at end of circular link list



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

- In Step 6, 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 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 first of the new node contains the address of the fi node which is denoted by START.

Delete first node of circular linked list

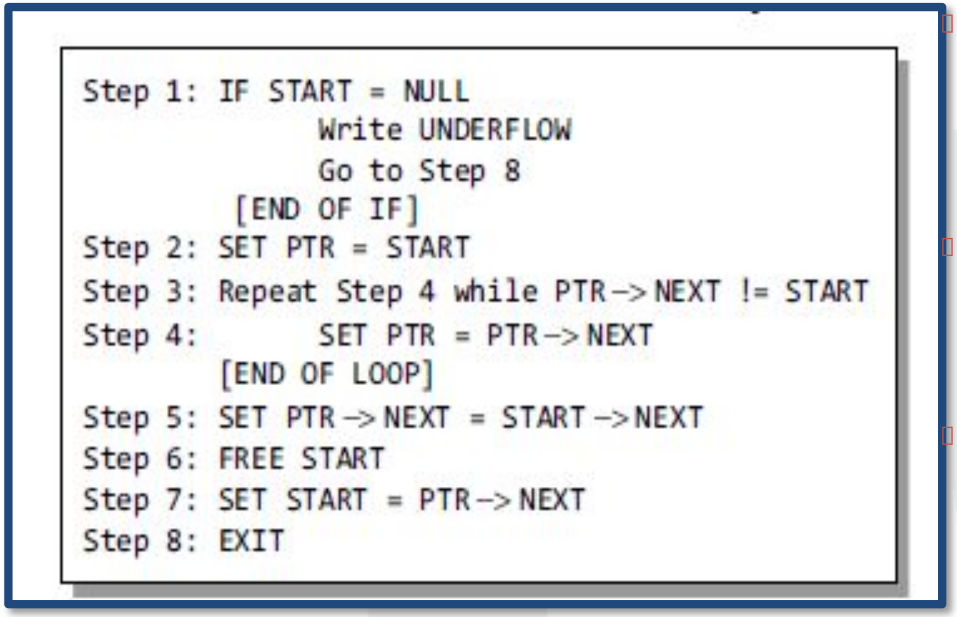


Take a variable PTR and make it point to the START node of the list.

Move PTR further so that it now points to the last node of the list.

The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

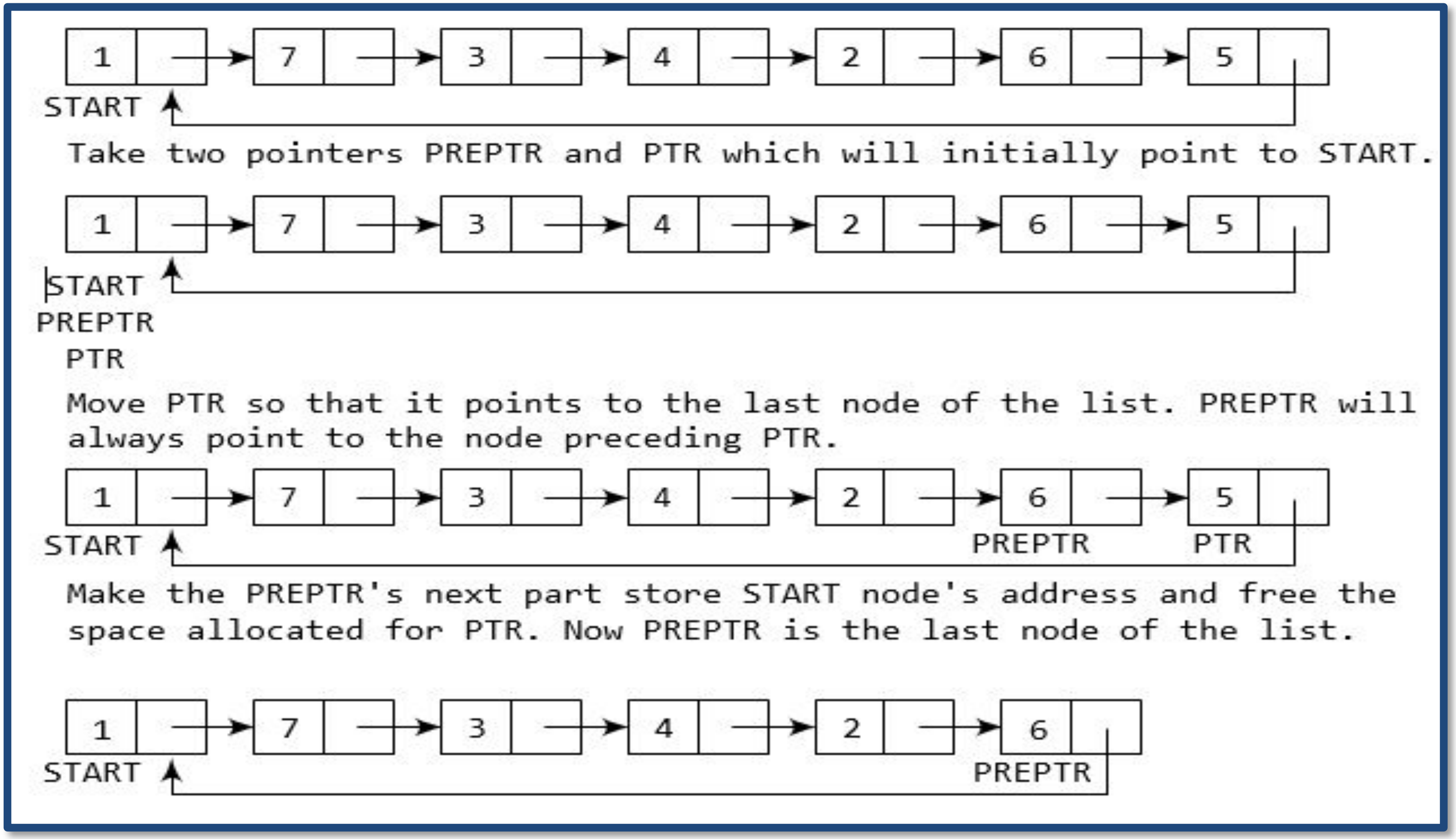
Delete first node of circular linked list



```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->NEXT != START
Step 4:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: SET PTR->NEXT = START->NEXT
Step 6: FREE START
Step 7: SET START = PTR->NEXT
Step 8: 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 pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list. In Step 6, the memory occupied by the first node is freed.
- Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START

Delete last node of circular linked list

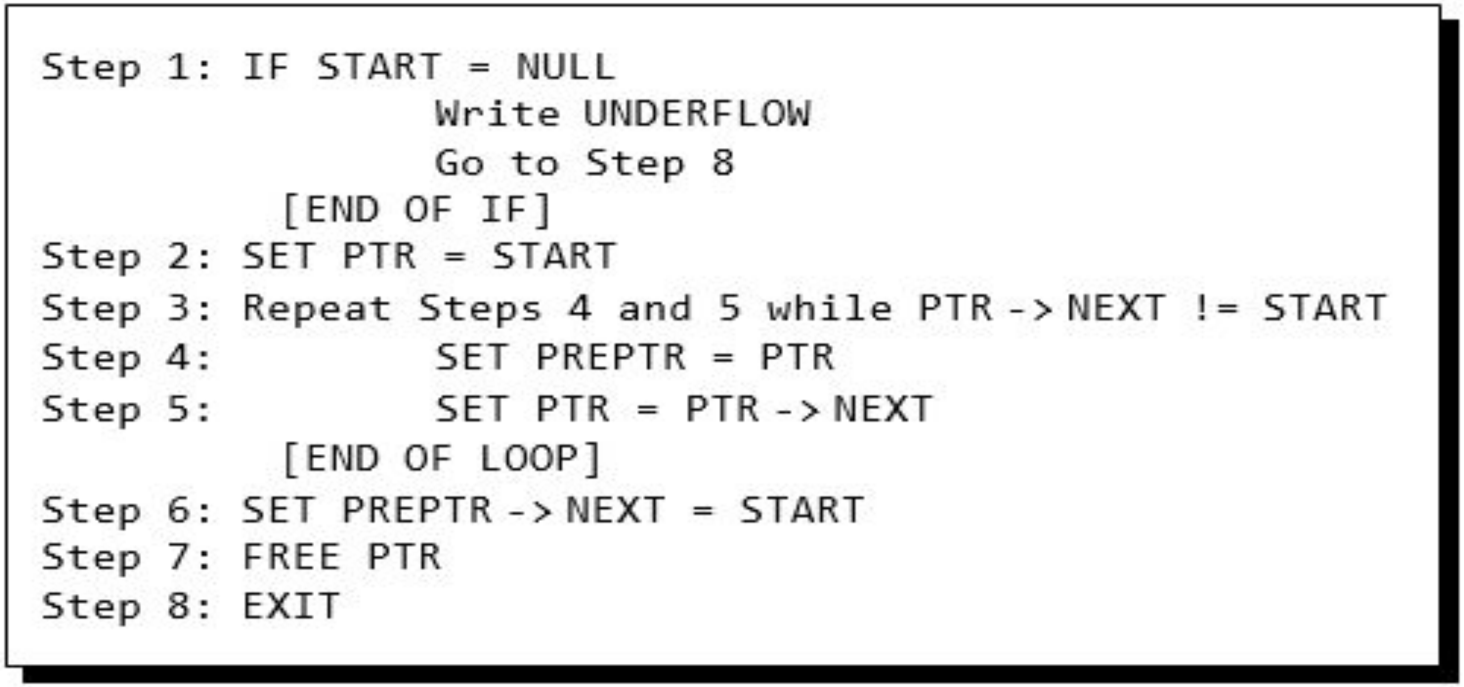


Take two pointers PREPTR and PTR which will initially point to START.

Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.

Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.

Delete last node of circular linked list



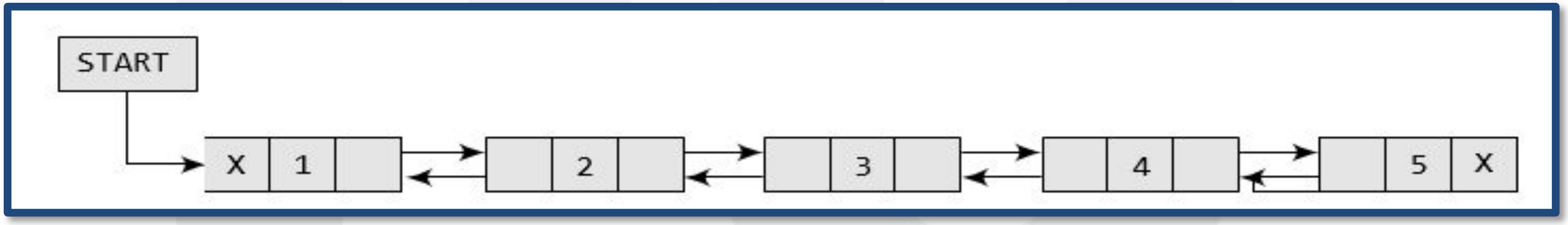
```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
          [END OF LOOP]
Step 6: SET PREPTR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

Delete last node of circular linked list

- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the fi node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR.
- Once we reach the last node and the second last node, we set the next pointer of the second last node to START, 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.

Doubly link list

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.



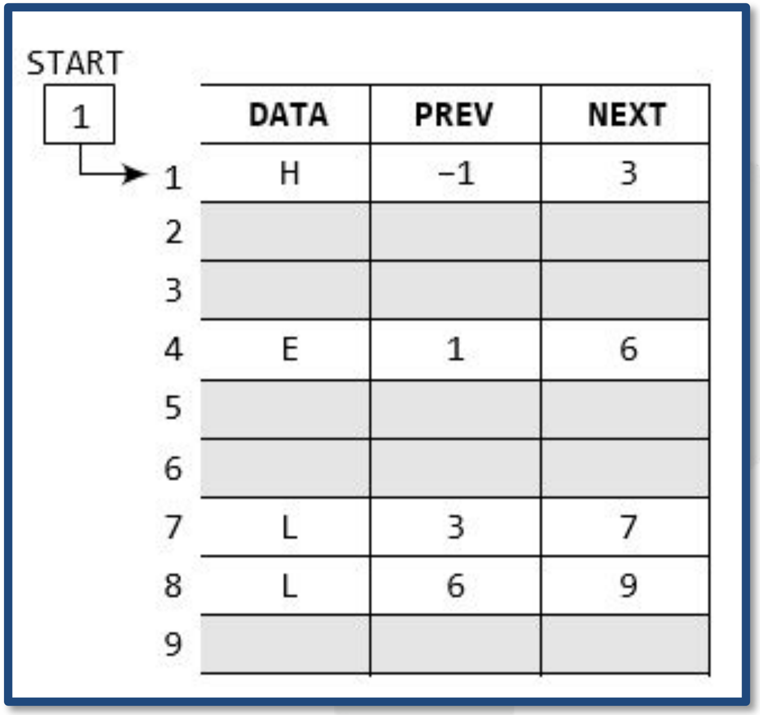
Doubly link list

In C, 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.

Memory representation of doubly linked list



START

1

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

- START is used to store the address of the first node
- 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.

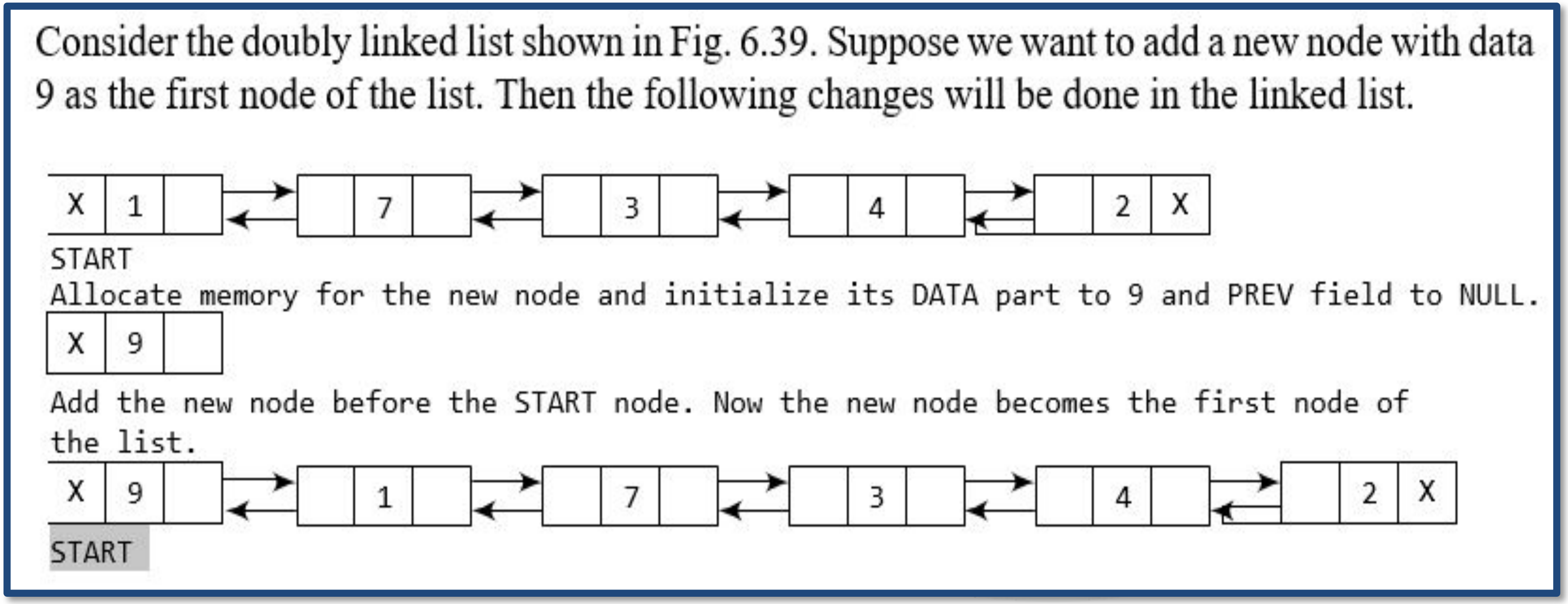
Insertion in doubly link list

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 1: The new node is inserted at the beginning



Consider the doubly linked list shown in Fig. 6.39. Suppose 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.

X
1
7
3
4
2
X
START

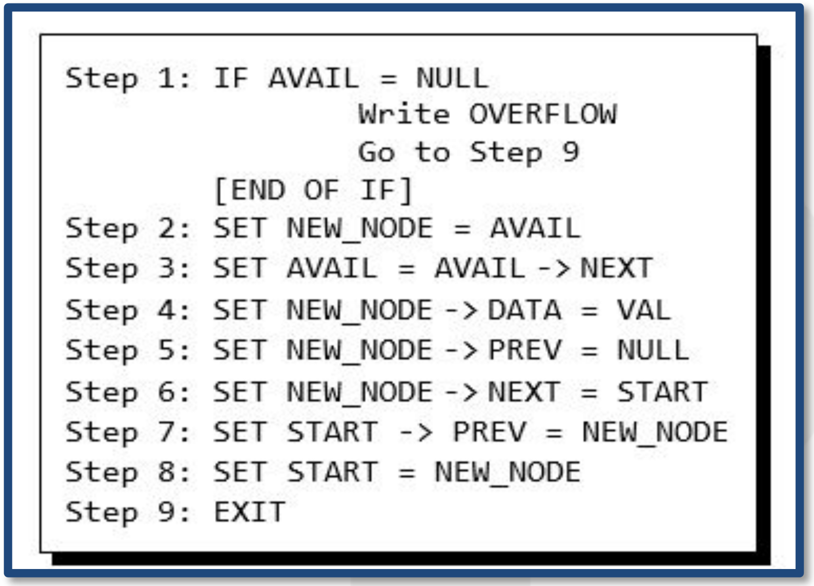
Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.

X
9

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

X
9
1
7
3
4
2
X
START

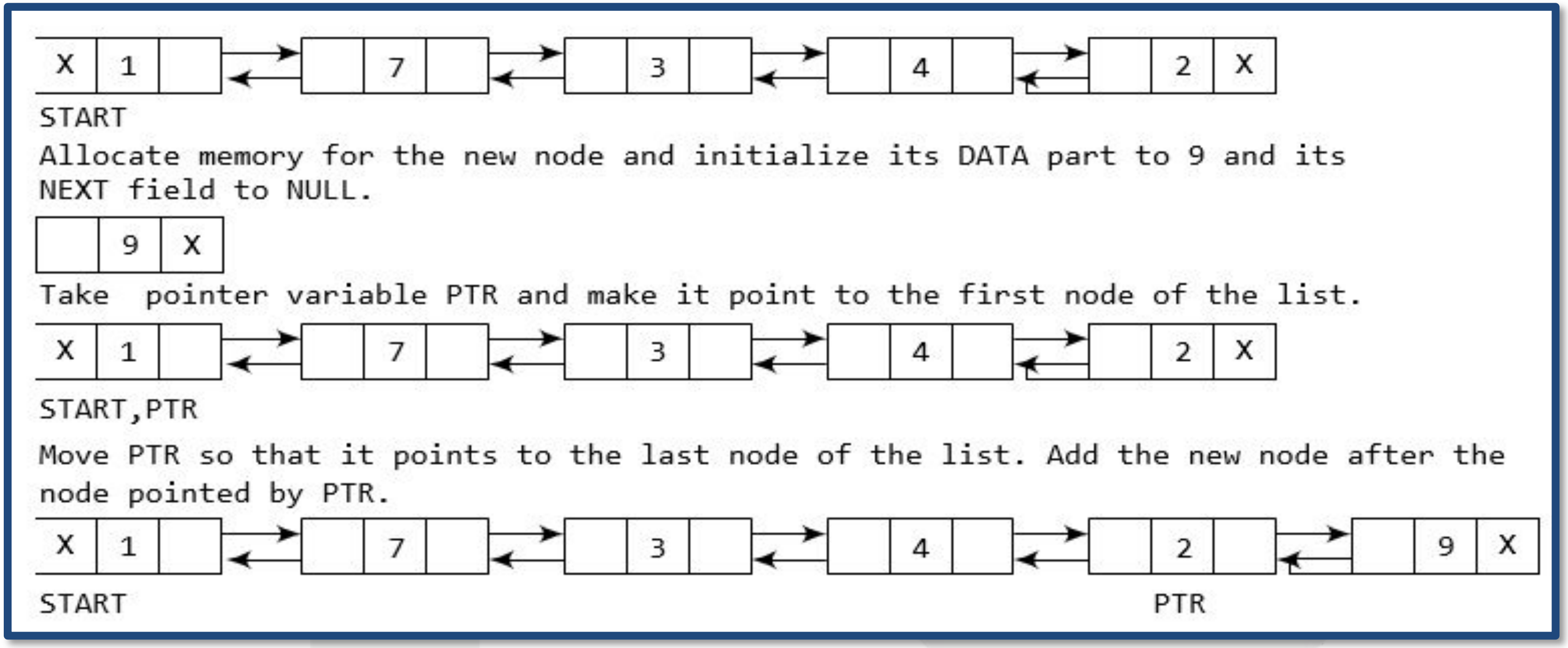
Case 1: The new node is inserted at the beginning



```
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



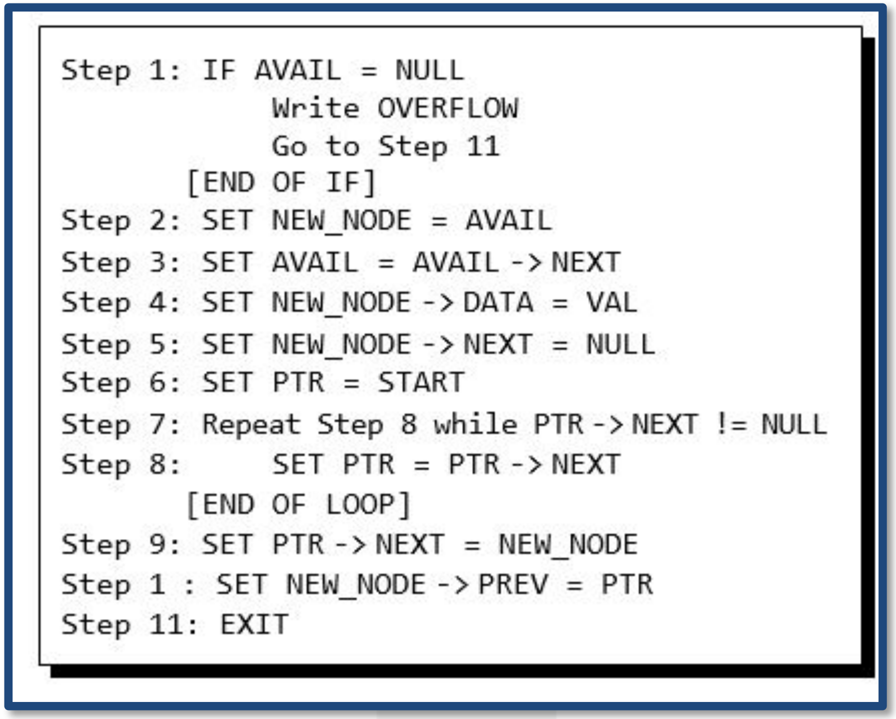
Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

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

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

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

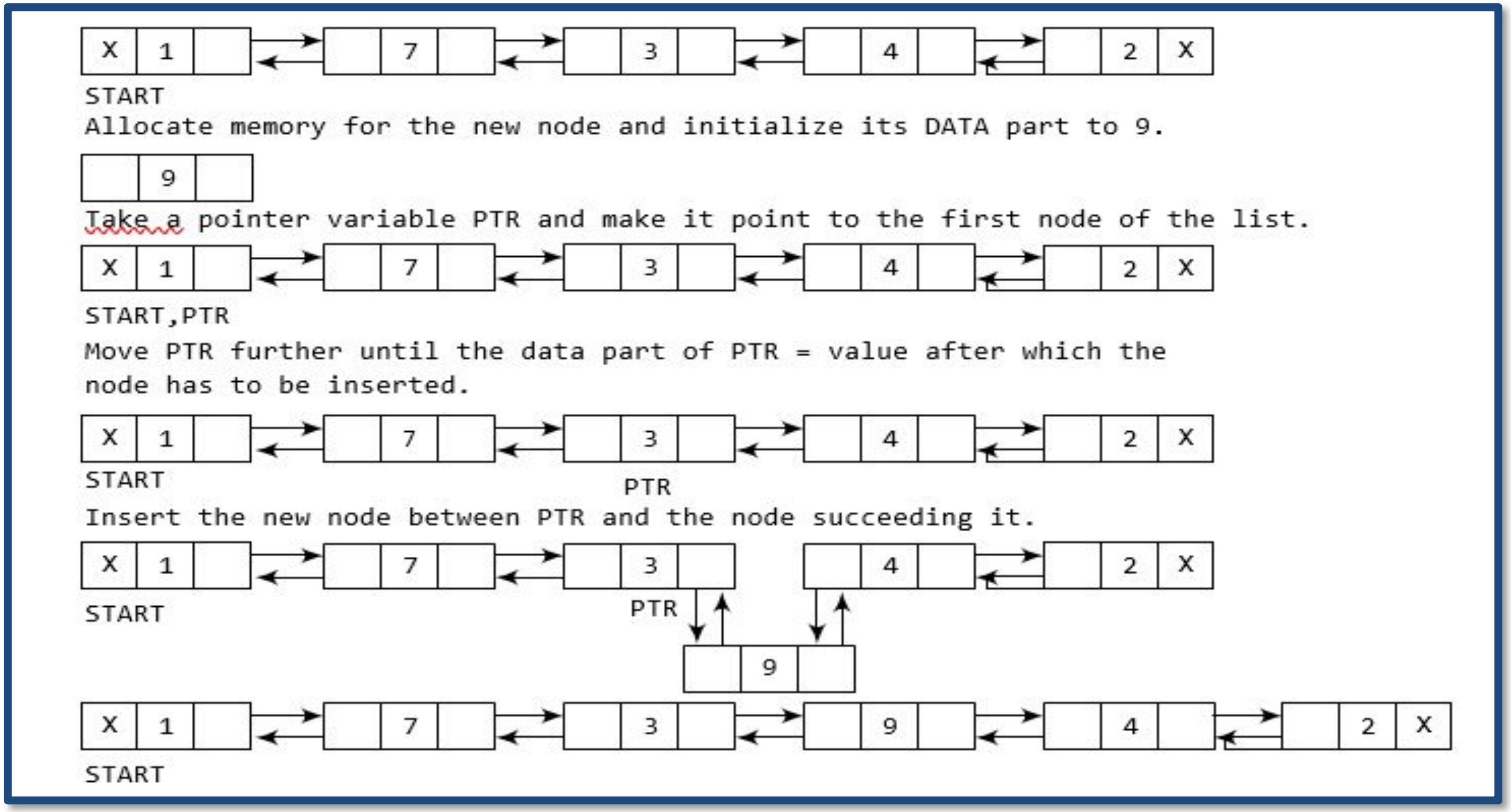
Case 2: The new node is inserted at the end



```
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 1 : 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



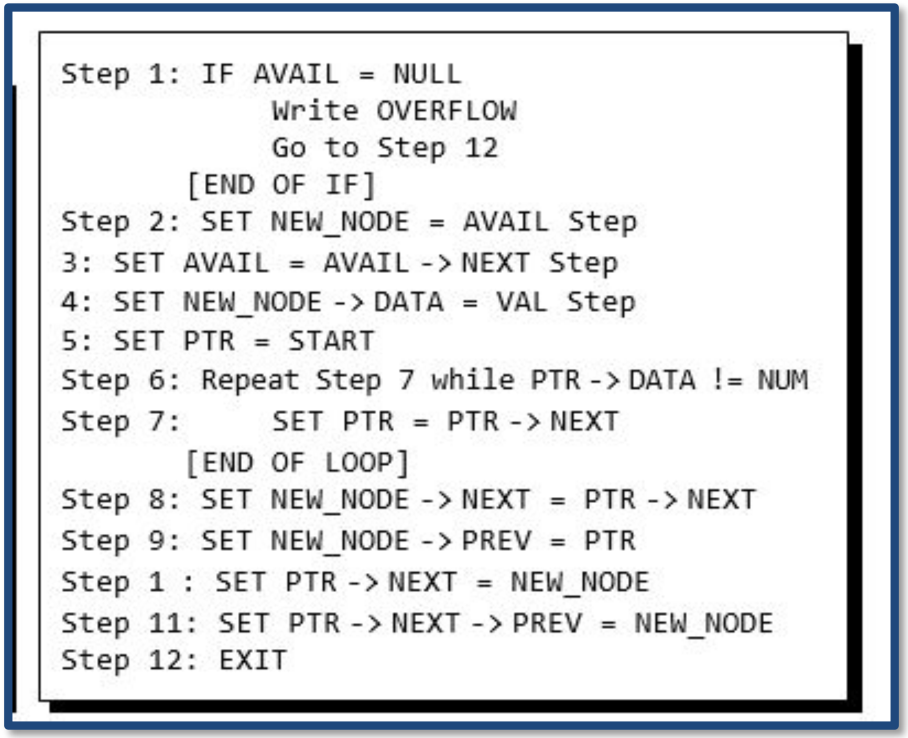
Allocate memory for the new node and initialize its DATA part to 9.

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

Move PTR further until the data part of PTR = value after which the node has to be inserted.

Insert the new node between PTR and the node succeeding it.

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



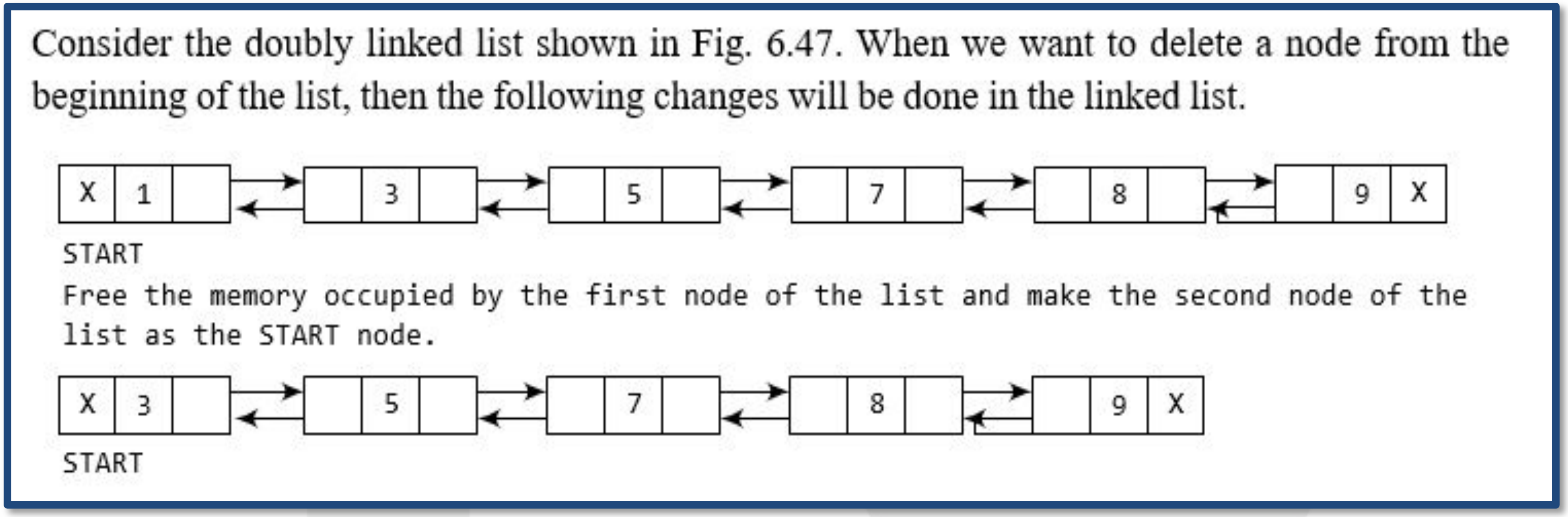
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL Step
3: SET AVAIL = AVAIL -> NEXT Step
4: SET NEW_NODE -> DATA = VAL Step
5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR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

Deletion in doubly link 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 1: The first node is deleted



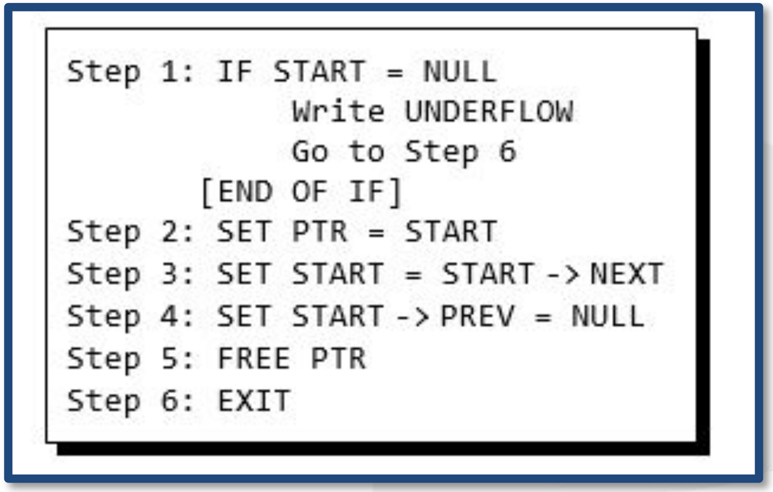
Consider the doubly linked list shown in Fig. 6.47. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

X 1
3
5
7
8
9 X
START

Free the memory occupied by the first node of the list and make the second node of the list as the START node.

X 3
5
7
8
9 X
START

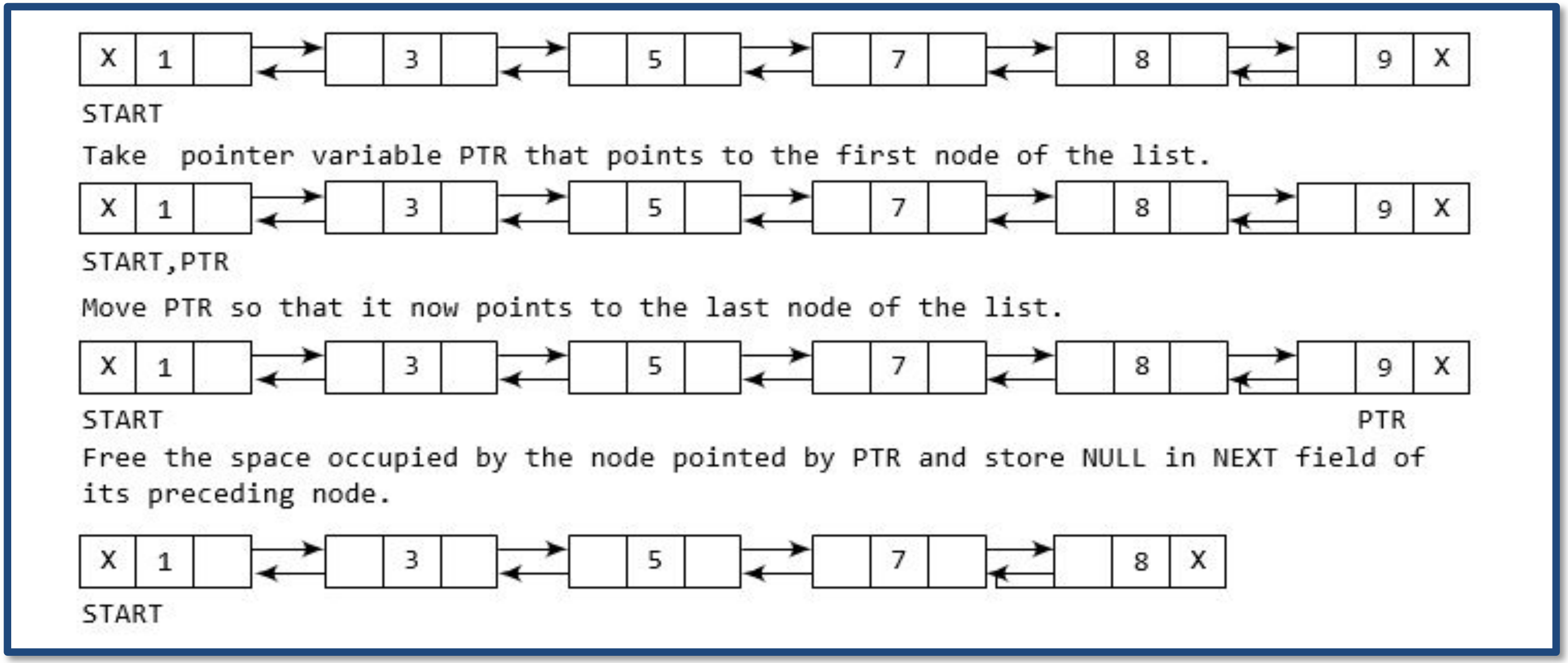
Case 1: The first node is deleted



```
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


X 1 → 3 → 5 → 7 → 8 → 9 X
START

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

X 1 → 3 → 5 → 7 → 8 → 9 X
START,PTR

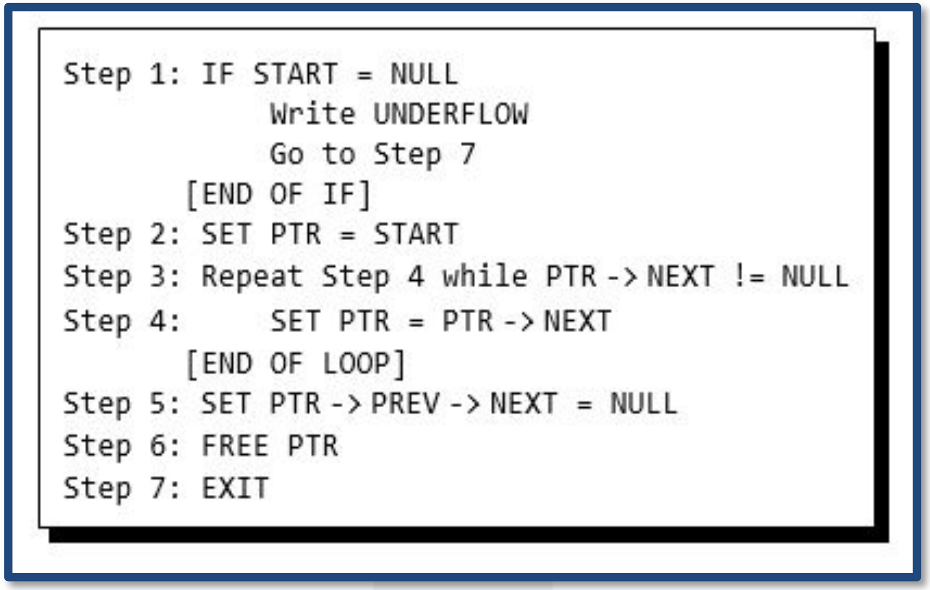
Move PTR so that it now points to the last node of the list.

X 1 → 3 → 5 → 7 → 8 → 9 X
START
PTR

Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.

X 1 → 3 → 5 → 7 → 8 X
START

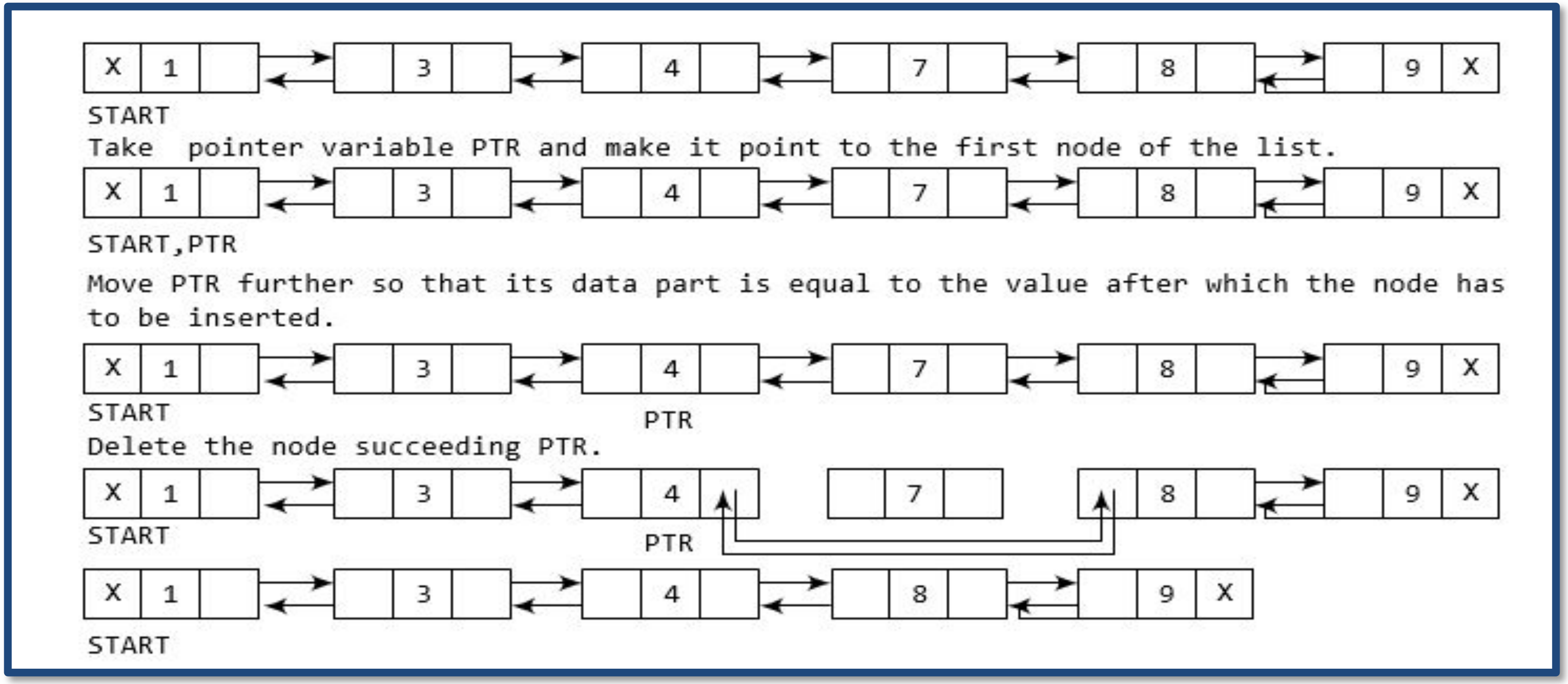
Case 2: The last node is deleted



```
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 fi 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

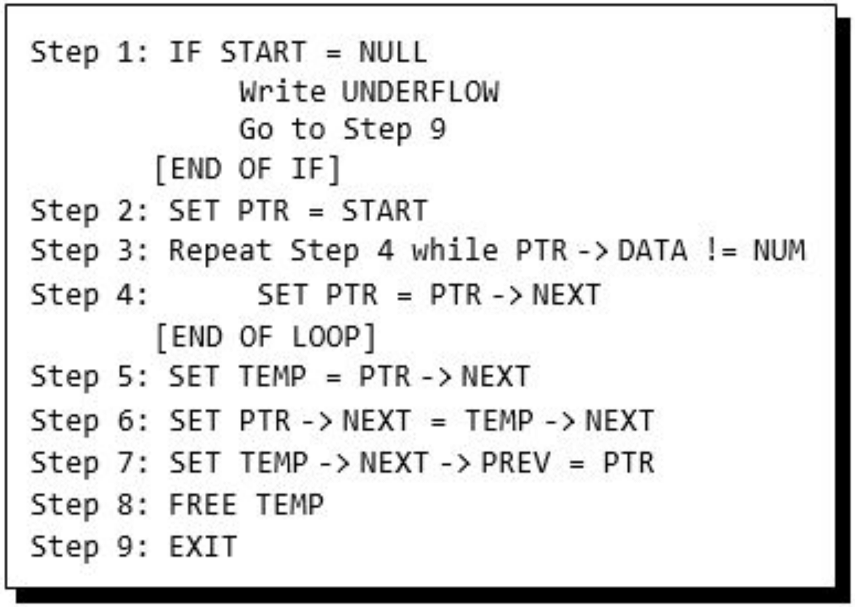


Take pointer variable PTR and make it point to the first node of the list.

Move PTR further so that its data part is equal to the value after which the node has to be inserted.

Delete the node succeeding PTR.

Case 3: The 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 an

Algorithm Complexities

Operation / type	Singly Linked List	Doubly Linked List	Circular Linked List
Insert at the begin	O(1)	O(1)	O(1)
Insert at the given index	O(n)	O(n)	O(n)
Insert at the end	O(n)	O(n)	O(1)
Delete at the begin	O(1)	O(1)	O(1)
Delete at the given index	O(n)	O(n)	O(n)
Delete the end	O(n)	O(n)	O(1)
Searching an element	O(n)	O(n)	O(n)

Applications of Linked List

Here is a concise list of applications of linked lists:

1. Dynamic Memory Allocation
2. Implementing Stacks
3. Implementing Queues
4. Graph Representation (Adjacency Lists)
5. Handling Large Numbers
6. Implementing Hash Tables (Chaining)
7. Dynamic Size Management
8. Efficient Insertions/Deletions
9. Memory Efficiency for Large Datasets
10. Polynomials Representation
11. Round-Robin Scheduling
12. Undo Mechanisms in Software
13. Sparse Matrix Representation
14. Real-time Applications (Video Buffering)
15. Navigational Systems (Browser History)
16. Music Playlists (Cyclic or Doubly LL)
17. Image Viewing Applications
18. File System Directories

This list highlights the versatility and wide-ranging uses of linked lists in computer science and software engineering.

References….

- Data structure through C , G.S Baluja
- Data structure through C in depth, S.K Srivastava
- AV Aho, J Hopcroft, JD Ullman, Data Structures and Algorithms, Addison- Wesley
- TH Cormen, CF Leiserson, RL Rivest, C Stein, Introduction to Algorithms, 3rd Ed., MIT Press

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.i