UNIT-II

1. LINKED LIST

1.1 Introduction:

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. 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.

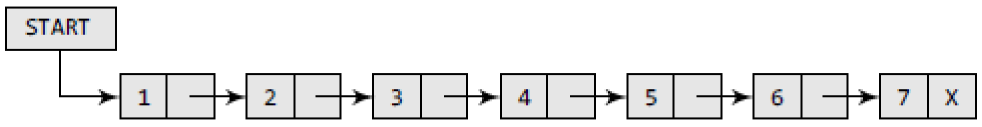


Figure 1.1 Simple linked list

In Fig. 1.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The left part of the node which contains data may include a simple data type, an array, or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL.

In Fig. 1.1, the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

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

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

Note:

Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.

1.2 SINGLE LINKED LISTS:

A single linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A single linked list allows traversal of data only in one way. Figure 1.2 shows a single linked list.

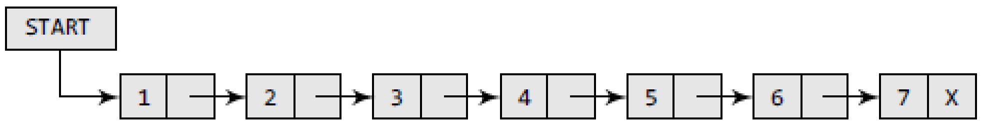


Figure 1.2 Single linked list

1.3 REPRESENTATION OF LINKED LIST IN MEMORY:

Let us see how a linked list is maintained (represented) in the memory. In order to form a linked list, we need a structure called *node* which has two fields, DATA and NEXT. DATA will store the information part and NEXT will store the address of the next node in sequence.

Consider Fig. 1.3.

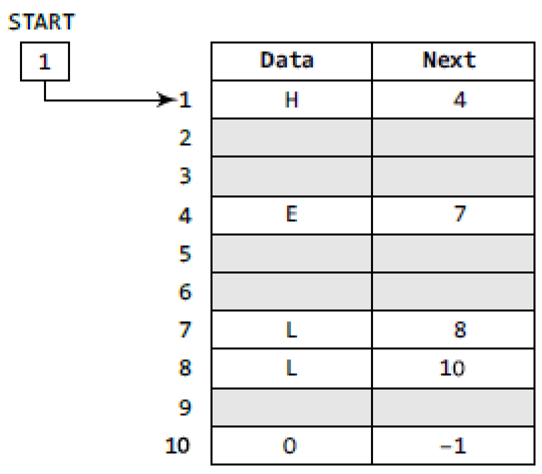


Figure 1.3 Memory Representation of a linked list

In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START = 1, so the first data is stored at address 1, which is H. The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.

The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data. We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list. When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

Note that Fig. 1.3 shows a chunk of memory locations which range from 1 to 10. The shaded portion contains data for other applications. Remember that the nodes of a linked list need not be in consecutive memory locations. In our example, the nodes for the linked list are stored at addresses 1, 4, 7, 8, and 10.

1.4 OPERATIONS ON SINGLE LINKED LIST:

1.4.1 Inserting a New Node in a Linked List:

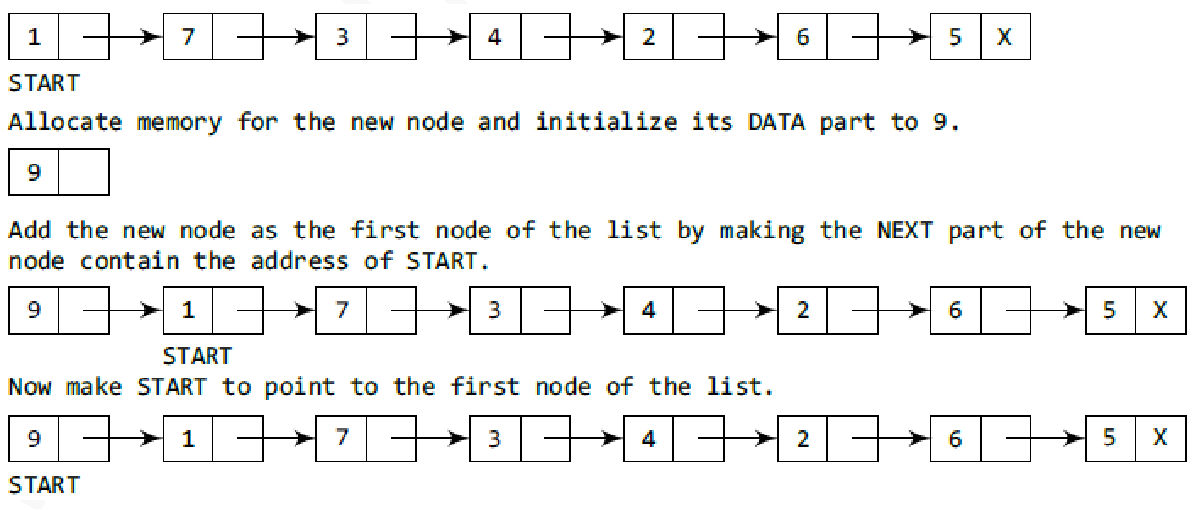
In this section, we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

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.

Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

Inserting a Node at the Beginning of a Linked List:

Consider the linked list shown in Fig. 1.4. Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.



1 7 3 4 2 6 5 X
START
Allocate memory for the new node and initialize its DATA part to 9.
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.
9 1 7 3 4 2 6 5 X
START
Now make START to point to the first node of the list.
9 1 7 3 4 2 6 5 X
START

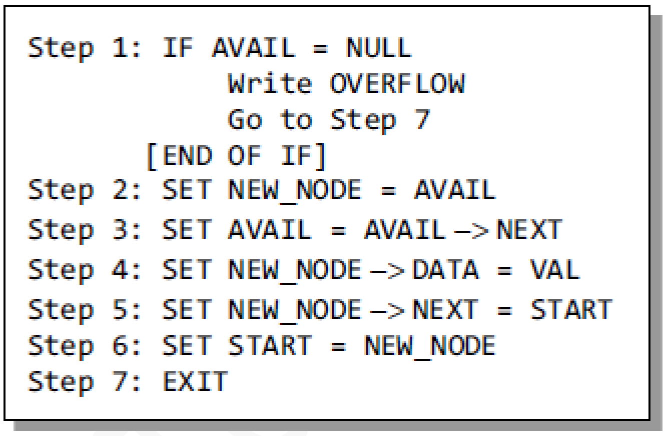
Figure 1.4 Inserting an element at the beginning of a linked list

Figure 1.5 shows the algorithm to insert a new node at the beginning of a linked list. 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 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. Note the following two steps:

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc(), and calloc() which automatically do the memory allocation on behalf of the user.

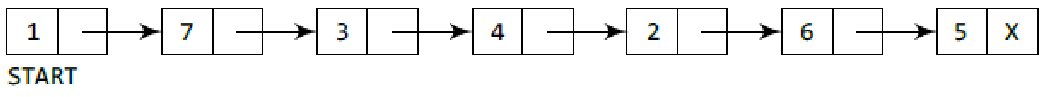


```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 7
           [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

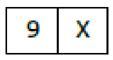
Figure 1.5 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Linked List:

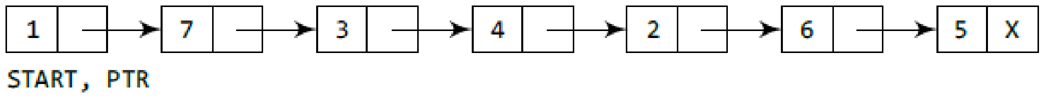
Consider the linked list shown in Fig. 1.6. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



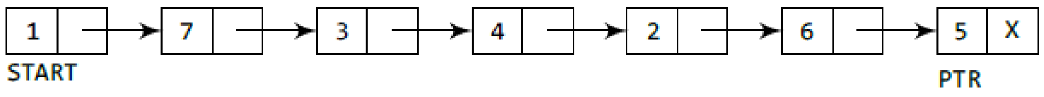
```
Allocate memory for the new node and initialize its DATA part to 9 and
NEXT part to NULL.
```



```
Take a pointer variable PTR which points to START.
```



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



```
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.
```

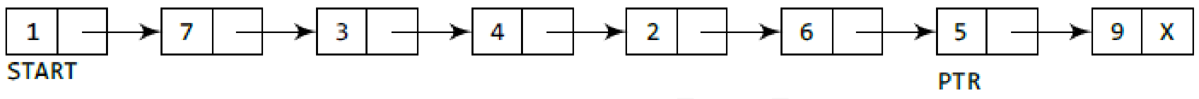
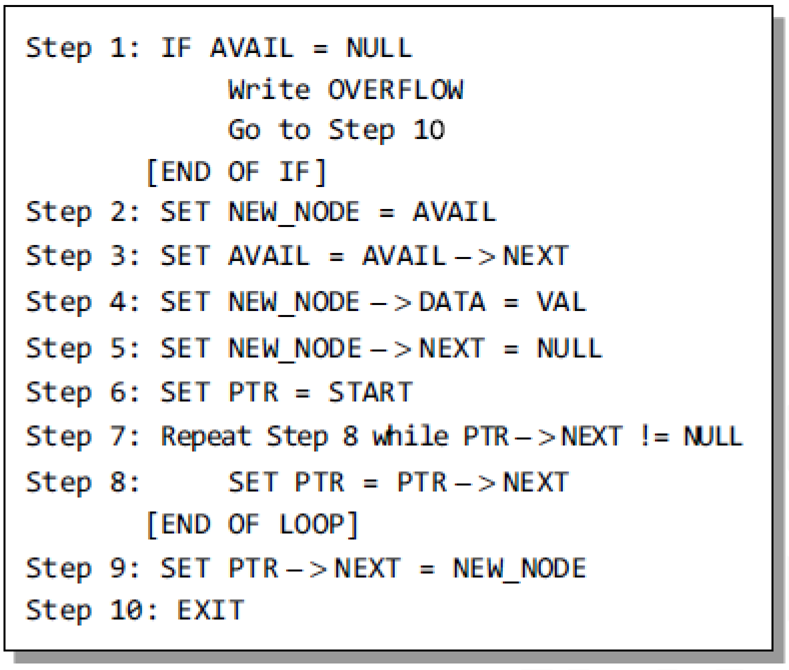


Figure 1.6 Inserting an element at the end of a linked list

Figure 1.7 shows the algorithm to insert a new node at the end of a linked list. 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.

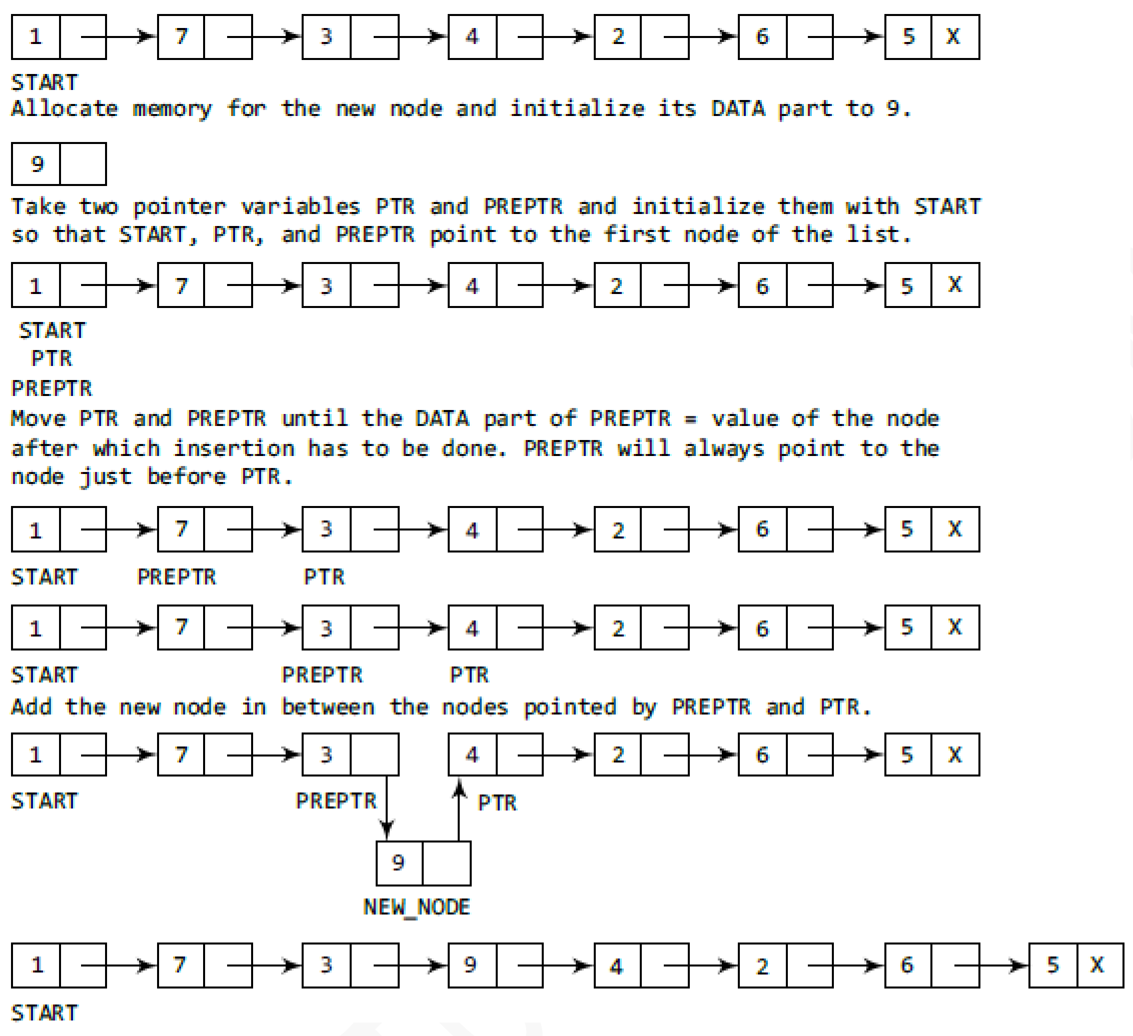


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step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

Figure 1.7 Algorithm to insert a new node at the end

Inserting a Node After a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.8. Suppose we want to add a new node with value 9 after the node containing data 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.9.



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.

Figure 1.8 Inserting an element after a given node in a linked list

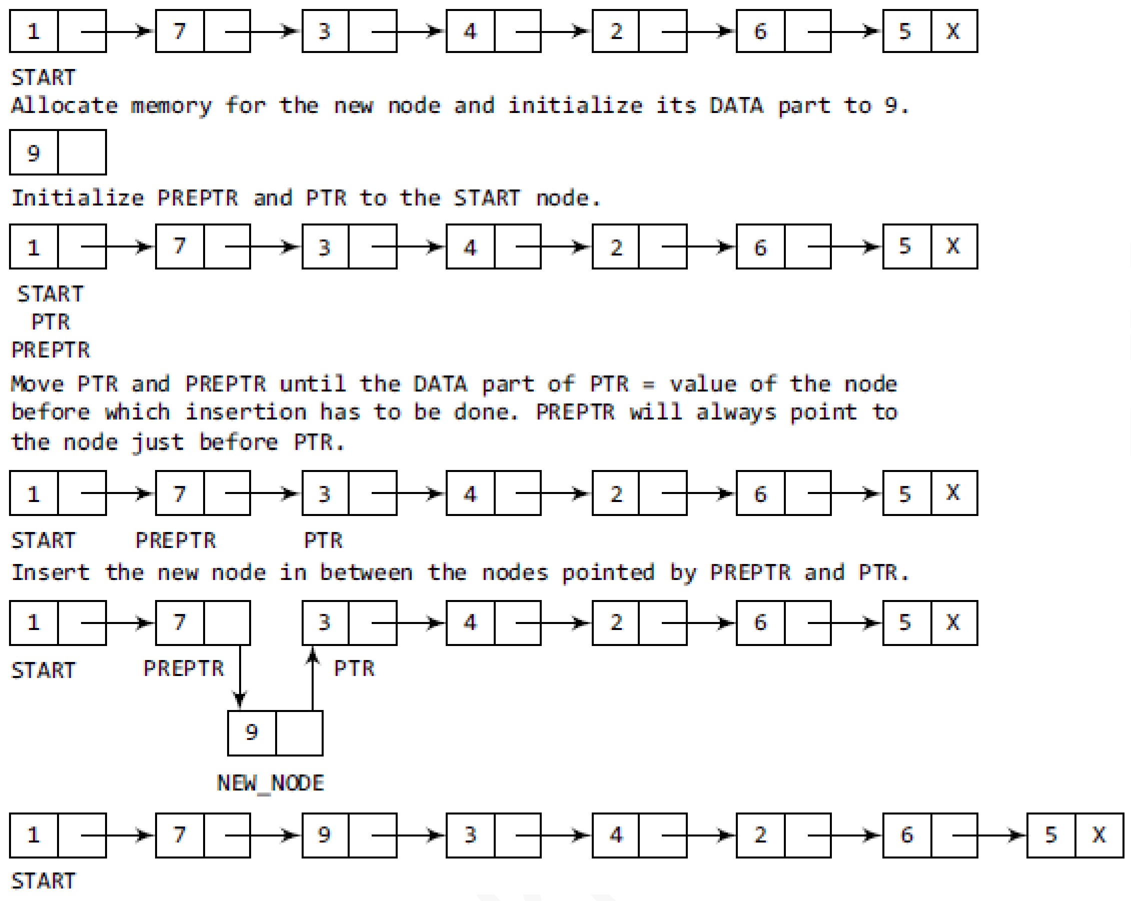
In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, 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.

```
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
```

Figure 1.9 Algorithm to insert a new node after a node that has value NUM

Inserting a Node Before a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.10. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.11.



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

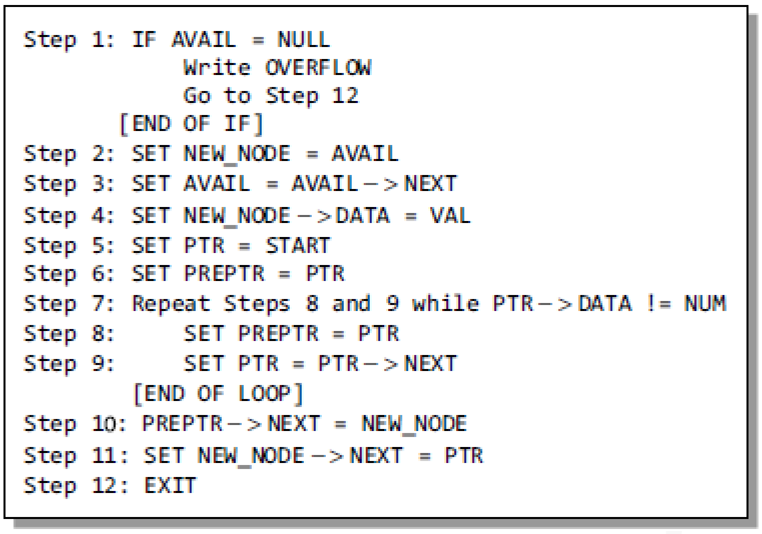
Initialize PREPTR and PTR to the START node.

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

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

Figure 1.10 Inserting an element before a given node in a linked list

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 and initialize it with 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 before this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired 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: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->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
```

Figure 1.11 Algorithm to insert a new node before a node that has value NUM

1.4.2 Deleting a Node from a Linked List:

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

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

Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

Deleting the First Node from a Linked List:

Consider the linked list in Fig. 1.12. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

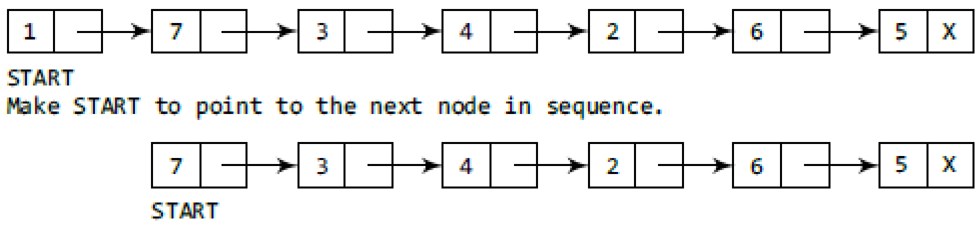


Figure 1.12 Deleting the first node of a linked list

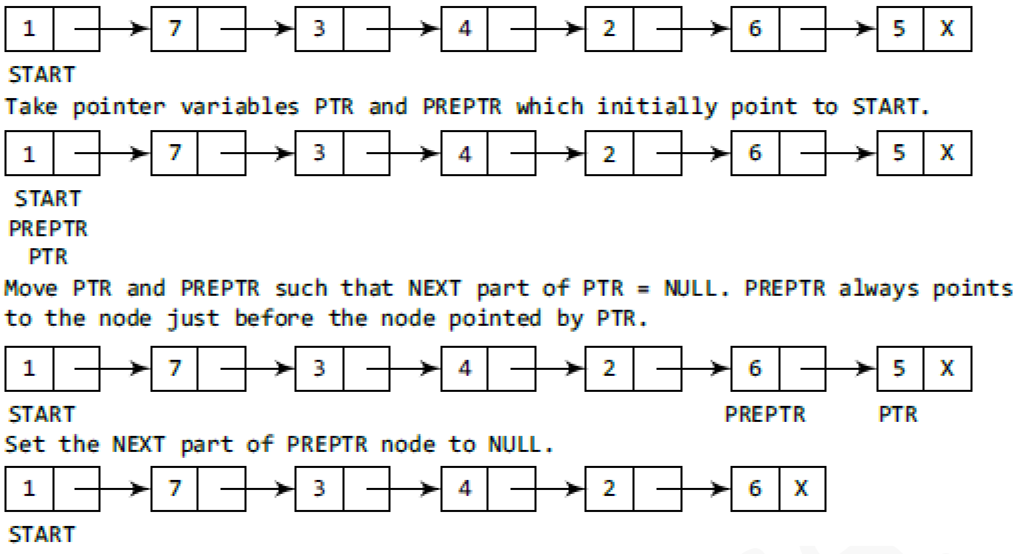
Figure 1.13 shows the algorithm to delete the first node from a linked list. In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes 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.

```
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
```

Figure 1.13 Algorithm to delete the first node

Deleting the Last Node from a Linked List:

Consider the linked list shown in Fig. 1.14. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



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

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

Set the NEXT part of PREPTR node to NULL.

Figure 1.14 Deleting the last node of a linked list

Figure 1.15 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.

```
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
```

Figure 1.15 Algorithm to delete the last node

Deleting the Node After a Given Node in a Linked List:

Consider the linked list shown in Fig. 1.16. Suppose 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.

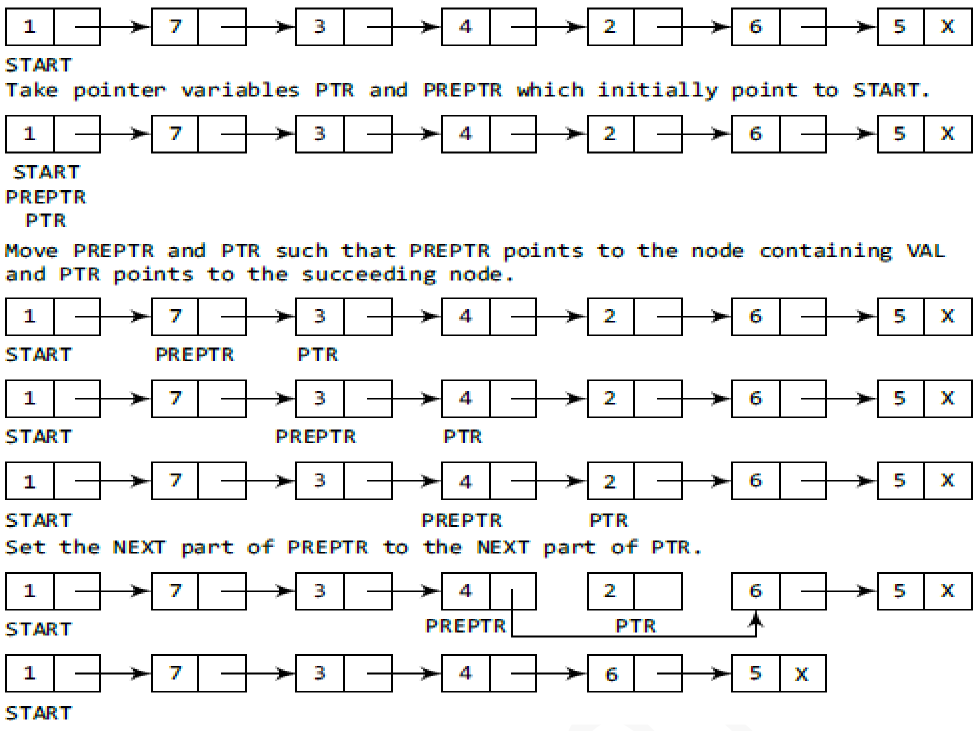

1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
Take pointer variables PTR and PREPTR which initially point to START.
1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR
Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.
1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR
1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR
1 → 7 → 3 → 4 → 2 → 6 → 5 X
START
PREPTR
PTR
Set the NEXT part of PREPTR to the NEXT part of PTR.
1 → 7 → 3 → 4 → 6 → 5 X
2
START
PREPTR
PTR
1 → 7 → 3 → 4 → 6 → 5 X
START

Figure 1.16 Deleting the node after a given node in a linked list

Figure 1.17 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.

```
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
```

Figure 1.17 Algorithm to delete the node after a given node

1.4.3 Searching for a Value in a Linked List:

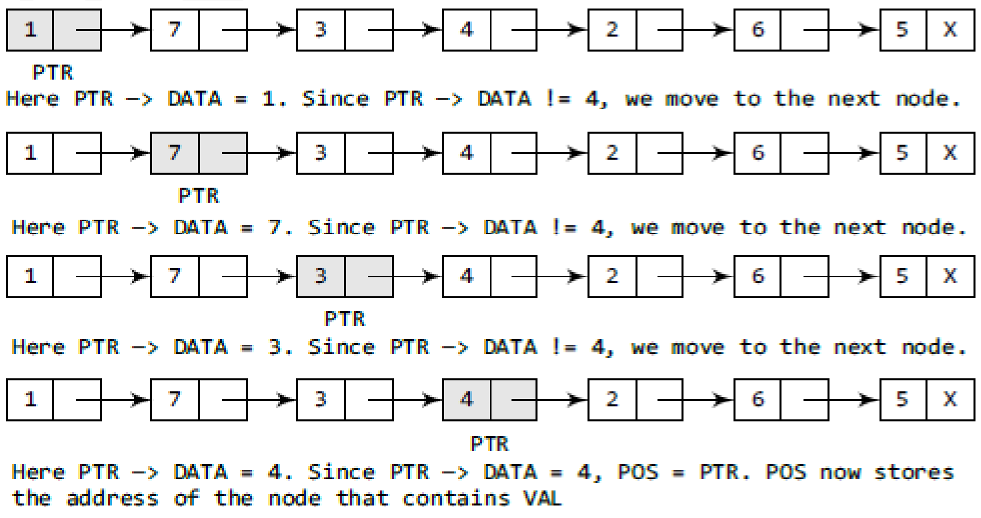
Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
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
```

Figure 1.18 Algorithm to search a linked list

Figure 1.18 shows the algorithm to search a linked list. 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.

Consider the linked list shown in Fig.1.19. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.



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

Figure 1.19 Searching a linked list

6.2.1 Traversing a Linked List:

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. The algorithm to traverse a linked list is shown in Fig. 1.20.

```
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
```

Figure 1.20 Algorithm for traversing a linked list

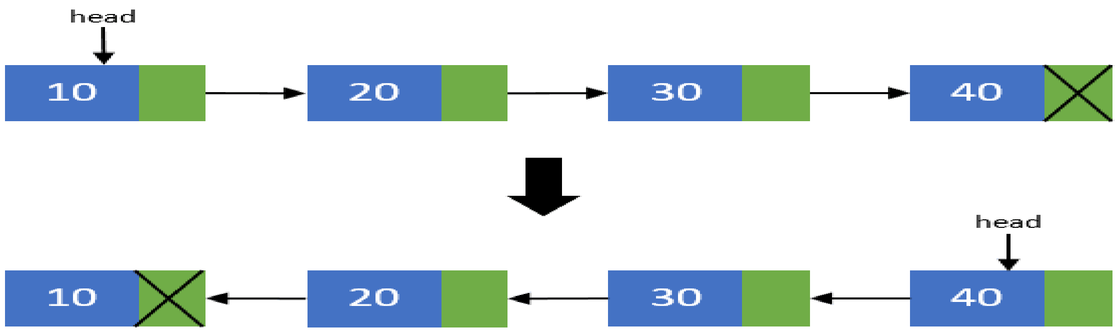
In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list. Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL. In Step 3, we apply the process (e.g., print) to 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.

Let us now write an algorithm to count the number of nodes in a linked list. To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Figure 1.21 shows the algorithm to print the number of nodes in a linked list.

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:             SET COUNT = COUNT + 1
Step 5:             SET PTR = PTR->NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Figure 1.21 Algorithm to print the number of nodes in a linked list

1.5 REVERSING SINGLE LINKED LIST:

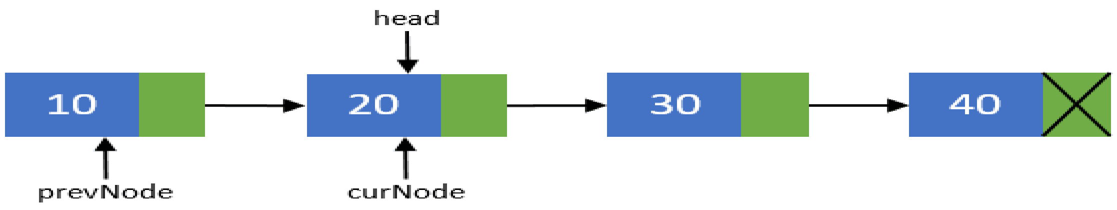


Algorithm to reverse a Single Linked List:

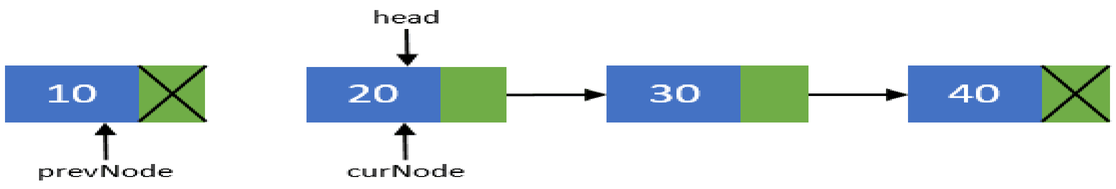
Input: **head** node of the linked list

```
Begin:
    If (head != NULL) then
        prevNode ← head
        head ← head.next
        curNode ← head
        prevNode.next ← NULL
        While (head != NULL) do
            head ← head.next
            curNode.next ← prevNode
            prevNode ← curNode
            curNode ← head
        End while
        head ← prevNode
    End if
End
```

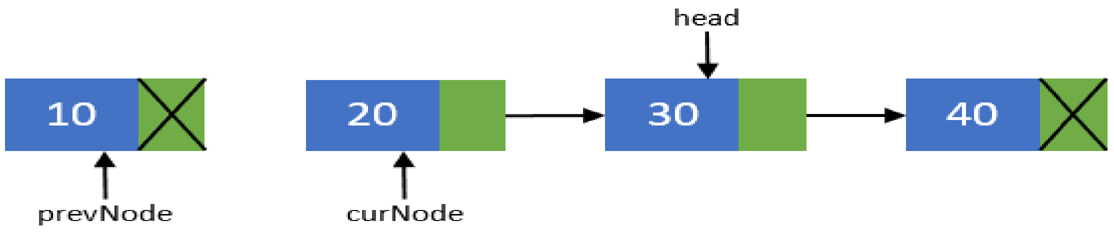
1) Create two more pointers other than head namely prevNode and curNode that will hold the reference of previous node and current node respectively. Make sure that prevNode points to first node i.e. prevNode = head. head should now point to its next node i.e. the second node head = head->next. curNode should also points to the second node i.e. curNode = head.



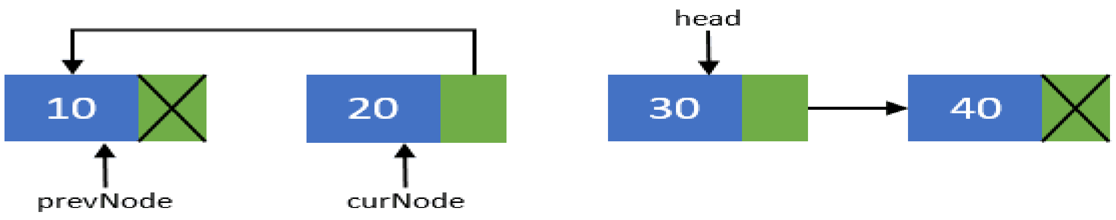
2) Now, disconnect the previous node i.e. the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation prevNode->next = NULL.



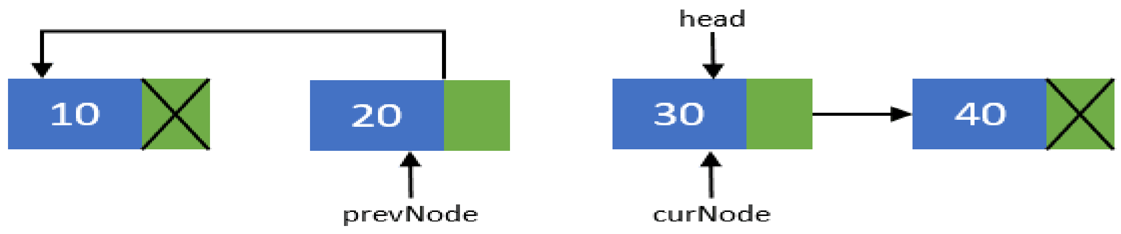
3) Move head node to its next node i.e. head = head->next.



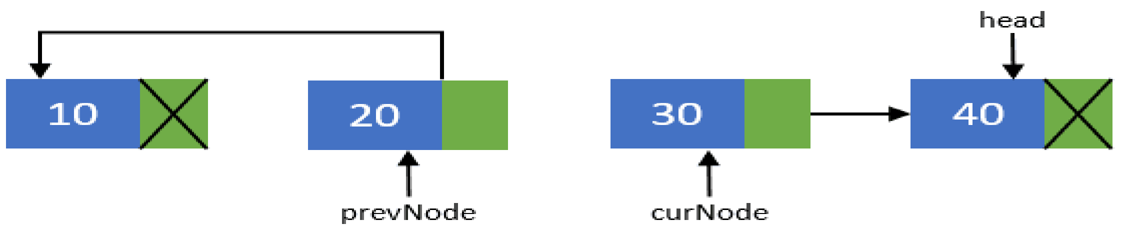
4) Now, re-connect the current node to its previous node i.e. curNode->next = prevNode;.

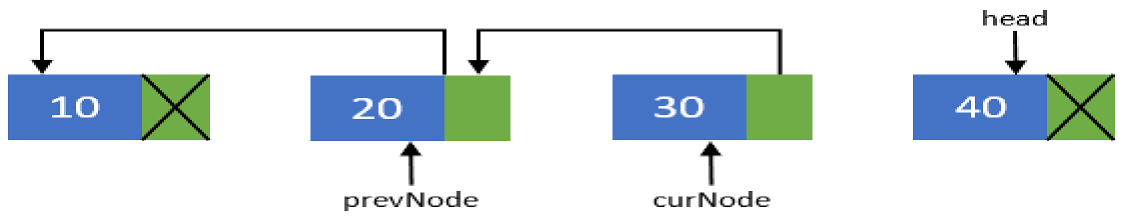


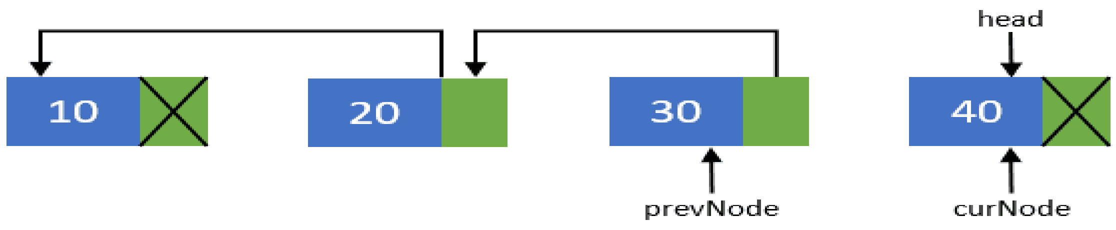
5) Point the previous node to current node and current node to head node. Means they should now point to prevNode = curNode; and curNode = head.

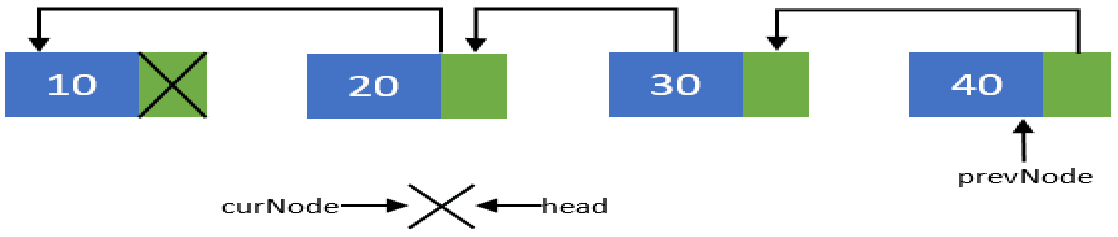


6) Repeat steps 3-5 till head pointer becomes NULL.

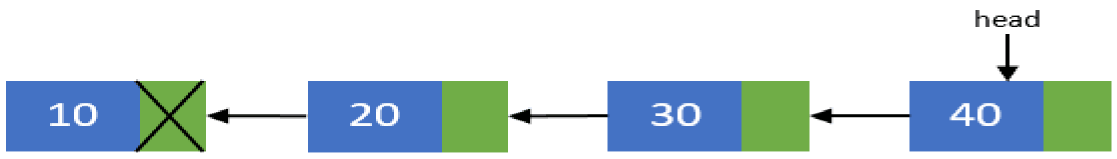








7) Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the **head** pointer should point to prevNode pointer. Perform head = prevNode;. Finally you end up with a reversed linked list of its original.



1.6 APPLICATIONS ON SINGLE LINKED LIST:

Linked lists can be used to represent polynomials and the different operations that can be performed on them. In this section, we will see how polynomials are represented in the memory using linked lists.

1.6.1 Polynomial Expression Representation:

Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.

Every term of a polynomial can be represented as a node of the linked list. Figure 1.22 shows the linked representation of the terms of the above polynomial.

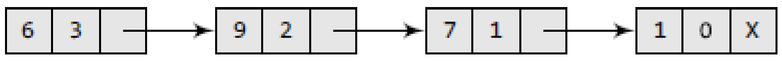


Figure 1.22 Linked representation of a polynomial

1.6.2 Addition and Multiplication:

Addition of Polynomials:

In this approach we will add the coefficients of elements having the same power in both the polynomial equations.

Example:

Input:

1st polynomial Expression = 5x^2 + 4x^1 + 2x^0

2nd polynomial Expression = 5x^1 + 5x^0

Output:

5x^2 + 9x^1 + 7x^0

Consider the below Fig 1.23, which adds the two polynomials. The two polynomials are represented in linked list. The elements which having the same power those corresponding coefficients are added.

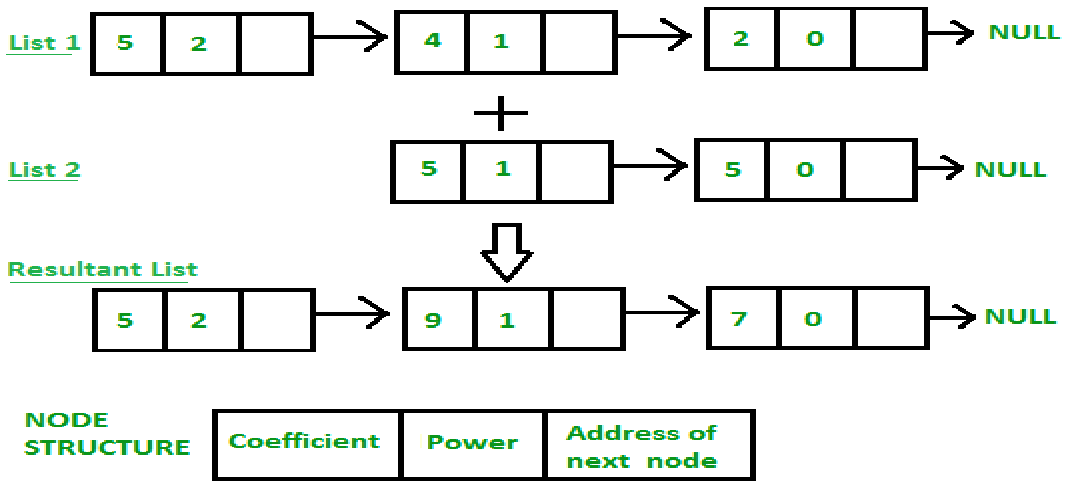


Figure 1.23 Addition of polynomials

Multiplication of Polynomials:

1. In this approach we will multiply the 2nd polynomial with each term of 1st polynomial.
2. Store the multiplied value in a new linked list.
3. Then we will add the coefficients of elements having the same power in resultant polynomial.

Example:

Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8

Output: 18x^3 + 54x^2 + 76x^1 + 48

On multiplying each element of 1st polynomial with elements of 2nd polynomial, we get

18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48

On adding values with same power of x,

18x^3 + 54x^2 + 76x^1 + 48

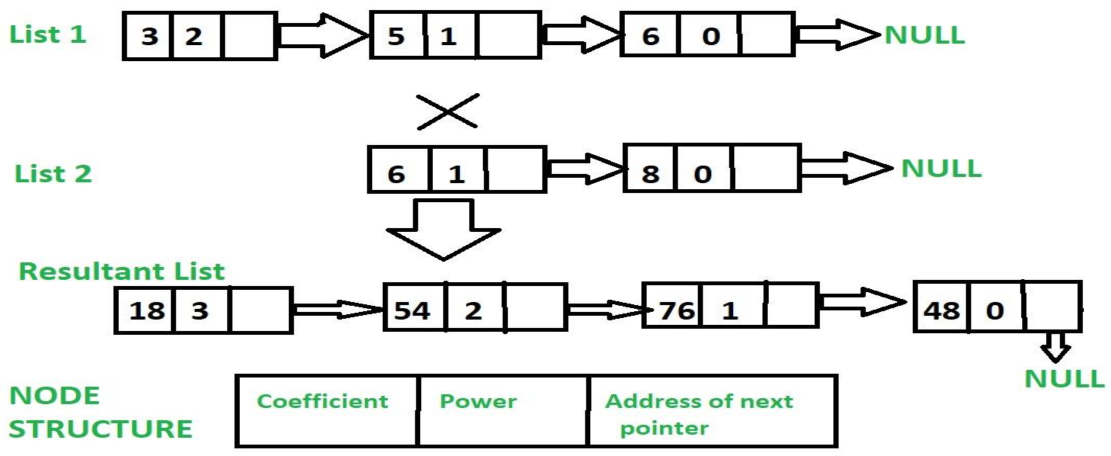


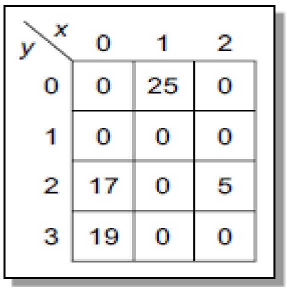
Figure 1.24 Multiplication of polynomials

1.6.3 Sparse Matrix Representation using Linked Lists:

Sparse Matrix:

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

The sparse matrix shown in Fig. 1.25 can be represented using a linked list for every row and column. Since a value is in exactly one row and one column, it will appear in both lists exactly once. A node in the multi-linked will have four parts. First stores the data, second stores a pointer to the next node in the row, third stores a pointer to the next node in the column, and the fourth stores the coordinates or the row and column number in which the data appears in the matrix. However, as in case of doubly linked lists, we can also have a corresponding inverse pointer for every pointer in the multi-linked list representation of a sparse matrix.



y \ x	0	1	2
0	0	25	0
1	0	0	0
2	17	0	5
3	19	0	0

Figure 1.25 Sparse matrix

Note:

When a non-zero value in the sparse matrix is set to zero, the corresponding node in the multi-linked list must be deleted.

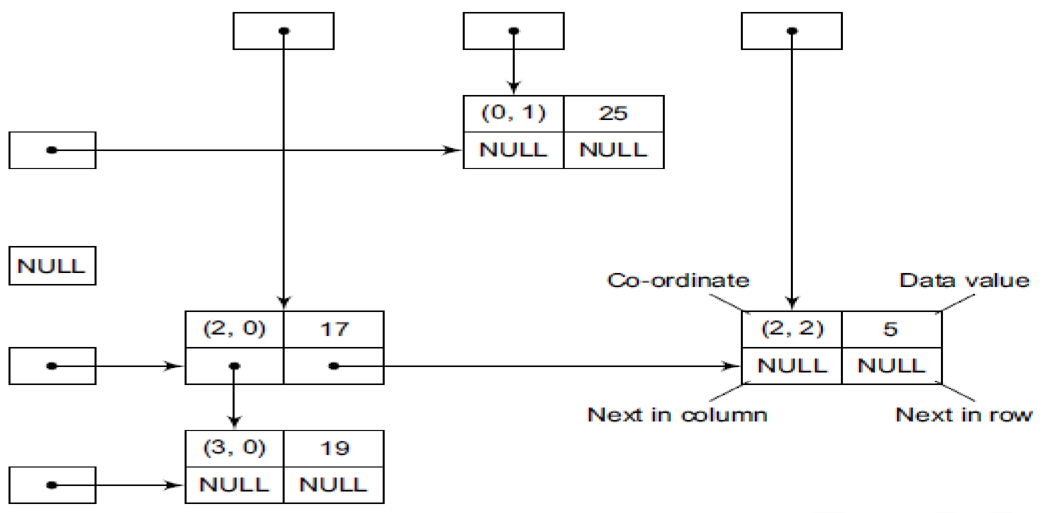


Figure 1.26 Multi-linked representation of sparse matrix shown in Fig. 1.25

1.7 ADVANTAGES AND DISADVANTAGES OF SINGLE LINKED LIST:

ADVANTAGES:

- Insertions and Deletions can be done easily.
- It does not need movement of elements for insertion and deletion.
- It space is not wasted as we can get space according to our requirements.
- Its size is not fixed.
- It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- It is less expensive.

DISADVANTAGES:

- It requires more space as pointers are also stored with information.
- Different amount of time is required to access each element.
- If we have to go to a particular element then we have to go through all those elements that come before that element.
- We cannot traverse it from last & only from the beginning.
- It is not easy to sort the elements stored in the linear linked list.

1.8 DOUBLE LINKED LIST:

A double 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 Fig. 1.27

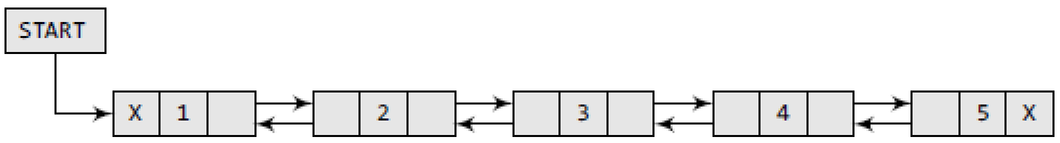


Figure 1.27 Double linked list

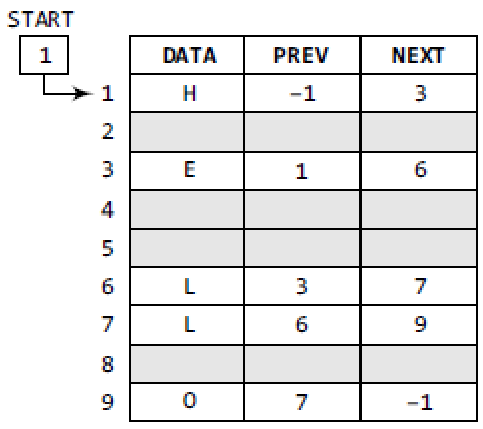
In C, the structure of a double linked list can be given as,

```
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.

Thus, we see that a double linked list calls for more space per node and more expensive basic operations. However, a double 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 double linked list is that it makes searching twice as efficient. Let us view how a double linked list is maintained in the memory.

Consider Fig. 1.28



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

Figure 1.28 Memory representation of a doubly linked list

In the figure, we see that a 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.

1.8.1 Inserting a New Node in a Double Linked List:

In this section, we will discuss how a new node is added into an already existing double linked list. We will take four cases and then see how insertion is done in each case.

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

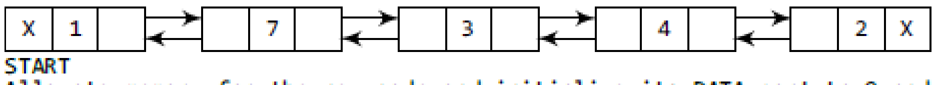
Case 2: The new node is inserted at the end.

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

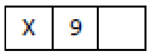
Case 4: The new node is inserted before a given node.

Inserting a Node at the Beginning of a Double Linked List:

Consider the double linked list shown in Fig. 1.29 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.



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.

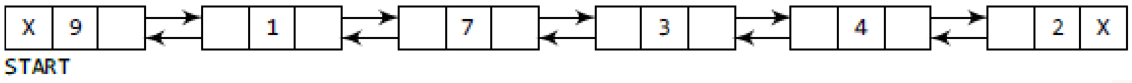


Figure 1.29 Inserting a new node at the beginning of a double linked list

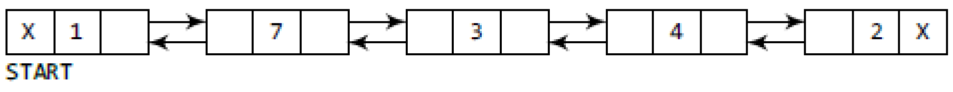
Figure 1.30 shows the algorithm to insert a new node at the beginning of a double linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space 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.

```
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
```

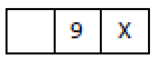
Figure 1.30 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Double Linked List:

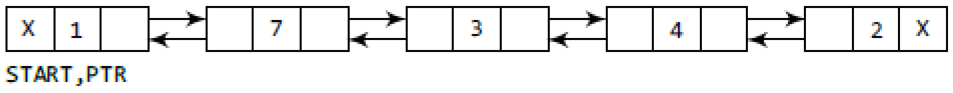
Consider the double linked list shown in Fig. 1.31. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



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.



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

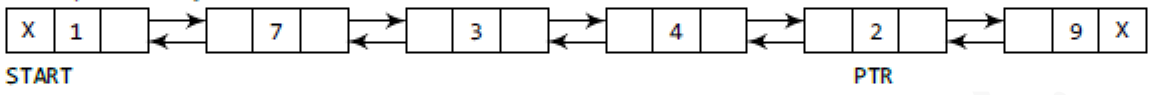


Figure 1.31 Inserting a new node at the end of a double linked list

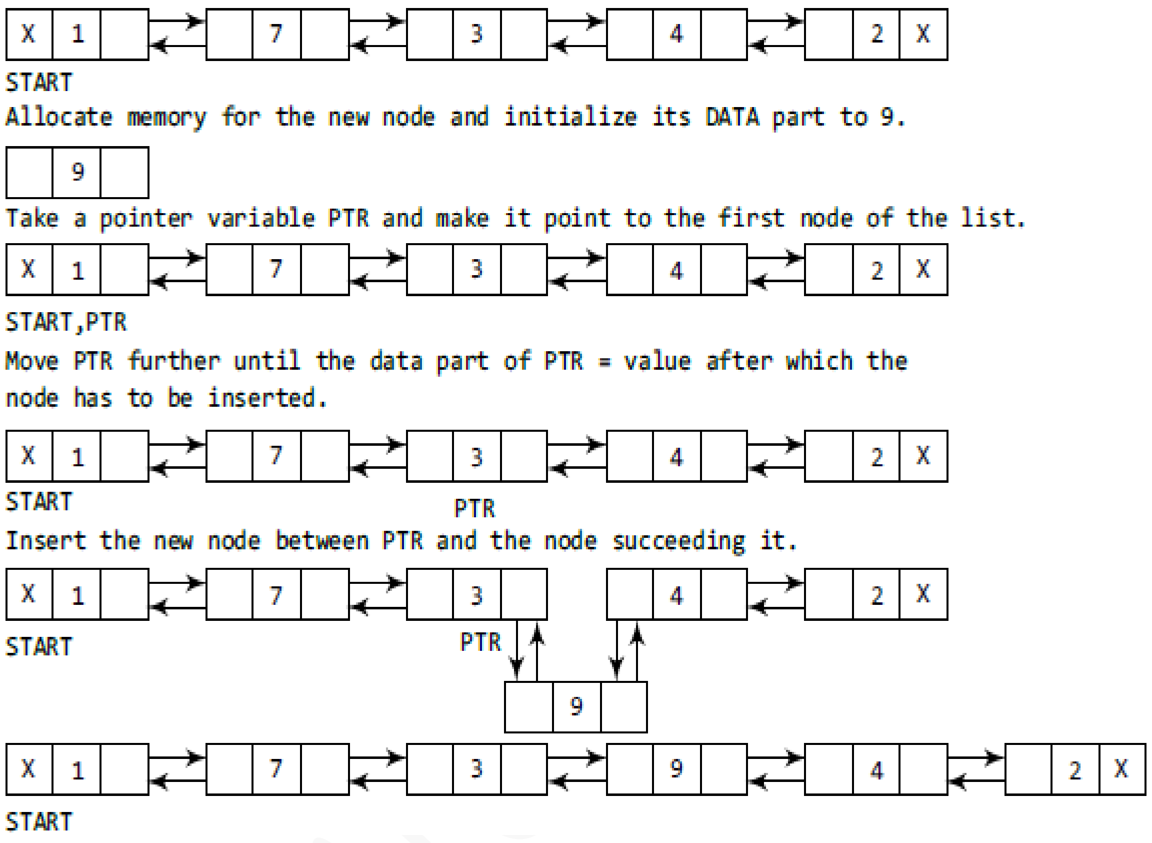
Figure 1.32 shows the algorithm to insert a new node at the end of a double linked list. In Step 6, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer 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).

```
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
```

Figure 1.32 Algorithm to insert a new node at the end

Inserting a Node After a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.33. Suppose we want to add a new node with value 9 after the node containing 3.


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.

Figure 1.33 Inserting a new node after a given node in a double linked list

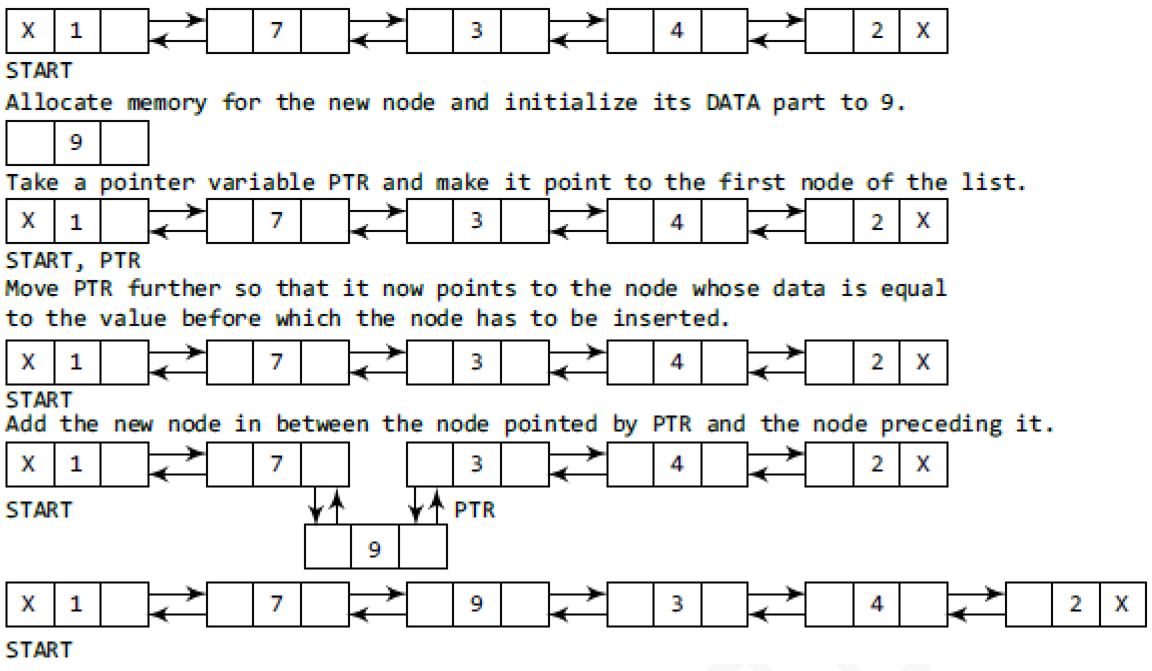
Figure 1.34 shows the algorithm to insert a new node after a given node in a double linked list. In Step 5, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has 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.

```
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
```

Figure 1.34 Algorithm to insert a new node after a given node

Inserting a Node Before a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.35. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig. 1.36 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.



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 so that it now points to the node whose data is equal to the value before which the node has to be inserted.

Add the new node in between the node pointed by PTR and the node preceding it.

Figure 1.35 Inserting a new node before a given node in a double linked list

```
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
```

Figure 1.36 Algorithm to insert a new node before a given node

1.8.2 Deleting a Node from a Double Linked List:

In this section, we will see how a node is deleted from an already existing double linked list. We will take four cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

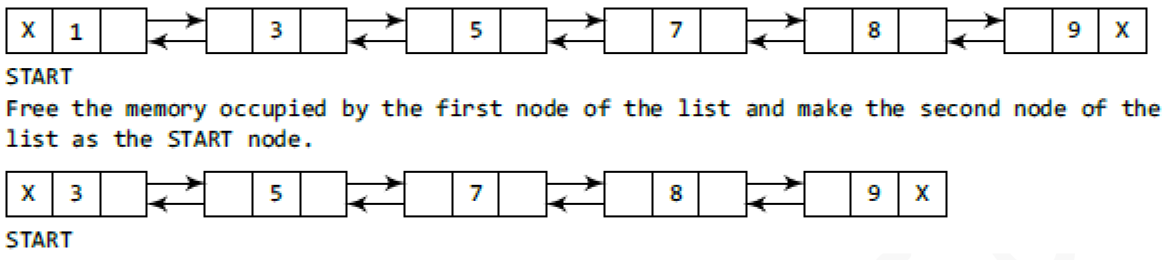
Case 2: The last node is deleted.

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

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

Deleting the First Node from a Double Linked List:

Consider the double linked list shown in Fig. 1.37. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



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

Figure 1.37 Deleting the first node from a double linked list

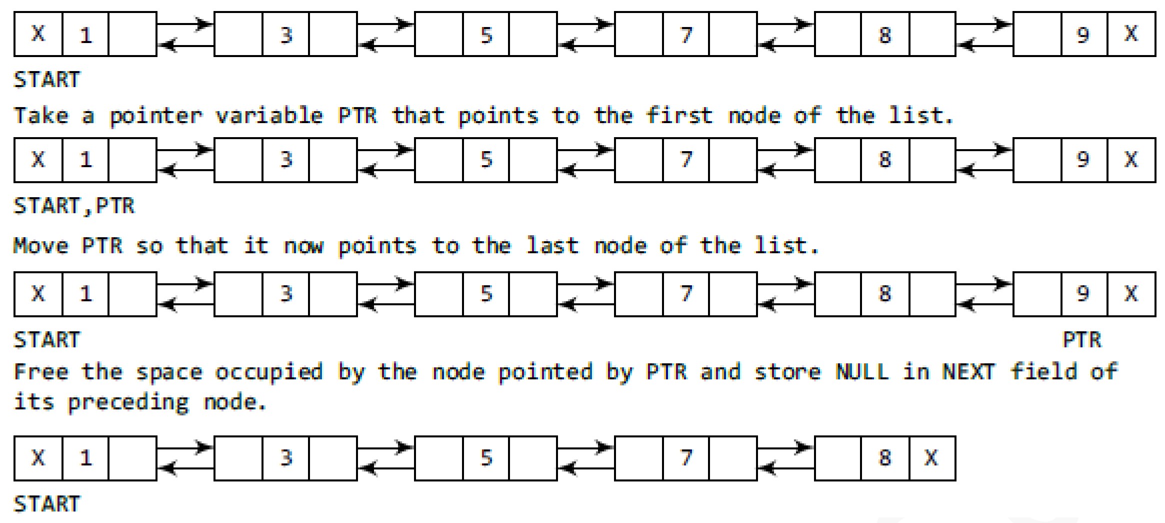
Figure 1.38 shows the algorithm to delete the first node of a double linked list. In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, 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.

```
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
```

Figure 1.38 Algorithm to delete the first node

Deleting the Last Node from a Double Linked List:

Consider the double linked list shown in Fig. 1.39. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



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

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

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

Figure 1.39 Deleting the last node from a double linked list

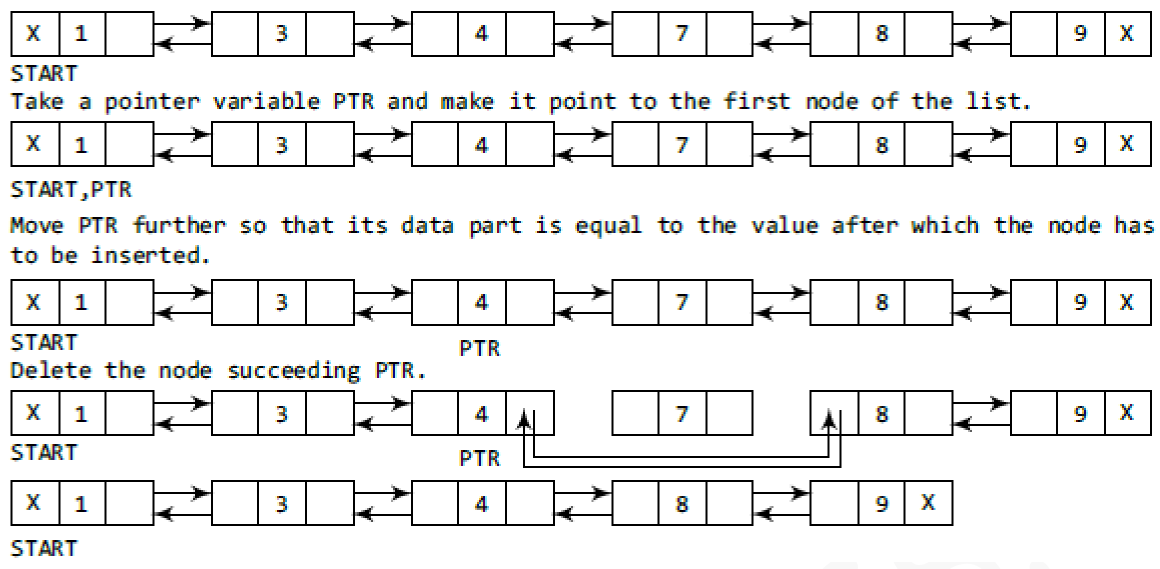
Figure 1.40 shows the algorithm to delete the last node of a double linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node. Once we reach the last 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.

```
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
```

Figure 1.40 Algorithm to delete the last node

Deleting the Node After a Given Node in a Double Linked List:

Consider the double linked list shown in Fig. 1.41. Suppose 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.



Take a 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.

Figure 1.41 Deleting the node after a given node in a double linked list

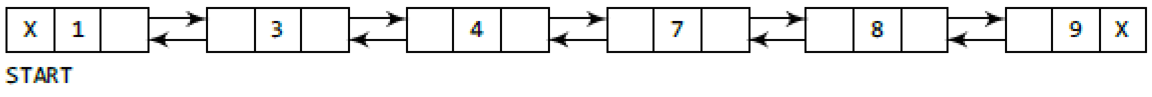
Figure 1.42 shows the algorithm to delete a node after a given node of a double linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given 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.

```
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
```

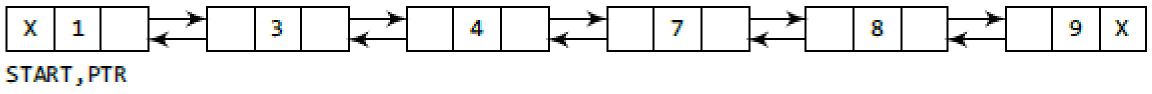
Figure 1.42 Algorithm to delete a node after a given node

Deleting the Node Before a Given Node in a Double Linked List:

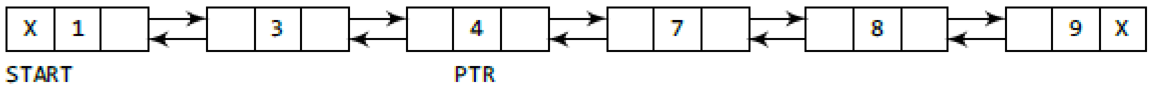
Consider the double linked list shown in Fig. 1.43. Suppose we want to delete the node preceding the node with value 4. Before discussing the changes that will be done in the linked list, let us first look at the algorithm.



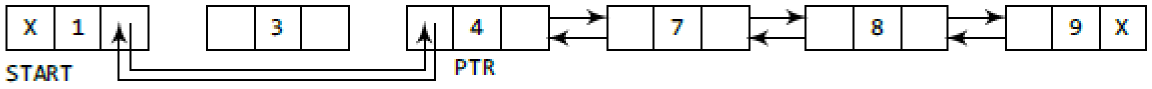
Take a pointer variable PTR that points to the first node of the list.



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.



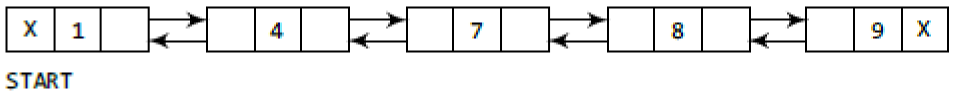


Figure 1.43 Deleting a node before a given node in a double linked list

Figure 1.44 shows the algorithm to delete a node before a given node of a double linked list. In Step2, 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. Hence, we see that we can insert or delete a node in a constant number of operations given only that node's address. Note that this is not possible in the case of a singly linked list which requires the previous node's address also to perform the same operation.

```
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
```

Figure 1.44 Algorithm to delete a node before a given node

1.9 CIRCULAR LINKED LIST:

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list.

1) Circular Single Linked List:

While traversing a circular Single linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending. Figure 1.45 shows a circular Single linked list.

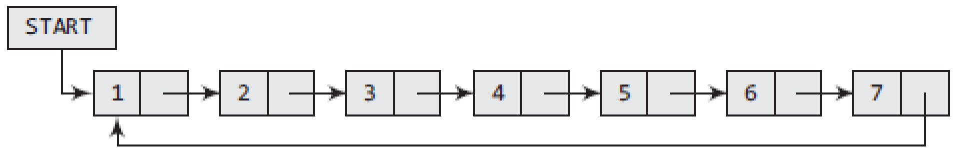
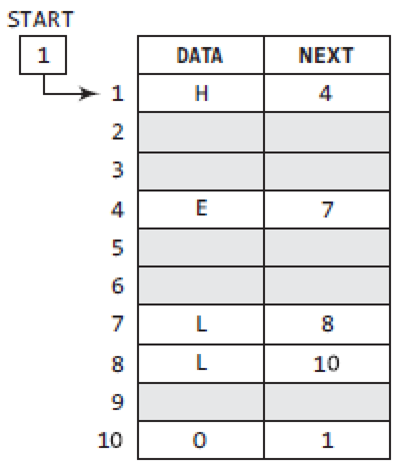


Figure 1.45 Circular Single linked list

The only downside of a circular linked list is the complexity of iteration. Note that there are no NULL values in the NEXT part of any of the nodes of list.

Consider Fig. 1.46.



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

Figure 1.46 Memory representation of a circular single linked list

We can traverse the list until we find the NEXT entry that contains the address of the first node of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list. When we traverse the DATA and NEXT in this manner, we will finally see that the linked list in Fig. 1.46 stores characters that when put together form the word HELLO.

1.9.1 Inserting a New Node in a Circular Linked List:

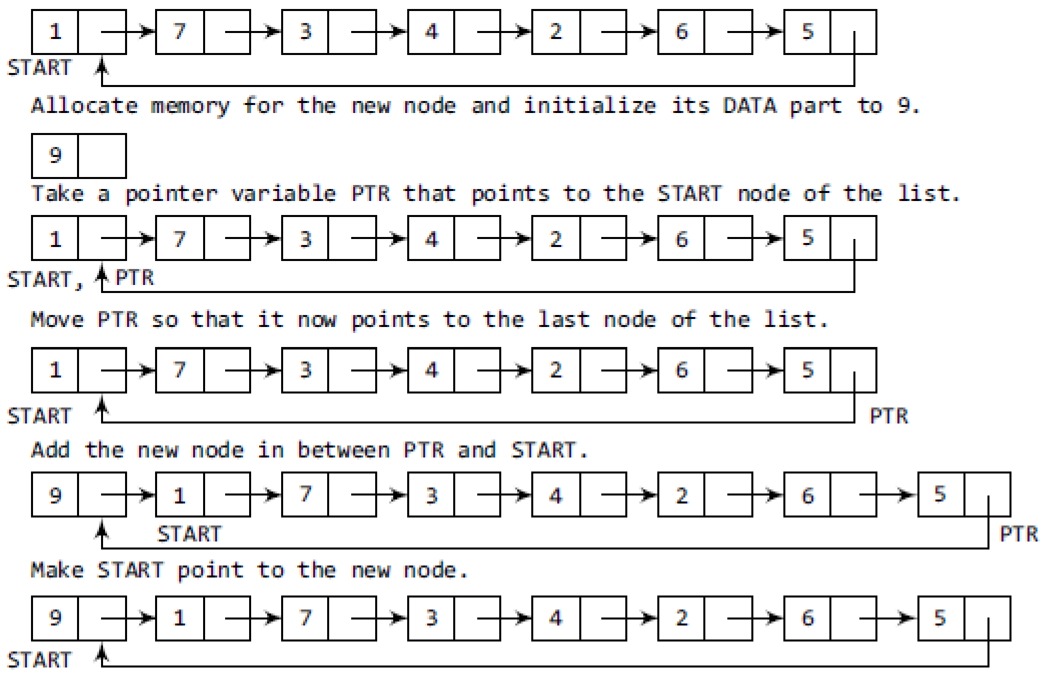
In this section, we will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.

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.

Inserting a Node at the Beginning of a Circular Linked List:

Consider the linked list shown in Fig. 1.47. 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.



1 7 3 4 2 6 5
START
Allocate memory for the new node and initialize its DATA part to 9.
9
Take a pointer variable PTR that points to the START node of the list.
1 7 3 4 2 6 5
START, PTR
Move PTR so that it now points to the last node of the list.
1 7 3 4 2 6 5
START
PTR
Add the new node in between PTR and START.
9 1 7 3 4 2 6 5
START
PTR
Make START point to the new node.
9 1 7 3 4 2 6 5
START

Figure 1.47 Inserting a new node at the beginning of a circular linked list

Figure 1.48 shows the algorithm to insert a new node at the beginning of a linked list. 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 the NEW_NODE.

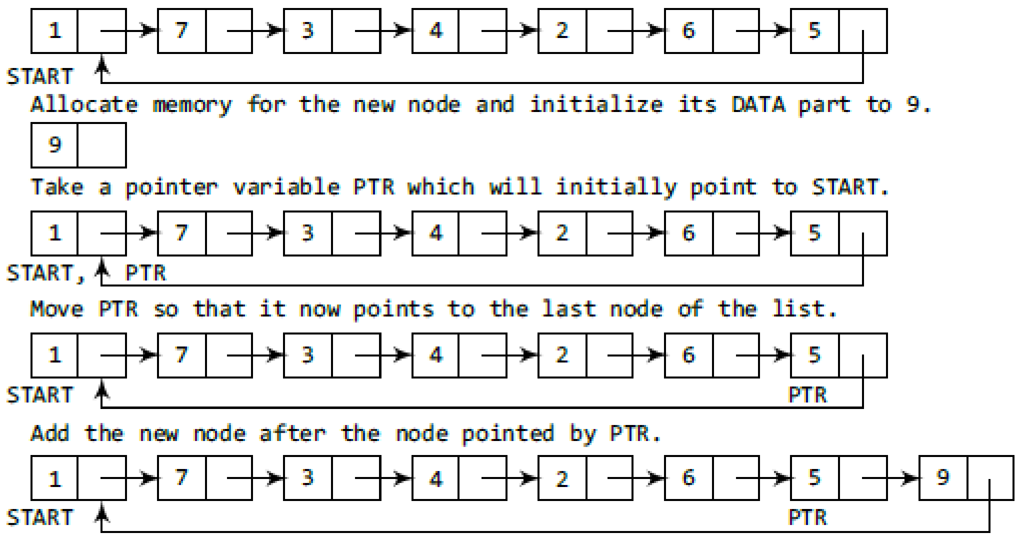
While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

```
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
```

Figure 1.48 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Linked List:

Consider the linked list shown in Fig. 1.49. Suppose 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.



1 7 3 4 2 6 5
START
Allocate memory for the new node and initialize its DATA part to 9.
9
Take a pointer variable PTR which will initially point to START.
1 7 3 4 2 6 5
START, PTR
Move PTR so that it now points to the last node of the list.
1 7 3 4 2 6 5
START
PTR
Add the new node after the node pointed by PTR.
1 7 3 4 2 6 5 9
START
PTR

Figure 1.49 Inserting a new node at the end of a circular linked list

Figure 1.50. shows the algorithm to insert a new node at the end of a circular linked list. 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 the address of the first node which is denoted by START.

```
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
```

Figure 1.50 Algorithm to insert a new node at the end

1.9.2 Deleting a Node from a Circular Linked List

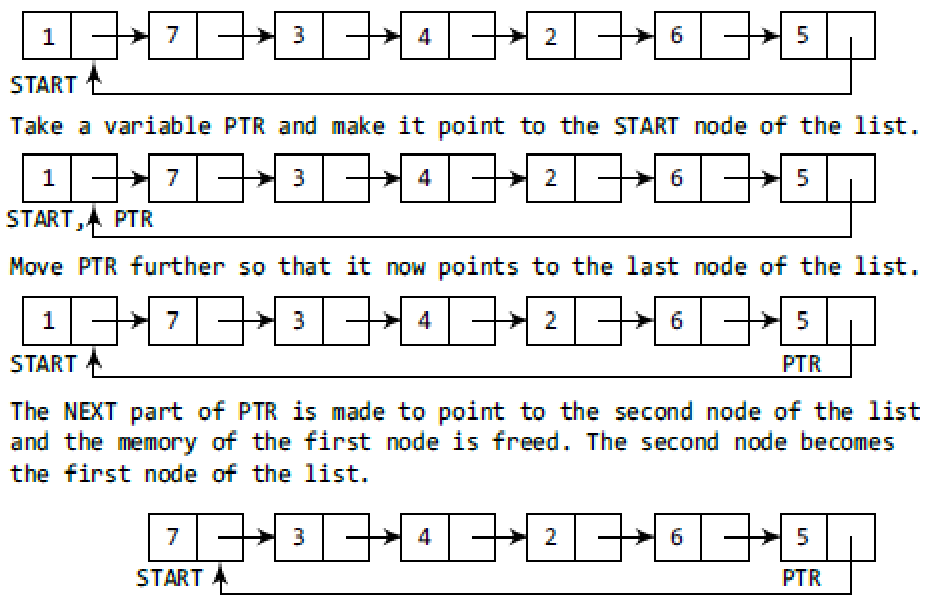
In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion are same as that given for singly linked lists.

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Deleting the First Node from a Circular Linked List:

Consider the circular linked list shown in Fig. 1.51. When we want to delete a node from the beginning of the list, then the following changes will be done in the 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.

Figure 1.51 Deleting the first node from a circular linked list

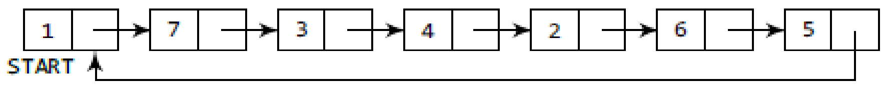
Figure 1.52 shows the algorithm to delete the first node from a circular linked list. In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, 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.

```
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
```

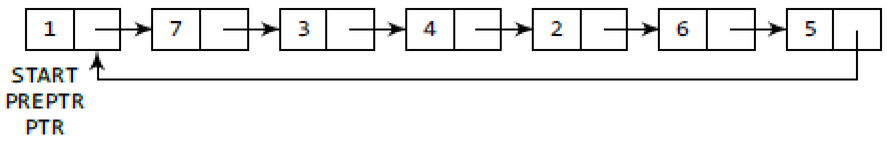
Figure 1.52 Algorithm to delete the first node

Deleting the Last Node from a Circular Linked List:

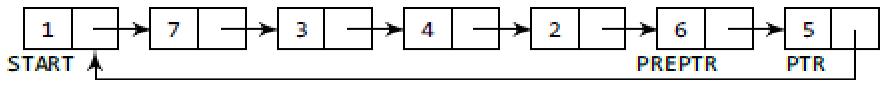
Consider the circular linked list shown in Fig. 1.53. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



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.

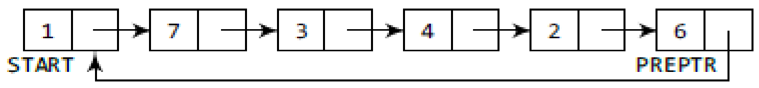


Figure 1.53 Deleting the last node from a circular linked list

Figure 1.54 shows the algorithm to delete the last node from a circular 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 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.

```
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
```

Figure 1.54 Algorithm to delete the last node

2) Circular Doubly Linked List:

A circular doubly linked list or a circular 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. The difference between a doubly linked and a circular doubly linked list is same as that exists between a singly linked list and a circular linked list. The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., START. Similarly, the previous field of the first field stores the address of the last node. A circular doubly linked list is shown in Fig. 1.55.

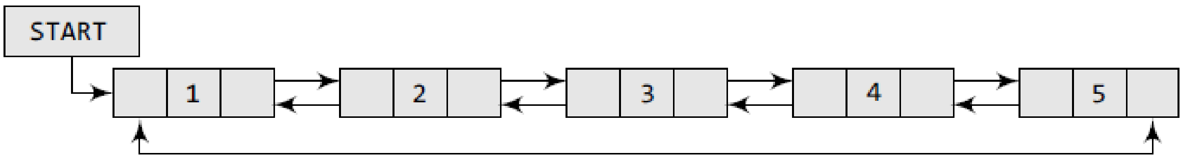
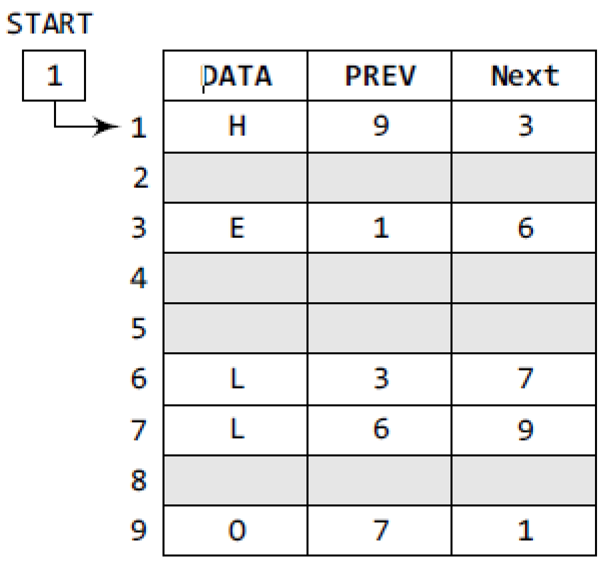


Figure 1.55 Circular doubly linked list

Since a circular doubly linked list contains three parts in its structure, it calls for more space per node and more expensive basic operations. However, a circular 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 circular doubly linked list is that it makes search operation twice as efficient.

Let us view how a circular doubly linked list is maintained in the memory. Consider Fig. 1.56. In the figure, we see that a variable START is used to store the address of the first node. Here in this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it stores the address of the last node of the list in its previous field. The corresponding NEXT stores the address of the next node, which is 3. So, we will look at address 3 to fetch the next data item. The previous field will contain the address of the first node. The second data element obtained from address 3 is E. We repeat this procedure until we reach a position where the NEXT entry stores the address of the first element of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.



START 1

	DATA	PREV	Next
1	H	9	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	0	7	1

Figure 1.56 Memory representation of a circular doubly linked list

1.9.3 Inserting a New Node in a Circular Doubly Linked List:

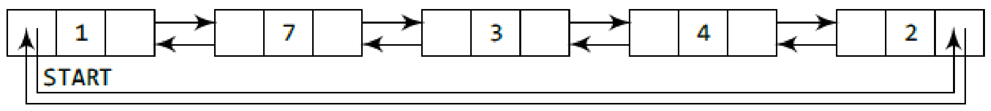
In this section, we will see how a new node is added into an already existing circular doubly linked list. We will take two cases and then see how insertion is done in each case. Rest of the cases are similar to that given for doubly linked lists.

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

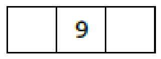
Case 2: The new node is inserted at the end.

Inserting a Node at the Beginning of a Circular Doubly Linked List:

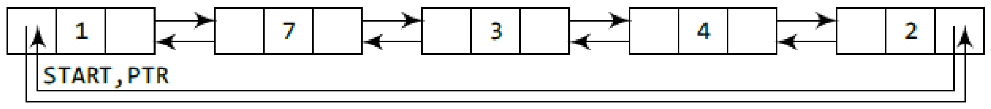
Consider the circular doubly linked list shown in Fig. 1.57. Suppose we want to add a new node with data 9 as the first node of the list. Then, the following changes will be done in the linked list.



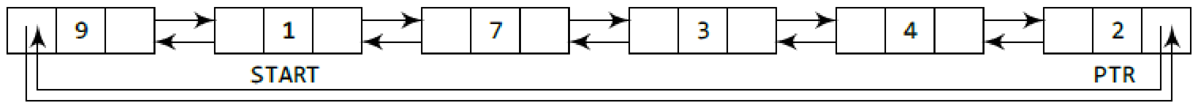
Allocate memory for the new node and initialize its DATA part to 9.



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



Move PTR so that it now points to the last node of the list. Insert the new node in between PTR and the START node.



START will now point to the new node.

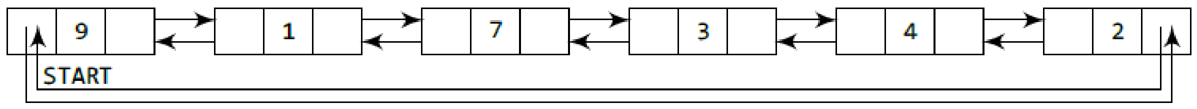


Figure 1.57 Inserting a new node at the beginning of a circular doubly linked list

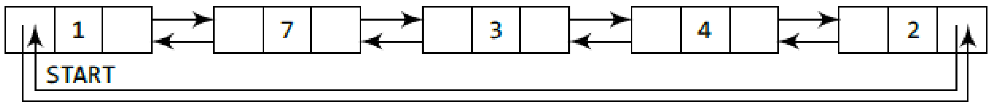
Figure 1.58 shows the algorithm to insert a new node at the beginning of a circular doubly linked list. In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, we allocate space for the new node. Set its data part with the given VAL and its 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. Since it is a circular doubly linked list, the PREV field of the NEW_NODE is set to contain the address of the last node.

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 13
         [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:         SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```

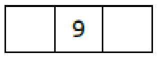
Figure 1.58 Algorithm to insert a new node at the beginning

Inserting a Node at the End of a Circular Doubly Linked List:

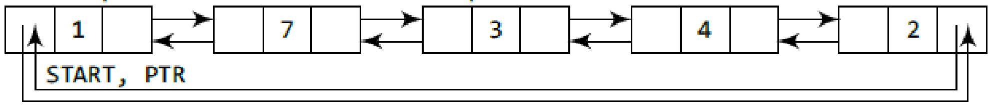
Consider the circular doubly linked list shown in Fig. 1.59. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



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



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



Move PTR to point to the last node of the list so that the new node can be inserted after it.

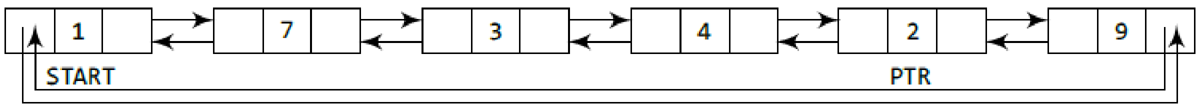


Figure 1.59 Inserting a new node at the end of a circular doubly linked list

Figure 1.60 shows the algorithm to insert a new node at the end of a circular doubly linked list. 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. 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).

```
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 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: SET NEW_NODE -> PREV = PTR
Step 11: SET START -> PREV = NEW_NODE
Step 12: EXIT
```

Figure 1.60 Algorithm to insert a new node at the end

1.9.4 Deleting a Node from a Circular Doubly Linked List:

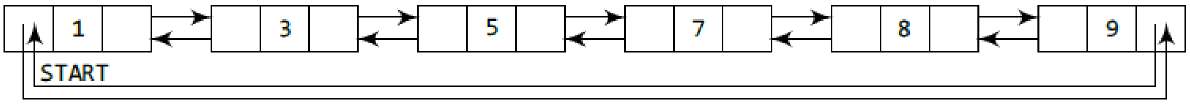
In this section, we will see how a node is deleted from an already existing circular doubly linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases are same as that given for doubly linked lists.

Case 1: The first node is deleted.

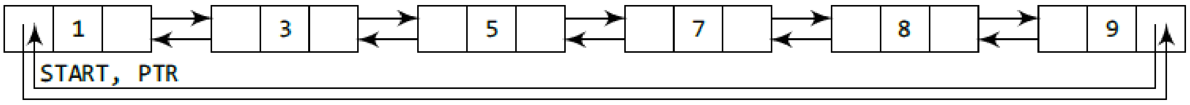
Case 2: The last node is deleted.

Deleting the First Node from a Circular Doubly Linked List:

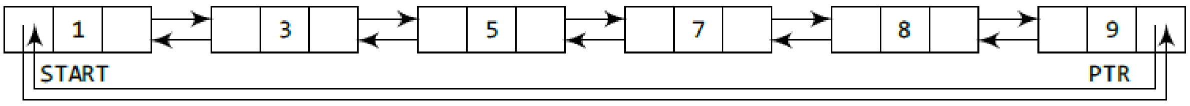
Consider the circular doubly linked list shown in Fig. 1.61. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



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



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



Make START point to the second node of the list. Free the space occupied by the first node.

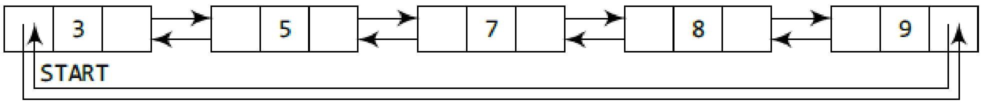


Figure 1.61 Deleting the first node from a circular doubly linked list

Figure 1.62 shows the algorithm to delete the first node from a circular doubly linked list. 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.

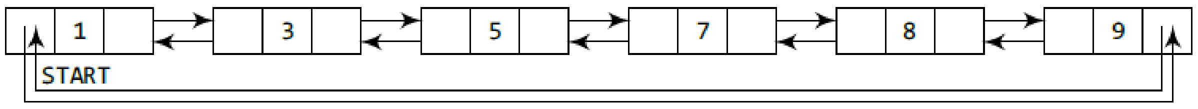
However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. The while loop traverses through the list to reach the last node. Once we reach the last node, the NEXT pointer of PTR is set to contain the address of the node that succeeds START. Finally, START is made to point to the next node in the sequence and the memory occupied by the first node of the list is freed and returned to the free pool.

```
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: SET START -> NEXT -> PREV = PTR
Step 7: FREE START
Step 8: SET START = PTR -> NEXT
```

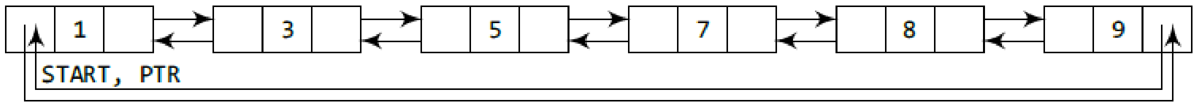
Figure 1.62 Algorithm to delete the first node

Deleting the Last Node from a Circular Doubly Linked List:

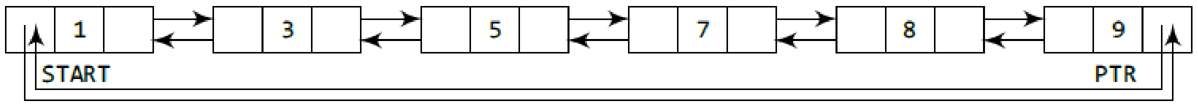
Consider the circular doubly linked list shown in Fig. 1.63. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



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



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



Free the space occupied by PTR.

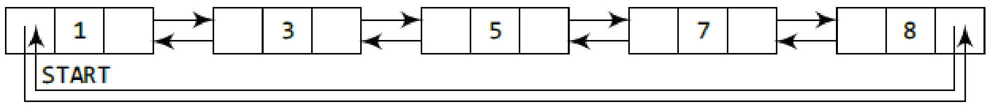


Figure 1.63 Deleting the last node from a circular doubly linked list

Figure 1.64 shows the algorithm to delete the last node from a circular doubly 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. 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 the second last node to contain the address of 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.

```
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 -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```

Figure 1.64 Algorithm to delete the last node