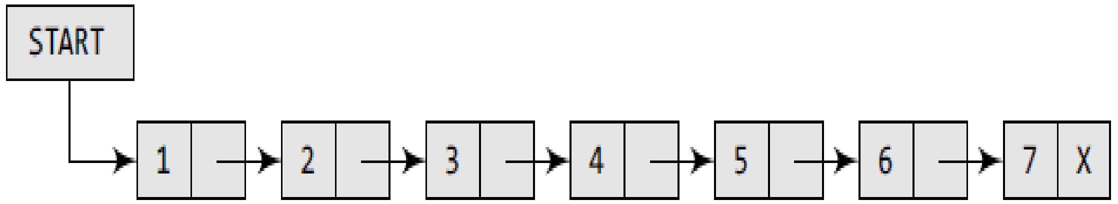
Linked List

Prof. Sushma Kadge
sushmakadge@somaiya.edu

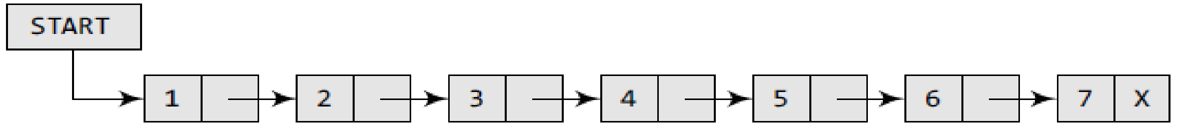
Linked List

- 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.
- 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.

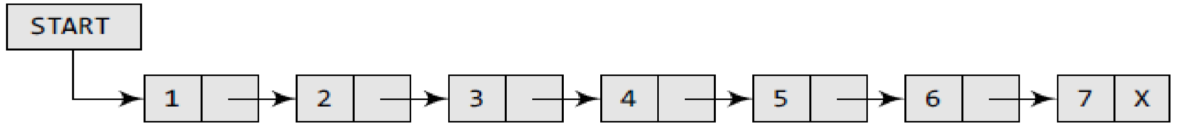


Linked List



- 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.
- The NULL pointer is represented by X. Define NULL as –1.
- Every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.
- Pointer variable START that stores the address of the first node in the list.

Linked List



```
struct node
{
int data;//information field
struct node *next;  //Pointer that points to the structure itself, thus Linked
};
```

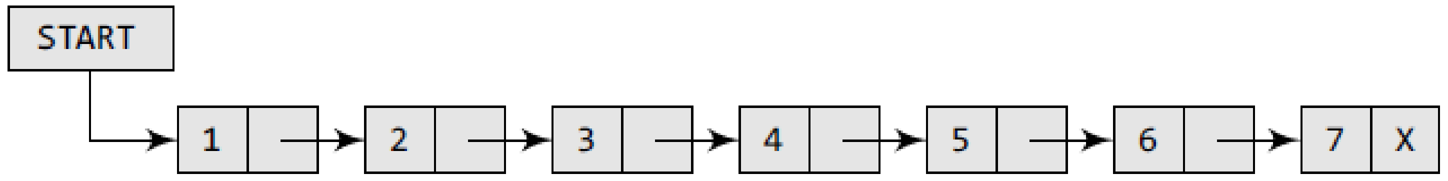
- Linked lists provide an efficient way of storing related data and perform basic operations such as insertion, deletion, and updation of information at the cost of extra space required for storing address of the next node.

Types of Linked List

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

Singly Linked list

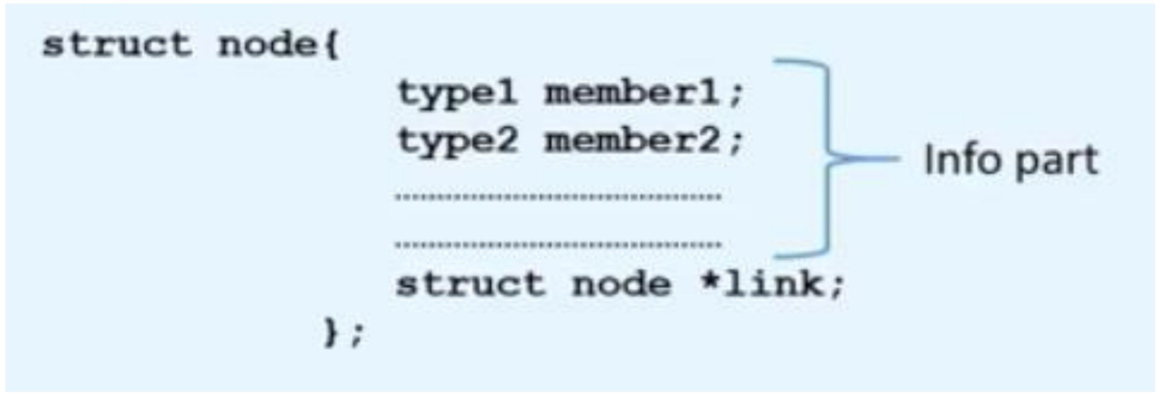
- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- All nodes are linked in sequential manner, Linear Linked List, One way chain, It has beginning and end.
- A singly linked list allows traversal of data only in one way.



Linked List Operations

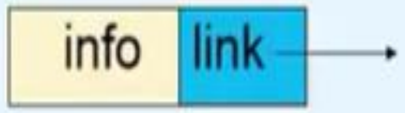
- Creation
- Insertion
- Deletion
- Traversal
- Searching

Creation of a new node



```
struct node{
              type1 member1;
              type2 member2;
              ......................................
              ......................................
              struct node *link;
           };
```

```
struct node
{
    int info;
    struct node *link;
};
```



```
struct node *start = NULL;
```

Creation of a new node

New node=temp

```
struct node *tmp;
tmp= (struct node *) malloc(sizeof(struct node));
tmp->info=data;
tmp->link=NULL;
```

Syntax-

tmp=(type_cast*)malloc(size)

tmp = name of pointer that holds the starting address of allocated memory block

type_cast* = is the data type into which the returned pointer is to be converted and Size = size of allocated memory block in bytes

Creating a Linked List

```
create_list(int data)
{
        struct node *q,*tmp;
        tmp= (struct node *) malloc(sizeof(struct node));
        tmp->info=data;
        tmp->link=NULL;

        if(start==NULL) /*If list is empty */
        {
                start=tmp;
        }
        else
        {       /*Element inserted at the end */
                        }
}/*End of create_list()*/
```

Traversing a Linked List:

- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.

Algorithm for traversing a linked list

```
Step 1:
SET PTR = START// initialize PTR with the address of START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply Process to PTR ->DATA// we apply the process
(e.g., print) to the current node, i.e, the node pointed by PTR.
Step 4: SET PTR = PTR NEXT
[END OF LOOP]
Step 5: EXIT
```

Algorithm to print(count) 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->LINK
[End of Loop]
Step 6:Print COUNT
Step 7:EXIT
```

Searching for a Value in a Linked List

Searching a linked list means to find a particular element in the linked list.
A linked list consists of nodes which are divided into two parts,
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.

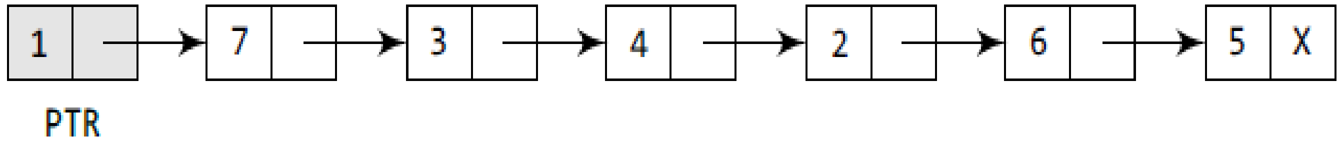
Searching for a Value in a Linked List

```
Step 1:[INITIALIZE] SET POSITION=1
Step 2:[INITIALIZE] SET PTR=START
Step 3: Repeat Steps 4 while PTR!=NULL
Step 4:                 If Val=PTR->Data
                            Print POSITION
                            Exit
                    [End of If]
                    Set PTR=PTR->LINK
                    Set POSITION=POSITION +1
        [End of Loop]
Step 5:If PTR=NULL
                    Print Search Unsuccessful
        [End of If]
Step 6: Exit
```

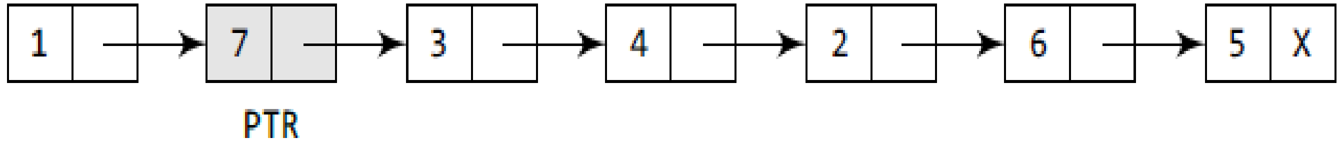
Searching for a Value in a Linked List

```
search(int data)
{
        struct node *ptr = start;
        int pos = 1;
        while(ptr!=NULL)
        {
                if(ptr->info==data)
                {
                        printf("Item %d found at position %d\n",data,pos);
                        return;
                }
                ptr = ptr->link;
                pos++;
        }
        if(ptr == NULL)
                printf("Item %d not found in list\n",data);
}/*End of search()*/
```

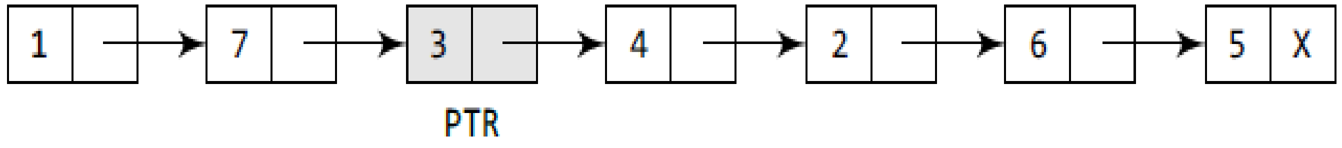
Searching for a Value in a Linked List



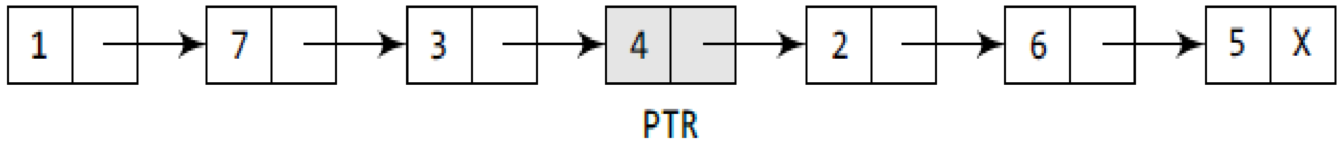
Here PTR -> DATA = 1. Since PTR -> DATA != 4, we move to the next node.



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.



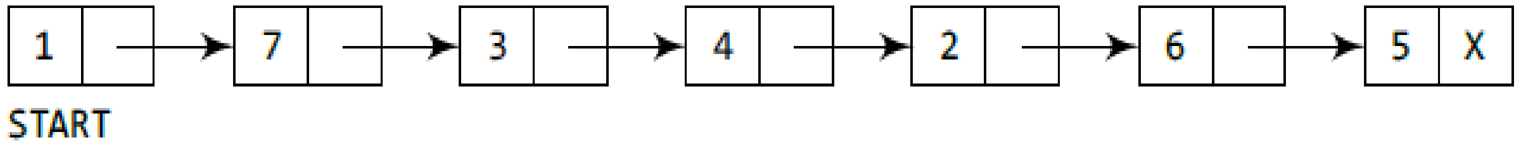
Here PTR -> DATA = 4. Since PTR -> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

Insertion into a Linked List

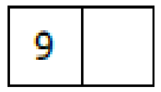
Insertion is possible in two ways:

- Insertion at Beginning
- Insertion in Between

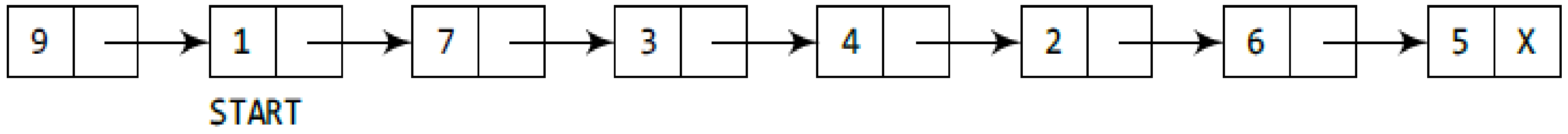
Case 1- Insertion at Beginning



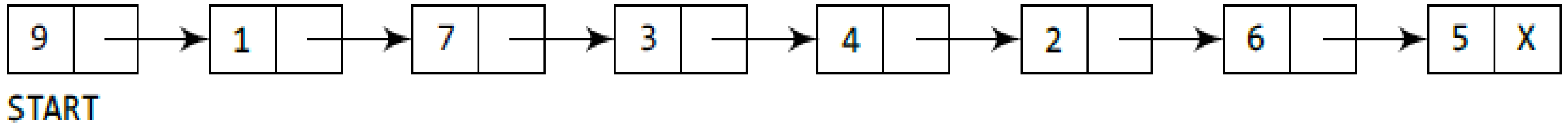
Allocate memory for the new node and initialize its DATA part to 9.



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.



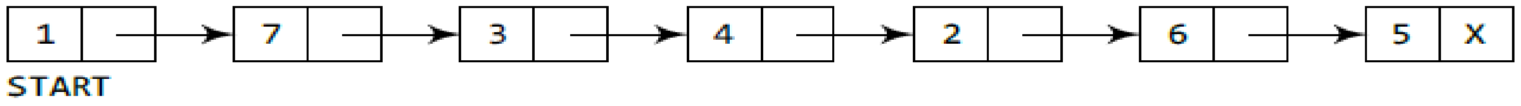
Case 1- Insertion at Beginning

- First check whether Memory is available for the new node.
- If the memory has exhausted then an Overflow message is printed
- Else We allocate memory for the new node

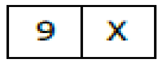
Case 1- Insertion at Beginning

```
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 NEWNODE -> DATA = VAL
Step 5: SET NEWNODE ->NEXT = START
Step 6: SET START = NEW NODE
Step 7: EXIT
```

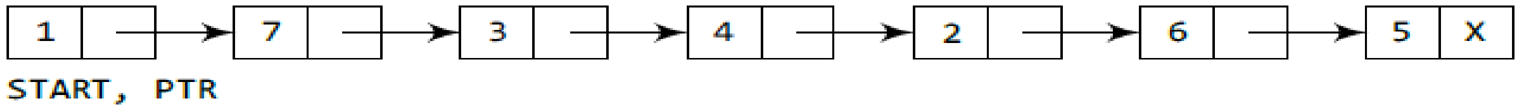
Case 2- Insertion at end



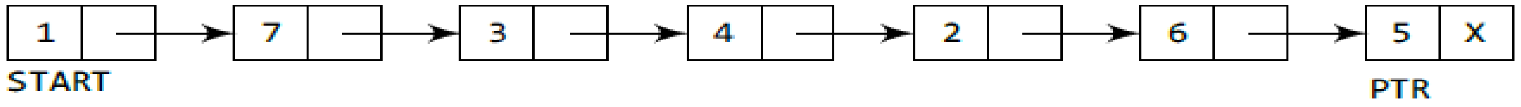
Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.



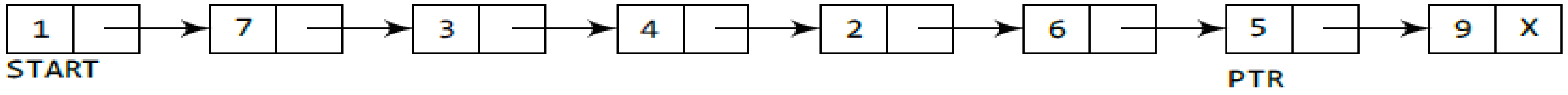
Take a pointer variable PTR which points to START.



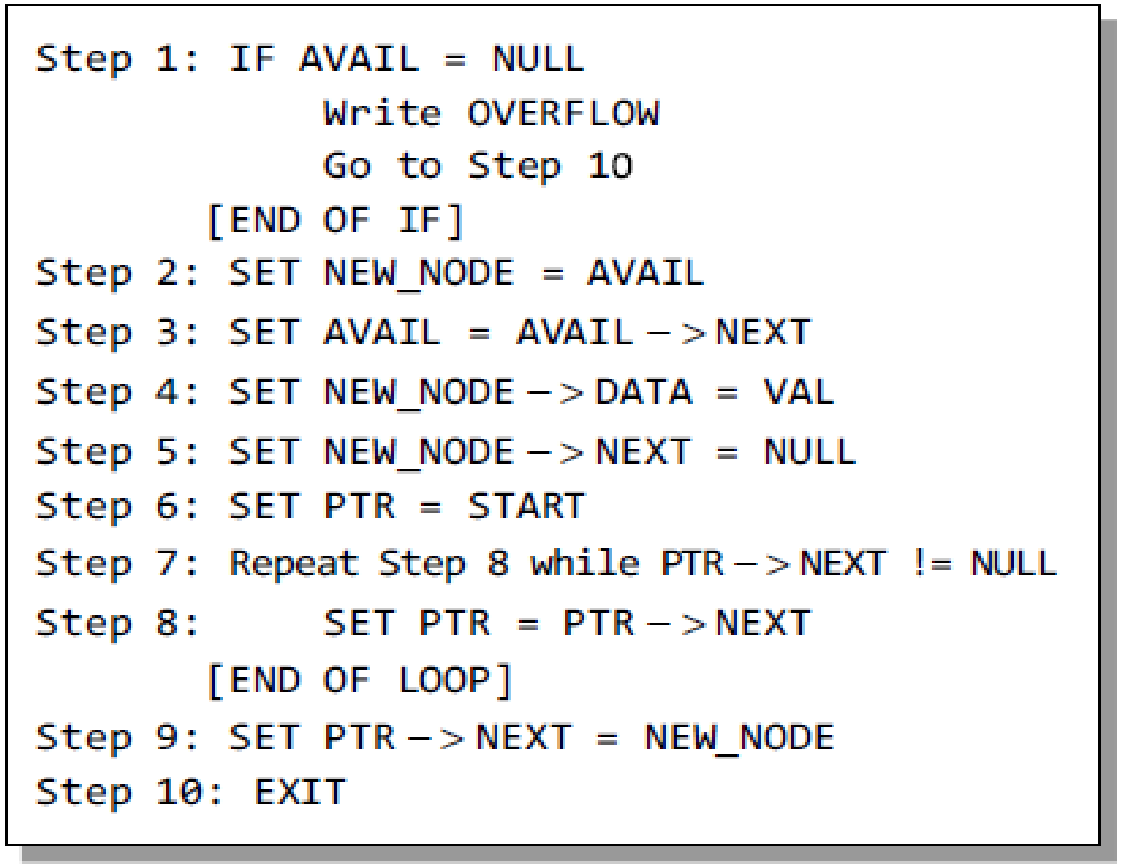
Move PTR so that it points to the last node of the list.



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.

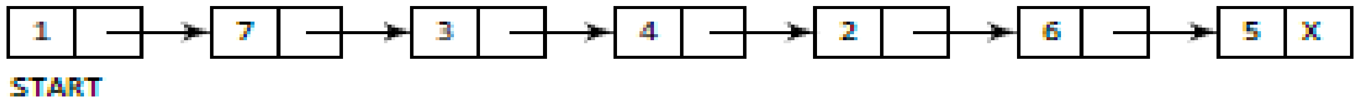


Case 2- Insertion at end

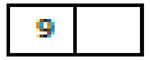


```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT !=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

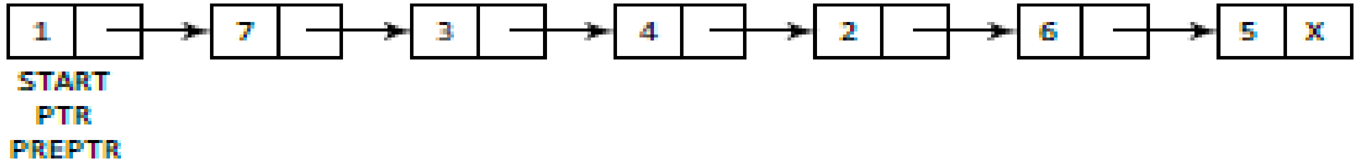
Inserting a Node After a Given Node in a Linked List



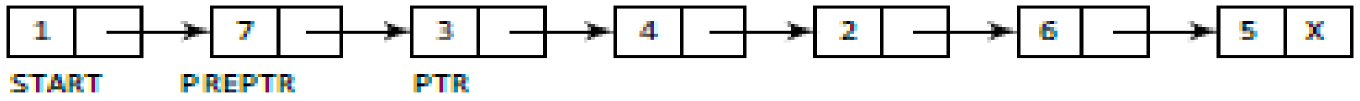
Allocate memory for the new node and initialize its DATA part to 9.

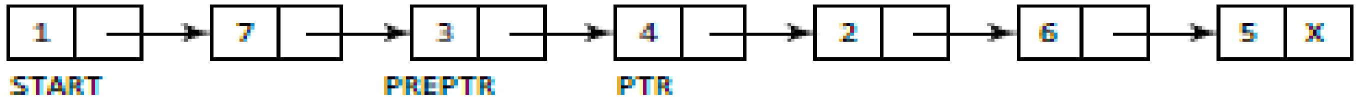


Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.

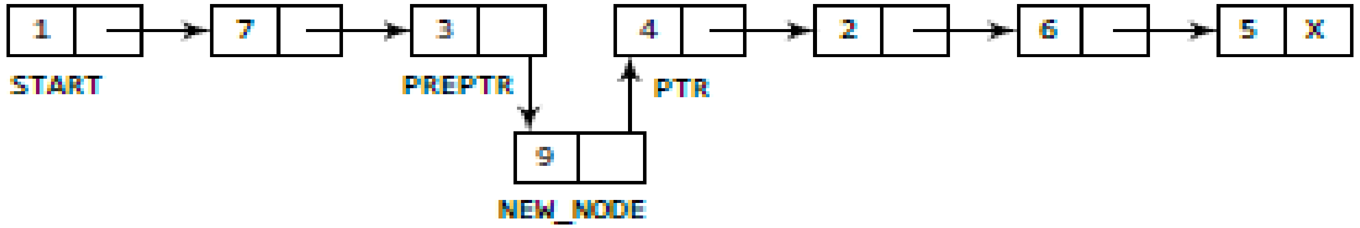


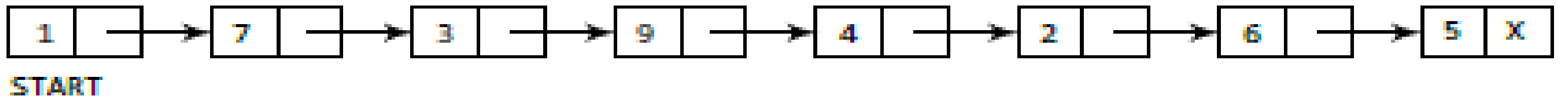
Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.



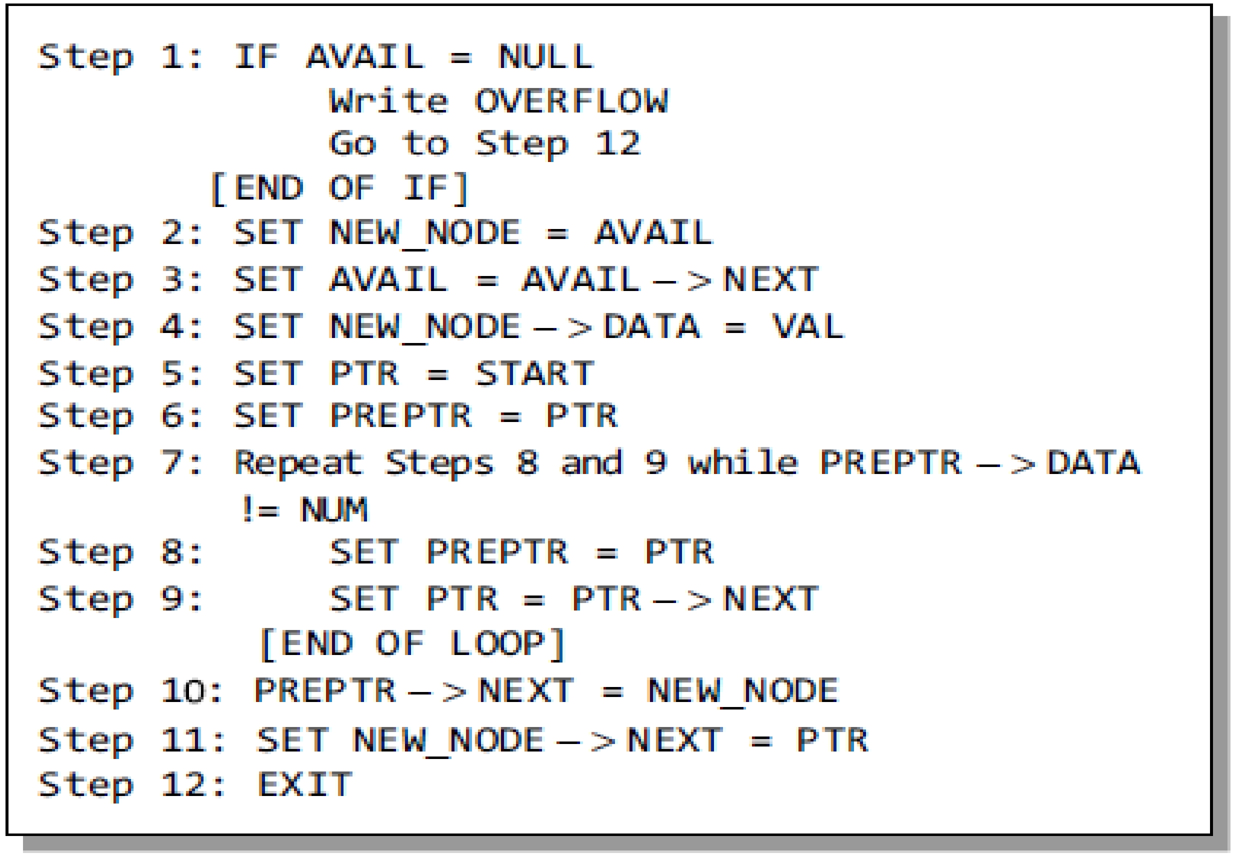


Add the new node in between the nodes pointed by PREPTR and PTR.





Inserting a Node After a Given Node in a Linked List



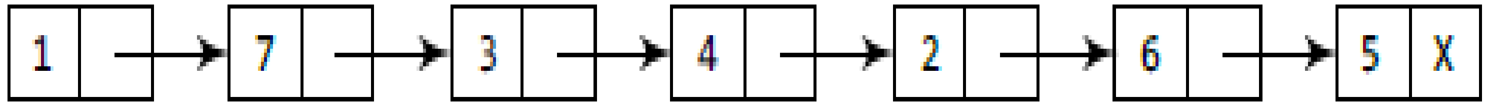
```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
   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

While loop: 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.

Deleting a Node from a Linked List

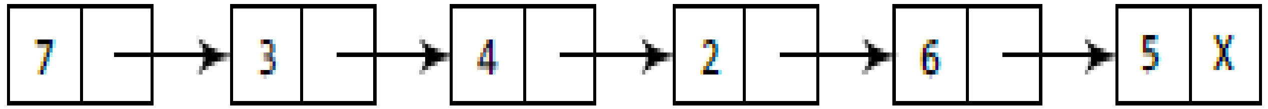
Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.

Deleting the First Node from a Linked List



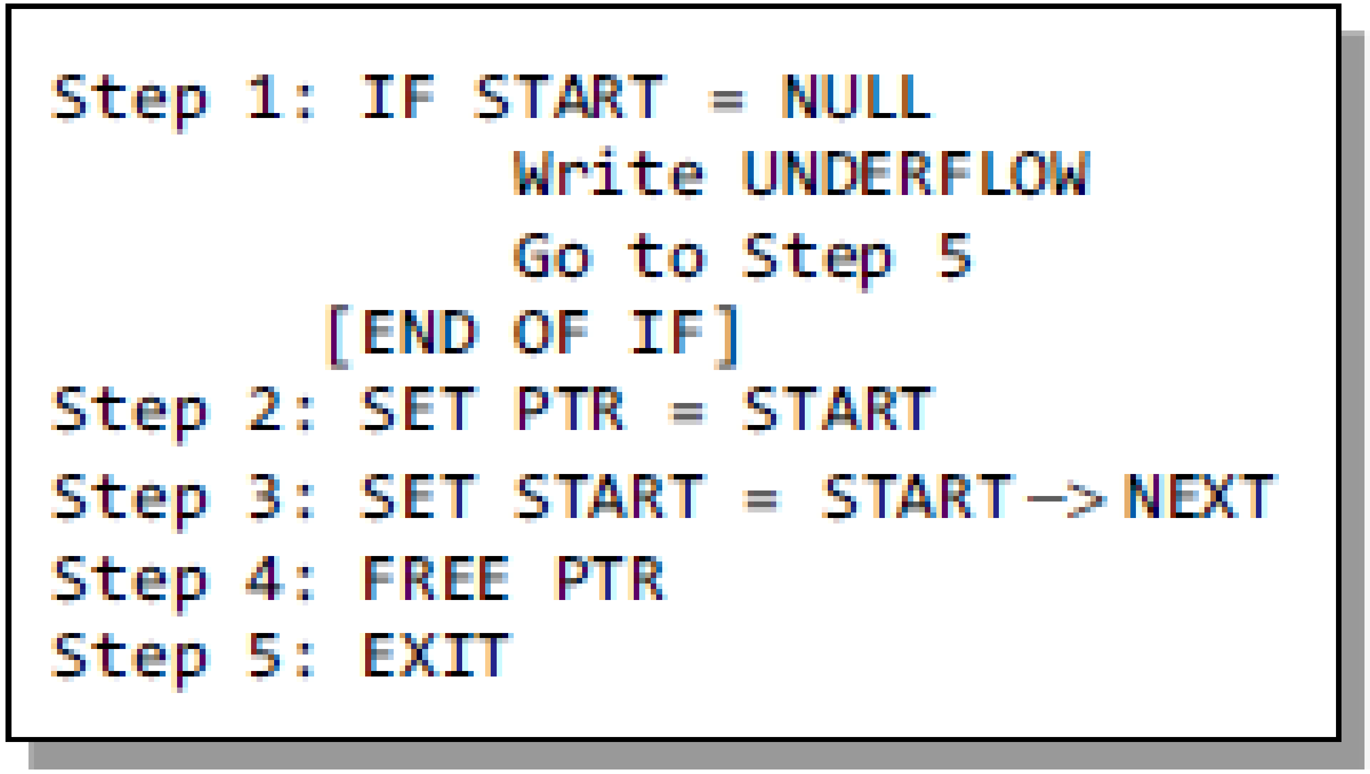
START

Make START to point to the next node in sequence.



START

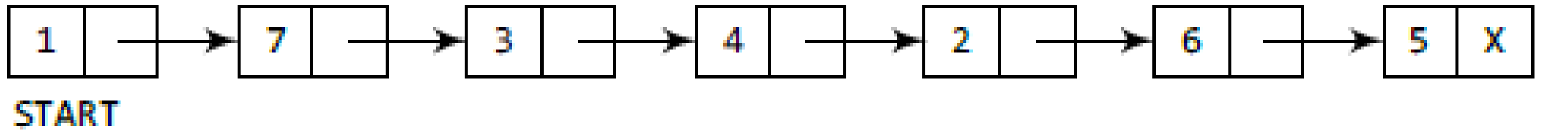
Deleting a Node from a Linked List



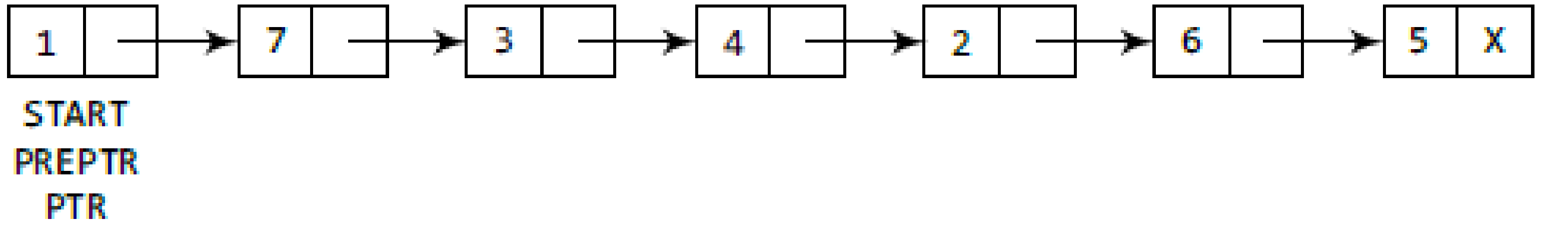
```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

Initialize PTR with START that stores the address of the first node of the list.
 Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR is freed and returned to the free pool.

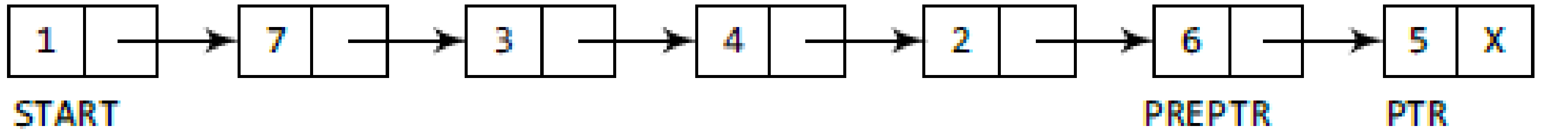
Deleting the Last Node from a Linked



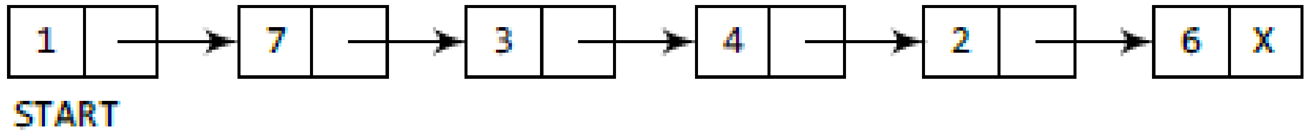
Take pointer variables PTR and PREPTR which initially point to START.



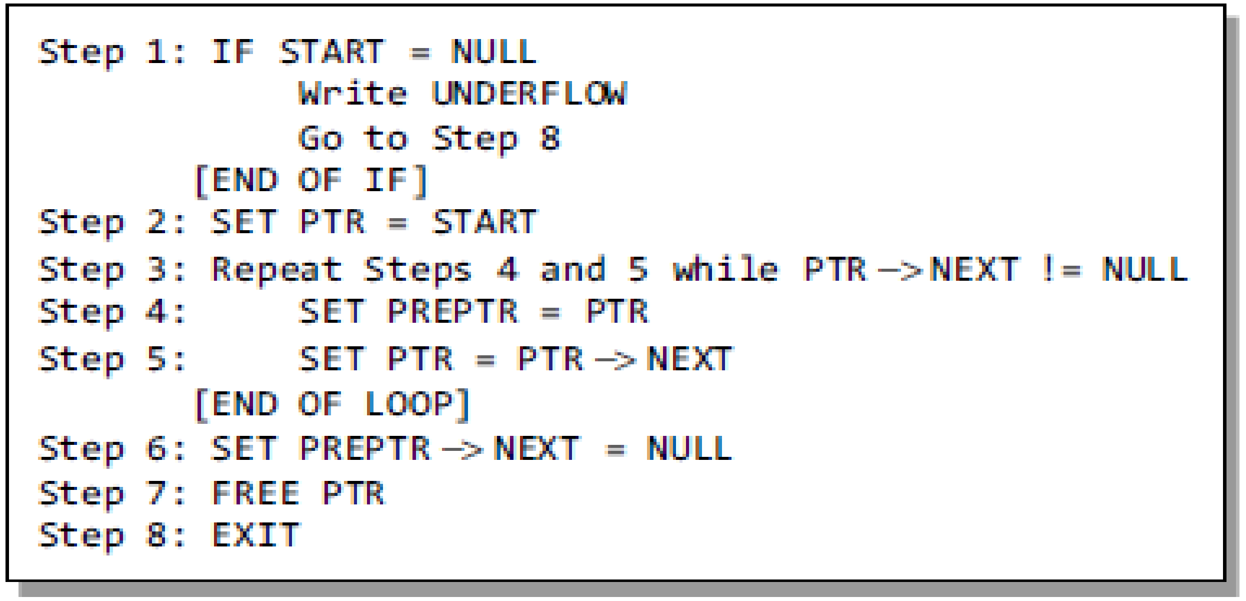
Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



Set the NEXT part of PREPTR node to NULL.



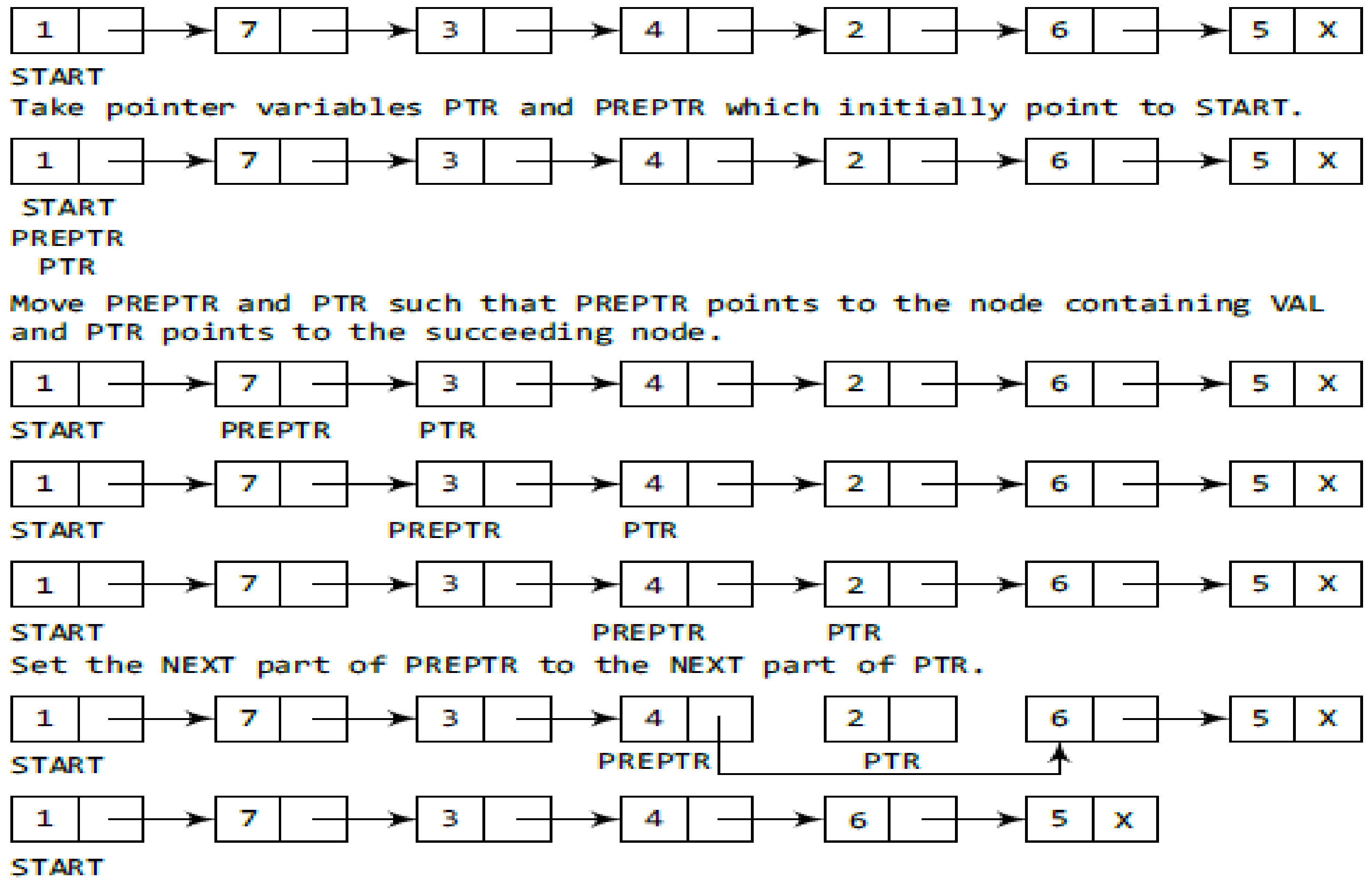
Deleting the Last Node from a Linked



```
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
```

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.

Deleting the Node after a Given Node in a Linked List

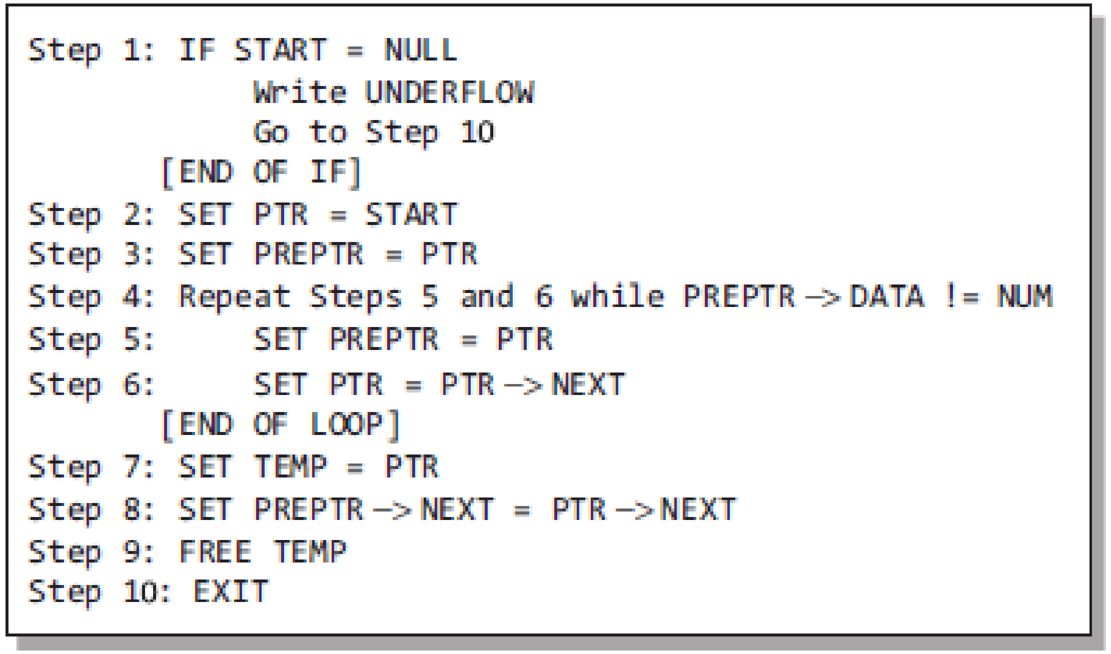


Take pointer variables PTR and PREPTR which initially point to START.

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.

Deleting the Node after a Given Node in a Linked List



```
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
```

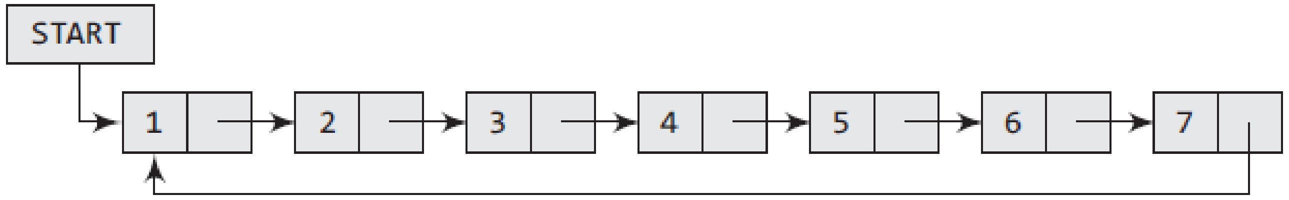
Step 2, we take a pointer variable PTR and initialize it with START.
 In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL 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

- In a singly linked list,
  <u>If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node.</u>
  This problem can be solved by slightly altering the structure of singly linked list.
- In a singly linked list, next part (pointer to next node) of the last node is NULL,
- If we utilize this link to point to the first node then we can reach preceding nodes.

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.
- While traversing a circular 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.

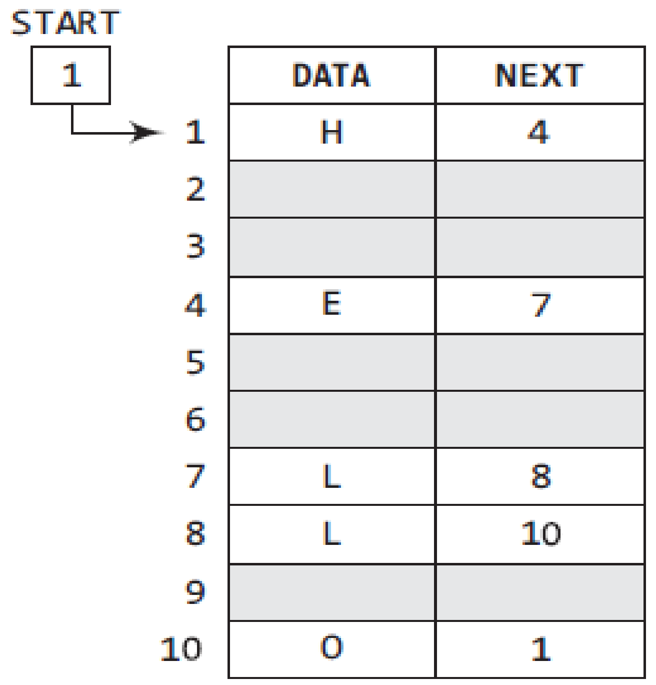


Circular Linked List

- Widely used in operating systems for task maintenance.
- Where a circular linked list is used?
- When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited.
- How is this done? The answer is simple.
- A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually, this is done using either the circular stack or the circular queue.

Circular 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.

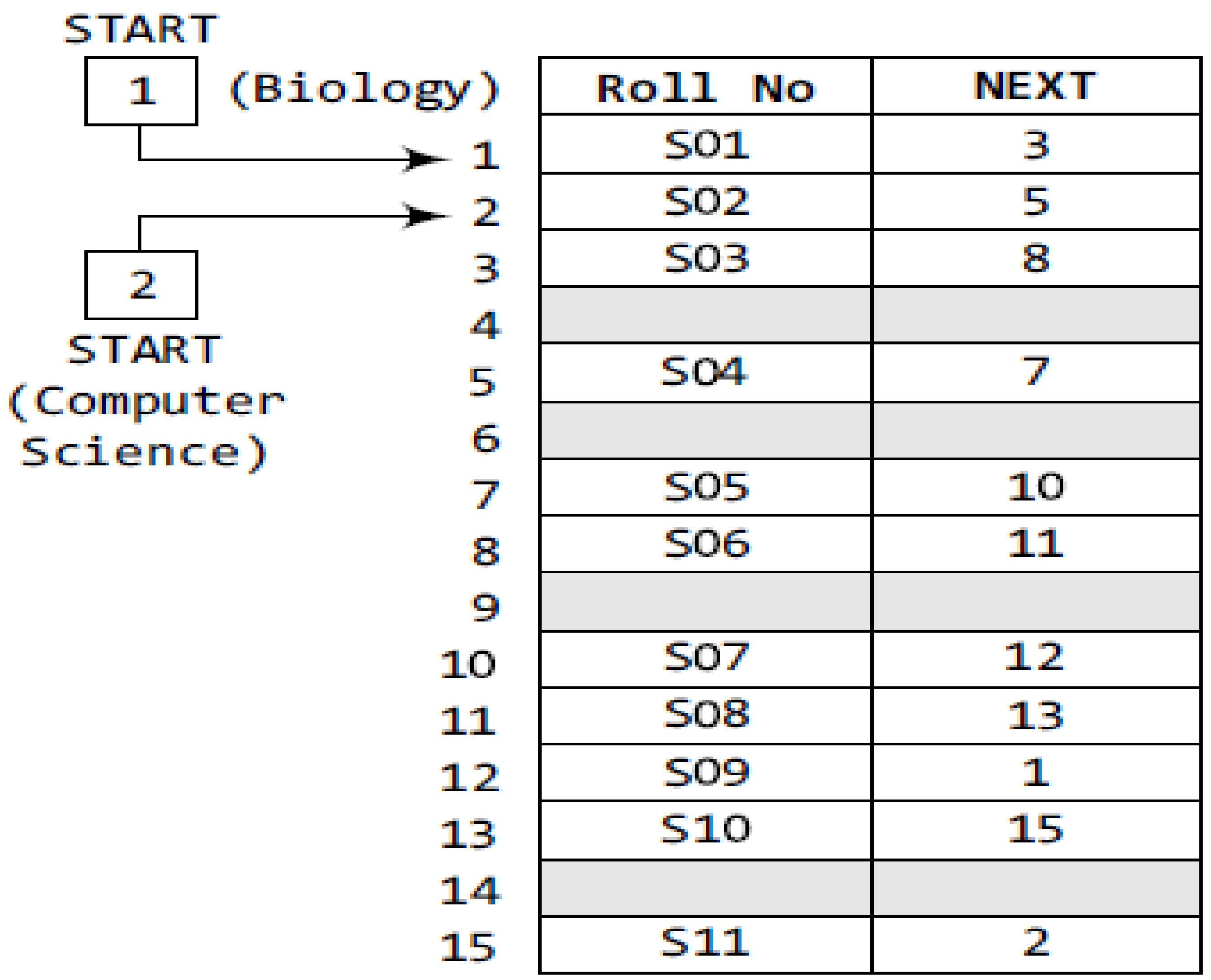


	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	1

Circular Linked List

- Two different linked lists are simultaneously maintained in the memory. There is no ambiguity in traversing through the list because each list maintains a separate START pointer which gives the address of the first node of the respective linked list.
- The remaining nodes are reached by looking at the value stored in NEXT.
- The roll numbers of the students who have opted for Biology are
- S01, S03, S06, S08, S10, and S11.
- Similarly, the roll numbers of the students who chose Computer Science are S02, S04, S05, S07, and S09.

Circular Linked List



	Roll No	NEXT
1	S01	3
2	S02	5
3	S03	8
4		
5	S04	7
6		
7	S05	10
8	S06	11
9		
10	S07	12
11	S08	13
12	S09	1
13	S10	15
14		
15	S11	2

Circular Linked List

Inserting a New Node in a Circular Linked List

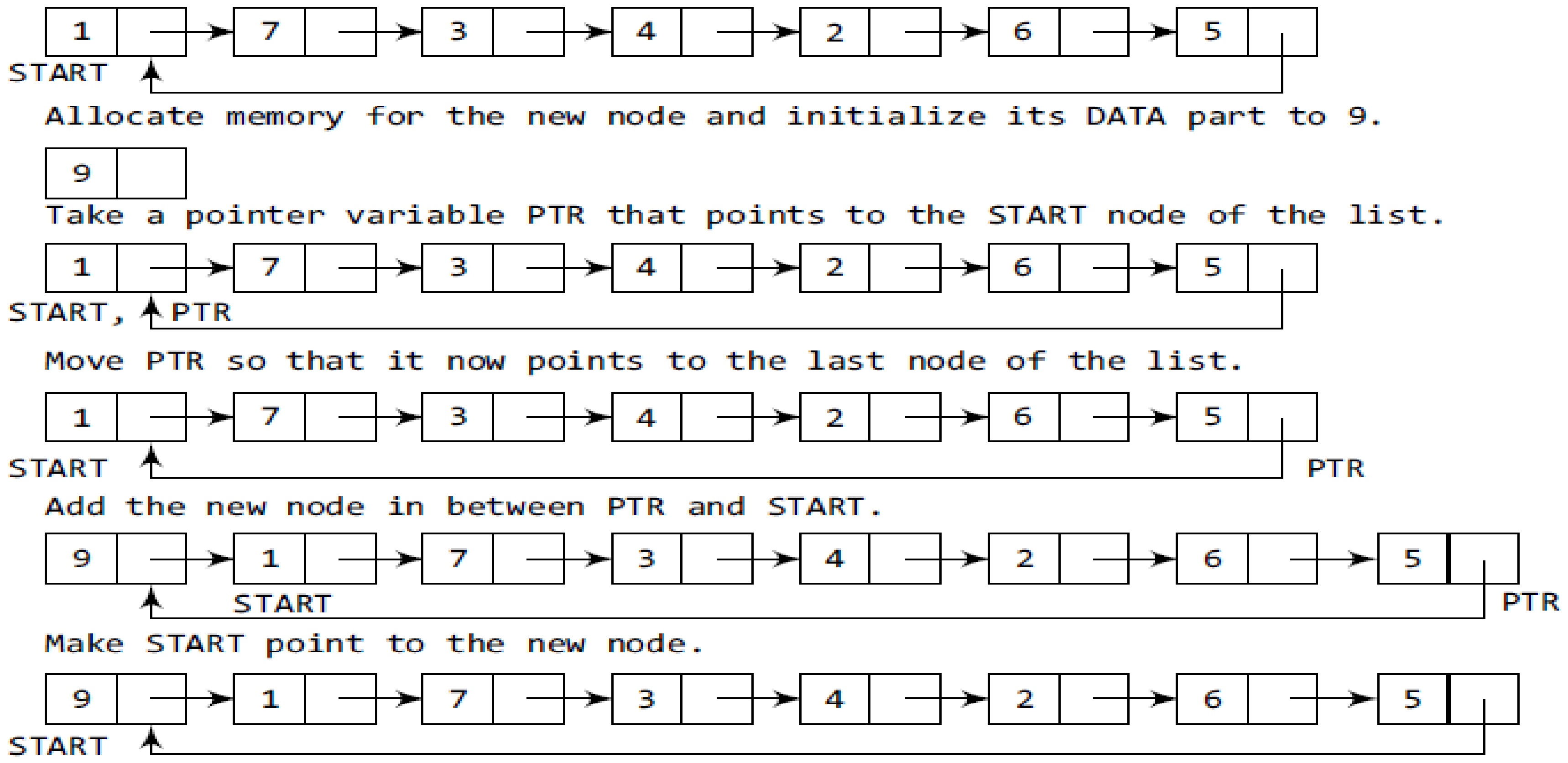
How a new node is added into an already existing linked list.

Two cases

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



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.

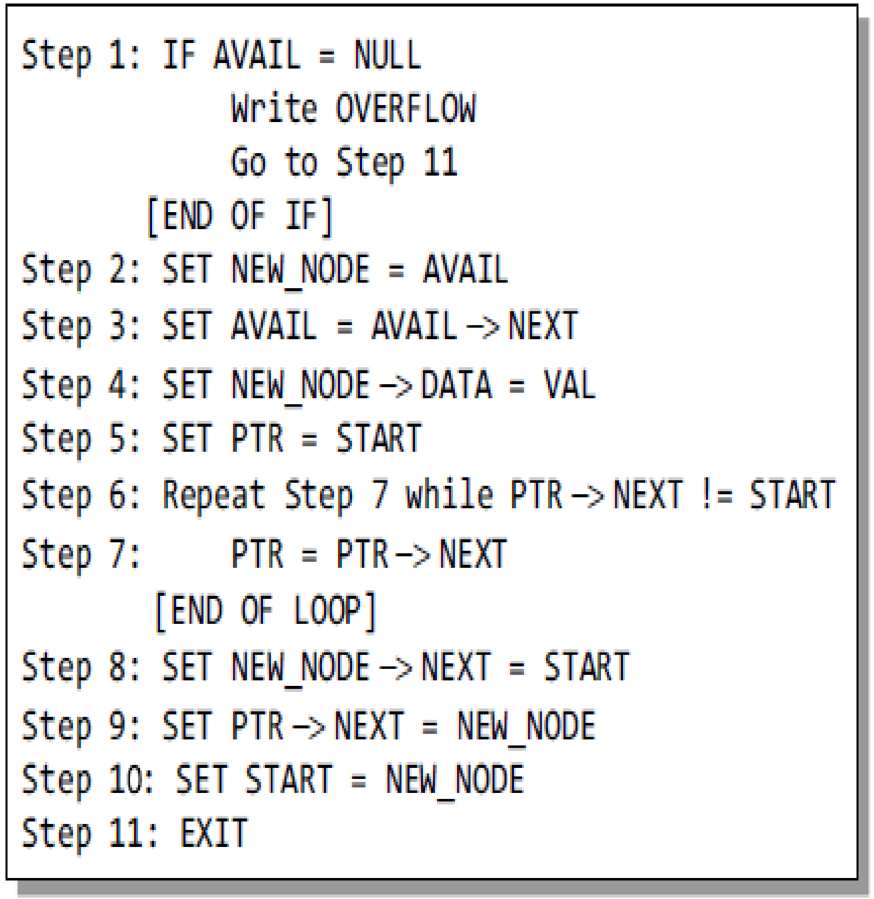
Move PTR so that it now points to the last node of the list.

Add the new node in between PTR and START.

Make START point to the new node.

Inserting a Node at the Beginning of a Circular Linked List

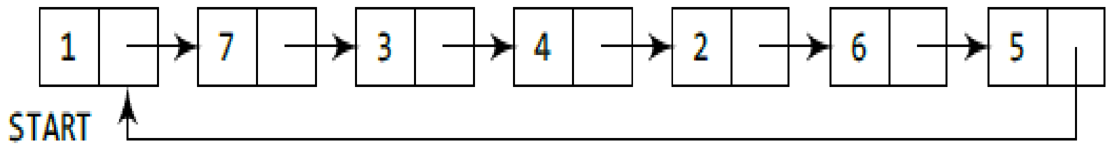
.



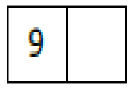
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:       PTR = PTR ->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

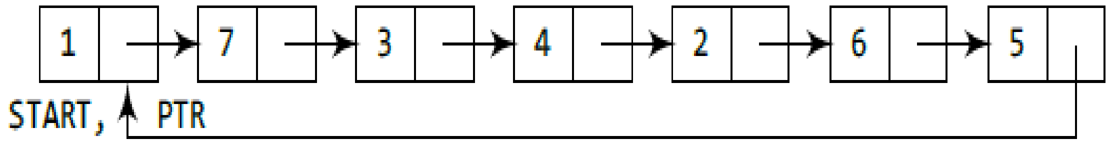
Inserting a Node at the end of a Circular Linked List



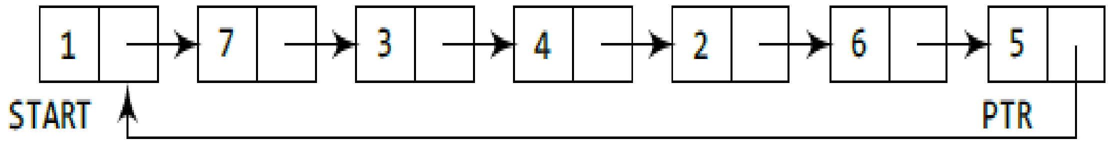
Allocate memory for the new node and initialize its DATA part to 9.



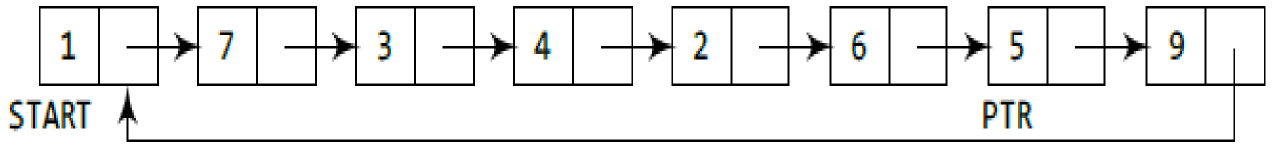
Take a pointer variable PTR which will initially point to START.



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

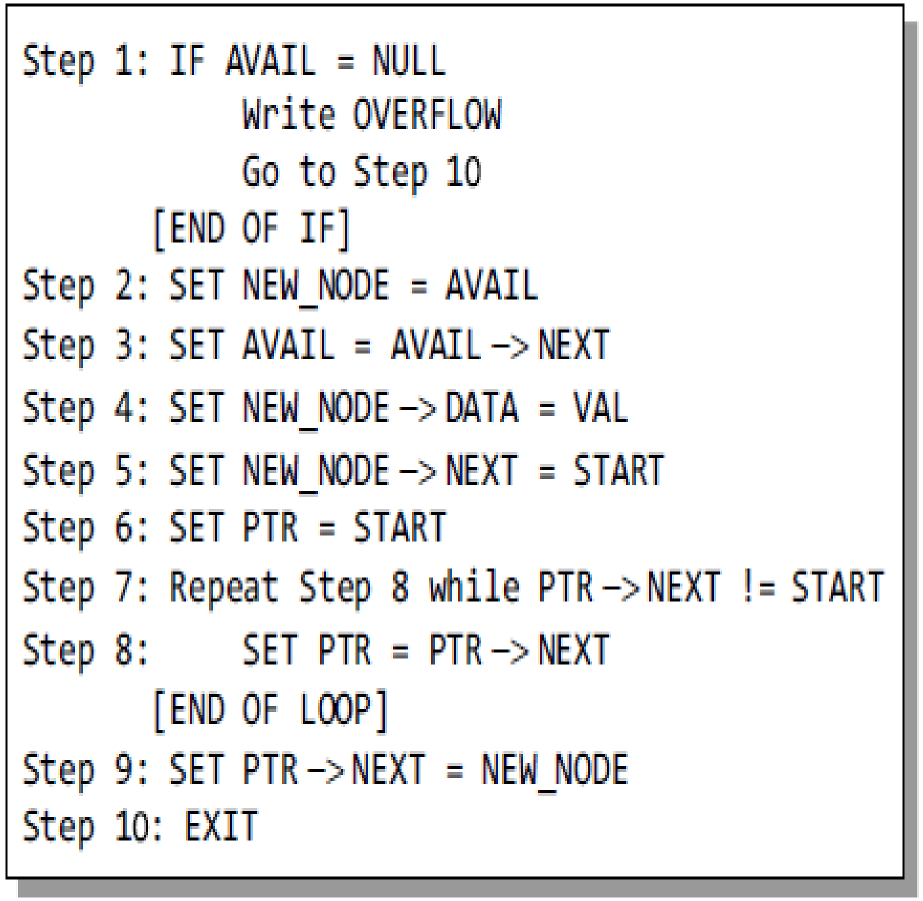


Add the new node after the node pointed by PTR.



Inserting a Node at the end of a Circular 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 = 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 field of the new node contains the address of the first node which is denoted by START.

Circular Linked List

Deleting a Node from a Circular Linked List

Case 1: The first node is deleted.

Case 2: The last node is deleted.

Circular Linked List

Deleting a Node from a Circular Linked List

Case 1: The first node is deleted.

Case 2: The last node is deleted.

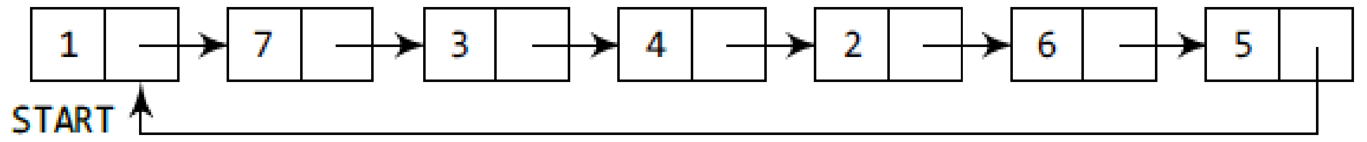
Circular Linked List

Deleting a Node from a Circular Linked List

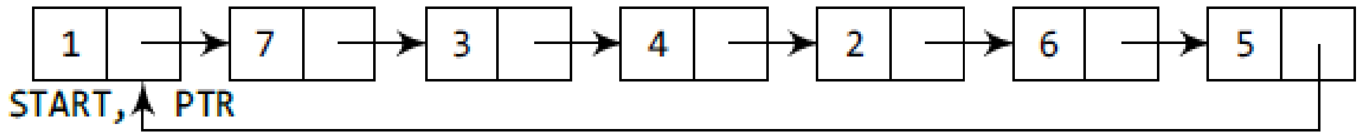
Case 1: The first node is deleted.

Case 2: The last node is deleted.

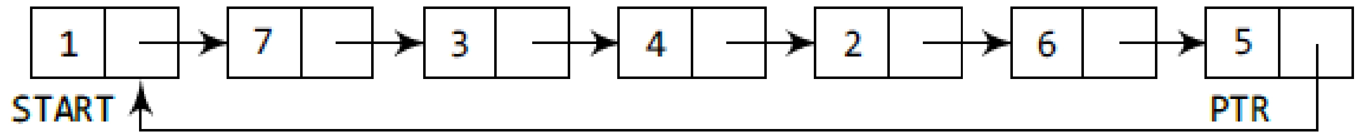
Deleting the first Node from a Circular Linked List



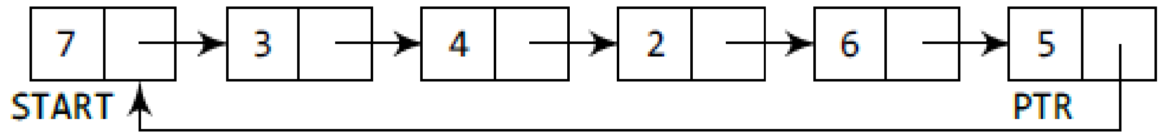
Take a variable PTR and make it point to the START node of the list.



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

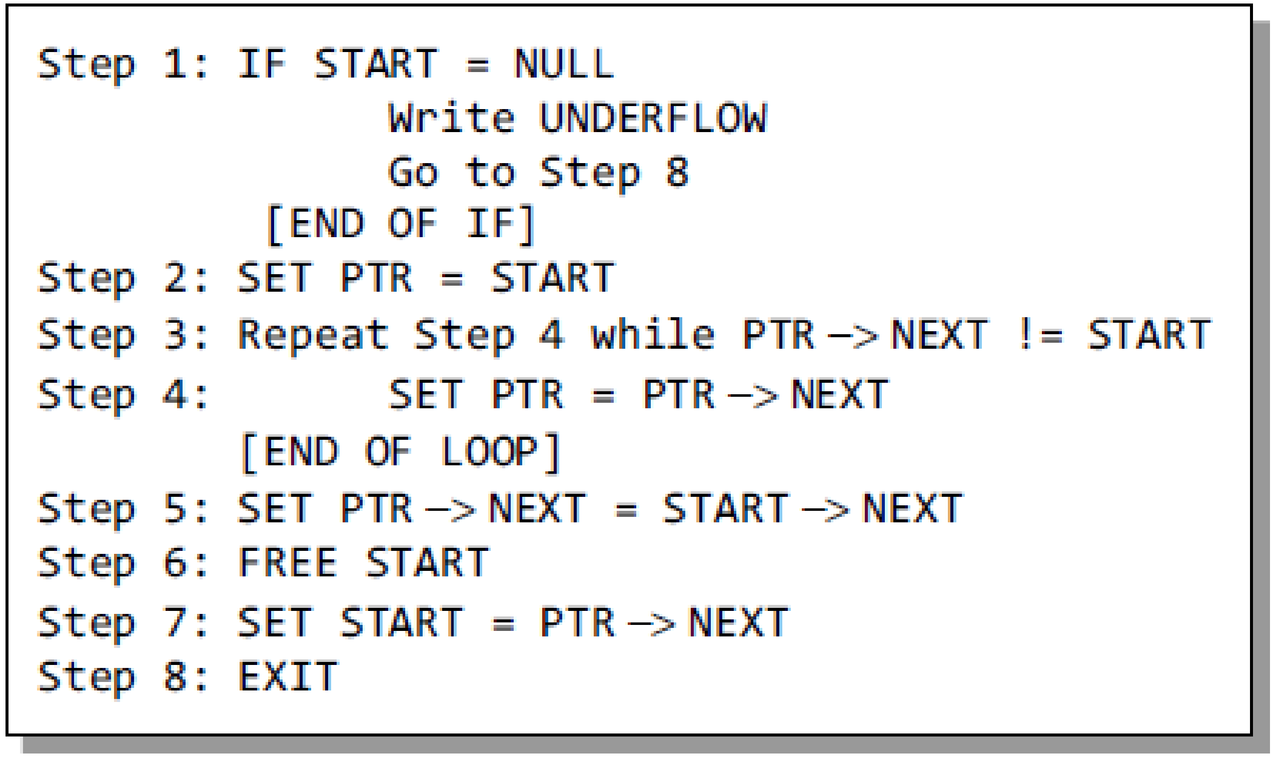


The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Deleting the first Node from a Circular Linked List

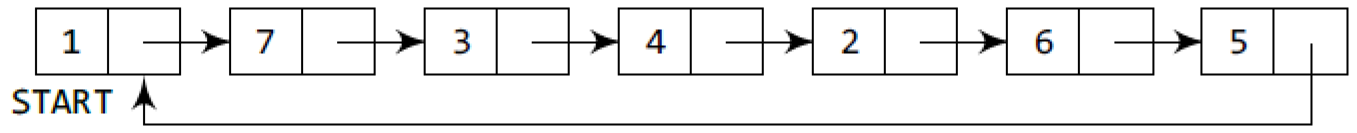


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step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

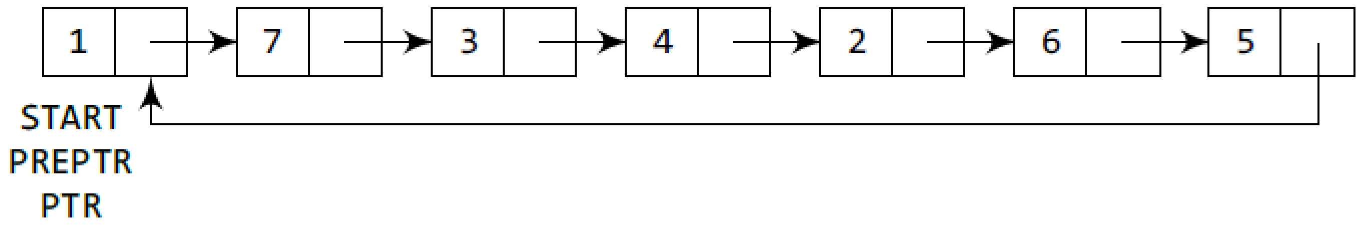
- However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node.
- In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list. In Step 6, the memory occupied by the first node is freed.
- Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.

Circular Linked List

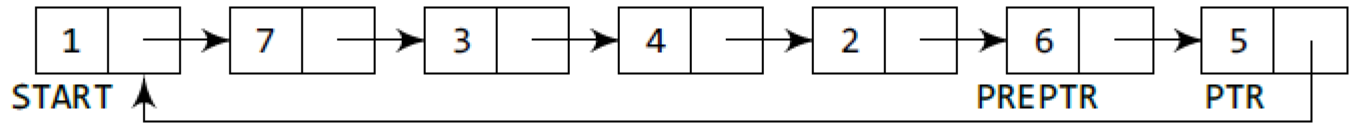
Deleting the last Node from a Circular Linked List



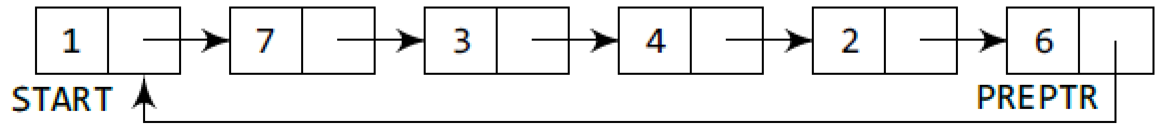
Take two pointers PREPTR and PTR which will initially point to START.



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

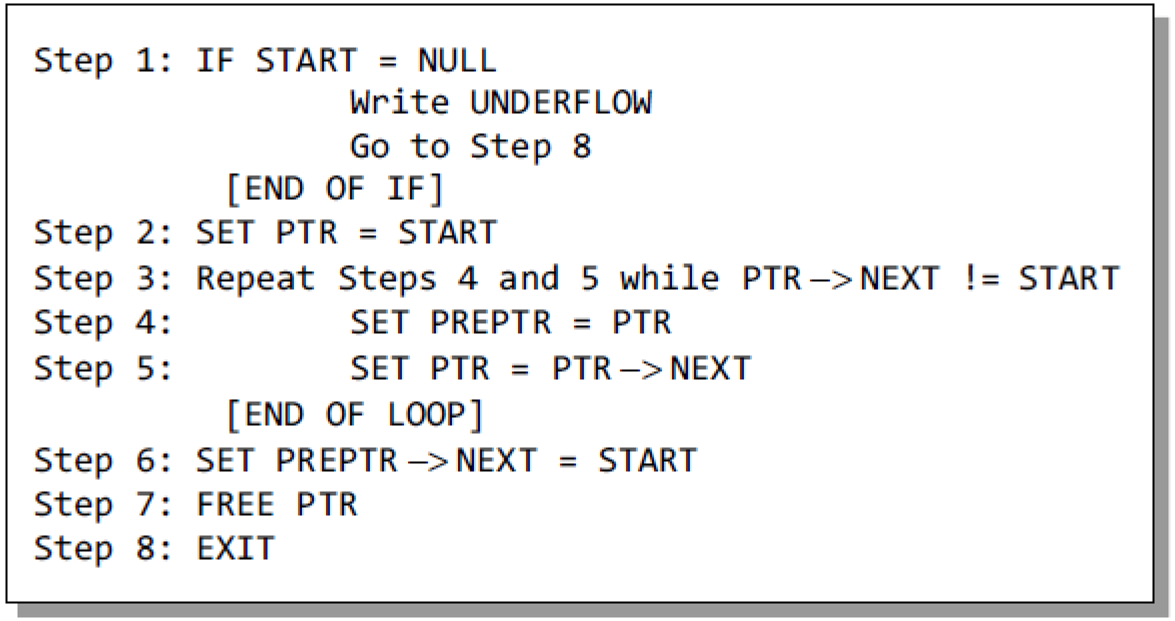


Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.



Circular Linked List

Deleting the last Node from a Circular Linked List



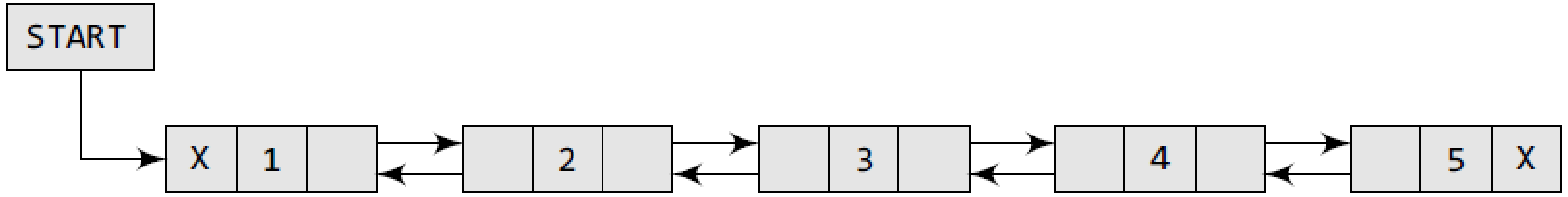
```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
          [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR->NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR->NEXT
          [END OF LOOP]
Step 6: SET PREPTR->NEXT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

- In Step 2, we take a pointer variable PTR and initialize it with START. PTR now points to the first node of the 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 Linked 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.

- It consists of three parts—data, a pointer to the next node, and a pointer to the previous node.



DOUBLY Linked List

- A doubly linked list calls for more space per node and more expensive basic operations.

- A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).

- The main advantage of using a doubly linked list is that it makes searching twice as efficient.

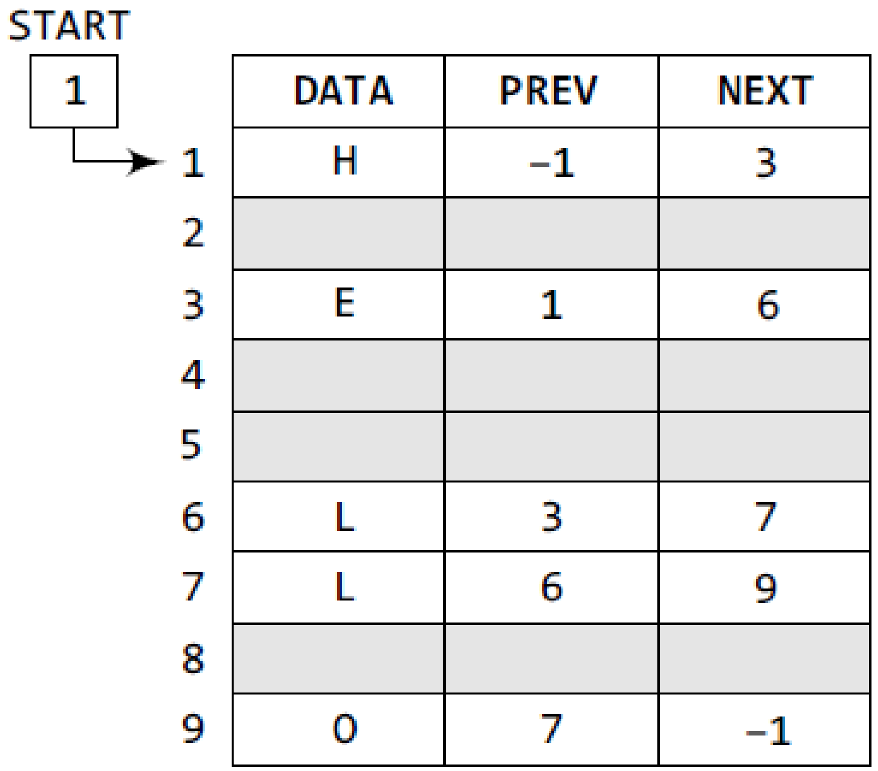
DOUBLY Linked List

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

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

- The PREV field of the first node and the NEXT field of the last node will contain NULL.
- The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.

DOUBLY Linked List



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

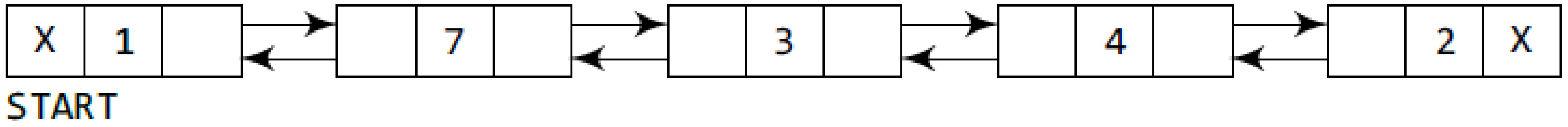
- A variable START is used to store the address of the first node.
- In this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL in the PREV field. We will traverse the list until we reach a position where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

Inserting a New Node in a DOUBLY Linked List

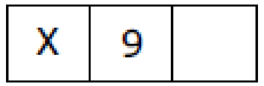
Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

DOUBLY Linked List

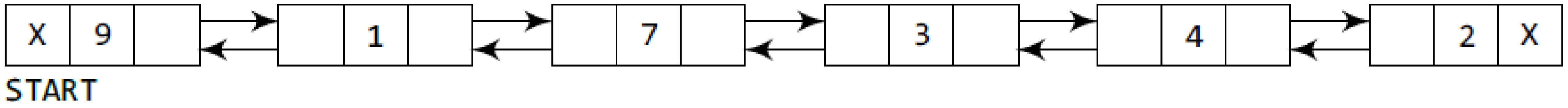
Inserting a Node at the Beginning of a Doubly Linked List



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.



DOUBLY Linked List

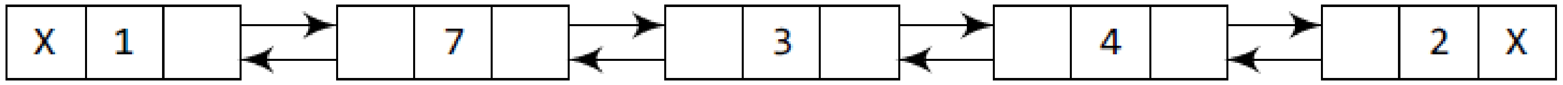
Inserting a Node at the Beginning of a Doubly Linked List

```
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.

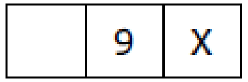
DOUBLY Linked List

Inserting a Node at the End of a Doubly Linked List

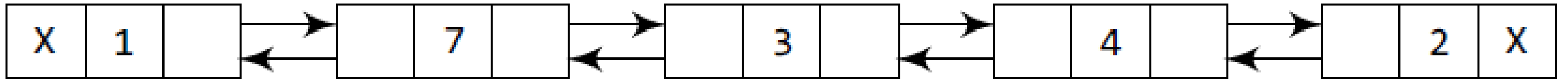


START

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

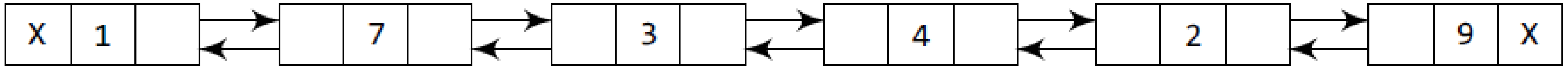


Take a pointer variable PTR and make it point to the first node of the list.



START,PTR

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



START PTR

DOUBLY Linked List

Inserting a Node at the End of a Doubly Linked 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
```

- 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).

DOUBLY Linked List

Deleting a Node from a Doubly Linked List

Case 1: The first node is deleted.

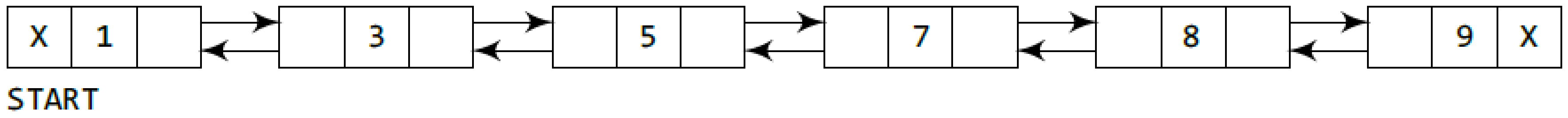
Case 2: The last node is deleted.

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

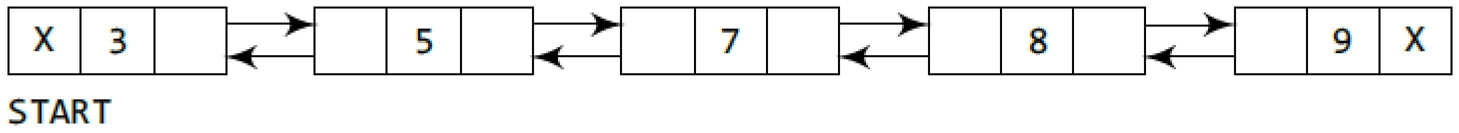
Case 4: The node before a given node is deleted.

DOUBLY Linked List

Deleting the First Node from a Doubly 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.



DOUBLY Linked List

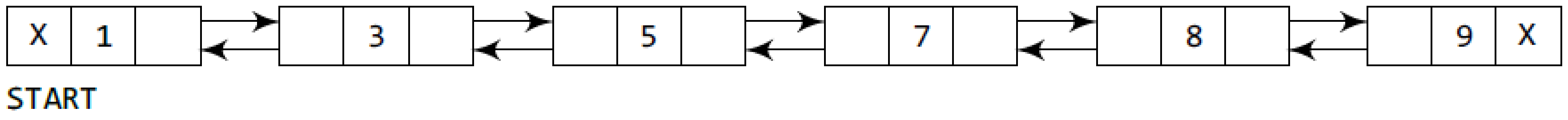
Deleting the First Node from a Doubly Linked List

```
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
```

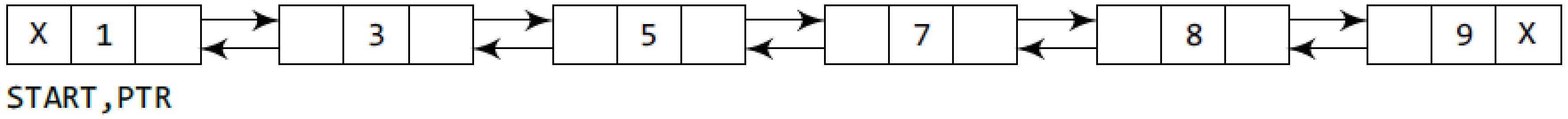
- In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm.
- However, if there are nodes in the linked list, then we use a temporary pointer variable 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.

DOUBLY Linked List

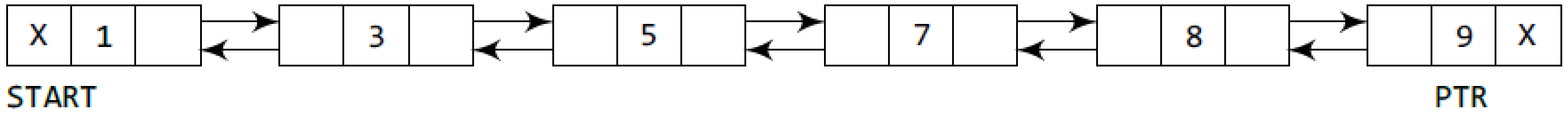
Deleting the last Node from a Doubly Linked List



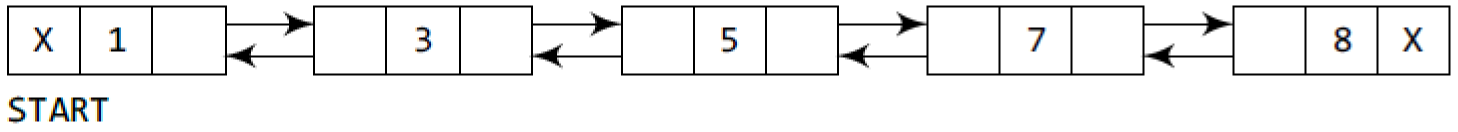
Take a pointer variable PTR that points to the first node of the list.



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

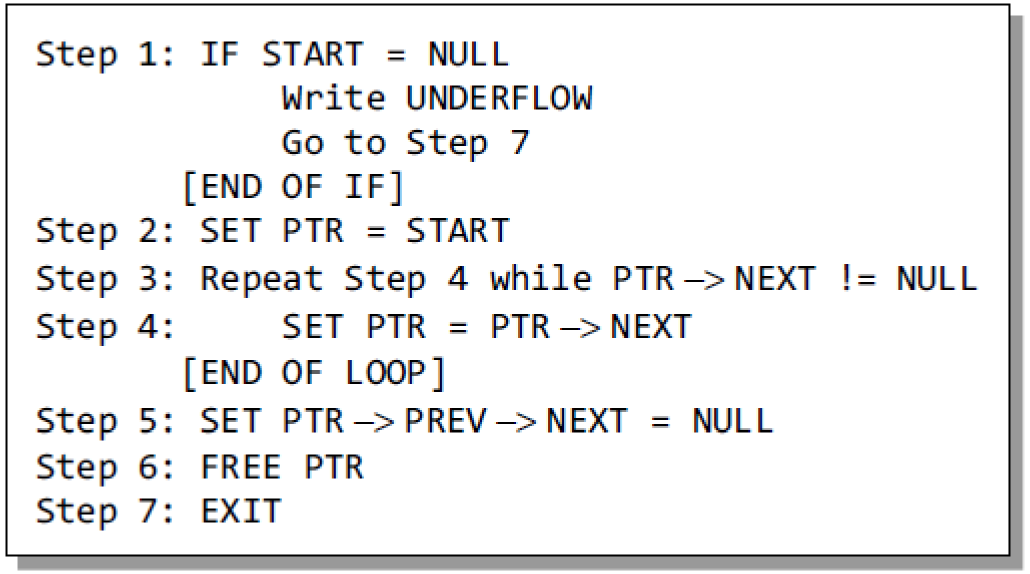


Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.



DOUBLY Linked List

Deleting the last Node from a Doubly Linked List



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

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

DOUBLY Linked List

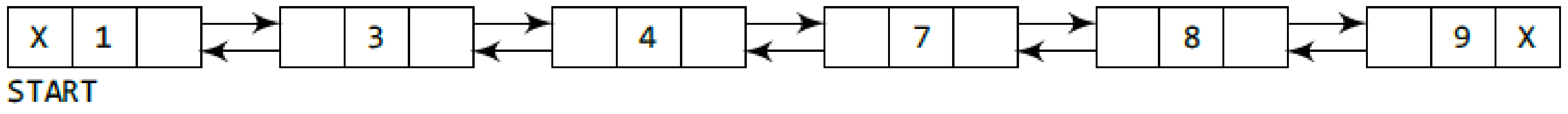
Deleting the last Node from a Doubly Linked List

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

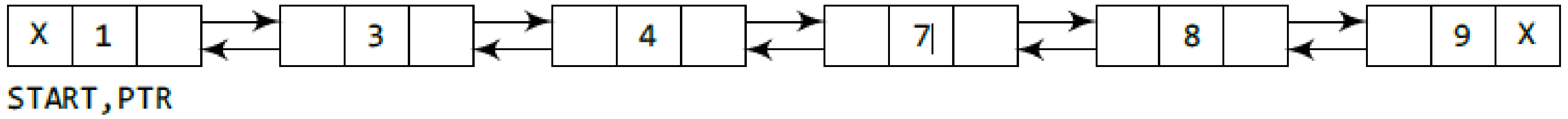
- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list.
- The while loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the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

DOUBLY Linked List

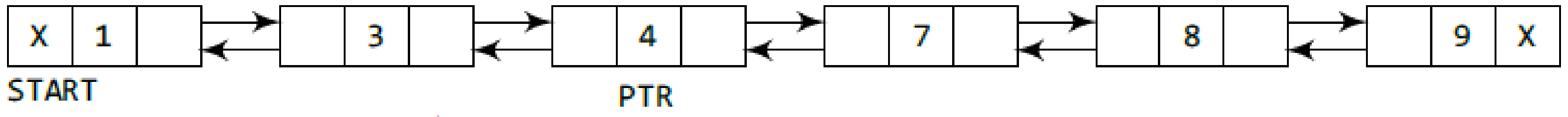
Deleting the Node After a Given Node in a Doubly Linked List



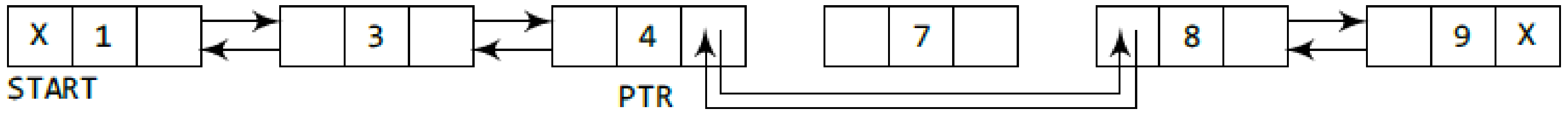
Take a pointer variable PTR and make it point to the first node of the list.

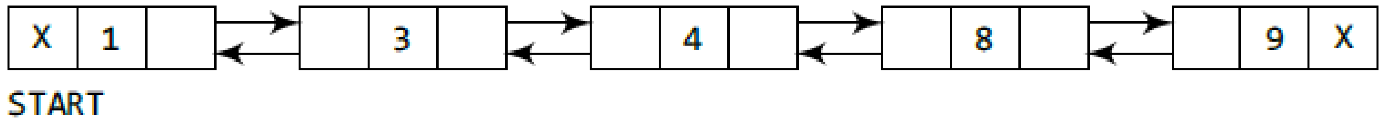


Move PTR further so that its data part is equal to the value after which the node has to be inserted.



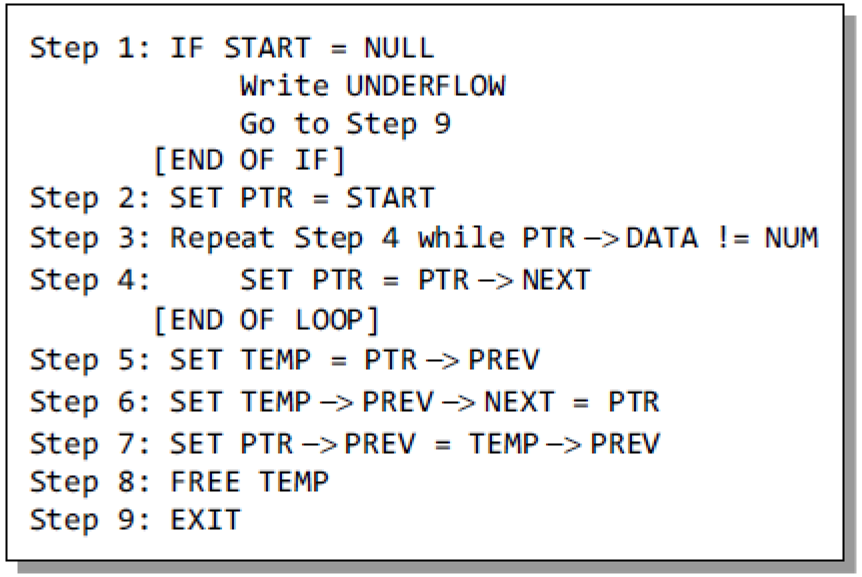
Delete the node succeeding PTR.





DOUBLY Linked List

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

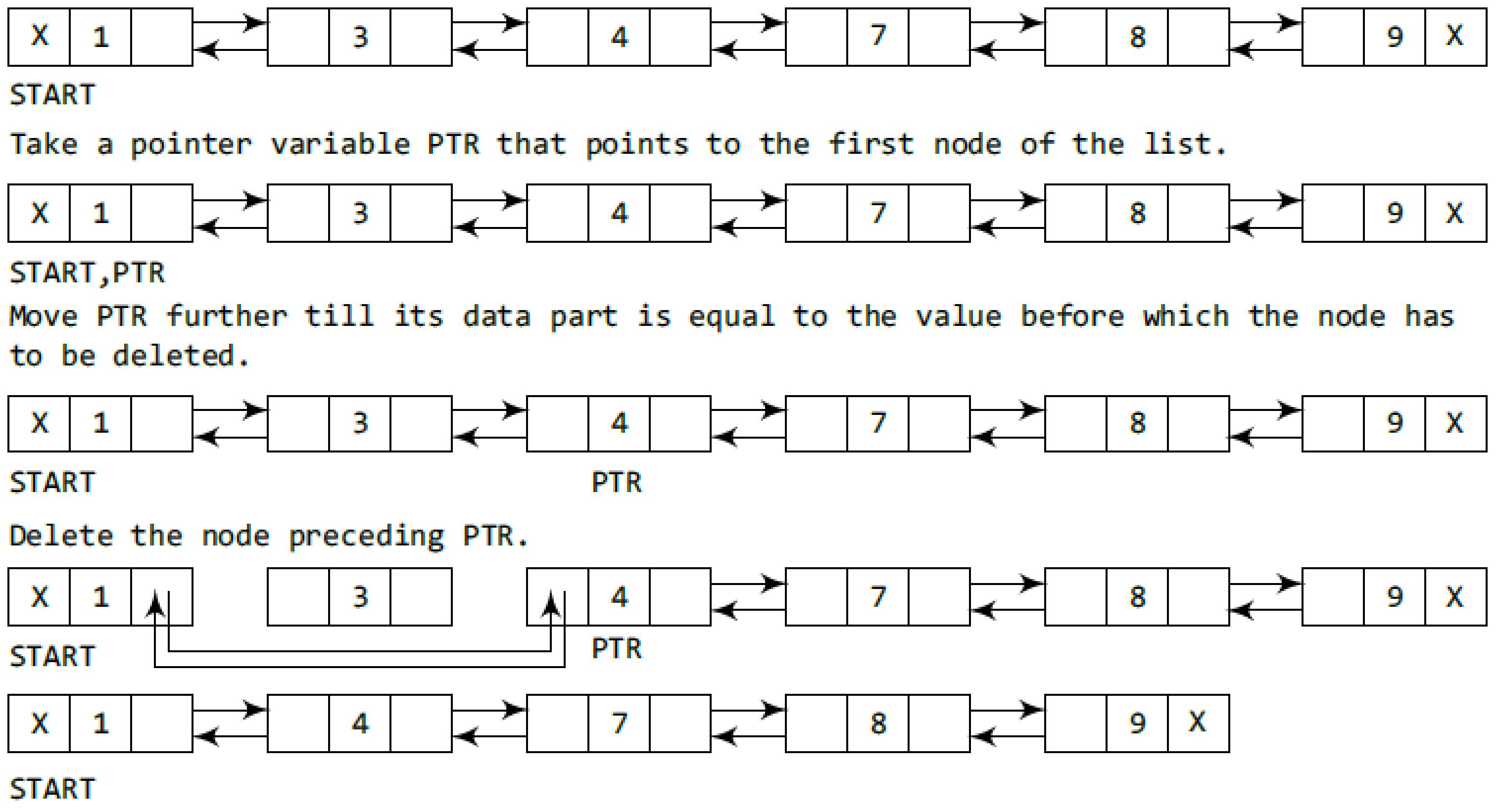


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR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

- In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the 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.
- The memory of the node succeeding the given node is freed and returned to the free pool.

DOUBLY Linked List

Deleting the Node Before a Given Node in a Doubly Linked List



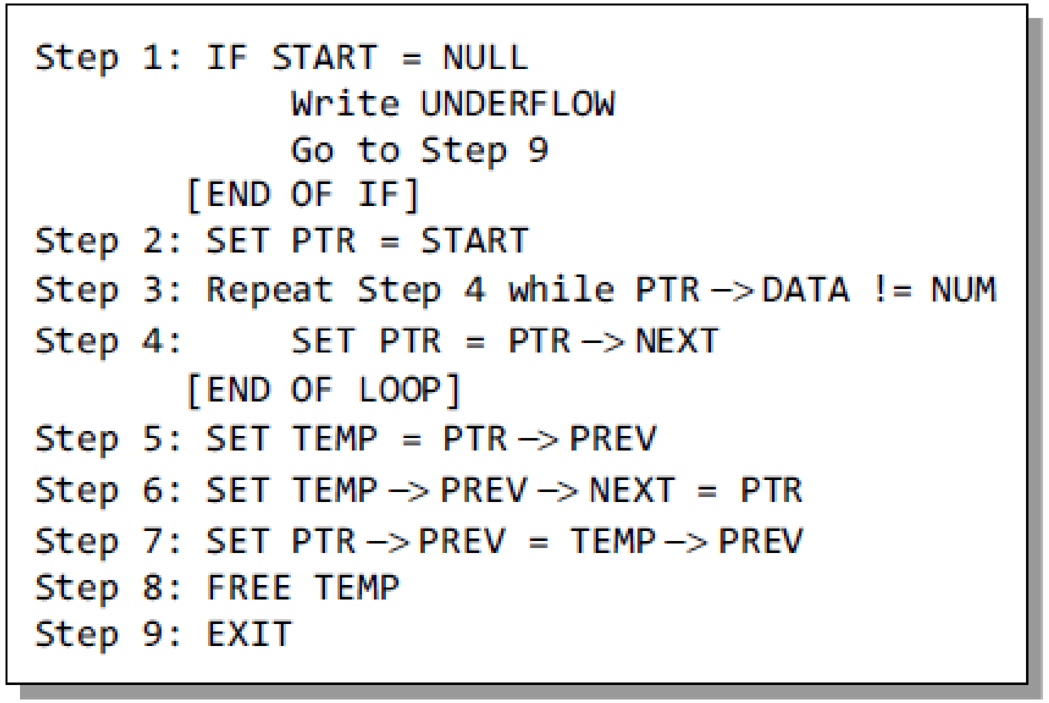
Take a pointer variable PTR that points to the first node of the list.

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.

DOUBLY Linked List

Deleting the Node Before a Given Node in a Doubly Linked List



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

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

Queries???

Thank you!!