Basic Data Structures

Unit 5 : Linear Data Structures III

Prepared by :
Prof. Khushbu Chauhan
Computer Engg. Dept.

MPSTME, NMIMS

Outlines

- Introduction : Linked List
- Basic terminologies
- Linked List versus Arrays
- Memory Allocation and Deallocation for a Linked List
- Types of Linked Lists: Singly Linked List, Doubly Linked List
- Operations of Linked List
- Traversing a Linked List
- Applications of Linked Lists
- Stack and queue implementations using Linked List

Introduction: Linked list

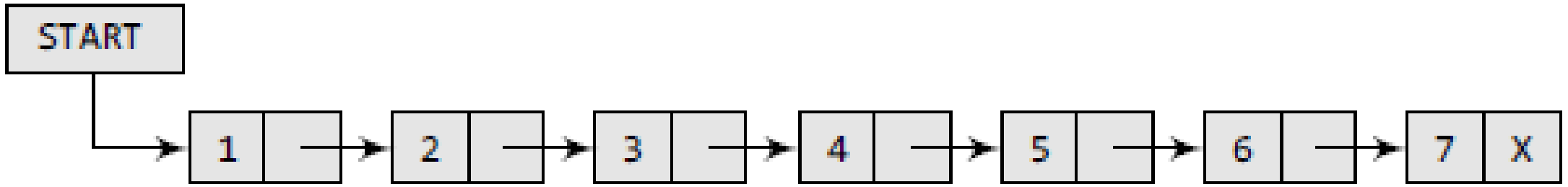
- Linked list is a linear data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data.
- Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

Why need Linked list

- An array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance?
- Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Basic terminologies

- 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.



- 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.
- 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. It 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, It can implement a linked list using the following code:
- Struct node
- {
- int data;
- struct node *next;
- };
- 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.

START

1

	Data	Next
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	O	–1

- 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, look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E. Again, the corresponding NEXT to go to the next node. From the entry in the NEXT, get the next address, that is 7, and fetch L as the data.
- Repeat this procedure until reach a position where the NEXT entry contains –1 or NULL.

Linked Lists versus Arrays

- Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations. Another point of difference between an array and a linked list is that a linked list does not allow random access of data.
- Nodes in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

Linked Lists versus Arrays

- Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array.
- For example, if declare an array as int marks[20], then the array can store a maximum of 20 data elements only. There is no such restriction in case of a linked list.

START

18

	Roll No	Name	Aggregate	Grade	Next
1	S01	Ram	78	Distinction	6
2	S02	Shyam	64	First division	14
3					
4	S03	Mohit	89	Outstanding	17
5					
6	S04	Rohit	77	Distinction	2
7	S05	Varun	86	Outstanding	10
8	S06	Karan	65	First division	12
9					
10	S07	Veena	54	Second division	-1
11	S08	Meera	67	First division	4
12	S09	Krish	45	Third division	13
13	S10	Kusum	91	Outstanding	11
14	S11	Silky	72	First division	7
15					
16					
17	S12	Monica	75	Distinction	1
18	S13	Ashish	63	First division	19
19	S14	Gaurav	61	First division	8

Memory Allocation & Deallocation for a Linked List

- A dynamic memory allocation method used in computer programming is called linked list allocation. In this method, a linked list data structure is used to distribute memory.
- Memory is divided into a number of blocks of similar size when allocating a linked list. In the linked list, each block is symbolized by a node. A pointer to the following piece of memory is present at each node in the linked list.

START

1

(Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	-1

(a)

Students' linked list

START

1

(Biology)

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4	S12	75	-1
5	S04	98	7
6			
7	S05	55	8
8	S06	34	10
9			
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			
15	S11	56	4

(b)

linked list after the insertion of new student's record

- Every linked list has a pointer variable START which stores the address of the first node of the list. Likewise, for the free pool (which is a linked list of all free memory cells), have a pointer variable AVAIL which stores the address of the first free space. Now, when a new student's record has to be added, the memory address pointed by AVAIL will be taken and used to store the desired information. After the insertion, the next available free space's address will be stored in AVAIL. For example, in example when the first free memory space is utilized for inserting the new node, AVAIL will be set to contain address 6.

START

1

(Biology)

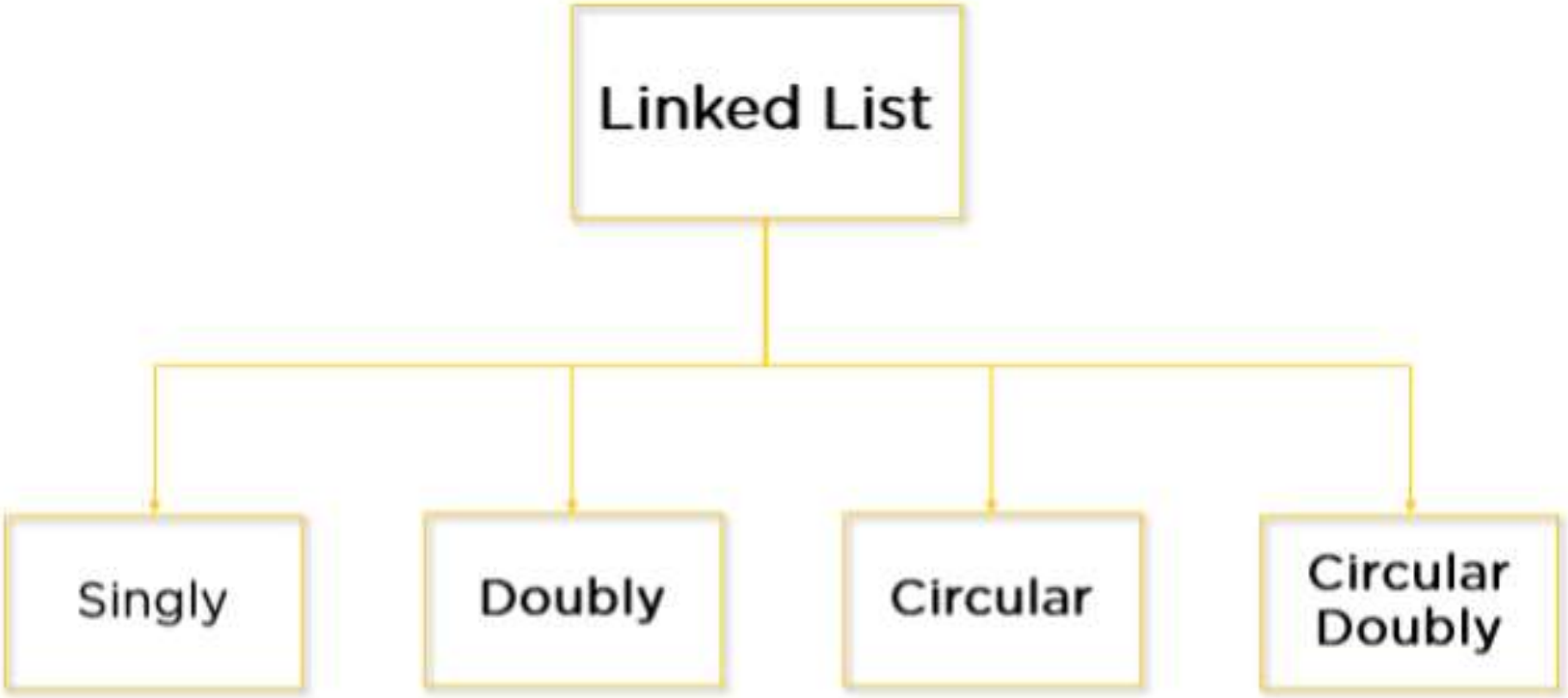
4

AVAIL

	Roll No	Marks	Next
1	S01	78	2
2	S02	84	3
3	S03	45	5
4			6
5	S04	98	7
6			9
7	S05	55	8
8	S06	34	10
9			14
10	S07	90	11
11	S08	87	12
12	S09	86	13
13	S10	67	15
14			-1
15	S11	56	-1

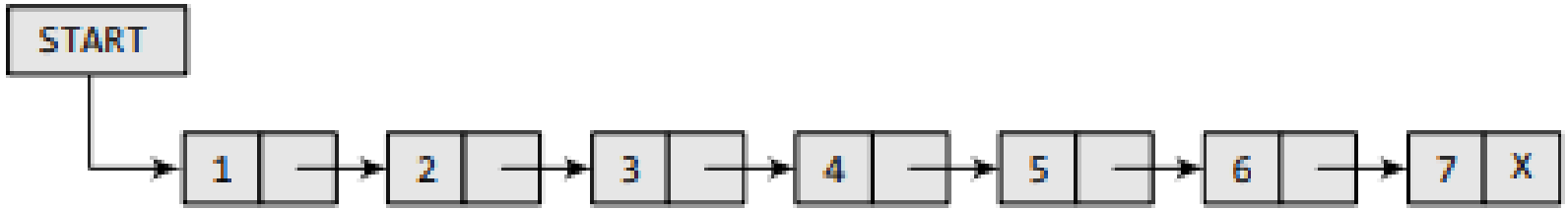
```
struct Node {
   int data;
   struct Node *next;
};
struct Node* head = NULL;
void insert(int new_data) {
   struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
   new_node->data = new_data;
   new_node->next = head;
   head = new_node;
}
void display() {
   struct Node* ptr;
   ptr = head;
   while (ptr != NULL) {
      cout<< ptr->data <<" ";
      ptr = ptr->next;
   }
}
int main() {
   insert(3);
   insert(1);
   insert(7);
   insert(2);
   insert(9);
   cout<<"The linked list is: ";
   display();
   return 0;
}
```

Types of 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. By saying that the node contains a pointer to the next node, mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way.



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.

```
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
```

Algorithm for traversing 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
```

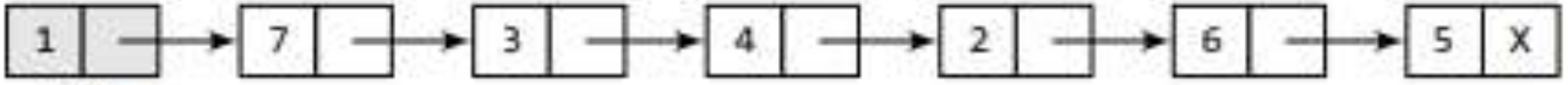
Algorithm to print the number of nodes in a linked list

Searching for a Value in a Linked List

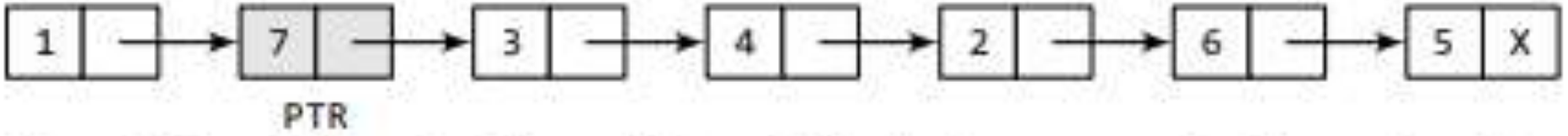
- Searching a linked list means to find a particular element in the linked list. A linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, 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
```

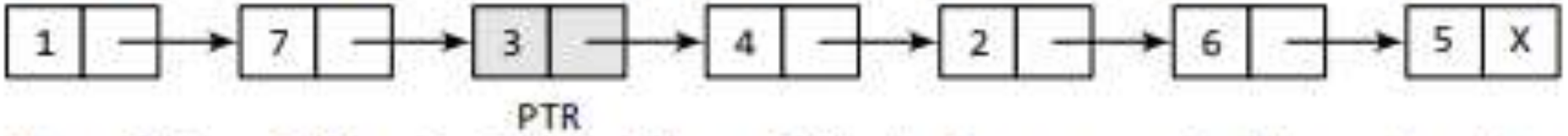
Example



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.

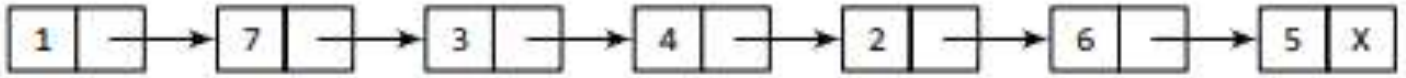
Here PTR -> DATA = 4. Since PTR -> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

Inserting a New Node in a Linked List

- Consider four cases
- 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.
- 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.

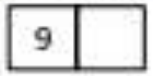
Case 1 : Inserting a Node at the Beginning of a Linked List

```
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
```

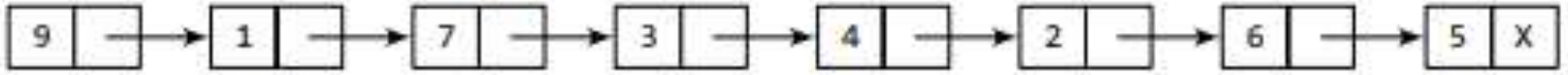


START

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

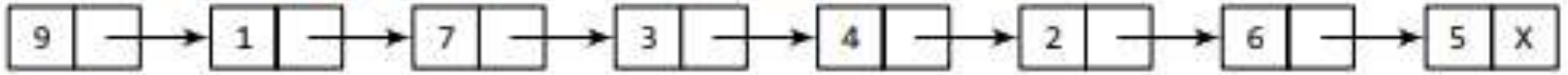


Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



START

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



START

SVKM'S NMIMS Deemed to be UNIVERSITY

Creating a head node

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node *link;
};

int main() {
    struct node *head = NULL;
    head = (struct node *)malloc(sizeof(struct node));

    head->data = 45;
    head->link = NULL;

    printf("%d", head->data);
    return 0;
}
```

```
struct Node {
    int data;
    Node* next;
    Node(int new_data)
    {
        data = new_data;
        next = nullptr;
    }
};
Node* insertAtFront(Node* head, int new_data)
{
    Node* new_node = new Node(new_data);
    new_node->next = head;
    return new_node;
}
void printList(Node* head)
{
    Node* curr = head;
    while (curr != nullptr) {
        cout << " " << curr->data;
        curr = curr->next;
    }
    cout << endl;
}
```

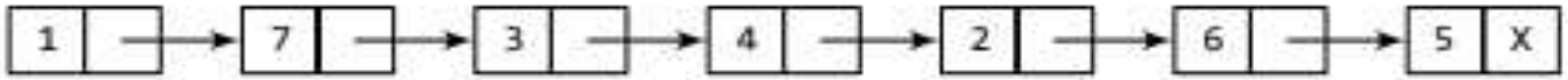
```
int main()
{
    Node* head = new Node(2);
    head->next = new Node(3);
    head->next->next = new Node(4);
    head->next->next->next = new Node(5);
    cout << "Original Linked List:";
    printList(head);
    cout << "After inserting Nodes at the front:";
    int data = 1;
    head = insertAtFront(head, data);
    printList(head);
    return 0;
}
```

```
/tmp/LWEq16WudF.o
Original Linked List: 2 3 4 5
After inserting Nodes at the front: 1 2 3 4 5
```

Case 2 : Inserting a Node at the End of a Linked List

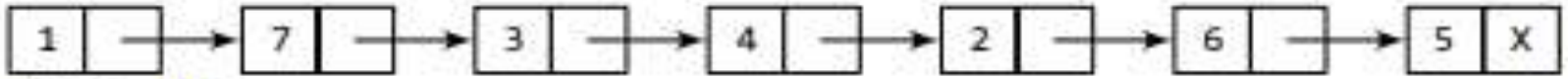
- The algorithm to insert a new node at the end of a linked list. In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, traverse through the linked list to reach the last node. Once we reach the last node, 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
```

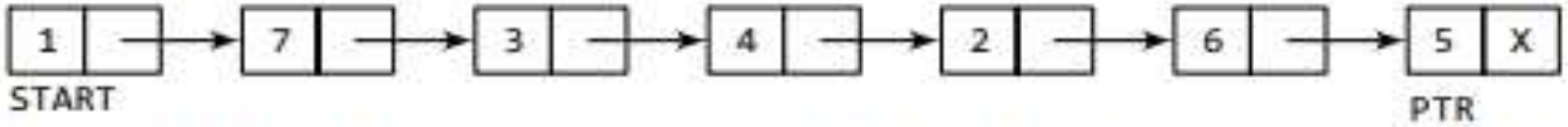


Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

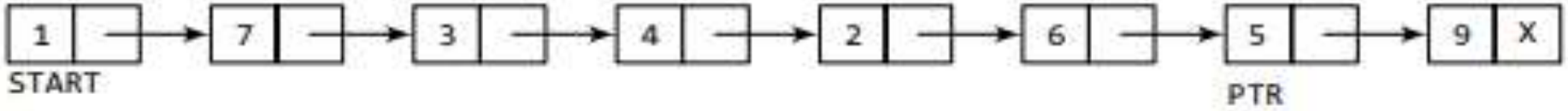
Take a pointer variable PTR which points to START.



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.



```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int new_data) {
    struct Node* new_node =
       (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    return new_node;
}

struct Node* append(struct Node* head, int new_data) {

    struct Node* new_node = createNode(new_data);
    if (head == NULL) {
        return new_node;
    }

    struct Node* last = head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = new_node;
    return head;
}
```

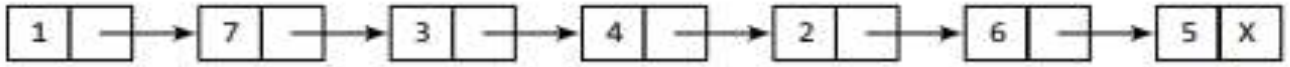
```
void printList(struct Node* node) {
    while (node != NULL) {
        cout << node->data;
        node = node->next;
    }
}

int main() {

    struct Node* head = createNode(2);
    head->next = createNode(3);
    head->next->next = createNode(4);
    head->next->next->next = createNode(5);
    head->next->next->next->next = createNode(6);

    cout<<"Created Linked list is:" << endl;
    printList(head);
    cout<< endl;
    head = append(head, 1);

    cout<<"After inserting 1 at the end:" << endl;
    printList(head);
       cout<< endl;

    return 0;
}
```

```
Created Linked list is:
23456
After inserting 1 at the end:
234561
```

Case 3 : Inserting a Node After a Given Node in a Linked List

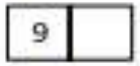
- In Step 5, take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then taken another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
- In the while loop, 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, changed 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
```

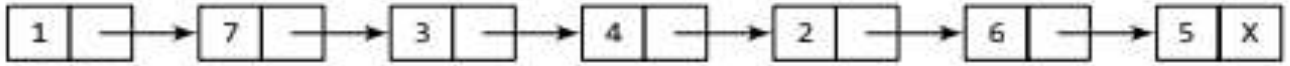


START

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.

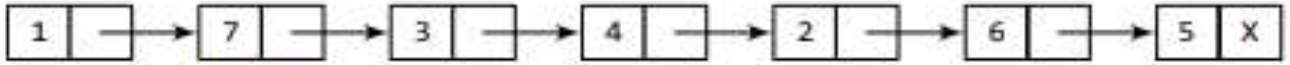


START

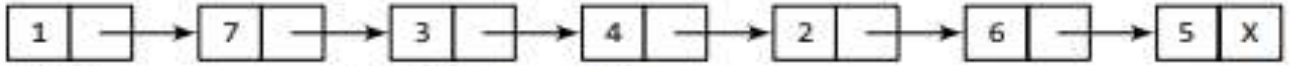
PTR

PREPTR

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.

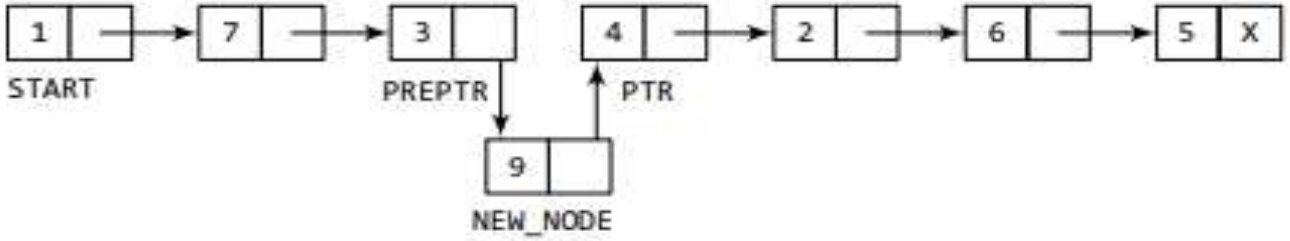


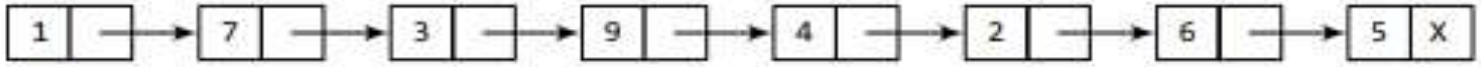
START PREPTR PTR



START PREPTR PTR

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





START

SVKM'S NMIMS Deemed to be UNIVERSITY

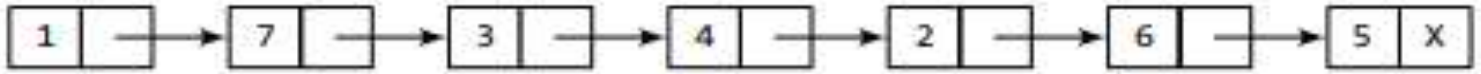
```cpp
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int data) {
    struct Node* newNode
        = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
struct Node* insertAfter(struct Node* head, int key,int newData) {
    struct Node* curr = head;
    while (curr != NULL) {
        if (curr->data == key)
            break;
        curr = curr->next;
    }
    if (curr == NULL) {
        printf("Node not found\n");
        return head;
    }
    struct Node* newNode
        = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->next = curr->next;
    curr->next = newNode;
    return head;
}
```

```cpp
void printList(struct Node* node) {
    while (node != NULL) {
        cout << node->data;
        node = node->next;
    }
    cout << endl;
}
int main() {

    struct Node* head = createNode(2);
    head->next = createNode(3);
    head->next->next = createNode(5);
    head->next->next->next = createNode(6);

    cout<<"Original Linked List: ";
    printList(head);
    cout << endl;
    int key = 3, newData = 4;
    head = insertAfter(head, key, newData);

    cout<<"Linked List after insertion: ";
    printList(head);
     cout << endl;
    return 0;
}
```

```
Original Linked List: 2356

Linked List after insertion: 23456
```

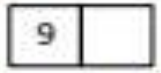
Case 4 : Inserting a Node Before a Given Node in a Linked List

```
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 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
```

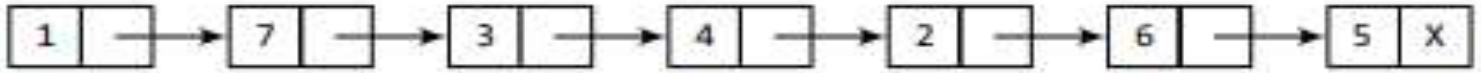


START

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



Initialize PREPTR and PTR to the START node.

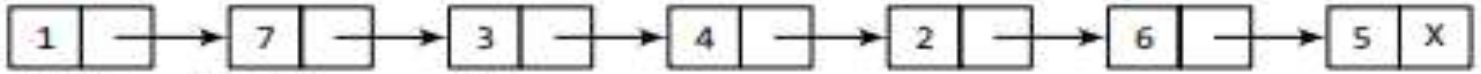


START

PTR

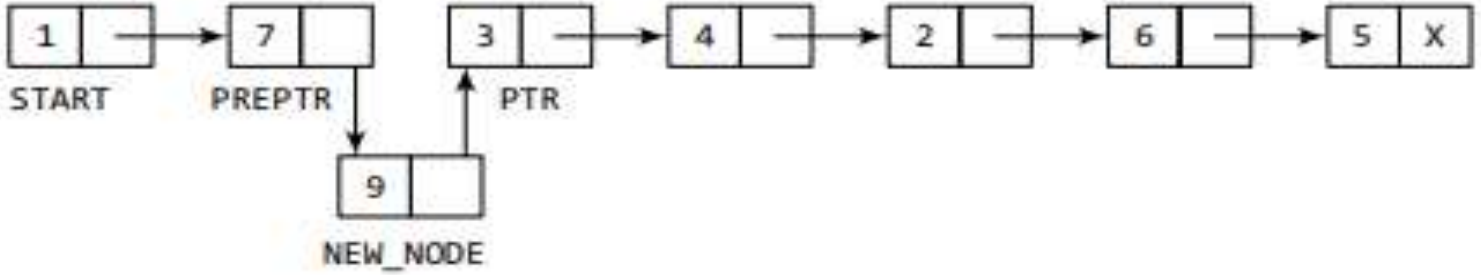
PREPTR

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.



START PREPTR PTR

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



START

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int x);
struct Node* insertBeforeKey(struct Node* head, int key, int newData) {

    if (head == NULL) {
        return NULL;
    }
    if (head->data == key) {
        struct Node* newNode = createNode(newData);
        newNode->next = head;
        return newNode;
    }
    head->next = insertBeforeKey(head->next, key, newData);
    return head;
}

void printList(struct Node* node) {
    struct Node* curr = node;
    while (curr != NULL) {
        cout<<curr->data;
        curr = curr->next;
    }
    cout<<endl;
}
```

```
struct Node* createNode(int x) {
    struct Node* new_node =
      (struct Node*)malloc(sizeof(struct Node));
    new_node->data = x;
    new_node->next = NULL;
    return new_node;
}
int main() {
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    int newData = 6;
    int key = 2;

    head = insertBeforeKey(head, key, newData);

    printList(head);

    return 0;
}
```

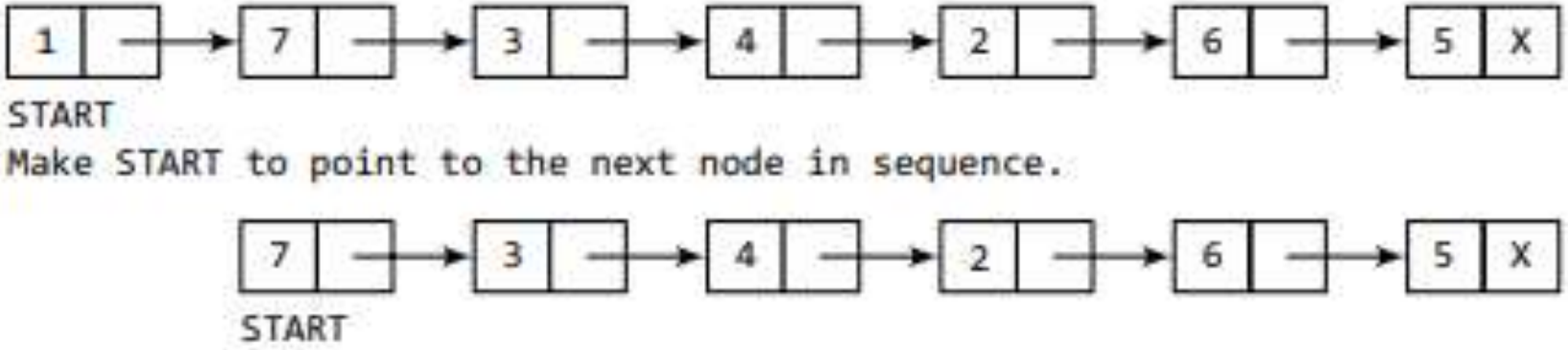
```
162345
```

Deleting a Node from a Linked List

- Consider three cases and 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 1 : Deleting the First 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
```



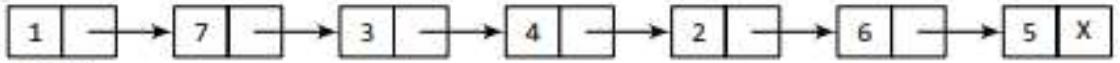
```cpp
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *next;
    Node(int x) {
        data = x;
        next = nullptr;
    }
};

Node* deleteHead(Node* head) {

    if (head == nullptr)
        return nullptr;

    Node* temp = head;

    head = head->next;

    delete temp;

    return head;
}
```

```cpp
void printList(Node* curr) {
    while (curr != nullptr) {
        cout << curr->data << " ";
        curr = curr->next;
    }
}

int main() {

    Node* head = new Node(3);
    head->next = new Node(12);
    head->next->next = new Node(15);
    head->next->next->next = new Node(18);
    head = deleteHead(head);
    printList(head);

    return 0;
}
```

```
12 15 18
```

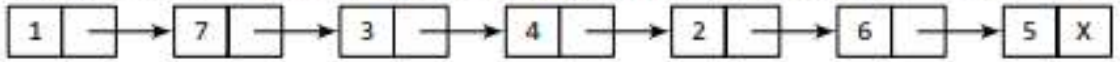
Case 2 : Deleting the Last Node from a Linked List

```
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 != 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
```



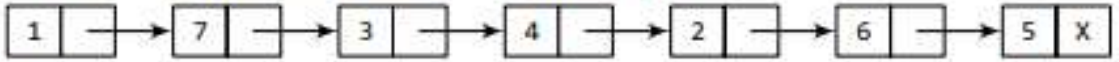
START

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



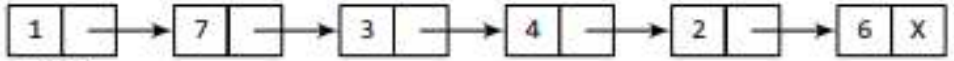
START
PREPTR
PTR

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



START PREPTR PTR

Set the NEXT part of PREPTR node to NULL.



START

SVKM'S NMIMS Deemed to be UNIVERSITY

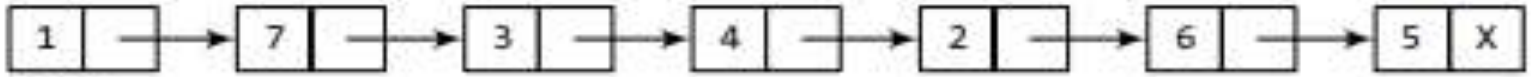
```
struct Node* createNode(int data)
{
    struct Node* newNode
        = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
struct Node* removeLastNode(struct Node* head)
{
    if (head == NULL) {
        return NULL;
    }
    if (head->next == NULL) {
        free(head);
        return NULL;
    }
    struct Node* second_last = head;
    while (second_last->next->next != NULL) {
        second_last = second_last->next;
    }
    free(second_last->next);
    second_last->next = NULL;

    return head;
}
```

```
void printList(struct Node* head)
{
    while (head != NULL) {
        cout<<head->data;
        head = head->next;
    }
}
int main()
{
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    cout<<"Original list: ";
    printList(head);
    cout<< endl;
    head = removeLastNode(head);
    cout<<"List after removing the last node: ";
    printList(head);
     cout<< endl;
    return 0;
}
```

```
Original list: 12345
List after removing the last node: 1234
```

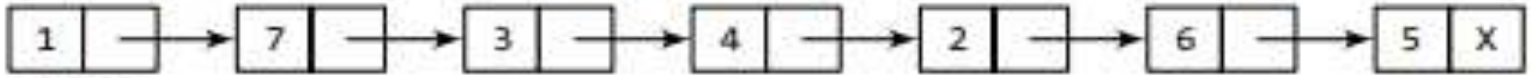
Case 3 : Deleting the Node After a Given Node in a Linked List

```
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
```



START

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

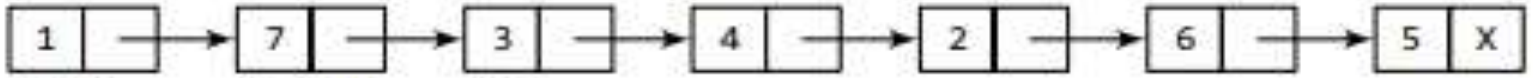


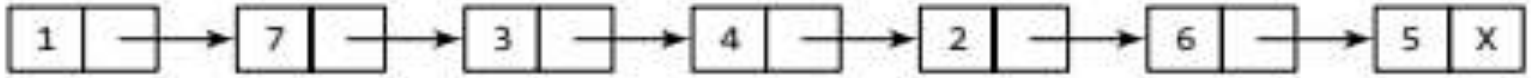
START

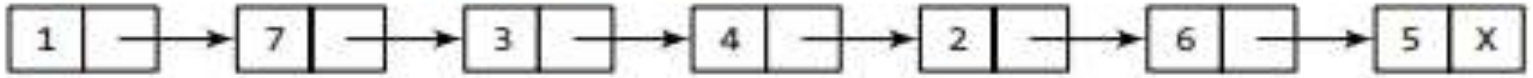
PREPTR

PTR

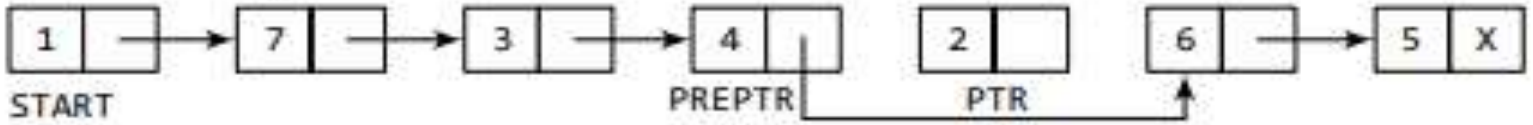
Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.







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



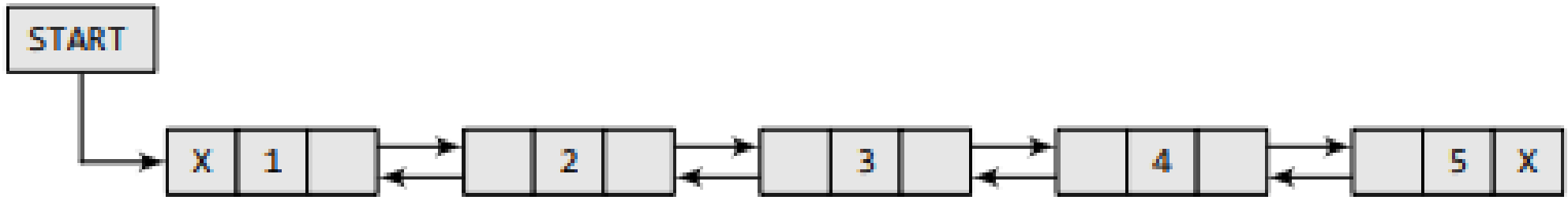
```
struct Node {
    int data;
    struct Node* next;
};
struct Node* newNode(int data)
{
    struct Node* node= (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}
struct Node* deleteNode(struct Node* head, int position)
{
    struct Node* temp = head;
    struct Node* prev = NULL;
    if (temp == NULL)
        return head;
    if (position == 1) {
        head = temp->next;
        free(temp);
        return head;
    }
    for (int i = 1; temp != NULL && i < position; i++) {
        prev = temp;
        temp = temp->next;
    }
    if (temp != NULL) {
        prev->next = temp->next;
        free(temp);
    }
    else {
        cout<<"Data not present";
    }
    return head;
}
```

```
void printList(struct Node* head)
{
    while (head != NULL) {
        cout<<head->data;
        head = head->next;
    }
    cout<<"NULL";
}
int main()
{
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    cout<<"Original list: ";
    printList(head);
    cout<<endl;
    int position = 2;
    head = deleteNode(head, position);

    cout<<"List after deletion: ";
    printList(head);
      cout<<endl;

    while (head != NULL) {
        struct Node* temp = head;
        head = head->next;
        free(temp);
    }

    return 0;
}
```

```
Original list: 12345NULL
List after deletion: 1345NULL
```

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



Structure of DLL

- In C/C++, the structure of a doubly linked list can be given as,

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

- START = 1, 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.
- Traverse the list until reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list. When traverse the DATA and NEXT in this manner, finally see that the linked list in the example stores characters that when put together form the word HELLO.

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

Inserting a New Node in a DLL

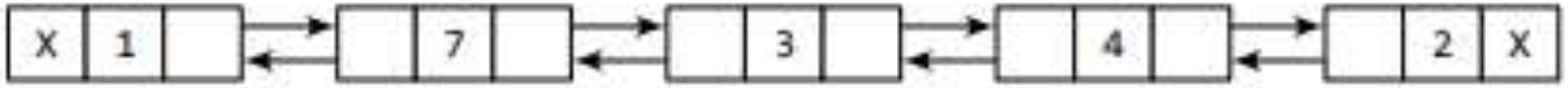
- Case 1: The new node is inserted at the beginning
- Case 2: The new node is inserted at the end.
- Case 3: The new node is inserted after a given node.
- Case 4: The new node is inserted before a given node

Case 1 :Inserting a Node at the Beginning of a DLL

In Step 1, 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 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.

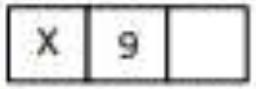
```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV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

- Consider the doubly linked list shown in Fig. . 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

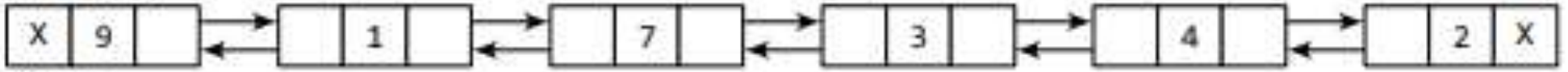


START

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



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



START

```
struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
void insertStart (struct Node **head, int data)
{
  struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));
  newNode->data = data;
  newNode->next = *head;
  newNode->prev = NULL;
  if (*head != NULL)
    (*head)->prev = newNode;
  *head = newNode;
}
void display (struct Node *node)
{
  struct Node *end;
  cout<<"List in Forward direction: ";
  while (node != NULL)
    {
      cout<<node->data<<"\t";
      end = node;
      node = node->next;
    }
    cout<<endl;
  cout<<"List in backward direction: ";
  while (end != NULL)
    {
      cout<<end->data<<"\t";
      end = end->prev;
    }
}
```

```
int main ()
{
  struct Node *head = NULL;
  insertStart (&head, 12);
  insertStart (&head, 16);
  insertStart (&head, 20);
  display (head);
  cout<<endl;
  return 0;
}
```

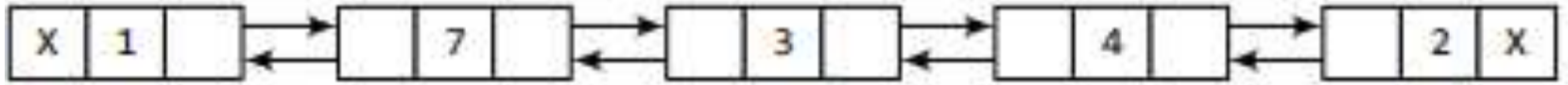
```
List in Forward direction: 20   16   12
List in backward direction: 12   16   20
```

Case 2: Inserting a Node at the End end of a DLL

In Step 6, take a pointer variable PTR and initialize it with START. In the while loop, traverse through the linked list to reach the last node. Once reach the last node, in Step 9, 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
```

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



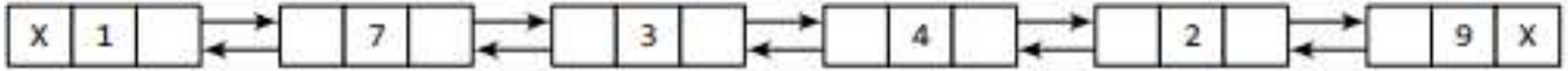
START

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

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

START,PTR

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



START

PTR

```
struct Node {
    int data;
    struct Node *next;
    struct Node *prev;
};
struct Node *createNode(int new_data) {
    struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    return new_node;
}
struct Node* insertEnd(struct Node *head, int new_data) {
    struct Node *new_node = createNode(new_data);
    if (head == NULL) {
        head = new_node;
    } else {
        struct Node *curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }
        curr->next = new_node;
        new_node->prev = curr;
    }

    return head;
}
```

```
void printList(struct Node *head) {
    struct Node *curr = head;
    while (curr != NULL) {
        cout<<curr->data<<"\t";
        curr = curr->next;
    }
    cout<<endl;
}

int main() {

    struct Node *head = createNode(1);
    head->next = createNode(2);
    head->next->prev = head;
    head->next->next = createNode(3);
    head->next->next->prev = head->next;
    cout<<"Original Linked List: ";
    printList(head);

    cout<<"Inserting Node with data 4 at the end: ";
    head = insertEnd(head, 4);
    printList(head);

    return 0;
}
```

```
Original Linked List: 1 2   3
Inserting Node with data 4 at the end: 1    2   3   4
```

Case 3: Inserting a Node After a Given Node in a DLL

In Step 5, 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, traverse through the linked list to reach the node that has its value equal to NUM. It need to reach this node because the new node will be inserted after this node. Once reach this node, 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
```

START

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.

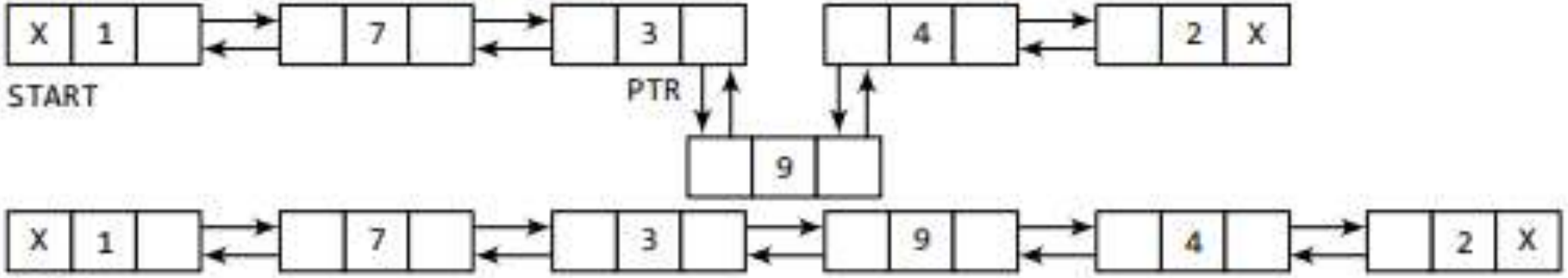
START,PTR

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

START

PTR

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



START

Case 4: Inserting a Node Before a Given Node in a DLL

In Step 1, first check whether memory is available for the new node. In Step 5, 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, traverse through the linked list to reach the node that has its value equal to NUM. need to reach this node because the new node will be inserted before this node. Once reach this node, change the NEXT and PREV fields 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: 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
```

START

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

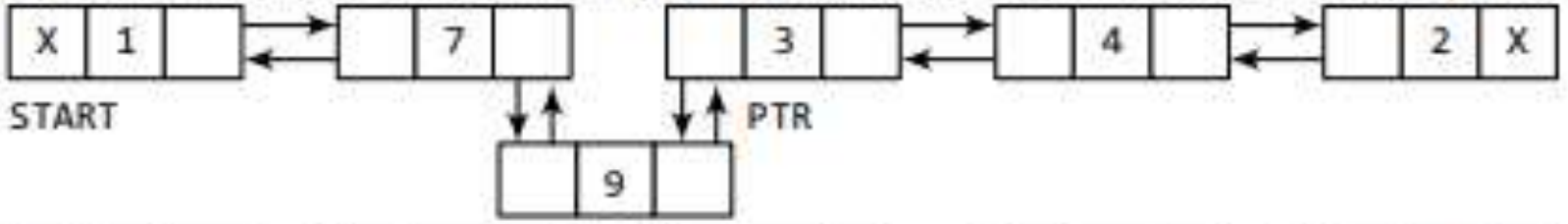
Take a pointer variable PTR and make it point to the first node of the list.

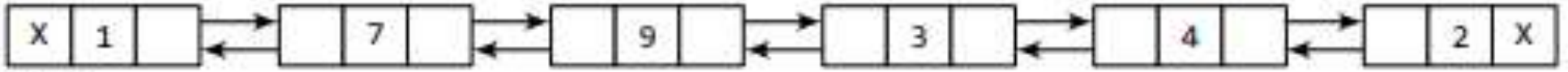
START, PTR

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.

START

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





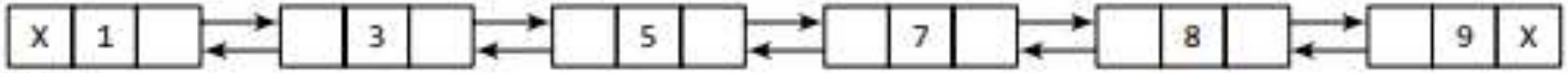
START

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

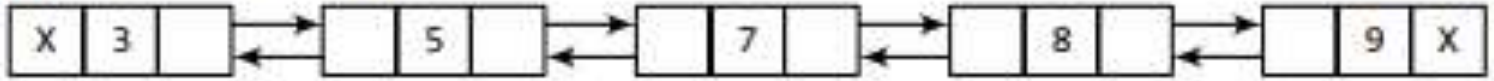
Case 1: The first node is deleted.

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



START

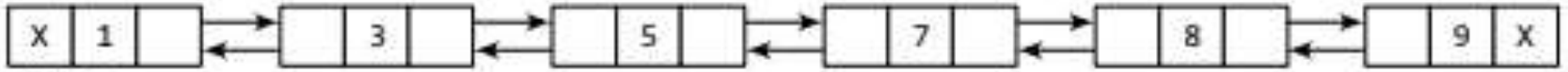
Free the memory occupied by the first node of the list and make the second node of the list as the START node.



START

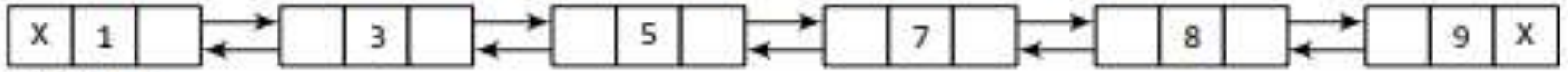
Case 2: The last node is deleted.

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



START

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



START,PTR

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

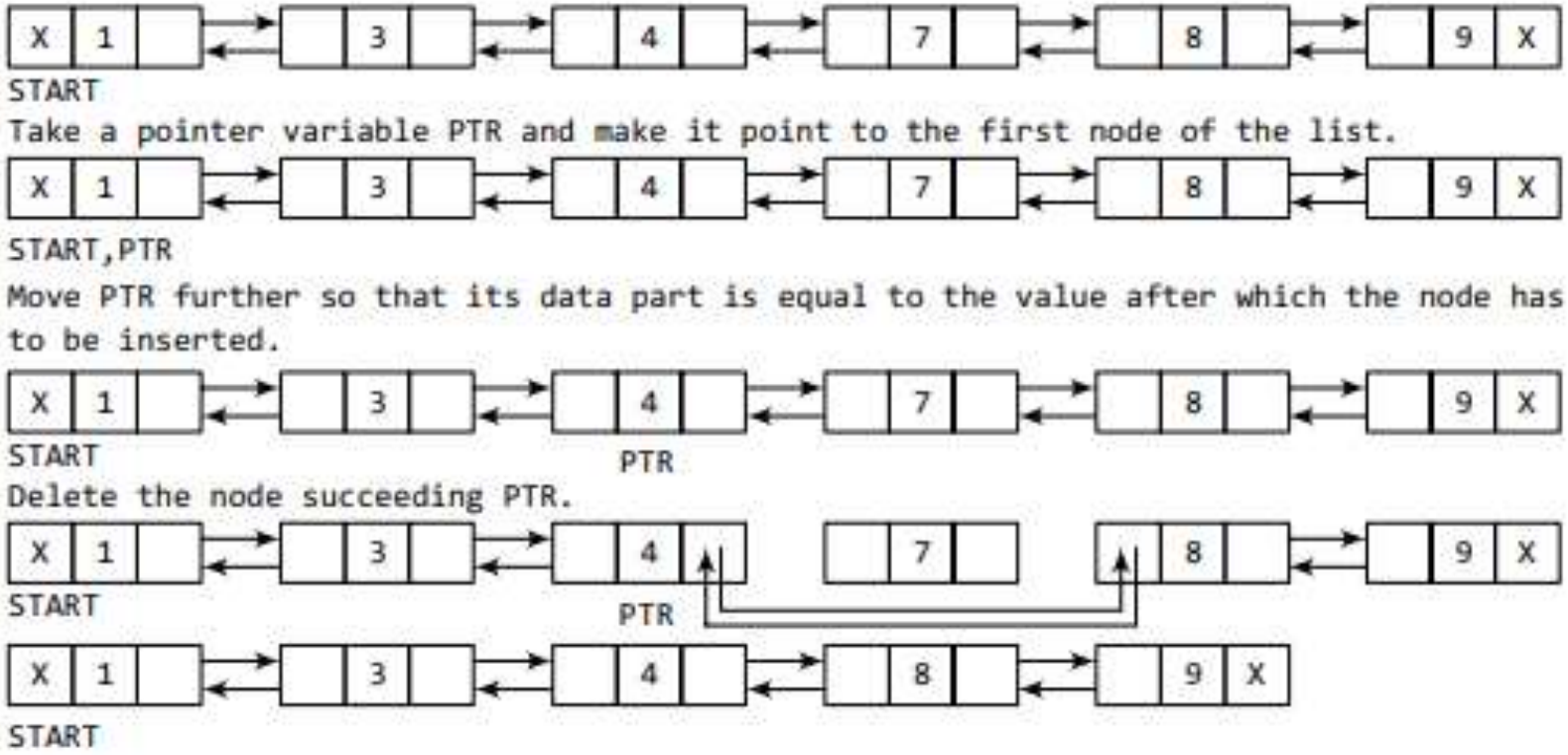
START PTR

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

START

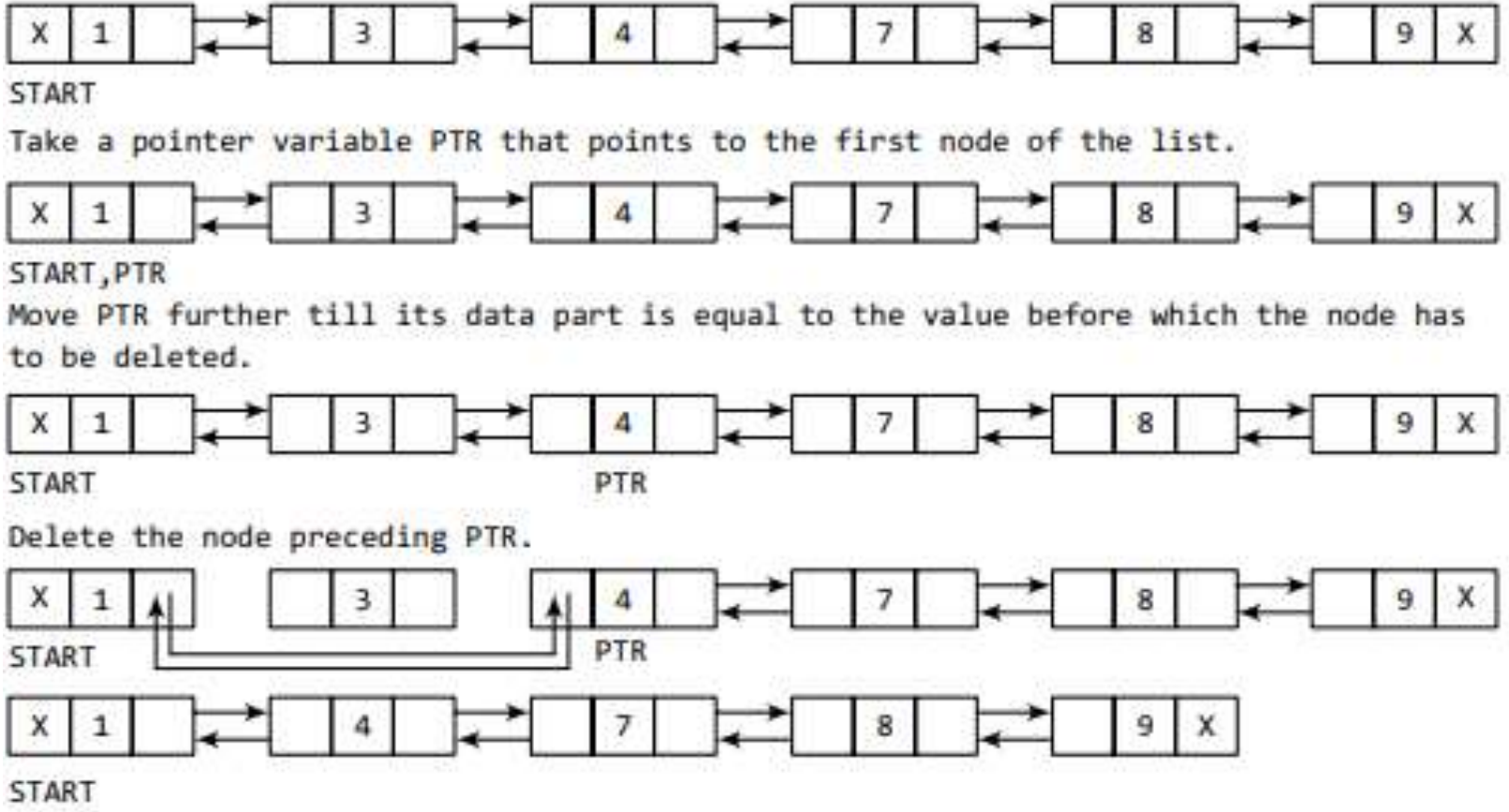
Case 3: The node after a given node is deleted.

```
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
```



Case 4: The node before a given node is deleted

```
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
```



Applications of Singly Linked Lists

- Implementation of stacks and queues
- Implementation of graphs: Adjacency list representation of graphs is the most popular which uses a linked list to store adjacent vertices.
- Dynamic memory allocation: use a linked list of free blocks.
- Maintaining a directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of the linked list
- Representing sparse matrices

Applications of Doubly Linked Lists

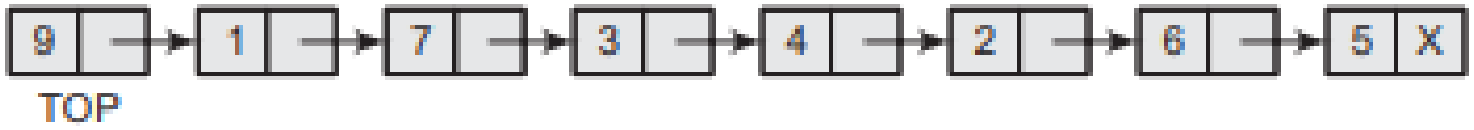
- Doubly linked list can be used in navigation systems where both forward and backward traversal is required.
- It can be used to implement different tree data structures.
- It can be used to implement undo/redo operations.
- A music player which has next and previous buttons.
- The Browser cache which allows to move front and back between pages.
- A deck of cards in a game

Stack implementation using Linked List

- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation is used.
- The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

Operations on a linked stack

- **Push Operation**:
- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.



- **Pop Operation:**

Operations on a linked stack

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

Queue implementation using Linked List

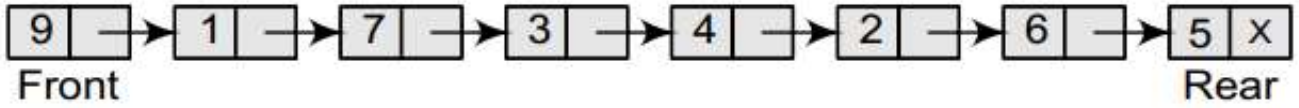
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

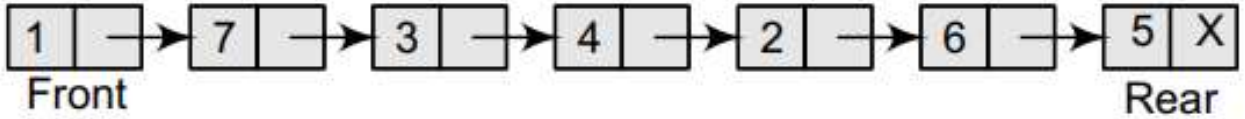
Linked Queue representation

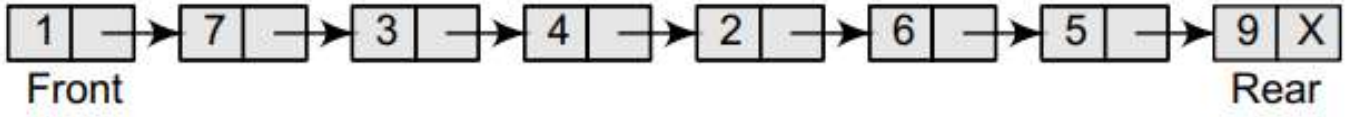
- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.

Operations on a linked queue

- **Insert Operation:**



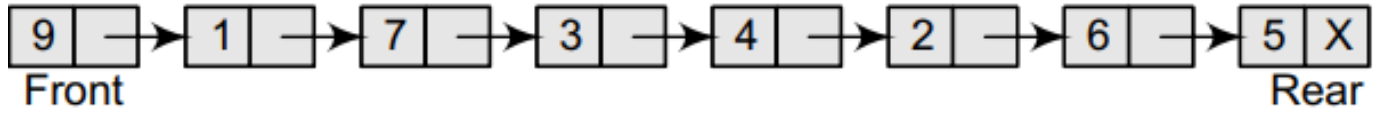


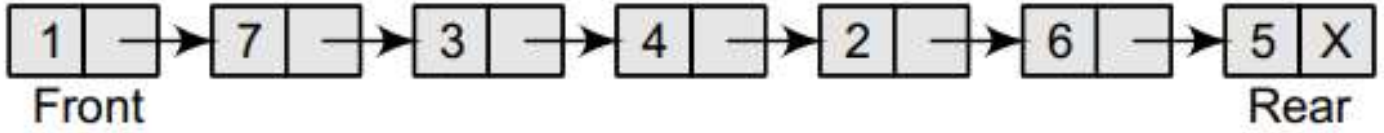



```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Operations on a linked queue

- **Delete Operation:**






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

DISCUSSION…

SVKM'S NMIMS
Deemed to be UNIVERSITY

THANK YOU