Unit II

Linked list:

→ A linked list is a collection of data elements called nodes in which linear representation is given by links from one node to next node.

→ It is a data structure that does not store its elements in consecutive memory locations and 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 elements can be accessed only in a sequential manner.

→ Like an array, insertions and deletions can be done at any point in the linked list.

Basic Terminology:

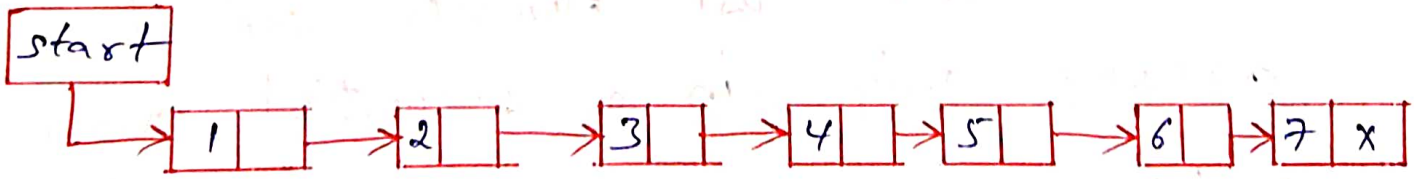
→ A linked list in simple terms is a linear collection of data elements.

→ 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 other variations.

→ A linked list can be perceived as a train or sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



Simple linked list

→ As seen above, every node in linked list contains two parts:

(i) The value of the node or data of the node

(ii) A pointer or link to the next node in the list.

→ 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".

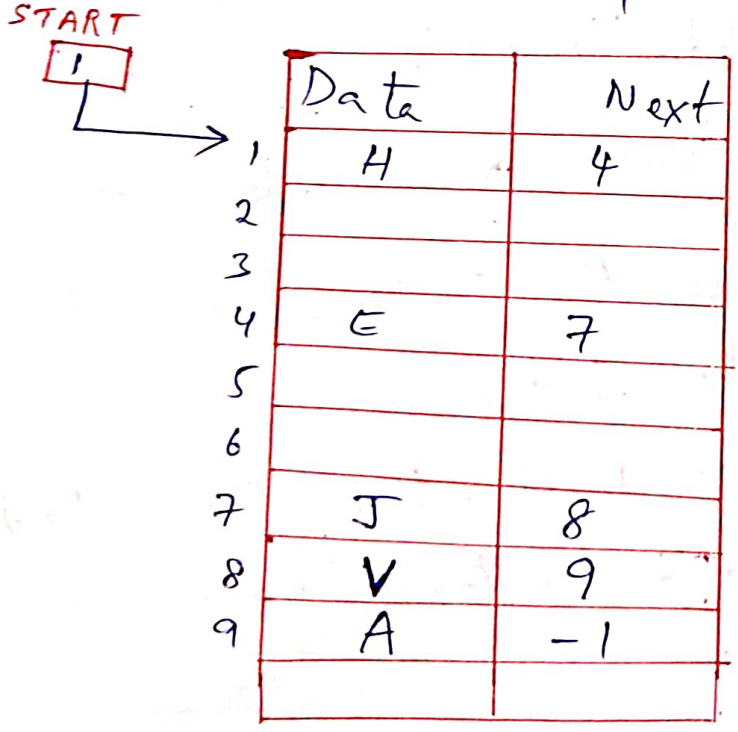
→ Linked list contains a pointer variable "START" that stores the address of the first node in the list.

→ If the START = NULL, then the linked list is empty and contains no nodes.

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

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

→ Linked list 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 point to the first element of the linked list in the memory

Advantages of linked list:

→ Dynamic Data structure: Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory.

→ Insertion and deletion of nodes are really easy.

→ No memory wastage

Disadvantages of linked list:

→ Pointers requires extra space

→ Linked list does not allow random access

→ Time consumed in traversing and changing the pointers.

Singly linked lists:

→ It is the simplest type of linked list in which every node contains some data and a pointer to the next node.

→ A singly linked list allows traversal of data only in one direction.

→ Figure in page 16 shows a singly linked list.

Traversing a linked list:

→ Means accessing the nodes of the list in order to perform some operations on them.

→ In addition to the first node pointer "START", we make use of one more pointer variable "TEMP", which points to the node that is currently being accessed.

Algorithm for traversing a linked list:

step 1: [Initialize] SET TEMP = START
step 2: Repeat steps 3 and 4 while TEMP != NULL
step 3: Apply Process to TEMP -> DATA
step 4: SET TEMP = TEMP -> NEXT
[END OF LOOP]
step 5: EXIT

Algorithm to print the number of nodes in a linked list:

step 1: [Initialize] SET COUNT = 0
step 2: [Initialize] SET TEMP = START
step 3: Repeat steps 4 and 5 while TEMP != NULL
step 4: SET COUNT = COUNT + 1
step 5: SET TEMP = TEMP → NEXT
[END OF LOOP]
step 6: Write COUNT
step 7: EXIT

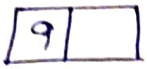
Singly linked list - Operations (Insertion)

1. Insertion at the beginning of the list.
2. The new node is inserted at the end.
3. Inserted after a given node.
4. Inserted before a given node.

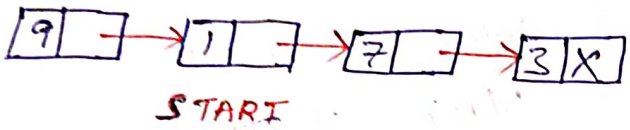
1. Insert a Node at the Beginning of a Linked list:

Consider the linked list shown above

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

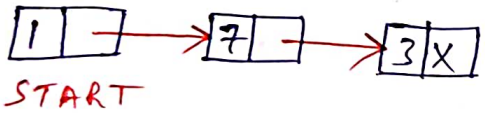
Creating singly linked list:

Singly linked list

```
# include<stdio.h>
# include<stdlib.h>
struct node
{
   int data;
   struct node *nt;
}*st, *nn, *tp;
int main ()
{
   int i,ch;
   do
{
nn = (struct node *)malloc(sizeof(struct node));
printf("Enter the data of new node: ");
scanf("%d", &nn->data);
if (st==NULL)
{
st=nn;
tp=nn;
}
```

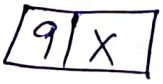
```
 else
 {
 tp->nt=nn;
 tp=tp->nt;
 tp->nt=NULL;
 }
 printf("Enter any non-zero value to add another node. Enter 0 otherwise: ");
 scanf("%d",&ch);
 }while(ch!=0);
 printf("The data elements of the linked list are: ");
 tp=st;
 while (tp!=NULL)
 {
 printf("%d\t",tp->data);
 tp=tp->nt;
 }
 return 0;
 }
```

Insert an element at the end of a linked list:



Consider the linked list shown above.

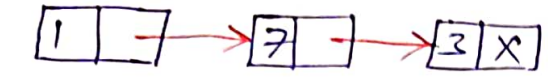
→ Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL



Take a pointer variable TEMP which points to START

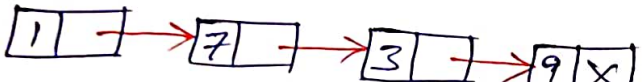
START, TEMP

Move TEMP so that it points to the last node of the list



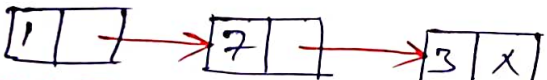
START TEMP

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



START TEMP

Inserting a node before a given node in a linked list

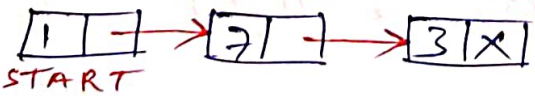


START

Consider the linked list shown above

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

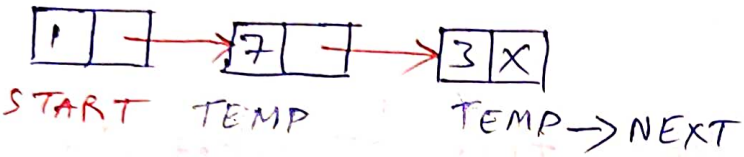
Initialize temporary variable TEMP to the START node



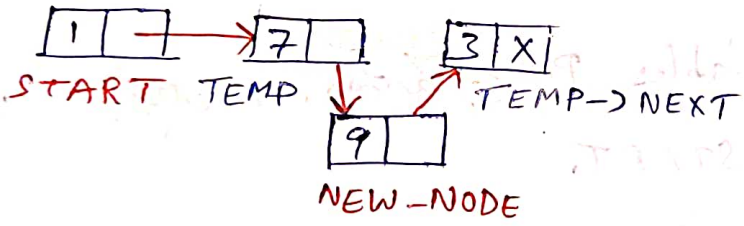
START TEMP

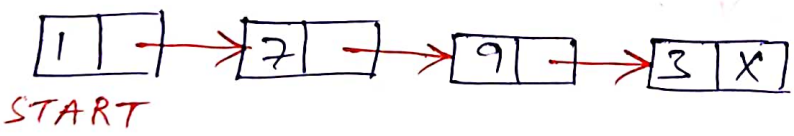
→ Mov TEMP → NEXT until the DATA part of TEMP→NEXT. equals value of node before which insertion has to be done.

→ TEMP should always point to the node just before TEMP→NEXT.



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





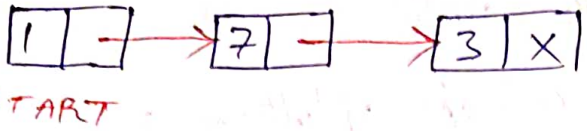
Deleting a Node from a Linked list:

Case 1: First Node is deleted

Case 2: Last node is deleted

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

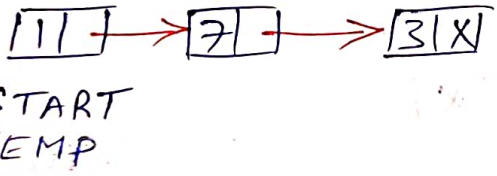
Deleting the first node



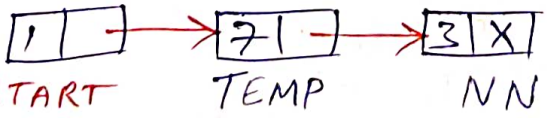
Make START to point to the next node in sequence

Deleting the last Node from a linked list

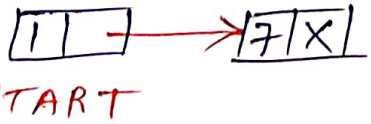
Take pointer variable TEMP (temparary) which initially point to START.



→ Move TEMP and NN such that NEXT part of NN = NULL.



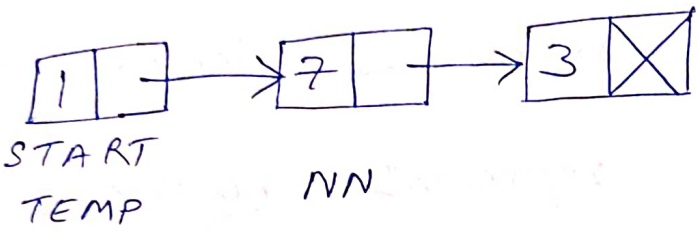
→ Set the NEXT part of TEMP node to NULL.



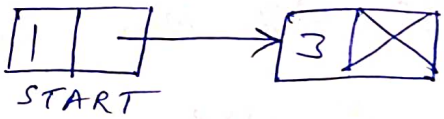
Delete a given element

In the previous linked list, to delete node with data 7,

→ Move TEMP and NN as shown below



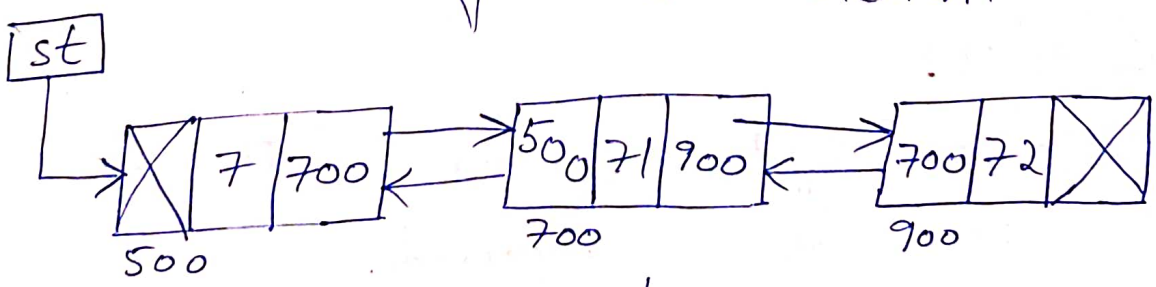
→ Set the next part of Temp to next part of NN.



Doubly linked list (DLL)

Each node contains 3 parts

(i) Address of Previous element
(ii) Data Element
(iii) Address of Next element



Example

Structure for Doubly linked list

```
struct node
{
int data;
struct node *next, *prev;
}*st, *tp, *nn;
```

Here st represents start,
tp represents temporary and
nn represents newnode.

Advantages of Doubly linked list

→ Dynamic Data structure; need not declare the number of elements at the beginning.
→ Easy to implement
→ Traversal in both directions
→ Reverse the list is simple and straight forward.

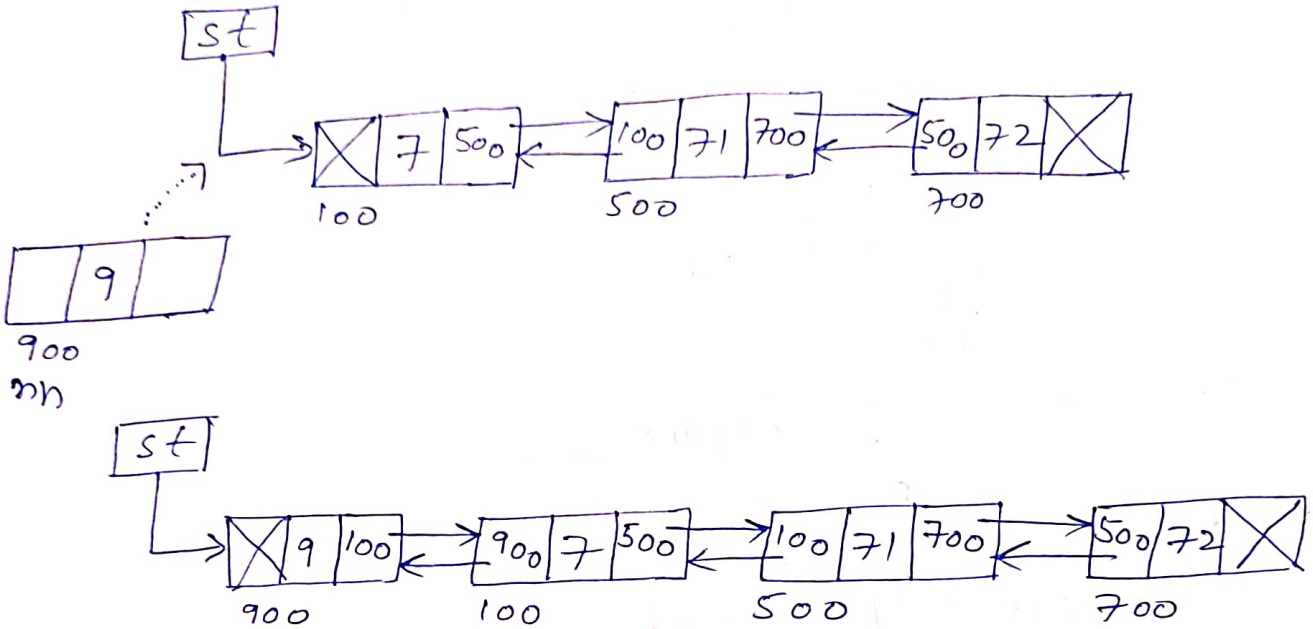
Applications

→ Navigation systems where both front and back navigation is required.
→ Used by browsers to implement backward and forward navigation of visited web pages.

→ To implement undo and redo. functionality.

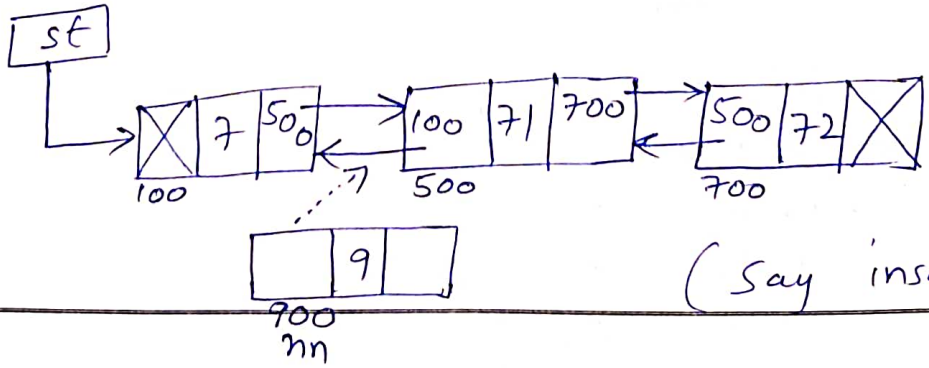
→ Used to represent various states of a game.

① Insert a new node at the beginning of a DLL




```
tp = st;
nn -> next = tp;
nn -> prev = NULL;
tp -> prev = nn;
st = nn;
```

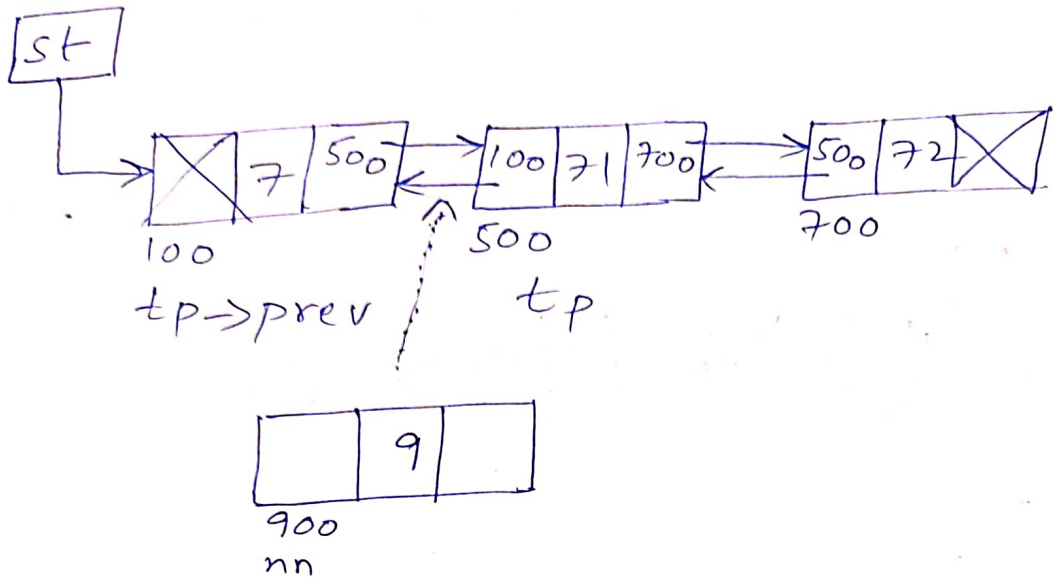
② Insert new node before a given element



(say insert 9 before 71)

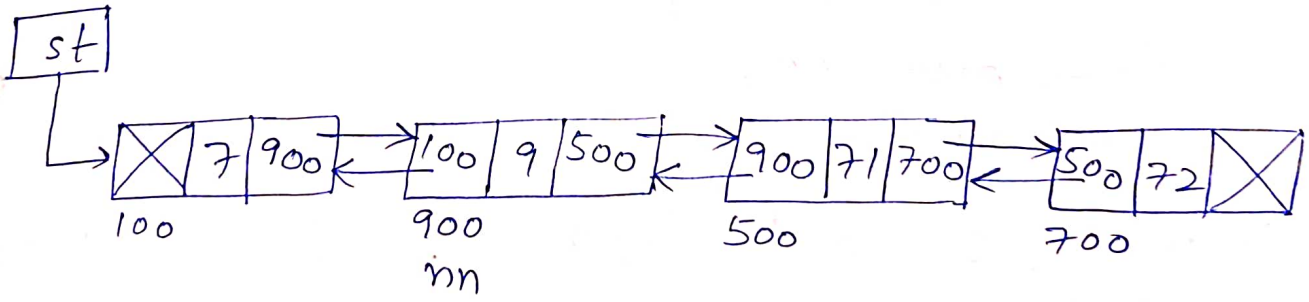
→ Position tp at 71

→ Then tp->prev points to node with data 7 as shown below



To insert 9 before 71

```
nn -> next = tp;
nn -> prev = tp -> prev;
tp -> prev -> next = nn;
tp -> prev = nn;
```



In the above case:

```
nn = 900
nn -> prev = 100
nn -> data = 9
nn -> next = 500
```

Try

(3) Insert a new node at end of DLL

(4) Insert new node after a given element

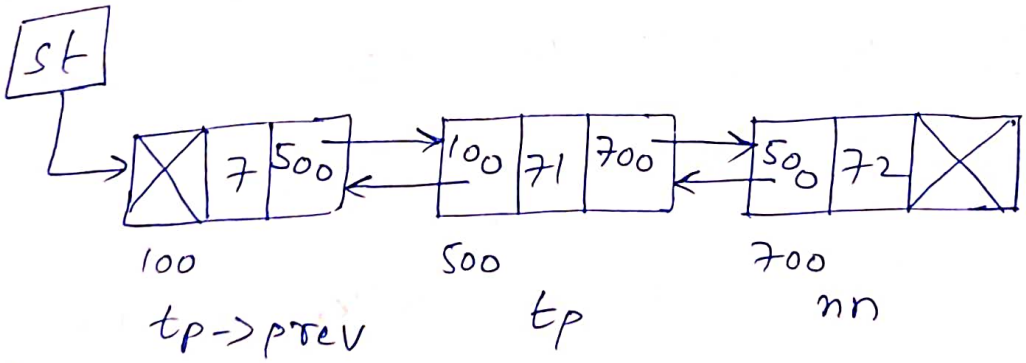
DLL – Deletion

(i) Delete a node at the beginning of DLL

```
tp = st;
st = st->next;
st->prev = NULL;
```

(ii) Delete a given node

say delete 71 in below DLL



→ First position tp to the selected node as shown above.

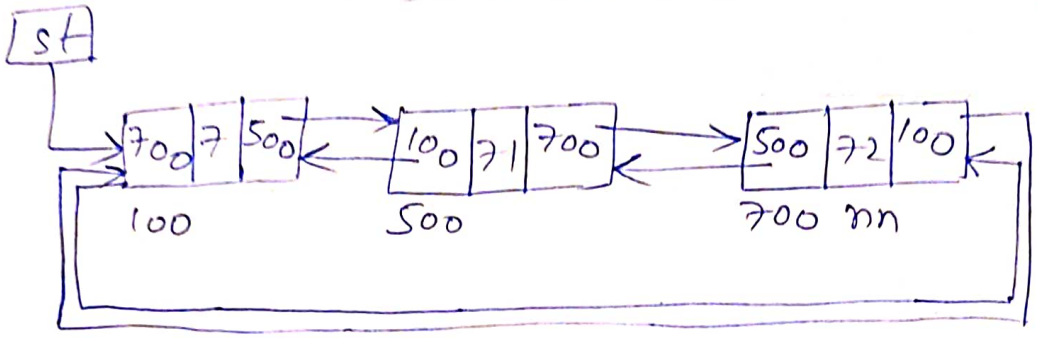
→ position nn next to tp.

→ tp-> prev is always before tp.

```
nn = tp -> next;
nn->prev = tp-> prev;
tp->prev->next = nn;
```

Try (iii) Delete at end

Circular doubly linked list (CDLL)



→ In a DLL when we create the additional links between the first and last node as shown above, we get a CDLL.

→ Useful to execute sequences repeatedly (ex: repeat song playlist)

Insert a new node before a given element

Same as that of DLL.

Insert a new node after a given element

same as that of DLL.

Try

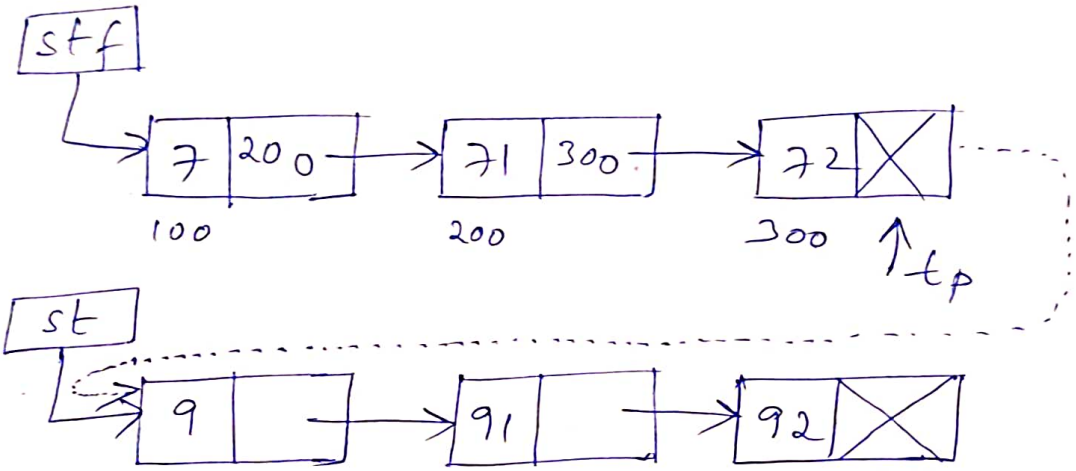
Insert at the end

Insert at beginning

Repeat insert at the end. Assign st = nn;

Try CDLL delete and CSLL (plan and program)

Concatenating two singly linked lists



→ Position tp as shown above after creating the above two linked lists.

→ use tp->next = st;