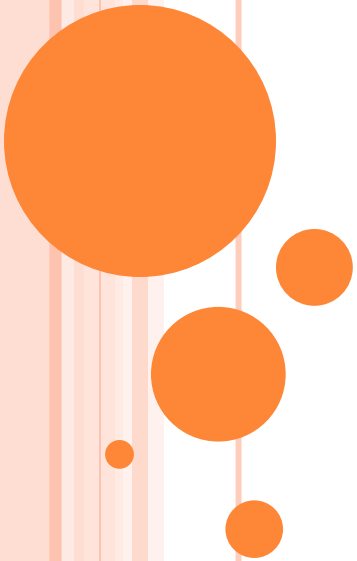


DATA STRUCTURES

LAB VII –Doubly Linked List, Circular Linked List.

OUTLINE

- Doubly Linked List.
- Circular Linked List.
- Exercise
- Prelab Questions

DOUBLY LINKED LIST

- In a singly linked list one can move from the header node to any node in one direction only (left-right).

- A doubly linked list is a two-way list because one can move in either direction. That is, either from left to right or from right to left.

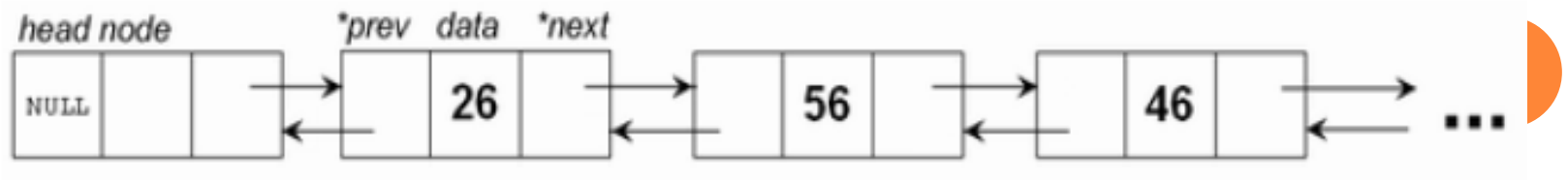
- It maintains two links or pointer. Hence it is called as doubly linked list.

PREV	DATA	NEXT

Structure of the node

- Where, DATA field - stores the element or data, PREV- contains the address of its previous node, NEXT- contains the address of its next node.

An example of a doubly linked list



Advantages and Disadvantages of Doubly Linked List

- Advantages:
  - We can navigate in both directions and delete the node even if we don't have the previous nodes address.
- Disadvantages:
  - Each node requires an extra pointer, requiring more space.
  - The insertion or deletion of a node takes a bit longer (more pointer operations).

DOUBLY LINKED LIST OPERATIONS

Insertion:

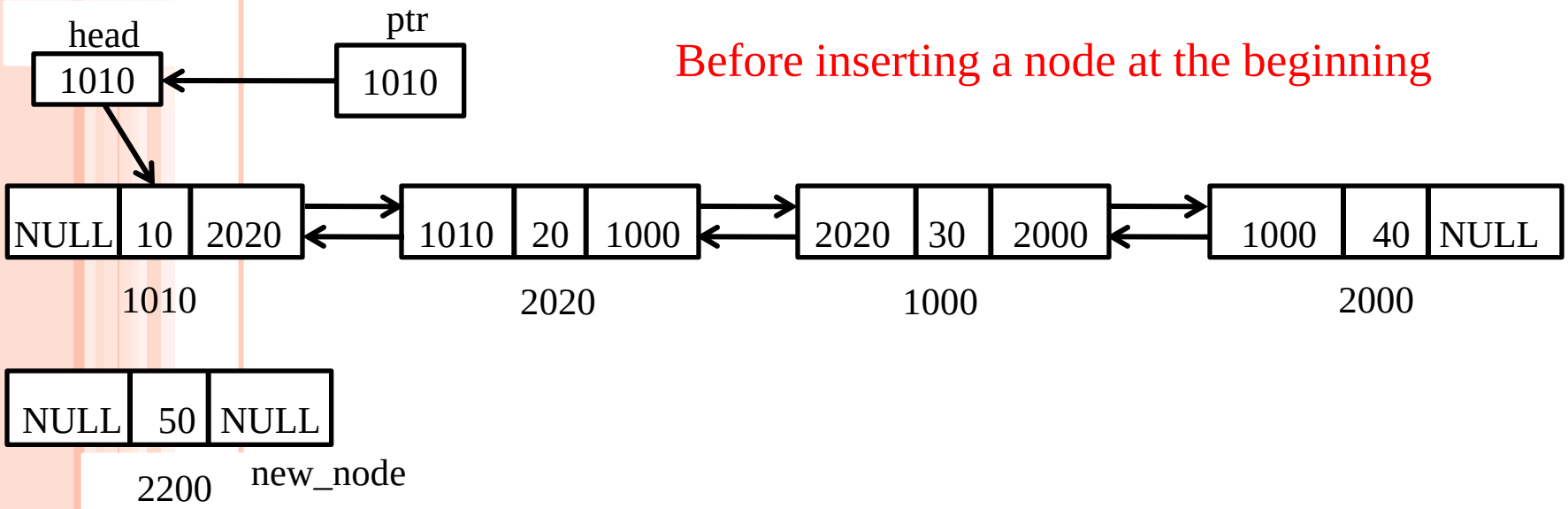
- **Insertion of a node at the front**
- **Insertion of a node at any position in the list**
- **Insertion of a node at the end**

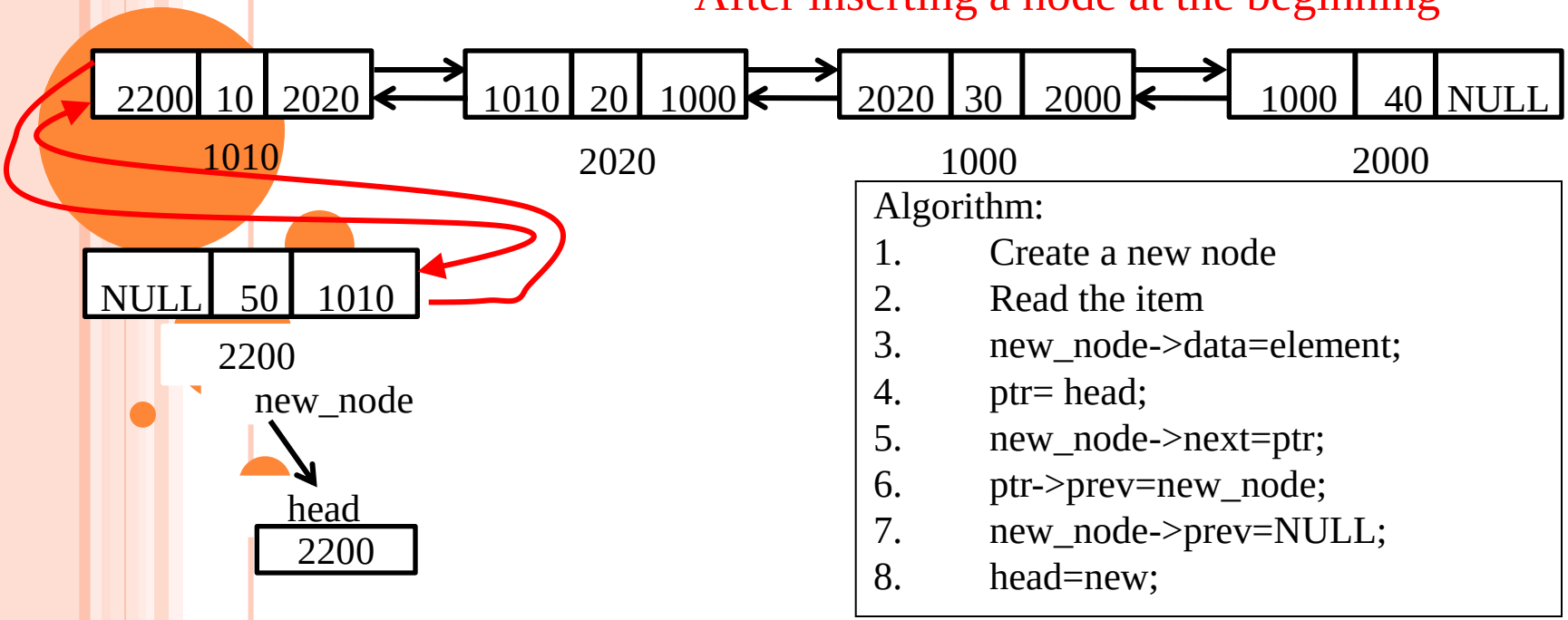
Deletion:

- **Deletion at front**
- **Deletion at any position**
- **Deletion at end**

Display:

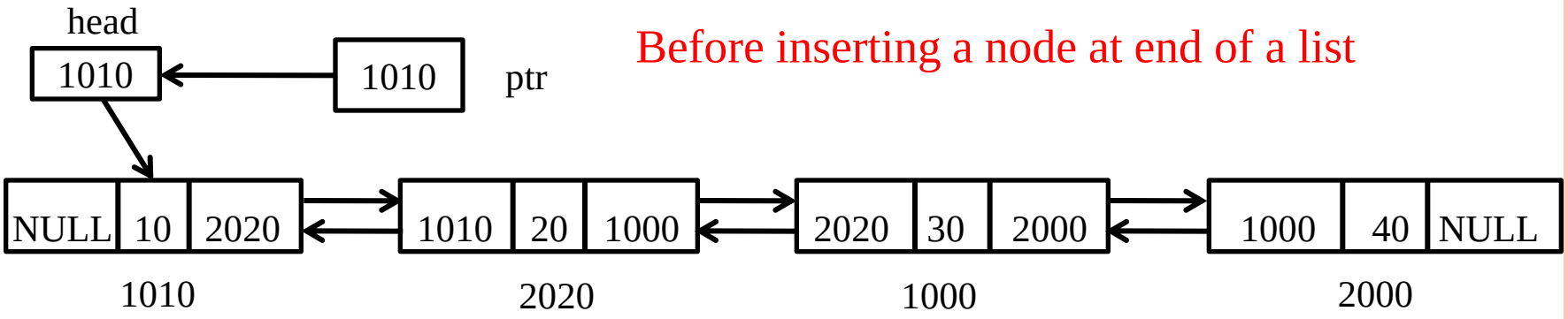
- **Displaying/Traversing the elements of a list**

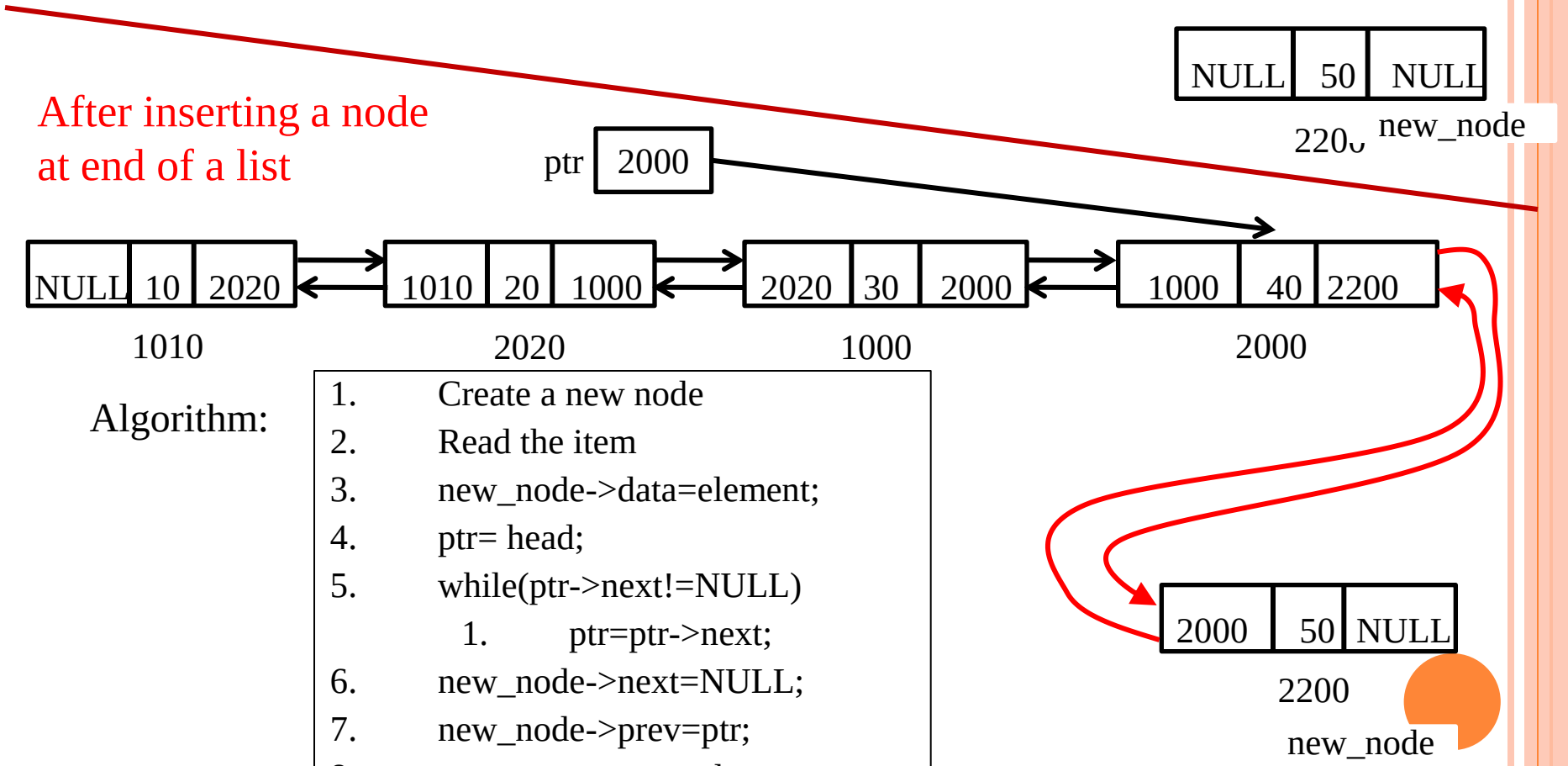




Algorithm:

1. Create a new node
2. Read the item
3. new_node->data=element;
4. ptr= head;
5. new_node->next=ptr;
6. ptr->prev=new_node;
7. new_node->prev=NULL;
8. head=new;





Algorithm:

1. Create a new node
2. Read the item
3. new_node->data=element;
4. ptr= head;
5. while(ptr->next!=NULL)
   1. ptr=ptr->next;
6. new_node->next=NULL;
7. new_node->prev=ptr;
8. ptr->next=new_node;

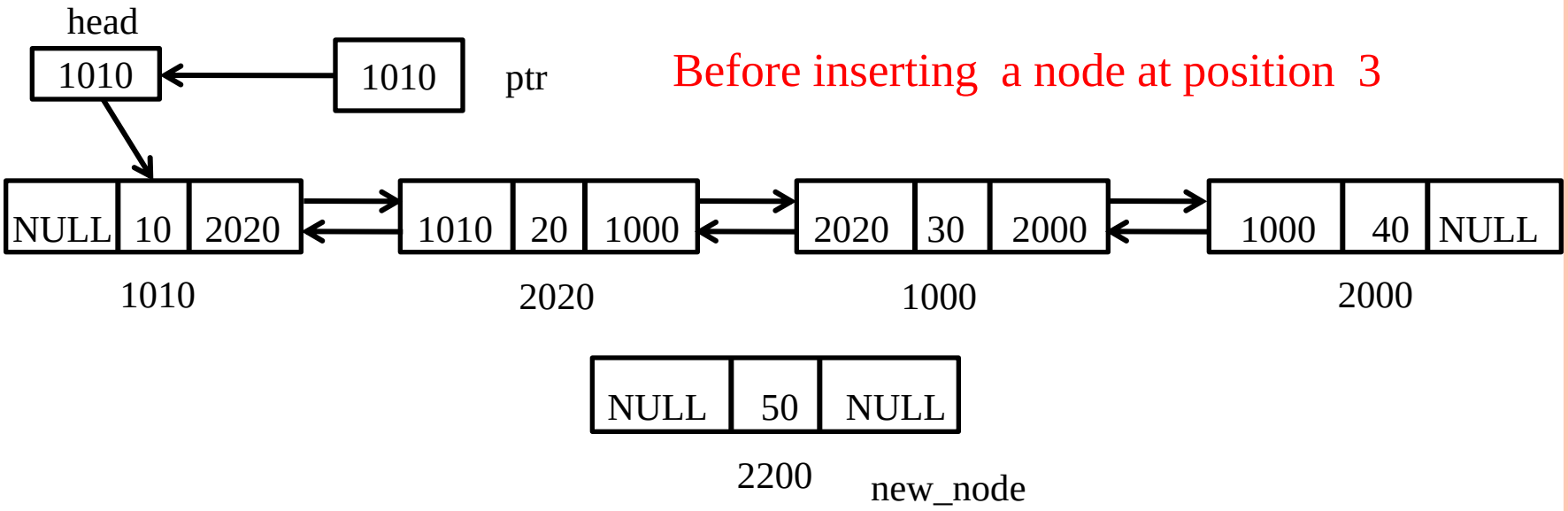
INSERTION OF A NODE AT ANY POSITION IN THE LIST

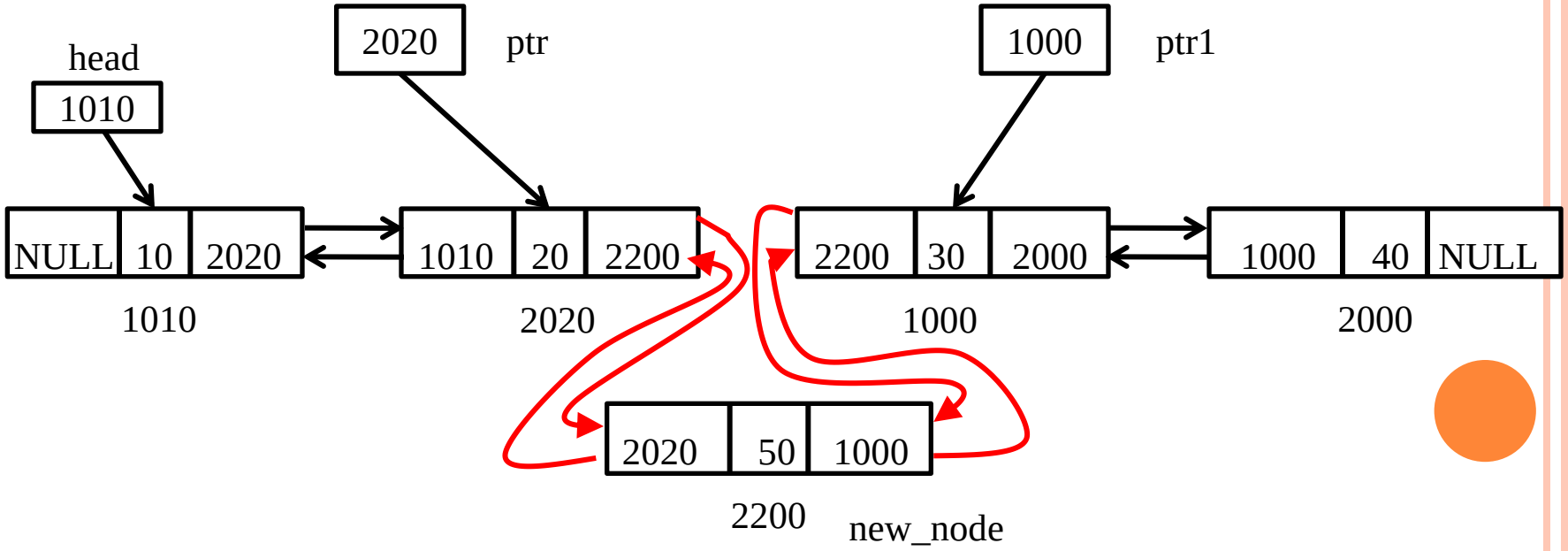
Algorithm:

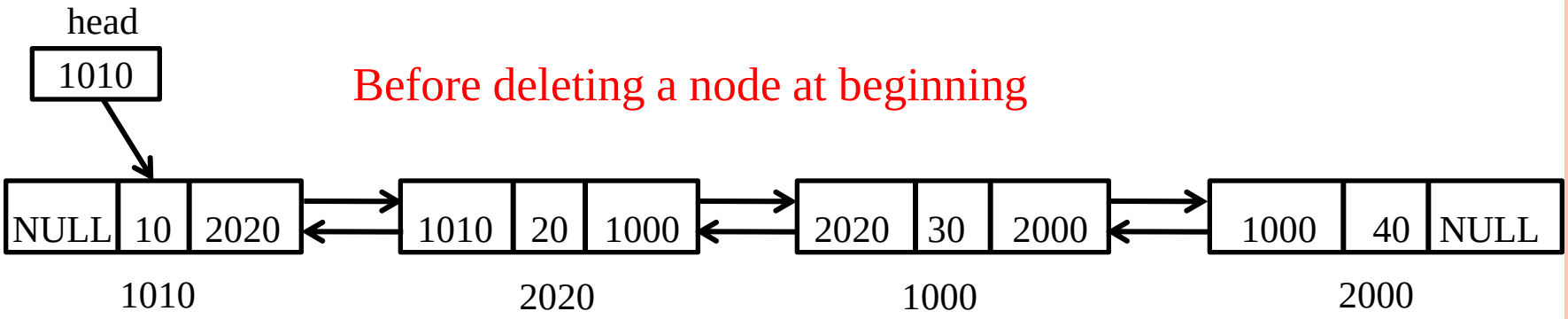
1. create a node new
2. read item
3. new_node->data=element;
4. ptr=head;
5. Read the position where the element is to be inserted
6. for(i=1;i<pos-1;i++)
   6.1 ptr=ptr->next;
7. if(ptr->next = = NULL)
   7.1 new_node->next = NULL;
   7.2 new_node->prev=ptr;
   7.3 ptr->next=new_node;
8. else
   8.1 ptr1=ptr->next;
   8.2 new_node->next=ptr1;
   8.3 ptr1->prev=new;
   8.4 new_node->prev=ptr;
   8.5 ptr->next=new_node;
9. end

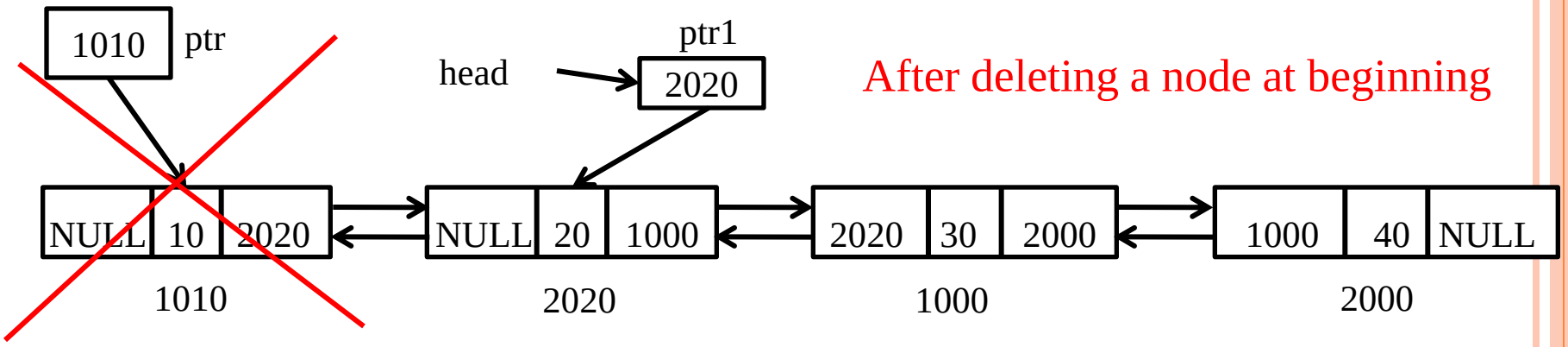
Time Complexity: O(n), since in the worst case, we may need to insert the node at the end of the list.

Space Complexity: O(1), for creating a temporary variable.


head
1010
1010
ptr
Before inserting a node at position 3
NULL
10
2020
1010
20
1000
2020
30
2000
1000
40
NULL
1010
2020
1000
2000
NULL
50
NULL
2200
new_node


After inserting a node at position 3
2020
ptr
1000
ptr1
head
1010
NULL
10
2020
1010
20
2200
2200
30
2000
1000
40
NULL
1010
2020
1000
2000
2020
50
1000
2200
new_node





Algorithm:

1.ptr=head;

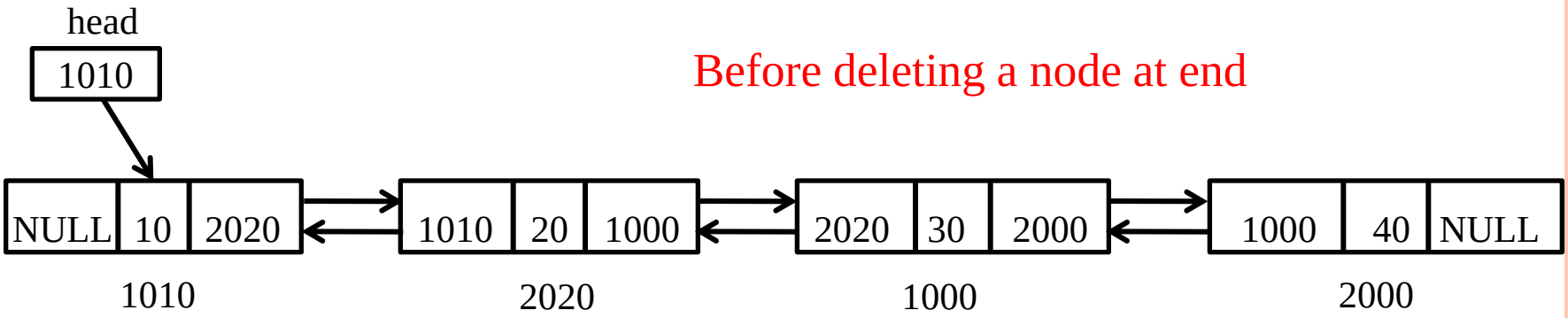
2.ptr1=ptr->next;

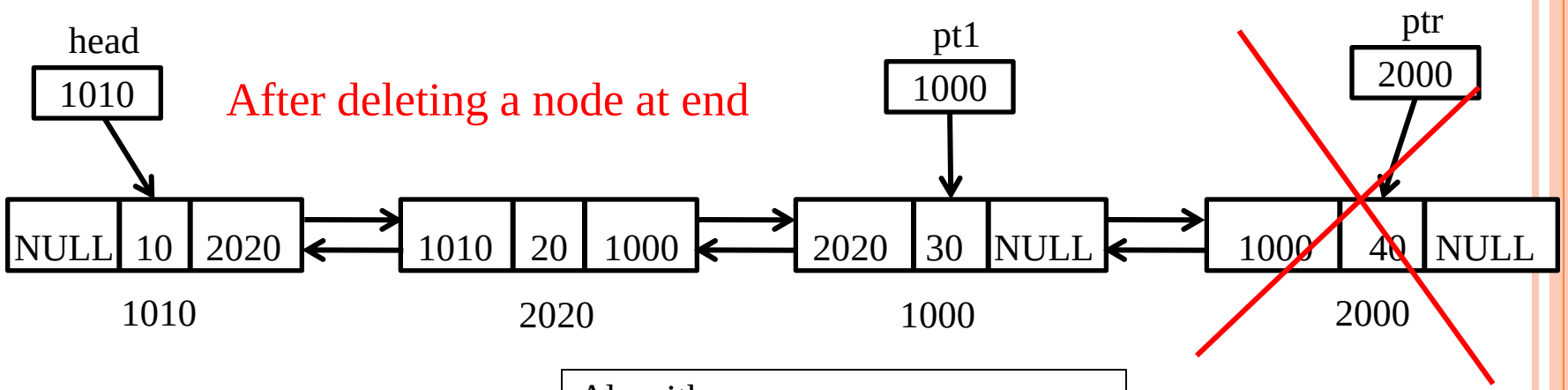
3.head=ptr1;

4.if(ptr1!=NULL)

    1.ptr1->prev=NULL;

5. free(ptr);





Algorithm:

1. ptr=head;
2. while(ptr->next!=NULL)
   1. ptr=ptr->next;
3. end while
4. ptr1=ptr->prev;
5. ptr1->next=NULL;

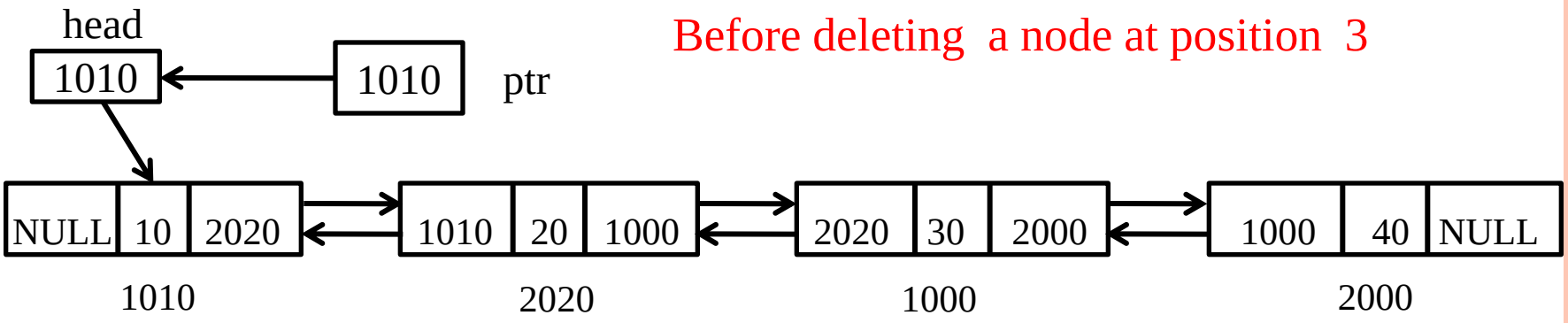
DELETION AT ANY POSITION

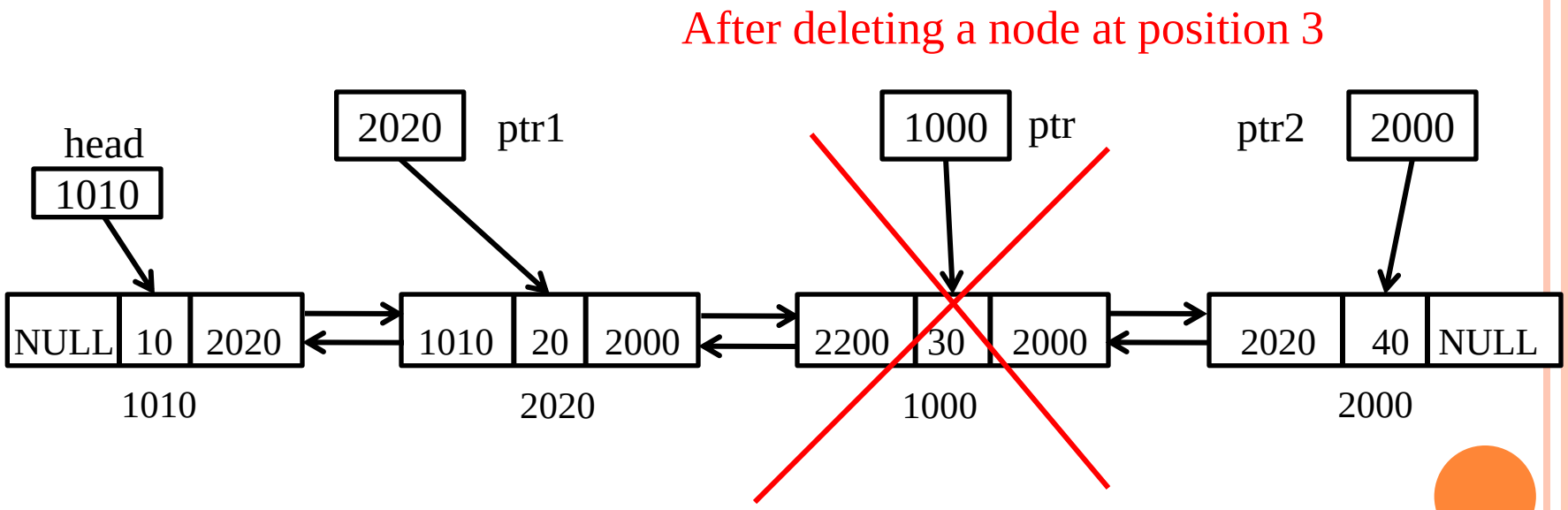
Algorithm:

1. ptr=head;
2. **while(ptr->next!=NULL)**
   1.for(i=0;i<pos-1;i++)
   1. ptr=ptr->next;
   2.if(i = = pos-1)
   1. break;
3. end while
4. if(ptr = = head)
   //if the deleted item is first node
   4.1 ptr1=ptr->next;
   4.2 ptr1->prev=NULL;
   4.3 head=ptr1;
   4.4 end if
5. else
   5.1 ptr1=ptr->prev;
   5.2 ptr2=ptr->next;
   5.3 ptr1->next=ptr2;
   5.4 ptr2->prev=ptr1;
6. end else
7. end if

Time Complexity: O(n), in the worst case, we may need to delete the node at the end of the list.

Space Complexity: O(1), for creating a temporary variable.

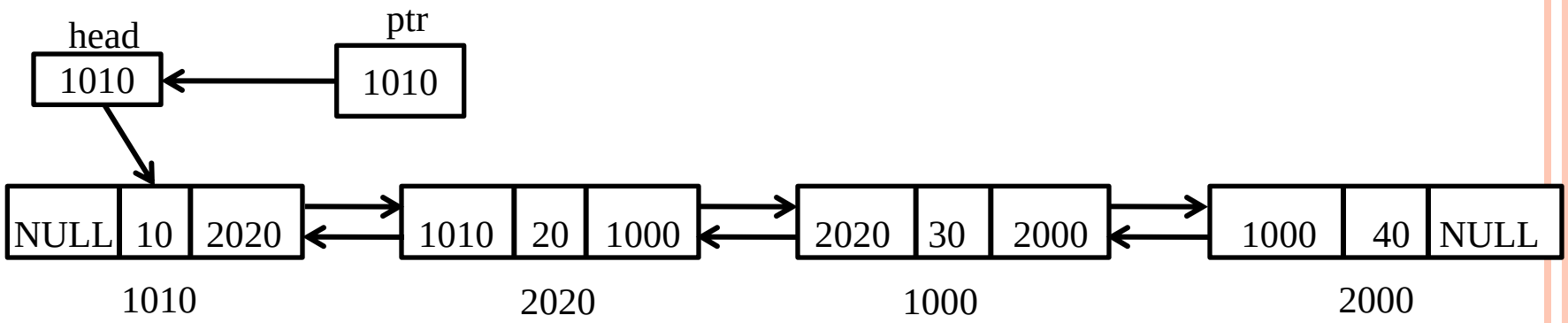

Before deleting a node at position 3
head
1010
1010
ptr
NULL
10
2020
1010
20
1000
2020
30
2000
1000
40
NULL
1010
2020
1000
2000


After deleting a node at position 3
head
1010
2020
ptr1
1000
ptr
ptr2
2000
NULL
10
2020
1010
20
2000
2200
30
2000
2020
40
NULL
1010
2020
1000
2000

Displaying elements of a list

Algorithm:

1. ptr=head;
2. if(head = = NULL)
    1. printf("The list is empty\n");
3. else
    1. print “The elements in farword order: “
    2. while(ptr!=NULL)
        1. print “ptr->data”;
        2. if(ptr->next = = NULL)
            1. break;
        3. ptr=ptr->next;
    3. print “The elements in reverse order: “
    4. while(ptr!=head)
        1. if(ptr->next = = NULL)
            1. print “ptr->data”;
        2. else
            1. print “ptr->data”;
            2. ptr=ptr->prev;
            3. print “ptr->data”;
        3.end else
4. end else

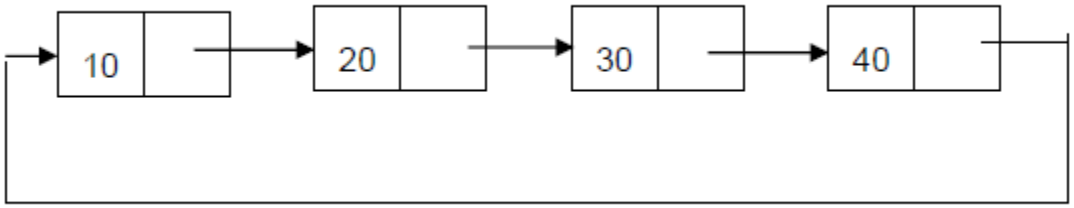


Forward Order : 10 20 30 40

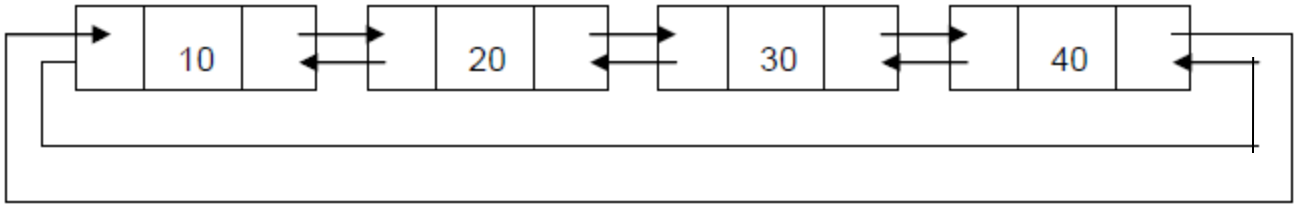
Reverse Order : 40 30 20 10

CIRCULAR LINKED LIST

- The linked list where the last node points the header node is called circular linked list.



Circular singly linked list



Circular doubly linked list

Applications of Circular Singly Linked List

- **Advantages of Circular Linked Lists:**
  - Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
  - Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
  - Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

Circular Singly linked list operations

Insertion:

- **Insertion of a node at the front**
- **Insertion of a node at any position in the list**
- **Insertion of a node at the end**

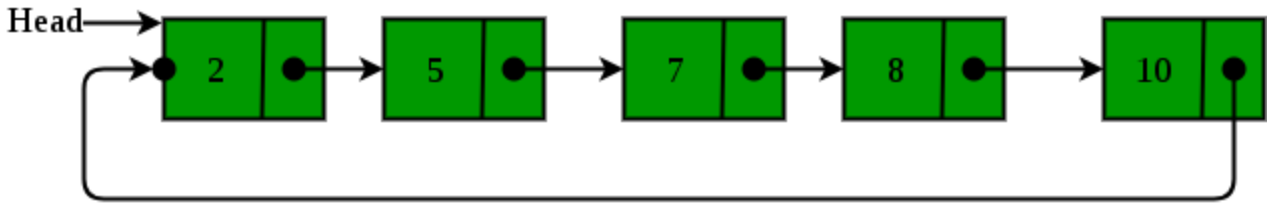
Deletion:

- **Deletion at front**
- **Deletion at any position**
- **Deletion at end**

Display:

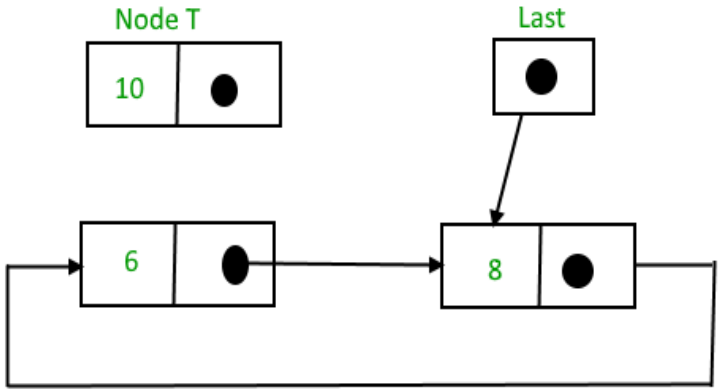
- **Displaying/Traversing the elements of a list**

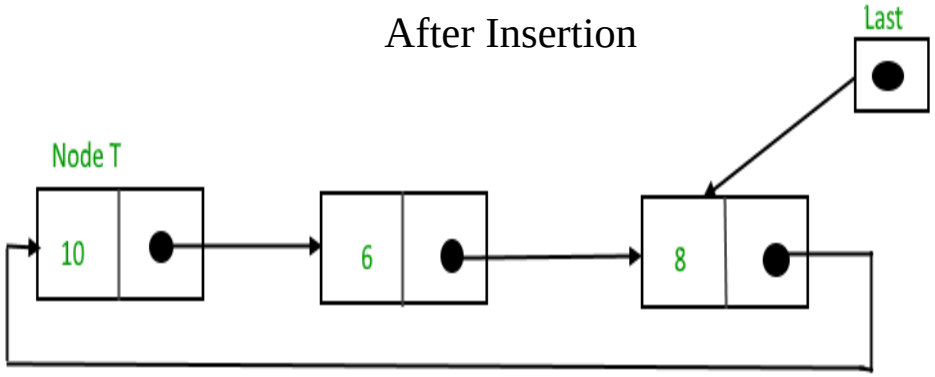
Displaying/Traversing the elements of a list




```
/* Function to traverse a given Circular linked list and print nodes */
void printList(struct Node *first)
{
        struct Node *temp = first;
        // If linked list is not empty
        if (first != NULL)
        {
                // Keep printing nodes till we reach the first node again
                do
                {
                        printf("%d ", temp->data);
                        temp = temp->next;
                }
                while (temp != first);
        }
}
```

INSERTION AT THE BEGINNING OF THE LIST

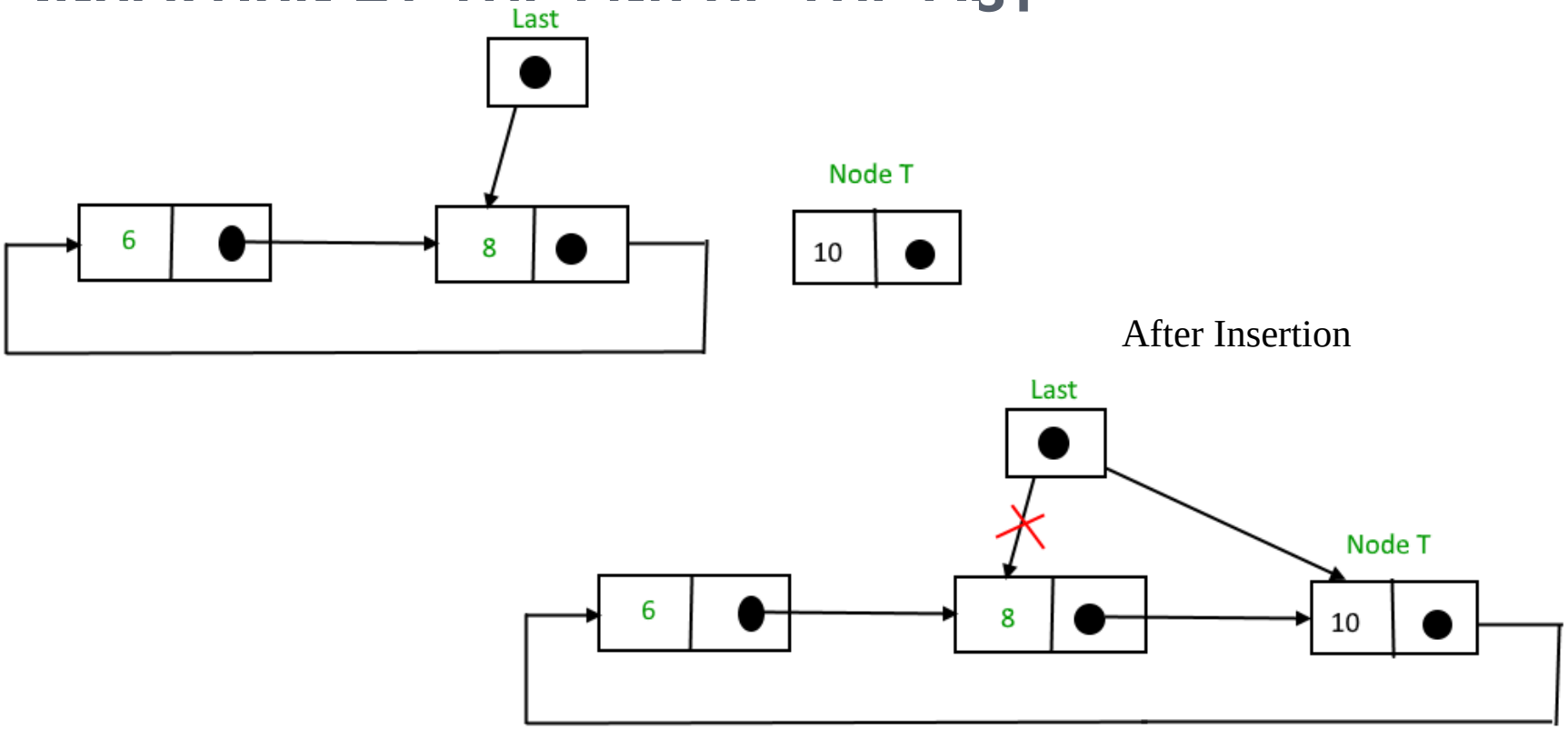




To Insert a node at the beginning of the list, follow these step:

1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T

Insertion at the end of the list



To Insert a node at the end of the list, follow these step:

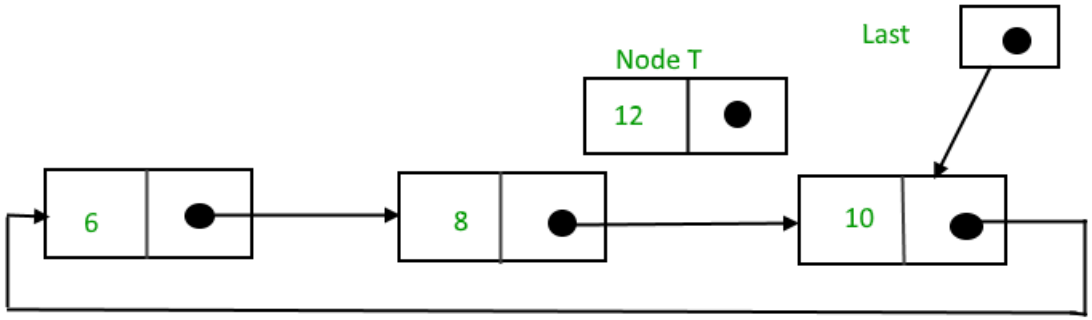
1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T

INSERTION IN BETWEEN THE NODES

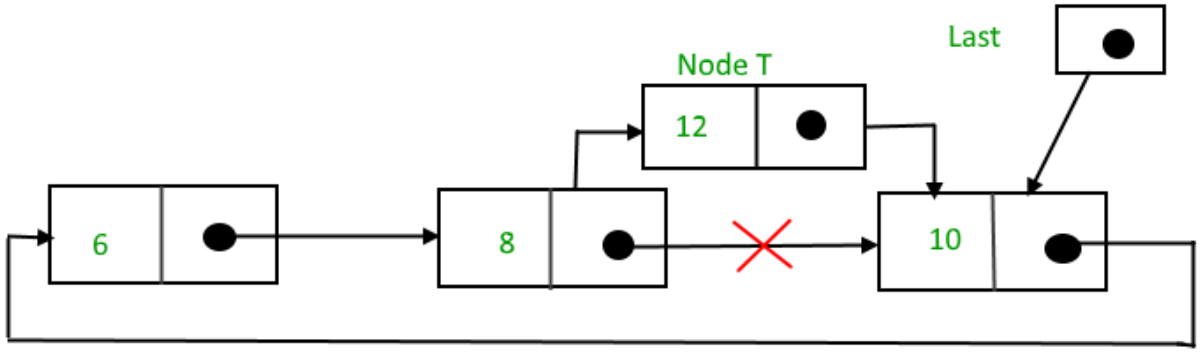
To Insert a node at the end of the list, follow these step:

1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

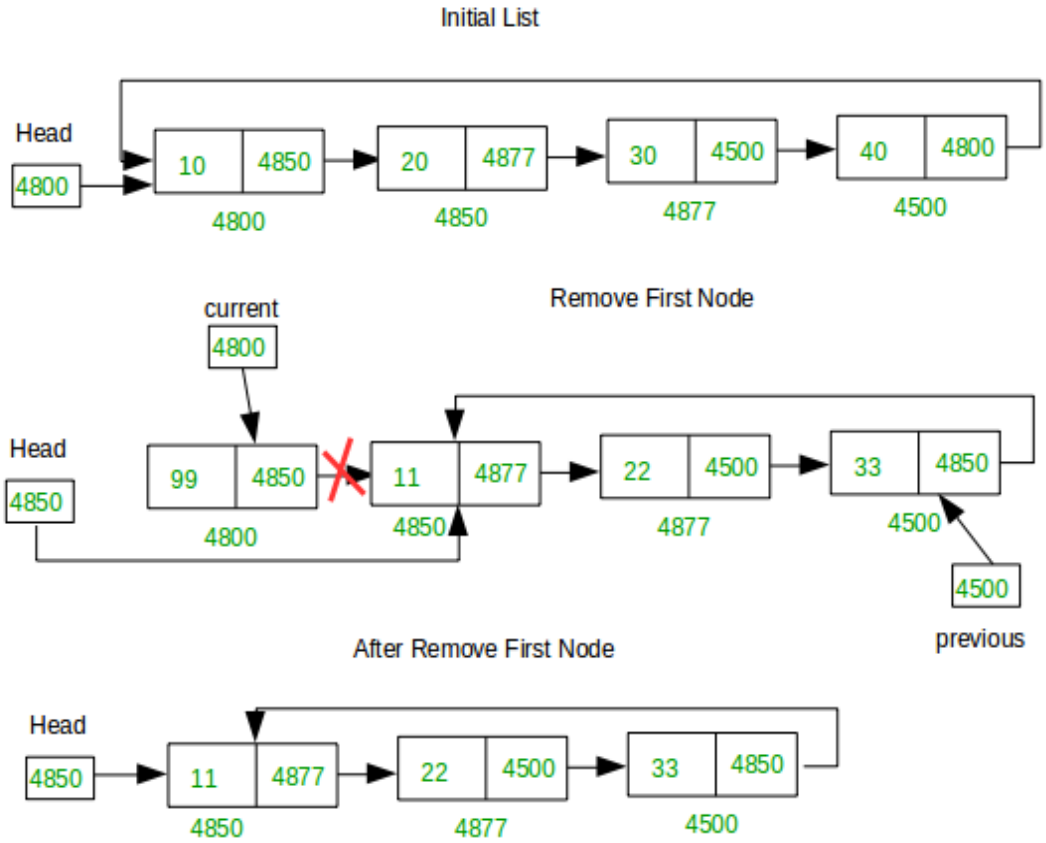
Suppose 12 need to be insert after node having value 10,



After Insertion



Deletion of CSLL



Delete first node

Approach:

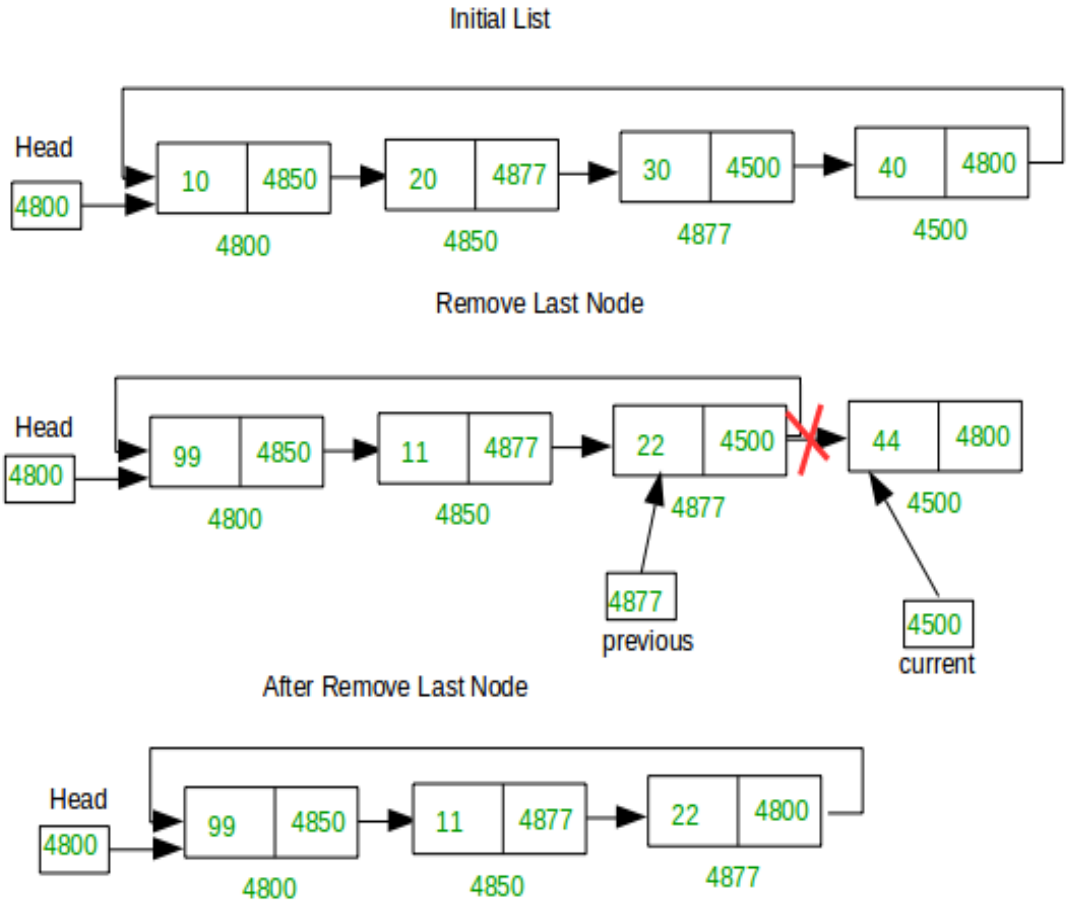
1.Take two pointers current and previous and traverse the list.

2.Keep the pointer current fixed pointing to the first node and move previous until it reaches the last node.

3.Once, the pointer previous reaches the last node, do the following:

1.previous->next = current-> next

2.head = previous -> next;



Delete last node

Approach:

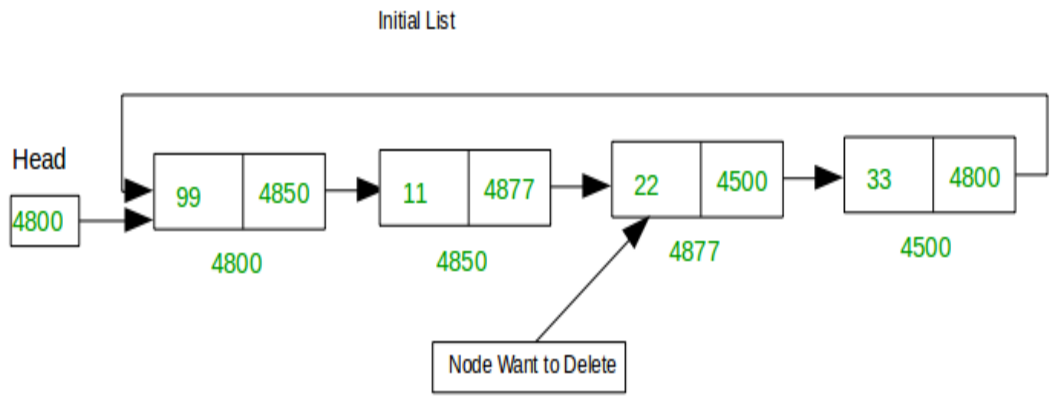
1.Take two pointers current and previous and traverse the list.

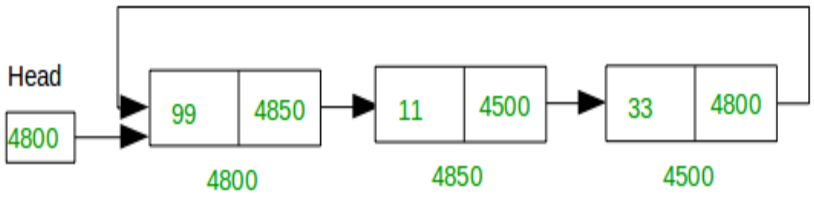
2.Move both pointers such that next of previous is always pointing to current. Keep moving the pointers current fand previous until current reaches the last node and previous is at second last node.

3.Once, the pointer current reaches the last node, do the following:

1.previous->next = current-> next

2.head = previous -> next;





After Delete index 2 element

Delete node at a particular position

Approach:

1.First, find the length of the list. That is, the number of nodes in the list.

2.Take two pointers previous and current to traverse the list. Such that previous is one position behind the current node.

3.Take a variable count initialized to 0 to keep track of the number of nodes traversed.

4.Traverse the list until the given index is reached.

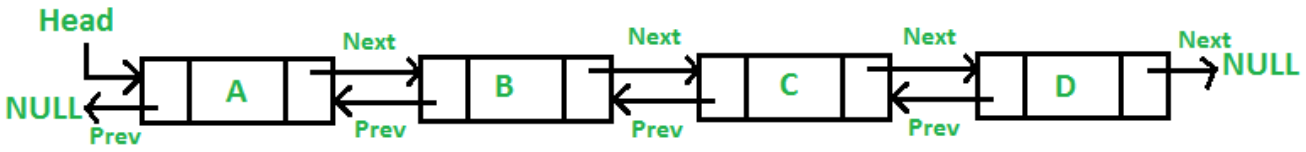
5.Once the given index is reached, do previous->next = current->next.

Complexities of CSLL

	Time Complexity	Space Complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Traversing	O(n)	O(1)
Searching	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of Linked List	O(1)	O(1)

Circular Doubly linked list

- A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Advantages and Disadvantages of DLL

- **Advantages over singly linked list:**
  1) A DLL can be traversed in both forward and backward direction.
  2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
  3) We can quickly insert a new node before a given node.
  In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.
- **Disadvantages over singly linked list**
  1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though.
  2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

Circular Doubly linked list operations

Insertion:

- **Insertion of a node at the front**
- **Insertion of a node at any position in the list**
- **Insertion of a node at the end**

Deletion:

- **Deletion at front**
- **Deletion at any position**
- **Deletion at end**

Display:

- **Displaying/Traversing the elements of a list**

INSERTION

A node can be added in four ways

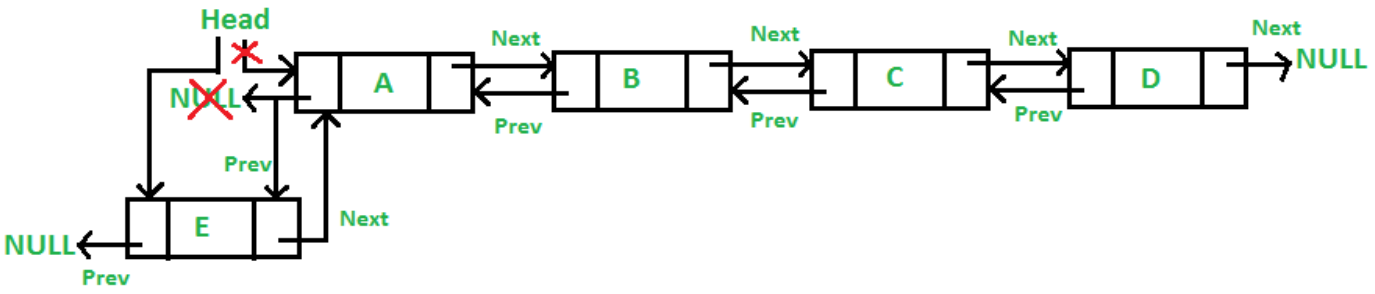
1) At the front of the DLL

2) After a given node.

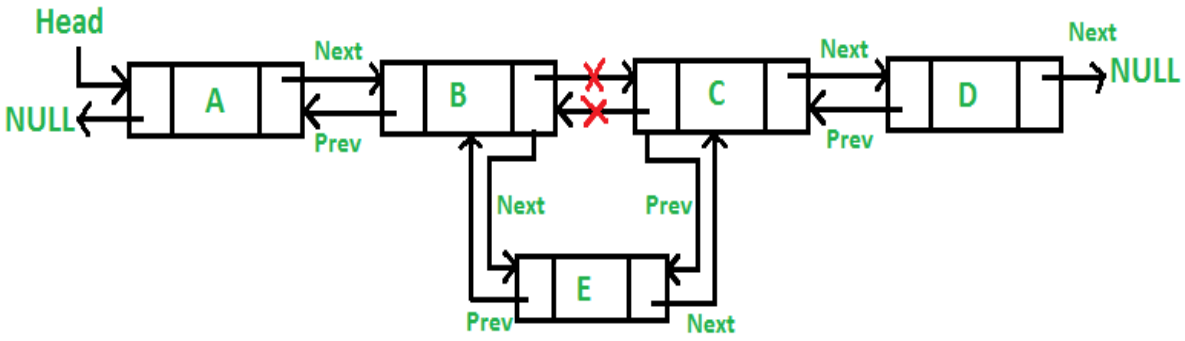
3) At the end of the DLL

4) Before a given node

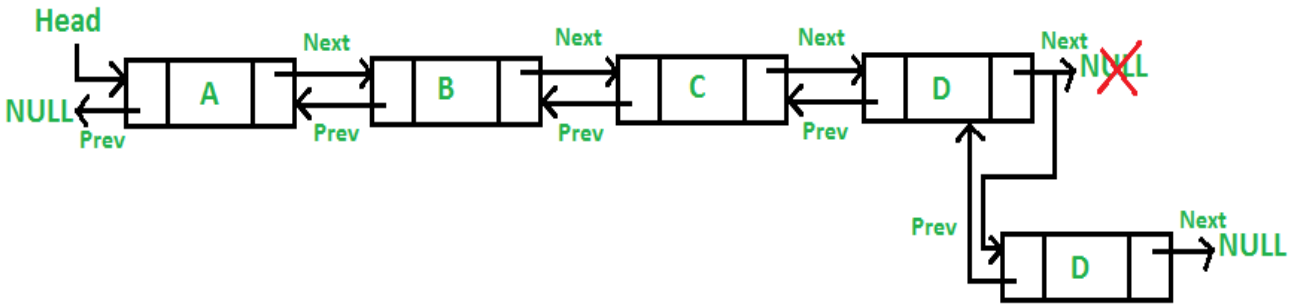
At the front of the DLL



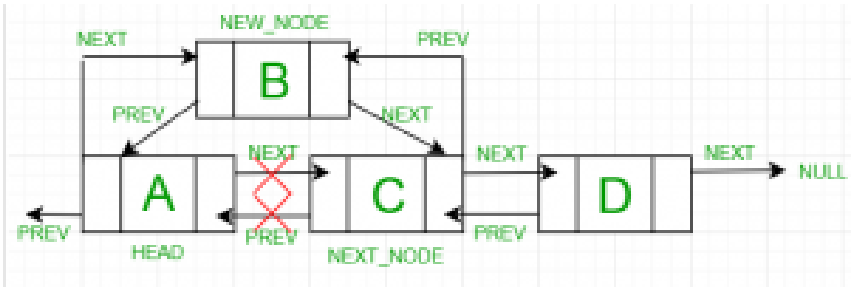
After a given node of the DLL



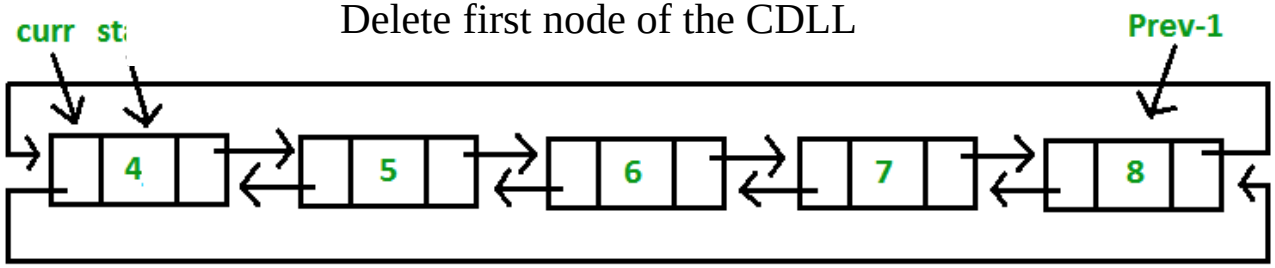
At the end of the DLL

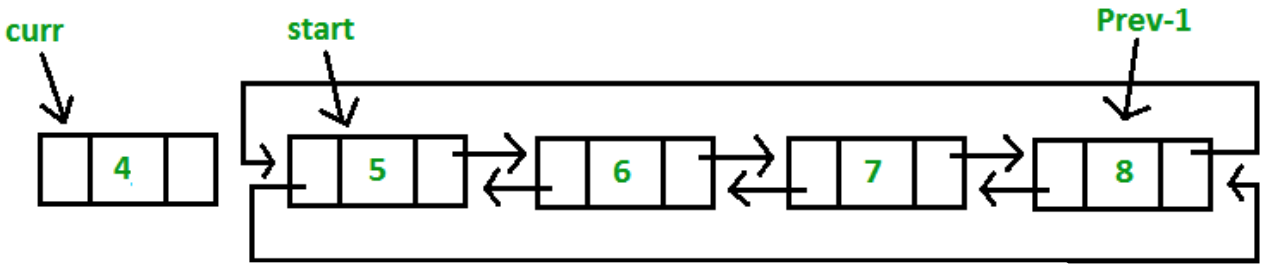


Before a given node of the DLL

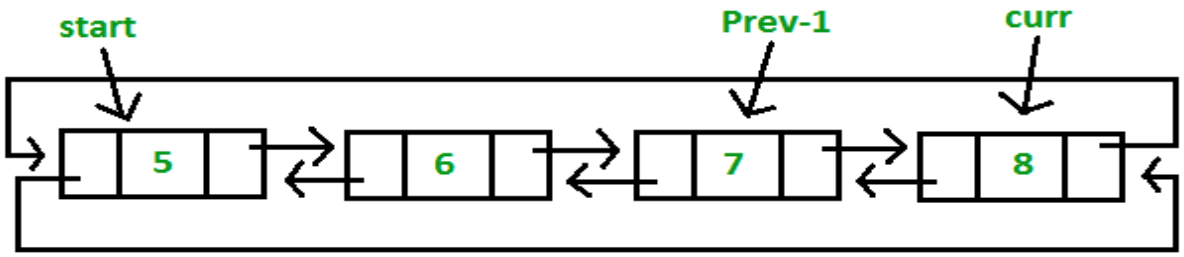


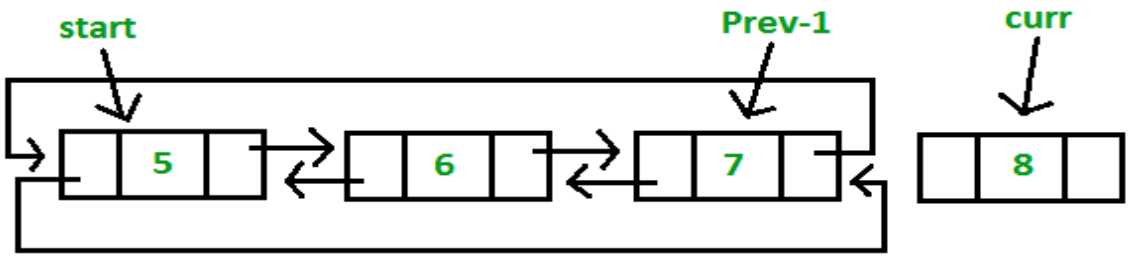
Delete first node of the CDLL



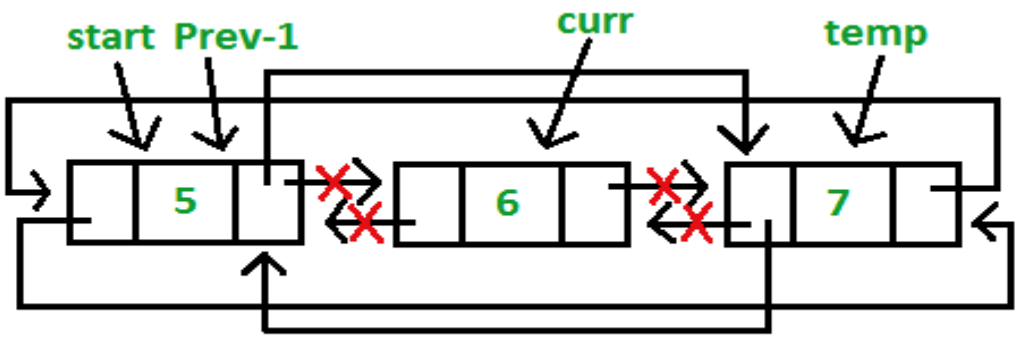


Delete last node of the CDLL





Delete node at a particular position of the CDLL



Complexities CDLL

Particulars	Time Complexity	Space Complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(n)
Traversing (Both)	O(n)	O(1)
Searching	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of Linked List	O(n)	O(1)

Exercise

- Write a program to implement Doubly Linked List operations.
- Write a program to implement Circular Linked List operations.

Prelab Questions

- What are unrolled linked lists.
- Compare linked lists and unrolled linked lists.
- Define skip lists.
- N people have decided to elect a leader by arranging themselves in a circle and eliminating every Mth person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining [with rank 1].
- If we want to concatenate two linked lists which of the following gives O(1) complexity?
  - Singly linked list
  - Doubly linked list
  - Circular doubly linked list