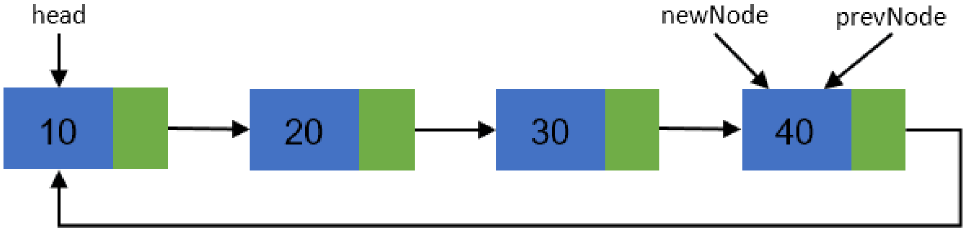
## Algorithm to create circular linked list

```
%%Input : N {Total number of nodes to be created}
Begin:
    alloc (head)
    read (data)
    head.data ← data;
    head.next ← NULL;
    prevNode ← head;
    For count ← 2 to N do
        alloc (newNode)
        read (data)
        newNode.data ← data;
        newNode.next ← NULL;
        prevNode.next ← newNode;
        prevNode ← newNode;
    End for
    prevNode.next ← head;

End
```



## Algorithm to create Doubly Linked list

```
Begin:
    alloc (head)
        read (data)
        head.data ← data;
        head.prev ← NULL;
        head.next ← NULL;
        last ← head;
        write ('List created successfully')

End
```

## Algorithm to traverse Doubly Linked list from beginning

```
%% Input :  head {Pointer to the first node of the list}
Begin:
    If (head == NULL) then
        write ('List is empty')
    End if
    Else then
```

```
        temp ← head;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.next;
        End while
    End else
End
```

### Algorithm to traverse Doubly Linked list from end

```
%% Input :  last {Pointer to the last node of the list}
Begin:
    If (last == NULL) then
        write ('List is empty')
    End if
    Else then
        temp ← last;
        While (temp != NULL) do
            write ('Data = ', temp.data)
            temp ← temp.prev;
        End while
    End else
End
```

## INSERTION:

### Algorithm to insert a node at the beginning of a Doubly linked list

```
%% Input : head {A pointer pointing to the first node of the list}
Begin:
    alloc (newNode)
    If (newNode == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        newNode.data ← data;
        newNode.prev ← NULL;
        newNode.next ← head;

        head.prev ← newNode;
        head ← newNode;
        write('Node added successfully at the beginning of List')
    End else
End
```

## Algorithm to insert a node at the end of Doubly linked list

```
%% Input : last {Pointer to the last node of doubly linked list}
Begin:
    alloc (newNode)
    If (newNode == NULL) then
        write ('Unable to allocate memory')
    End if
    Else then
        read (data)
        newNode.data ← data;
        newNode.next ← NULL;
        newNode.prev ← last;

        last.next ← newNode;
        last ← newNode;
        write ('Node added successfully at the end of List')
    End else
End
```

## Algorithm to insert node at any position of doubly linked list

```
%% Input : head {Pointer to the first node of doubly linked list}
        : last {Pointer to the last node of doubly linked list}
        : N {Position where node is to be inserted}
Begin:
    temp ← head
    For i←1 to N-1 do
        If (temp == NULL) then
            break
        End if
        temp ← temp.next;
    End for
    If (N == 1) then
        insertAtBeginning()
    End if
    Else If (temp == last) then
        insertAtEnd()
    End if
    Else If (temp != NULL) then
        alloc (newNode)
        read (data)

        newNode.data ← data;
        newNode.next ← temp.next
        newNode.prev ← temp
        If (temp.next != NULL) then
            temp.next.prev ← newNode;
        End if
        temp.next ← newNode;
        write('Node added successfully')
    End if
End
```

## Algorithm to delete node from end

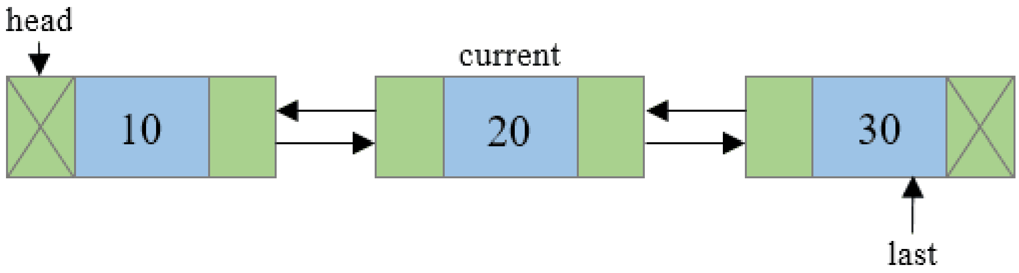
```
%% Input: last {Pointer to last node of the linked list}
Begin:
    If (last == NULL) then
        write ('Can't delete from an empty list')
    End if
    Else then
        toDelete ← last;
        last ← last.prev;
        last.next ← NULL;
        unalloc (toDelete)
        write ('Successfully deleted last node from the list')

    End if
End
```

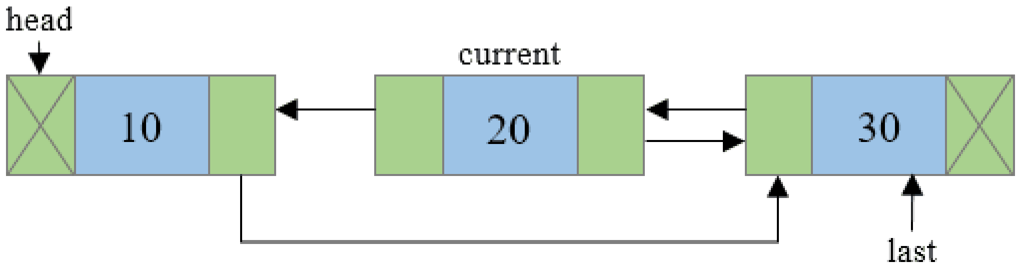
## Algorithm to delete node from any position

```
%% Input : head {Pointer to the first node of the list}
         last {Pointer to the last node of the list}
         N {Position to be deleted from list}
Begin:
    current ← head;
    For i ← 1 to N and current != NULL do
        current ← current.next;
    End for
    If (N == 1) then
        deleteFromBeginning()
    End if
    Else if (current == last) then
        deleteFromEnd()
    End if
    Else if (current != NULL) then
        current.prev.next ← current.next
        If (current.next != NULL) then
            current.next.prev ← current.prev;
        End if
        unalloc (current)
        write ('Node deleted successfully from ', N, ' position')
    End if
    Else then
        write ('Invalid position')
    End if
End
```

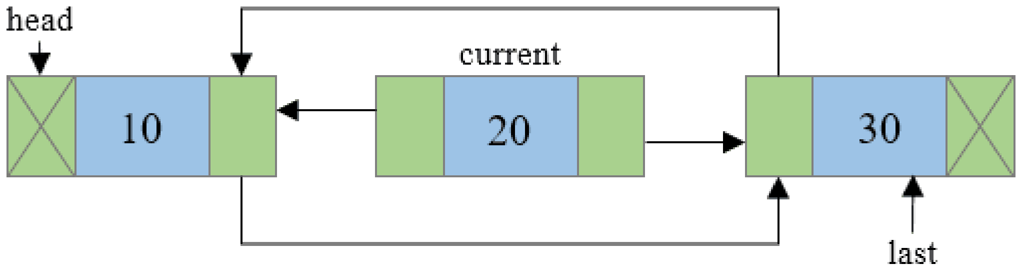
1. Traverse to Nth node of the linked list, lets say a pointer current points to Nth node in our case 2 node.



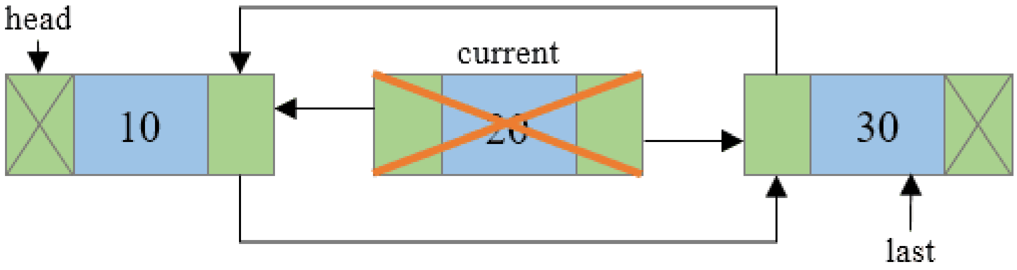
2. Link the node behind current node with the node ahead of current node, which means now the N-1th node will point to N+1th node of the list. Which can be implemented as current->prev->next = current->next.

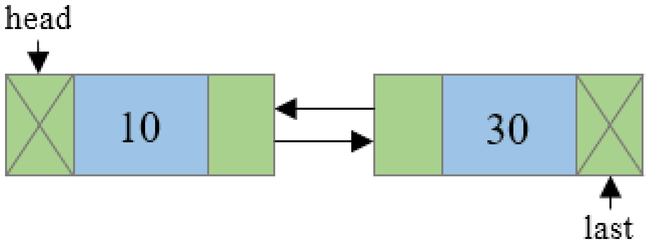


3. If N+1th node is not NULL then link the N+1th node with N-1th node i.e. now the previous address field of N+1th node will point to N-1th node. Which can be implemented as current->next->prev = current->prev.



4. Finally delete the current node from memory and you are done.





CODE:

```
struct node
{
int value;
struct node* next;
struct node* prev;
};
struct node* head;
struct node* tail;
void init()
{
   head=NULL;
   tail=NULL;
}
```

**Insertion at First:**

```
void insertFirst(int element)
{
   struct node* newItem;
   newItem=new node;
   if(head==NULL)
   {
      head=newItem;
```

```
        newItem->prev=NULL;
        newItem->value=element;
        newItem->next=NULL;
        tail=newItem;
    }
    else
    {
        newItem->next=head;
        newItem->value=element;
        newItem->prev=NULL;
        head->prev=newItem;
        head=newItem;
    }
}
```

**Deletion at First:**

```
void deleteFirst()
{
    if(head==NULL)
    {
        return;
    }
    if(head==tail)///one element in the list
    {
        struct node* cur;
        cur=head;
        head=NULL;
        tail=NULL;
        delete cur;
        return;
    }
    else
    {
        struct node* cur;
        cur=head;
        head=head->next;
        head->prev=NULL;
        delete cur;
    }
}
```