

# Data Structures & Algorithms

Chapter - 03.02

Doubly Linked List

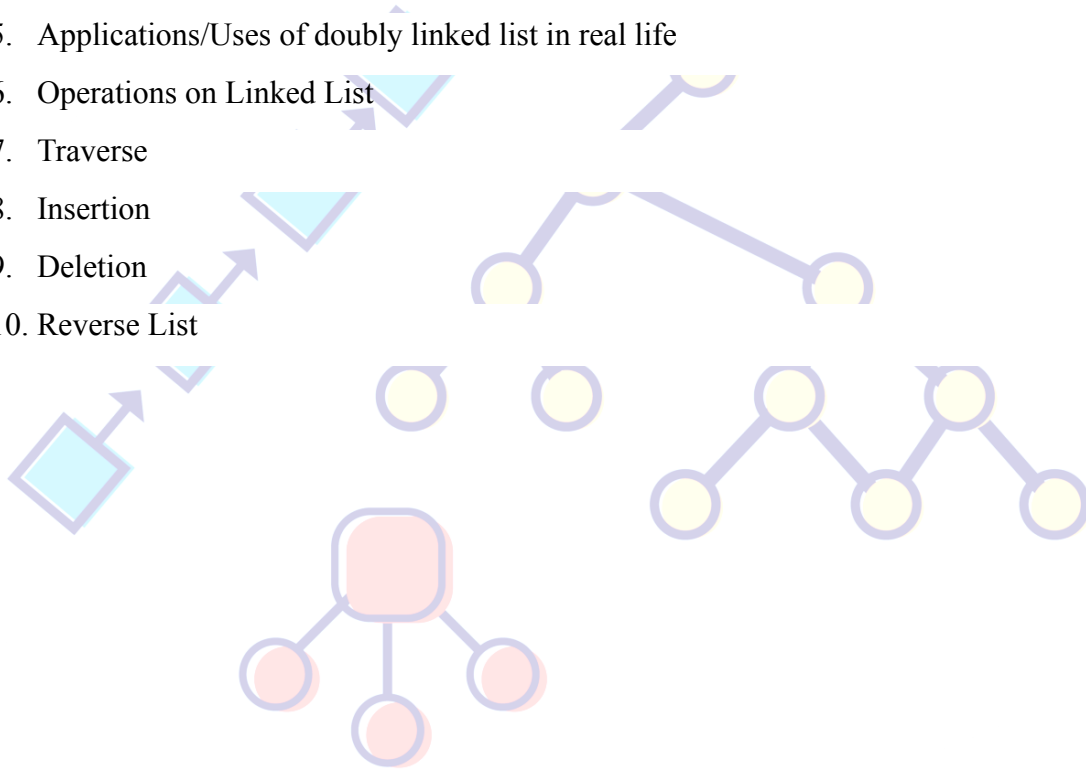
Rakin Mohammad Sifullah  
Computer Science Engineer

## Overview

In this chapter, we will cover Linked Lists.

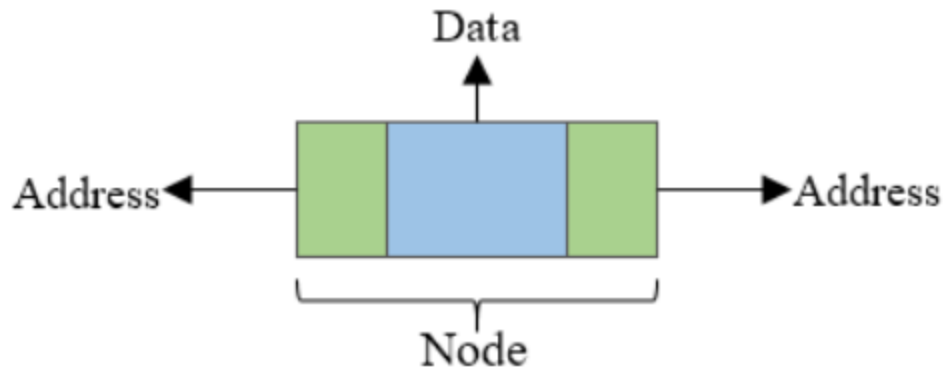
We will cover the following -

1. Introduction
2. Doubly Linked List Representation
3. The basic structure of a doubly linked list
4. Advantages and Disadvantages
5. Applications/Uses of doubly linked list in real life
6. Operations on Linked List
7. Traverse
8. Insertion
9. Deletion
10. Reverse List

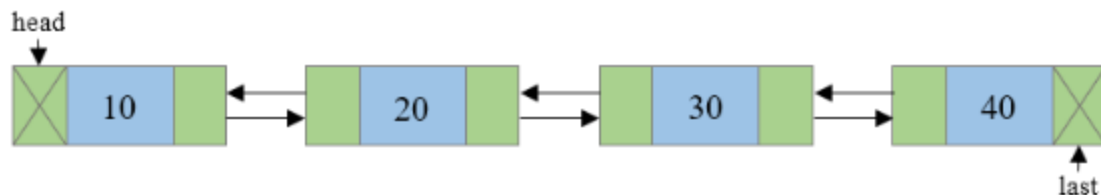



## Introduction

A doubly linked list is a collection of nodes connected together in a sequential way. Each node of the list contains two parts data part and the reference or address part.



A doubly linked list is almost similar to a singly linked list except it contains two address or reference fields, where one of the address fields contains a reference to the next node and the other contains a reference to the previous node. The first and last node of a linked list contains a terminator generally a NULL value, that determines the start and end of the list. Doubly linked list is sometimes referred to as a bi-directional linked list since it allows the traversal of nodes in both directions.



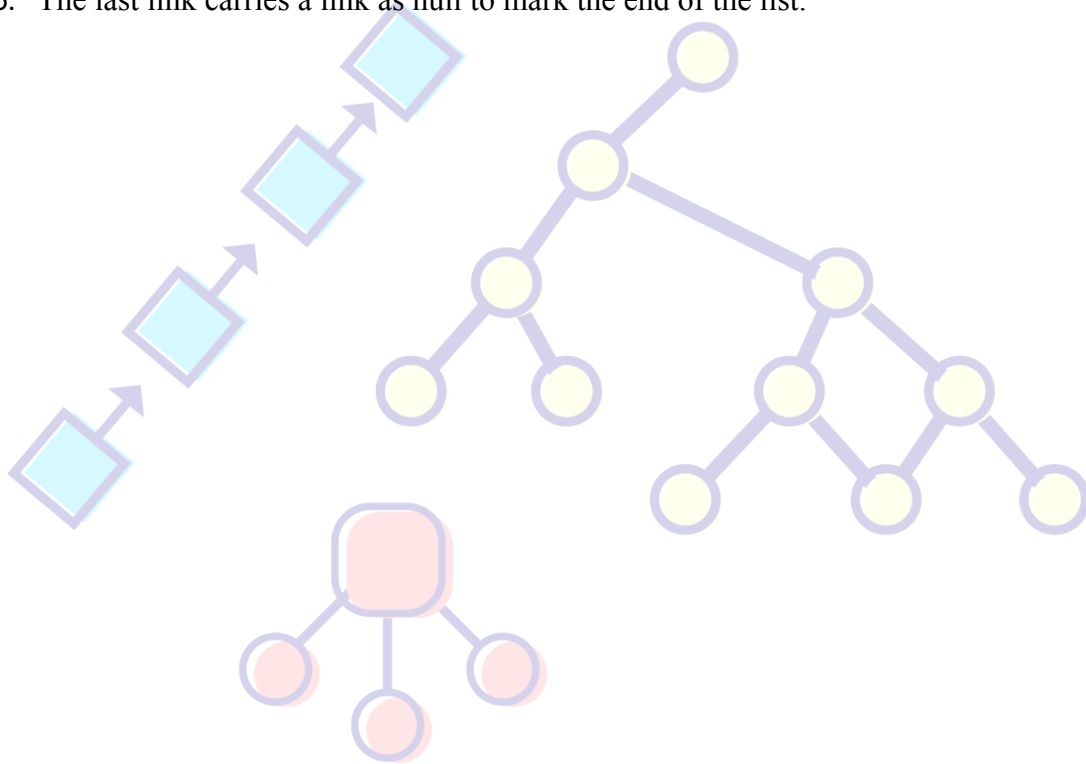
 Doubly Linked List

**Since a doubly linked list allows the traversal of nodes in both directions, we can keep track of both the first and last nodes.**

## Representation

The following are the important points to be considered:

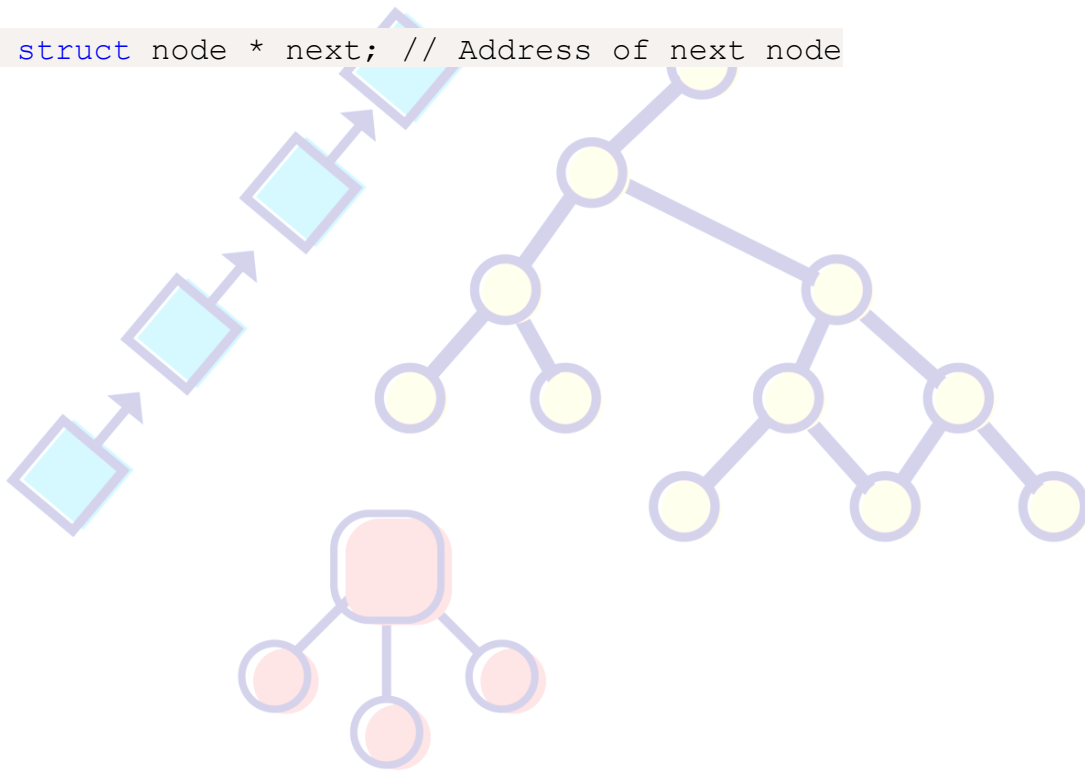
1. Doubly Linked List contains a link element called first and last.
2. Each link carries a data field(s) and two link fields called next and prev.
3. Each link is linked with its next link using its next link.
4. Each link is linked with its previous link using its previous link.
5. The last link carries a link as null to mark the end of the list.



## Basic Structure

The basic structure of a doubly linked list contains a data field and two address fields. Here is how it can be represented in C programming language.

```
struct node {  
    int data;           // Data field  
    struct node * prev; // Address of previous node  
    struct node * next; // Address of next node  
};
```



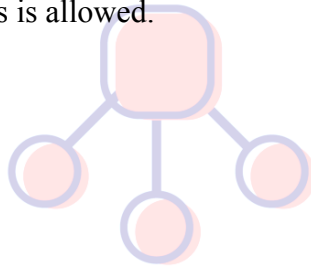
## Advantages and Disadvantages

**Here are various advantages of a doubly linked list:**

1. As with a singly linked list, it is the easiest data structure to implement.
2. Allows traversal of nodes in both directions which is not possible in the singly linked list.
3. Deletion of nodes is easy when compared to a singly linked list, as in singly linked list deletion requires a pointer to the node and previous node to be deleted. This is not in the case for the doubly linked list we only need the pointer which is to be deleted.
4. Reversing the list is simple and straightforward.
5. Can allocate or de-allocate memory easily when required during its execution.
6. It is one of the most efficient data structures to implement when traversing in both directions is required.

**Not many but doubly linked lists have a few disadvantages also which can be listed below:**

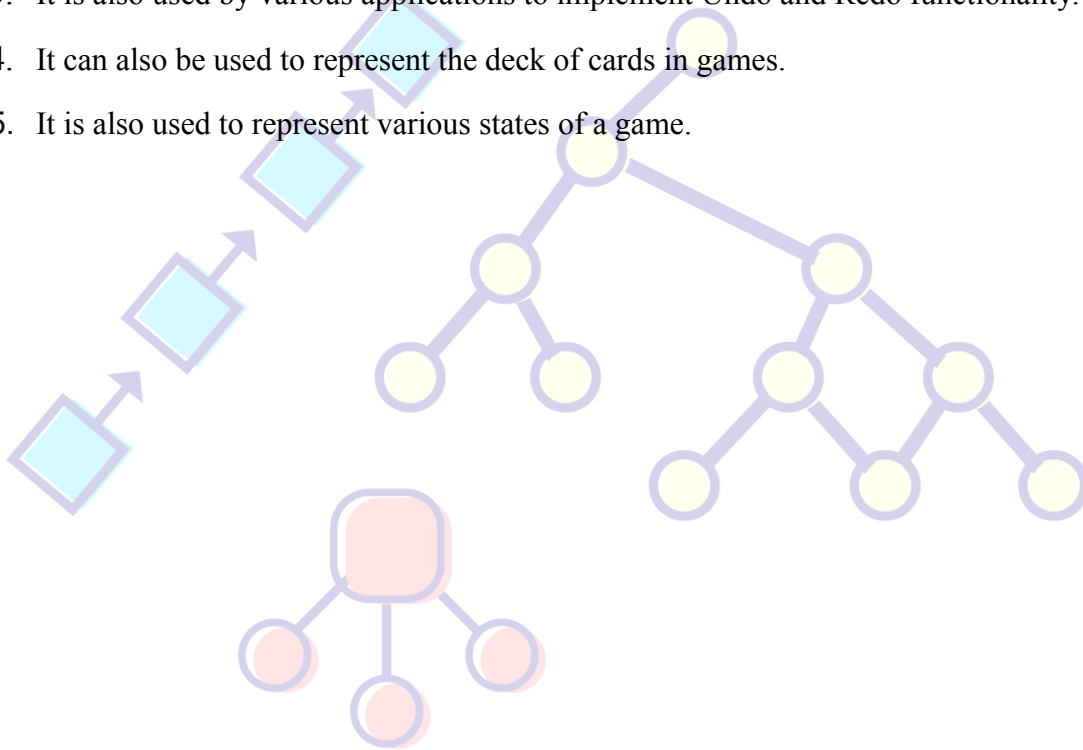
1. It uses extra memory when compared to an array and singly linked list.
2. Since elements in memory are stored randomly, hence elements are accessed sequentially no direct access is allowed.



## Applications/Uses

**There is various application of doubly linked list in the real world. Some of them can be listed as:**

1. A doubly linked list can be used in navigation systems where both front and back navigation is required.
2. It is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward button.
3. It is also used by various applications to implement Undo and Redo functionality.
4. It can also be used to represent the deck of cards in games.
5. It is also used to represent various states of a game.



## Operations

### Basic operations performed on Doubly linked list

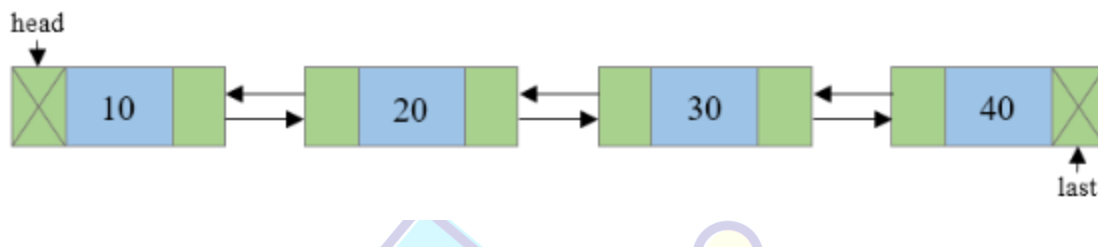
1. Creation of list.
2. Traversal of the list in the forward direction.
3. Traversal of the list in the backward direction.
4. Insertion of node at the beginning of the list.
5. Insertion of node at the end of the list.
6. Insertion of node at any position in the list.
7. Deletion of the first node from the list.
8. Deletion of the last node from the list.
9. Deletion of a node from the middle of the list.
10. Deletion of the entire list.
11. Counting the total number of nodes.
12. Searching an item in the list.
13. Reversing the list.





## Traverse

Write a C program to implement the Doubly linked list data structure. Write a C program to create a doubly linked list and display all nodes of the created list. How to create and display a doubly linked list in C. Algorithm to create and traverse a doubly linked list.



## Algorithm (Doubly Link List)

Algorithm to create Doubly Linked list

Begin:

```
alloc (head)
```

```
If (head == NULL) then
```

```
    write ('Unable to allocate memory')
```

```
End if
```

```
Else then
```

```
    read (data)
```

```
    head.data ← data;
```

```
    head.prev ← NULL;
```

```
    head.next ← NULL;
```

```
    last ← head;
```

```
    write ('List created successfully')
```

```
End else
```

End

### Algorithm of Traverse Part (display from the 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**

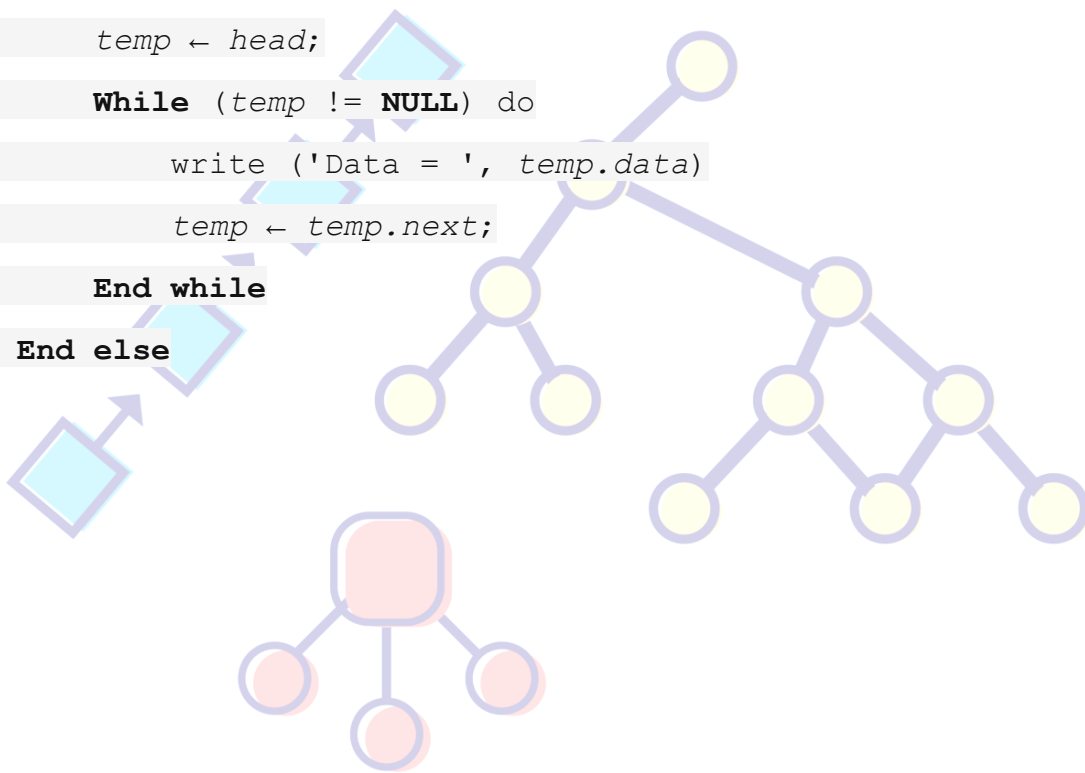
write ('Data = ', *temp.data*)

*temp* ← *temp.next*;

**End while**

**End else**

**End**



### Algorithm of Traverse Part (display from the 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

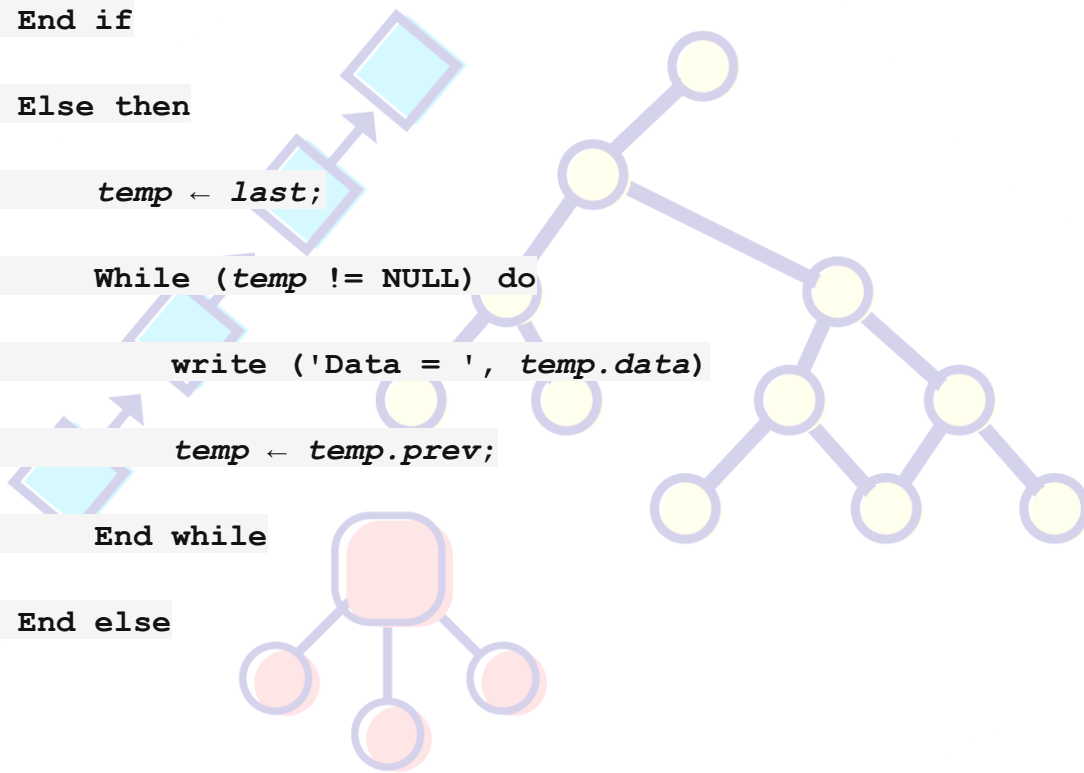
write ('Data = ', *temp.data*)

*temp* ← *temp.prev*;

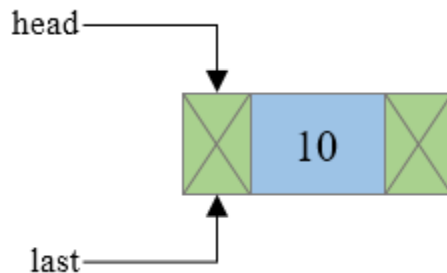
End while

End else

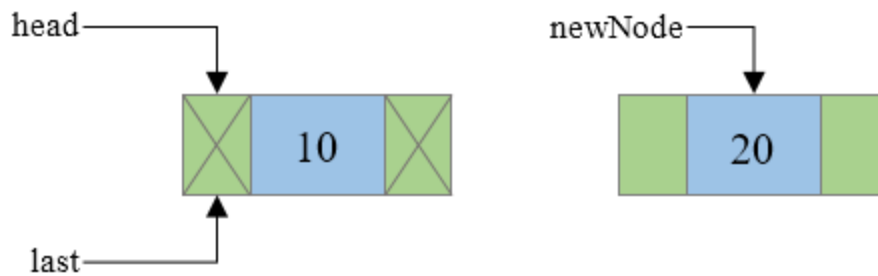
End



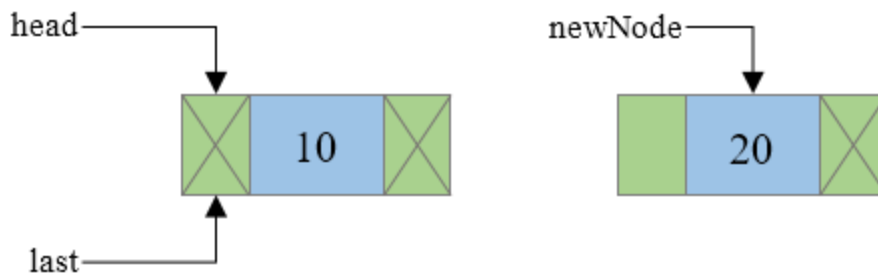
1. Create a head node and assign some data to its data field.
2. Make sure that the previous and next address field of the head node must point to NULL.
3. Make the head node as last node.



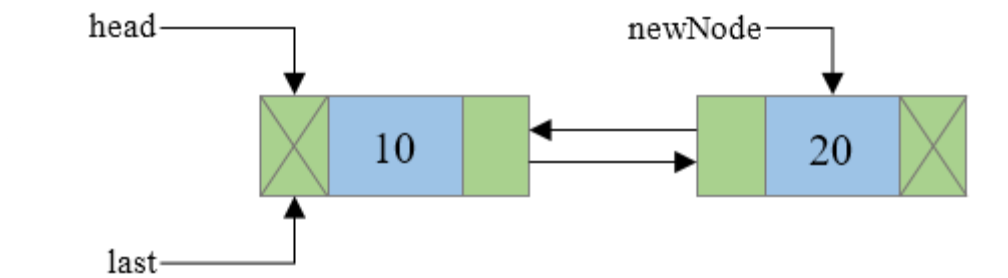
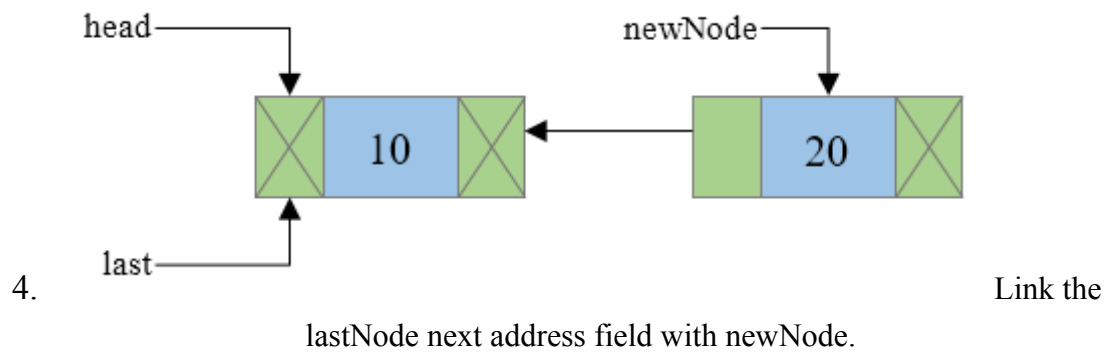
1. If you want to create more nodes then follow these steps: Create a new node and assign some data to its data field.



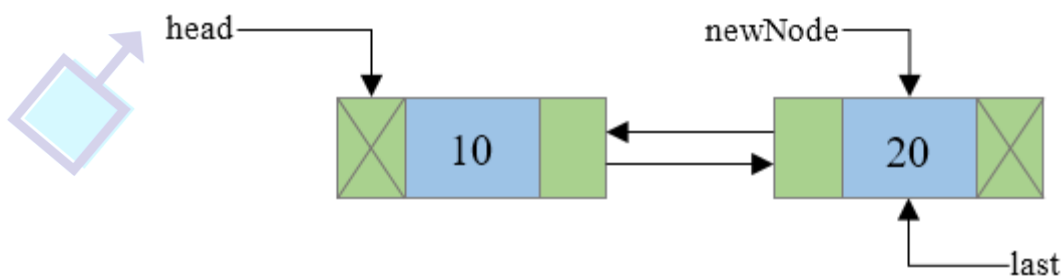
2. Make sure that the next address field of new node must point to NULL.



3. Link the new node previous address field with lastNode.



5. Move the lastNode to newNode i.e. last node will now point to new node.



6. Repeat Steps 4-8 if you want to add more nodes to the list.

## Insertion

Write a C program to create a doubly linked list and insert a new node in beginning, end or at any position in the list. How to insert a new node at beginning of a Doubly linked list. How to insert a new node at the end of a doubly linked list. How to insert a new node at any position of a doubly linked list in C. Algorithm to insert node in a doubly linked list.

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

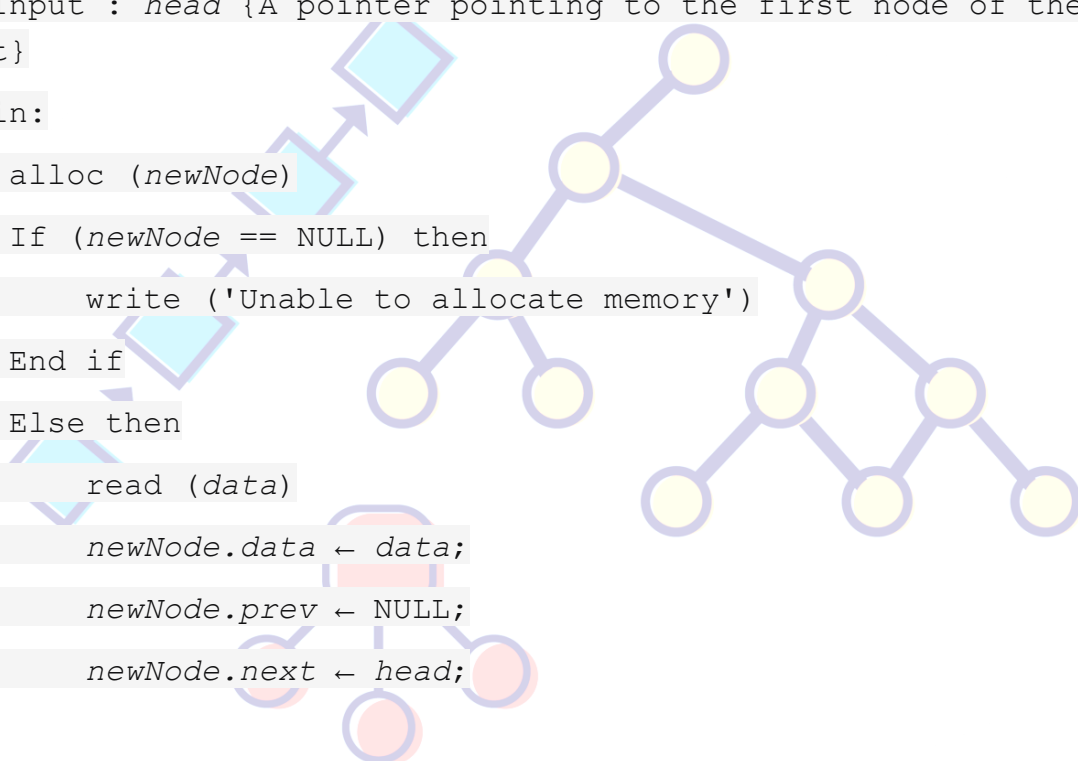
```

%% 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;
```

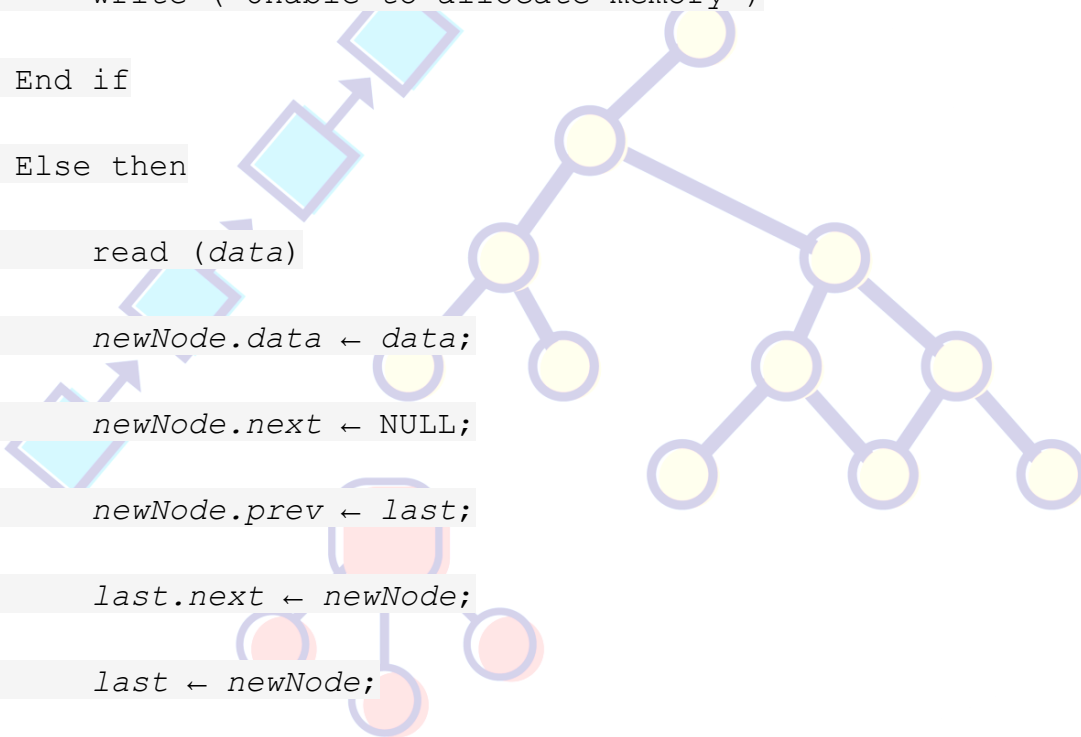
```
        last.next ← newNode;
```

```
        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()
```

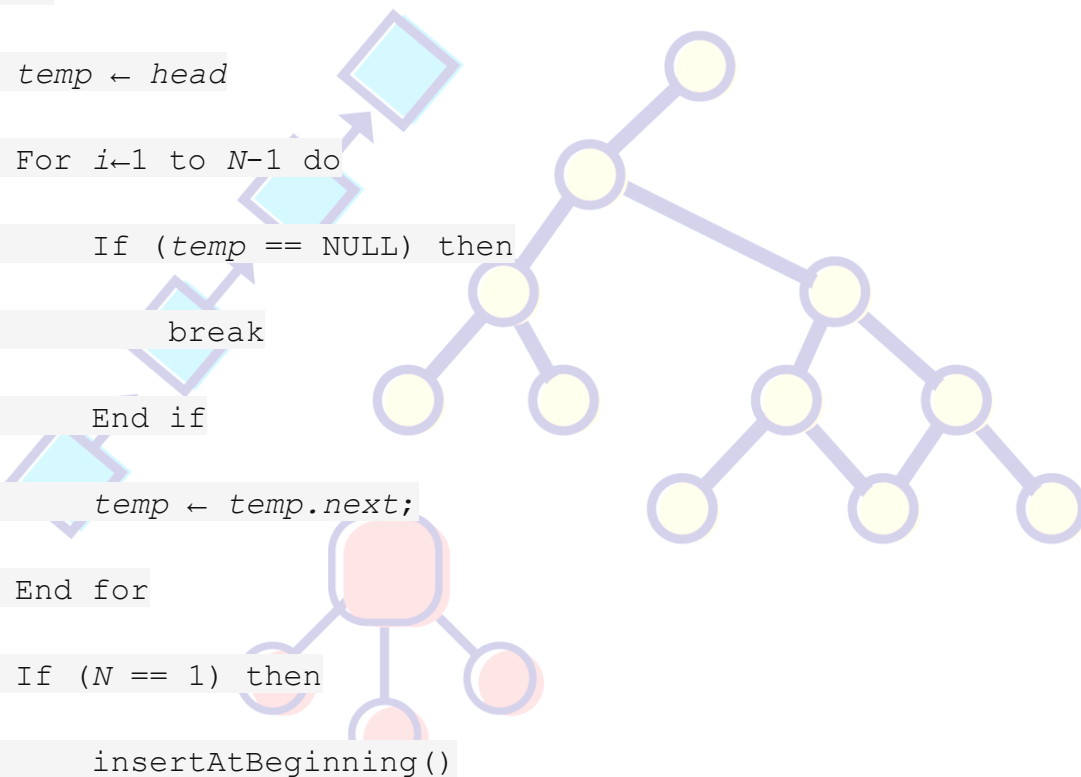
```
End if
```

```
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;
```

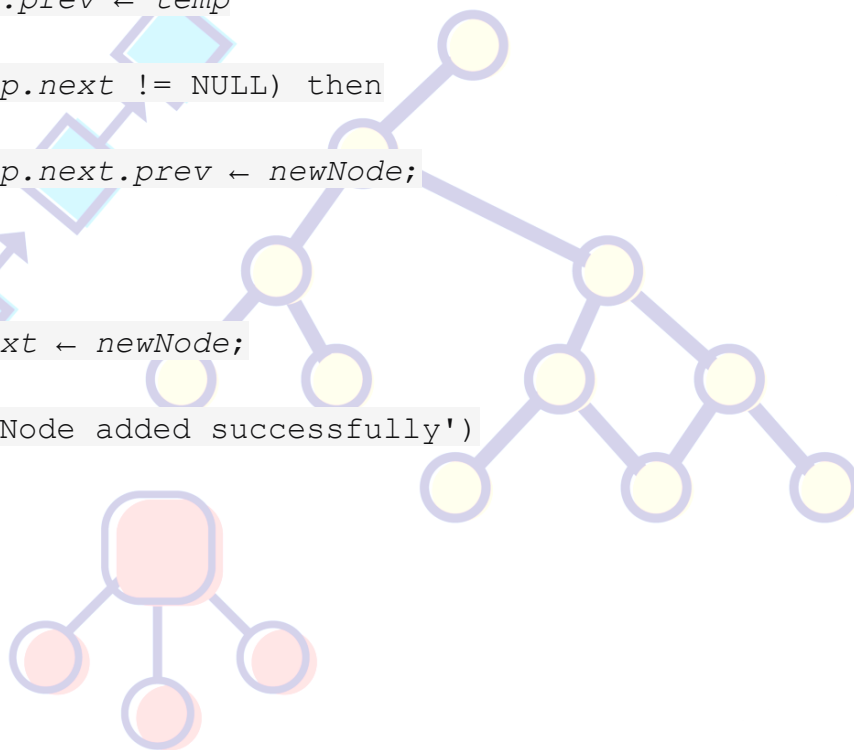
```
End if
```

```
temp.next ← newNode;
```

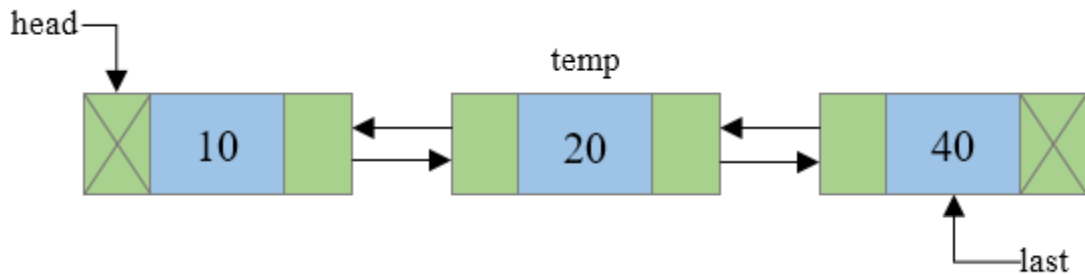
```
write('Node added successfully')
```

```
End if
```

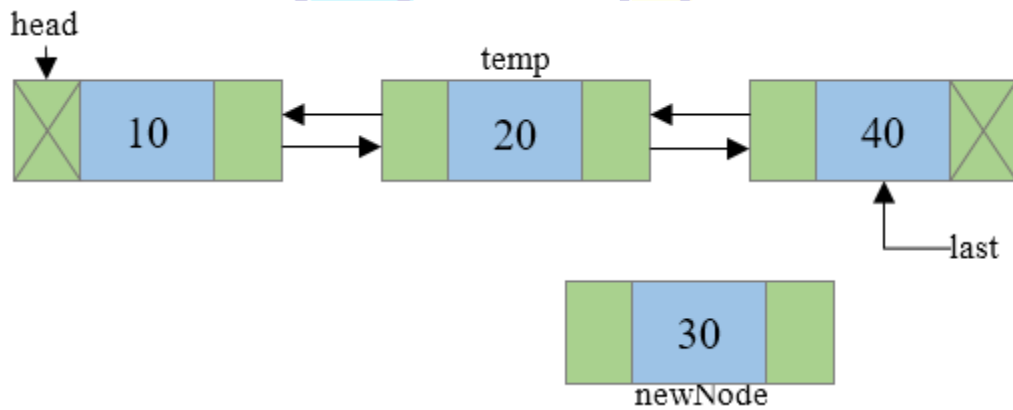
```
End
```



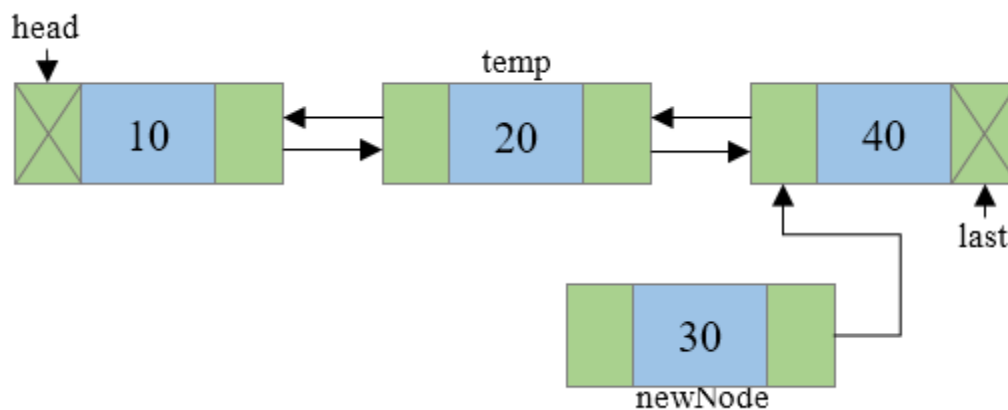
1. Traverse to N-1 node in the list. Where N is the position to insert. Say temp now points to N-1th node.



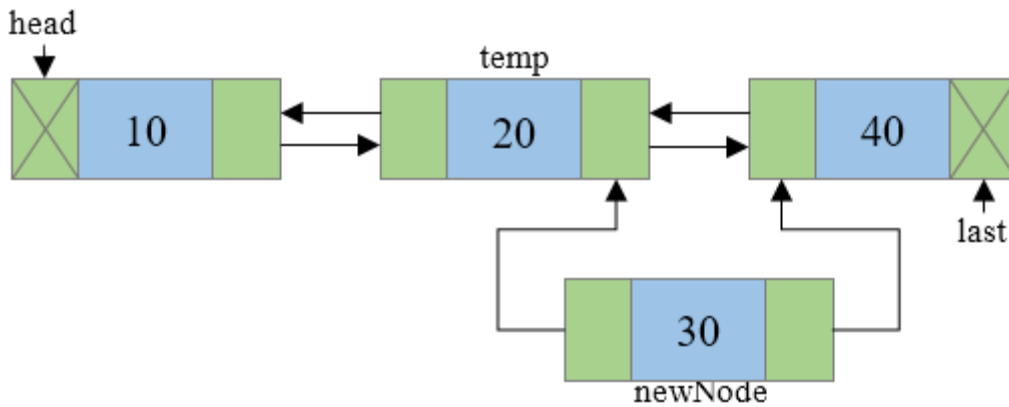
2. Create a newNode that is to be inserted and assign some data to its data field.



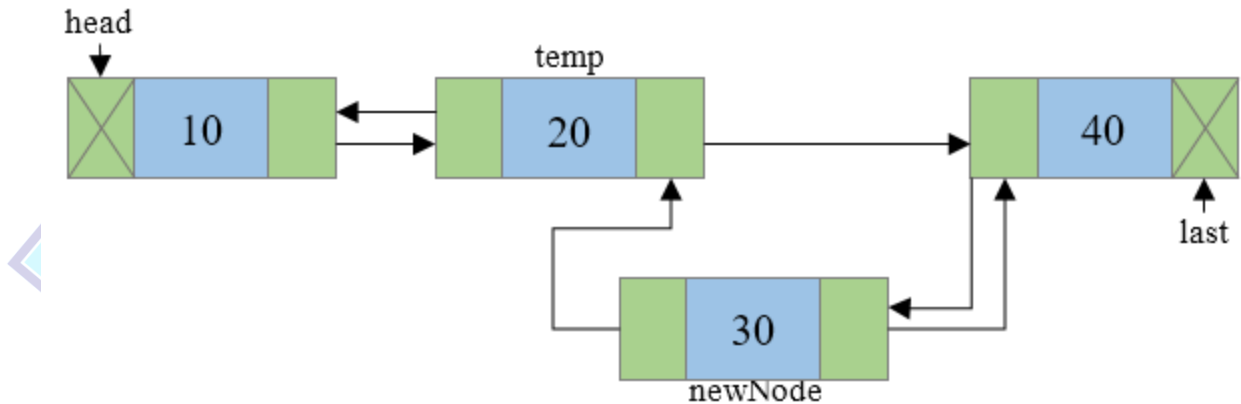
3. Connect the next address field of newNode with the node pointed by next address field of temp node.



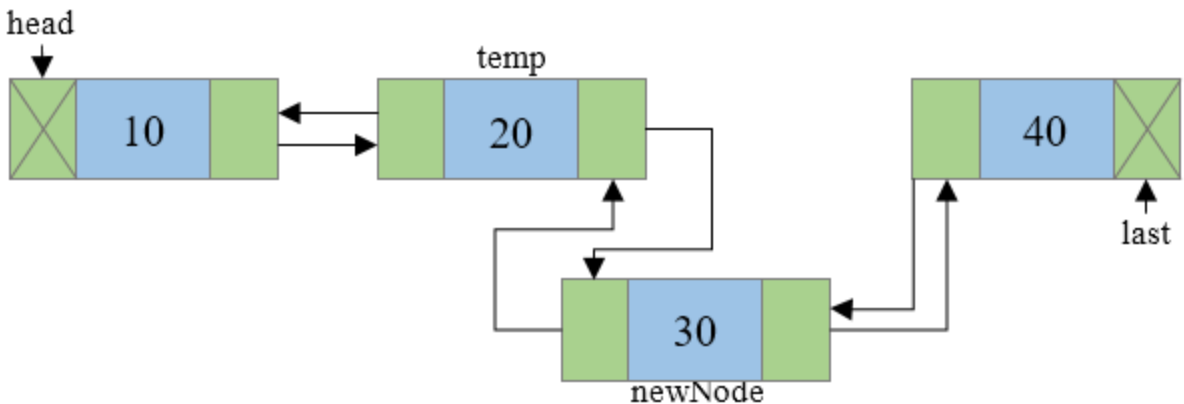
4. Connect the previous address field of newNode with the temp node.



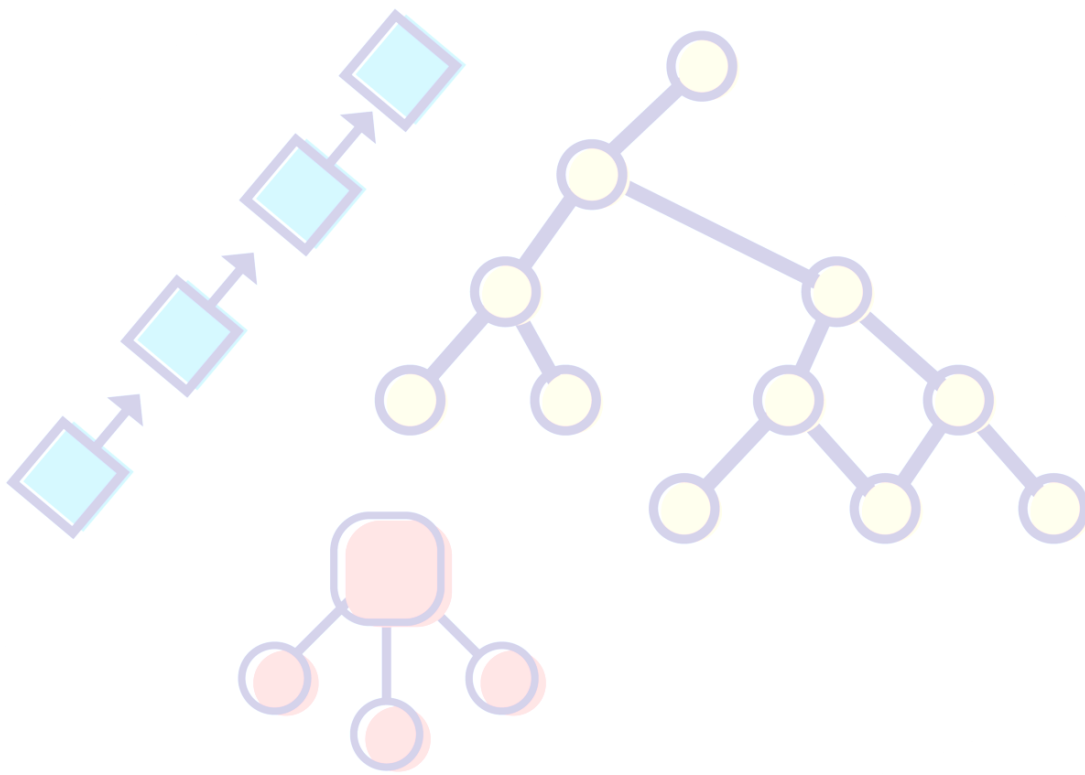
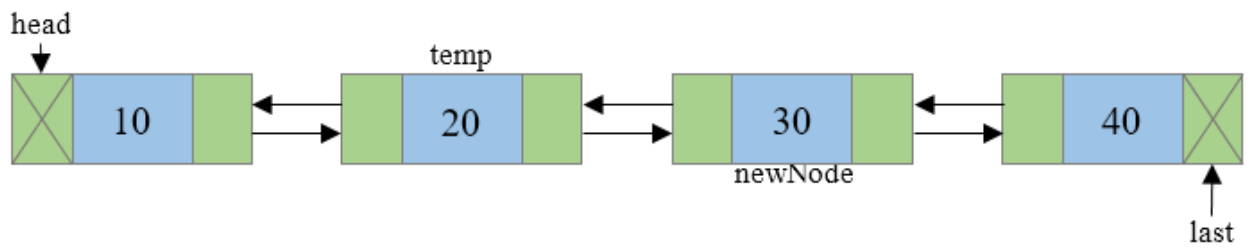
5. Check if temp->next is not NULL then, connect the previous address field of node pointed by temp.next to newNode.



6. Connect the next address field of temp node to newNode i.e. temp->next will now point to newNode.



## 7. Final list



## Deletion

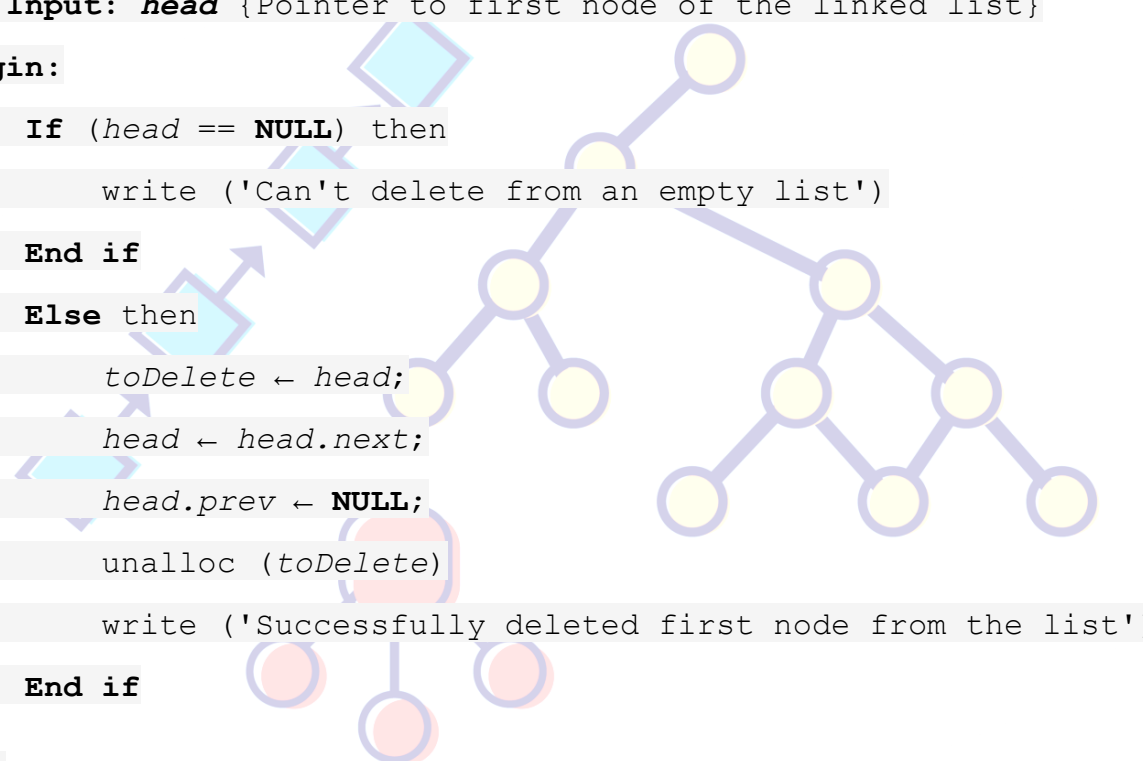
Write a C program to create a doubly linked list and delete a node from beginning, end or at any position of the linked list. How to delete a node from beginning of a doubly linked list. How to delete a node from end of a doubly linked list. How to delete a node from any position of a doubly linked list in C. Algorithm to delete a node from doubly linked list.

### Algorithm to delete node from beginning

```

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

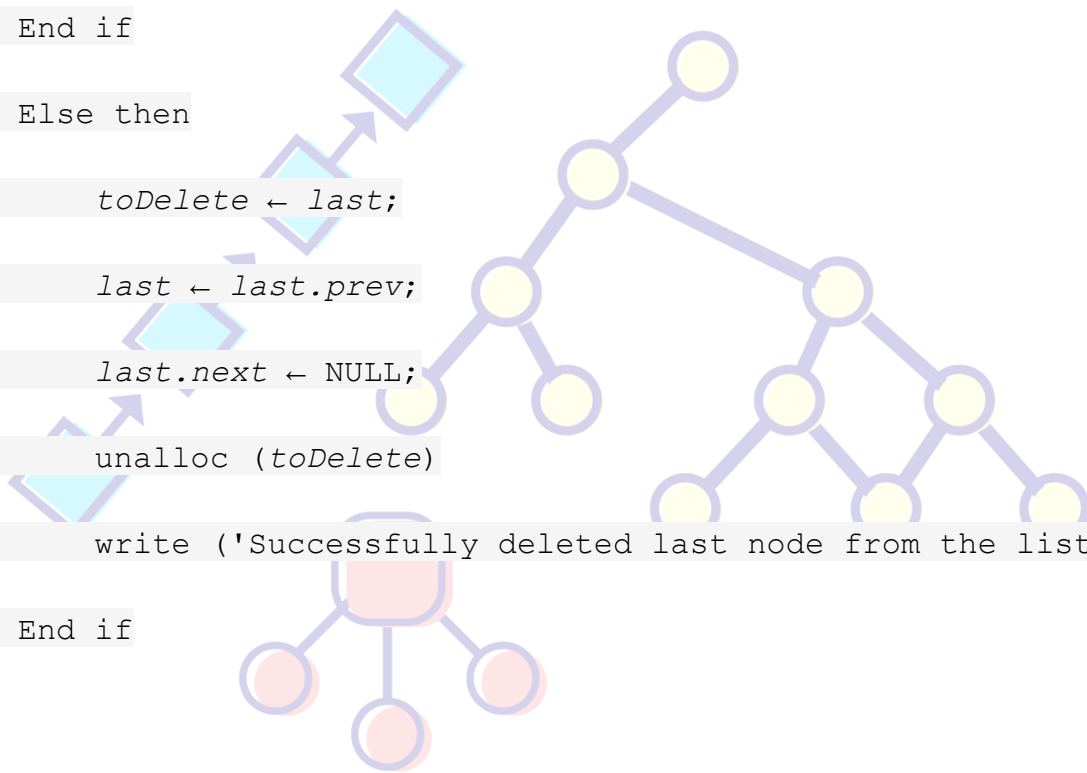
*last*.next ← NULL;

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
```

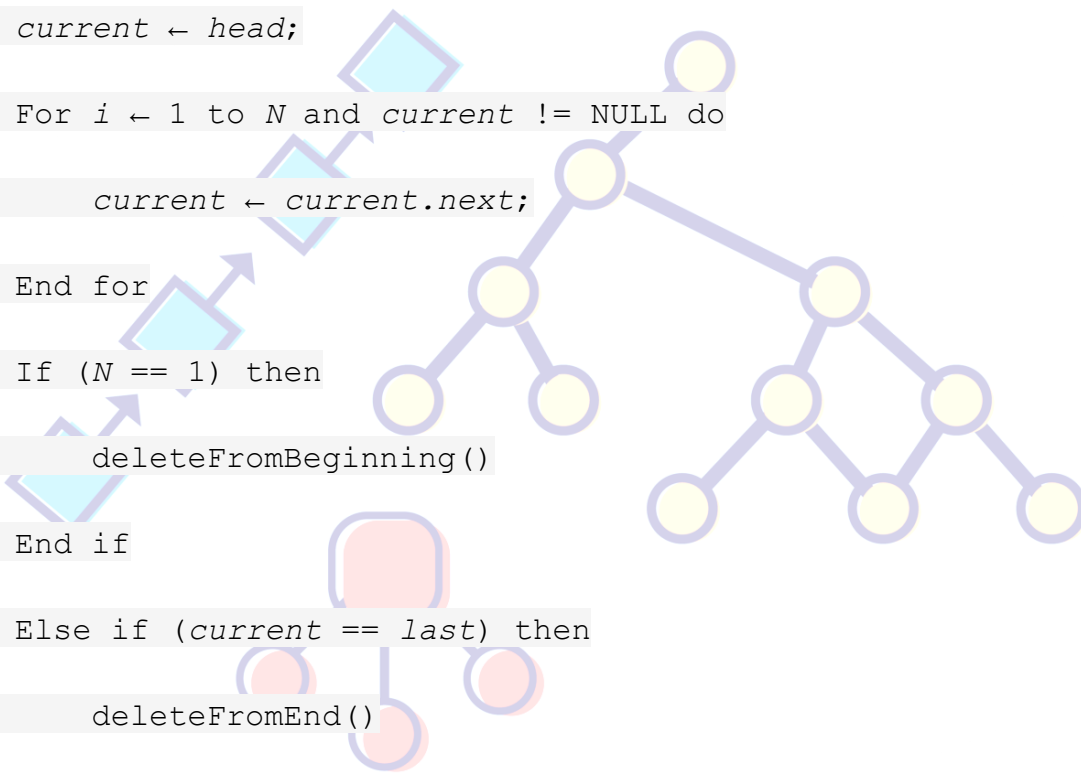
```
Else if (current != NULL) then
```

```
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')

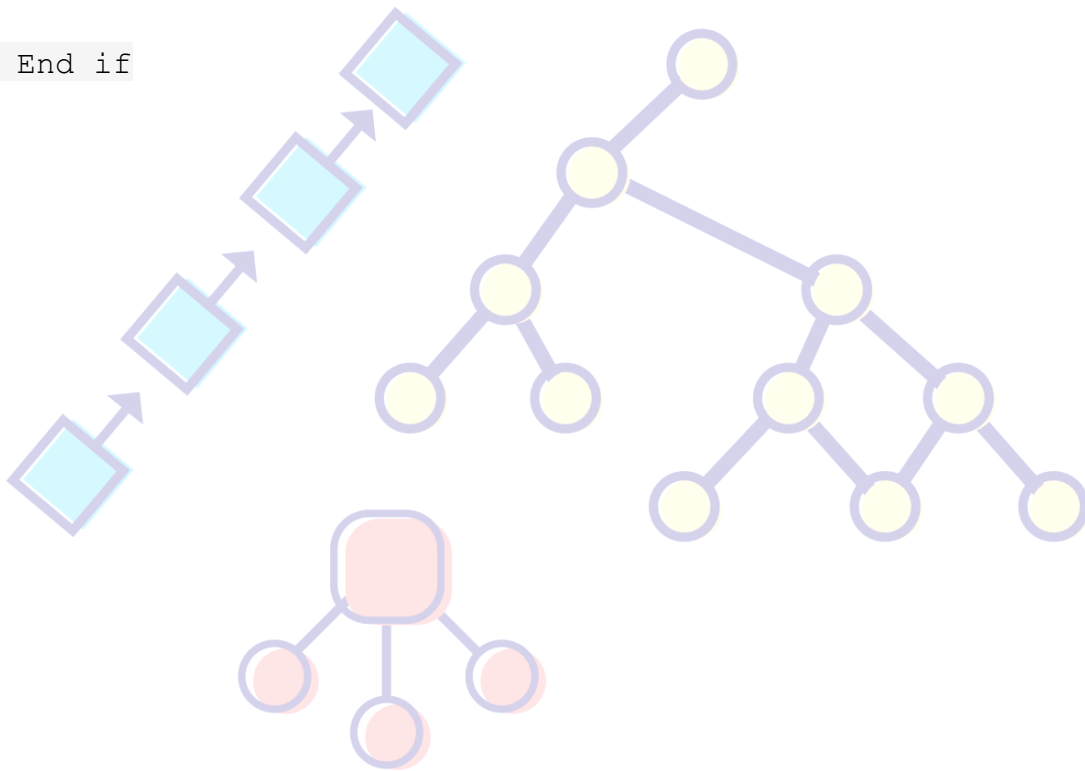
    End if

    Else then

        write ('Invalid position')

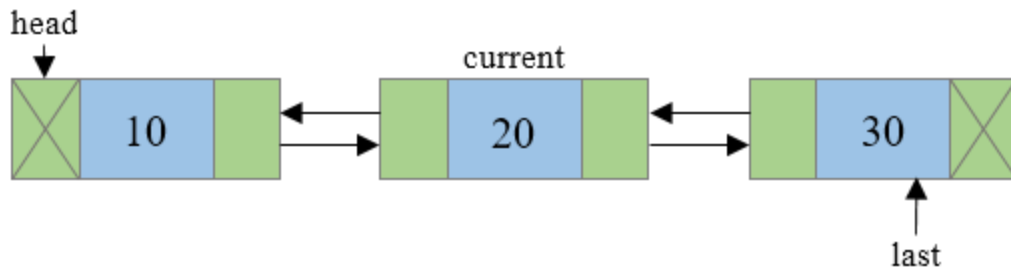
    End if

End
```

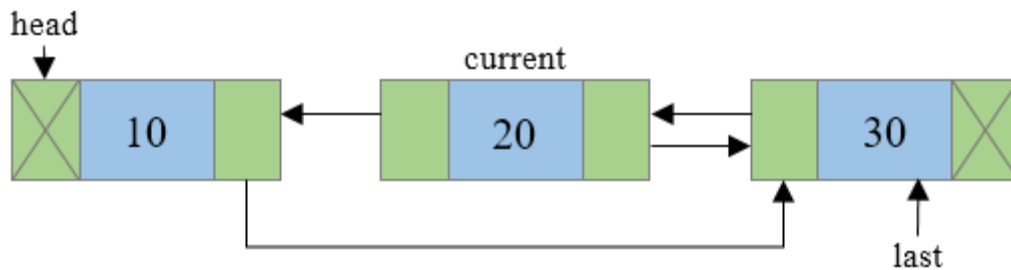




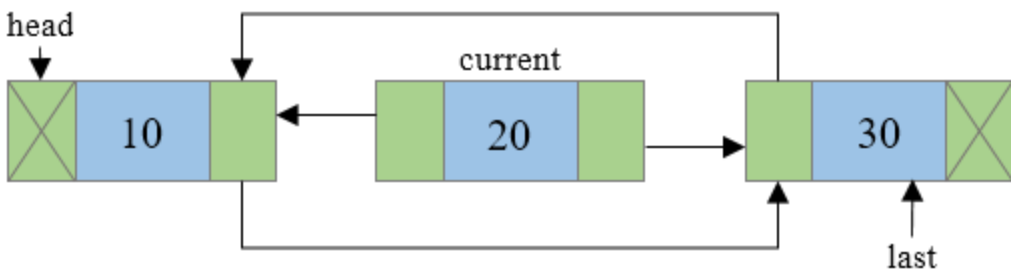
1. Traverse to Nth node of the linked list, let's say a pointer current points to Nth node in our case 2nd node.



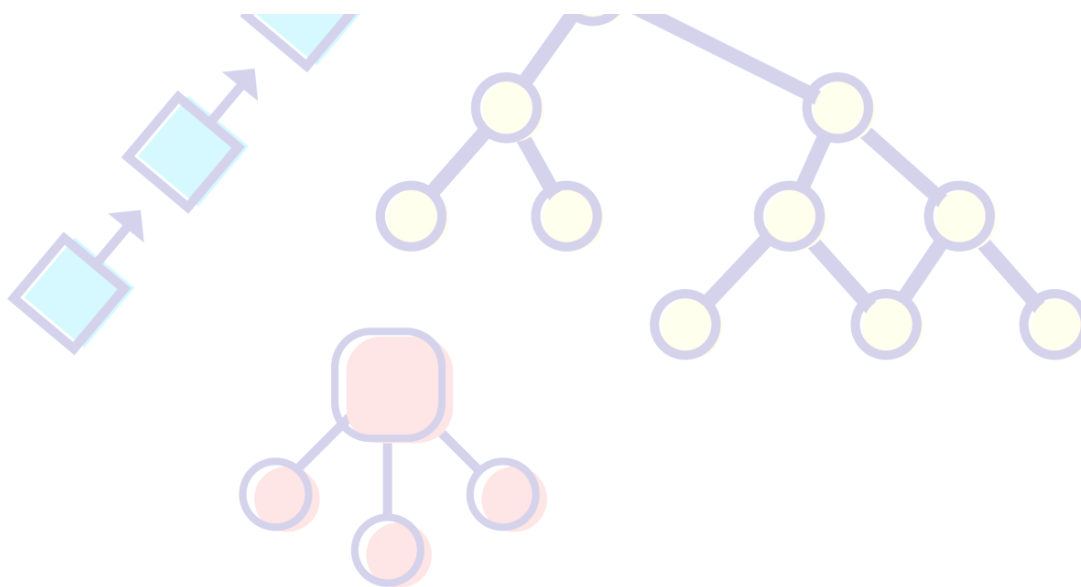
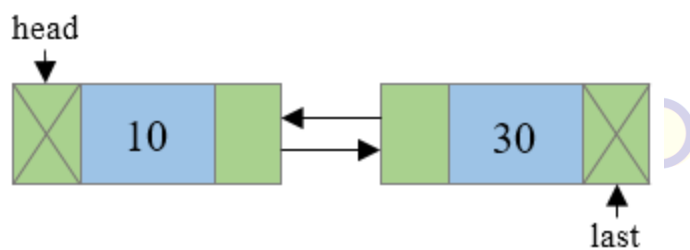
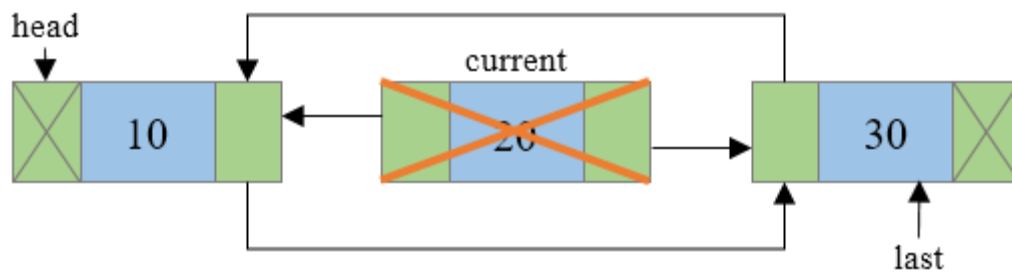
1. Link the node behind current node with the node ahead of current node, which means now the N-1th node will point to N+1th node of the list. Which can be implemented as  $\text{current} \rightarrow \text{prev} \rightarrow \text{next} = \text{current} \rightarrow \text{next}$ .



2. 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  $\text{current} \rightarrow \text{next} \rightarrow \text{prev} = \text{current} \rightarrow \text{prev}$ .



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



## Reverse List

Write a C program to create a doubly linked list and reverse the linked list. How to reverse the doubly linked list in C programming. Algorithm to reverse a doubly linked list.

### Algorithm to reverse a doubly linked list

```
%% Input : head {Pointer to first node of the list}
```

```
last {Pointer to last node of the list}
```

Begin:

```
current ← head;
```

```
While (current != NULL) do
```

```
temp ← current.next;
```

```
current.next ← current.prev;
```

```
current.prev ← temp;
```

```
current ← temp;
```

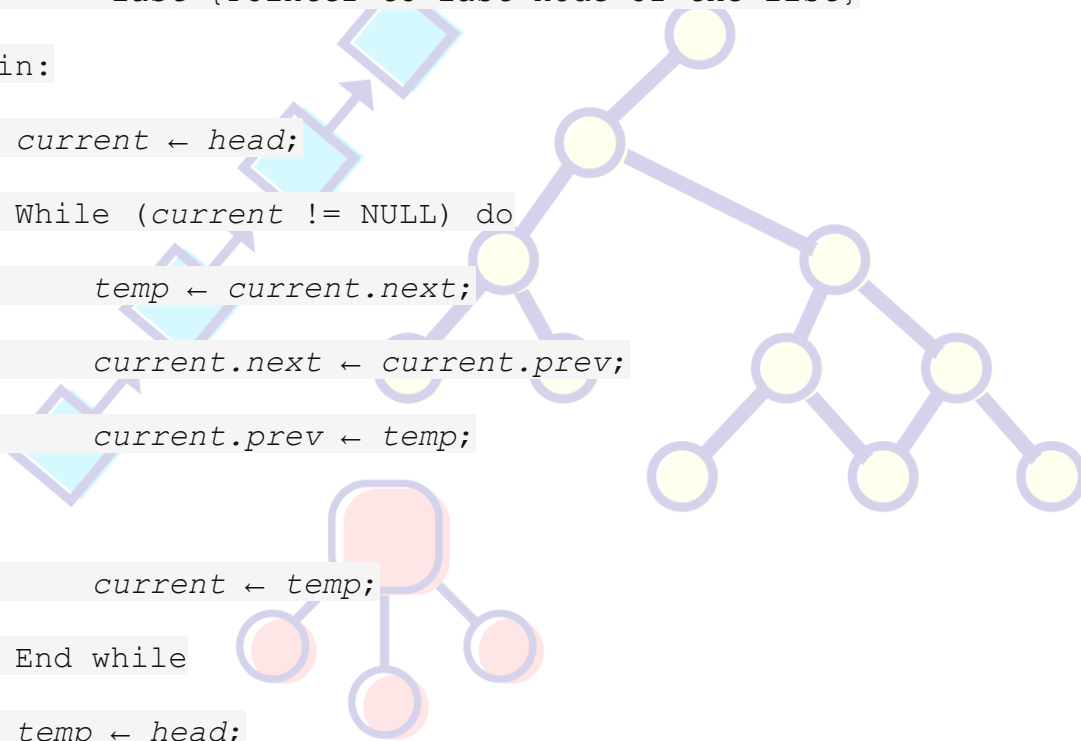
```
End while
```

```
temp ← head;
```

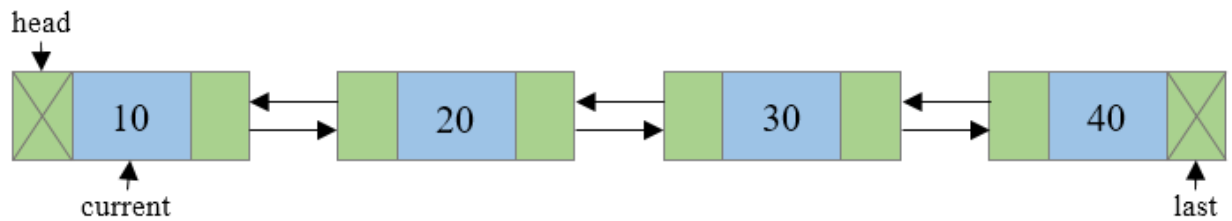
```
head ← last;
```

```
last ← temp;
```

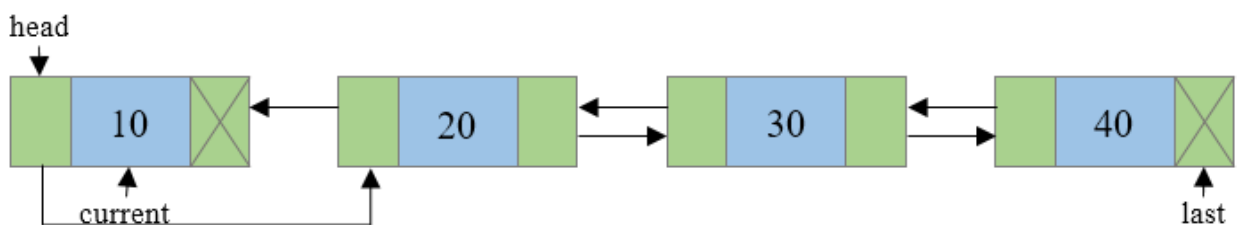
End



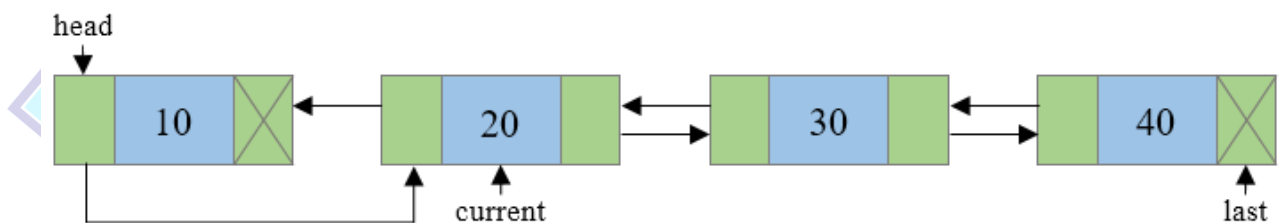
1. To reverse the list we start with the first node. Say a pointer current keeps track of the current node. Now initially current points to head node.



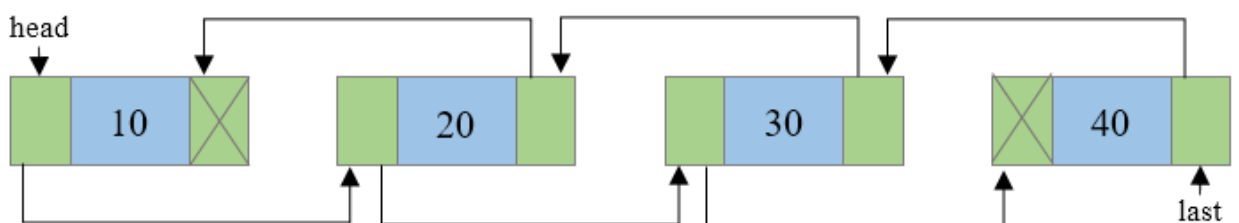
2. Swap the previous and next pointer fields of current node.



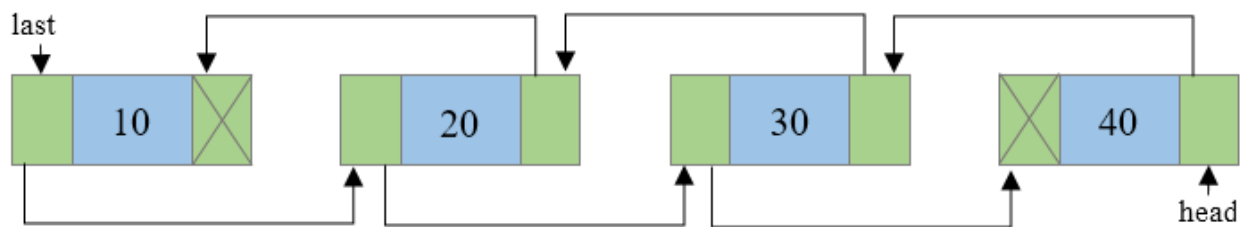
3. Move the position of current pointer to its next node. In general, now current.prev holds the address of next node.



4. Repeat Step 2-3 until current pointer becomes NULL.
5. When, the current pointer becomes NULL then the list will look something like.



6. Finally, swap the head and last pointers and you are done.



7. Now, if you compare the above image to the below given image you will find both are similar linked list.

