Types of Linked List and Operation on Linked List

afteracademy.com/blog/types-of-linked-list-and-operation-on-linked-list

In the previous blog, we have seen the structure and properties of a Linked List. In this blog, we will discuss the types of a linked list and basic operations that can be performed on a linked list.

Types of Linked List

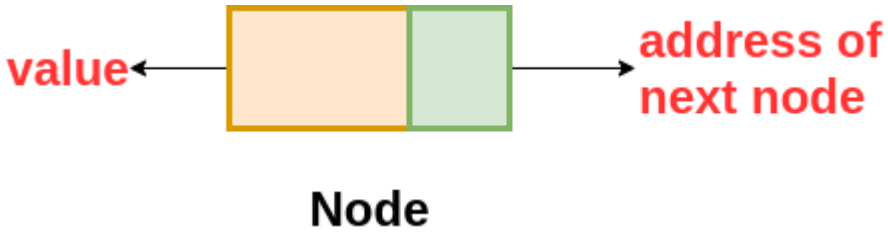
Following are the types of linked list

1. Singly Linked List.
2. Doubly Linked List.
3. Circular Linked List.

Singly Linked List

A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node.

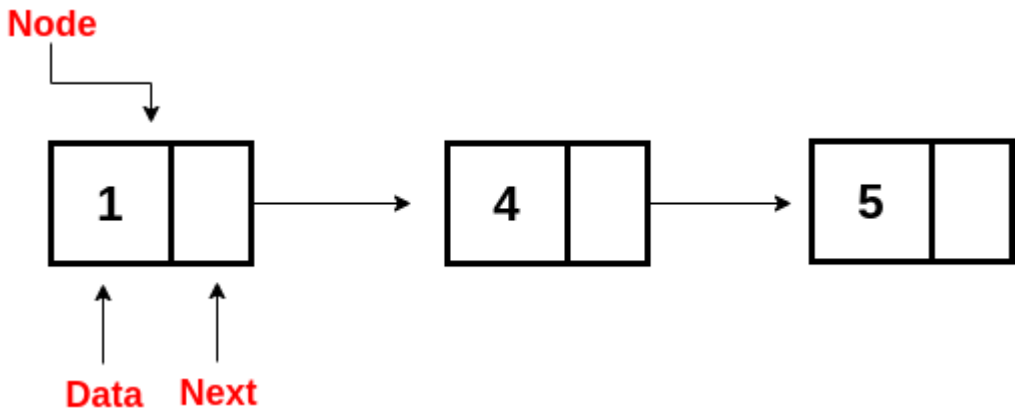
The structure of the node in the Singly Linked List is



Node

```
class Node {
    int data // variable to store the data of the node
    Node next // variable to store the address of the next node
}
```

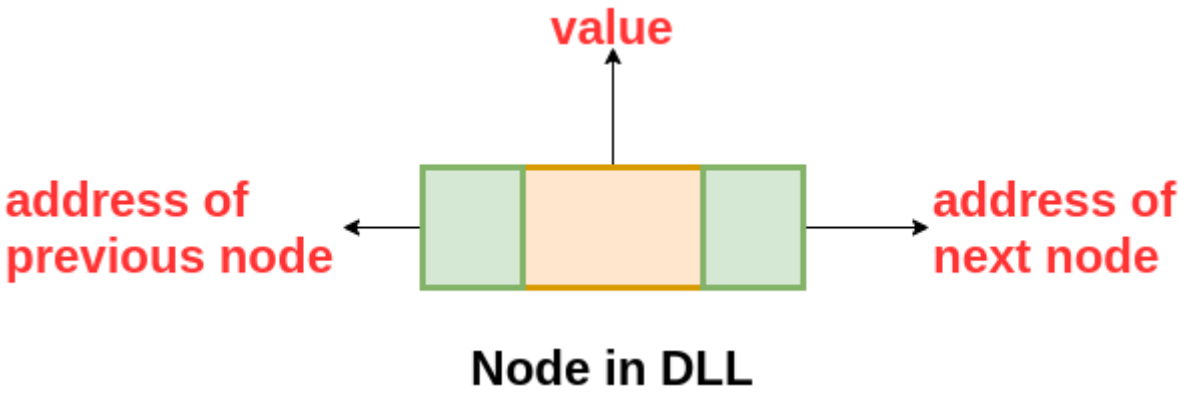
The nodes are connected to each other in this form where the value of the next variable of the last node is NULL i.e. ***next*** = ***NULL***, which indicates the end of the linked list.



Doubly Linked List

A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes.

The following is the structure of the node in the Doubly Linked List(DLL):



Node in DLL

```
class DLLNode {
    int val // variable to store the data of the node
    DLLNode prev // variable to store the address of the previous node
    DLLNode next // variable to store the address of the next node
}
```

The nodes are connected to each other in this form where the first node has **prev** = **NULL** and the last node has **next** = **NULL**.

Doubly Linked List

Advantages over Singly Linked List-

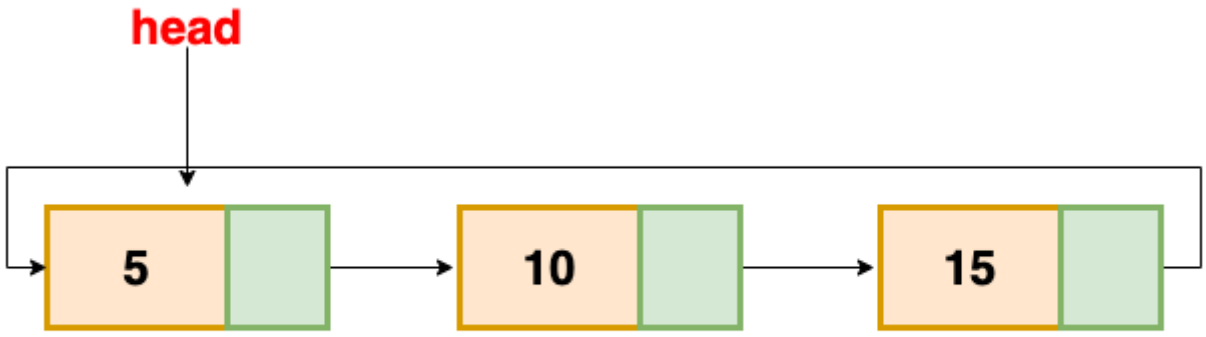
- It can be traversed both forward and backward direction.
- The delete operation is more efficient if the node to be deleted is given. (**Think! you will get the answer in the second half of this blog)**
- The insert operation is more efficient if the node is given before which insertion should take place. (**Think!**)

Disadvantages over Singly Linked List-

- It will require more space as each node has an extra memory to store the address of the previous node.
- The number of modification increase while doing various operations like insertion, deletion, etc.

Circular Linked List

A circular linked list is either a singly or doubly linked list in which there are no ***NULL*** values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List. In the case of a singly linked list, the next of the last node contains the address of the first node and in case of a doubly-linked list, the next of last node contains the address of the first node and prev of the first node contains the address of the last node.



Circular Linked List

Advantages of a Circular linked list

- The list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- We can easily traverse to its previous node in a circular linked list, which is not possible in a singly linked list. **(Think!)**

Disadvantages of Circular linked list

- If not traversed carefully, then we could end up in an infinite loop because here we don't have any ***NULL*** value to stop the traversal.
- Operations in a circular linked list are complex as compared to a singly linked list and doubly linked list like reversing a circular linked list, etc.

Basic Operations on Linked List

- **Traversal**: To traverse all the nodes one after another.
- **Insertion**: To add a node at the given position.
- **Deletion**: To delete a node.
- **Searching**: To search an element(s) by value.
- **Updating**: To update a node.
- **Sorting:** To arrange nodes in a linked list in a specific order.
- **Merging:** To merge two linked lists into one.

We will see the various implementation of these operations on a singly linked list.

Following is the structure of the node in a linked list:

```
class Node{
    int data // variable containing the data of the node
    Node next // variable containing the address of next node
}
```

Linked List Traversal

The idea here is to step through the list from beginning to end. ***For example***, we may want to print the list or search for a specific node in the list.

The algorithm for traversing a list

- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

```
void traverseLL(Node head) {
    while(head != NULL)
    {
        print(head.data)
        head = head.next
    }
}
```

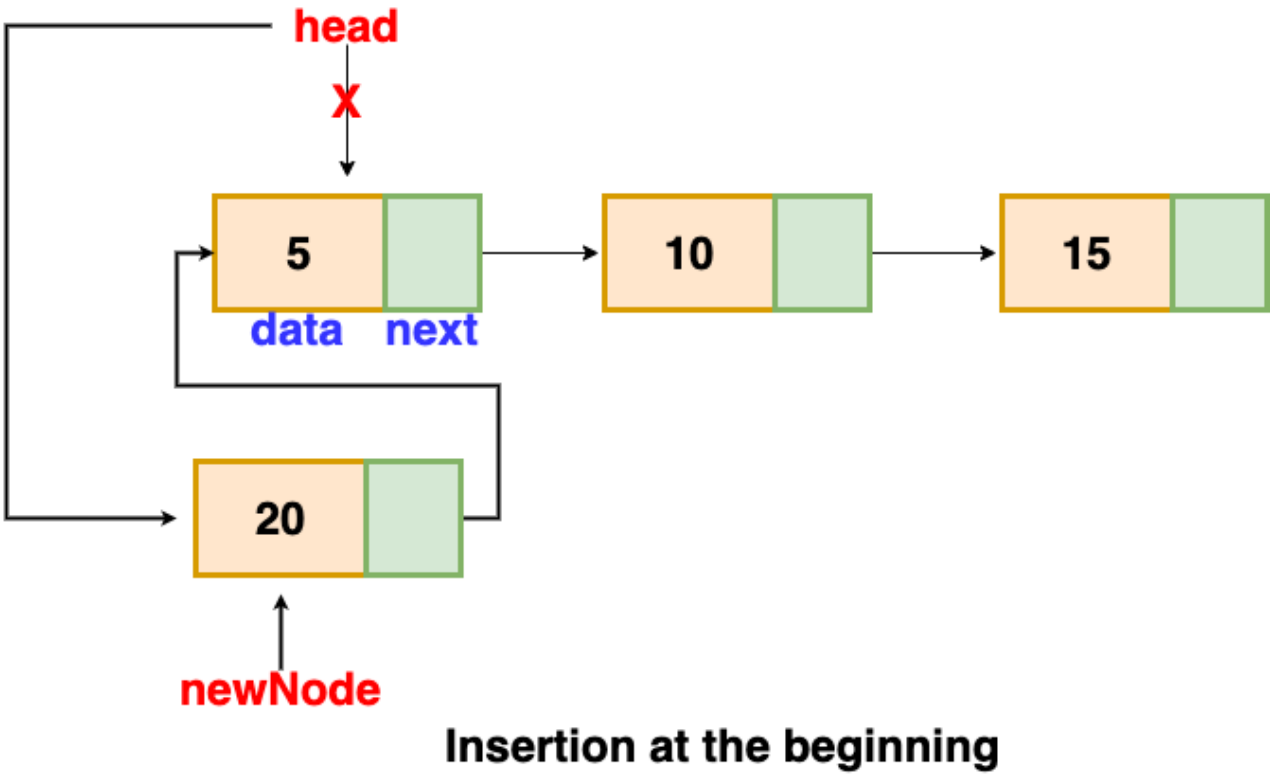
Linked List node Insertion

There can be three cases that will occur when we are inserting a node in a linked list.

- Insertion at the beginning
- Insertion at the end. (Append)
- Insertion after a given node

Insertion at the beginning

Since there is no need to find the end of the list. If the list is empty, we make the new node as the head of the list. Otherwise, we we have to connect the new node to the current head of the list and make the new node, the head of the list.



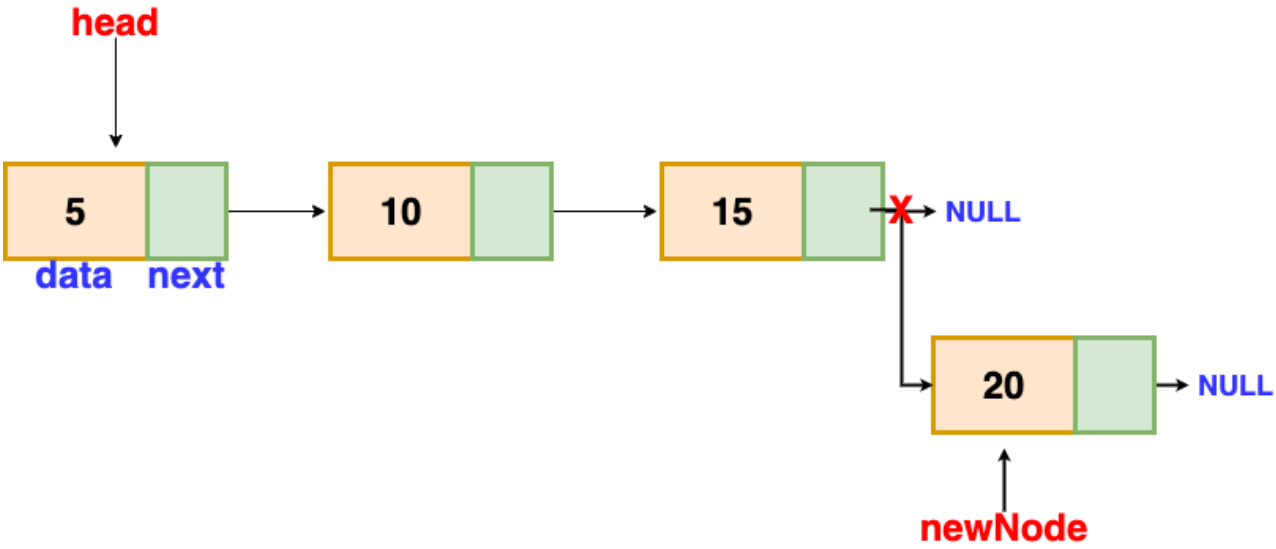
Insertion at the beginning

```
// function is returning the head of the singly linked-list
Node insertAtBegin(Node head, int val)
{
    newNode = new Node(val) // creating new node of linked list
    if(head == NULL) // check if linked list is empty
        return newNode
    else // inserting the node at the beginning
    {
        newNode.next = head
        return newNode
    }
}
```

Insertion at end

We will traverse the list until we find the last node. Then we insert the new node to the end of the list. Note that we have to consider special cases such as list being empty.

In case of a list being empty, we will return the updated head of the linked list because in this case, the inserted node is the first as well as the last node of the linked list.

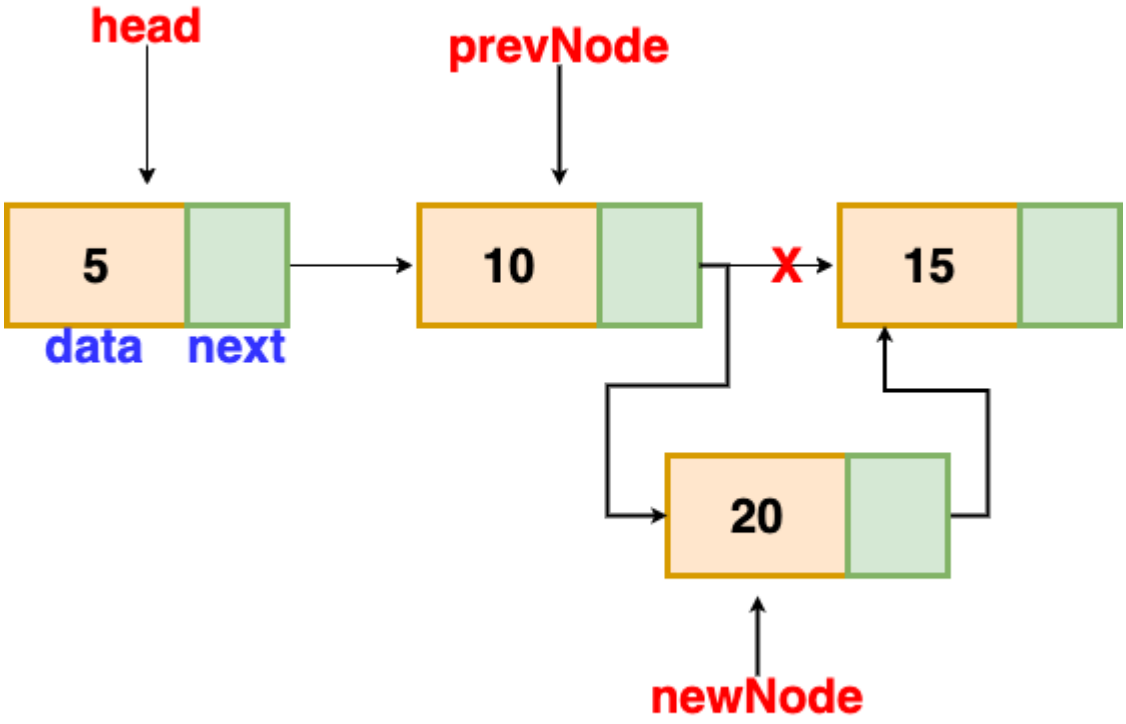


Insertion at the end

```
// the function is returning the head of the singly linked list
Node insertAtEnd(Node head, int val)
{
    if( head == NULL ) // handing the special case
    {
        newNode = new Node(val)
        head = newNode
        return head
    }
    Node temp = head
    // traversing the list to get the last node
    while( temp.next != NULL )
    {
        temp = temp.next
    }
    newNode = new Node(val)
    temp.next = newNode
    return head
}
```

> **Insertion after a given node**

We are given the reference to a node, and the new node is inserted after the given node.



Insertion after a given node

```
void insertAfter(Node prevNode, int val)
{
    newNode = new Node(val)

    newNode.next = prevNode.next
    prevNode.next = newNode
}
```

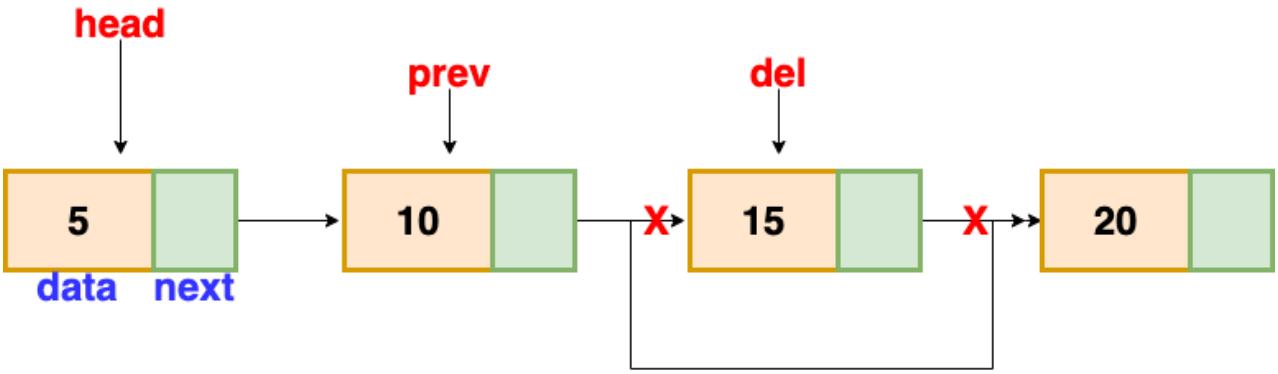
NOTE: If the address of the ***prevNode*** is not given, then you can traverse to that node by finding the data value.

Linked List node Deletion

To delete a node from a linked list, we need to do these steps

- Find the previous node of the node to be deleted.
- Change the next pointer of the previous node
- Free the memory of the deleted node.

In the deletion, there is a special case in which the first node is deleted. In this, we need to update the head of the linked list.



Deleting a Node in Linked List

```
// this function will return the head of the linked list
Node deleteLL(Node head, Node del)
{
    if(head == del) // if the node to be deleted is the head node
    {
        return head.next // special case for the first Node
    }
    Node temp = head

    while( temp.next != NULL )
    {
        if(temp.next == del) // finding the node to be deleted
        {
            temp.next = temp.next.next
            delete del // free the memory of that Node
            return head
        }
        temp = temp.next
    }
    return head // if no node matches in the Linked List
}
```

Linked List node Searching

To search any value in the linked list, we can traverse the linked list and compares the value present in the node.

```
bool searchLL(Node head, int val)
{
    Node temp = head // creating a temp variable pointing to the head of the
linked list
    while( temp != NULL) // traversing the list
    {
        if( temp.data == val )
            return true
        temp = temp.next
    }
    return false
}
```

Linked List node Updation

To update the value of the node, we just need to set the data part to the new value.

Below is the implementation in which we had to update the value of the first node in which data is equal to **val** and we have to set it to **newVal**.

```
void updateLL(Node head, int val, int newVal)
{
    Node temp = head
    while(temp != NULL)
    {
        if( temp.data == val)
        {
            temp.data = newVal
            return
        }
        temp = temp.next
    }
}
```

Suggested Problems to solve in Linked List

Happy coding! Enjoy Algorithms.