Data Structures Lab 03

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Linked List

Like arrays, Linked List is a **linear data structure**. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using **pointers**.

A **singly linked list** is indeed a type of linked list where each element (node) consists of two parts:

- **Data:** This part of the node holds the **actual value or data** that you want to store in the list. It can be any data type, such as integers, characters, or custom data structures.

- **Pointer to the Next Node:** This part of the node contains a reference or pointer to the next node in the list. This pointer helps maintain the structure of the linked list and allows you to traverse the list in a unidirectional manner, usually from the **head (the first node)** to the last node. The last node typically points to NULL to indicate the end of the list.

Singly Linked List

Implement a singly Linked List class:

It is called Singly as it holds **one data member and one link member** associated to each node in the list. In order to create a singly Linked List class, we first need a **helper class** to implement nodes. Each node object will have **3 public data members**:

(a) Int type Key (unique to each node)

(b) Data

(c) Next Pointer.

The singly Linked List class will only have a public Node type pointer variable **"head"** to point to the first node of the list. Together with the help of default and parameterized constructor the **head's value** will be manipulated in the singly Linked List class.

//Node Class

```
Class Node
{
      //public members: key, data, next
       Node ()
      {
            //initialize both key and data with zero while next pointer with
      NULL.
      }
      Node (int k, int d)
      {
            //assign the data members initialized in default constructor to
      these arguments.
      }
};
```

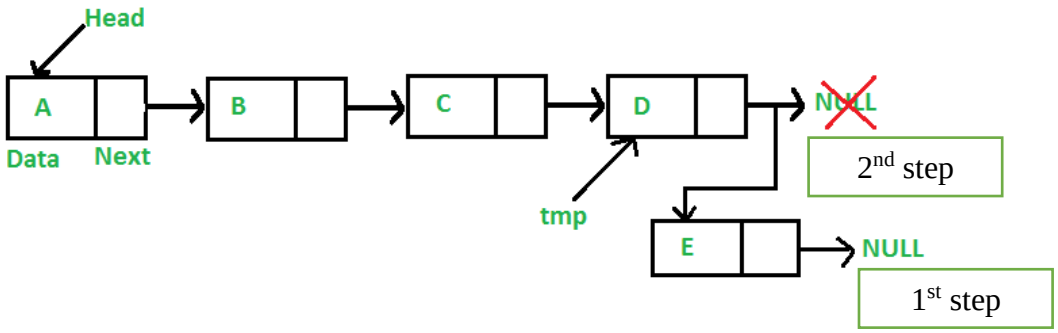
//SinglyLinkedList Class

```
Class SinglyLinkedList
{
      // create head node as a public member
      //Default Constructor
      SinglyLinkedList()
      {
            //initialize the head pointer with NULL;
      }
      //Parameterized Constructor with node type 'n' pointer
      SinglyLinkedList (Node* n)
      {
            //Assign the head pointer to value of pointer n;
      }
};
```

Add a Node at the End:

The new node is always added after the last node of the given Linked List. For example, if the given Linked List is 20->1->2->5->10 and we add an item 25 at the end, then the Linked List becomes 20->1->2->5->10->25.

Since a Linked List is typically represented by the head of it, we have to **traverse the list till the end** and then change the next to last node to a new node.



To append a node at the end of the list, first check if there **exists a node already** with that key or not. For that you may need a helper function inside the singly Linked List class to perform the test. In case a node already exists with that key value, Intimate the programmer to use another key value to append a node. On the contrary, If the node doesn't exist, append a node at the end. Before that check if the list has some node or not, i.e., Check if the **head pointer is null or not**. If it is null, access the head and assign node n to it. Otherwise, traverse through the list to find which node's **next is Null**, i.e., the last node in the list. Then, assign the node n to next pointer of the last node. Also, make sure the next pointer of node n (new last node) is null.

//Creating a Node type helper Function named node Exists (argument key)

```
 Node nodeExists (int k)
{
      //temporary pointer var to hold Null value;
      //node type pointer to hold head pointers value;
      // loop condition (Traverse through the node until the ptr variable
      points to null)
      {
       If the ptr variable's key = passed key argument
      {
            //assign the pointer ptr to temp variable;
      }
       Else
      {
            //assign the pointer ptr to point to the next pointer's value.
      }
      return the temp variable;
}
```

//Append 1st approach Function

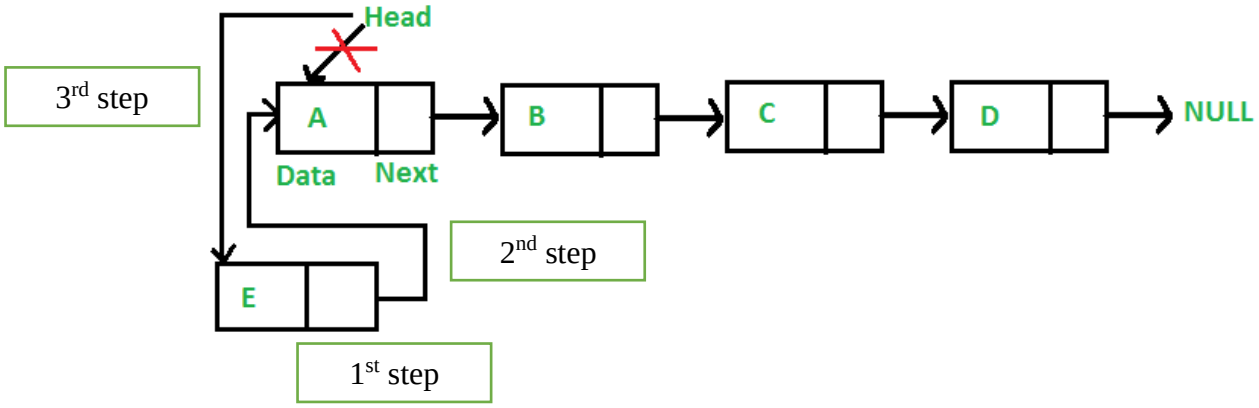
```
 Void appendNode (int key, int data)
{
     //create new node and assign data to it.
     //make next of new node to Null.

     //Node pointer temp assign to head.

     // loop condition (Traverse through the node until the ptr variable
     points to null)
     {
               //assign the pointer temp to next temp value;

     }
     //assign temp next to the new node.
}
```

//Append 2nd approach Function

```
 Void appendNode (Node* n, int data)
{
     //create new node and assign data to it.
     //check if the head pointer points to Null or not with a condition
      if (head == NULL)
     {
          //If it does, assign the head pointer the passed pointer 'n'. This
     will put the address of node n in the head pointer.
     }

     // Else traverse through the list to find the next pointer holding Null
     as address (last node)
     Else
     {
          if (nodeExists(n->key) != NULL  ) {
              // Print an intimation that a node holding passed key already
          exists.
          }
          else
          {
               If (head != NULL)
               {
               // assign the head pointer to a new pointer of type Node (say,
               ptr).
               //loop through the list using ptr until the next of any node
               contains null.
               //when ptr->next = NULL. Then assign the n node to ptr->next
               to append that node after the last found node.
               }} }}
```

Add a Node at the Front:

The new node is always added before the head of the given Linked List. The newly added node becomes the new head of the Linked List. For example, if the given Linked List is 20->10->15->17 and we add an item 5 at the front, then the Linked List becomes 5->20->10->15->17.



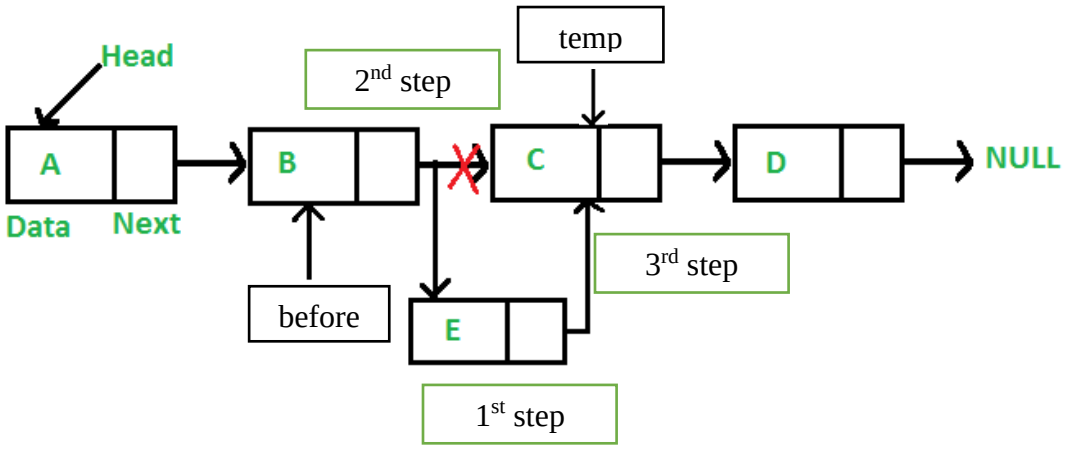
To prepend a node, simply check if the key that is passed already exists or not, if yes, intimate the programmer to pass a new key value or else assign the new head node's value to the next pointer of new node. Then set the new head to be the new node's address.

//Prepending a Node.

```
  void prependNode(Node* n)
 {
 // checking the value of node's key if the node with that key already
exists.

     if (nodeExists(n->key) != NULL  ) {
    // Print an intimation that a node holding passed key already exists.
     }
     else
     {
       // new node's next is pointing to the head i.e. address of first node.
       // since we have changed the head pointer's value from first node to
the new node, now the new.
       // head will be pointing to address of new node.
     }
 }
```

Insert a New Node After Some Nodes:

Consider a singly linked list 4 -> 1 -> 5 - > 7 -> 2 and want to add the value 3 after node 1. The List should be transformed as 4 -> 1 -> 3-> 5-> 7 -> 2. To insert a new node after some node, create a void function named insertNodeAfter () carrying two arguments, one for the key of the node after which the insertion is to be done, the new node.

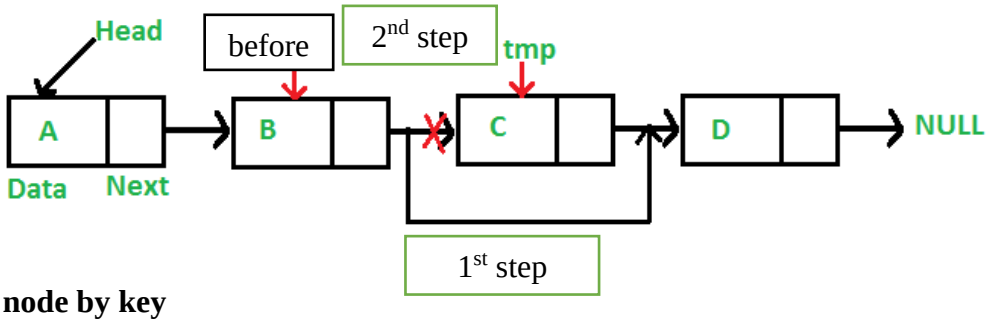


//Inserting a new node after some node.

```
Void insertAfterNode (int key , int data)
{
          If {
               //key already zero then insert node.
         }

     Else {
               //create new node and assign data to it.
               //Node pointer temp assign to head
             //Make a before node pointer
          // loop condition (Traverse through the node until the temp
     variable points to null)
     {
          // if temp key equal to coming key. Tell that the key is already
     exist.
          // else {
                    //if temp key greater than insert key {
                     //Node pointer temp_node assign temp
                     //assign before next to new node
                     //assign new node next to traverse
                   }
                }
          //assign before to temp
          //temp assign to temp next
     }

}
```

Delete an existing Node:
Consider a singly linked list 4 -> 1 -> 5 - > 7 -> 2 and want to delete the value 5. The List should be transformed as 4 -> 1 -> 7 -> 2.

To delete a node from the linked list, we need to do the following steps.
1) Find the previous node of the node to be deleted.
2) Change the next of the previous node to hold the node next to the node to be deleted.
3) Free memory for the node to be deleted.



// Delete a node by key

```
void deleteNode(key)
{
   // create a Node type pointer to hold head (say, del)
   // create a Node type pointer to hold previous node (say, prev)

   // check if head contains the key
   if (del!=NULL && del ->key==key){
        // assign next of del to del, which will unlink the node pointed
by head
        // delete the del node
        // return
        }
   Else{
        While(del!=NULL && del ->key != key){   // traverse the list
                                                until temp is not NULL //and del's
                                                key is same as the key.
             // set prev to del
             // set del to del's next pointer
        }
        If(del == NULL){    // key not found.
             // return
             }
        // unlink by setting del's next to prev's next.
        // free memory by deleting del
        }
}
```

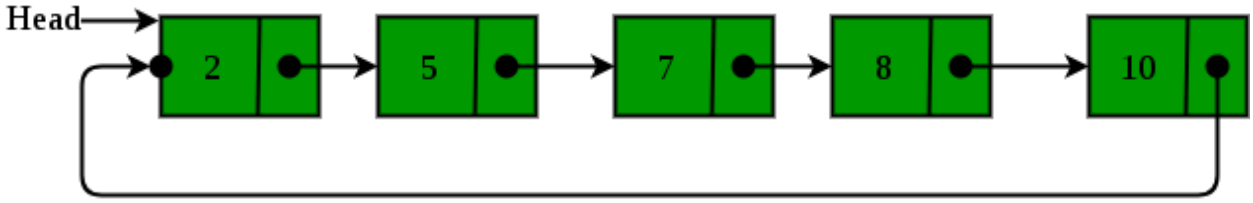
Update the Data of Nodes:
Consider a singly linked list 4 -> 1 -> 5 - > 7 -> 2 and want to update the node 7 to 6. The List should be transformed as 4 -> 1 -> 5-> 6 -> 2.

Updating Linked List or modifying Linked List means replacing the data of a particular node with the new data.

//Update a node's data by key

```
void updateNode(key, new_data)
{
        // call the nodeExists function and hold the return value in a Node type pointer (say, ptr)
        If(ptr!=NULL){
                // set ptr's data as new data
        }
        Else{
                // print a message sating node does not exist.
        }
}
```

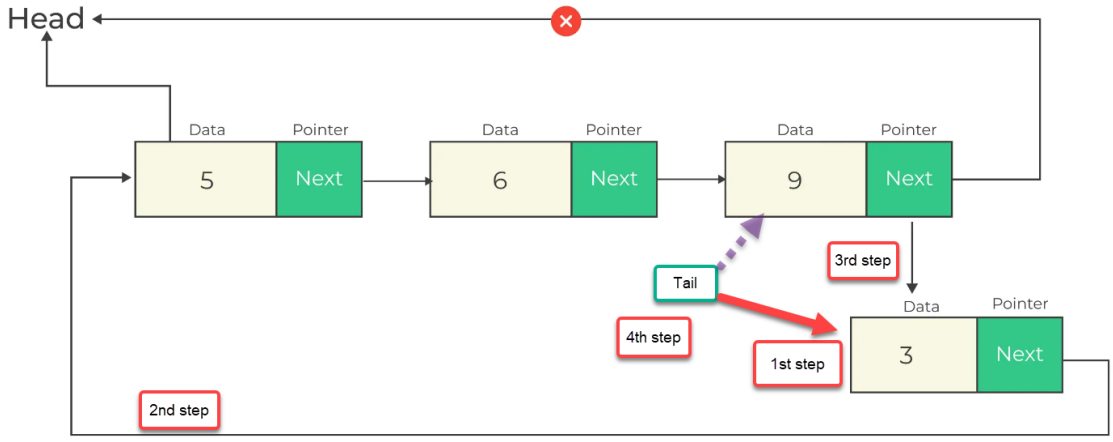
Circular Link List



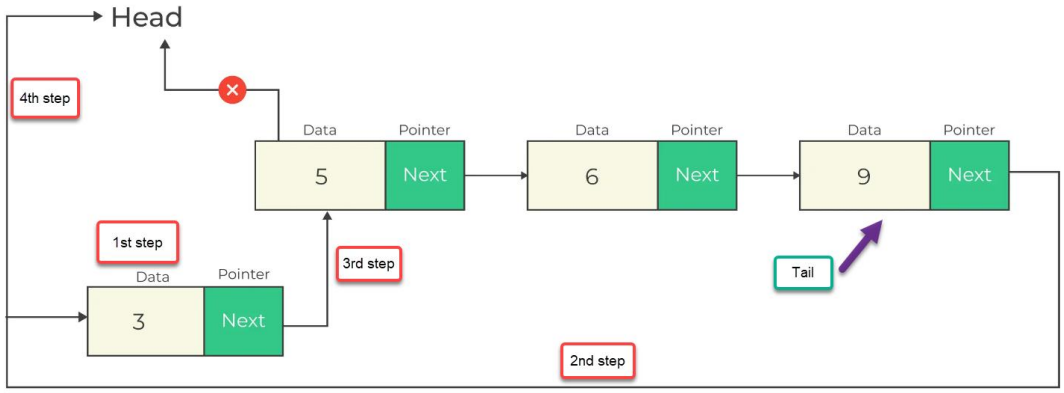
The **circular linked list** is a linked list where all nodes are connected to form a circle. In a circular linked list, the first node and the last node are connected to each other which forms a circle. There is no NULL at the end.

```
class Node {
     public:
          int key;
          int data;
          Node * next;

          Node() {
               key = 0;
               data = 0;
               next = NULL;
          }

          Node(int k, int d) {
               key = k;
               data = d;
          }
};

class CircularLinkedList {
     public:
          Node * head;
          Node * tail;

          CircularLinkedList() {
               head = new Node(k,d);
               tail = head;
          }
          appendNode();
          prependNode();
          insertNodeAfter();
          deleteNodeByKey();
          updateNodeByKey();
          print();
};
```

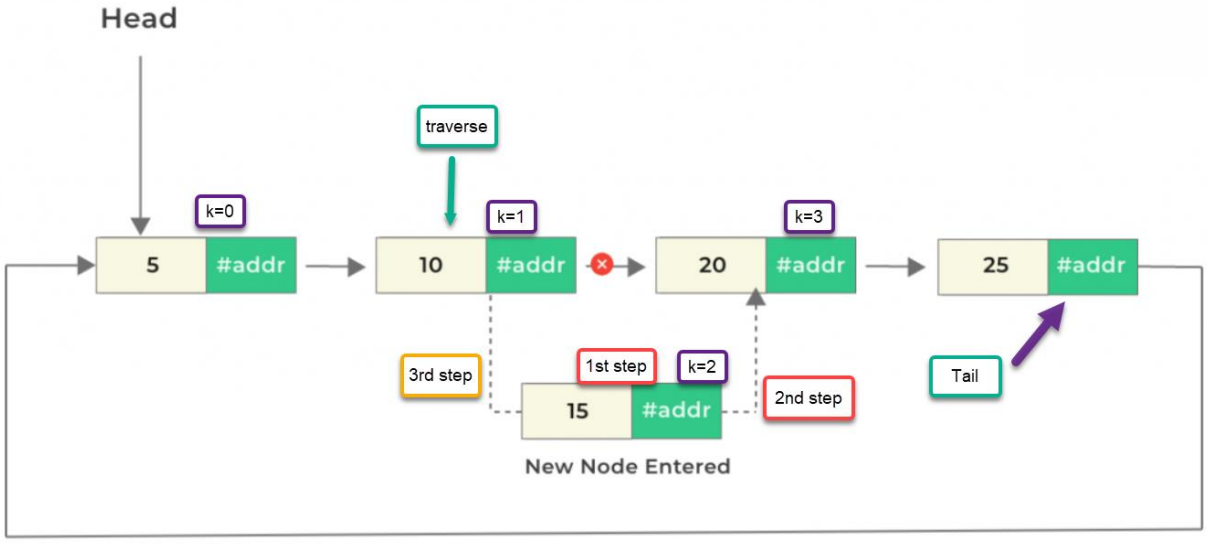
i. Insert a new node at the end of the list.



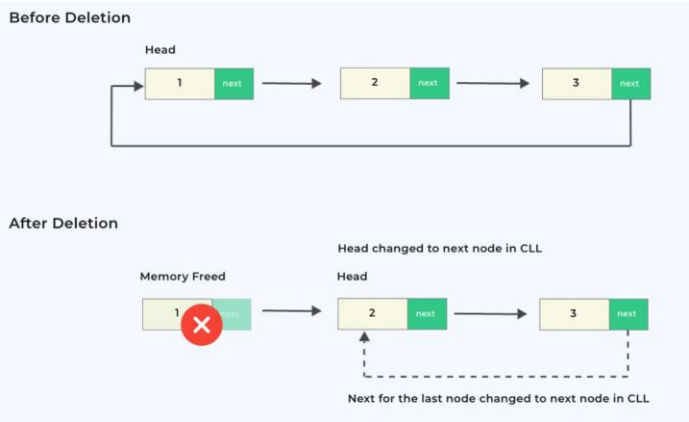
ii. Insert a new node at the beginning of list.

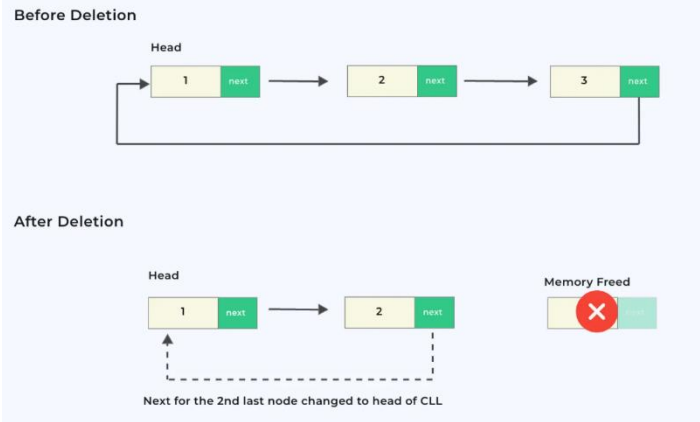


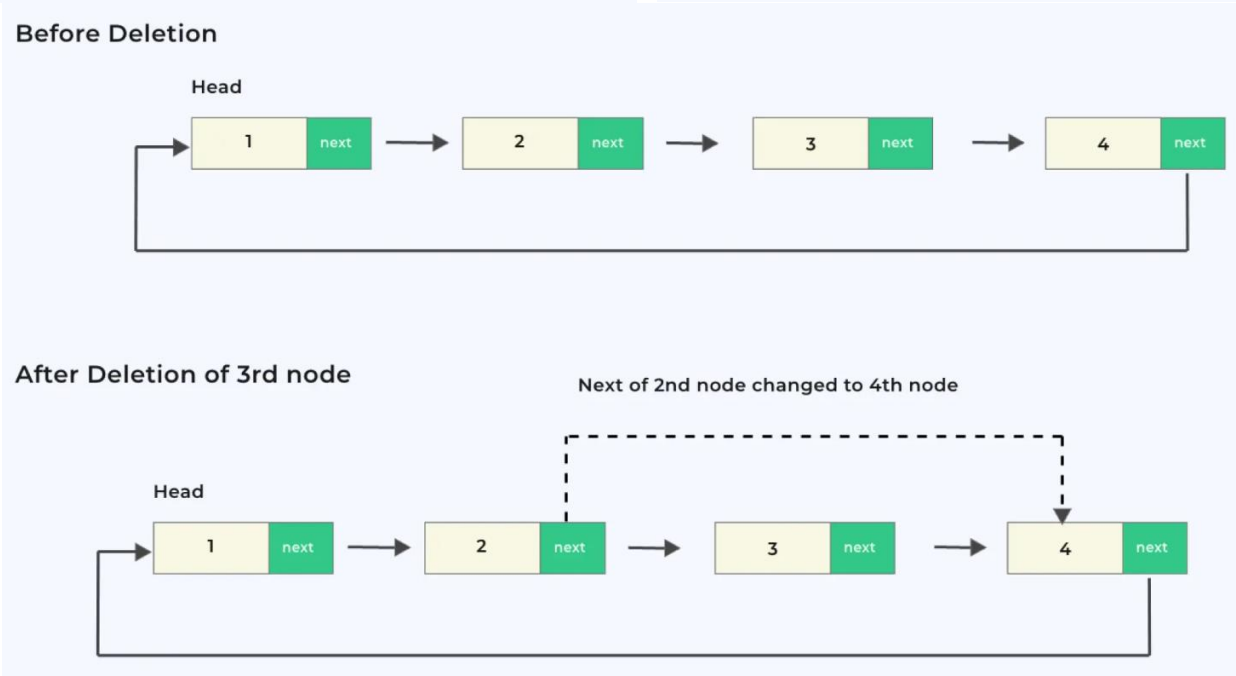
iii. Insert a new node at given position.



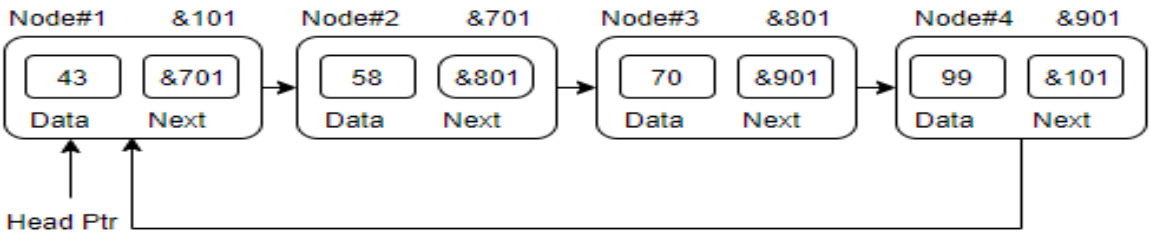
iv. Delete any node.



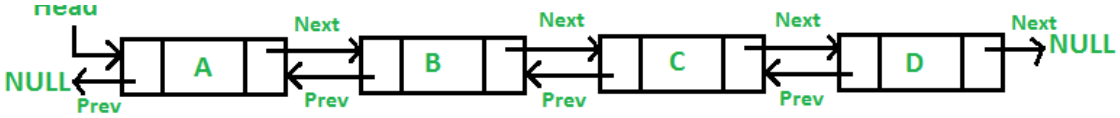




v. Print the complete circular link list.



Doubly Link List

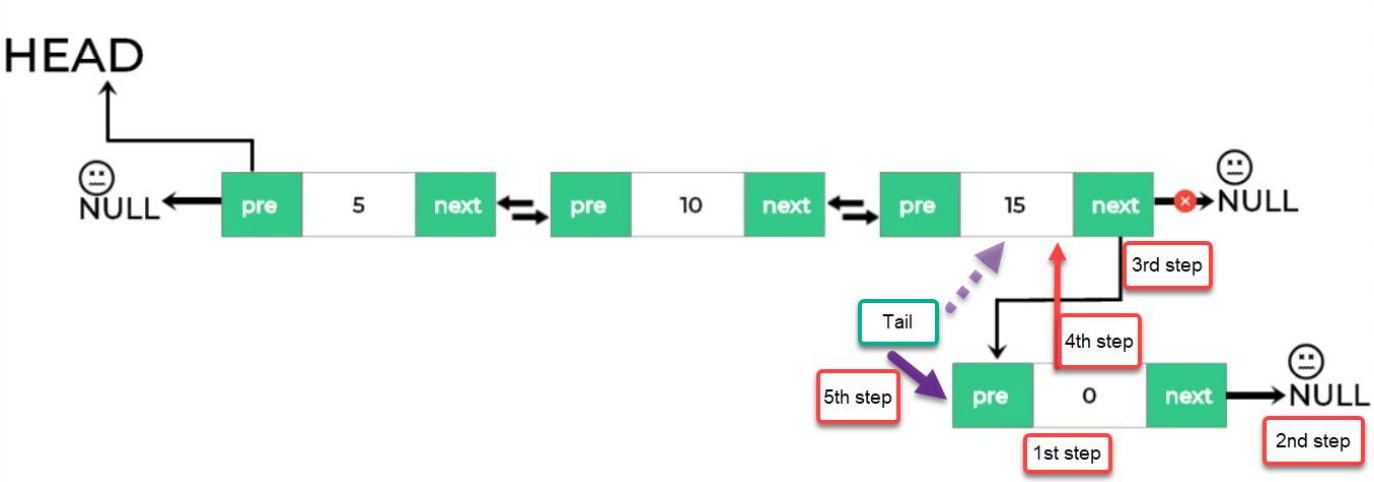


Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward or backward easily as compared to Single Linked List.

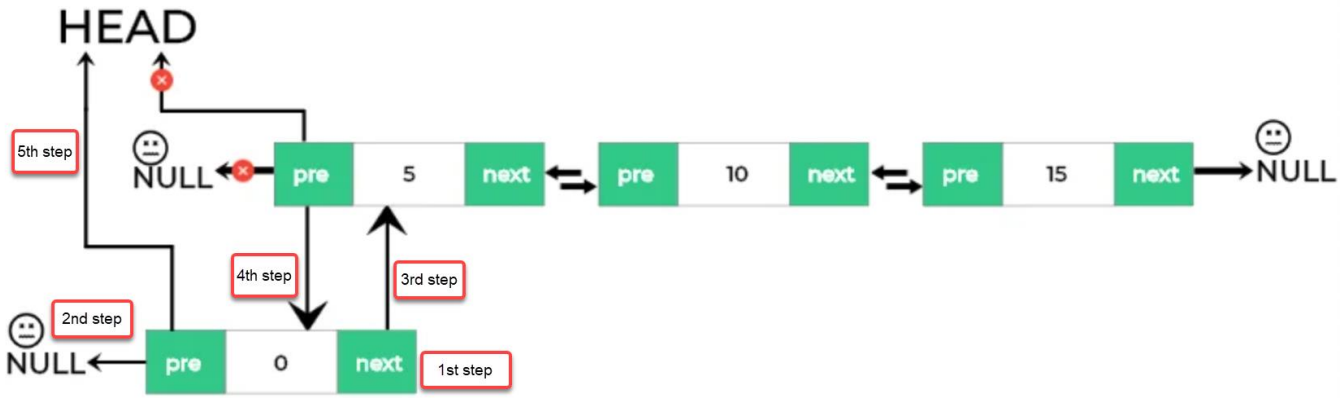
- **Link** − each link of a linked list can store a data called an element.
- **Next** − each link of a linked list contains a link to the next link called Next.
- **Prev** − each link of a linked list contains a link to the previous link called Prev.

```
class Node {
     public:
          int key;
          int data;
          Node * next;
          Node * previous;

          Node() {
               key = 0;
               data = 0;
               next = NULL;
               previous = NULL;
          }

          Node(int k, int d) {
               key = k;
               data = d;
          }
};

class DoublyLinkedList {
     public:
          Node* Head;
          Node* Tail;
          DoublyLL(int k,int d){
                  Head=new Node(k,d);
                  Tail=Head;
          }
          appendNode();
          prependNode();
          insertNodeAfter();
          deleteNodeByKey();
          updateNodeByKey();
};
```

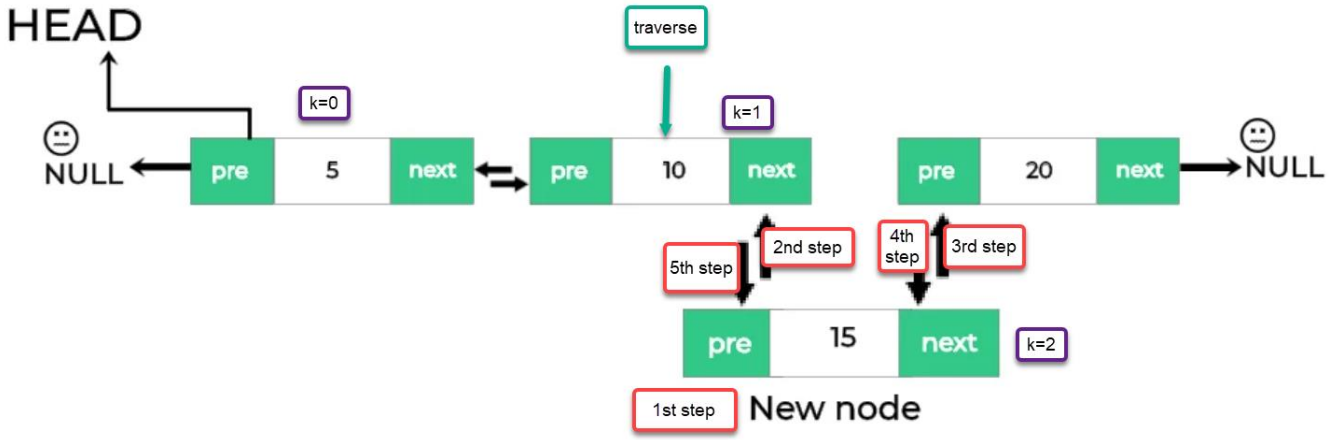
i. Insert a new node at the end of the list.



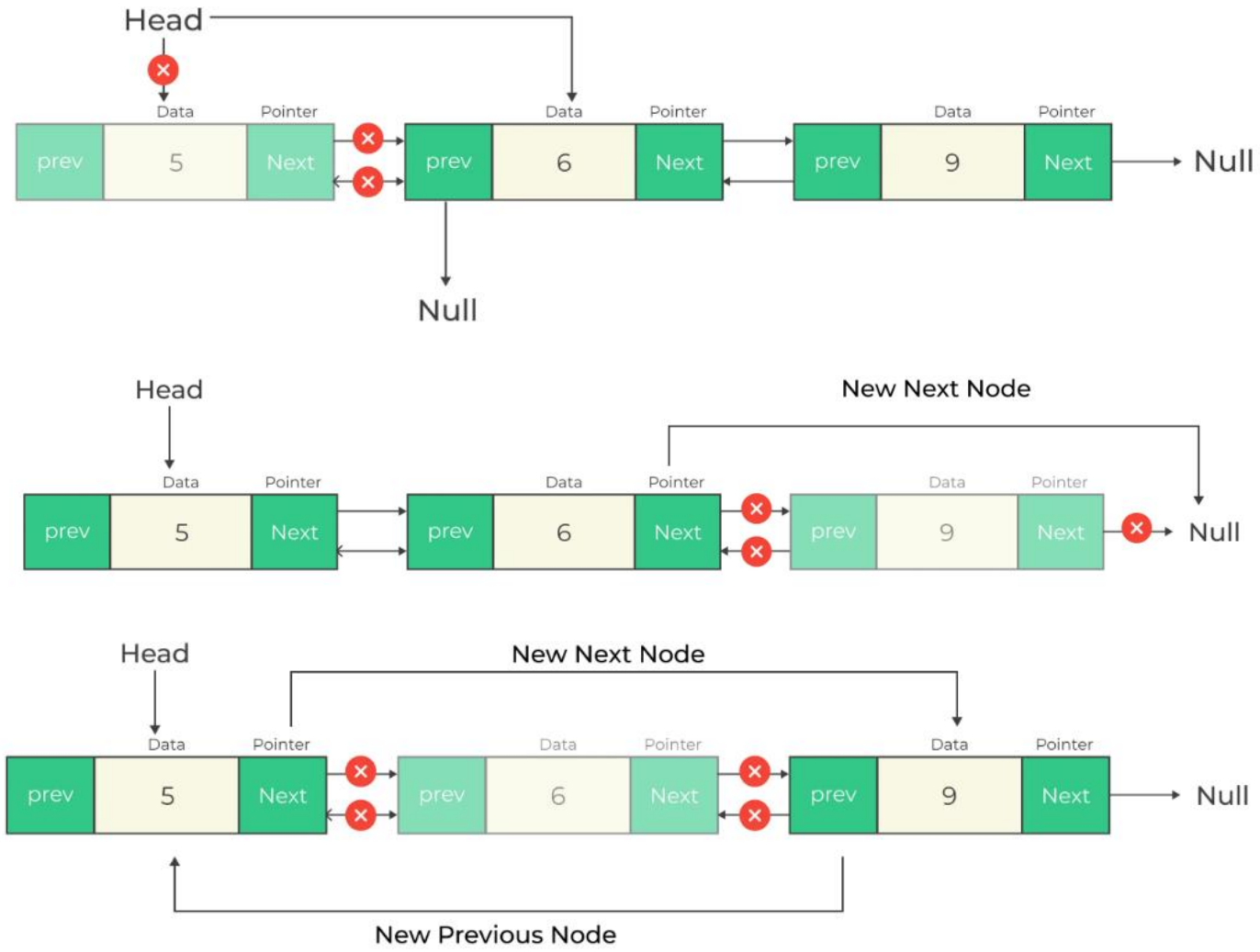
ii. Insert a new node at the beginning of list.



iii. Insert a new node at given position.

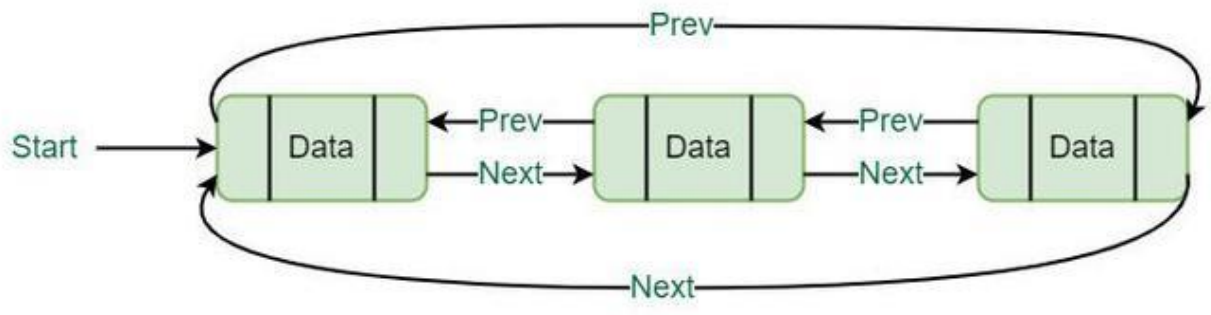


iv. Delete any node.



v. Print the complete doubly link list.

Circular Double Link List

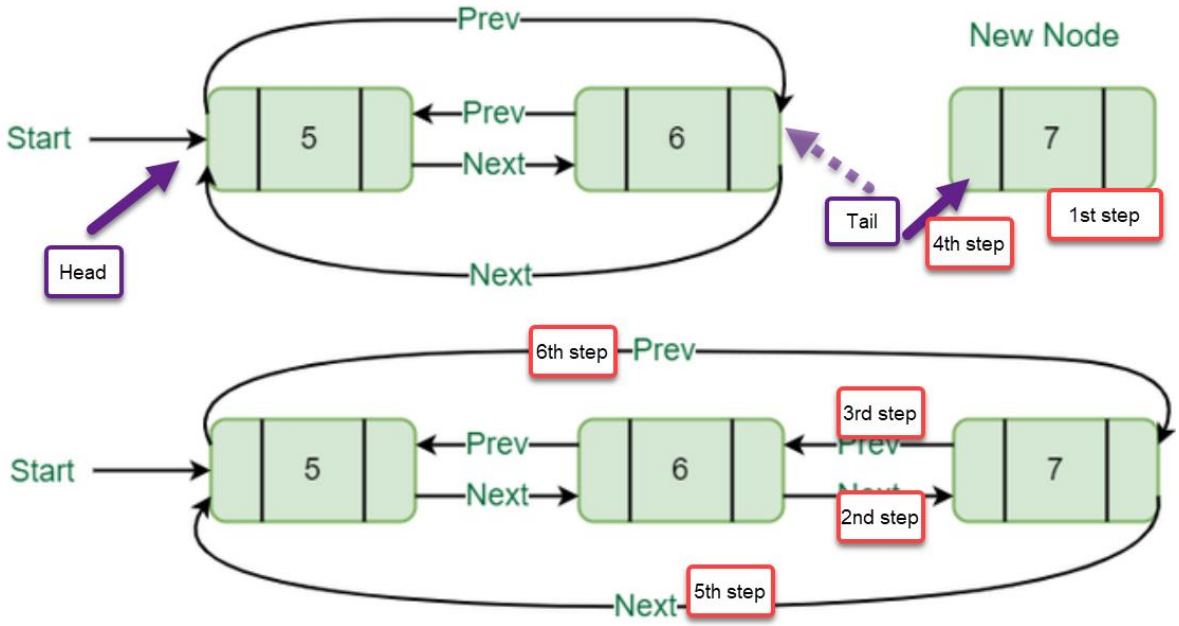


In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.

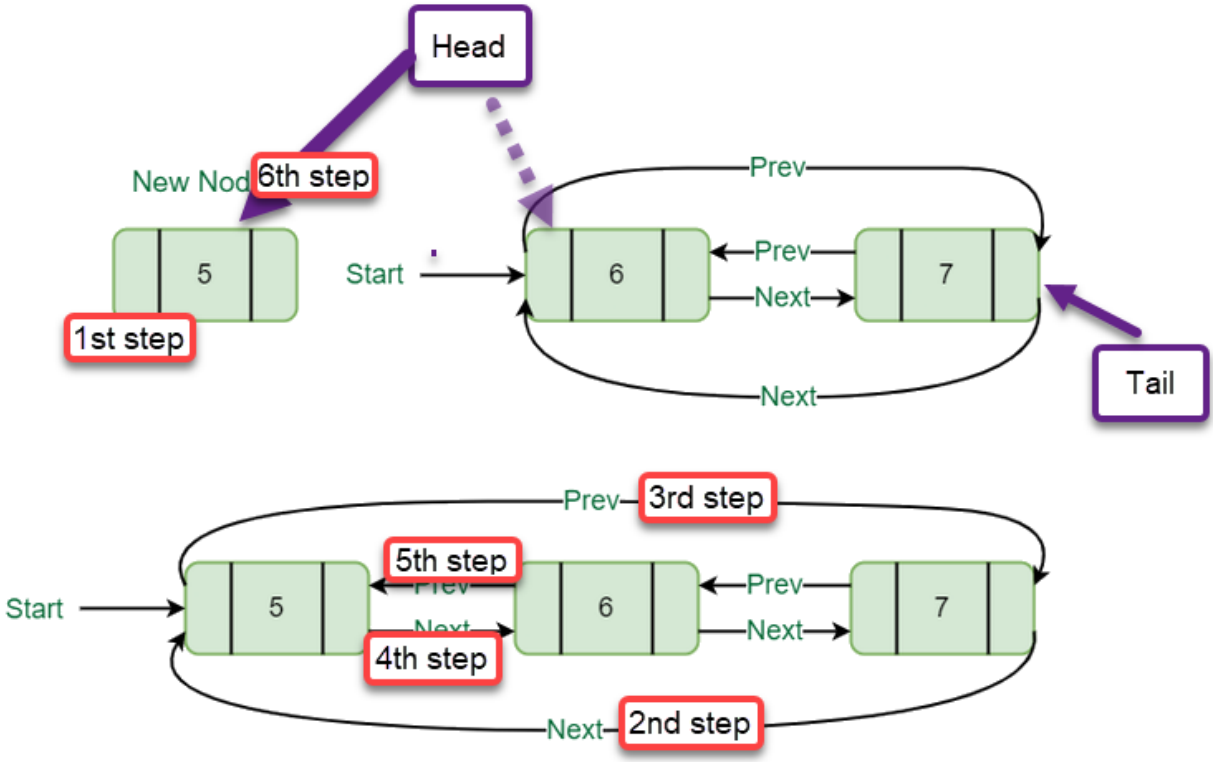
- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

```
class  node {
public:
          int key;
          node *next;
          node *prev;
};

class double_clist {
     public:
          Node* Head;
          Node* Tail;
          DoublyCircularLL(int k,int d){
                Head=new Node(k,d);
                Tail=Head;
          }
          insert_begin();
          insert_last();
          insert_pos();
          delete_pos();
          update();
          display();
          }
};
```

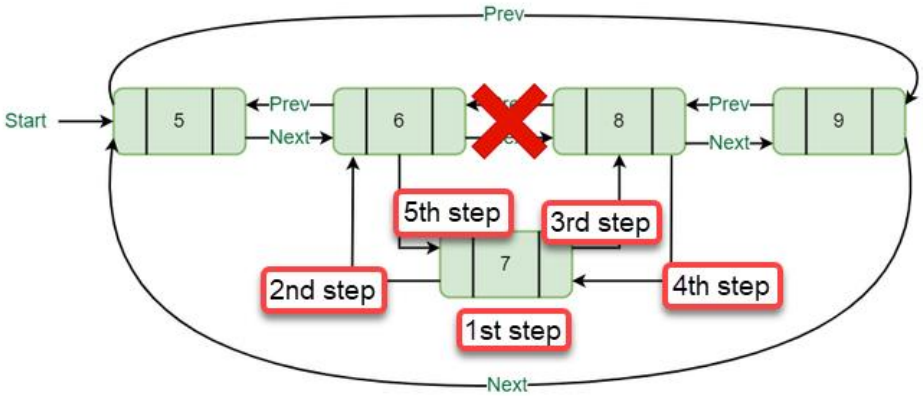
i. Insert a new node at the end of the list.



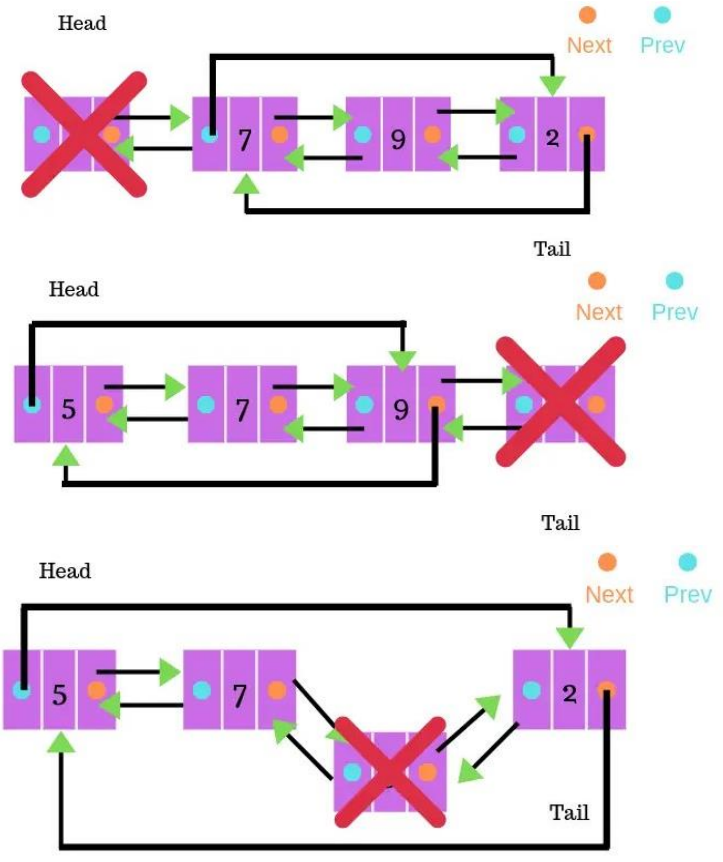
ii. Insert a new node at the beginning of list.



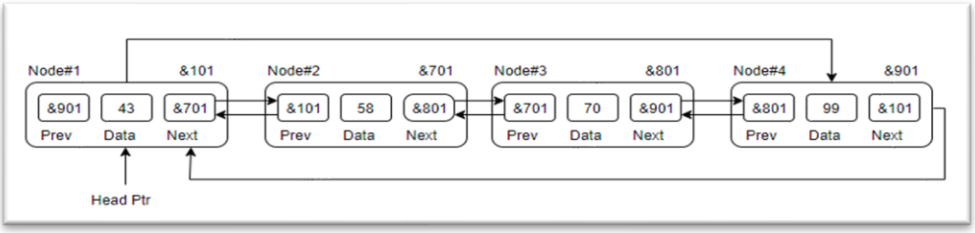
iii. Insert a new node at given position.



iv. Delete any node.



v. Print the complete circular double link list.



Lab Exercise

Task # 1

Create a Singly Linked List and perform the mentioned tasks:

1. Insert a new node at the end of the list.
2. Insert a new node at the beginning of list.
3. Insert a new node at given position.
4. Delete any node.
5. Update the data of node.
6. Print the complete singly linked list.

Task # 2

Solve the following problem using a Singly Linked List. Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.
Examples:

```
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL
```

Task # 3

Solve the following problem using a Singly Linked List. Given a Linked List of integers or string, write a function to check if the entirety of the linked list is a palindrome or not
Examples:

```
Input: 1->0->2->0->1->NULL
Output: Linked List is a Palindrome

Input: B->O->R->R->O->W->O->R->R->O->B->NULL
Output: Linked List is a Palindrome
```

Task # 4

Create a Circular Singly Linked List and perform the mentioned tasks:

1. Insert a new node at the end of the list.
2. Insert a new node at the beginning of list.
3. Insert a new node at given position.
4. Delete any node.
5. Update the data of node.
6. Print the complete circular singly linked list.

Task # 5

Create a Doubly Linked List and perform the mentioned tasks:

1. Insert a new node at the end of the list.
2. Insert a new node at the beginning of list.
3. Insert a new node at given position.
4. Delete any node.
5. Update the data of node.
6. Print the complete doubly linked list.

Task # 6

Give an efficient algorithm for concatenating two doubly linked lists L and M, with head and tail preserved nodes, into a single list that contains all the nodes of L followed by all the nodes of M.

Task # 7

Given a Double Circular Linked List 1-7-4-2-6-4-5-3-9-8 your task is to swap the nodes (not values) given their indexes along with its previous and next pointers.

```
Input: Enter two nodes' keys = 4 8
Output:
Initial Linked List = 1-7-4-2-6-4-5-3-9-8
After Swapping      = 1-7-4-3-6-4-5-2-9-8
```