Linked Lists

DR. SANGA CHAKI

Contents

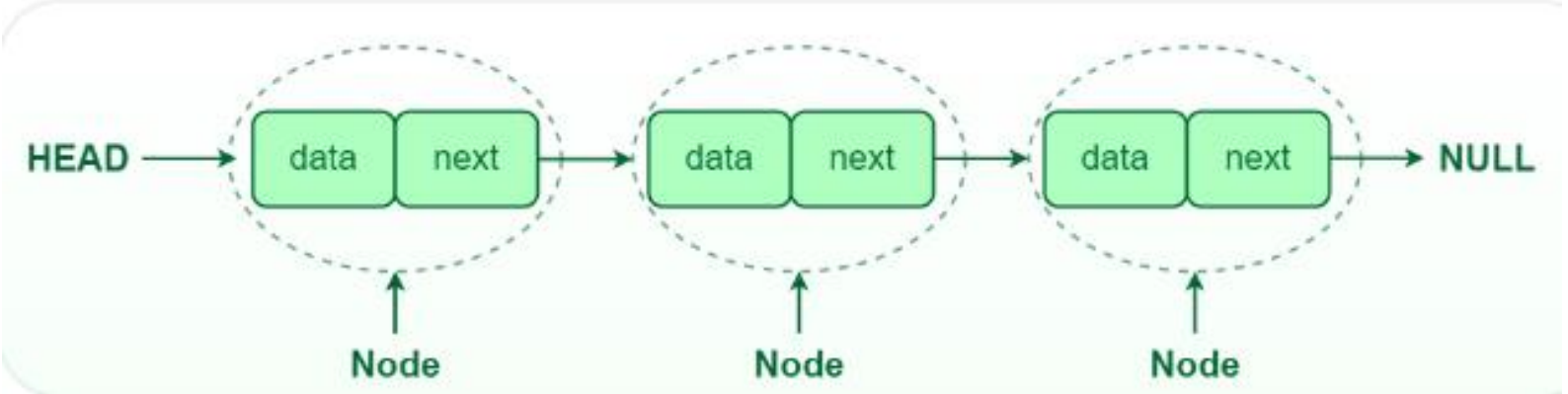
Linked Lists

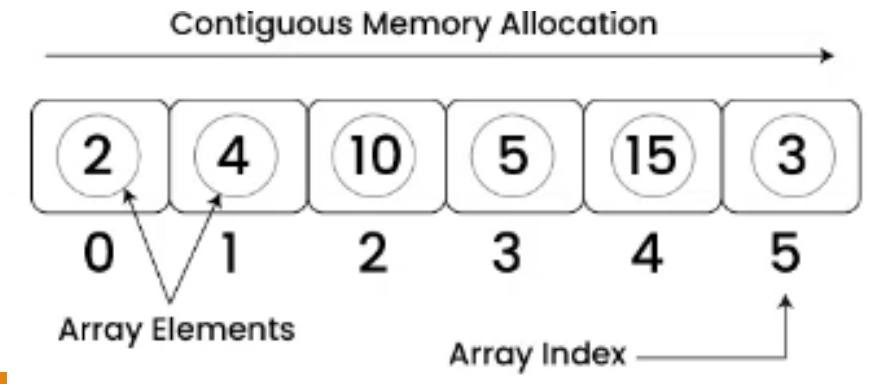
What is a linked list?

1. A linked list is a fundamental linear data structure in computer science.
2. It consists of nodes where each node contains data and a reference (link) to the next node in the sequence.
3. This allows for dynamic memory allocation and efficient insertion and deletion operations compared to arrays.

What is a linked list?

1. Data Structure for storing collection of data of same type



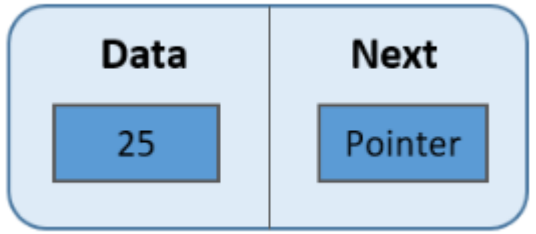


What is a linked list?

1. LL properties:
   a) Successive elements are connected by pointers and last element points to NULL
   b) Can grow/shrink in size during execution of a program – unlike arrays
   c) Max length of a linked list? – as long as necessary until memory exhausts
   d) It does not waste memory space, but takes extra memory for storing pointers

Nodes of a Linked List

1. The nodes are themselves represented by a class with two attributes
   - Data – info we want to store in the node
   - Pointer – address of the next node



2. Why pointer is needed?
   - Nodes can be allocated space anywhere in computer memory.
   - We need a way to access the next node.
   - This problem is not there in arrays – all elements are stored in contiguous memory locations from the start of the array - that is why arrays don't need pointers.

Types and Operations of Linked Lists

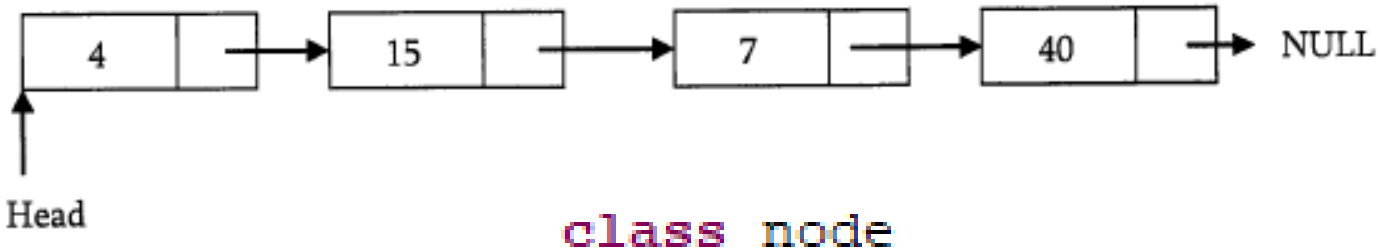
1. Depending on number of pointers in each node and their arrangement, LL can be of three types
    1. Singly linked lists
    2. Doubly linked lists
    3. Circular linked lists

1. Possible operations:
    a) Creating single nodes
    b) Insertion of nodes,
    c) Deletion of nodes,
    d) Display contents of the linked list
    e) Traversal of linked list
    f) Reversal
    g) Searching/sorting

Singly Linked Lists

Singly Linked Lists

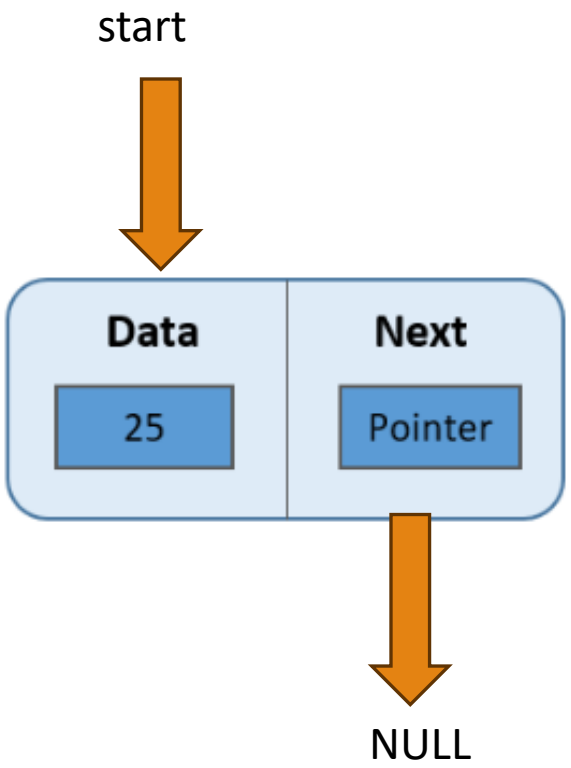
1. Collection of nodes
2. Each node has **one data member**, **and one next pointer** to the following node
3. Last node points to NULL




```
class node
{
    int info;//data member
    node next;//reference to an object of type node
}
```

Creating Nodes

1. Points to remember for creating a singly linked list with one node:
   a) Create the node object from the node class.
   b) Assign data value to the data/info attribute
   c) Assign null to pointer attribute.
   d) Assign the address of this first node, in a special node called start/head – why?
      - This is the reference address of where your linked list starts.
      - If this step is missed, there is no way to locate your linked list – it is lost.

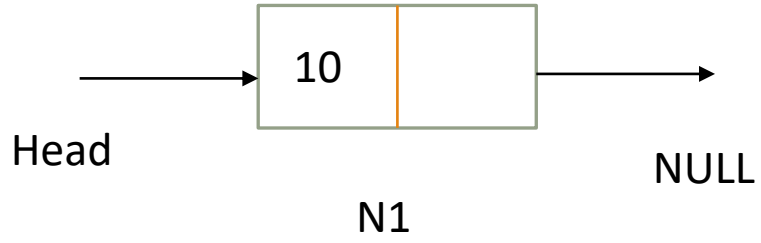


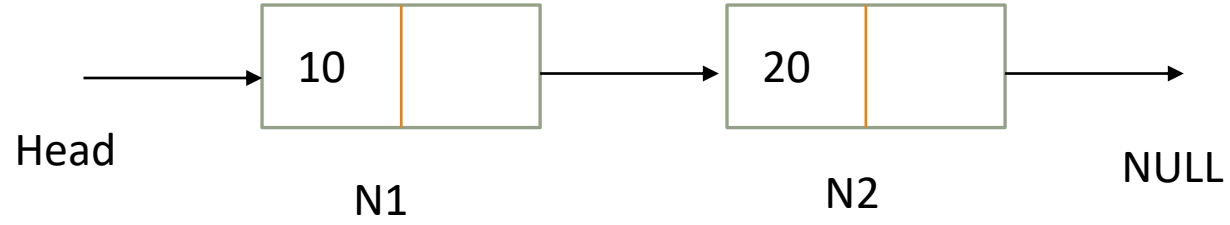
Creating One Node

```
class node{
    int info;//data member
    node next;//reference to an object of type node
}
public class Linkedlist{
 static node start=null;
 public static void create()
 {
     Scanner sc=new Scanner(System.in);
     node p=new node();
     System.out.println("Enter info");
     p.info=sc.nextInt();
     p.next=null;
     start=p;
 }
```

Adding Multiple Nodes at End of LL

1. Create the first node N1 from the node class.
2. Assign data value to the data/info attribute
3. Assign null to pointer attribute.
4. Ask if more nodes are needed – if yes
5. Create a second node N2. Assign data value/info.
6. Assign null to pointer attribute of N2.
7. Assign address of N2 to pointer/next attribute of N1

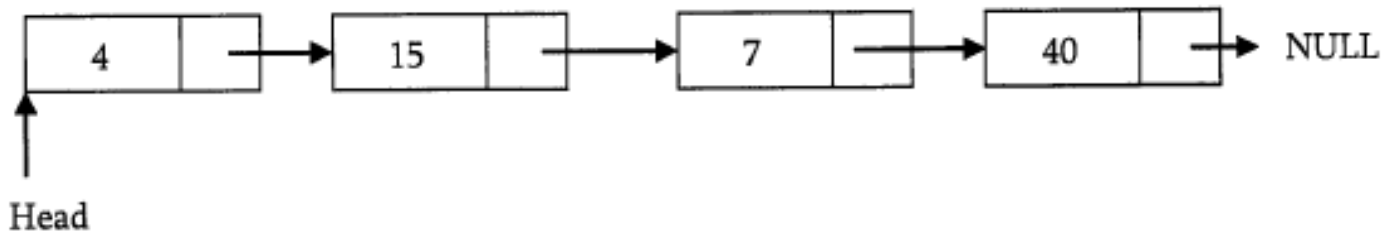




Creating Multiple Nodes

```
public class Linkedlist{
 static node start=null;
 public static void create() {
     Scanner sc=new Scanner(System.in);
     node p=new node();
     System.out.println("Enter info");
     p.info=sc.nextInt();
     p.next=null;
     start=p; //till here
     node q=p; // q points to current node
     System.out.println("Do you want to create more number of nodes(y/n)");
     char ch=sc.next().charAt(0);
     while(ch!='n')  {
         p=new node();
         System.out.println("Enter info");
         p.info=sc.nextInt();
         p.next=null;
         q.next=p;
         q=p;
         System.out.println("Do you want to create more number of nodes(y/n)");
         ch=sc.next().charAt(0);
```

Display LL by Traversing

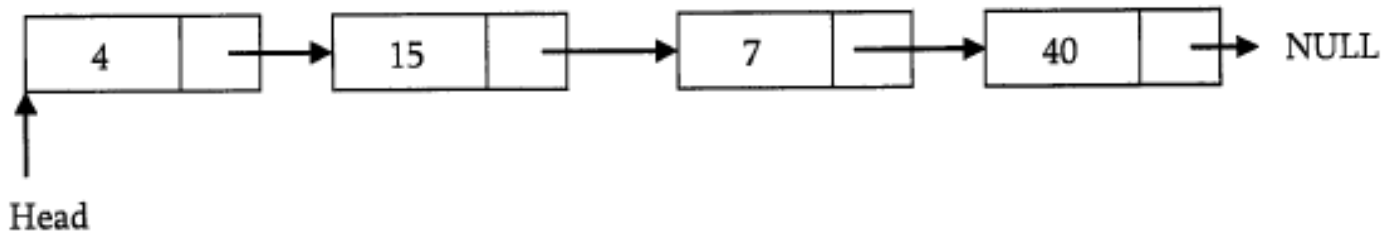


1. Start at head of LL
2. Print the info in node
3. Use pointer to next node to visit next node
4. Print info in next node
5. Continue till NULL is reached – indicating LL is exhausted

Display LL by Traversing

```
public static void display()
{
    node p=start; //p is the current node
    while(p!=null)
    {
        System.out.print(p.info+"-->");
        p=p.next;
    }
    System.out.print("NULL"); //at end it prints null
}
```

Count Number of Nodes in LL

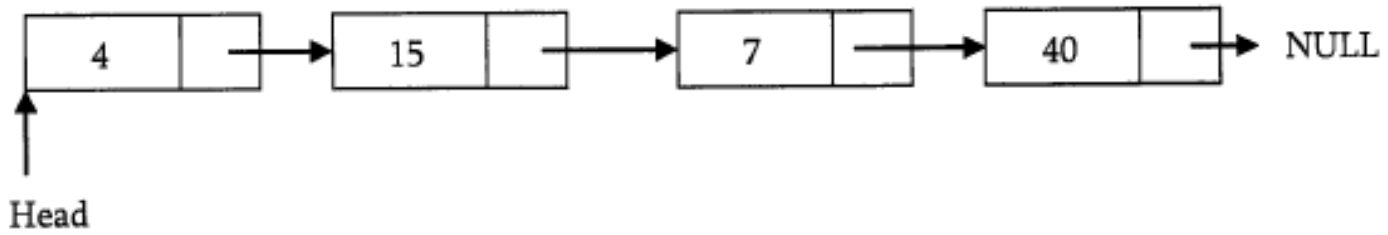


1. Initialize a counter variable to 0.
2. Start at head of LL
3. Visit the first node and increment counter by 1.
4. Use pointer to next node to visit next node
5. Increment counter by 1.
6. Continue till NULL is reached – indicating LL is exhausted

Count Number of Nodes in LL

```
public static void count() {
    node p=start; //p is the current node
    int counter=0;
    while(p!=null)
    {
        counter++;
        p=p.next;
    }
    System.out.println("\nNumber of nodes in LL = "+counter);
}
```

Search for a Key value in LL

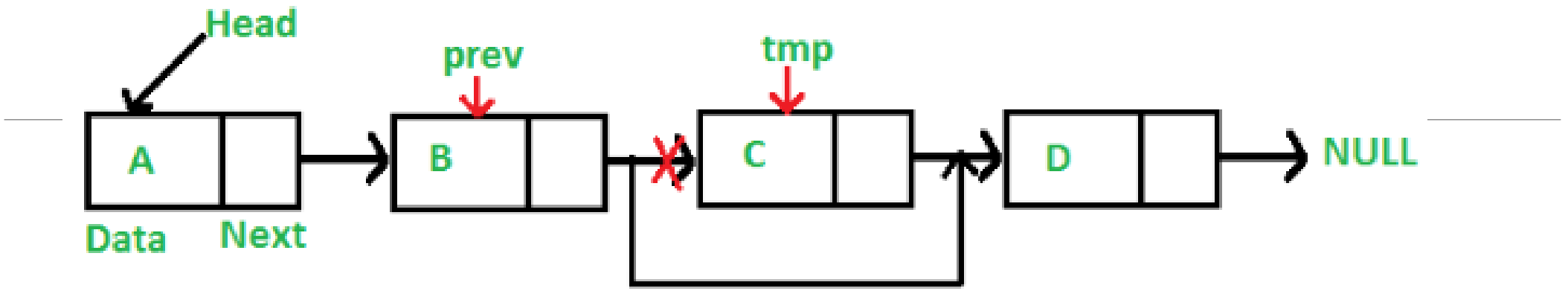


1. Start at head of LL and initialize counter to 0.
2. Visit the first node, increment counter by 1 and compare the info with the key.
3. If match, return counter value.
4. Else, use pointer to next node to visit next node
5. Increment counter by 1 and compare the info with the key
6. Continue till:
    - Either key is found
    - Or NULL is reached – indicating LL is exhausted

Search for a Key value in LL

```
public static void linear_search() {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter key to search: ");
        int n = sc.nextInt();
        node p = start; // Store head node
         int counter=0;
         while (p != null && p.info != n) {
             // If currNode does not hold key
             // continue to next node - traverse till n is found
             counter++;
             p = p.next;
         }
         if (p.info == n) {// If n is found
             counter++;
             System.out.println(n + " found at node number "+ counter + " of LL");
         }
         else
             System.out.println(n + " is not present in LL");
    }
```

Deletion of a Node in LL



1. One info value is provided and you need to delete that node which contains that info.
2. If C (in diagram) is deleted, it means that, we need to update the content of its previous node's pointer.
3. So, after deletion of C, node B should directly point to D.
4. How to do this? – when we traverse LL to find C node, keep track of the previous node as well.

Delete Node in LL by Traversing

```
public static void delete() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter key to delete: ");
    int n = sc.nextInt();
   // Store head node
    node p = start, prev = null;
    while (p != null && p.info != n) {
        // If currNode does not hold key
        // continue to next node - traverse till n is found
        prev = p;
        p = p.next;
    }
 // If n is found, it should be at p (current node)
    // Therefore p shall not be null
    if (p != null && p.info==n) {
        // Since n is at p
        // Unlink p from linked list
        prev.next = p.next; //previous node's link points to next node
        // Display the message
        System.out.println(n + " found and deleted");
    }
}
```

Arrays vs Linked Lists

1. Array advantages:
   - Simple
   - Faster access to elements – constant access
2. Array disadvantages:
   - Fixed size
   - Difficult to insert elements at a given index – expensive shifting process
3. Linked list advantages:
   - Expansion and depletion easy
4. Linked list disadvantages:
   - Access time to individual elements is more
   - Storage of pointers take space

Insertion of a Node in LL

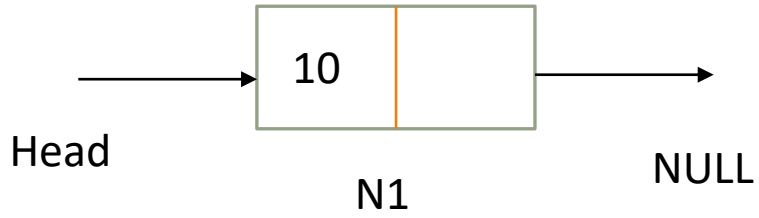
Insertion of Node in a LL

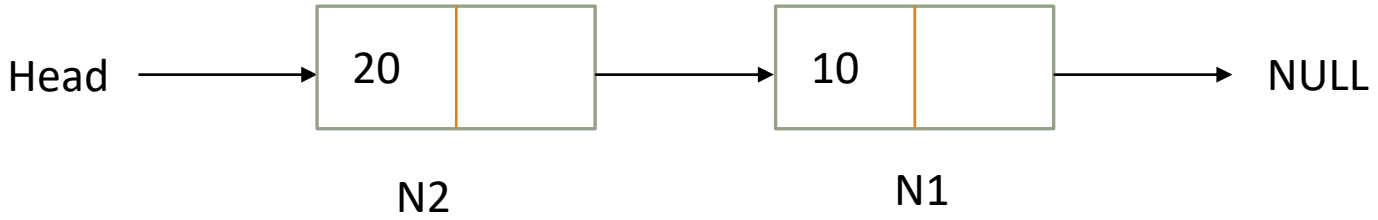
1. 3 cases might occur:
   a) Insertion at the beginning
   b) Insertion at the end
   c) Insertion at a random location
2. The reference/link/pointer modifications will be different in each case

Insertion at Beginning

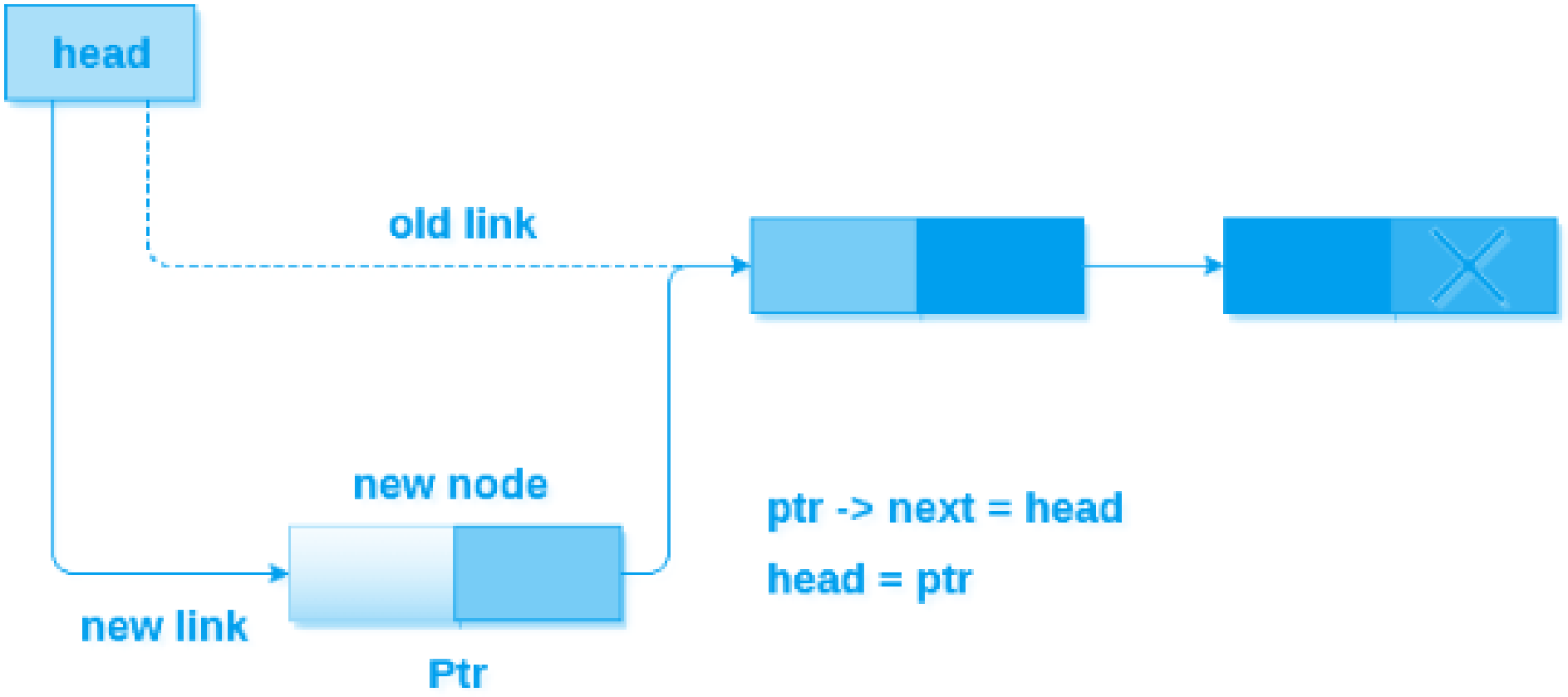
Insertion of Node at Beginning of LL

1. Make a new node using the given data.
2. If the linked list's head is null,
   1. set the new node as the linked list's head and return.
3. Else
   1. Set the new node's next pointer to the current head of the linked list.
   2. Set the linked list's head to point to the new node.
4. Return.





Insertion of Node at Beginning of LL



Insertion of Node at beginning of LL

1. How many next pointers are modified?
   - Modify only 1 next pointer
   - Update the next pointer of the new node to point to the current head
   - Update head pointer to point to the new node

Insertion of Node at Beginning of LL

```
function insertAtBeginning(data):
    newNode = Node(data)
    if head is null:
        head = newNode
        return
    newNode.next = head
    head = newNode
    return
```

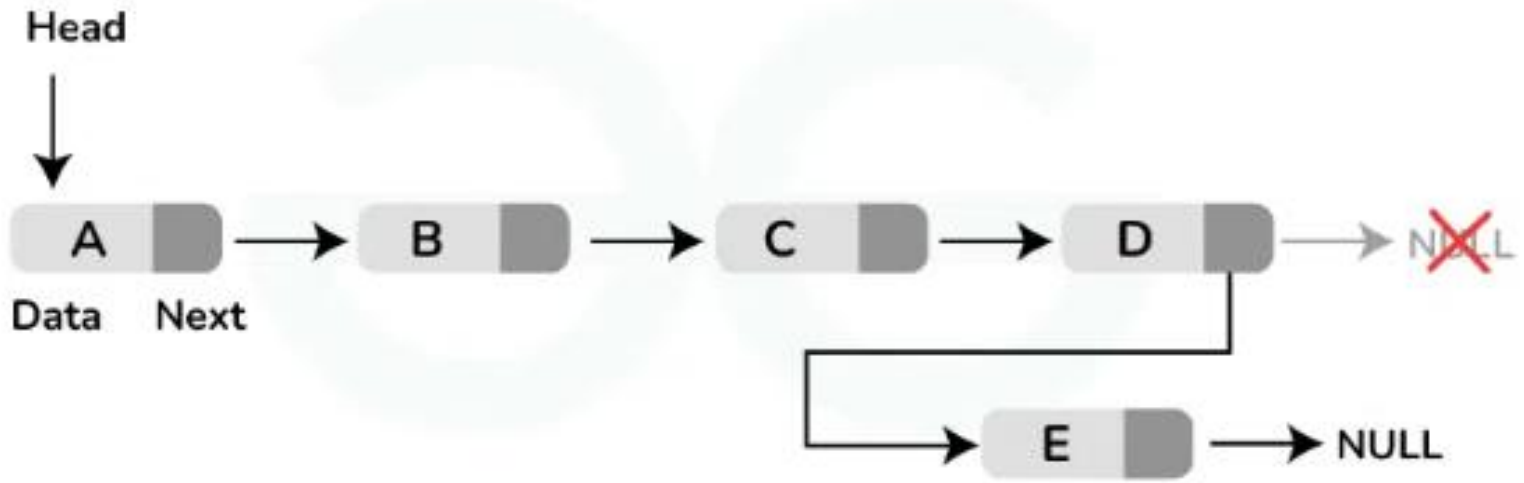
Insertion of Node at Beginning of LL

```
public static void insert_at_beg() {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter info: ");
        int n = sc.nextInt();
        node new_node = new node();
        new_node.info = n;
        new_node.next = start;
        start = new_node;
}
```

Insertion at End

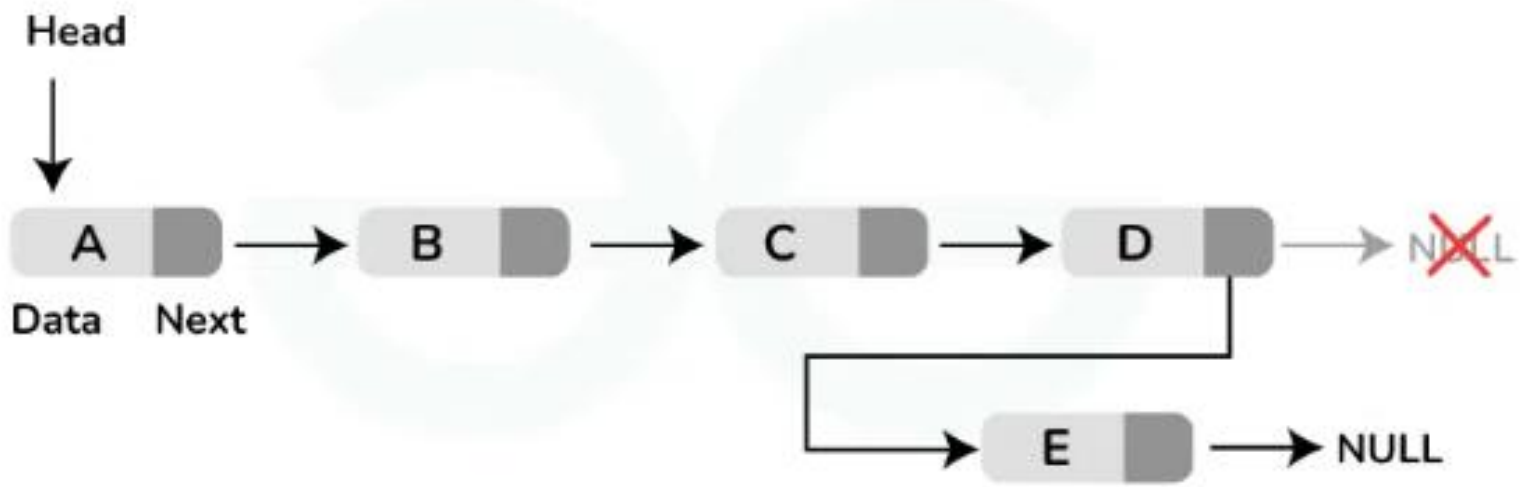
Insertion of Node at end of LL

1. Create the new node with given data
2. Go to the last node of the Linked List
3. Change the next pointer of last node from NULL to the address of new node
4. Make the next pointer of new node as NULL to show the end of Linked List



Insertion of Node at end of LL

1. How many next pointers modified?
   - Modify 2 next pointers – last node and new node
   - New node's next pointer points to NULL
   - Last node's next pointer points to new node



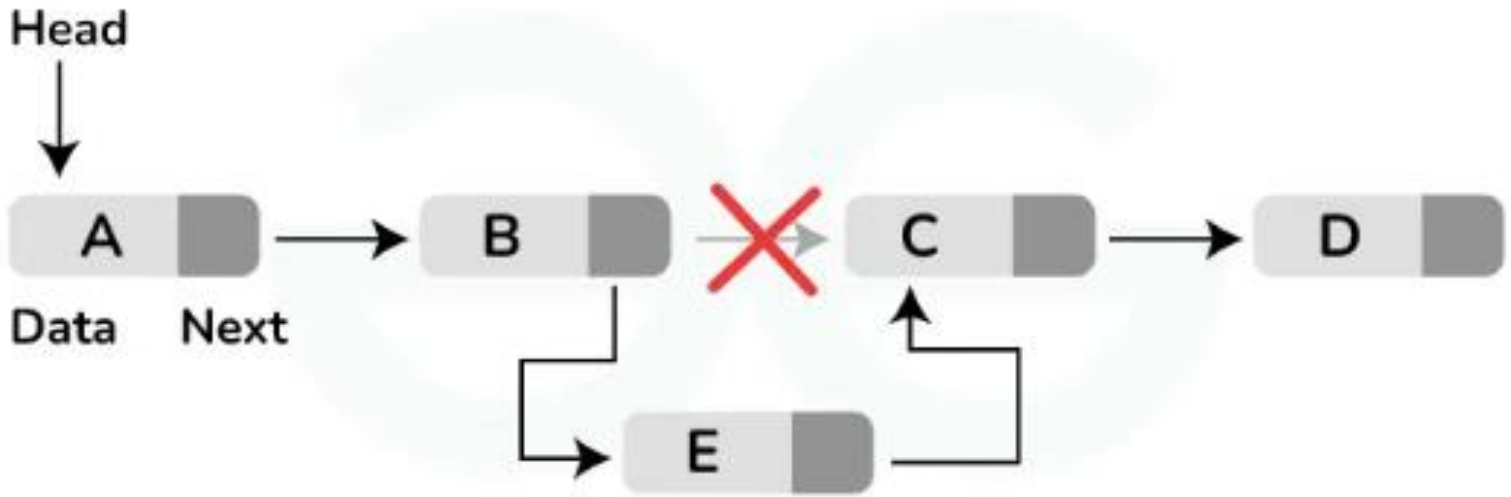
Insertion of Node at end of LL

```
public static void insert_at_end()  {
      Scanner sc=new Scanner(System.in);
      System.out.println("Enter info: ");
      int n = sc.nextInt();
      node p = start;
      node new_node = new node();
      new_node.info = n;
      //check if start is null
      if (p == null) {
            p = new_node;
            return;
            }
      //traverse till last node
      while (p.next != null) {
            p = p.next;
      }
      //p is the last node at this point.
      //insert new node after p
      p.next = new_node;
      new_node.next = null;
}
```

Insertion at Any Position

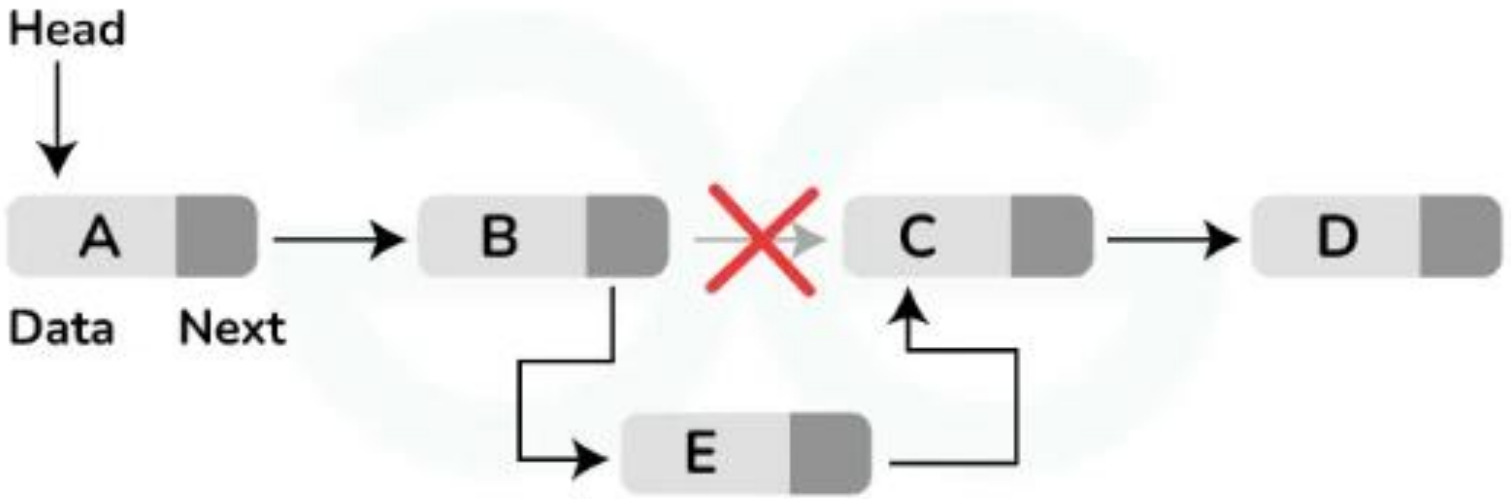
Insertion of Node at n^{th} position in LL

1. Traverse the Linked list till n-1 nodes. The $(n-1)^{th}$ node is the current node.
2. Create a new node with the given data
3. Update the next pointer of the new node to the next of current node.
4. Update the next pointer of current node to the new node.



Insertion of Node at end of LL

1. How many next pointers modified?
   - Modify 2 next pointers – new node's and current nodes



Insertion of Node at specific position of LL

```
public static void insert_at_n() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter position of new node: ");
    int n = sc.nextInt();
    System.out.println("Enter info: ");
    int data = sc.nextInt();
    node new_node = new node();
    new_node.info = data;
    int counter =1;
    if(n==1) { // same as insert_at_beg
        new_node.next = start;
        start = new_node;
    }
    else {//traverse till n-1 position
        node p = start;
        while(counter<n-1) {
            counter++;
            p = p.next;
            } //at this point p is the (n-1)th node
        new_node.next = p.next;
        p.next = new_node;
    }
}
```

Deletion of a Node in LL

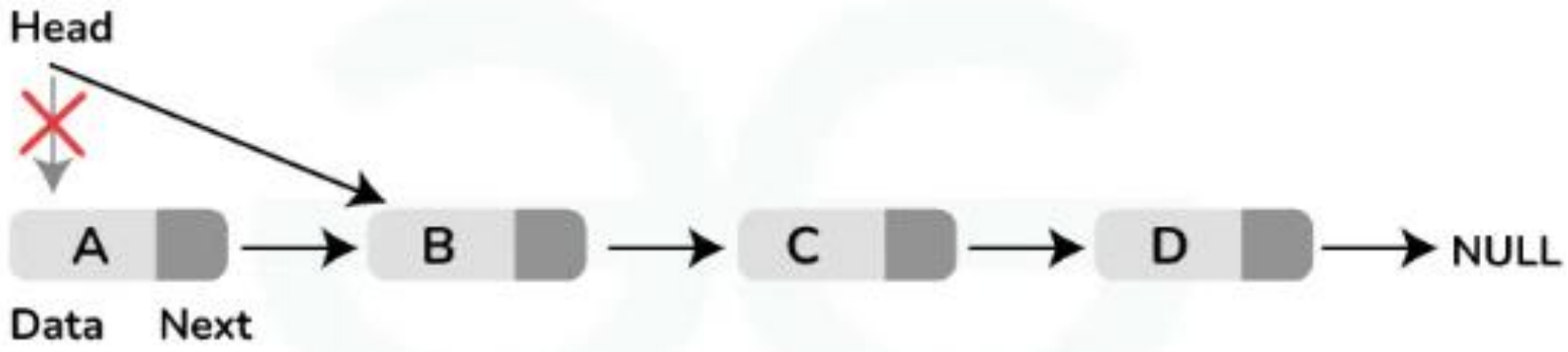
Deletion of Node in a LL

1. 4 cases might occur:
   a) Deletion at the beginning
   b) Deletion at the end
   c) Deletion at a random location
   d) Deletion by value
2. The reference/link/pointer modifications will be different in each case

Deletion at Beginning

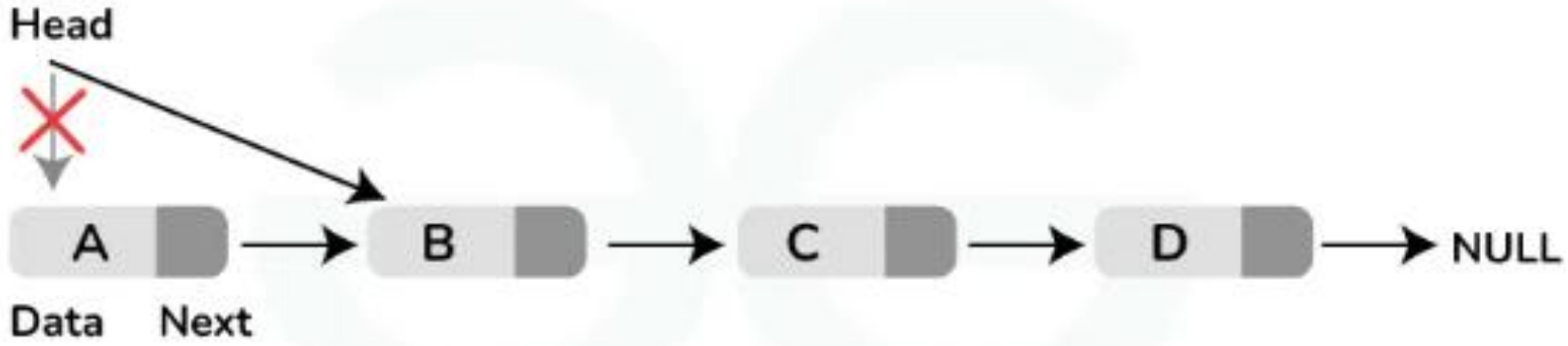
Deletion of Node from Beginning in LL

1. Access LL by the start/head node.
2. Point start/head to the next node i.e. second node



Deletion of Node from Beginning in LL

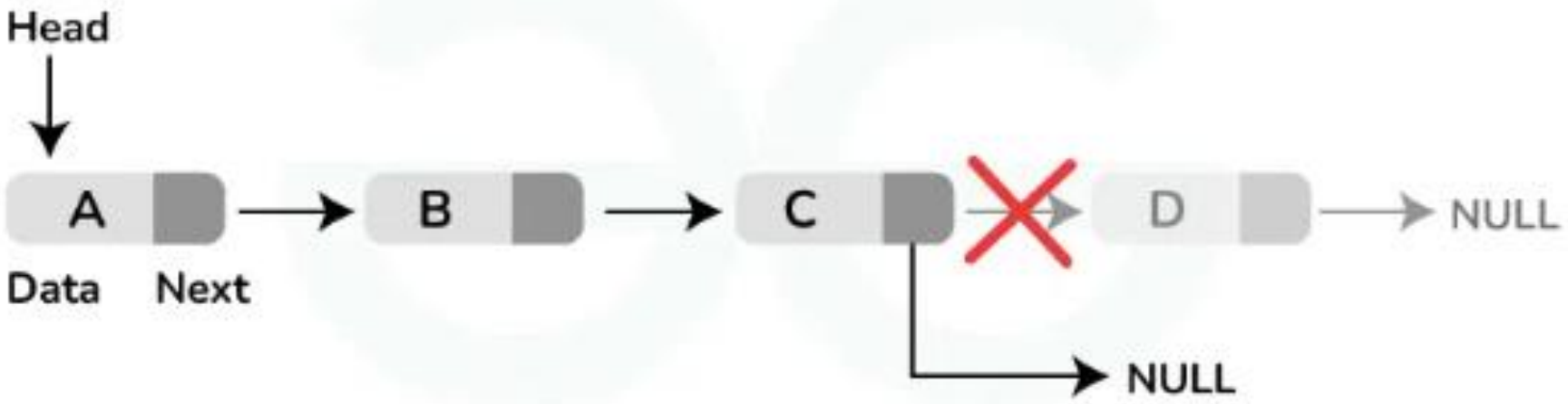
```
public static void delete_at_beg() {
      node p = start;
      start = p.next;
}
```



Deletion at End

Deletion of Node from End in LL

1. Access LL by the start/head node.
2. Traverse to the second last node
3. Update second last node's next to null



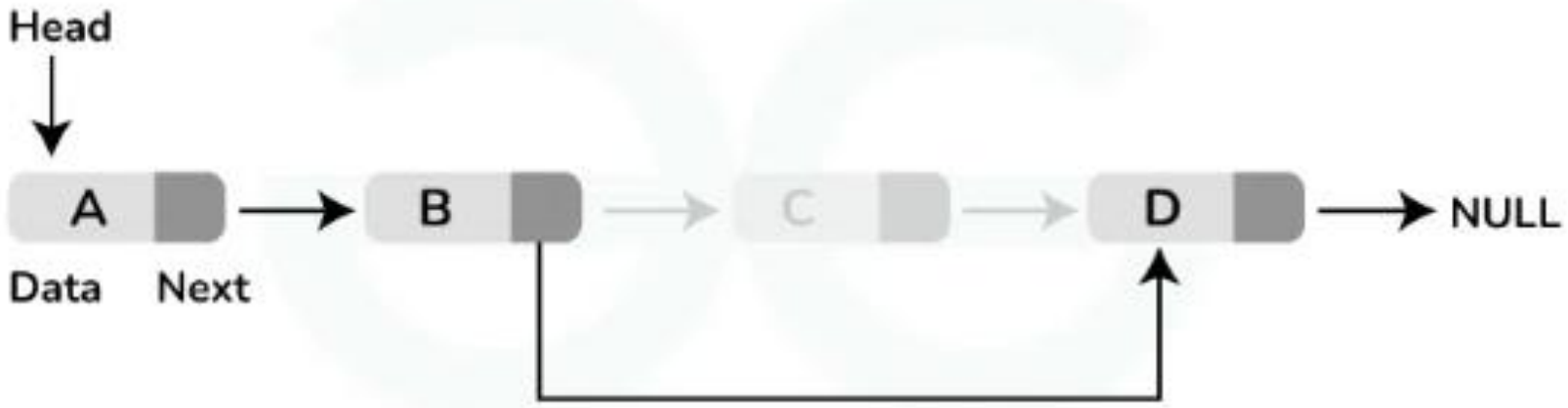
Deletion of Node from End in LL

```
public static void delete_at_end() {
    node p = start;
    if (p == null)//no nodes in LL
        return;
    if (p.next == null) { //only one node in LL
        start = null;
     }
    // Find the second last node
    node second_last = p;
    while (second_last.next.next != null)
        second_last = second_last.next;
    // Change next of second last
    second_last.next = null;
}
```

Deletion at Any Position

Deletion of Node at n^{th} position in LL

1. Access LL by the start/head node.
2. Traverse to the specific position provided – n^{th} node
3. Update $(n-1)^{th}$ node's next to n.next

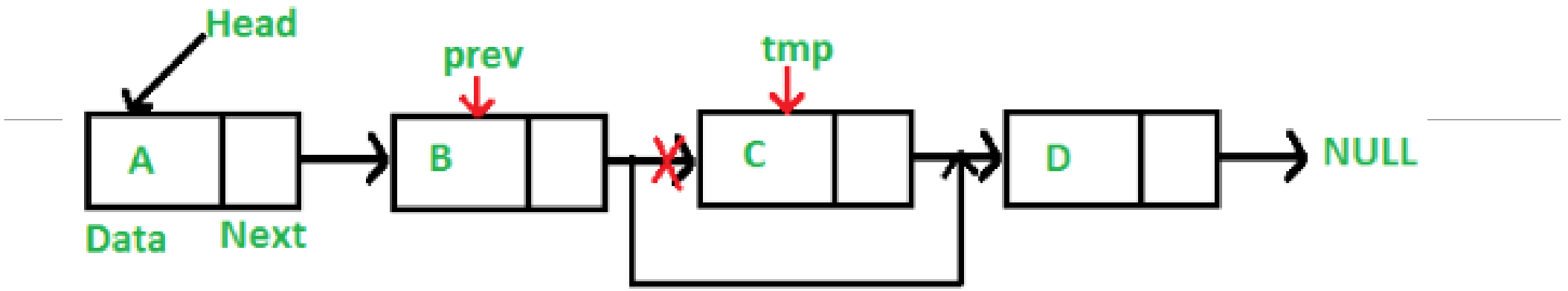


Deletion of Node at n^{th} position in LL

```
public static void delete_at_n() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter position of node to be deleted: ");
    int n = sc.nextInt(); //assume its not first or last node, add checks if needed
    node p = start;
    for(int i=1;p != null && i < n - 1;i++) {
        p = p.next;
    }
    //p is the (n-1)th node
    // Node p->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    node next = p.next.next;
    p.next = next; // Unlink the deleted node from list

}
```

Deletion by Value

Deletion of a Node in LL



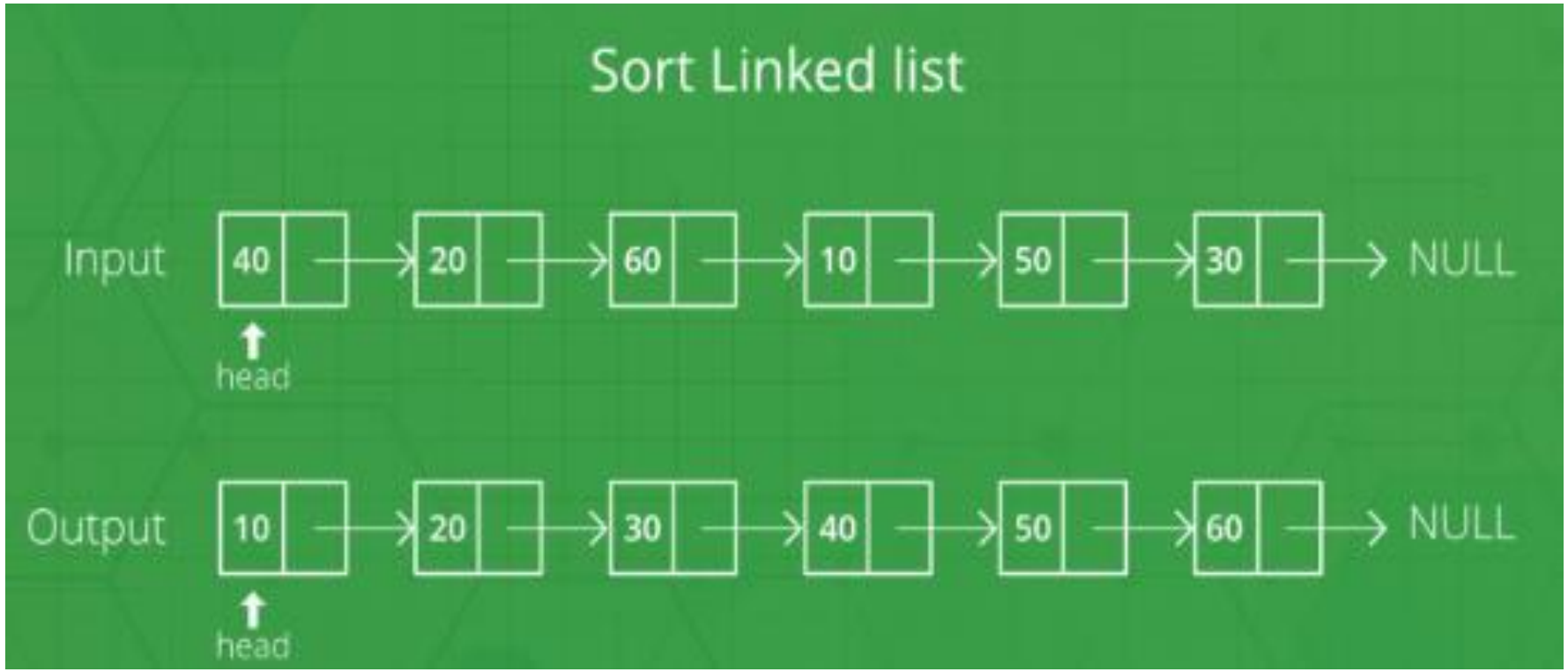
1. One info value is provided and you need to delete that node which contains that info.
2. If C (in diagram) is deleted, it means that, we need to update the content of its previous node's pointer.
3. So, after deletion of C, node B should directly point to D.
4. How to do this? – when we traverse LL to find C node, keep track of the previous node as well.

Delete Node in LL by Traversing

```
public static void delete() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter key to delete: ");
    int n = sc.nextInt();
   // Store head node
    node p = start, prev = null;
    while (p != null && p.info != n) {
        // If currNode does not hold key
        // continue to next node - traverse till n is found
        prev = p;
        p = p.next;
    }
 // If n is found, it should be at p (current node)
    // Therefore p shall not be null
    if (p != null && p.info==n) {
        // Since n is at p
        // Unlink p from linked list
        prev.next = p.next; //previous node's link points to next node
        // Display the message
        System.out.println(n + " found and deleted");
    }
}
```

Sort the LL

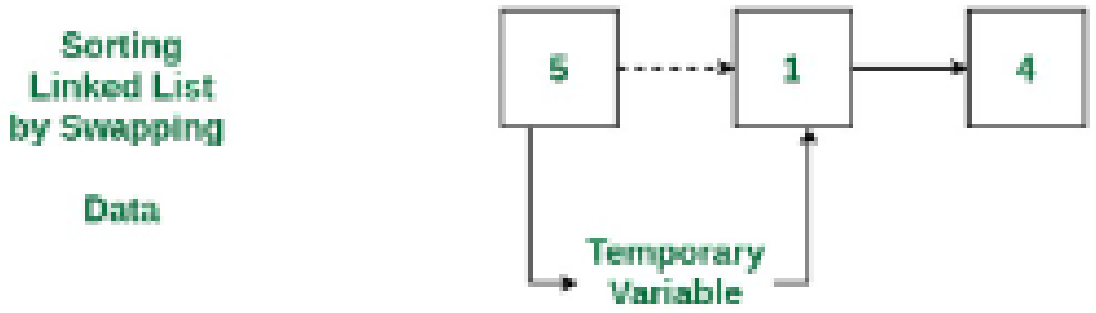
Sorting of Linked List by Bubble Sort



1. Bubble sort takes place by swapping elements not in correct order
2. Types:
   a) **Swap values**
   b) **Swap nodes**

Sorting LL: Swap Values

1. Access LL by the start/head node.
2. Traverse each consecutive pair of nodes
3. Compare the N1.info with N2.info
4. Swap if necessary.
5. Go ahead in the LL
6. Passes will continue till all values in correct order.



Sorting LL: Swap Values

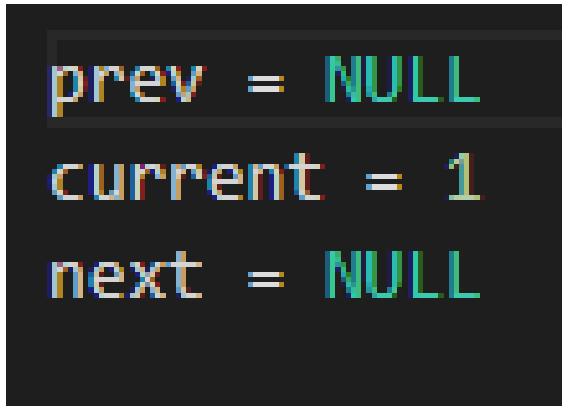
```
public static void bubbleSort() {
    boolean swapped;
    node current, p = start;
    if (p == null)
        return;
    do {
        swapped = false;
        current = p;

        while (current.next != null) {
            if (current.info > current.next.info) {//swapping
                int tmp = current.next.info;
                current.next.info = current.info;
                current.info = tmp;
                swapped = true;
            }
            current = current.next; // traverse to next node
        }
    } while (swapped);
}
```

Reverse the LL

Reversing Linked List

Reversing Linked List

1. Initialize 3 pointers for this job, prev, current, next



Reversing Linked List

1. Start from the first node (head node) of the LL.
2. Reverse the first node in the list

```
prev = NULL
current = 1 -> 2 -> 3 -> 4 -> NULL
next = 2 -> 3 -> 4 -> NULL

1    -> NULL
2    -> 3 -> 4 -> NULL
```

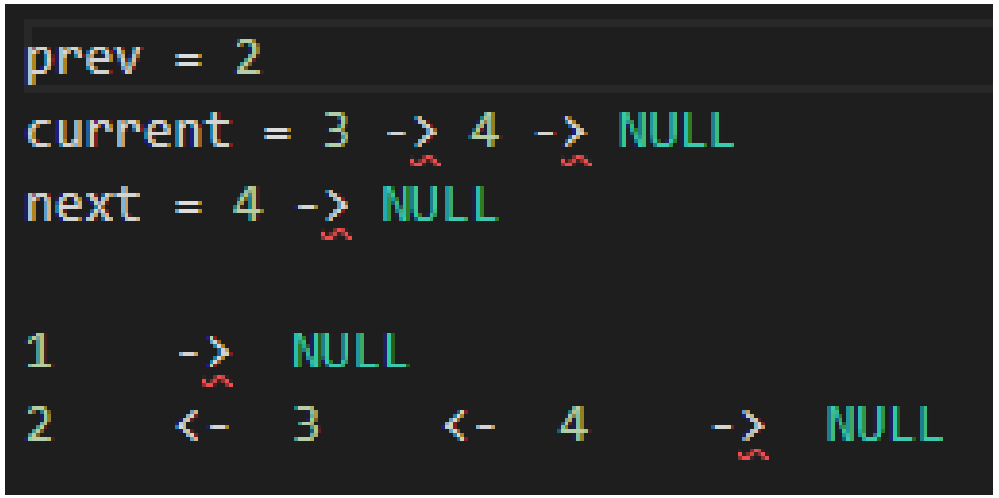
Reversing Linked List

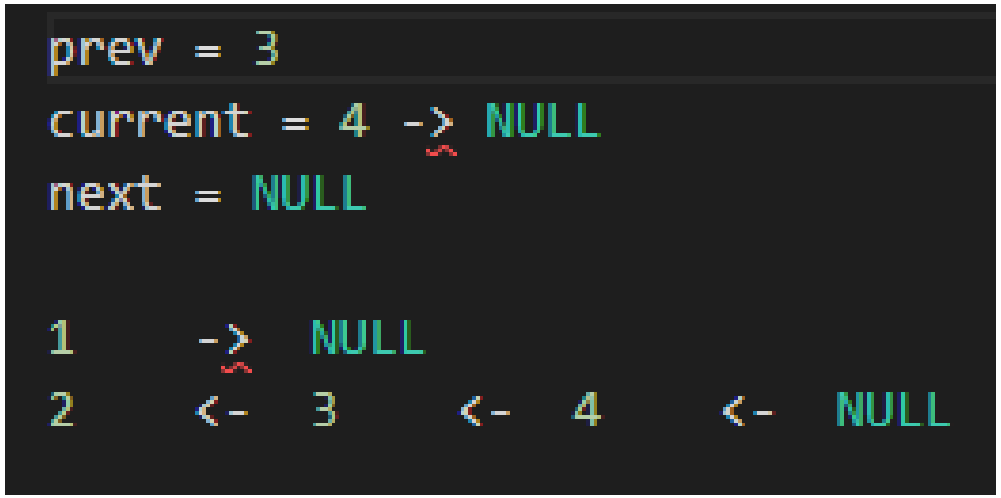
1. Move the prev, current, and next pointers to the next node:

```
prev = 1
current = 2 -> 3 -> 4 -> NULL
next = 3 -> 4 -> NULL
```

Reversing Linked List

1. Repeat this process for each node in the list until we reach the end





2. At this point, the list has been fully reversed.
3. We update the head pointer to the last node, which is 4.

Reversing Linked List

```
public static void reverse() {
    node prev = null;
    node current = start;
    node next = null;
    while (current != null) {
        next = current.next;//traversing LL
        current.next = prev;//reversing link of current node
        prev = current;//updating prev
        current = next;//updating current node under consideration
    }
    start = prev; //you have changed the head of the LL
}
```

Singly Linked List Operations

1. Creation of one node
2. Insertion of node
   a) At beginning
   b) At end
   c) At nth position
3. Deletion of node
   a) At beginning
   b) At end
   c) At nth position
   d) By value

1. Traversal and Display of node contents
2. Counting of number of nodes
3. Search a value
4. Sort a SLL
5. Reverse a SLL

Doubly Linked Lists

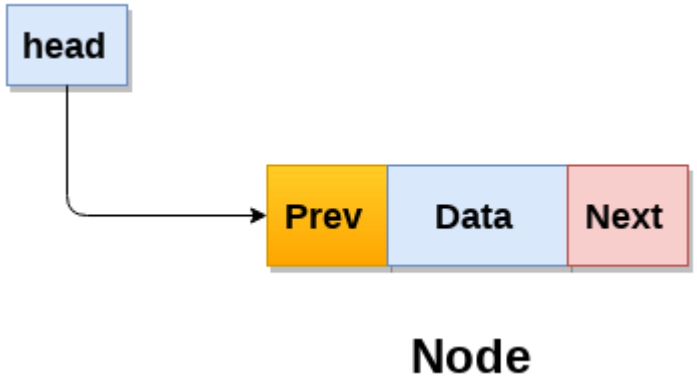
Doubly Linked List Operations

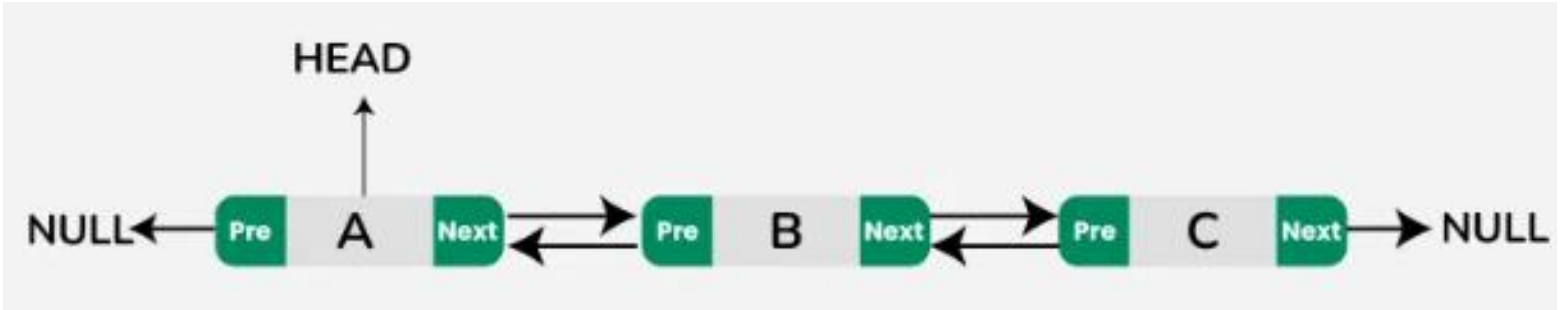
1. Creation of one node
2. Insertion of node
   a) At beginning
   b) At end
   c) At nth position
3. Deletion of node
   a) At beginning
   b) At end
   c) At nth position
   d) By value

1. Traversal and Display of node contents
2. Counting of number of nodes
3. Search a value

Doubly Linked Lists

1. A doubly linked list (DLL) is a special type of linked list in which each node contains a pointer to the previous node as well as the next node of the linked list.





Doubly Linked Lists

1. Representation of a DLL node in Java with one data member

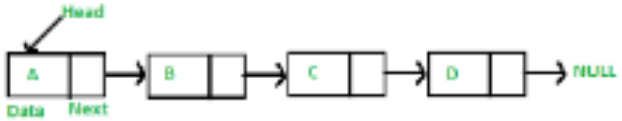
```
class DLL_Node{
    int info;//data member
    DLL_Node prev;//reference to the previous node
    DLL_Node next;//reference to the next node
}
```

DLL: Multiple Data Members

1. Representation of a DLL node in Java with three data members

```
class DLL_Node{
    int info1;//data member
    double info2;
    String info3;
    DLL_Node prev;//reference to the previous node
    DLL_Node next;//reference to the next node
}
```

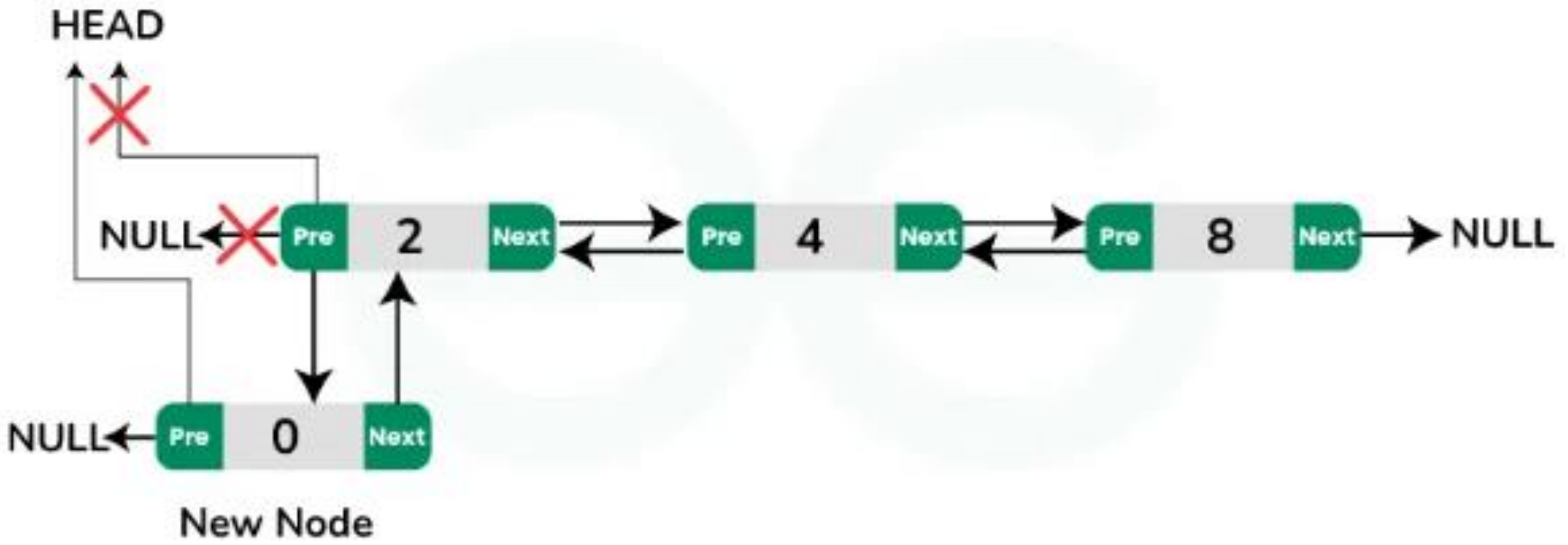
DLL vs SLL:

Singly linked list (SLL)	Doubly linked list (DLL)
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.
	
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.

DLL Operations

1. DLL Creation
2. Insertion of node
   a) At beg
   b) At end
   c) At n^{th} position
3. Deletion of node
   a) At beg
   b) At end
   c) At n^{th} position
4. Search for a value – same as SLL
5. Display – same as SLL
6. Count – same as SLL

DLL: Insertion at Beginning



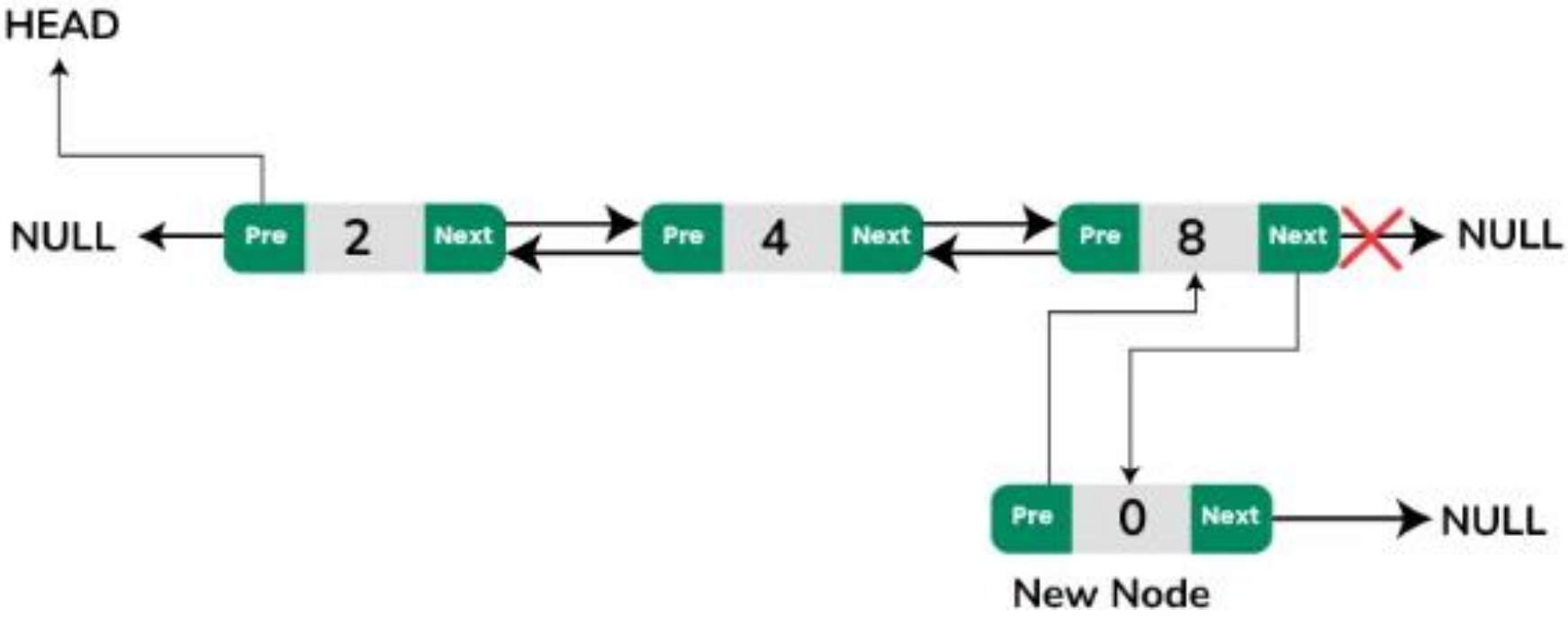
DLL: Insertion at Beginning

1. Create the new node
2. Set the previous pointer of the new node to null.
3. If the list was empty:
   a) Set the next pointer of the new_node to null.
   b) Update the head pointer to point to the new_node.
4. If the list is not empty:
   a) Set the next pointer of the new_node to the current head.
   b) Update the previous pointer of the current head to point to the new_node.
   c) Update the head pointer to point to the new_node.

DLL: Insertion at Beginning

```
public static void insert_at_beg() {
        DLL_Node new_node = new DLL_Node();//new node created
        Scanner sc=new Scanner(System.in);//taking care of data part
        System.out.println("Enter int info1");
        new_node.info1=sc.nextInt();
        System.out.println("Enter double info2");
        new_node.info2=sc.nextDouble();
        System.out.println("Enter string info3");
        new_node.info3=sc.next();
        //taking care of pointer
        new_node.next = start;
        start = new_node;
        new_node.prev = null;
}
```

DLL: Insertion at End



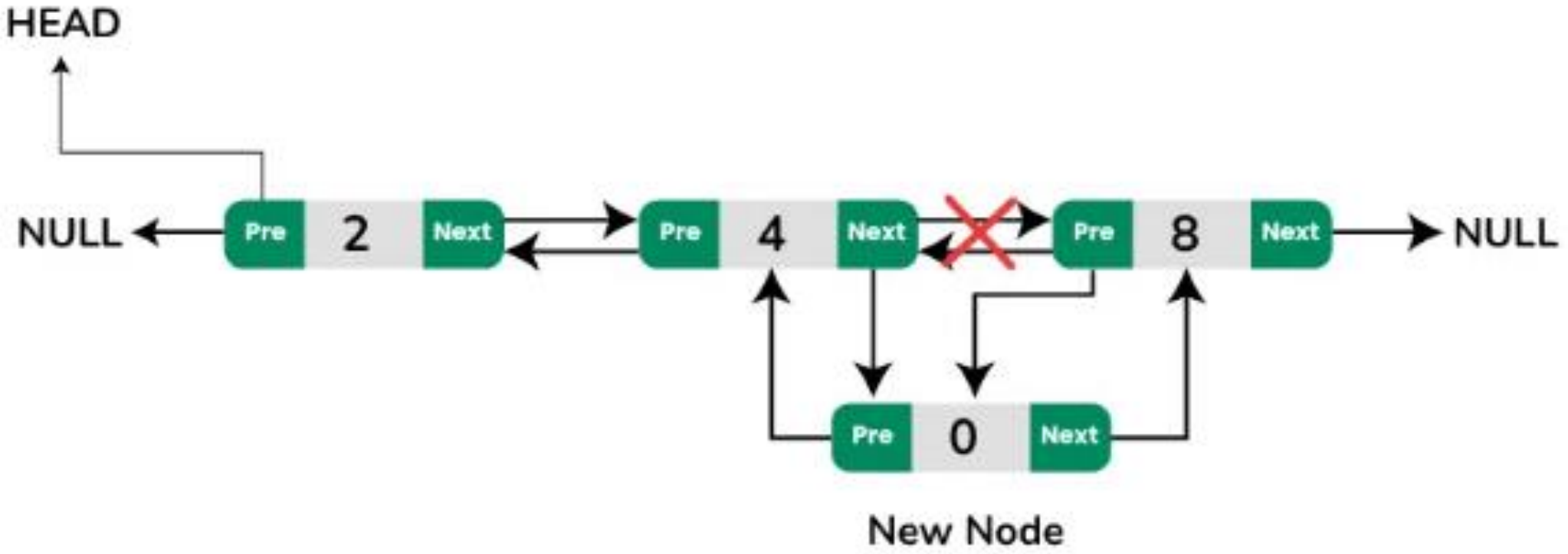
DLL: Insertion at End

1. Create the new node
2. Make the next pointer of new_node as null.
3. If the list was empty, make new_node as the head.
4. Else, traverse to the end of the DLL.
   a) Update the next pointer of last node to point to new_node.
   b) Update the previous pointer of new_node to the last node of the list.

DLL: Insertion at End

```
public static void insert_at_end() {
      DLL_Node new_node = new DLL_Node();//new node created
      Scanner sc=new Scanner(System.in);//taking care of data part
      System.out.println("Enter int info1");
      new_node.info1=sc.nextInt();
      System.out.println("Enter double info2");
      new_node.info2=sc.nextDouble();
      System.out.println("Enter string info3");
      new_node.info3=sc.next();
      //taking care of pointer
      DLL_Node p = start;
     //traverse till last node
     while (p.next != null) {
         p = p.next;
     } //p is the last node at this point.
     new_node.prev = p;
     p.next = new_node;
     new_node.next = null;
}
```

DLL: Insertion at n^{th} position



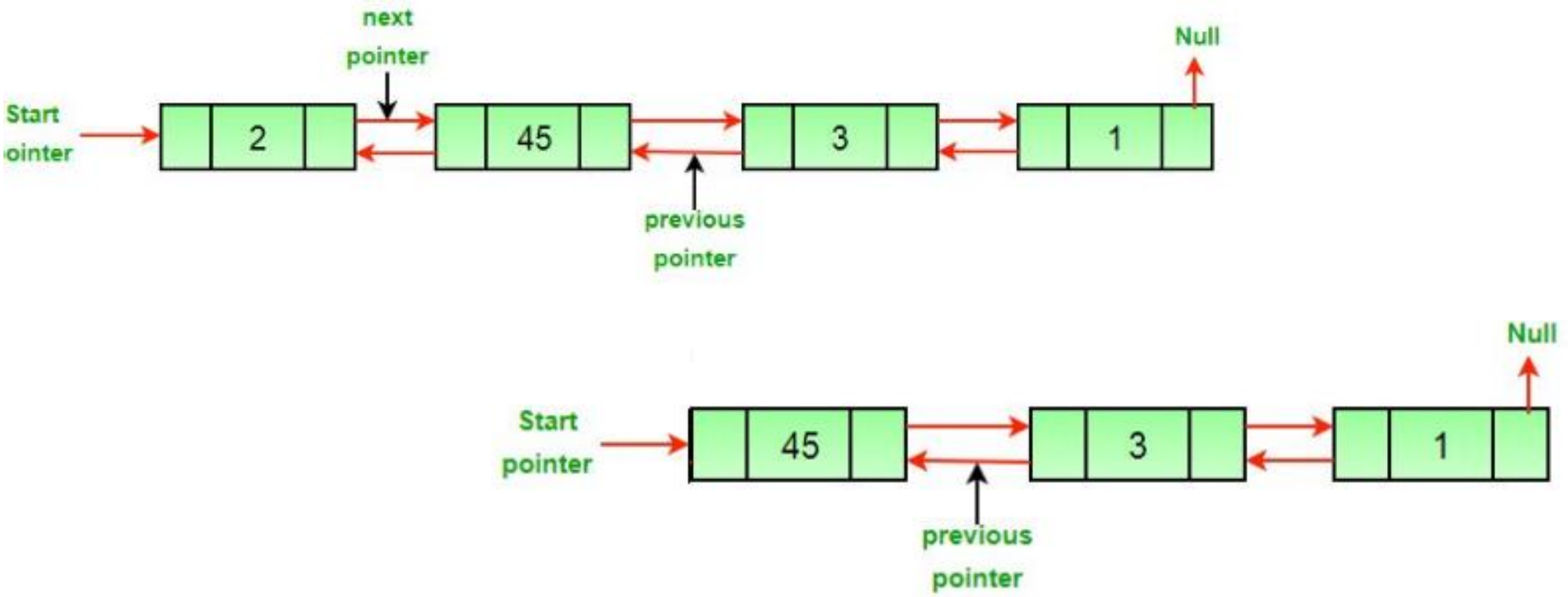
DLL: Insertion at n^{th} position

1. Create the new_node
2. Point the next of new_node to the next of prev_node.
3. Point the next of prev_node to new_node.
4. Point the previous of new_node to prev_node.
5. Point the previous of next of new_node to new_node..
6. Total 4 pointers are updated

DLL: Insertion at n^{th} position

```
public static void insert_at_n() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter position of new node: ");
    int n = sc.nextInt();
    DLL_Node new_node = new DLL_Node();//new node created
     System.out.println("Enter int info1");
     new_node.info1=sc.nextInt();
     System.out.println("Enter double info2");
     new_node.info2=sc.nextDouble();
     System.out.println("Enter string info3");
     new_node.info3=sc.next();
     int counter =1; //taking care of pointer from here
     DLL_Node p = start;
     while(counter<n-1) { //traverse till n-1 position
            counter++;
            p = p.next;
        } //at this point p is the (n-1)th node
    new_node.prev = p; //insert new node after p
    new_node.next = p.next;
    p.next = new_node;
}
```

DLL: Deletion at Beginning



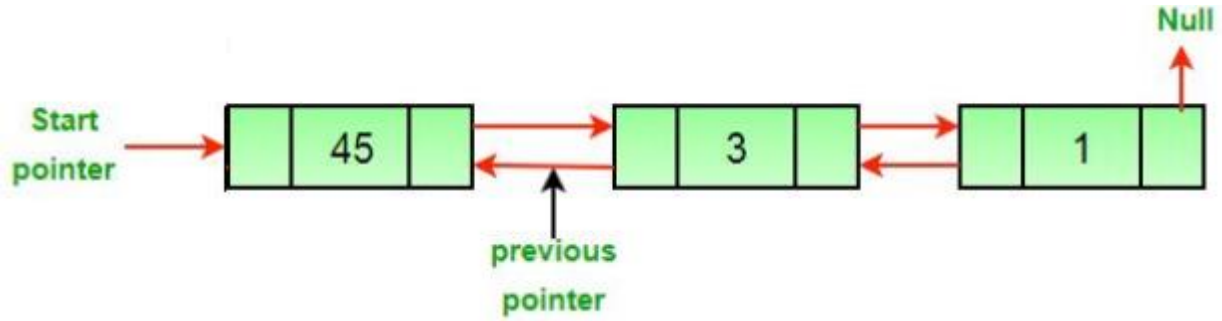
DLL: Deletion at Beginning

```
public static void delete_at_beg() {
    DLL_Node p = start;//p points to first node
    DLL_Node q = p.next;//q points to second node
    start = q; //q becomes first node
    //but q.prev is still pointing to p
    q.prev = null; //update it to null
    //all connection of p with DLL lost
}
```

DLL: Deletion at Beginning

1. Update the start pointer of the DLL to point to second node
2. Update the second.prev pointer to NULL

DLL: Deletion at n^{th} position



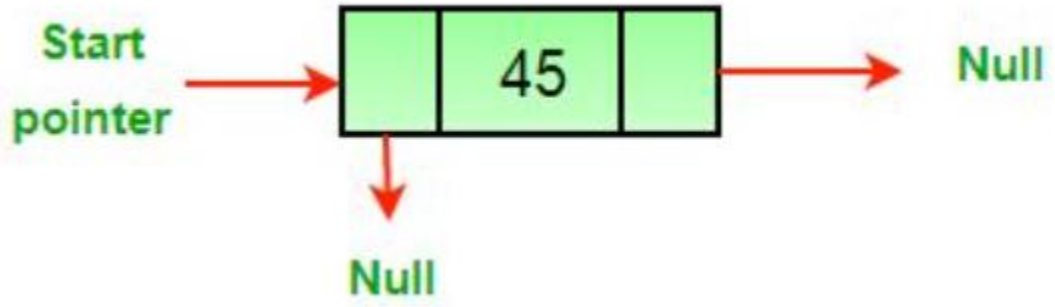
DLL: Deletion at n^{th} position

```
public static void delete_at_n() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter position of node to be deleted: ");
    int n = sc.nextInt(); //assume its not first or last node, add checks if needed
    DLL_Node p = start;
    for(int i=1;p != null && i < n - 1;i++) {
        p = p.next;
    }
    //p is the (n-1)th node
    // p's next node is the node to be deleted
    // Store address of p's next's next node.
    DLL_Node next = p.next.next;
    p.next = next; // Unlink the node to be deleted by creating link between p and p's next's next
    next.prev = p; //Link next.prev with p

}
```

DLL: Deletion at n^{th} position

1. Take user input for n
2. Traverse till (n-1) node
3. Update (n-1).next to n.next
4. Update (n+1).prev to n.prev

DLL: Deletion at End



DLL: Deletion at End

```
public static void delete_at_end() {
    DLL_Node p = start;
    if (p == null)//no nodes in LL
        return;
    if (p.next == null) { //only one node in LL
        start = null;
     }
    // Find the second last node
    DLL_Node second_last = p;
    while (second_last.next.next != null)
        second_last = second_last.next;
    // Change next of second last
    second_last.next = null;
}
```

DLL: Deletion at End

1. Traverse to end of DLL
2. Update second last node’s next to null

DLL: Linear Search

```
public static void linear_search() {
    Scanner sc=new Scanner(System.in);
    System.out.println("Enter int key to search: ");
    int n = sc.nextInt();
    DLL_Node p = start; // Store head node
     int counter=0;
     boolean found=false; //boolean variable to indicate if key is found or not
     while (p != null) {
         if(p.infol==n) {
             counter++;
             System.out.println(n + " found at node number "+ counter + " of LL");
             found = true; //key found
             break;
         }
         else {
         // If currNode does not hold key
         // continue to next node - traverse till n is found
         counter++;
         p = p.next;
         }
     }
     if(!found)
         System.out.println("Key NOT FOUND");
}
```

Thank You