Algorithms and Data Structures

-Soumya

Linked list

Linked list is a linear data structure that includes a series of connected nodes.

- Linked list can be defined as the nodes that are randomly stored in the memory.
- A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
- The last node of the list contains a pointer to the null.
- In a linked list, every link contains a connection to another link.
- Linked Lists are used to create trees and graphs.

Linked List: Why?

Linked list is a data structure that overcomes the limitations of arrays.

some of the limitations of arrays -

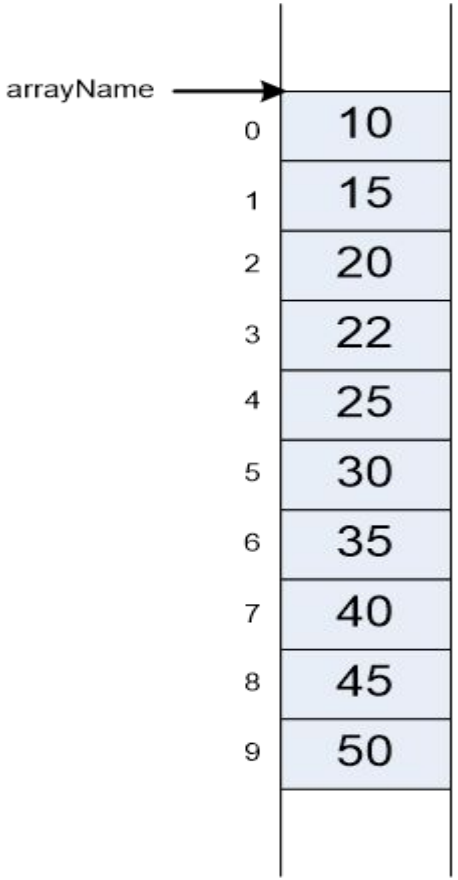
– The size of the array must be known in advance before using it in the program.

– Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

– All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Arrays versus Linked Lists

Array: Contagious Storage



Arrays versus Linked Lists

- In arrays, elements are stored in a contiguous memory locations
- Arrays are static data structure unless we use dynamic memory allocation.
- Arrays are suitable for
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.

Arrays versus Linked Lists

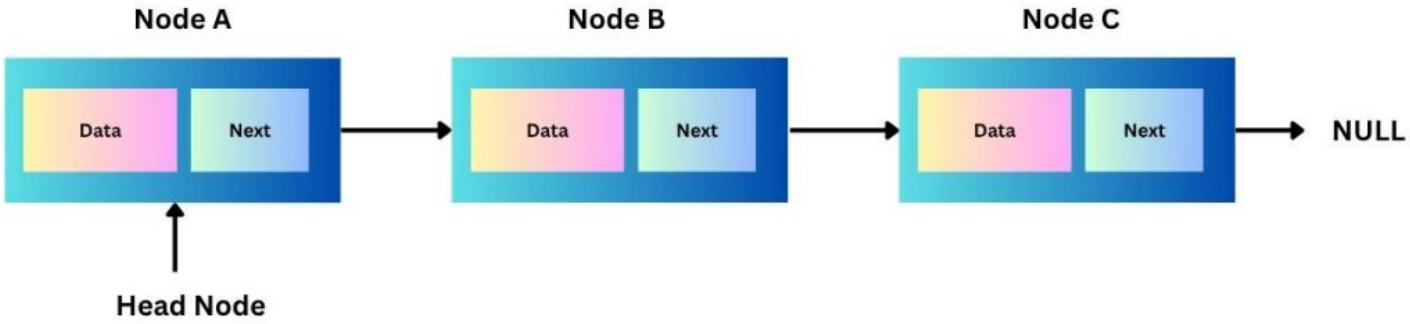
- In Linked lists ,adjacency between any two elements are maintained by means of links or pointers
- It is essentially a dynamic data structure
- Linked lists are suitable for
- Inserting an element at any position.
- Deleting an element from anywhere.
- Applications where sequential access is required.
- In situations, where the number of elements cannot be predicted beforehand.

Linked List: Representation

- A node is a fundamental building block of a linked list. It mainly contains two components:

1. **Data**: The actual value or payload stored in the node.
2. **Next Pointer**: A reference or pointer to the next node in the sequence.

- The first node is called the head. If the linked list is empty, then the value of the head is NULL.



Linked List: Representation

- The tail is the last node in the linked list.

Linked list Blocks

- ➔ In case of railway we have peoples seating arrangement inside the coaches is called as data part of linked list while connection between two buggies is address field of linked list.
- ➔ Like linked list, trains also have last coach which is not further connected to any of the buggie.
- ➔ Engine can be called as first node of linked list.

Linked List: Advantages

Dynamic data structure - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.

• **Insertion and deletion** - Unlike arrays, insertion, and deletion in linked list is easier.

• To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.

Linked List: Advantages

- **Memory efficient** - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- **Implementation** - We can implement both stacks and queues using linked list.

Linked List: Disadvantages

Memory Usage: Each node in a linked list requires extra memory for the next pointer, which can be a disadvantage for large lists.

Traversal: Elements are stored in non-contiguous memory locations. Therefore, traversal can be slower than that of arrays.

Random Access: Direct/random access is not allowed. One must traverse the list from the head node to find the desired element.

Linked List Implementation

```
class Node{
  int data;   // the data part
  Node next; //the reference to the next node
  }
```

Linked List:Operations

Basic Operations on Linked Lists

1. **Insertion**: Adding a new node to the linked list.
2. **Deletion**: Removing an existing node from the linked list.
3. **Traversal**: Accessing each node in the linked list in a sequential manner.
4. **Searching**: Finding a node with a specific value.

Types of Linked List

- **Following are the various types of linked list.**

1. **Singly Linked List**: Each node points to the next node.

2. **Doubly Linked List**: Each node points to both the next and the previous nodes.

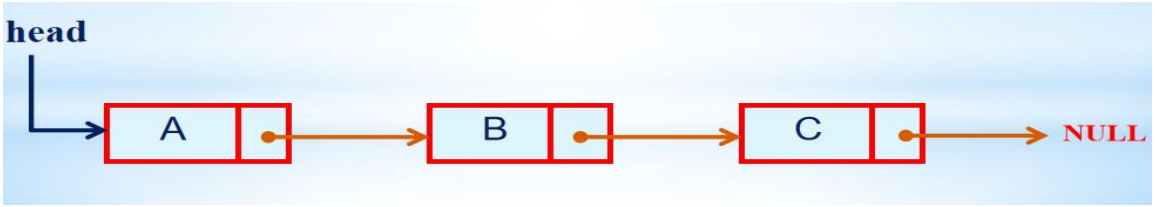
3. **Circular Linked List**: The last node points back to the first node.

Singly Linked List

Each node contains data and a single reference to the next node. The last node's next reference is null.

Characteristics:

- Simple structure.
- Efficient insertion and deletion at the beginning.
- Traversal is only forward.
- The tail's next pointer is always null in a singly linked list, signifying the end of the list.



LinkedList/Singly Linked List Implementation

```
class Node
{
 int data; // Value stored in the node
Node next; // Reference to the next node in the linked list
// Constructor for initializing a Node with a given data value
 public Node(int data) {
this.data = data;
this.next = null; // Initially, the node doesn't point to any other node
}
}
```

LinkedList/Singly Linked List Implementation

```
class Node {

int data; // Value stored in the node

Node next; // Reference to the next node in the linked list

// Constructor for initializing a Node with a given data value

public Node(int data) {

this.data = data;

this.next = null; // Initially, the node doesn't point to any other node

}

}
```

LinkedList/Singly Linked List Implementation

```
class LinkedList {

Node head; // The head node of the linked list

// Method to add a new node to the end of the linked list

public void add(int data) {

Node newNode = new Node(data); // Create a new node with the given data

}

}
```

Operations:Insertion

- There are three different ways in which we can insert a node in a linked list -:

1. Insertion at beginning
2. Insertion at end
3. Insertion at nth position

Insertion at the Beginning:

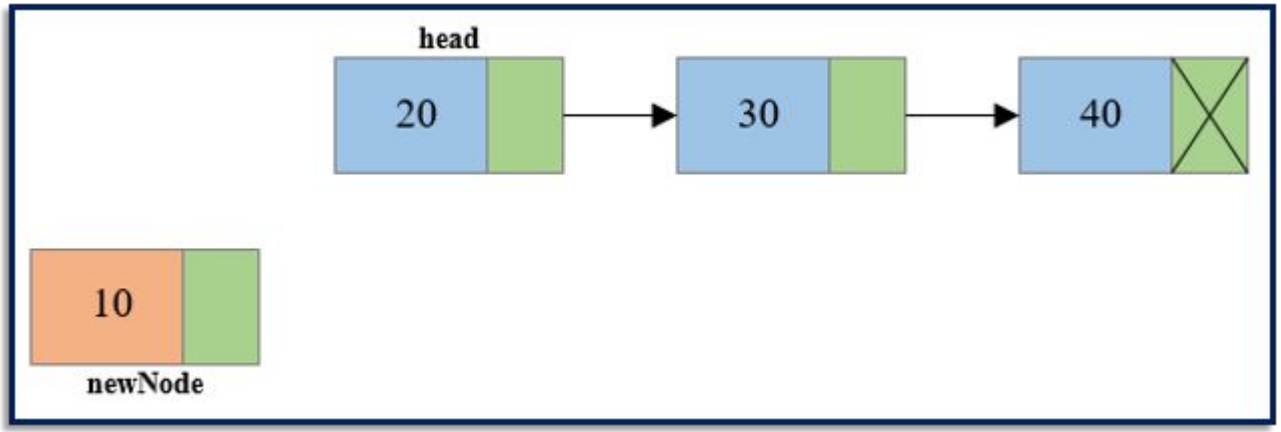
Algorithm:

1. Declare a head pointer and make it as NULL.
2. Create a new node with the given data.
3. Make the new node points to the head node.
4. Finally, make the new node as the head node.

Insertion at Beginning

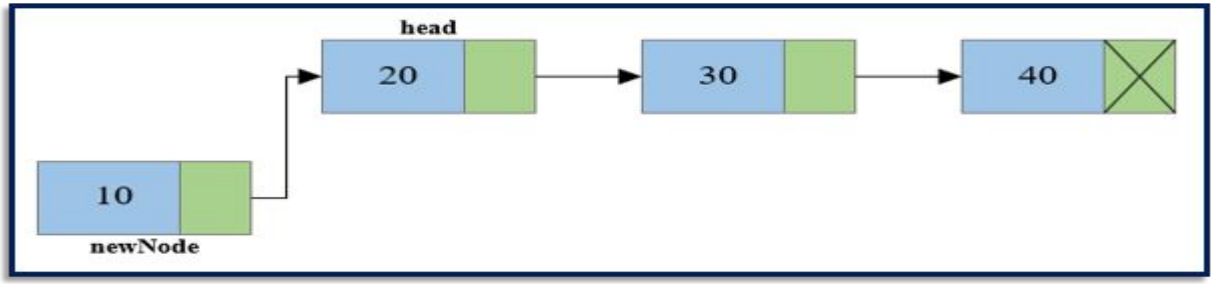
Steps to insert node at the beginning of singly linked list

Step 1: Create a new node.

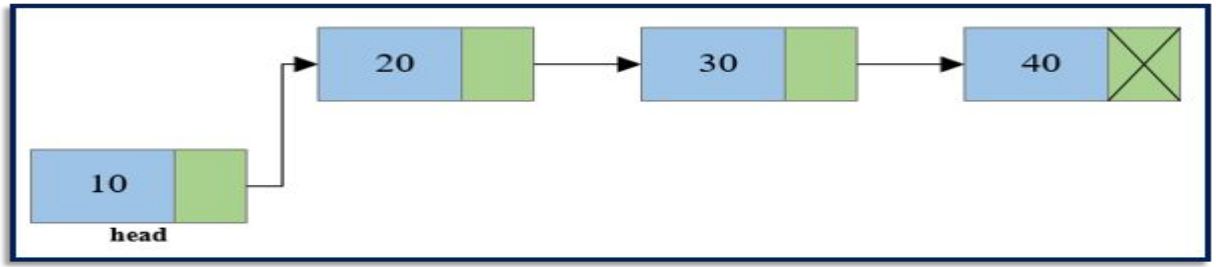


Insertion at Beginning

Step 2: Link the newly created node with the head node, i.e. the newNode will now point to head node.



Step 3: Make the new node as the head node, i.e. now head node will point to newNode.



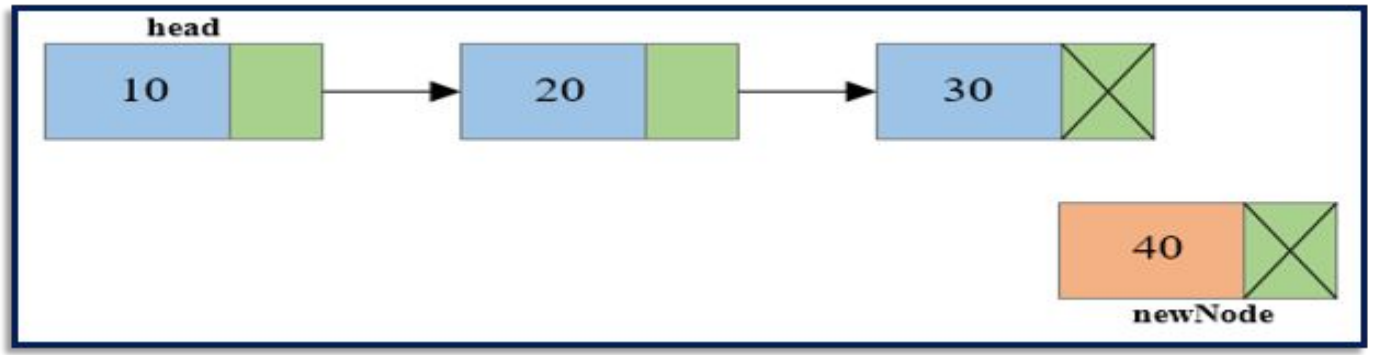
Insertion at Beginning

```
public void insertAtBeginning(int data) {

Node newNode = new Node(data);

newNode.next = head;

head = newNode;

}
```

Insertion at End

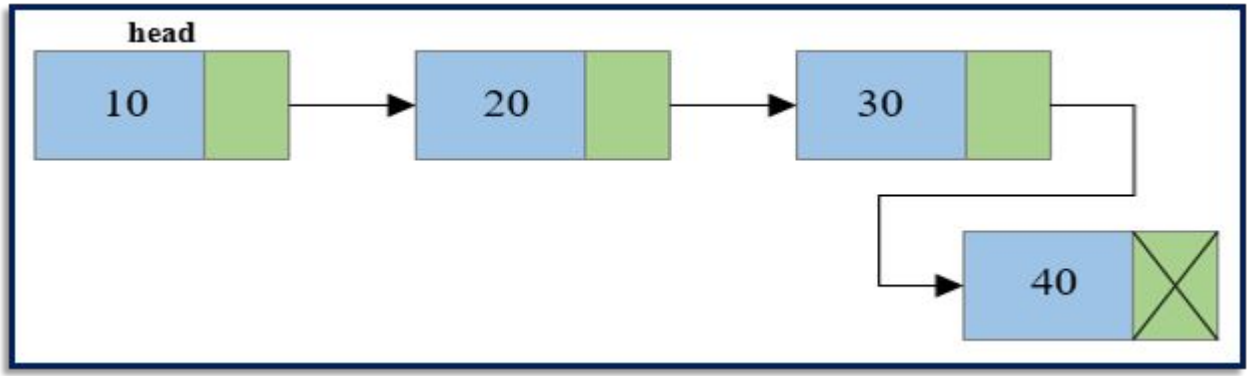
Steps to insert node at the end of Singly linked list

Step 1: Create a new node and make sure that the address part of the new node points to NULL. i.e. newNode->next=NULL



Insertion at End

Step 2: Traverse to the last node of the linked list and connect the last node of the list with the new node, i.e. last node will now point to new node. (lastNode->next = newNode).



Insertion at the End

1. Declare head and make it as NULL.

2. Create a new node with the given data. And make the new node => next as NULL.
   (Because the new node is going to be the last node.)

3. If the head node is NULL (Empty Linked List), make the new node as the head.

4. If the head node is not null, (Linked list already has some elements),find the last node.
   make the last node => next as the new node.

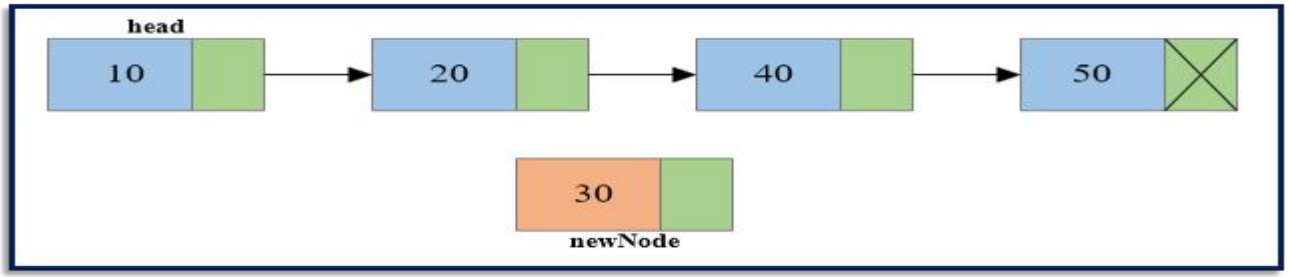
Insertion at the End

```
public void insertAtEnd(int data) {
Node newNode = new Node(data);
if (head == null) {
head = newNode;
return;
}
Node last = head;
while (last.next != null) {
last = last.next;
}
last.next = newNode;
}
```

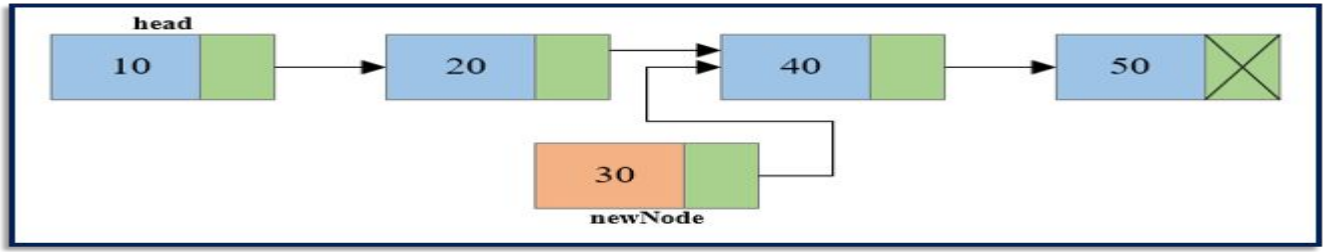
Single Linked List: Insertion at any Position

Steps to insert node at any position of Singly Linked List

Step 1: Create a new node.

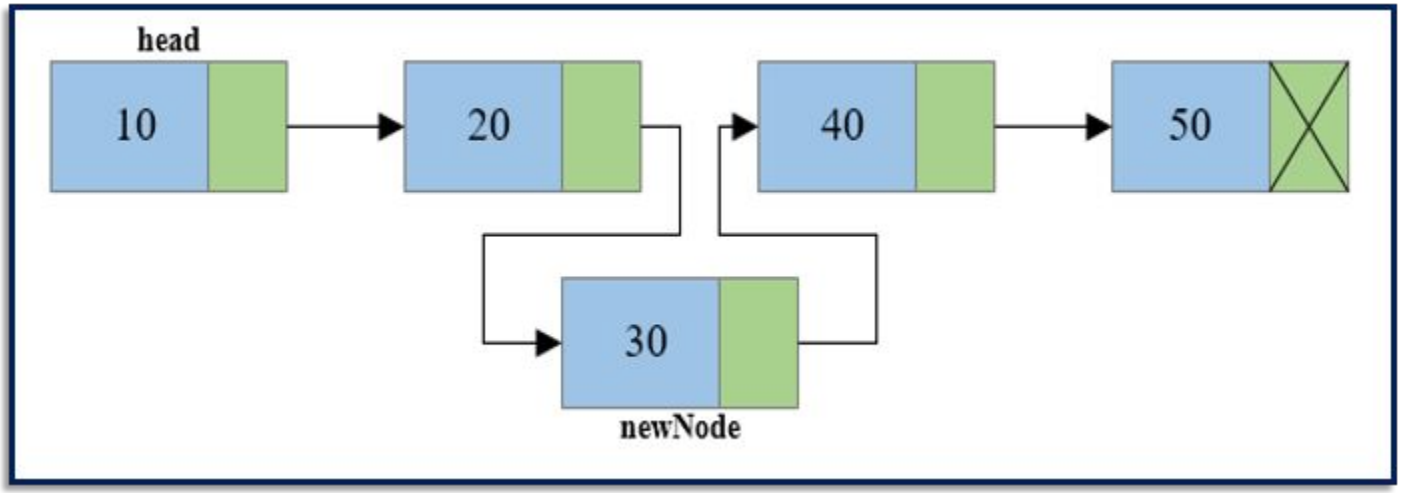


Step 2: Traverse to the n-1th position of the linked list and connect the new node with the n+1th node. (newNode->next = temp->next) where temp is the n-1th node.



Single Linked List: Insertion at any Position

Step 3: Now at last connect the n-1th node with the new node i.e. the n-1th node will now point to new node. (temp->next = newNode) where temp is the n-1th node.



Single Linked List: Deletion

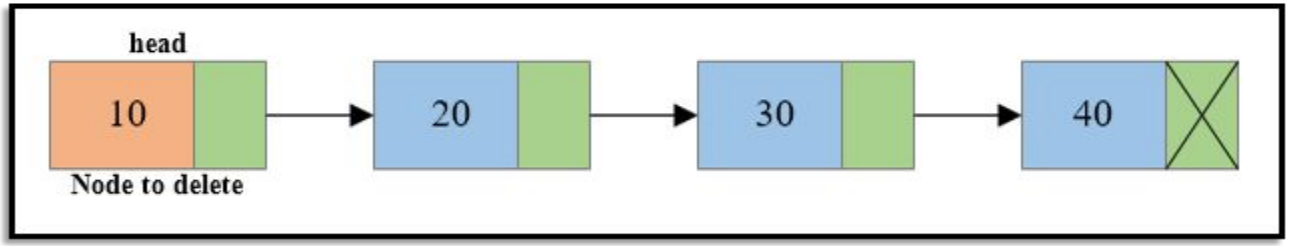
Deletion steps

- Start from the header node
- Manage links to
- Delete at front
- Delete at end
- Delete at any position

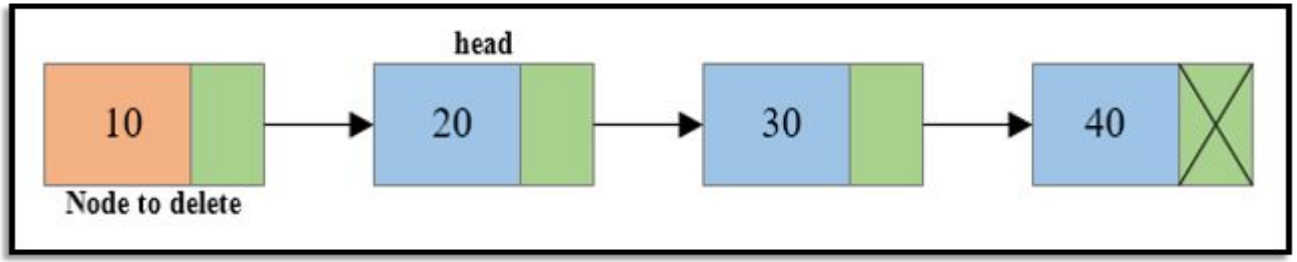
Single Linked List: Deletion at Front

Steps to delete first node of Singly Linked List

Step 1: Copy the address of first node i.e. head node to some temp variable say to Delete.

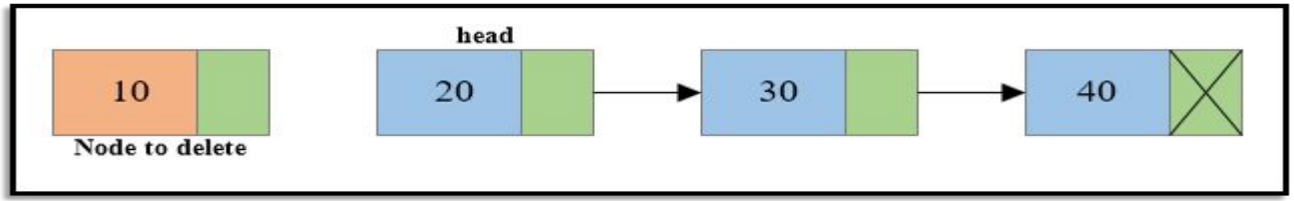


Step 2: Move the head to the second node of the linked list (head = head->next).

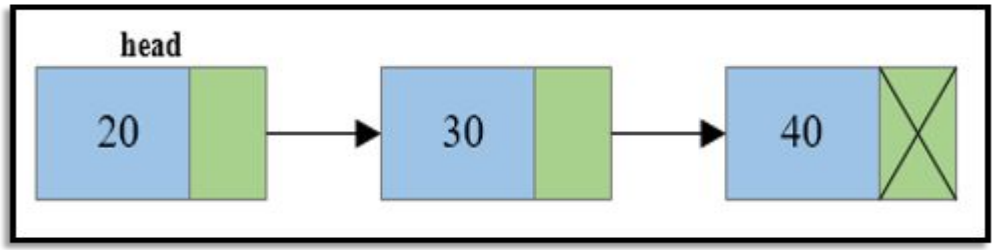


Single Linked List: Deletion at Front

Step 3: Disconnect the connection of first node to second node.



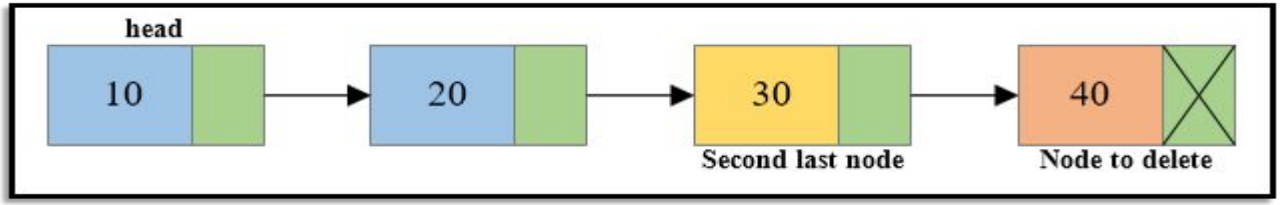
Step 4: Free the memory occupied by the first node.



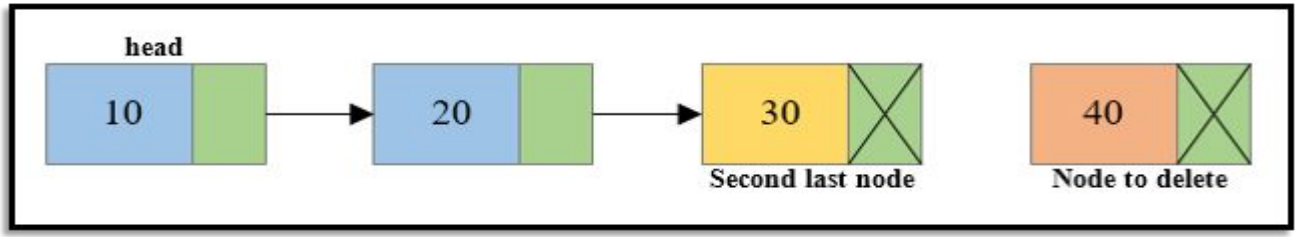
Single Linked List: Deletion at End

Steps to delete last node of a Singly Linked List

Step 1: Traverse to the last node of the linked list keeping track of the second last node in some temp variable say second Last Node.

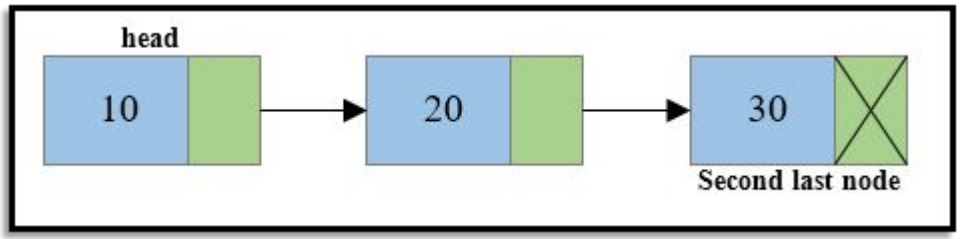


Step 2: If the last node is the head node then make the head node as NULL else disconnect the second last node with the last node i.e. second Last Node->next = NULL



Single Linked List: Deletion at End

Step 3: Free the memory occupied by the last node.



Deletion

1. If the head node has the given key,

    make the head node points to the second node and free its memory.

2. Otherwise,

    From the current node, check whether the next node has the given key

    if yes, make the current->next = current->next->next and free the memory.

    else, update the current node to the next and do the above process (from step 2) till the last node.

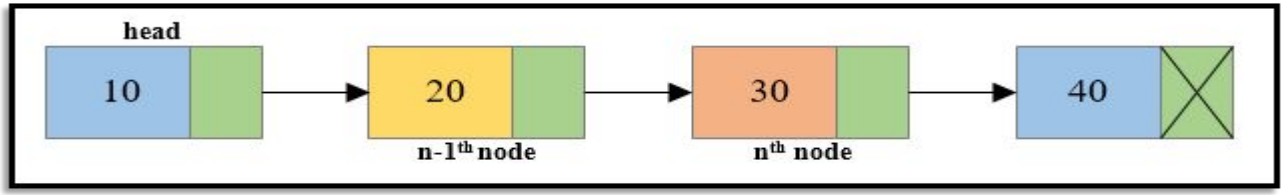
Deletion

```
public void deleteNode(int data) {

Node temp = head, prev = null;

if (temp != null && temp.data == data) {

head = temp.next;

return;

}

}
```

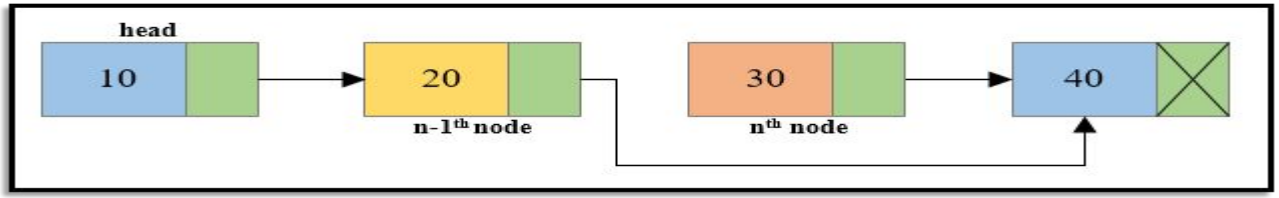
Single Linked List: Deletion at any position

Steps to delete a node at any position of Singly Linked List

Step 1: Traverse to the nth node of the singly linked list and also keep reference of n-1th node in some temp variable say prevNode.

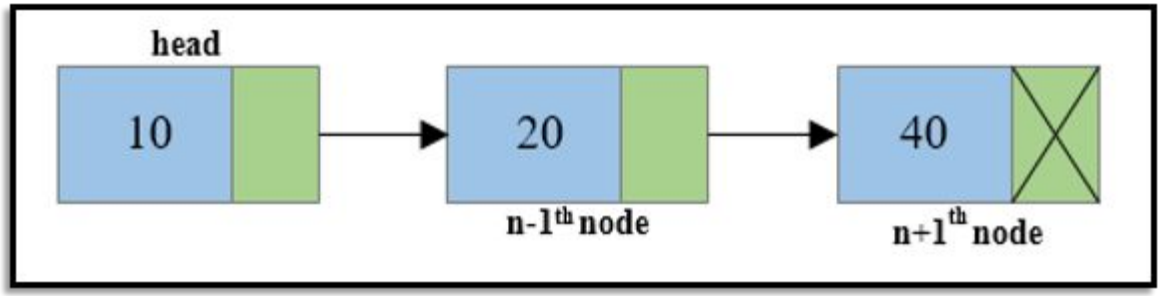


Step 2: Reconnect n-1th node with the n+1th node i.e. prevNode->next = toDelete->next (Where prevNode is n-1th node and toDelete node is the nth node and toDelete->next is the n+1th node).



Single Linked List: Deletion at any position

Step 3: Free the memory occupied by the nth node i.e. toDelete node.



Traversal

Traversal means going through each node in the list from the beginning to the end to access or modify data.

```
void display(Node head){

  Node temp = head;

  while(temp != null){

    System.out.println(temp.data);

    temp = temp.next;

  }


 }
```

Search

1. Iterate the linked list using a loop.

2. If any node has the given key value, return 1.

3. If the program execution comes out of the loop (the given key is not present in the linked list), return -1.

```
public boolean search(int data) {

Node current = head;

while (current != null) {

if (current.data == data)

return true;

current = current.next;

}

return false;

}
```

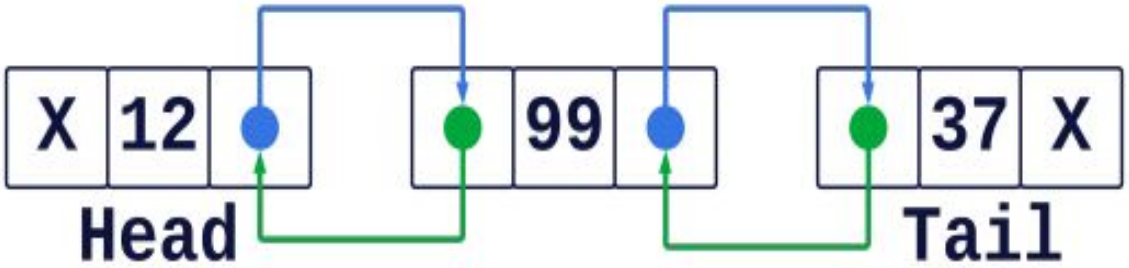
Doubly Linked List

Each node of doubly linked list (DLL) consists of three fields:

1.Item (or) Data

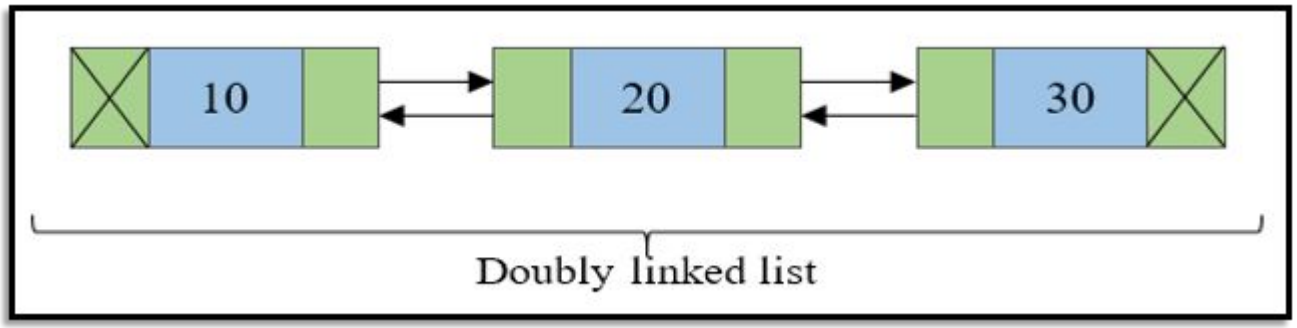
2.Pointer of the next node in DLL

3.Pointer of the previous node in DLL



Doubly Linked List

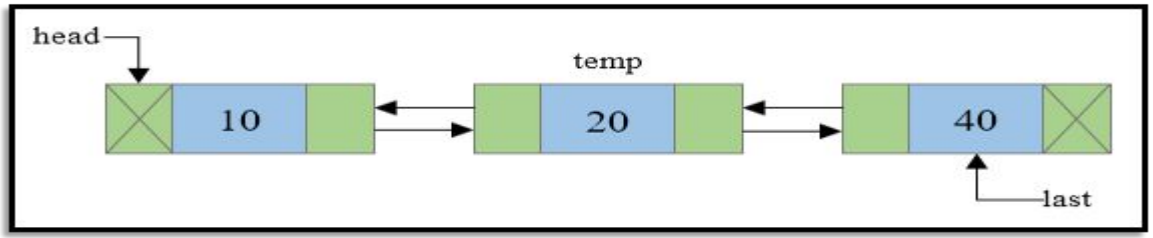
- Doubly linked list is a collection of nodes linked together in a sequential way.
- Doubly linked list is almost similar to singly linked list except it contains two address or reference fields, where one of the address field contains reference of the next node and other contains reference of the previous node.
- 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 also referred as bi-directional linked list since it allows traversal of nodes in both direction.
- Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.



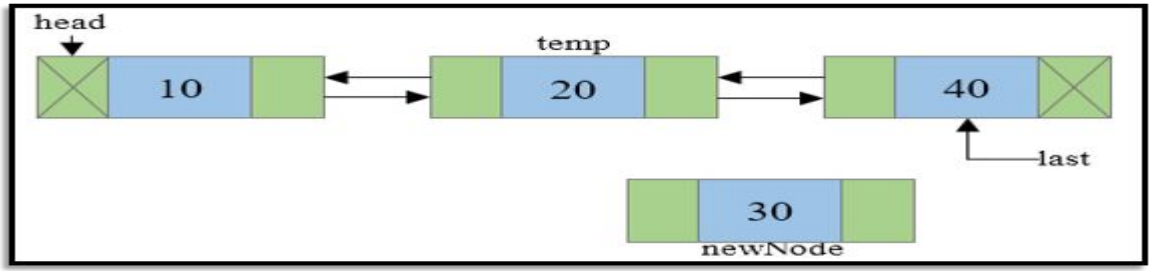
Double Linked List: Insertion at any Position

Steps to insert a new node at nth position in a Doubly linked list.

Step 1: Traverse to N-1 node in the list, where N is the position to insert. Say temp now points to N-1th node.

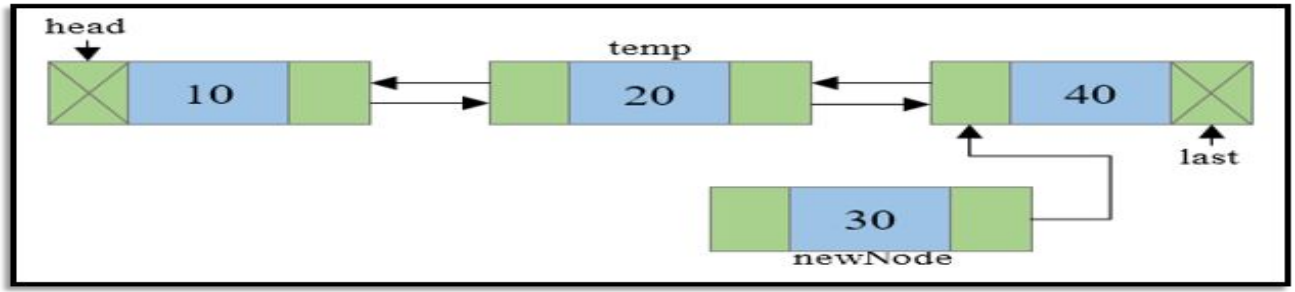


Step 2: Create a new Node that is to be inserted and assign some data to its data field.

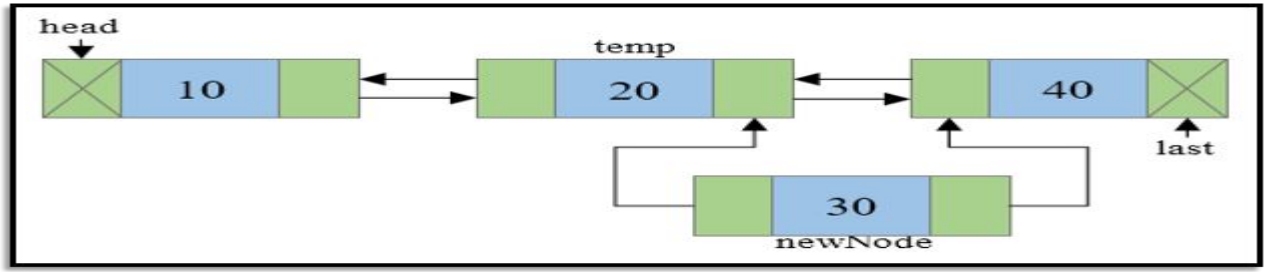


Double Linked List: Insertion at any Position

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

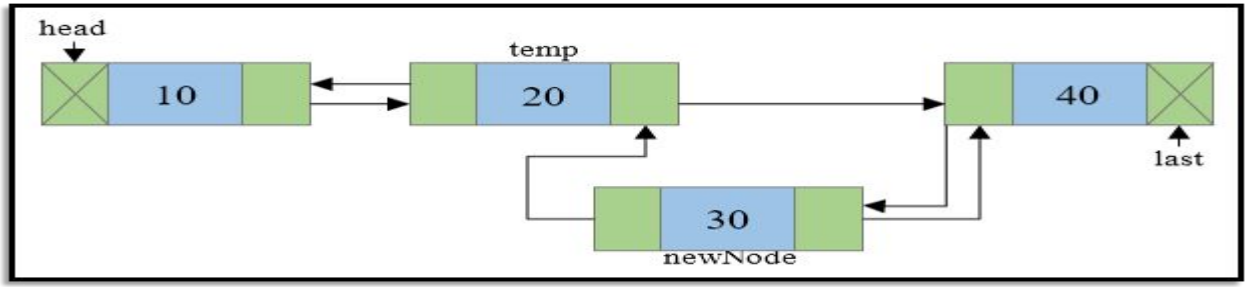


Step 4: Connect the previous address field of newNode with the temp node.

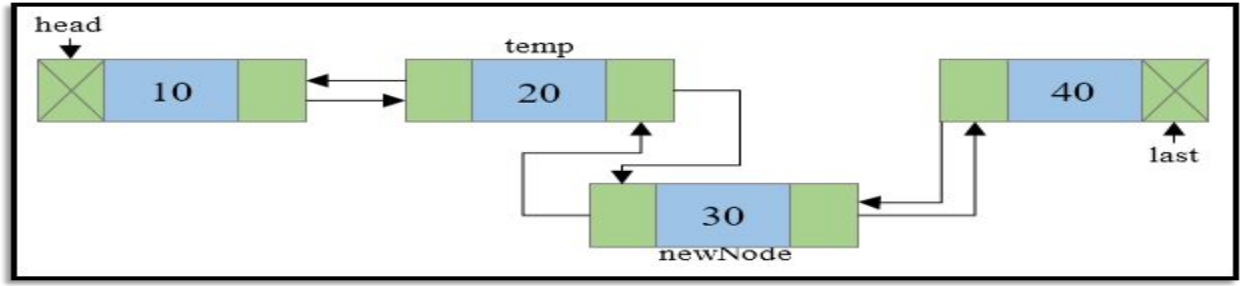


Double Linked List: Insertion at any Position

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

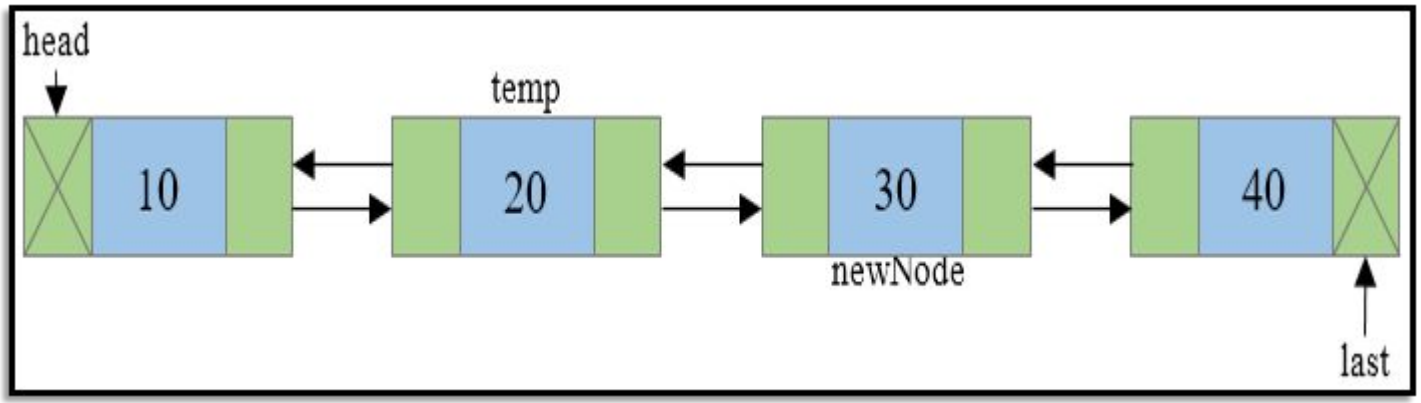


Step 6: Connect the next address field of temp node to newNode.



Double Linked List: Insertion at any Position

Step 7: Final doubly linked list looks like



Double versus Single Linked List

➢ **Advantages over singly linked list**

- A DLL can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

➢ **Disadvantages over singly linked list**

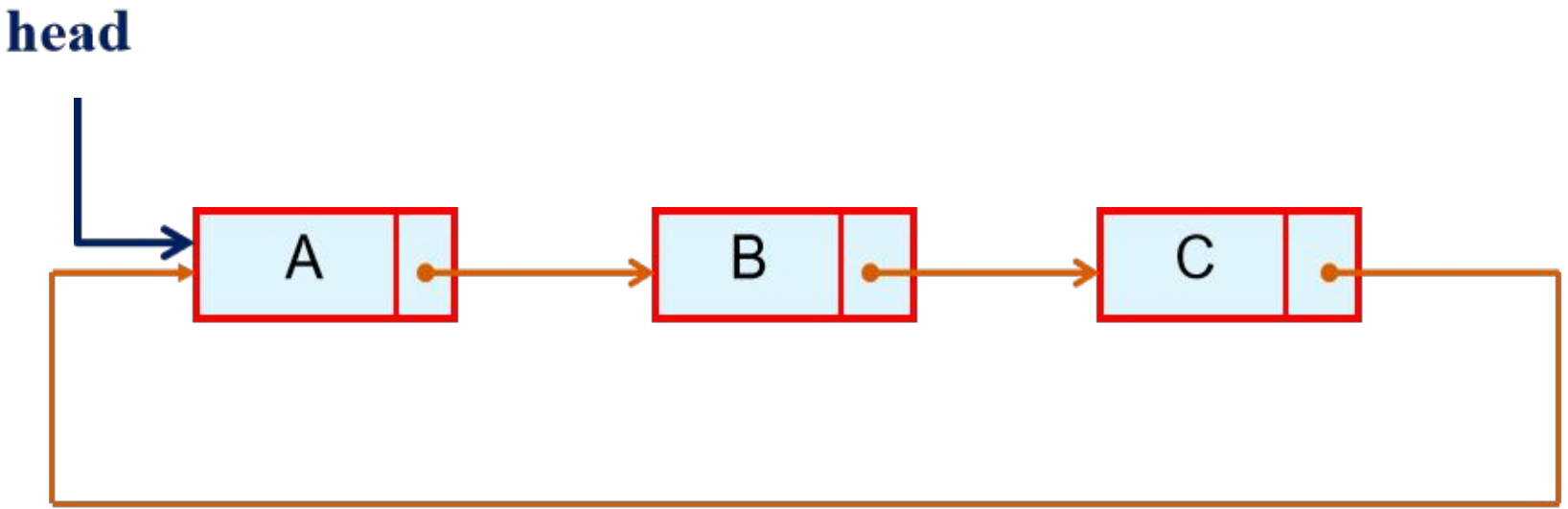
- Every node of DLL Require extra space for an previous pointer.
- All operations require an extra pointer previous to be maintained.

Implementing a Doubly Linked List

```
class Node {

int data;

Node next;

Node prev;

Node(int data) {

this.data = data;

this.next = null;

this.prev = null;

}

}
```

Circular Linked List

The pointer from the last element in the list points back to the first element.

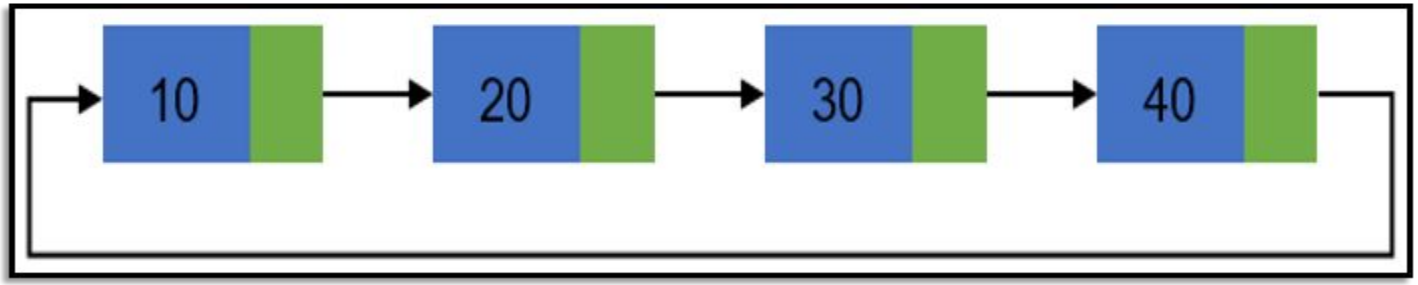


Circular Linked List

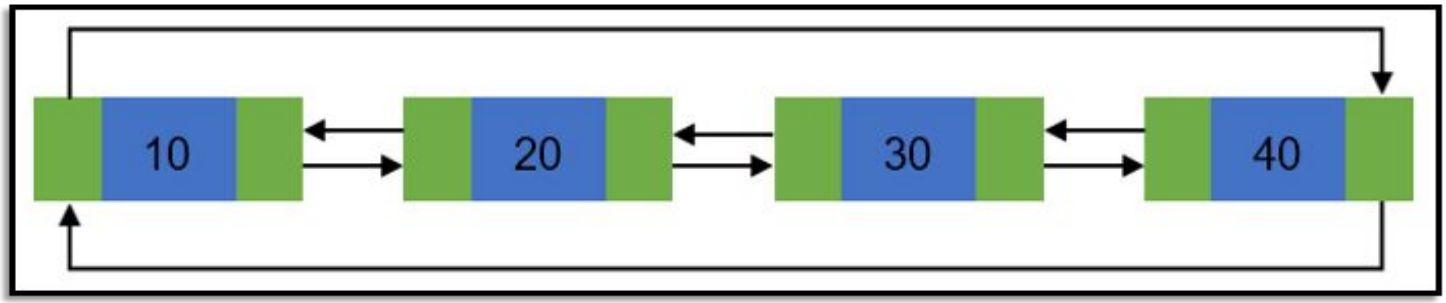
- A circular linked list is basically a linear linked list that may be single- or double-linked.
- The only difference is that there is no any NULL value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop it is called as circular linked list.
- In circular linked list there can be no starting or ending node, whole node can be traversed from any node.
- In order to traverse the circular linked list, only once we need to traverse entire list until the starting node is not traversed again.
- A circular linked list can be implemented using both singly linked list and doubly linked list.

Circular Linked List

Basic structure of singly circular linked list:



Doubly circular linked list:



Operations on circular linked list

- Creation of list
- Traversal of list
- **Insertion of node**
- At the beginning of list
- At any position in the list
- **Deletion of node**
- Deletion of first node
- Deletion of node from middle of the list
- Deletion of last node
- Counting total number of nodes
- Reversing of list

Circular Linked List

- **Advantages of a Circular linked list**
- Entire 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.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

- **Disadvantages of Circular linked list**
- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn’t supports direct accessing of elements.