

# Java Day12



## Linked List

### 1. Introduction

- A **Linked List** is a linear data structure where elements (nodes) are connected using pointers.
- Unlike arrays, linked lists do not require contiguous memory allocation.
- Each node typically contains:
  - **Data field** → stores the element.
  - **Pointer field** → stores the address of the next (or previous) node.

### 2. Types of Linked Lists

#### ◆ Singly Linked List

- Each node points to the next node.
- Traversal is **forward only**.
- Last node points to `NULL`.

#### ◆ Doubly Linked List

- Each node has two pointers:
  - `next` → points to the next node.
  - `prev` → points to the previous node.
- Allows **forward and backward traversal**.

#### ◆ Circular Linked List

- Last node points back to the head.
- Can be singly or doubly circular.
- Useful for **round-robin scheduling**.

## 4. Operations on Linked Lists

Operation	Description	Complexity
Traversal	Visit each node sequentially	$O(n)$
Insertion at Beginning	Add node before head	$O(1)$
Insertion at End	Add node after last node	$O(n)$
Insertion at Position	Add node at given index	$O(n)$
Deletion by Key	Remove node containing specific data	$O(n)$
Searching	Find node with given value	$O(n)$
Length Calculation	Count nodes until <code>NULL</code>	$O(n)$

## 5. Advantages

- Dynamic size (no need to predefine length).
- Efficient insertion/deletion compared to arrays.
- Useful for implementing **stacks, queues, graphs**.

## 6. Disadvantages

- Extra memory overhead (pointers).
- Sequential access only (no random access).
- Reverse traversal is difficult in singly linked lists.
- More complex implementation compared to arrays.

## 7. Applications

- **Stacks & Queues** implementation.
- **Graph adjacency lists**.
- **Polynomial arithmetic** (terms stored as nodes).
- **Dynamic memory allocation**.
- **Music/playlist navigation** (circular lists).

```
package DSA.Day12;

//ListNode creation
class ListNode{
    int data;
    ListNode next;
    public ListNode(int data){
        this.data=data;
        this.next=null;
    }
}
public class SinglyLinkedList {
    private ListNode head;

    public static void main(String[] args) {
        SinglyLinkedList sll=new SinglyLinkedList();
        //Creating nodes
        sll.head=new ListNode(10);
        ListNode second=new ListNode(20);
        ListNode third=new ListNode(30);
        ListNode fourth=new ListNode(40);
        //Linking nodes
        sll.head.next=second;
        second.next=third;
        third.next=fourth;
        //Operations in Linked List
        sll.display();
        sll.insertFirst(5);
        sll.display();
        sll.insert(3,75);
        sll.display();
        System.out.println(sll.deleteFirst().data);
        sll.display();
        System.out.println(sll.deleteLast().data);
        sll.display();
        System.out.println(sll.delete(4).data);
        sll.display();
    }
}
```

```

}

//Traversal
public void display(){
    ListNode current=head;
    while(current!=null){
        System.out.print(current.data+"→");
        current=current.next;
    }
    System.out.println("null");
}

//Insert at the beginning of the list
public void insertFirst(int data){
    ListNode newNode=new ListNode(data);
    if(head==null){
        head=newNode;
        return;
    }
    newNode.next=head;
    head=newNode;
}

//Insert at the end of the list
public void insertLast(int value){
    ListNode newNode=new ListNode(value);
    ListNode current=head;
    while(current.next!=null){
        current=current.next;
    }
    current.next=newNode;
}

//Insert at specific position
public void insert(int position,int value){
    ListNode newNode=new ListNode(value); //75 3

    if(position==1){
        newNode.next=head;
        head=newNode;
        return;
    }else{

```

```
ListNode previous=head;
int count=1;
while(count<position-1){
    previous=previous.next;
    count++;//2
}
ListNode current=previous.next;
previous.next=newNode;
newNode.next=current;
}
}

//Delete a first node
public ListNode deleteFirst(){
    if(head==null){
        return null;
    }
    ListNode current=head;
    head=head.next;
    current.next=null;
    return current;
}

//Delete the last node
public ListNode deleteLast(){
    if(head==null) return null;
    ListNode current=head;
    ListNode previous=null;
    while(current.next!=null){
        previous=current;
        current=current.next;
    }
    previous.next=null;
    return current;
}

//delete the node at specific position
public ListNode delete(int position){
    if(head==null){
        return null;
    }
```

```
if(position==1){  
    ListNode current=head;  
    head=head.next;  
    current.next=null;  
    return current;  
}  
else{  
    ListNode previous=head;  
    int count=1;  
    while(count<position-1){  
        previous=previous.next;  
        count++;  
    }  
    ListNode current=previous.next;  
    previous.next=current.next;  
    return current;  
}  
}  
}
```