# Lesson:

# Linked Lists in Java

# Pre Requisites:

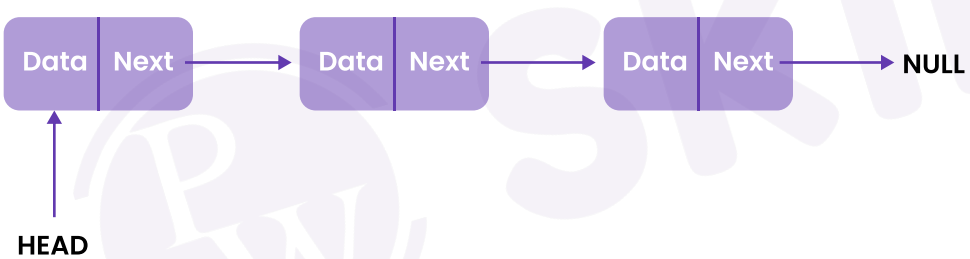
- Basic JAVA syntax

# List of concepts involved :

- Introduction to linked list
- Types of linked list
- Node in a linked list
- Insertion in a linked list
- Display of a linked list
- Deletion in a linked list
- Reverse string iteratively
- Reverse string recursively
- Middle of a linked list
- Cycle detection in a linked list

# What is a linked list ?

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

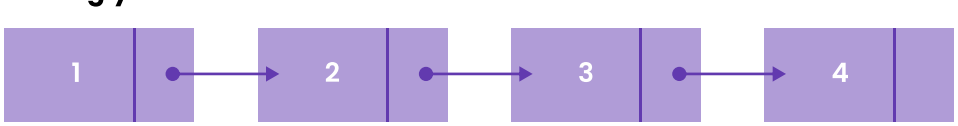



The biggest feature of a linked list is that the elements are not at contiguous memory locations. There is a possibility that the first element or node(to be more precise) is at the 1200th memory location while the second node is at the 3210th location.

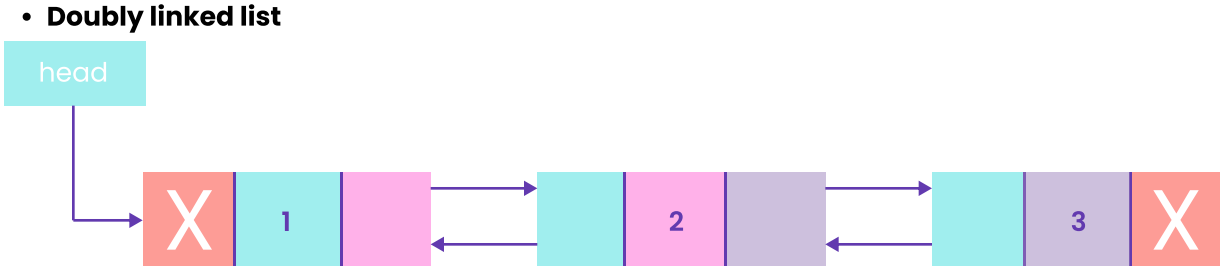
In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

There are different types of linked list :

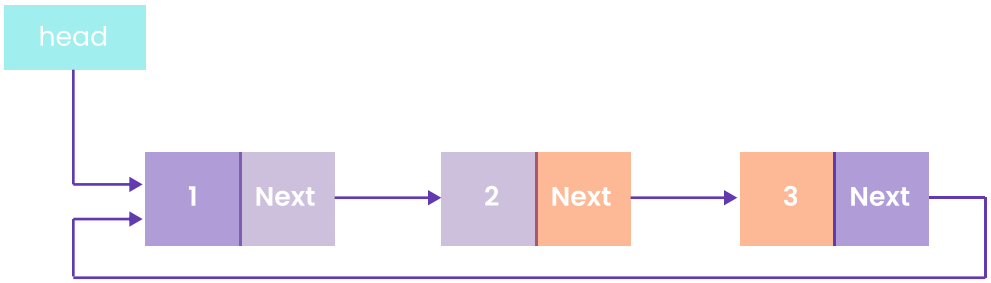
- **Singly linked list:**




- **Doubly linked list**



- **Circular linked list**




# Node :

- A node is the most basic element of a linked list. It has essentially two components. One is the data and the other one is a pointer or address of the next node.
- The first node of any linked list is termed as head node while the last node is referred as tail node. A tail node has an address field as “null” since there is no node ahead.

Implementation of a node in a singly linked list where each node has one data element and one address field.

**Code :** LP_Code1.java

**Approach :** Create a class Node which has two attributes: data and next. Next is a pointer to the next node.

**Note :** here note point is that the data can be an integral value , if you want to store string the data type can be changed to string as in “String name”.

# Insertion in a singly linked list :

Usually there are 3 types of insertion in a linked list:

- Insert at front
- Insert at middle
- Insert at last

# Insertion at last in a linked list :

**Code :** LP_Code2.java

**Approach :**

- addNodeAtLast() will add a new node to the list:
  - Create a new node.
  - It first checks whether the head is equal to null which means the list is empty.
  - If the list is empty, both head and tail will point to the newly added node.
  - If the list is not empty, the new node will be added to the end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.

Displaying the linked list: To see what we have stored in the linked list we need to print our linked list. We can traverse over the linked list just as we traverse on an array.

**Code :** LP_Code3.java

```
Nodes :
1 2 3 4
```

**Approach :**

- In the displayNodes() function, we have simply started traversing the linked list using a while loop until the node does not become null.
- We keep on printing the data stored in the current node and move to the next node.

# Deletion in a singly linked list :

Usually there are 3 types of deletion in a linked list:

- Delete at front
- Delete at middle
- Delete at last

Deletion from any general position in a linked list :

**Code :** LP_Code4.java

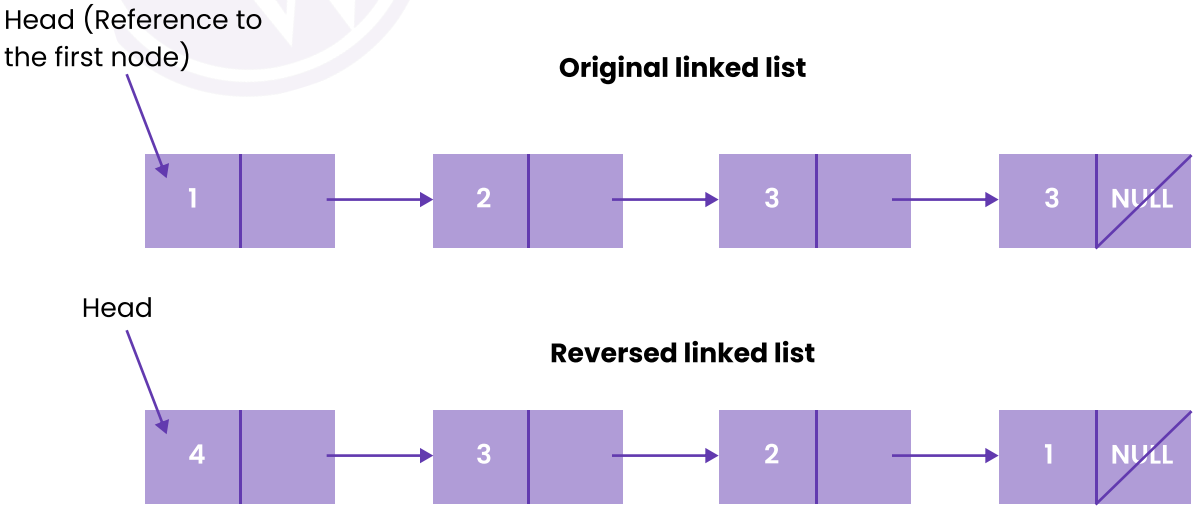
**Output :**

```
Created Linked list is:
4 3 2 1
Linked List after Deletion of 2:
4 3 1
```

**Approach :**

- To delete a node from the linked list, we need to do the following steps:
  - Find the previous node of the node to be deleted.
  - Change the next of the previous nodes.
  - Free memory for the node to be deleted.

# Reverse a linked list :




A linked list can be reversed in two different ways :

One can be an iterative approach to reverse the linked list and the second could be a recursive approach.
Iterative approach :

**Code :** LP_Code5.java

**Output :**

```
Given linked list
1 2 3 4
Reversed linked list
4 3 2 1
```

**Approach :**

- The idea is to use three pointers current, previous, and next to keep track of nodes to update reverse links.
- Initialize three pointers previous as NULL, current as head, and next as NULL.
- Iterate through the linked list. In a loop, do the following:
- Before changing the next of current, store the next node
  - next = current -> next
  - Now update the next pointer of curr to the prev
  - current -> next = previous
  - Update prev as curr and curr as next
  - previous = current
  - current = next

# Recursive approach :

**Code :** LP_Code6.java

**Output :**

```
Given linked list
4 3 2 1
Reversed linked list
1 2 3 4
```

**Approach :**

- The main concept of using recursion to reverse a linked list is to reach the last node of the linked list using recursion then start reversing the linked list.
- Divide the list in two parts – first node and rest of the linked list.
- Recursively call reverse for the rest of the linked list.
- Link the rest linked list to first and fix the head pointer to NULL.

# Middle of a linked list :

**Question : Given the head of a singly linked list, return the middle node of the linked list.If there are two middle nodes, return the second middle node.**

**Example 1:**


**Input:** head = [1,2,3,4,5]
**Output:** 3
**Explanation:** The middle node of the list is node 3.

**Example 2:**

**Input:** head = [1,2,3,4,5,6]
**Output:** 4
**Explanation:** Since the list has two middle nodes with values 3 and 4, we return the second one.

**Solution : Code :** LP_Code7.java

**Output :**

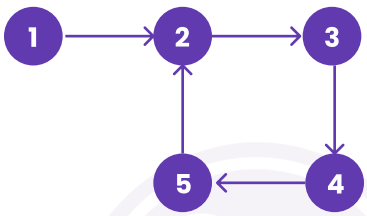
```
Given linked list
4 3 2 1
The value of middle element is : 2
```

**Approach :**

- Traverse linked list using two-pointers. Move one pointer by one and the other pointers by two.
- When the fast pointer reaches the end, the slow pointer will reach the middle of the linked list.

# Cycle detection in a linked list :

**Question : Given a linked list, check if the linked list has a loop or not. The below diagram shows a linked list with a loop.**




**Solution :**

**Code :** LP_Code8.java

**Output :**

```
Loop does exists and starts from 5
```

**Approach :**

- We will be using Floyd's cycle finding algorithm.
- Floyd's cycle finding algorithm or Hare-Tortoise algorithm is a pointer algorithm that uses only two pointers, moving through the sequence at different speeds. This algorithm is used to find a loop in a linked list. It uses two pointers, one moving twice as fast as the other one. The faster one is called the faster pointer and the other one is called the slow pointer.
- While traversing the linked list one of these things will occur-
  - The Fast pointer may reach the end (NULL) this shows that there is no loop in the linked list.
  - The Fast pointer again catches the slow pointer at some time therefore a loop exists in the linked list.

Proof of floyd's cycle finding algorithm:

Let,
X = Distance between the head(starting) to the loop starting point.
Y = Distance between the loop starting point and the first meeting point of both the pointers.
C = The distance of the loop

So before both the pointer meets-

The slow pointer has traveled X + Y + s * C distance, where s is any positive constant number.

The fast pointer has traveled X + Y + f * C distance, where f is any positive constant number.Since the fast pointer is moving twice as fast as the slow pointer, we can say that the fast pointer covered twice the distance the slow pointer covered. Therefore-

$X + Y + f * C = 2 * (X + Y + s * C)$

$X + Y = f * C - 2 * s * C$

We can say that,

$f * C - 2 * s * C = (\text{some integer}) * C = K * C$

Thus,

$X + Y = K * C \quad -(1)$

$X = K * C - Y \quad -(2)$

Where K is some positive constant.

Now if reset the slow pointer to the head(starting position) and move both fast and slow pointer by one unit at a time, one can observe from 1st and 2nd equation that both of them will meet after traveling X distance at the starting of the loop because after resetting the slow pointer and moving it X distance, at the same time from loop meeting point the fast pointer will also travel K * C – Y distance(because it already has traveled Y distance).

From equation (2) one can say that X = K * C – Y therefore, both the pointers will travel the distance X i.e. same distance after the pink node at some point to meet at the starting point of the cycle.

Here, by some point, it means that the fast pointer can complete the K * C distance out of which it has already covered the Y distance.

Conclusively, we find that slow pointer moves 1 step at a time while fast pointer moves 2 steps at a time so the fast pointer catches or surpasses slow pointer if there is a loop, just like start < end in Binary Search. Else if there exists no loop then fast pointer reaches the NULL before than slow.