24/01/25

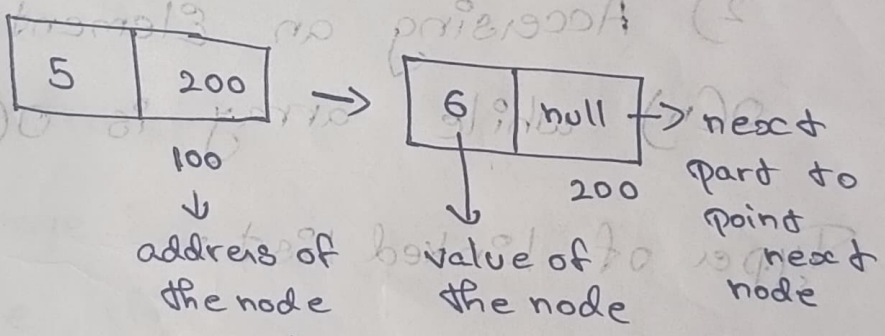
Linked List :

Linked List is an Linear Data structure where elements (nodes) are connected with each other.

Each Node has two parts :

1) Data -> Actual value stored
2) Next -> Link/Reference the point to the next node

Eg.



Basically A linked list has 2 single. Elements

* Head -> starting Element
* Last / Tail -> Final Element

In a Linked List Last Always point to next as null.

Advantages:

1) The list can Grow or Shrink [Dynamic Memory]

2) Easy Insertions / Deletions.

Disadvantage:

1) Each Node Requires additional Memory for the pointer

2) Accessing an Element has O(n) while array is O(1).

Type of Linked List:

1) Single
2) Doubly
3) Circular.

20) Find / Delete N^{th} Node from the Last [Important *]

Basic Problems for Single Linked List.

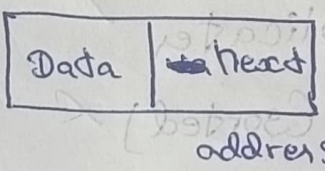
1) Traversal ✓
2) Insertion ✓
3) Deletion ✓ [updating an List] ✓ (Not written)
4) Search an Element ✓
5) Length of List ✓
6) Middle Node ✓
7) Find Loop & Remove ✓
8) Reverse a List ✓
9) Palindrome / Not ✓
10) Remove Duplicates ✓
11) Merge List (Sorted) ✓
12) Sorting of a List ✓
13) Delete Node without head / start. ✓
14) Intersection of two List ✓
15) Reverse Nodes with specified Index
16) Delete Nodes greater/less than value
17) Rotate a List ✓
18) Add two List & two Numbers of List
19) Split a List.

Single Linked List

★ Structure :

```
Class Node {
    int data;
    Node next;
    Node (int data) {
        this.data = data;
        this.next = null;
    }
}.
```

This will help the operations to create an Element/Node with next and Data Value.



★ Traversal :

Traversing through the node.

```
Public void print() {
    if (start == null)
        print ("Empty");
    Node dptr = start;
    while (dptr != null) {
        print (dptr.data + " ");
        dptr = dptr.next;
    }
```

★ Insertion :

1) At Beginning :

```
insertBegin(int data) {
    Node nn = new Node(data);
    if (start == null) {
        start = nn;
        return;
    }
    else {
        nn.next = start;
        start = nn
    }
}.
```

// Create a node (nn)
// nn's next should be start
So, it will come at start.
// change the start to nn.

2) Insert at End :

```
insertEnd(int data) {
    Node nn = new Node(data);
    if (start == null) {
        start = nn;
        return;
    }
```

```
else {
    Node tptr = start;
    for (; tptr != null; tptr = tptr.next);
    tptr.next = nn;
}
```

// Iterate to the Last Node.
// change the Last Node's next to nn.

* Insert At Position :

```
Insert At (int index, int data) {
    if (index == 0) {
        insert At Begin (data);
    } else {
        Node nn = new Node (data);
        Node tptr = start;
        for (int i = 0; i < index - 1 &&
                tptr != null; tptr = tptr.nex
        if (tptr == null)
            System.out.print ("Invalid
                Index");
        nn.next = tptr.next;
        tptr.next = nn
    }
} // Iterate till the index at change
```

★ Search an Element or Value:

```
search(int value) {
    if (start==null) {
        return -1;
    } else {
    Node tptr = start ; int i=0;
    for( tptr!=null && tptr.data != val ; i++);
     return i;
    }
}
```

// check for an Empty List

// Iterate the Node and the index till the Node (tptr) reaches the val

// Also check wheather ther tptr is null, if it is null return -1

// else Return the index that has been Iterated.

* Length of the List:

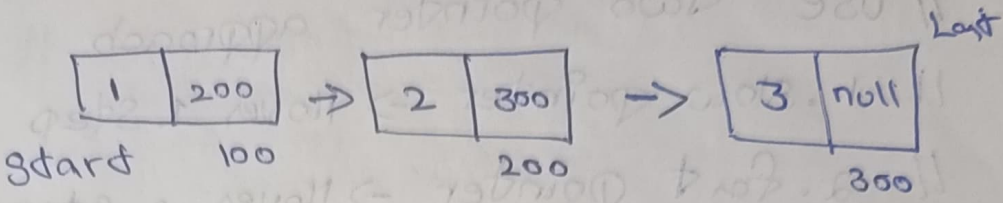
```
length() {
    Node tptr;
    int Count = 0;
    tptr = head;
    while (tptr != null)
    {
        Count ++;
        tptr = tptr.next;
    }
return Count
}
```

// Iterate the Node till the end, and Count.

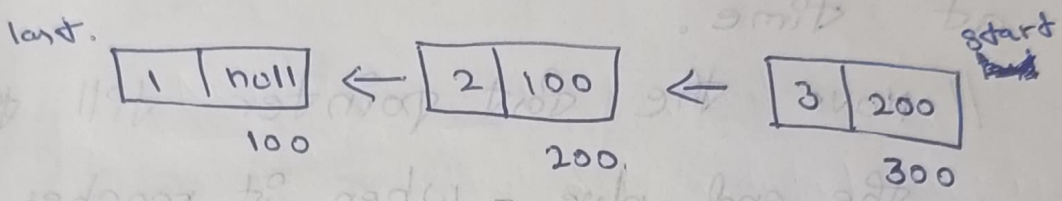
* Revers Program:

```
reverse() {
    Node prev = null;
    Node curr = start;
    Node Next = null;
    while (curr != null) {
        Next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = Next;
    }
    head = prev
}
```

* Reversing an List :



Reverse



// use three Nodes
// Prev, Curr, next
// Iterate the curr.
// Store the current's next in Next
// change the curr.next to prev
// change the current to next.

At start, Prev = null, curr = start, Next = null

Curr	Prev	Next	curr.next
100	null	20	100.next = null
200	10	30	200.next = 100
300	20	null	300.next = 20
null	30	null	Reverse Completed.

// Prev will point at last hence change start to prev

* Middle Node :

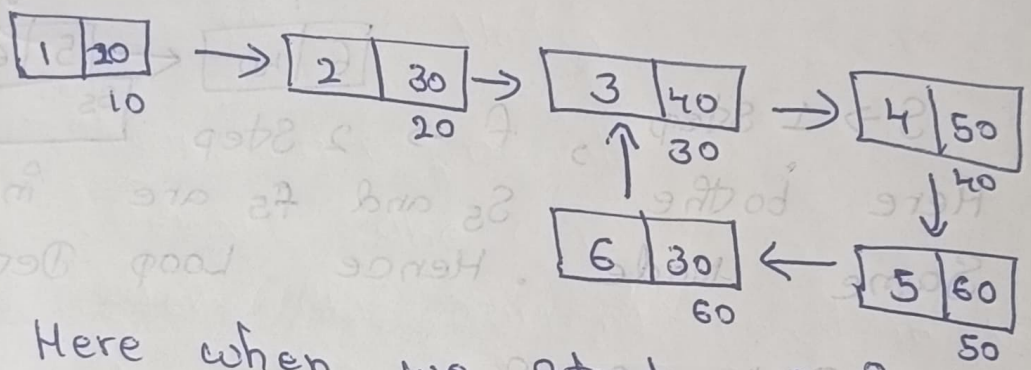
// use two pointer approach
// 1. slow pointer -> Move 1 step
// 2. fast pointer -> Move 2 steps
// Move both the Pointers at at time.
// Move the fast pointer till the the end value - when it reaches null, the data pointed by slow is middle value

```
middle (){
    Node slow = head;
    Node fast = head;
    if(head == null){
        System.print("List is Empty");
        return;
    }else{
        while(fast!=null && fast.next!=null)
        {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

★ Finding Loop & Removing it:

// In a SLL Loop occors when the Last node points the nexct to any of its previous Nodes.
[Last must point to null]

// Example:



// Here when we start printing the node

The values from address 30 to 60 Repeats, Such that

1 2 3 4 5 6 3 4 5 6 3 4 5 6 ...

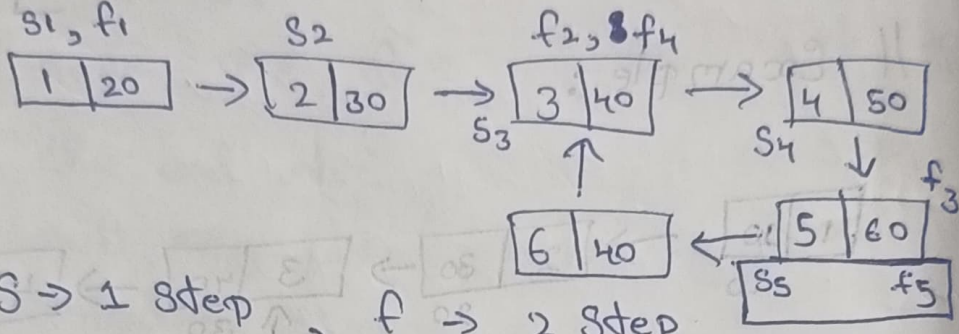
Algorithm: [Finding Loop]

Same as Middle use two pointer approach.

// slow -> Move 1 Step

// fast -> Move 2 Step.

If slow and fast meets at a certain point [Then Loop exi
Else Loop does not Exists.



S -> 1 Step , f -> 2 Step

Here both the Same Node. Hence Loop Detecte

s_5 and f_5 are in

```
loopDetect() {
    Node slow = head;
    Node fast = head;
  while (fast != null && fast.next != null)
  {
      slow = slow.next;
      fast = fast.next.next;
      if (slow == fast) {
          return true // has Loop
      }
  }
  return false
}
```

Algorithm : Loop Removal:

// use 4 pointers:

// slow = as same as Detect

// fast = as same as Detect

// prev = To keep track of slow's prev

// ptr = To remove Loop

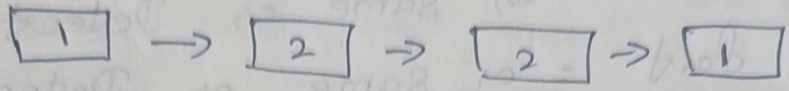
// If Loop is detected (fast == slow) start moving the ptr and slow same [1 step] . They will meet at a point [Loop start] use prev to remove the Loop.

```
RemoveLoop () {

    Node fast = head;
    Node slow = head;
    Node prev = null;
    Node ptr  = head;
    while (fast!=null && fast.next!=null){
        fast = fast.next.next;
        prev = slow;
        slow = slow.next;
        if (slow == fast) {
                ptr = ptr.next;
                prev = slow;
                slow = slow.next;
        } prev.next = null;
          return;
} }
```

★ List is Palindrome / Not :

Eg. [palindrome List]



① USING Stack :

```
palindrome (Node head) {
    Node curr = head;
    while (curr != null) {
        Stack.push(curr.data);
        curr = curr.next;
    }
    while (head != null) {
        int c = stack.pop();
        if (head.data != c) {
            return false;
        }
        head = head.next;
    }
    return true;
}
```

// Create a stack and push the elements, Check wheather the popped and head data is Equal.

② Optimal Solution: [Splitting and checking]

```
is Palindrome (head) {
    if (head == null || head.next == null)
        return true;
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    Node right = reverse (slow);
    Node left = head;
    while (right != null) {
        if (right.data != left.data {
            return false;
        }
        right = right.next;
        left = left.next;
    }
    return true;
}
// Split the List into 2
// reverse the 2nd half (right)
// Check wheather Both the lists are Equal.
```

★ Remove Duplicates:

```
remove (Node head) {
    Node dptr = head;
    while (dptr != null && dptr.next != null)
    {
        if (dptr.val == dptr.next.val)
        {
            dptr.next = dptr.next.next;
        }
        else
        {
            dptr = dptr.next;
        }
    }
    return head;
}
```

// Traverse through the List

// If dptr (data) & dptr's next (data) is equal remove one of the Node

Eg => ① -> ① -> ② | ① -> ②

Input | output.

★ Sorting of an Linked List:

★ MERGE SORT

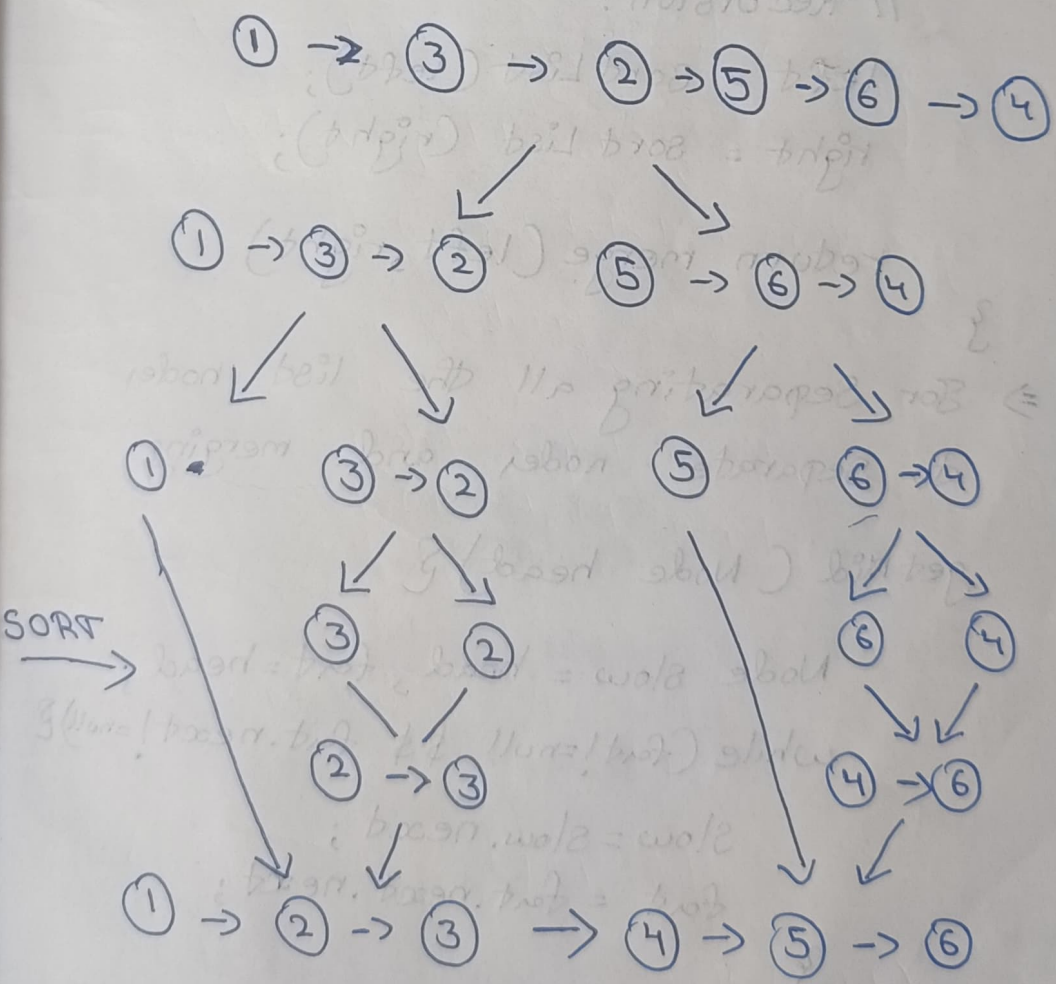
Algoritdhm:

1) Find the Middle Node.

2) Recurssively sort the Lists that are Seperated

3) Merge them.

Example:



Code:

```
* sortList (Node head) {
    if (head == null || head.next == null) {
        return head
    }
    Node mid = getMid(head);
    Node left = head;
    Node right = mid.next;
    mid.next = null;
    // Recursion:
    left = sortList(left);
    right = sortList(right);
    return merge(left, right)
}
```

⇒ For Separating all the list nodes to separate nodes and merging

```
* getMid (Node head) {
    Node slow = head, fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow
}
```

⇒ For getting the Middle Node.

* merge (Node left, Node right) {

```
Node merged = new Node(-1);
Node curr = merged;
while(left != null && right != null) {
    if(left.data < right.data) {
        curr.next = left;
        left = left.next;
    }
    else {
        curr.next = right;
        right = right.next;
    }
    curr = curr.next;
}
if(left != null)
    curr.next = left;
else
    curr.next = right;
return merged.next;
```

}

⇒ Lastly create an Empty node and add the the values of Both lists in such way that which is Smaller.

* Delete an Node without head [The node to be Deleted is given as input]

```
Delete_Node (Node value) {

    Node dptr = value;
    Node prev = null;
    while (dptr != null && dptr.next != null)
    {
        dptr.val = dptr.next.val;
        prev = dptr;
        dptr = dptr.next;
    }
    prev.next = null;
}
```

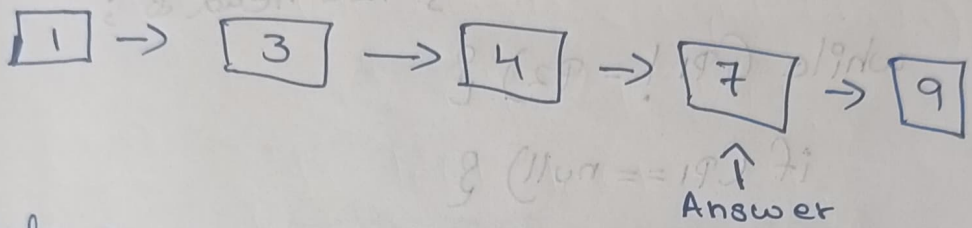
// when an head is not given, we cannot access the node's [to be Deleted] previous value.

// So, just swap the node's next value to the node and traverse for all elements.

★ Find / Delete K^{th} Node from last.

Since it is an linked-list we can't have access to last element or count of nodes, So, we have to do it manually.

Eg. K = 2



Code :

```
Delete_Nth_node (Node head, int k) {
    Node slow = head, fast = head;
    for (int i=0; i<k; i++) {
        fast = fast.next;
    }
    if (fast == null) {
        return head.next
    }
    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }
    Slow.next = slow.next.ne
```

[This condition is if head is the element from last]

★ Intersection of Two lists :

```
intersection_of_list (Node headA, HeadB)
{
  if(headA == null || headB == null) {
      return null}
  Node P1 = headA , P2 = headB ;
  while (P1 != P2) {
      if (P1 == null) {
          P1 = headB;
      } else {
          P1 = P1.next;
      }
      if (P1 == null) {
          P2 = headA;
      } else {
          P2 = P2.next;
      }
  }
  return P1;
```

// Iterate through Both the lists till any of Node is equal, If a ptr reaches end change it direction/pos to another List and vice versa.

// Both the Lists will meet at the intersection point.

★ Rotate an List :

```
rotate_list (Node head, int k) {
    if (head == null || head.next == null
        || k = 0) {
    return head
    } /
    Node dptr = head;
    int count = 1;
    while (dptr.next != null) {
        count ++;
        dptr = dptr.next;
    }
    dptr.next = head;
    k = k % count;
    k = count - k;
    while (k > 0) {
        dptr = dptr.next;
        k++
    }
    head = dptr.next;
    dptr.next = null;
    return head;
}
```

// Read the Code you will understand Automatically.