Linked List

1. Create a Node.
2. Create LinkedList.
3. Insert a node at last & at given position
4. Delete a node at Start, Last & at given position
5. find the length of LinkedList (iterative & recursive)
6. Search in LL (iterative & recursive).
7. get Nth node from start & end.
8. Print middle of LL.
9. Detect Loop
10. Check if Single LL is palindrome or not.
11. Remove duplicate from Sorted LL
12. ———————— un-Sorted LL
13. Swap node without Swapping data
14. Pairwise Swap Element of LL
15. Move last Element to front of LL
16. Intersection point of two LL.
17. Intersection of two Sorted LL
18. Reverse LL (iterative & recursive)
19. LL length Even/odd
20. Count pair whose sum is Equal to X.
21. Rotate LL (iterative & recursive)

(22) Delete alternate node.

(23) Add two number represented by LL.

(24) Identical LL.

(25) Add 1 to a number represented as LL.

(26) Union of two LL.

(27) Delete middle of LL.

(28) clone LL with next & random pointer.

(29) Sort 0s, 1s & 2s

(30) Intersection point in Y shaped LL.

(31) Remove duplicate from un-sorted LL.

(32) Remove Loop in LL.

(33) find n/Kth node in LL.

(34) Move all zeros to front of LL.

(35) Flattening LL.

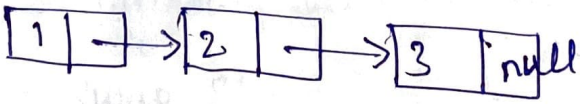
(36) LRU cache.

① Create a Node.

```
Class Node {

    int data;
    Node next;

    Node(int data) {
        data = data
    }
}
```

② Create LL.

```
Class LinkedList {
    Node head
    LinkedList LL = new LinkedList()

    LL.head = new Node(1)
    LL.head.next = new Node(2)
    LL.head.next.next = new Node(3)
}
```

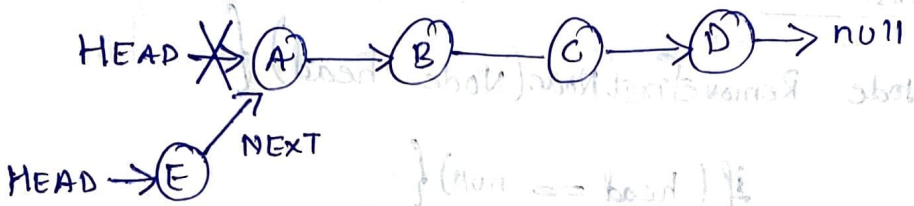


Note: traverse can be done using while loop.

```
PrintList() {

    Node n = head
    while(n.next != null){
        Print(n.data)
        n = n.next
    }
```

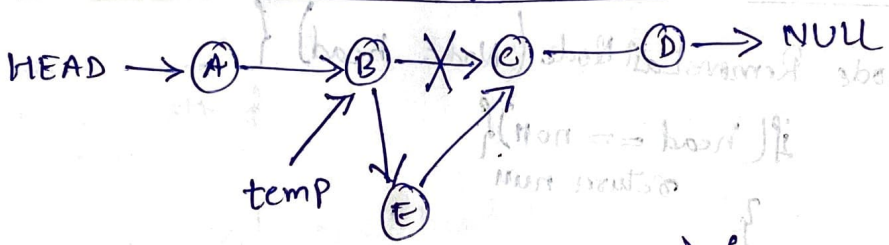
3 Insert node at last & at given position.

Add node at front




```
Add( data) {
    Node n = new Node(data)
    n.next = head
    head = n
}
```

Add node after a given node




```
insertAfter( prevNode, data) {
    Node n = new Node(data);
    n.next = prevNode.next
    PrevNode.next = n
}
```

Add Node at End

```
Node n = new Node (data);
Node last = head;
while(last.next != null){
        last = last.next
}
last.next = n
```

(4) Delete node at start, last & at given position.

Delete first Node

```
Node RemoveFirstNode(Node head) {
    if (head == null) {
        return null
    }
    Node temp = head;
    head = head.next;
    return head;
}
```

Delete last Node

```
Node RemoveLastNode(Node head) {
    if(head == null){
        return null
    }
    if(head.next == null){
        return null
    }
    Node Second_last = head
    while( Second_last.next.next != null) {
        Second_last = Second_last.next
    }
    Second_last.next = null;
    return head;
}
```

Delete given node

```
DeleteNode( int key) {

    Node temp = head , Prev = null
    if(temp == null) return;
    if( temp ≠ null && temp.data == key) {
            head = temp.next;
            return;
    }

    While( temp ≠ null && temp.data != key){
            prev = temp
            temp = temp.next
    }

    Prev.next = temp.next;

}
```

OR

```
delete Node (node) {
node.data = node.next.data
node.next = node.next.next
}
```

Recursive method

```
DeleteNode (Node head, int val) {

    if(head == null) {
        return null
    }

    if( head.data == val) {
```

⑤ find the length of linked list

iterative

```
getCount() {
    Node temp = head;
    int count = 0;
    while( temp != null){
        count++;
        temp = temp.next;
    }
    return count;
}
```

Recursive

```
getCount(Node node) {
    if (node == null){
        return 0
    }
    return 1 + getCount(node.next)
}
```

⑥ Search in linked list

iterative

```
Search(Node head, int x) {
    Node current = head;
    while( current != null) {
        if( current.data == x){
            return current
        }
        else {
            current= current.next;
```

Recursive

```
Search(Node head, int x){
    if(head == null){
        return false
    }
    if(head.data == x){
        return head
    }
    else{
        return Search(head.next, x)
    }
}
```

② get Nth Node ~~from start & end~~.

Iterative

```
GetNth(int index){
    Node current = head
    int count = 0
    while(current ≠ null){
        if(count == index){
            return current.data;
        }
        count++;
        current = current.next;
    }
}
```

Recursive

```
GetNth(Node head, int n){
    int count = 0;
    if(head == null){
        return null
    }
    if(count == n){
        return head.data
    }
    return GetNth(head.next, n-1)
}
```

⑧ Print Middle Element of LL

Method 1:

→ Traverse whole list & count total no of nodes.
→ Now traverse the list again till count/2 & return the node at count/2.

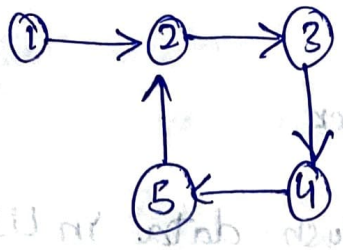
Method 2: ~~Floyd Cycle~~

→ Traverse the LL using 2-pointers.
→ Move one pointer by one step & other by 2 step.
→ When fast pointer reach end, the same time slow pointer will be at middle of LL.

```
PrintMiddle( ) {
    Node slow = head
    Node fast = head
    if( head != null) {
        while( fast ≠ null & fast.next ≠ null){
            slow = slow.next
            fast = fast.next.next
        }
    Print (slow.data);
        OR
    return slow.data;
    }
}
```

(9) Detect Loop in LL & remove loop

I/P :




```
boolean DetectCycle(node head) {
    node slow = head
    node fast = head
    while(slow && fast && fast.next) {
        slow = slow.next
        fast = fast.next.next
        if(slow == fast) {
            removeLoop(slow, head)
            return true
        }
    }
}

void RemoveLoop(node slow, node head) {
    node p1 = slow
    node p2 = head
    while(p1.next ≠ p2.next) {
        p1 = p1.next
        p2 = p2.next
    }
    p1.next = null;
}
```

(10) Check LL is pallindrome or not.

Method 1: using Stack

→ Traverse & push data in LST
→ Traverse again & pop-up data
→ Compare value if it is same till end
If is pallindrome else not.

Method 2: By Reversing the list

(1) Get Middle of LL.
(2) Reverse the second half.
(3) Check 1st half & 2nd half are same
(4) Construct the original LL by
→ reversing 2nd half again
→ Attach it back to 1st half.

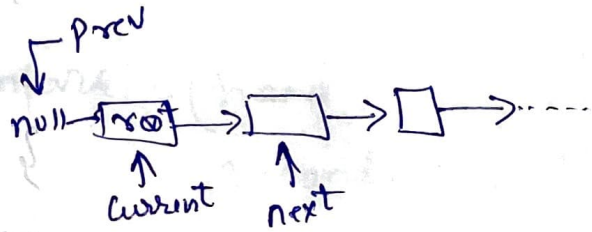
Middle of LL

Method 1: – count total node
– traverse again till count/2

Method 2:
– slow & fast pointer

```
Node slow = fast = head;
while( fast ≠ null & fast.next ≠ null) {
        slow = slow.next
        fast = fast.next.next
}
return slow
}
```

Reverse LL



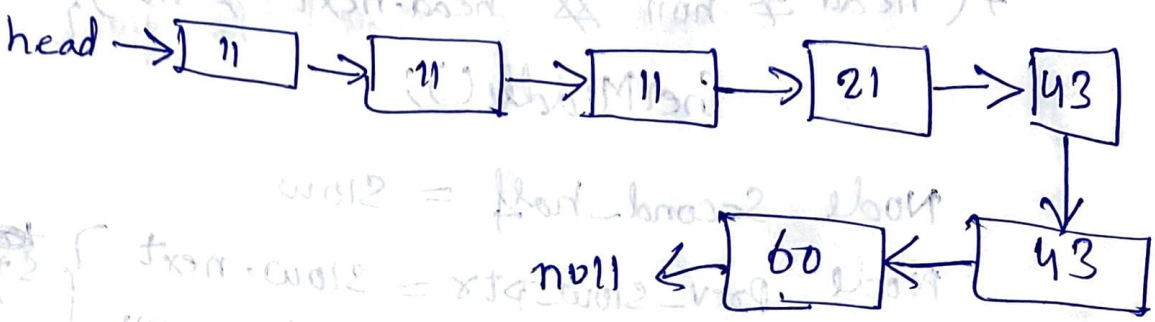

```
ReverseLL(node root) {
    Node prev = null
    Node current = root
    Node next
    while( current ≠ null) {
        next = current.next
        current.next = prev
        prev = current
        current = next
    }
    root = prev
    return root
}
```

Compare LL

```
CompareLL(Node n1, Node n2) {
    Node n1 = h1
    Node n2 = h2
    while(n1 != null && n2 != null) {
        if(n1.data == n2.data){
            n1 = n1.next
            n2 = n2.next
        } else {
            return false
        }
    }
    if(n1 == null && n2 == null){
        return true
    }
    return false
}
```

```
(Node head){
if( head ≠ null && head.next ≠ null){
        GetMiddle();
Node second_half = slow
Node prev_slow_ptr = slow.next
     Prev_slow_ptr.next = null
                                  } stop 1st half
        ReverseLL( second_half)
boolean res = compareLL(head,
                        second_half);
prev_slow_ptr.next = ~~Reverse~~
                     ReverseLL(second_half)
}
return res
}
```

(11) Remove Duplicate from Sorted LL.



Soln

```
RemoveDuplicate() {

    Node curr = head;
    while( curr ≠ null) {

        Node temp = curr
        while( temp ≠ null &&
                temp.data == curr.data)
        {

            temp = temp.next
        }

        curr.next = temp
        curr = curr.next
    }
```

(12) Remove duplicate from un-sorted LL.

Method 1: Using two-loops

```
RemoveDuplicate() {
    Node p1 = p2 = null, dup = null
    p1 = head
    while(p1 ≠ null && p1.next ≠ null) {
        p2 = p1
        while(p2.next ≠ null) {
            if(p1.data == p2.next.data) {
                dup = p2.next;
                p2.next = p2.next.next;
            } else {
                p2 = p2.next
            }
        }
        p1 = p1.next
    }
}
```

Method 2: Sorting

→ Sort the Element

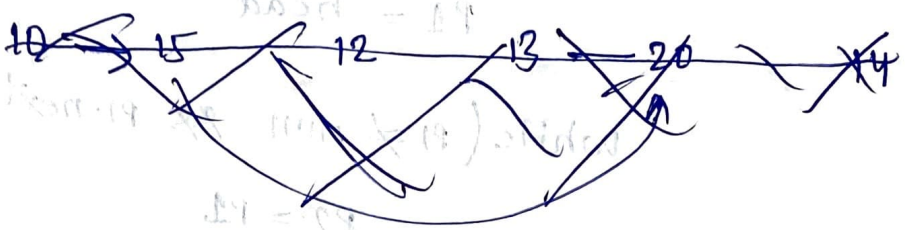
→ apply algorithm for sorted LL

(13) Swap node without swapping data.

I/P

10 → 15 → 12 → 13 → 20 → 14

x = 12, y = 20



O/P

10 → 15 → 20 → 13 → 12 → 14

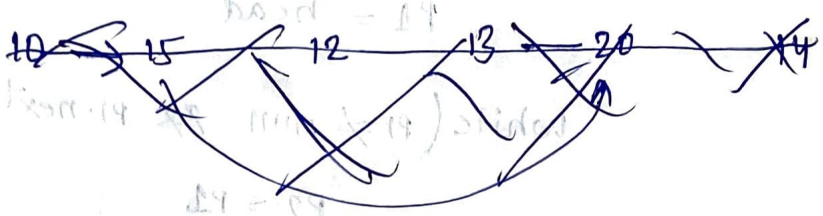
```
SwapNode(int x, int y) {
    if(x == y) return
    Node PrevX = null  CurrX = head
    while(CurrX ≠ null && CurrX.data ≠ x){
        PrevX = CurrX
        CurrX = CurrX.next
    }

    Node PrevY = null  CurrY = head
    while(CurrY ≠ null && CurrY.data ≠ y){
        PrevY = CurrY
        CurrY = CurrY.next
    }

    if(CurrX == null || CurrY == null){
        return
    }

    if(PrevX ≠ null){
        PrevX.next = CurrY
    }else{
```

(13) Swap node without swapping data.

I/P

10 → 15 → (12) → 13 → (20) → 14

x = 12, y = 20

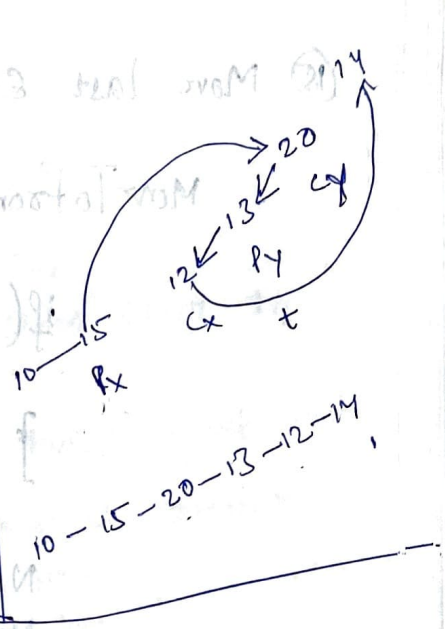


O/P

10 → 15 → (20) → 13 → (12) → 14

```
SwapNode (int x, int y) {

    if(x == y) return
    Node PrevX = null   CurrX = head
    while( CurrX ≠ null && CurrX.data ≠ x){
            PrevX = CurrX
            CurrX = CurrX.next
    }

    Node PrevY = null   CurrY = head
    while( CurrY ≠ null && CurrY.data ≠ y){
            PrevY = CurrY
            CurrY = CurrY.next
    }

    if( currX == null || currY == null){
            return
    }

    if( PrevX ≠ null) {
            PrevX.next = currY
    }else {
            head = currY
    }
```

```
if( PrevY ≠ null){
        PrevY.next = currX
}else{
        head = currX
}

// Swap next pointers
Node temp = currX.next
currX.next = currY.next
currY.next = temp
}
```



(iv) Pairwise Swap Element of LL.

I/P : (10) — (20) — (30) — (40) — (50) — (60)

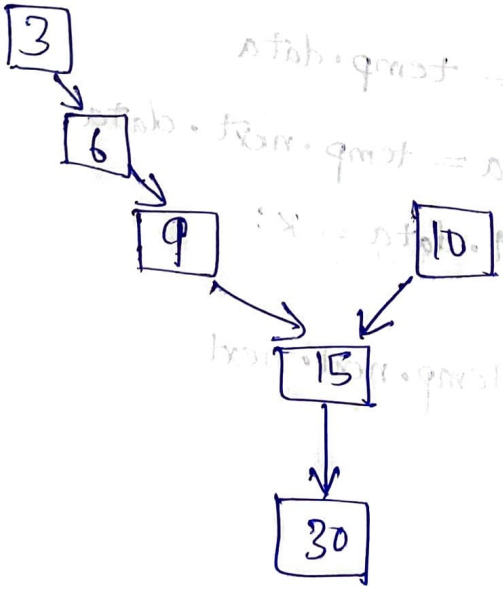
O/P : (20) — (10) — (40) — (30) — (60) — (50)

```
PairwiseSwap( ){

    Node temp = head
    while( temp ≠ null && temp.next ≠ null){

        int k = temp.data
        temp.data = temp.next.data
        temp.next.data = k;

        temp = temp.next.next
    }
}
```

(15) Move last Element to front of LL

```
MoveToFront() {
    if( head == null || head.next == null){
        return;
    }

    Node Seclast = null
    Node last = head

    while( last.next != null){
        Seclast = last
        last = last.next
    }

    Seclast.next = null
    last.next = head
    head = last
}
```

(16) Intersection point of two LL.



Approach : Using difference of node count

① Get total count of 1st list = c1
② ——— 2nd ——— = c2
③ Get difference = abs(c1 - c2)
④ Now traverse big list from start to no of difference step.
⑤ Now from here both list have equal node
⑥ Traverse both list if value is common that is intersection point.

Sim

```
getNode() {
    int c1 = getCount(head 1);
    int c2 = getCount(head 2);
    int d;
    if(c1 > c2){
        d= c1 - c2
        return getIntersectionNode(d, head1, head2)
    }
    else {
        d = c2 - c1
        return getIntersectionNode(d, head2, head1)
    }
}

getIntersectionNode(d, node1, node2) {
    Node c1 = node1
    Node c2 = node2
    for(i=0; i<d; i++) {
        if(c1 == null) return -1
        c1 = c1.next
    }
```

```
while( c1 ≠ null && c2 ≠ null) {
    if( c1.data == c2.data) {
        return c1.data
    }
    c1 = c2
    c2 = c2.next;
}
return -1
}

getCount( node) {
    Node c = node, int Count = 0
    while( c ≠ null){
        Count ++
        c = c.next
    }
    return Count
}
```

(17) Intersection of 2 Sorted LL

(18) Reverse LL

Given pointer to head node of LL, the task is to reverse the LL, we need to reverse list by changing only links

I/P 1 → 2 → 3 → 4 → null

O/P 4 → 3 → 2 → 1 → null

(1) Iterative Method

```
Reverse (Node node) {

    Node prev = null
    Node curr = node
    Node next = null

    while( curr ≠ null) {

        next = curr.next
        current.next = prev
        prev = current
        current = next
    }

    node = prev
    return node
}
```

(2) Recursive Method

```
Reverse(Node head){
    if(head == null || head.next == null){
        return head
    }
    Node rest = Reverse(head.next);
    head.next.next = head
    head.next = null
    return rest
}
```

(2) Using Stack

(19) LL length Even/odd

Check if length is Even or odd.

Method 1: Count the total nodes by traversing linearly & check if it is Even or odd.

Method 2: Stepping 2 nodes at a time

→ Take a pointer & move it two nodes at a time.

→ At the End if pointer is null then length is Even else odd.

```
LinkedListLength(Node head) {
    while( head ≠ null && head.next ≠ null) {
        head = head.next.next
    }
    if ( head == null) {
        return "ODD"
    }else {
        return "EVEN"
    }
}
```

Count pair from two LL, whose sum is equal to X.

Method 1: Use two nested for loop.

Method 2: Use Sorting

→ Sort the 1st LL in ascending

→ ——— 2nd ——— descending.

```
countPairs(LL h1, LL h2, int x) {

    int count = 0
    Collections.sort(h1)
    Collections.sort(h2, Collections.reverseOrder());

    Iterator itr1 = h1.Iterator();
    ———— itr2 = h2. ————

    Integer num1 = itr1.hasNext() ? itr1.next() : null;
    ———— num2 = itr2. ———— itr2. ————

    while(num1 ≠ null && num2 ≠ null) {

        if( (num1 + num2) == x) {
            num1 = itr1.hasNext() ? itr1.next() : null;
            num2 = itr2. ————
            count++
        }
        else if( (num1 + num2) > x) {
            num2 = itr2.hasNext() ? itr2.next() : null
        }
        else {
            num1 = itr1. ————
        }

        return count
    }
}
```

(21) Rotate LL by K nodes.

```
Void Rotate(Int K) {

    if( K == 0) {
        return
    }

    Node Current = head;
    Int Count = 1

    while(count < K && Current ≠ null){
        Current = Current.next
        Count ++
    }

    if( Current == null){
        return;
    }

    Node KNode = Current
    while( Current.next ≠ null){
        Current = Current.next
    }

    Current.next = head;
    head = KNode.next
    KNode.next = null;

}
```

(22) Delete alternate node.

Method 1: Iterative

```
Void deleteAt() {
    if(head == null) return
    Node prev = head
    Node now = head.next

    while( prev ≠ null & now ≠ null) {
        Prev.next = now.next
        now = null;

        prev = prev.next
        if( prev ≠ null) {
            now = prev.next
        }
    }
}
```

Method 2: Recursive

```
Node deleteAt( Node head) {
    if(head == null) return
    Node node = head.next
    if(node == null) return
    head.next = node.next;
    head.next = deleteAt(head.next);
}
```