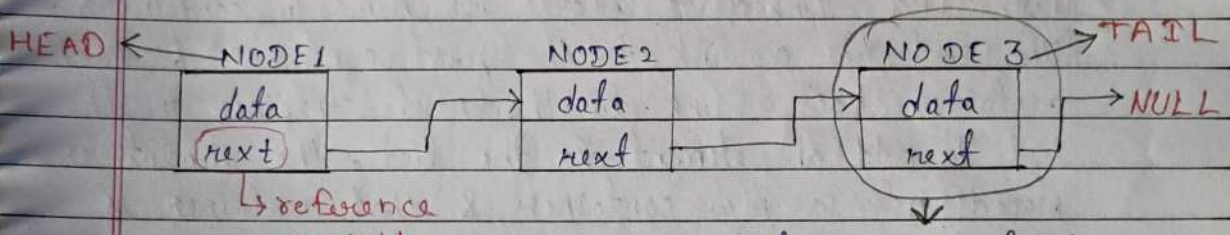
LINKED LIST - I

* Linked List is a linear data structure in which elements, called nodes, are connected using pointers.
* Each node contains two parts :-
  - Data - the actual value or information contained in the node.
  - Next - a pointer or reference to the next node in sequence.

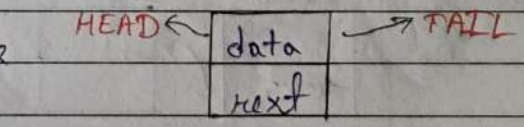


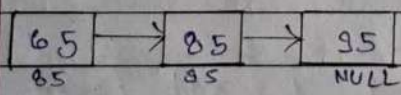
* Last node points to null, indicating the end of the list.

HEAD - first node in the Linked list.
TAIL - Last node in the Linked list.

* In Java, a Linked List can be implemented using a custom class or by using the built-in LinkedList class provided by the Java Collections Framework.

* Same Head & Tail of same node.





⇒
```
class Node {
    int data;
    Node next;
    public Node (int data) {
        this.data = data;
        this.next = null;
    }
}
```

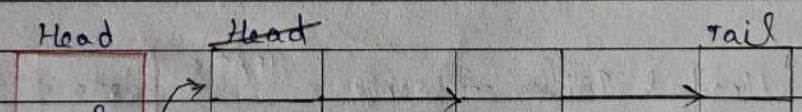
★ DIFFERENCE B/T AL & LL

* ArrayList : ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one & the old one is removed.
* LinkedList : LinkedList stores its items in "containers". The list has a link to the first container & each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container & that container is linked to one of the other containers in the list.

NOTE : Use an ArrayList for storing & accessing data, LinkedList to manipulate data.

★ ADD IN LINKED LIST

i) add First (as head) ⟶ O(1).
ii) add Last (as tail) ⟶ O(1)



1. Create new node.
2. new node's next = head
3. head = new Node.

} add first.

* Remember, if we have add any node in the list firstly we have to create.

// User input is possible but it is lengthy.

DATE

240

* adding at last:
1. Create a new Node.
2. tail.next = newNode.
3. tail = newNode

★ PRINTING A LINKED LIST

⇒
```
public class LinkedList {
    public static class Node {
        int data;
        Node next;
        public Node (int data) {
            this.data = data;
            this.next = null;
        }
    }
    public static Node head;
    public static Node tail;
    public void addFirst (int data) {
        //S1 : Create new node
        Node newNode = new Node (data);
        if (head == null) {
            head = tail = newNode;
            return;
        }
        //S2 : new node's next = head
        newNode.next = head;
        //S3 : Head = new Node
        head = newNode;
    }
    public void addLast (int data) {
        Node newNode = New Node (data);
        if (head == null) {
            head = tail = newNode;
            return;
        }
        tail.next = newNode;
        tail = newNode;
    }
}
```

making a LL

HEAD

TAIL

Adding At beginning of LL

Adding At Last of LL

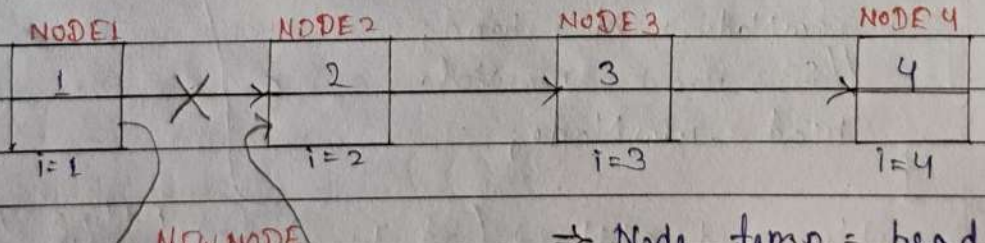
```
public void print() {//O(n)
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data+"->");
        temp = temp.next;
    }
    System.out.println("null");
}
```

Printing a LL

```
public static void main (String args[]) {
    LinkedList LL = new LinkedList();
    LL.print();
    LL.addFirst(2);
    LL.print();
    LL.addFirst(1);
    LL.print();
    LL.addLast(3);
    LL.print();
    LL.addLast(4);
    LL.print();
}
}
```

Output: null
2->null
1->2->null
1->2->3->null
1->2->3->4->null

★ ADD IN THE MIDDLE ⇒ add (index, data).



⇒ Node temp = head
i = 0
while (i < idx-1) {
temp -> next
i++
}

```
⇒ public class LinkedList {
    public void add (int idx , int data) {
        Node newNode = new Node (data);
        Node temp = head;
        int i = 0;
        while (i < idx -1) {
            temp = temp.next;
            i++;
        }
        //i = idx -1;  temp -> prev
        newNode.next = temp.next;
        temp.next = newNode;
    }
    public void print () {
        Node temp = head;
        while (temp != null) {
            System.out.print (temp.data + "->");
            temp = temp.next;
        }
        System.out.println ("null");
    }
    public static void main (String args []) {
        LinkedList ll = new LinkedList();
        ll.addFirst (2);
        ll.addFirst (1);
        ll.addLast (3);
        ll.addLast (4);
        ll.add (2, 9);
        ll.print ();
    }
}
```

public static class Node {
}

addFirst () {
}
addLast () {
}

Output : 1->2->9->3->4->null

★ SIZE OF A LINKED LIST

⇒
```
public class LinkedList {
  public static class Node {
  }
  public static Node head;
  public static Node tail;
  public static int size;
  public void addFirst (int data) {
    Node newNode = new Node (data);
    size ++;
  }
  public void addLast (int data) {
    Node newNode = new Node (data);
    size ++;
  }
  public void add (int idx, int data) {
    if (idx == 0) {
      addFirst (data);
      return;
    }
    Node newNode = new Node (data);
    size ++;
  }
  public void print () {
  }
  public static void main (String args[]) {
    System.out.println (ll.size);
  }
}
```

JAVA supports Garbage Collector.

DATE

★ REMOVE IN LINKED LIST

i) removeFirst() ⟶ In both garbage collector plays an
ii) removeLast() ⟶ essential role for deleting the first or last node.

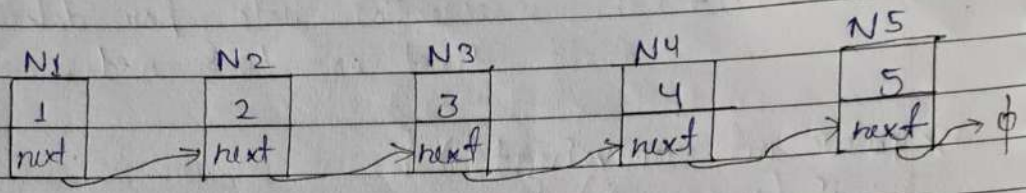
⇒
```
public int removeFirst() {
    if (size == 0) {
        System.out.println("LL is Empty");
        return Integer.MIN_VALUE;
    } else if (size == 1) {
        int val = head.data;
        head = tail = null;
        size = 0;
        return val;
    }
    int val = head.data;
    head = head.next;
    size--;
    return val;
}
```

Output: Original L.List
removeFirst L.List
removeLast L.List
Size of L.List

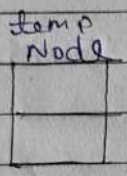
```
public int removeLast() {
    if (size == 0) {
        System.out.println("LL is empty");
        return Integer.MIN_VALUE;
    } else if (size == 1) {
        int val = head.data;
        head = tail = null;
        size = 0;
        return val;
    }
    Node prev = head;
    for (int i = 0; i < size-2; i++) {
        prev = prev.next;
    }
    int val = prev.next.data;
    prev.next = null;
    tail = prev;
    size--;
    return val;
}
```

★ ITERATIVE SEARCH

→ Search for key in a linked list. Return the position where it is found. If not found, return -1.



Key = 4



* Temp Node will check all node Linearly O(n) & if not found then return -1.

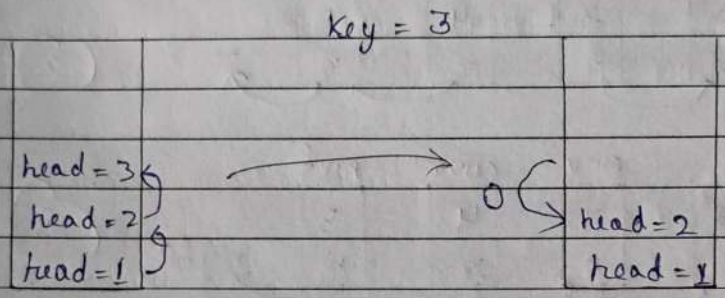
⇒
```
public int itrSearch (int key) {
    Node temp = head;
    int i = 0;
    while (temp != null) {
        if (temp.data == key) {
            return i; // key found
        }
        temp = temp.next;
        i++;
    }
    return -1; // key not found.
}
public static void main (String args[]) {
    LinkedList ll = new LinkedList ();
    ll.addFirst (2); ll.addFirst (1); ll.addLast (4);
    ll.addLast (5); ll.add (2, 3);
    ll.print (); // 1->2 -> 3->4 -> 5
    System.out.println (ll.itrSearch (3)); // 2
    System.out.println (ll.itrSearch (10)); // -1
}
```

★ RECURSIVE SEARCH - Search for a key in a Linked list. Return the position where it is found. If not found, return -1. Use Recursion.

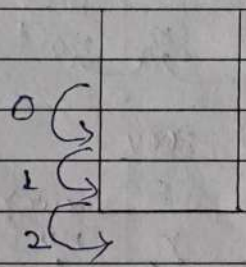
→ 1 → 2 → 3 → 4 → 5 → null ⇒ head = null return -1

if (head.data == key) return 0 search (head.next)

* We search recursively which time complexity is also O(n).



* idx = ~~-1~~ ~~0~~ ~~1~~ 2



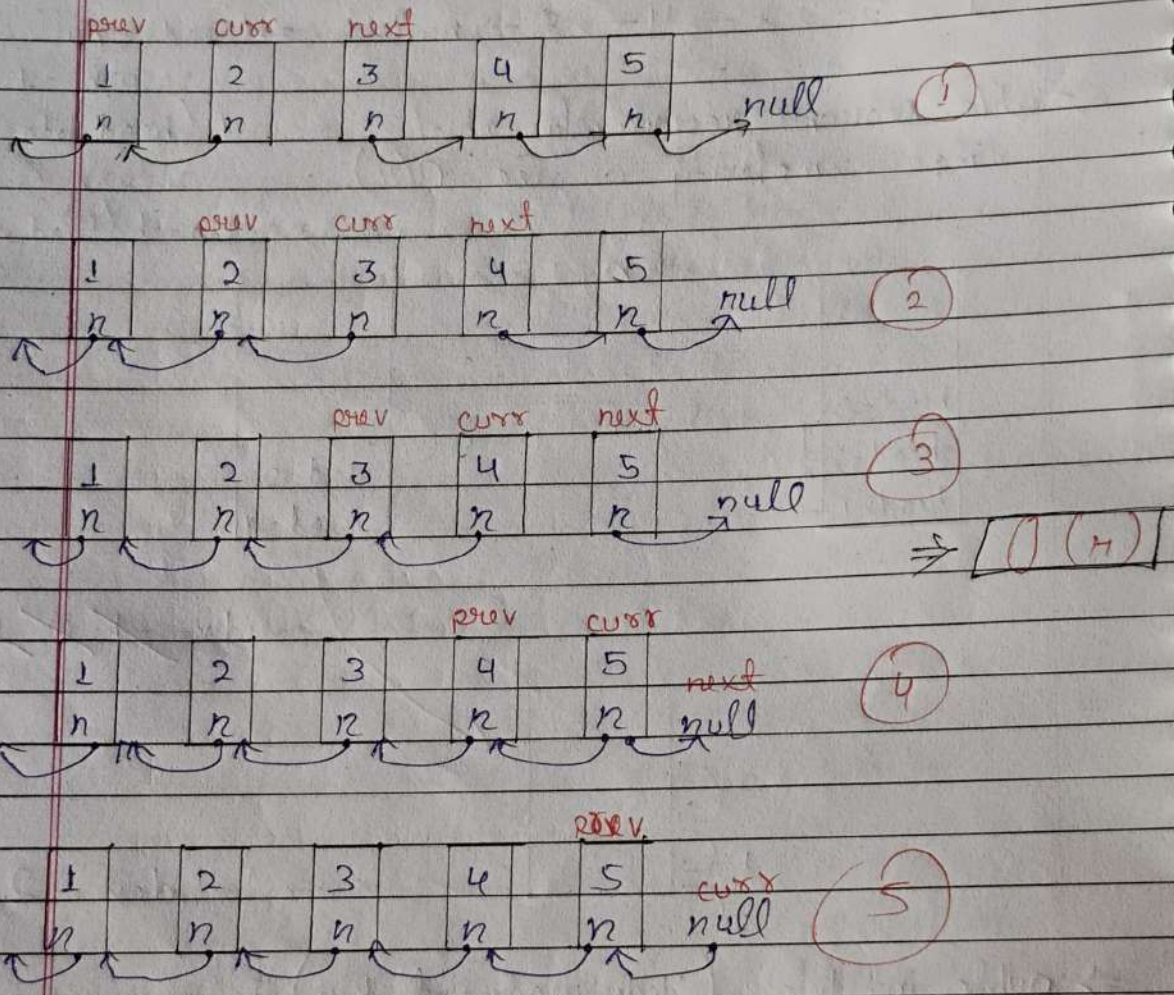
* final index = 2.

⇒
```
public int helper (Node head, int key) {
    if (head == null) {
        return -1;
    }
    if (head.data == key) {
        return 0;
    }
    int idx = helper (head.next, key);
    if (idx == -1)
        return -1;
    return idx + 1;
}
```

⇒
```
public int recSearch (int key) {
    return helper (head, key);
}
```

★ REVERSE A LINKED LIST - Iterative Approach

1 → 2 → 3 → 4 → 5 → null reverse null ← 1 ← 2 ← 3 ← 4 ← 5



⇒ while (curr != null) {

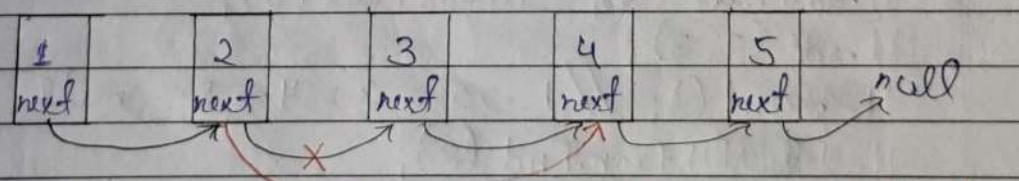
1. next = curr.next;
2. curr.next = prev;
3. prev = curr;
4. curr = next;

}

→ Four imp. steps for reversing LL.

⇒
```
public void reverse() {
    Node prev = null;
    Node curr = tail = head;
    Node next;
    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
}
public static void main (String args[]) {
    LinkedList ll = new LinkedList();
    ll.addFirst(2);
    ll.addFirst(1);
    ll.addLast(4);
    ll.addLast(5);
    ll.add(2,3);
    ll.print(); //1 -> 2 -> 3 -> 4 -> 5 -> null
    ll.reverse();
    ll.print(); //5 -> 4 -> 3 -> 2 -> 1 -> null
}
```

★ FIND & REMOVE Nth NODE FROM END
- Iterative Approach - O(n)



→ nth end
→ (size - n + 1)th start

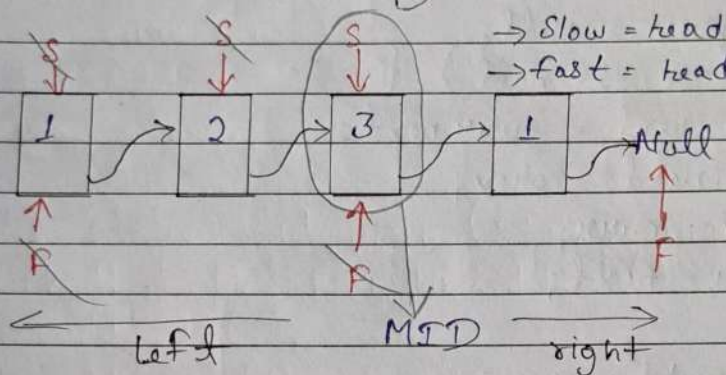
* Garbage Collector plays an vital role.

⇒
```
public void deleteNthFromEnd (int n) {
    //calculate size
    int size=0;
    Node temp = head;
    while (temp != null) {
        temp = temp.next;
        size++;
    }
    if (n == size) {
        head = head.next; //remove first
        return;
    }
    //size - n
    int i = 1;
    int iToFind = size - n;
    Node prev = head;
    while (i< iToFind) {
        prev = prev.next;
        i++;
    }
    prev.next = prev.next.next;
    return;
}
public static void main (String args[]) {
    LinkedList ll = new LinkedList();
    ll.addFirst (2); ll.addFirst(1);
    ll.addLast (4); ll.addLast (5);
    ll.add (2, 3);
    ll.print(); //1 ->2 -> 3-> 4-> 5-> null
    ll.deleteNthfromEnd (3);
    ll.print(); //1 -> 2-> 4 ->5 -> null
}
```

★ CHECK IF LL IS A PALINDROME

* For checking there are 3 imp. points or steps :-
  1. Find midNode
  2. Reverse 2nd half
  3. Check if 1st half == 2nd half
     (Left) (right)

* To find midNode : (1) Slow - Fast Approach



⇒ //Slow - Fast Approach

```
public Node findMid (Node head) { //helper
    Node slow = head;
    Node fast = head;
    While (fast != null && fast.next != null) {
        slow = slow.next ; //+1
        fast = fast.next.next ; //+2
    }
    return slow; //slow is my midNode
}
```

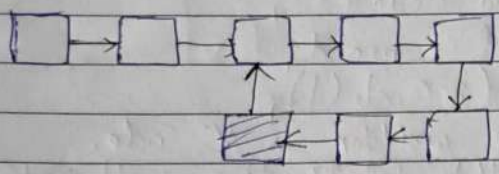
// Now we will perform (1) Find MidNode
(2) Reverse 2nd half
(3) 1st half == 2nd half (check)

```
public boolean checkPalindrome() {
  if (head == null || head.next == null) {
    return true;
  }
  // find middle
  Node mid = findMiddleNode(head);
  // Reverse 2nd half
  Node curr = mid;
  Node prev = null;
  while (curr != null) {
    Node next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  Node right = prev;
  Node left = head;
  // check if equal
  while (right != null) {
    if (left.data != right.data) {
      return false;
    }
    left = left.next;
    right = right.next;
  }
  return true;
}

public static void main (String args[]) {
  LinkedList ll = new LinkedList();
  ll.addFirst(1); ll.addFirst(2); ll.addLast(2); ll.addLast(1);
  ll.print(); // 1 -> 2 -> 2 -> 1
  System.out.println(ll.checkPalindrome());
}
```

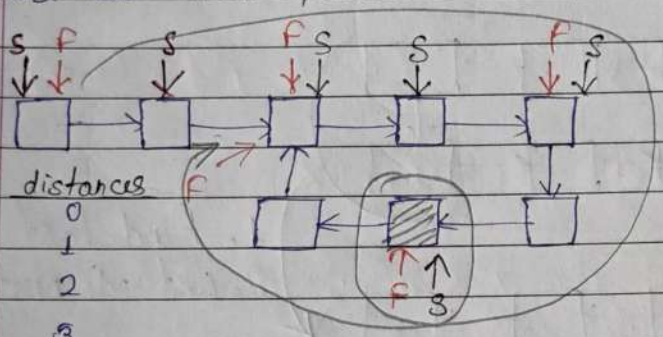
LINKED LIST - II

★ DETECT A LOOP/CYCLE IN A LINKED LIST



* So, to find out Loop or Cycle in a Linked List, we use ⇒ Floyd's Cycle Finding Algorithm

* We use two pointers in this Algorithm, → Slow (+1) → Fast (+2)



moves	distances
0	0
1	1
2	2
3	3

* It will move like this untill slow & fast meet at same node.

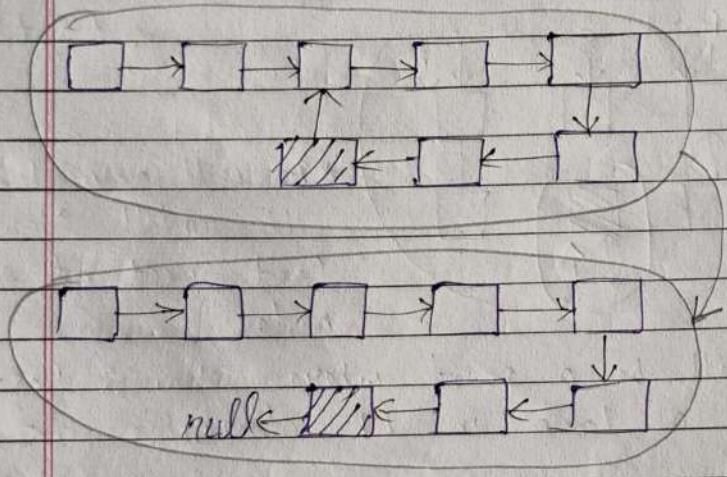
* If there will be a point where Slow & Fast melts, it's a loop in LL otherwise not.

⇒
```
public static boolean isCycle() { //Floyd's CFA
    Node slow = head;
    Node fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next; //+1
        fast = fast.next.next; //+2
        if (slow == fast) {
            return true; //cycle exists
        }
    }
    return false; //cycle doesn't exist
}
```

```
public static void main (String args[]) {
    head = new Node(1);
    head.next = new Node(2);
    head.next.next = new Node(3);
    head.next.next.next = head;
    System.out.println(isCycle());
    ll.print();
}
```

I have written in VS Code. print() before this will not work on loop of LL.

★ REMOVE A LOOP / CYCLE IN A LINKED LIST

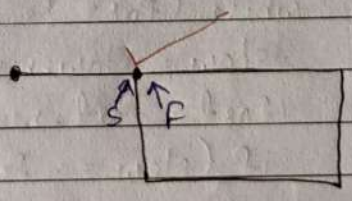


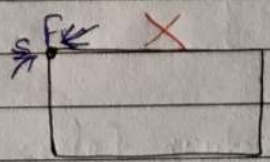
1. Find last Node
2. Last node.next = Null

* Algorithm / Approach

1. Detect cycle

```
slow = head
fast = head
prev = null
while (slow == fast) {
    slow -> +1
    prev = fast
    fast -> +1
}
```





2. prev.next = null

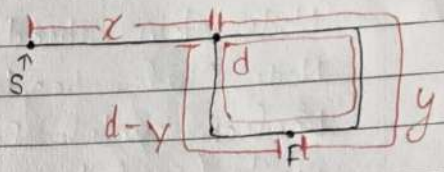
Algorithm is not suitable for where slow & fast both meets at HEAD

⇒
```
public static void removeCycle () {
    //detect cycle
    Node slow = head;
    Node fast = head;
    boolean cycle = false;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (fast == slow) {
            cycle = true;
            break;
        }
    }
    if (cycle == false) {
        return;
    }
    //find meeting point
    slow = head;
    Node prev = null;
    while (slow != fast) {
        prev = fast;
        slow = slow.next;
        fast = fast.next;
    }
    //remove cycle → last.next = null
    prev.next = null;
}

public static void main (String args[]) {
    head = new Node(1); Node temp = new Node(2);
    head.next = temp; head.next.next = new Node(3);
    head.next.next.next = temp; //1 → 2 → 3 → 2
    Syso (isCycle());
    removeCycle ();
    Syso (isCycle());
```

* Mathematical Explanation of removing Loop/Cycle in LL

→ fast = 2 * slow



→ slow distance = $x + a*d + y$

→ fast distance = $x + b*d + y$

$$x + b*d + y = 2*(x + ad + y)$$
$$bd - 2ad = x + y$$
$$x = d(b-2a) - y$$

$$\Rightarrow \boxed{x = kd - y} \qquad \therefore y = (k-1)d + (d-y)$$

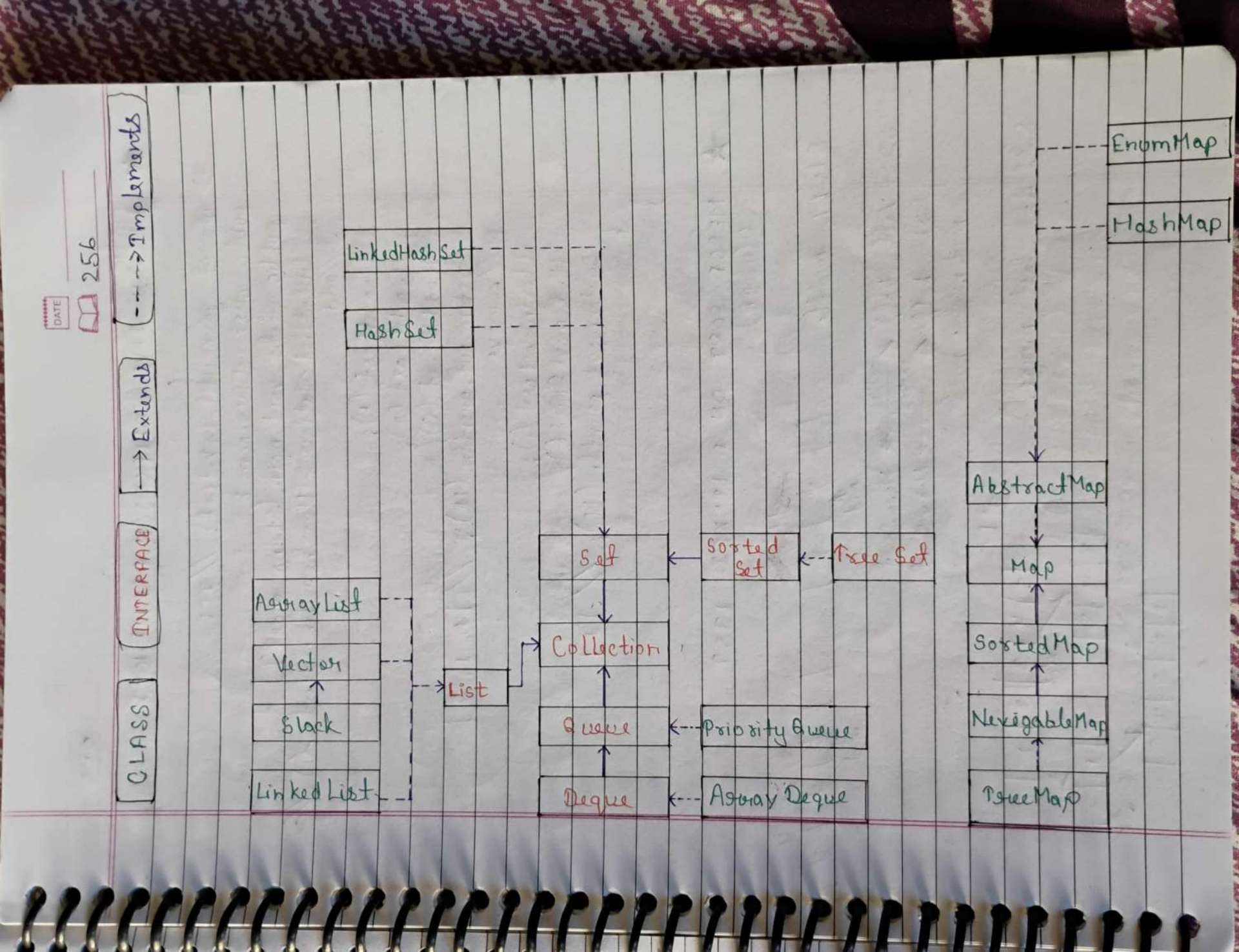
★ JAVA COLLECTIONS FRAMEWORK

* JCF is a set of classes & interfaces that implement commonly reusable collection data structure & algorithms.

* It provides a well-designed set of interfaces and classes that make it easier to work with groups of objects (collections) in a consistent and efficient manner.

• Advantages of JCF

→ Consistency
→ Performance
→ Easy of Use
→ Interoperability


DATE
256
CLASS
INTERFACE
--->Extends
- - - ->Implements
LinkedHashSet
HashSet
EnumMap
HashMap
AbstractMap
Map
SortedMap
NavigableMap
TreeMap
Set
SortedSet
TreeSet
Collection
List
Queue
Deque
PriorityQueue
ArrayDeque
ArrayList
Vector
Stack
LinkedList

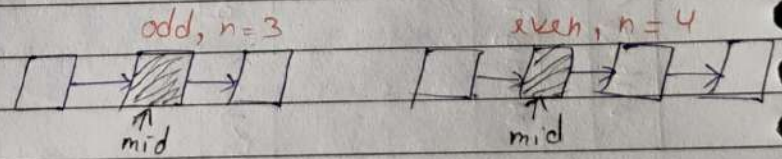
★ LL IN JAVA COLLECTION FRAMEWORK

⇒
```
import java.util.LinkedList; //JCF
public class Classroom {
  public static void main (String args[]) {
    LinkedList<Integer> ll = new LinkedList<>(); //create
    //add
    ll.addLast(1);
    ll.addLast(2);
    ll.addFirst(0);
    System.out.println(ll); // [0, 1, 2]
    //remove
    ll.removeLast();
    ll.removeFirst();
    System.out.println(ll); // [1]
  }
}
```

★ MERGE SORT ON A LINKED LIST

$O(n \log n)$

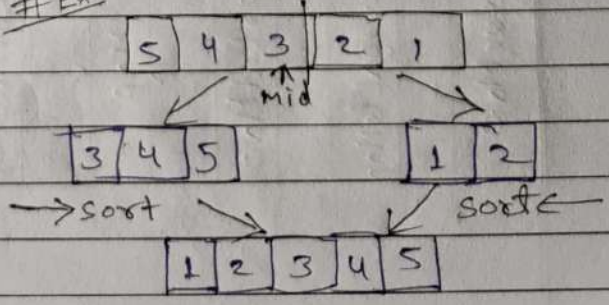
S1: LL middle



S2: Left half ← calling ● MergeSort
Right half ← calling MergeSort
mid.next = null

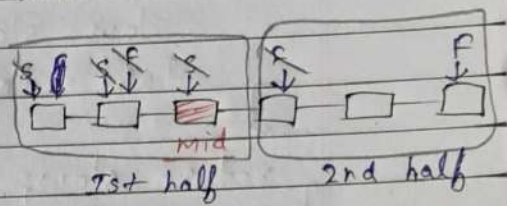
S3: At last merge.

#Ex:



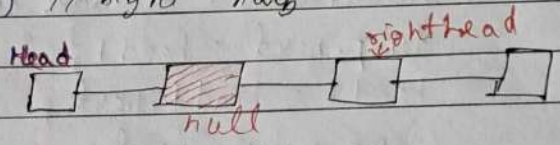
S1: Find mid in LL

slow = head
fast = head.next ← अगर .next नहीं लगाते तो middle node 2nd half में आता।
while (fast != null && fast.next != null)
    slow → +1
    fast → +2
slow → middle node
1st half last node

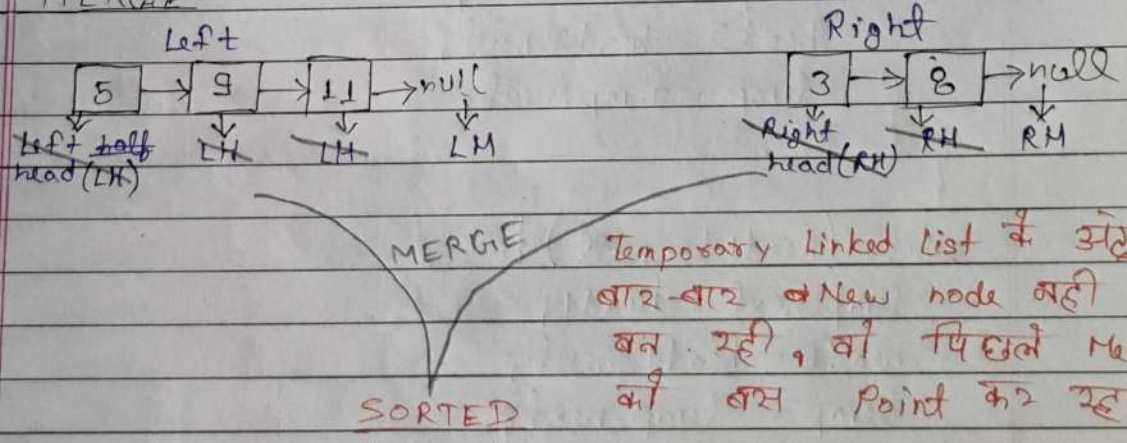


S2: Separate Left & Right & calling MS

rightHead = mid.next
mid.next = null
ms (head) // left half
ms (rightHead) // right half



S3: MERGE

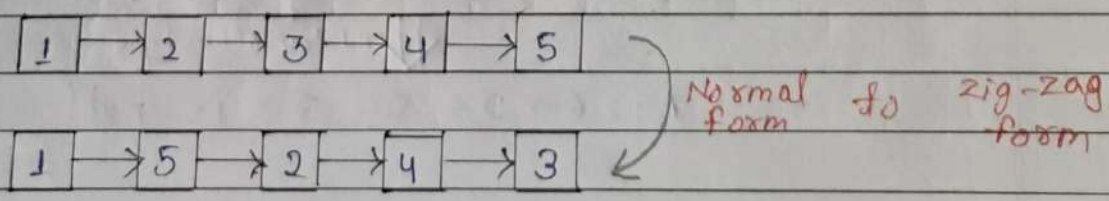


⇒
```
public static
~~public private~~ Node getMid (Node head) {
    Node slow = head;
    Node fast = head.next;
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow; //mid node
}
public static
~~private~~ Node merge (Node head1, Node head2) {
    Node mergedLL = new Node (-1);
    Node temp = mergedLL;
    while (head1 != null && head2 != null) {
        if (head1.data <= head2.data) {
            temp.next = head1;
            head1 = head1.next;
            temp = temp.next;
        } else {
            temp.next = head2;
            head2 = head2.next;
            temp = temp.next;
        }
    }
    while (head1 != null) {
        temp.next = head1;
        head1 = head1.next;
        temp = temp.next;
    }
    while (head2 != null) {
        temp.next = head2;
        head2 = head2.next;
        temp = temp.next;
    }
    return mergedLL.next;
}
```

```
public Node mergeSort (Node head) {
  if (head == null || head.next == null) {
    return head;
  }
  //Find Mid
  Node mid = getMid (head);
  //Node Left & right MS
  Node rightHead = mid.next;
  mid.next = null;
  Node newLeft = mergeSort (head);
  Node newRight = mergeSort (rightHead);
  // merge
  return merge (newLeft, newRight);
}
public static void main (String args[]) {
  LinkedList LL = new LinkedList();
  LL.addFirst (1);
  LL.addFirst (2);
  LL.addFirst (3); LL.addFirst(4); LL.addFirst(5);
  LL.print(); //5 -> 4 -> 3 -> 2 -> 1 -> null
  LL.head=LL.mergeSort (LL.head);
  LL.print(); //1 -> 2 -> 3 -> 4 -> 5 -> null
}
```

★ ZIG-ZAG LINKED LIST

* For a linked list of the form : L(1) → L(2) → L(3) → L(4) : ... L(n-1) → L(n)
convert it into zig-zag form i.e., : L(1) → L(n) → L(2) → L(n-1) → L(3) → L(n-2) ...



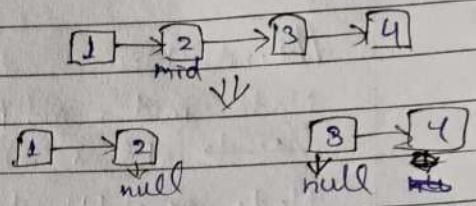
* Approach

S1: Find mid-Node (mid = 1st half last node)

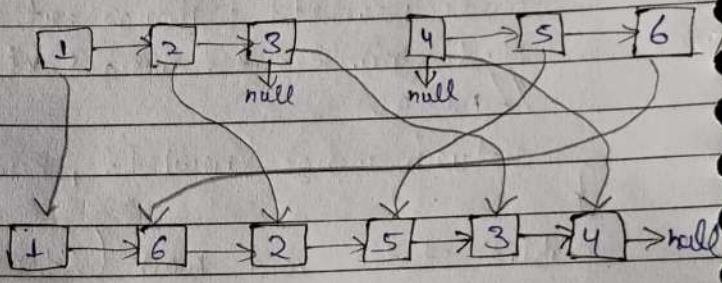
S2: IInd Half reverse

divide

Left right



S3: Let's discuss alternate merging:



⇒ Node LH = 1st half head
Node RH = 2nd half head
Node nextL, nextR
while (LH != null & RH != null) {

```
nextL = LH.next
LH.next = RH
nextR = RH.next
RH.next = nextL
```

} Zigzag

```
RH = nextR
LH = nextL
```

} Update

}

1 → 2 → 3 → 4 → 5 → 6 → null

After all steps.

1 → 6 → 2 → 5 → 3 → 4 → null

⇒
```
public void zigZag() {
  //find mid
  Node slow = head;
  Node fast = head.next;
  while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
  }
  Node mid = slow;
  //reverse 2nd half
  Node curr = mid.next;
  ~~curr.next = prev;~~ mid.next = null;
  ~~prev = curr;~~ Node prev = null;
  ~~curr = next;~~ Node next;
  while (curr != null) {
    next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  Node left = head;
  Node right = prev;
  Node nextL, nextR;
  //alt merge - zig-zag merge
  while (left != null && right != null) {
    nextL = left.next;
    left.next = right;
    nextR = right.next;
    right.next = nextL;
    left = nextL;
    right = nextR;
  }
}
```

```
public static void main (String args[]) {
    LinkedList ll = new LinkedList();
    ll.addLast(1);
    ll.addLast(2);
    ll.addLast(3);
    ll.addLast(4);
    ll.addLast(5);
    ll.addLast(6);
    ll.print(); //1->2->3->4->5->6->null
    zigZag();
    ll.print(); //1->6->2->5->3->4->null
}
```

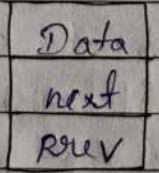
★ DOUBLY LINKED LIST

* In DLL, each node contains three parts :
  1. Data : value or data stored in the node.
  2. Next Pointer : reference to next node in the sequence.
  3. Previous Pointer : reference to previous node in the sequence.

* null ← [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ [Prev | Data | Next] ↔ Null

* A doubly linked list can be traversed in both directions - forward & backward.

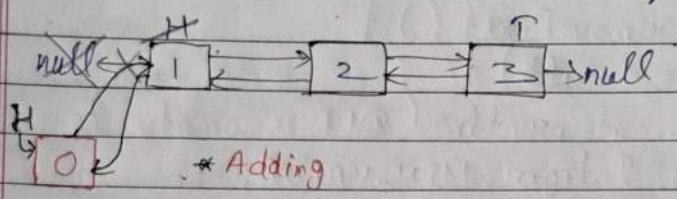
```
DLL ⇒ Node {
        int data;
        Node next;
        Node prev;
    }
```



जब Code करना है तब addfirst last & removeLast यूज में करना है

DATE

* We will write code for DLL, S1. Create node



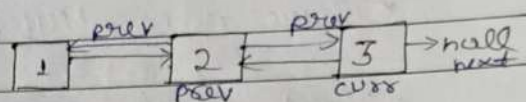
S2. newNode.next = head
head.prev = newNode
head = newNode

```
⇒ public class DoubleLL {
    public class Node {
        int data;
        Node next;
        Node prev;
        public Node (int data) {
            this.data = data;
            this.next = null;
            this.prev = null;
        }
    }
    public static Node head;
    public static Node tail;
    public static int size;
    //add First
    public void addFirst (int data) {
        Node newNode = new Node (data);
        size++;
        if (head == null) {
            head = tail = newNode;
            return;
        }
        newNode.next = head;
        head.prev = newNode;
        head = newNode;
    }
```

```
///removeFirst
public int removeFirst() {
    if (head == null) {
        System.out.println("DLL is empty");
        return Integer.MIN_VALUE;
    }
    if (size == 1) {
        int val = head.data;
        head = tail = null;
        size--;
        return val;
    }
    int val = head.data;
    head = head.next;
    head.prev = null;
    size--;
    return val;
}
//printing
public void print() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + "<->");
        temp = temp.next;
    }
    System.out.println("null");
}
public static void main(String args[]) {
    DoubleLL dll = new DoubleLL();
    dll.addFirst(3); dll.addFirst(2); dll.addFirst(1);
//1<->2<->3<->null dll.print(); Syso(dll.size); //3
    dll.removeFirst(); dll.print(); //2<->3<->null
    System.out.println(dll.size); //2
}
```

★ REVERSE A DLL

It is same as Reverse a LL, only one line we have to add.



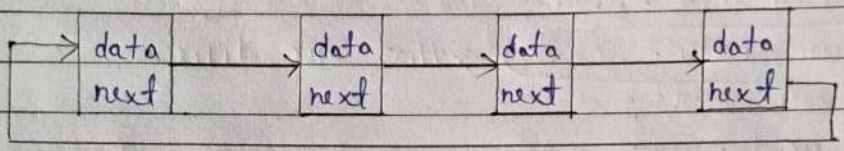
⇒
```
while (curr != null) {
    next = curr.next;
    curr.next = prev;
    curr.prev = next;   → that one line
    prev = curr;
    curr = next;
}
```

```
public void reverse() {
    Node curr = head;
    Node prev = null;
    Node next;
    while (curr != null) {
        next = curr.next;
        curr.next = prev;
        curr.prev = next;
        prev = curr;
        curr = next;
    }
    head = prev;
}
public static void main (String args[]) {
    DoubleLL dll = new DoubleLL();
    dll.addFirst(3); dll.addFirst(2); dll.addFirst(1);
    dll.print(); // 1 <-> 2 <-> 3 <-> null
    dll.reverse();
    dll.print(); // 3 <-> 2 <-> 1 <-> null
}
```

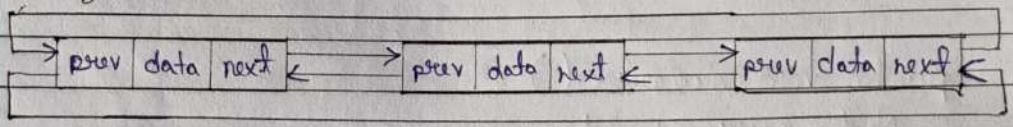
★ CIRCULAR LINKED LIST

* It is a linked list in which all nodes form a circle.
* There is no NULL at the end.

* Singly Circular Linked List



* Doubly Circular Linked List



⇒
```
public class Circular {
  public static class Node {
    int data;
    Node next;
  }
  public static Node addToEmpty (Node last, int data) {
    if (last != null) {
      return last;
    }
    Node newNode = new Node();
    newNode.data = data;
    last = newNode;
    newNode.next = last
    return last;
}
```

P.T.O

```
public static Node addFront(Node last, int data){
  if (last == null){
    return addToEmpty(last, data);
  }
  Node newNode = new Node();
  newNode.data = data;
  newNode.next = last.next;
  last.next = newNode;
  return last;
}
public static Node addEnd(Node last, int data){
  if (last == null){
    return addToEmpty(last, data);
  }
  Node newNode = new Node();
  newNode.data = data;
  newNode.next = last.next;
  last.next = newNode;
  last = newNode;
  return last;
}
public static Node addAfter(Node last, int data, int item){
  if (last == null){
    return null;
  }
  Node newNode, p;
  p = last.next;
  do{
    if (p.data == item){
      newNode = new Node();
      newNode.data = data;
      newNode.next = p.next;
      p.next = newNode;
```

```
        if (p == last) {
            last = newNode;
            return last;
        }
        p = p.next;
    } while (p != last.next);
    System.out.println(item + "Given node is not Present");
    return last;
}
public static Node deleteNode(Node last, int key) {
    if (last == null) {
        return null;
    }
    if (last.data == key && last.next == last) {
        last = null;
        return last;
    }
    Node temp = last, d = new Node();
    if (last.data == key) {
        while (temp.next != last) {
            temp = temp.next;
        }
        temp.next = last.next;
        last = temp.next;
    }
    while (temp.next != last && temp.next.data != key) {
        temp = temp.next;
    }
    if (temp.next.data == key) {
        d = temp.next;
        temp.next = d.next;
    }
    return last;
}
```

```
public static void traverse (Node last) {
    Node p;
    if (last == null) {
        System.out.println ("List is empty");
        return;
    }
    p = last.next;
    do {
        System.out.print (p.data + " ");
        p = p.next;
    } while (p != last.next);
}
public static void main (String args []) {
    Node last = null;
    last = addToEmpty (last, 6);
    last = addEnd (last, 8);
    last = addFront (last, 2);
    last = addAfter (last, 10, 2);
    traverse (last);
    deleteNode (last, 8);
    traverse (last);
}
}
```

QUES 1. Merge K Sorted Lists – We have K sorted linked lists of size N each, merge them & print the sorted output.

Sample Input: K = 2, n = 2

l_1 = 1 → 3 → Null

l_2 = 6 → 8 → NULL

l_3 = 9 → 10 → Null

Sample Output: 1 → 3 → 6 → 8 → 9 → 10 → Null

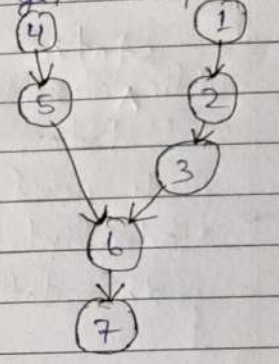
Sol →

```
public static Node SortedMerge (Node a, Node b) {
  Node result = null;
  if (a==null) {
    return b;                    // Time Complexity, => O(nlogk)
  }                              // Space Complexity, => O(n)
  else if (b==null) {
    return a;
  }
  if (a.data <= b.data) {
    result = a;
    result.next = SortedMerge (a.next, b);
  }
  else {
    result = b;
    result.next = SortedMerge (a, b.next);
  }
  return result;
}

public static Node mergeKLists (Node arr[], int last) {
  while (last != 0) {
    int i=0, j = last;
    while (i<j) {
      arr[i] = SortedMerge (arr[i], arr[j]);
      i++;
      j--;
      if (i>=j) {
        last = j;
      }
    }
  }
  return arr[0];
}
```

```
public static void printList (Node node) {
  while (node != null) {
    System.out.print (node.data + " ");
    node = node.next;
  }
}
  static
public class Node {
  int data;
  Node next;
  Node (int data) {
    this.data = data;
  }
}
public static void main (String args[]) {
  int k = 3;
  int n = 4;
  Node arr[] = new Node[k];
  arr[0] = new Node (1);
  arr[0].next = new Node (3);
  arr[0].next.next = new Node (5);
  arr[0].next.next.next = new Node (7);
  arr[1] = new Node (2);
  arr[1].next = new Node (4);
  arr[1].next.next = new Node (6);
  arr[1].next.next.next = new Node (8);
  arr[2] = new Node (0);
  arr[2].next = new Node (10);
  arr[2].next.next = new Node (11);
  arr[2].next.next.next = new Node (12);
  Node head = mergeKLists (arr, k-1);
  printList (head);
}
```

QUES 2: Intersection of 2 Linked lists - In a system there are two singly linked lists. By some programming error, the end node of one of the linked lists got linked to the second list, forming an inverted Y-shaped list. Write a program to get the point where two linked lists merge.



* We have to find the intersection part in this system.

Sol →

```
static
public class Node {
    int data;
    Node next;          // TC, ⇒ O(m*n)
    Node (int d) {      // SC, ⇒ O(1)
        data = d;
        next = null;
    }
}

public Node getIntersectionNode (Node head1, Node head2){
    while (head2 != null) {
        Node temp = head1;
        while (temp != null) {
            if (temp == head2) {
                return head2;
            }
            temp = temp.next;
        }
        head2 = head2.next;
    }
    return null;
}
```

```
public static void main (String args []) {
  Solution lst = new Solution();
  Node head1, head2;
  head1 = new Node (10);
  head2 = new Node (3);
  Node newNode = new Node (6);
  head2.next = newNode;
  newNode = new Node (9);
  head2.next.next = newNode;
  newNode = new Node (15);
  head1.next = newNode;
  head2.next.next.next = newNode;
  newNode = new Node (30);
  head1.next.next = newNode;
  head1.next.next.next = null;
  Node intersectionPoint = lst.getIntersectionNode (head1, head2);
  if (intersectionPoint == null) {
    System.out.print ("No Intersection Point \n");
  }
  else {
    System.out.print ("Intersection Point : "+intersectionPoint
                      .data);
  }
}
```

QUES 3. Swapping Nodes in a Linked List - We have a linked list & two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields. It may be assumed that all keys in the linked list are distinct.

Sample Input: 1→2→3→4, x=2, y=4

Sample Output: 1→4→3→2.

Sol ⇒

```
class Node {
    int data;
    Node next;
    Node (int d) {                    // TC-O(n)
        data = d;                     // SC-O(1)
        next = null;
    }
}
Node head;
public void swapNodes (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;
}
Node temp = currX.next;
currX.next = currY.next;
currY.next = temp;
}
public void push (int new_data) {
    Node new_Node = new Node (new_data);
    new_Node.next = head;
    head = new_Node;
}
printpublic void printList () {
    Node tNode = head;
    while (tNode != null) {
        System.out.print (tNode.data + " ");
        tNode = tNode.next;
    }
}
public static void main (String args []) {
    Solution list = new Solution ();
    list.push (7); list.push (6); list.push (5); list.push (4);
    list.push (3); list.push (2); list.push (1);
    System.out.println ("Linked list Before");
    list.printList ();
    list.swapNodes (4, 3);
    System.out.println ("Linked list after");
    list.printList ();
}
```

QUES 4. Delete N Nodes After M Nodes of a Linked List - We have a linked list & two integers M & N. Traverse the linked list such that you retain M nodes then ~~del~~ delete next N nodes, continue the same till end of the linked list.

* Sample Input 1: M = 2, N = 2, LL: 1→2→3→4→5→6→7→8
* Sample Output 1: 1→2→5→6

* Sample Input 2: M = 3, N = 2, LL: 1→2→3→4→5→6→7→8→9→10
* Sample Output 2: 1→2→3→6→7→8

Sol →

```
import java.util.*;
class Solution {
    public static class Node {          //Time Complexity, => O(n)
        int data;                       //Space Complexity. => O(1)
        Node next;
    }
    public static Node push (Node head_ref, int new_data) {
        Node new_node = new Node();
        new_node.data = new_data;
        new_node.next = (head_ref);
        (head_ref) = new_node;
        return head_ref;
    }
    public static void printList (Node head) {
        Node temp = head;
        while (temp != null) {
            System.out.println (temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
```

```
public static void skipMdeleteN (Node head, int M, int N) {
  Node curr = head, t;
  int count;
  while (curr != null) {
    for (count = 1; count < M && curr != null; count++) {
      curr = curr.next;
    }
    if (curr == null) {
      return;
    }
    t = curr.next;
    for (count = 1; count <= N && t != null; count++) {
      Node temp = t;
      t = t.next;
    }
    curr.next = t;
    curr = t;
  }
}
public static void main (String args[]) {
  Node head = null;
  int M = 2, N = 3;
  head = push (head, 10);
  head = push (head, 9);
  head = push (head, 8);
  head = push (head, 7);
  head = push (head, 6);
  head = push (head, 5);
  head = push (head, 4);
  head = push (head, 3);
  head = push (head, 2);
  head = push (head, 1);
```

```
        System.out.println();
        printList(head);
        skipMdeleteN(head, M, N);
        System.out.println("Linked list on deletion is: ");
        printList(head);
    }
}
```

QUES 5. Odd Even Linked List - We have a Linked List of integers, write a function to modify the Linked List such that all even numbers appear before all the odd numbers in the modified Linked list. Also, keep the order of even & odd numbers same.

Sample Input 1: 8 → 12 → 10 → 5 → 4 → 1 → 6 → NULL
Sample Output 1: 8 → 12 → 10 → 4 → 6 → 5 → 1 → NULL

Sample Input 2: 1 → 3 → 5 → 7 → NULL
Sample Output 2: 1 → 3 → 5 → 7 → NULL

Sol →

```
class Solution {
    Node head;
    class Node {
        Node int data;
        Node next;
        Node (int d) {
            data = d;
            next = null;
        }
    }
    // Time Complexity, ⇒ O(n)
    // Space Complexity, ⇒ O(1)
```

```
public void segregateEvenOdd() {
    Node end = head;
    Node prev = null;
    Node curr = head;
    while (end.next != null) {
        end = end.next;
    }
    Node new_end = end;
    while (curr.data % 2 != 0 && curr != end) {
        new_end.next = curr;
        curr = curr.next;
        new_end.next.next = null;
        new_end = new_end.next;
    }
    if (curr.data % 2 == 0) {
        head = curr;
        while (curr != end) {
            if (curr.data % 2 == 0) {
                prev = curr;
                curr = curr.next;
            } else {
                prev.next = curr.next;
                curr.next = null;
                new_end.next = curr;
                new_end = curr;
                curr = prev.next;
            }
        }
    } else prev = curr;
    if (new_end != end && end.data % 2 != 0) {
        prev.next = end.next;
        end.next = null;
        new_end.next = end;
    }
}
```

```
public void push (int new_data) {
    Node new_node = new Node (new_data);
    new_node.next = head;
    head = new_node;
}
public void printList () {
    Node temp = head;
    while (temp != null) {
        System.out.print (temp.data + " ");
        temp = temp.next;
    }
    System.out.println ();
}
public static void main (String args []) {
    Solution llist = new Solution ();
    llist.push (11);
    llist.push (10);
    llist.push (8);
    llist.push (6);
    llist.push (4);
    llist.push (2);
    llist.push (0);
    System.out.println ("Linked List");
    llist.printList ();
    llist.segregateEvenOdd ();
    System.out.println ("Updated Linked List");
    llist.printList ();
}
}
```