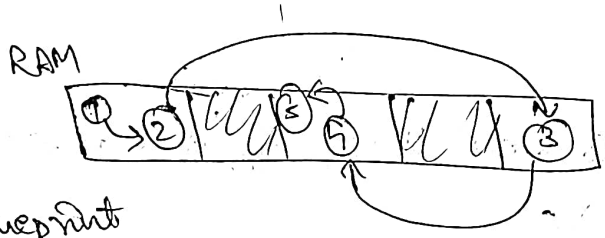
Linked List

* int[] arr = new int[5]; // fixed
* ArrayList<Integer> list = new AL<>(); // Dynamically growing
* Stack → LIFO
* Queue → FIFO

→ Maintains Discipline

→ Continuous space allocation

Linked List : If continuous space not available, then linked list to occupy space from random (fragmented) locations maintaing continuous flow (linked)



Blueprint

Syntax (Node):

```
public static class Node{
    int val;
    Node next; // Reference to next node (linked)
}
```

Syntax Blueprint (Linked List):

```
Node head;
Node tail;
int size;
```

Add Last in Linked List

1. Create new node (nn)
2. if head == null => head = nn, tail = nn
3. else tail.next = nn, tail = nn
4. increase size

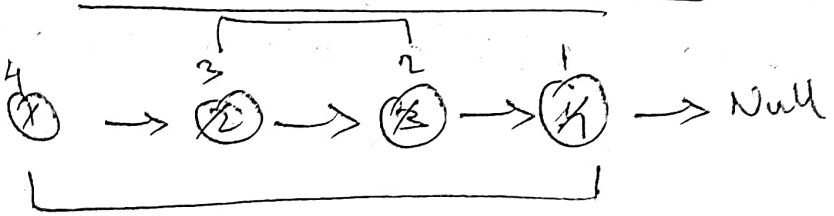
Add At index in Linked List

1. If idx == 0 => Add first
2. If idx == size => Add Last
3. Else => Loop till required position (L-1 target)
   - Create new node and add accordingly. (Make left & right connection).
4. Increment size

Remove At index in Linked List

- If idx == 0 => remove First
- If idx == size - 1 => remove Last
- Else → Loop till required position (before target)
  - curr node.next = curr.next.next
  - Decrement size

Reverse LinkedList ~~Pointers~~ Data Iterative



TC : $O(n^2)$

* Swap 1st node data by traveling till last node.
* Same like reversing an array, just that you have to travel till end destination node.

Reverse Linked List pointer Iterative

TC: O(n)

* Take two pointers prev & curr
* Here initial curr pointer will be tail.
* Travel in list & make curr to point to prev
  - Update prev to curr
  - Store cur.next, then point cur to cur.next
  - Process repeated until cur != null

At end, your prev pointer is head

Kth node from end of Linked List (0-based indexing)

TC: O(n)

* Take two variables t1, t2 (temp)
  - points to head.
* Traverse t2 k steps ahead to maintain gap of k between t1 & t2.
* Then move both t1 & t2 each step until t2 != null.

Mid of Linked List

* Take two variables slow, fast
* slow travels 'x' times, fast travels '2x' times

TC: O(n)

Condition: while (fast.next != null && fast.next.next != null)

(or) while(fast != tail && fast.next != tail)

Merge two sorted LinkedLists

* Two variables t1 → List1.head
  t2 → List2.head
* Iterate by comparing t1 & t2 data, then add to new LinkedList. while (t1 & t2 != null).
* Finally, either of t1 or t2 is not empty, iterate & add to new List.

MergeSort A LinkedList

TC: O(nlogn)

* The calls & work are same like mergesort an array.
* Find middle
* sort left part (head, middle) & ~~second~~ right part
* Sort right part (middle.next, tail)
* Merge these two sorted Lists & return.

Condition: If (head == tail)

↳ Return new list with head.data

Remove Duplicates in a Sorted Linked List

Approach 1:

* Create new List
* Remove first from original list
  - If newlist.tail.data != org.list.data (removed)
    - Add to new list
  - Else traverse

TC : O(n)
SC : O(1)

"this" keyword is readonly.

Since we're always removing & adding, which doesn't exceed org.list size.

Instead,

```
newlist.head = orglist.head
"  .tail = "  .tail
"  .size = "  .size
```

Approach 2:

* Two variables t_1, t_2 pointing head.
  $t_1 \rightarrow$ head ; $t_2 \rightarrow$ head.next
* ~~while~~ if (t_1.data == t_2.data)
  - ~~move~~ $t_2 = t_2$.next;
  - duplicatecount++;

else (t_1.next == t_2)

$t_1 = t_2$;
$t_1 = t_1$.next;

TC : O(n)
SC : O(1)

Process Condition: while (t_2 != null)

End condition: t_1.next = null

Approach 2 :

* t1 -> head
  t2 -> head.next
  d = 0 ; //duplicate count ;

TC : O(n)
SC : O(1)

* If t1.data == t2.data
  ↳ t2 = t2.next //keep moving t2 since //duplicates & count //duplicates
  ↳ d++
  Else if t1.data != t2.data
  ↳ t1.next = t2
  t1 = t2
  t2 = t2.next

* while (t2 != null) (process until)
* Finally t1.next = null
  org.list.size -= d; //Reduce duplicate //count in //original list size

---

K Reverse in Linked List

Approach 1

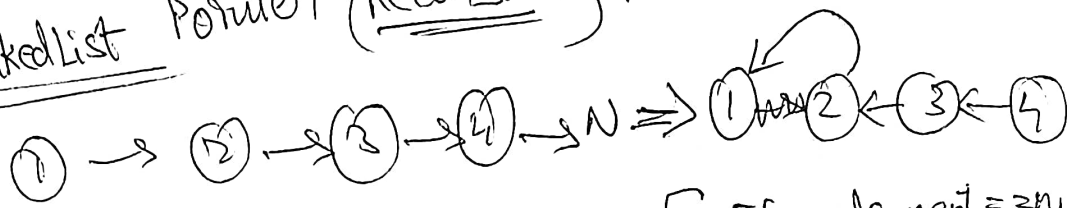
* Two Lists L1, L2 (Final answer will be in L2)
* while (this.size >= k)

TC : O(n)
SC : O(1)

↳ for (1 to k)
  ↳ remove First in this & add First in L1
↳ if (L2.size == 0) //when encountering for L2 the first time
  ↳ L2 = L1
↳ else
  ↳ Adjust L2 (tail, size) to make connection
  Make L1 point new reference (new List)

Integrate final ans L2 in this.
(head, tail, size) (head, tail, size)

Reverse LinkedList Pointer (Recursive):



Faith: `node.next.next = node`

[∵ If node.next = = null
↳ No work]

At end * swap head & tail.
* tail.next = null

Reverse Linked List (Data Recursive)

Initial call (head, head, 0)

call(left, right.next, count+1)

Prblm: left wipes out after call, even you alter left = left.next - though

So create, Left in heap

↳ Then change left.

`left = left.next`

This reflects in change of left which is correct.

Or you can return left.next & catch as return value left from call.

Is Linked List Palindrome

Recursive Same like reverse List (Data Recursive)

↳ Here comparing only till half of LinkedList doesn't matter.

↳ At any point, if left.data != right.data
↳ return false

TC: O(n)
SC: O(n) ↳ stack space

Iterative:

TC : O(n)
SC : O(1)

① * Find mid
② Reverse right part (after mid)
③ Traverse & Compare both left & right parts

Another space complex approach

org. list: 1 → 2 → 3 (given org. list)

(add Last) L1: 1 → 2 → 3 (List 1)

(add First) L2: 3 ← 2 ← 1 (List 2)

Not a palindrome

Intersection Point Of Linked list

① Diff = List 1.size − List 2.size

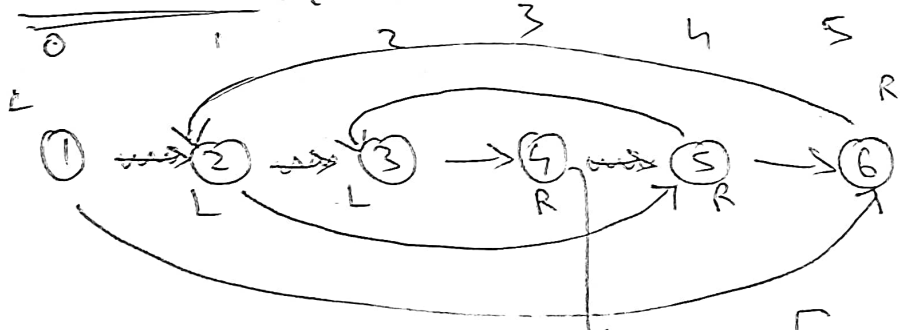
② Travel further in longest list 'diff' steps ahead. Why? ⇒ Both l1 & l2 contains (l1.size − diff) elements a common nodes (in both lists)

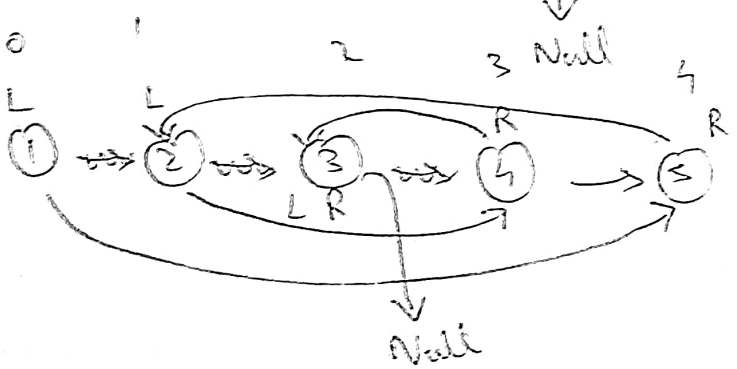
TC : O(n)
SC : O(1)

③ Then, on further travel l1 node == l2's node
↳ return data

Note: This question can't be solved just depending on data value

Fold A Linked List





∵ Pointing right to null in both cases

Approach: Similar to Reverse LinkedList (Data Recursive)

* Maintain Left, Right
* Incr count @ each recursion.

if (count > List.size/2)
- nextLeft = left.next // Preserve upcoming Left
- left.next = right.next
- right.next = nextLeft
- left = nextLeft // Update left

Else if (count == List.size/2)
- right.next = null
- tail = right

Odd Even LinkedList

Maintain

* Two Lists odd, even
  - ① Travel org.list, Remove first
    - If node.data → odd
      - AddLast in odd
    - Else Add Last in even
  - ② Manage head and tail
    - Check for odd, even list empty cases

TC: O(n)
SC: O(1)
(since we remove & add)

LinkedList to StackAdapter

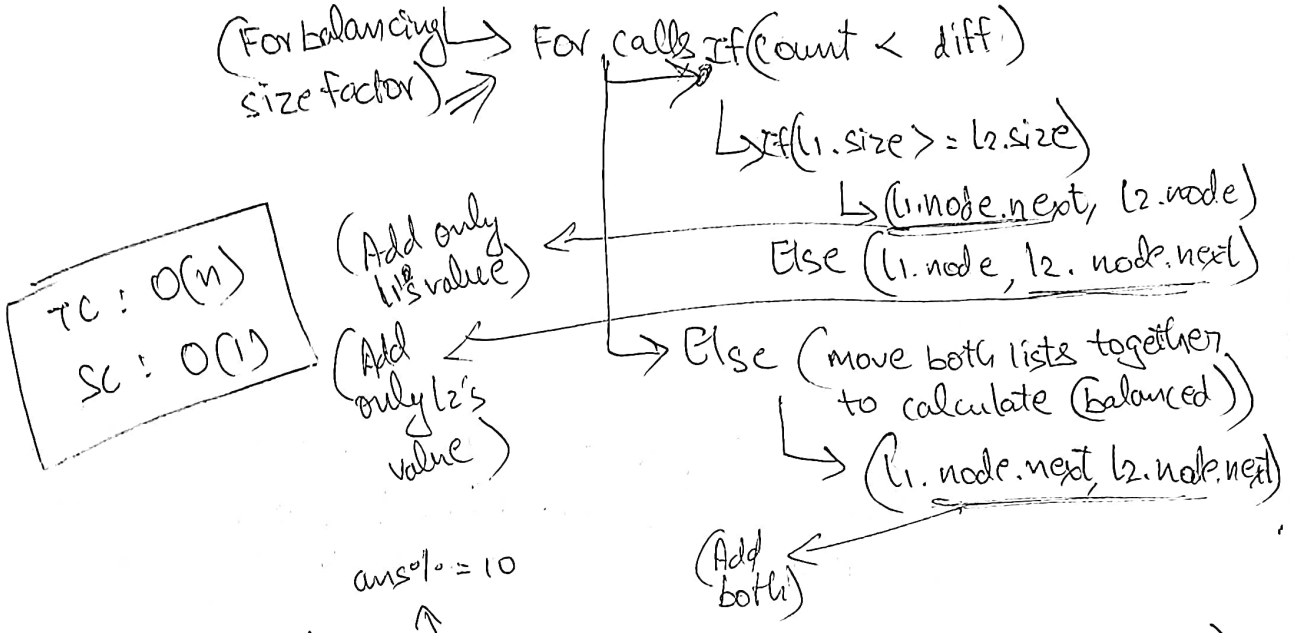
push → addLast()
pop → removeLast()
peek → getLast()

LinkedList to Queue Adapter

add → addLast()
remove → removeFirst()
peek → getFirst()

Add Two Linked Lists

Recursive ① Find diff (list1.size - list2.size)
↳ Incr count @ every recursion level
↳ In recursion



ans%10 = 10

Find remainder from sum (to form node of list)
↳ return carry to find next set of sum.
↓
ans /= 10;