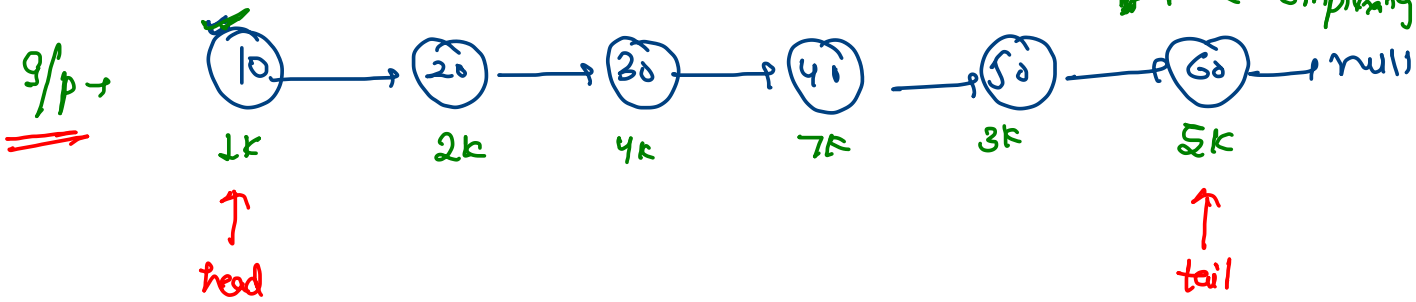
Reverse in linked list (Data Iterative)

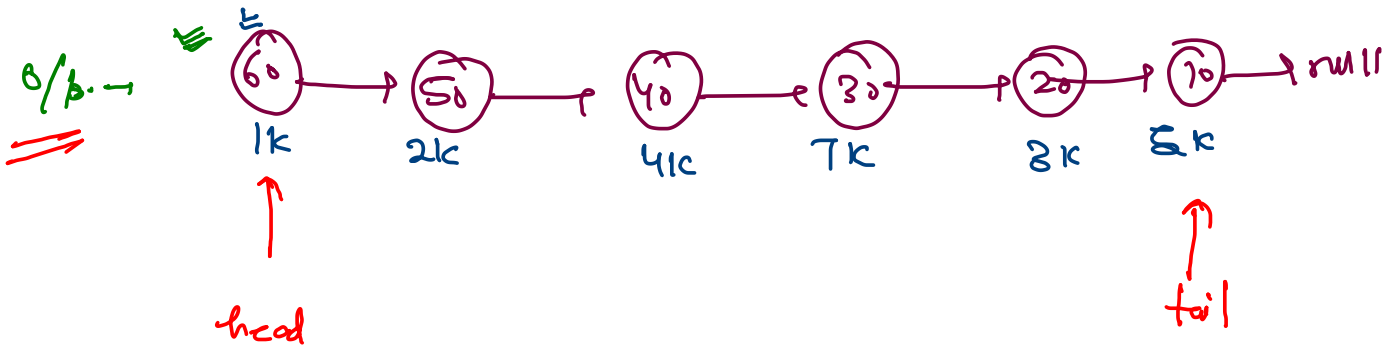
change in data, not in pointer

→ allowed

→ space complexity → O(n) // - Time : O(n) ✓

complexity → $O(n^2)$ ✗





linked list

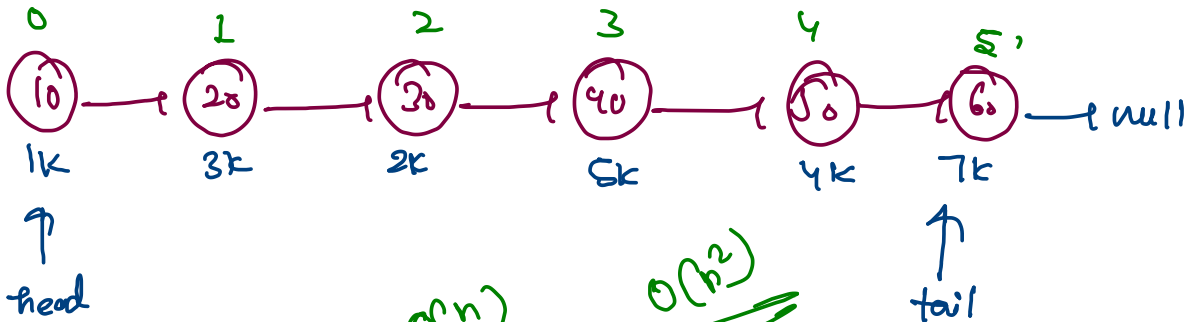
→ Recursion ✗ Stack

→ arrays ✗ "

max. size of linked list ↳ depend on memory.

max of recursive stack → 10^4

arrays → 10^5



left = 0

right = size - 1

O(n)

reverse array

O(n²)

lnode = 1k // getting Nth node → O(n)

rnode = 7k // → getting Nth node → O(n)

Swap data.

left ++;

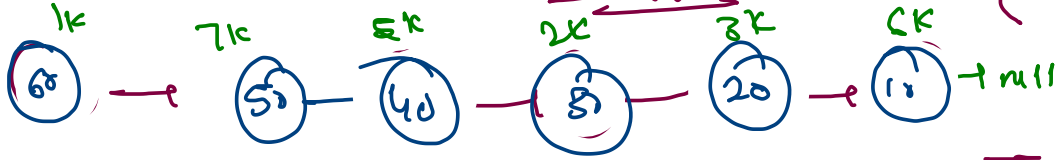
right --;

private functions

2n

Arrays / Recursion

Travel and change-Strategy



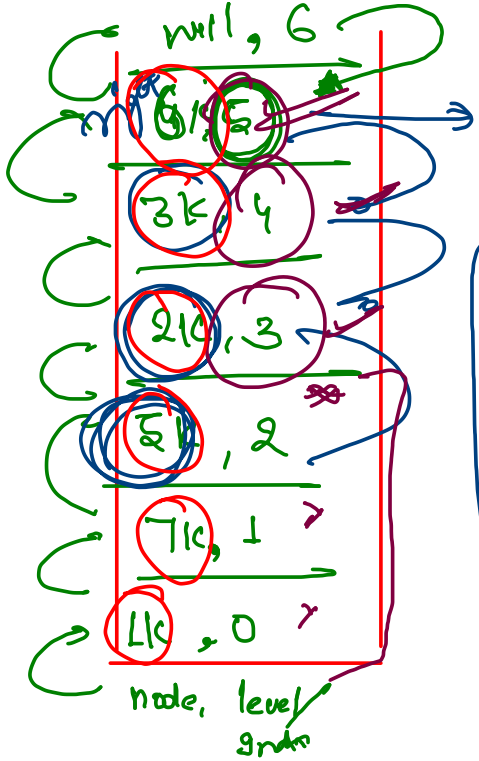
left = 1K, 5K, 2K

size = 6

Time Complexity → O(n)

Space complexity - O(n)

Recursive Space



Post Area

level ≥ Size

global variable

→ swap data, from left

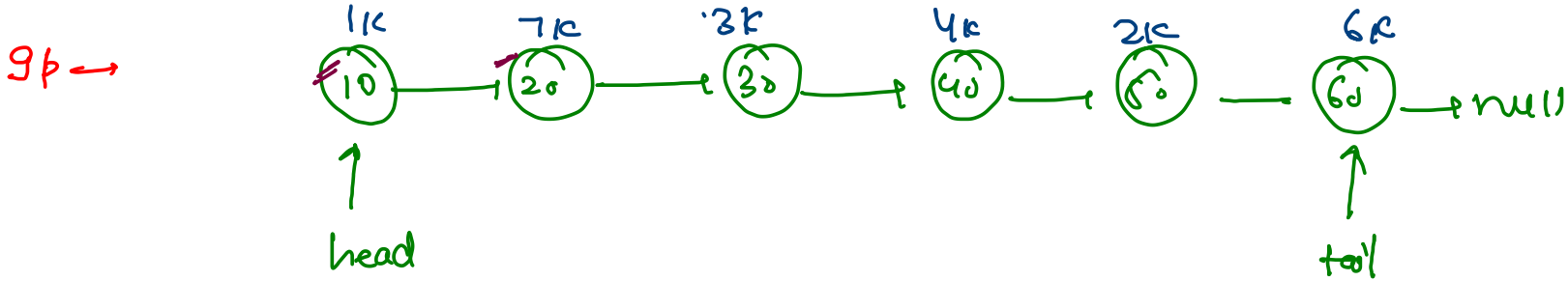
→ left = left.next;

node

stack variable

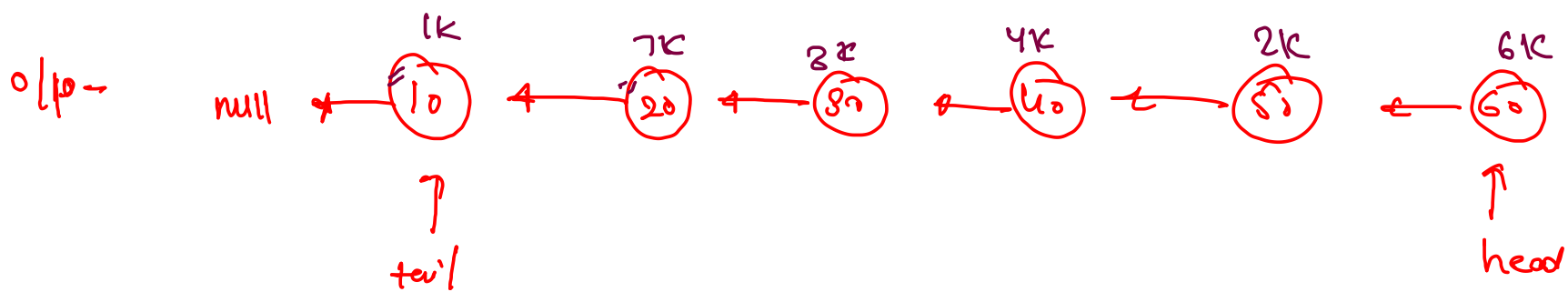
Reverse data pointer

node ✗] change in connection / pointer.



space → O(1)

time - O(n)

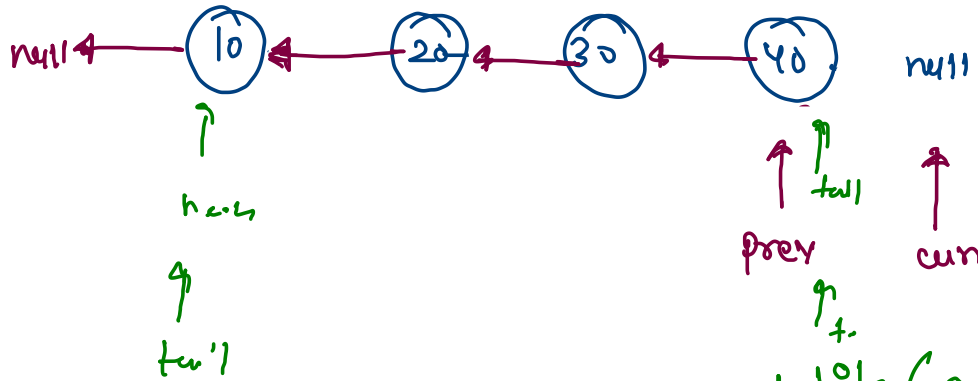


head = 1K ~~6K~~

tail = ~~6K~~ 1K

temp = 1K

h



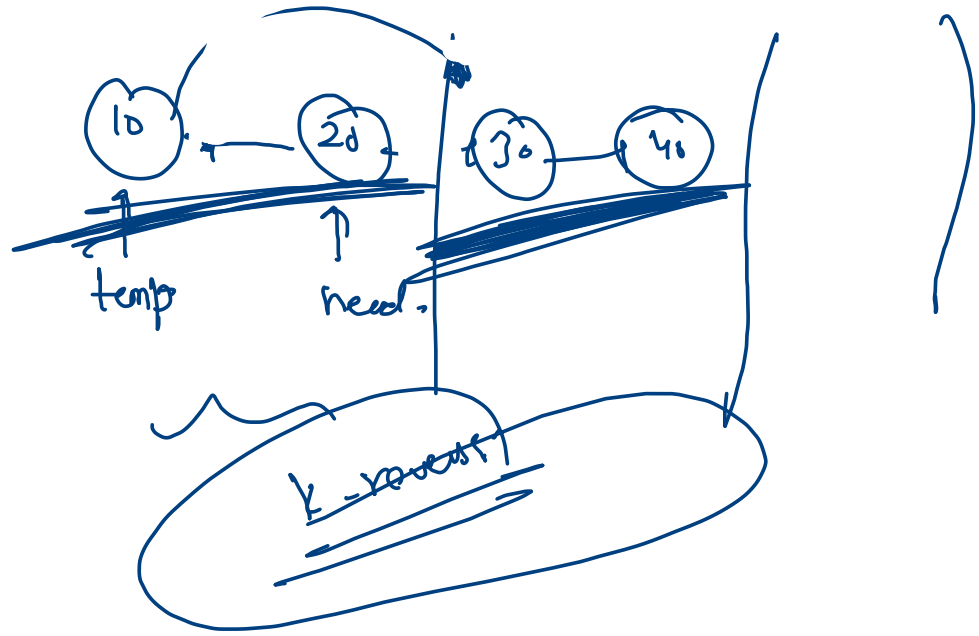
In loop.

```
while (curr != null) {
    Node next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
}
```

Pointer Reverse,

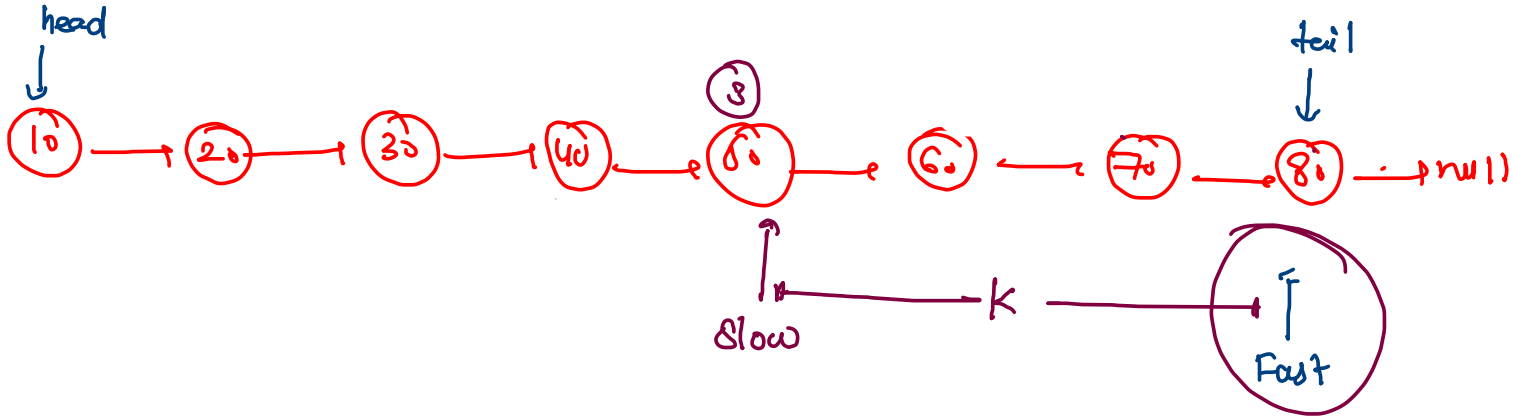
head & tail pointers → swap head, tail Reference

```
Node temp= null;

while(          ) {

    temp= head;
    head= temp.next;
    temp.next= head.next;

    head.next= temp;

}
```



k^{th} Node from End of LinkedList:

k=2



slow = head

fast = head

(1) move fast k steps → k no. of jumps.

(2) make a slow, which is pointing at head.

(3) Move slow & fast. Simultaneously.

fast.next ≠ null

head = null

head.next = null

head.next.next = null

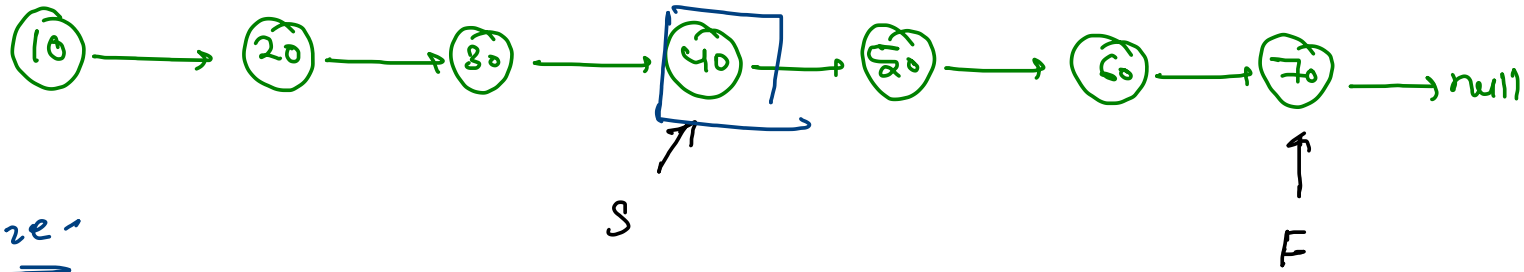
Size - Even

Size - odd

slow ?? → k^{th} Node from End.

Mid of LinkedList :

odd - no. of node



odd - size -

slow = slow.next; = x

fast = fast.next.next; = 2x

condition (1) fast.next == null

Next Page →

slow = head speed of slow = x time = T d = ? $D_2 = xT$ — (1)

fast = head speed of fast = 2x time = T. d = ? $D_1 = 2xT$. — (2)

Speed = dist / Time.

distance = 2x * T

distance = 2xT

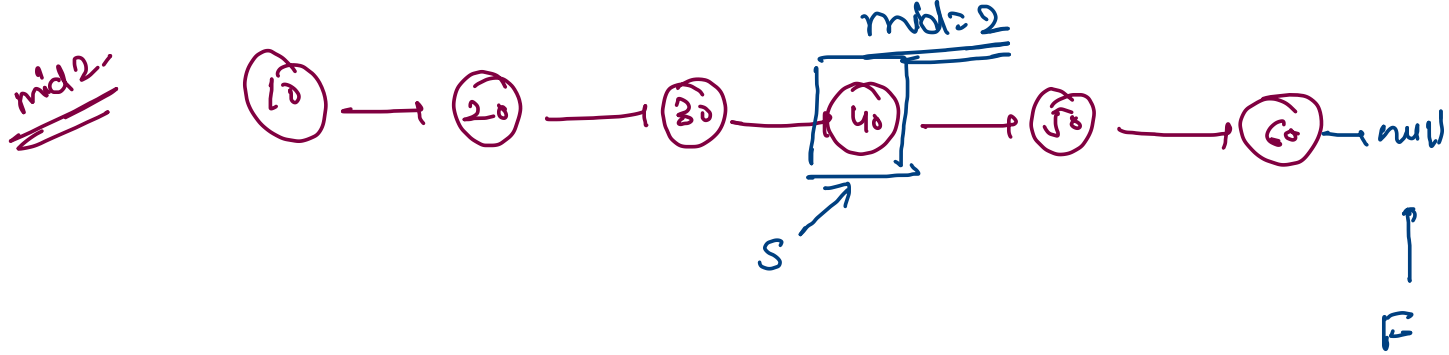
$d_2 = x * T$.

$d_2 = xT$

from eq (1) and (2)

$D_1 = 2D_2$

fast is at Null then slow is at middle of LinkedList.



odd || mid

Even || mid2

```
slow = head;
fast = head.
```

condition ② fast == null

```
while( fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
```

Mid:-

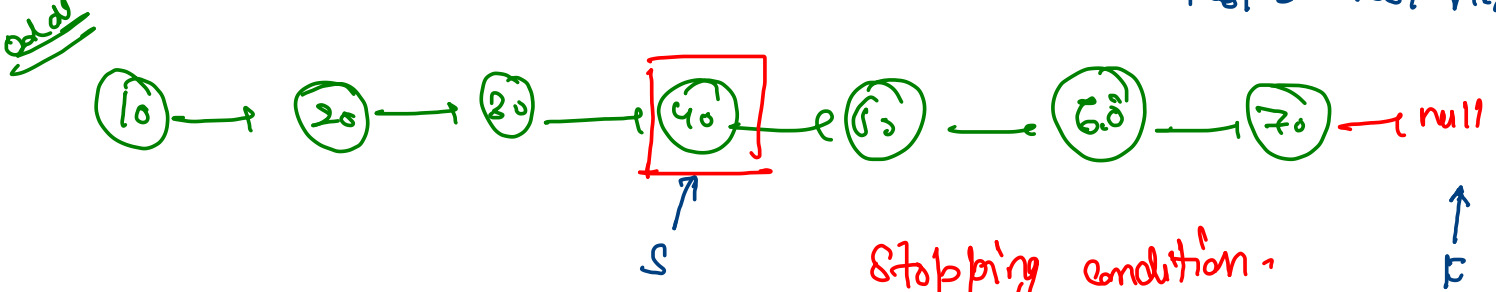
slow = head;

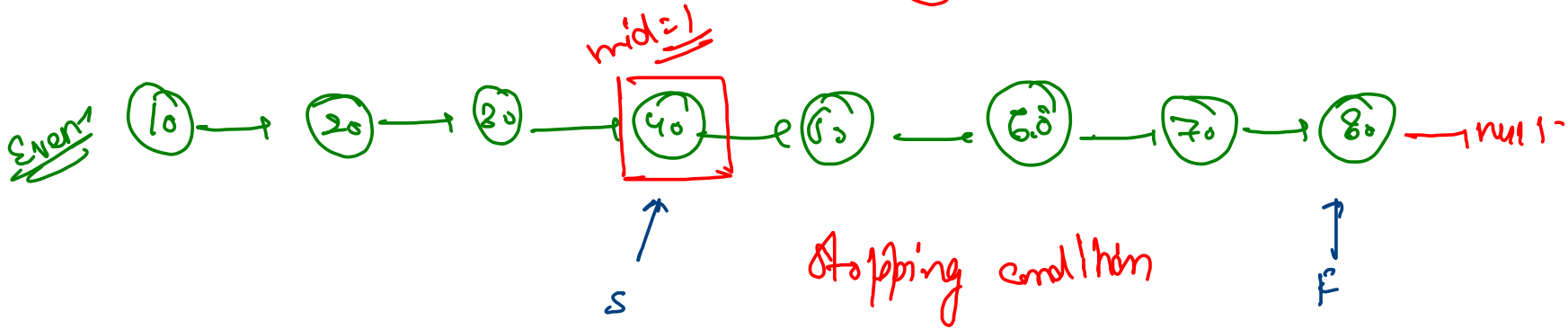
fast = head.next;

move

slow = slow.next

fast = fast.next.next





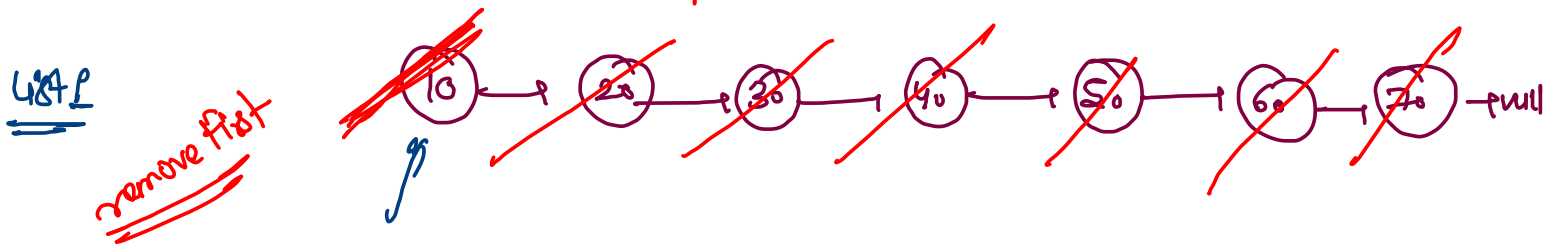
running condition = (fast != null && fast.next != null) { slow = slow.next; fast = fast.next.next; }

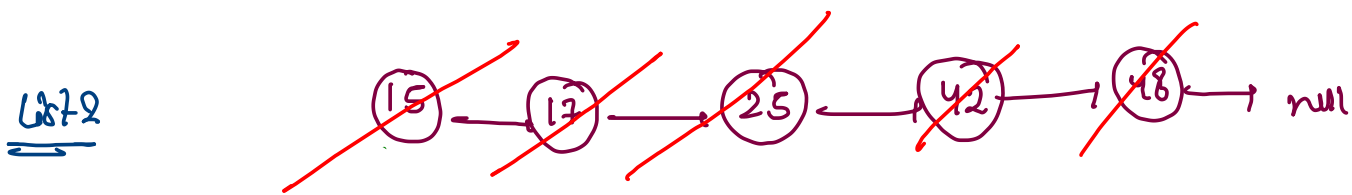
Merge two Sorted Linked List

removeFirst
↳ O(1)

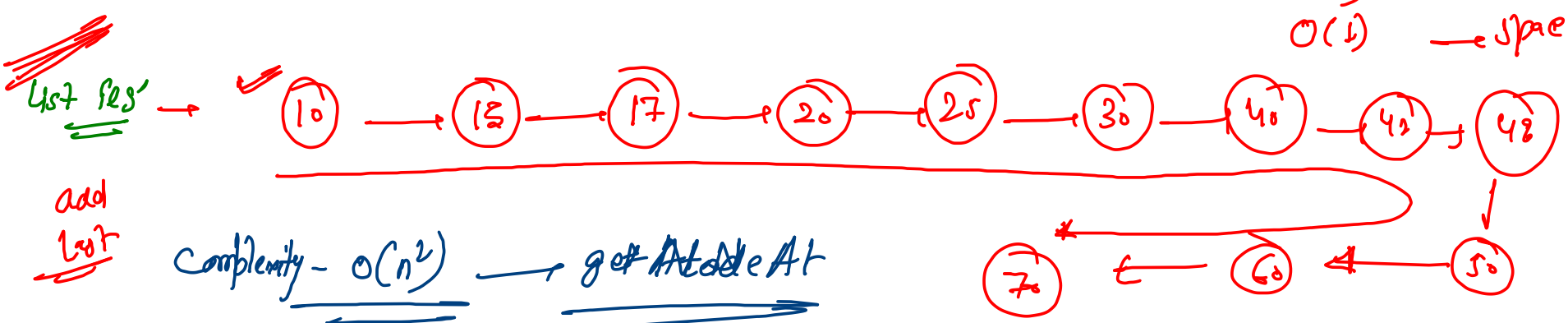
addLast
↳ O(1)

Space
↳ O(1)

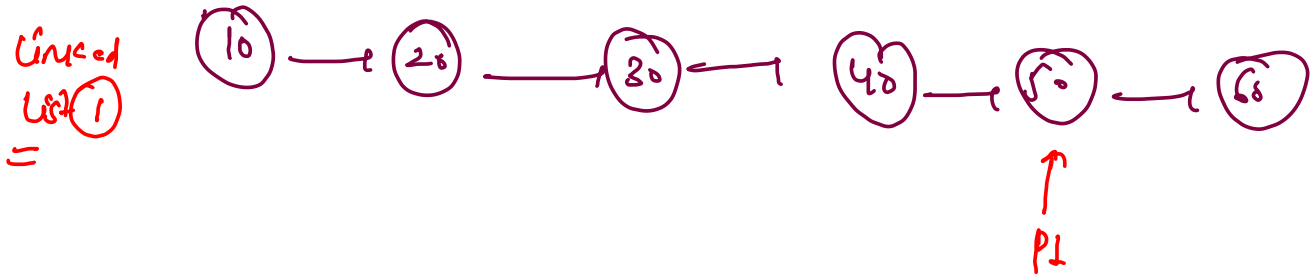


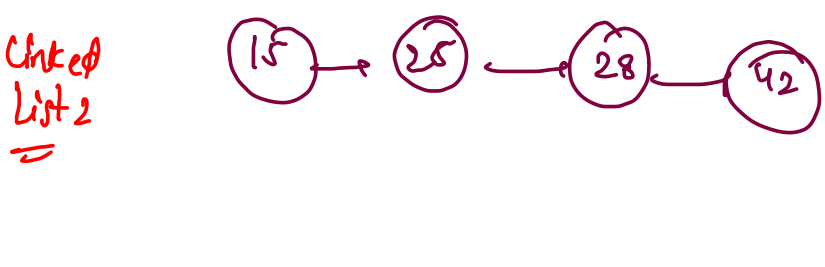


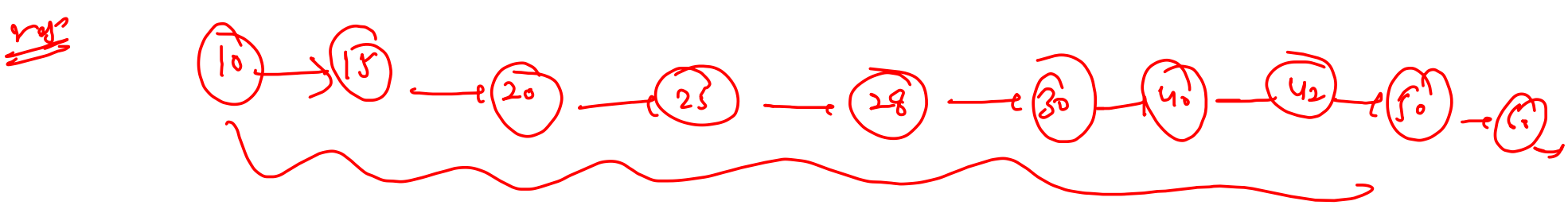
O(n) → Time
O(1) → Space.



Complexity - $O(n^2)$ → getAtAddAt


Linked List 1
10
20
30
40
50
60
P1


Linked List 2
15
25
28
42
P2


10
15
20
25
28
30
40
42
50
60

Complexities

Functions	Java Class	written by us creation
addFirst	O(1)	O(1)
addLast	O(1)	O(1) with tail, O(n) without tail
add At	O(n)	O(n)
getFirst	O(1)	O(1)
getLast	O(1)	O(1)
get At	O(n)	O(n)
removeFirst	O(1)	O(1)
removeLast	O(1) → doubly linked list	O(n)
remove At	O(n)	O(n)
size	O(1)	O(1)
display	O(n)	O(n)

doubly linked list →
1. next
2. prev

Both are managed in DLL

Because of second last node, our code is take O(n) complexity.

Stack

Last In
Last Out

- addFirst
- removeFirst

with class

- addLast
- removeLast

Queue

- addFirst
- removeLast

with class

with creation

- removeFirst
- addLast