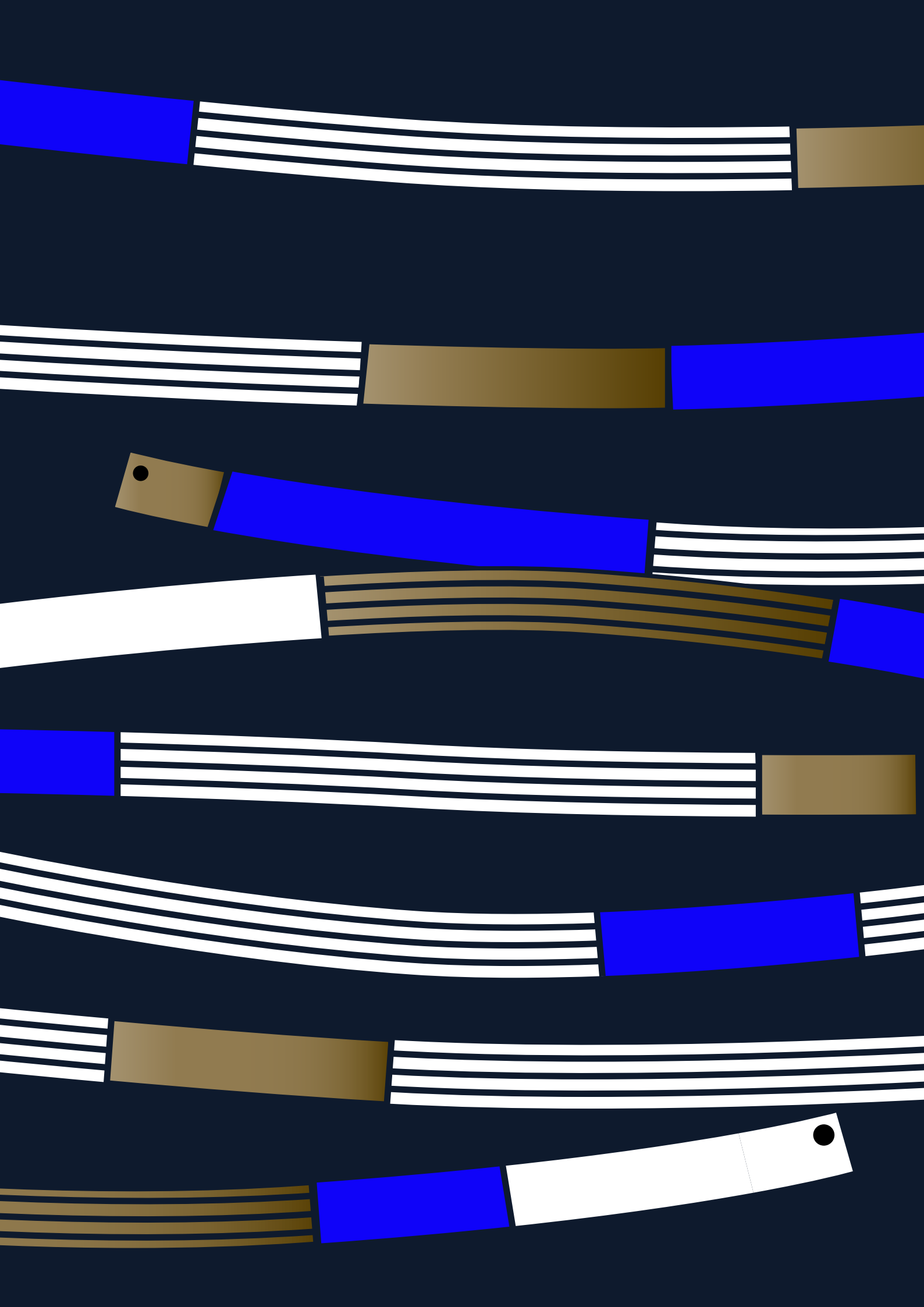


Linked List

Lecture 44

Linked List

↳ a linear data structure

↳ collection of nodes

→ Dynamic DS (grow / shrink at runtime)

Thus, no memory wastage

NODE: 

Why?

→ In array, size cannot be changed at runtime

→ In vector, size can be increasing (doubles up) which results in storing and copying of values thus NOT OPTIMAL.

✗ Deletion is easy, as no shift needed.

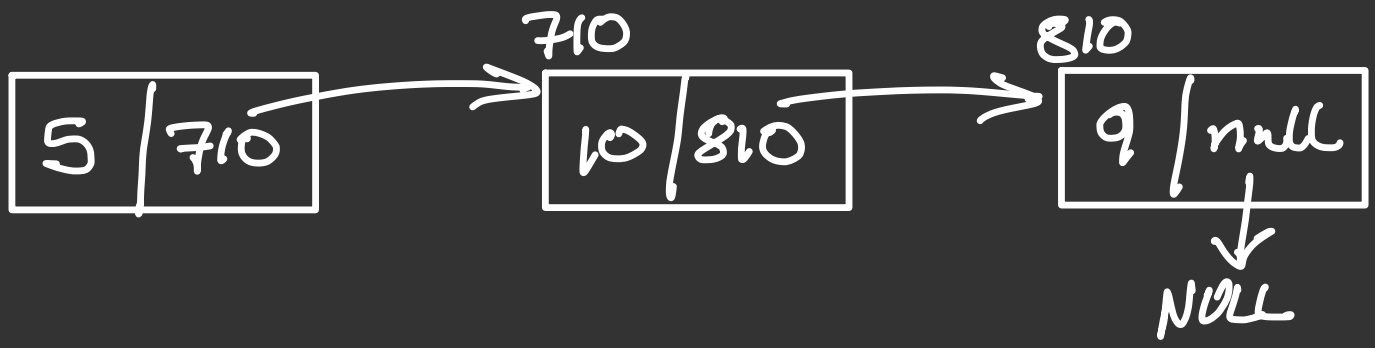
Types of Linked List

→ Singly LL
→ Doubly LL
→ Circular LL
→ Circular Doubly LL

Node →

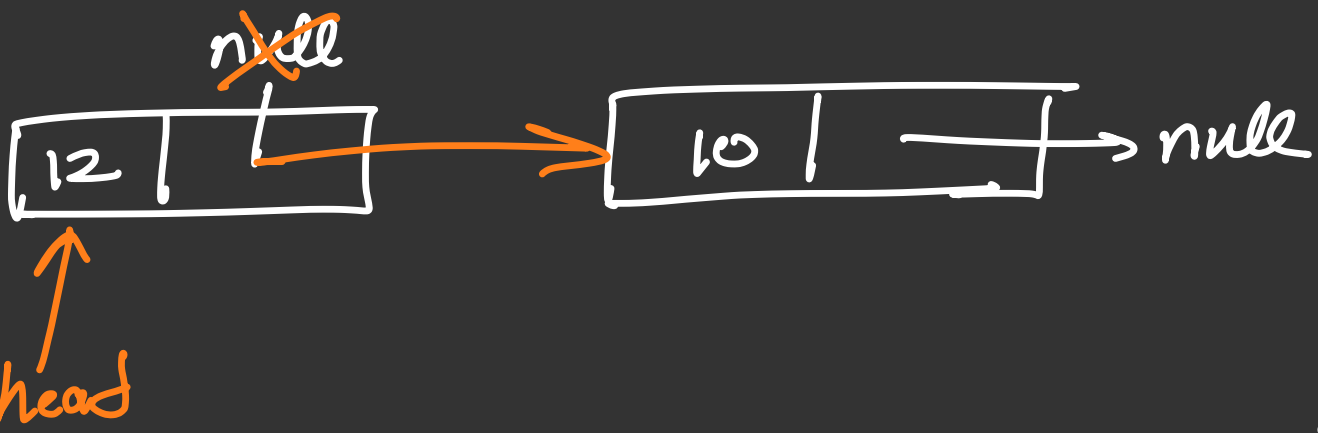
```
class Node
{
    public:
        int data;
        Node* next;

        Node(int data)
        {
            this -> data = data;
            this -> next = NULL;
        }
};
```

Singly Linked List



Insertion →

```
Node *head = NULL;
Node *node1 = new Node(10);
head = node1;
```

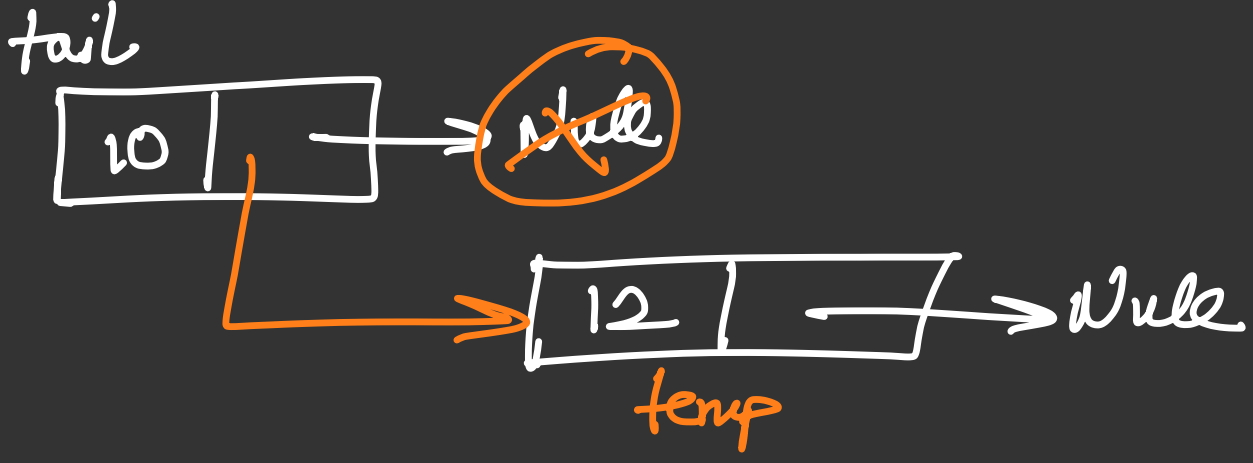



```
Insert_at_Head(Node *&head, int d)
{
    Node *temp = new Node(d);
    temp -> next = head;
    head = temp;
}
```

→ singlyLinkedList.cpp

☆ Insert at → a given position

↙ Head ↘ Tail



→ Better approach is to have an initial node, and point head and tail to that, then add other nodes accordingly using functions.

This is not necessary just check if head/tail == NULL and modify the code.

Insert at any position

i/p ⟶ 3rd position, 22

head

Thus, inserting at head
↳ pos = 1

inserting at tail
↳ when next pointing to null

Deletion

- at pos
- at value

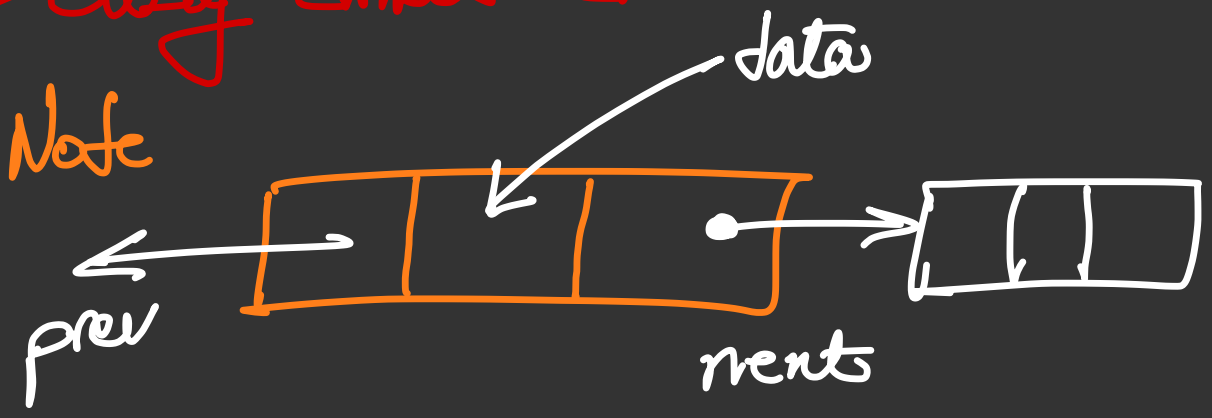
⭐ Whenever deleting "temp"

1. temp -> next = NULL;
2. delete temp2;

```
if (this -> next != NULL)
{
    delete next;
    this -> next = NULL;
}
```

3 → we do this so if it is not equal to null, then it will recursively delete all the nodes it has been connected to.

Doubly Linked List



→ doublyLinkedList.cpp

→ print

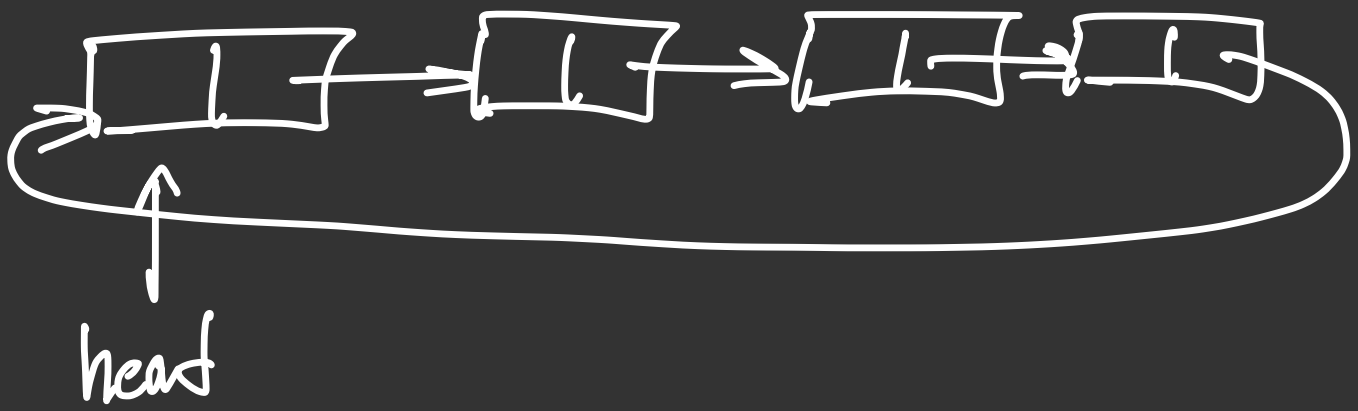
→ length of the list

→ Insert at → head
→ tail
→ any position

☆ Whatever code is written based on that there is already a node in the list to which head and tail are pointing to.

Thus, for empty list, code must have to differ.

Circular Linked List



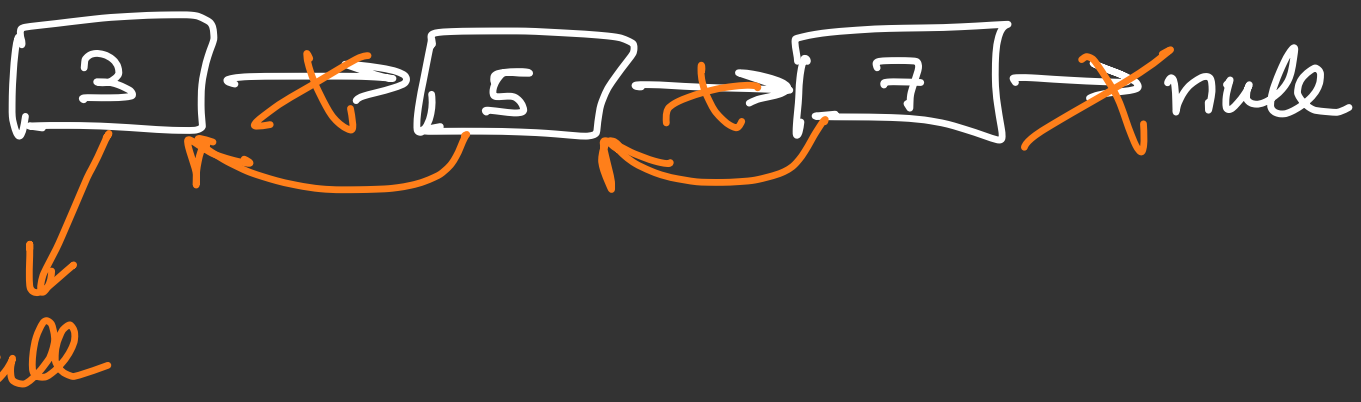
→ circularLinkedList.cpp

☆ In my code, I have considered a head, but in general there just has to be a reference @ current variable.

☆ Also, practice inserting a node after a certain element is there in a node.

Lecture 45

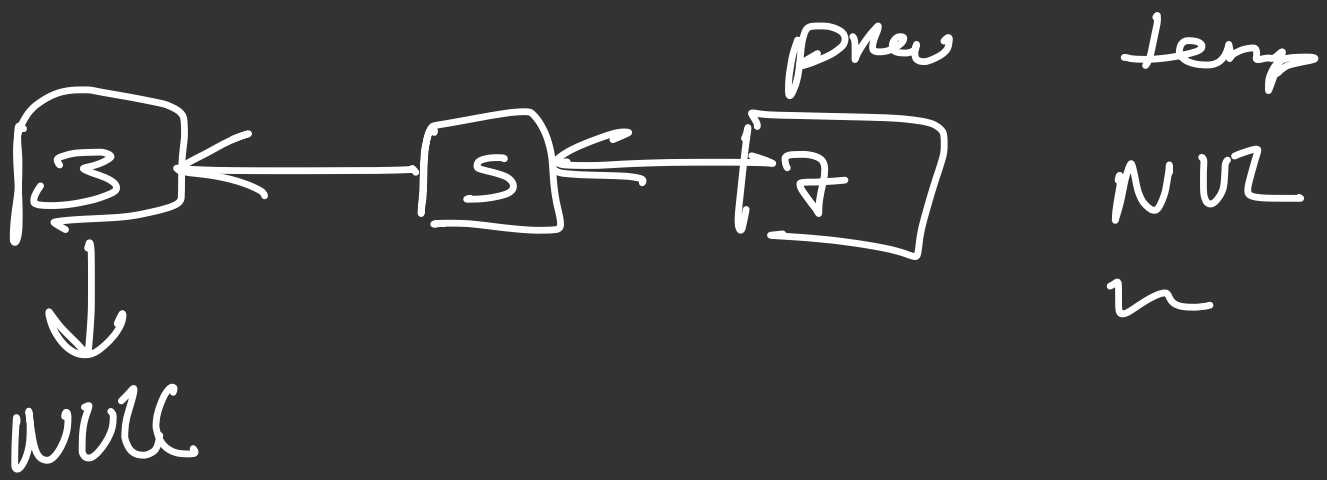
Reverse LL



head = 3 = temp

prev = NULL 3

n = 5



⟶ reverse_LL.cpp

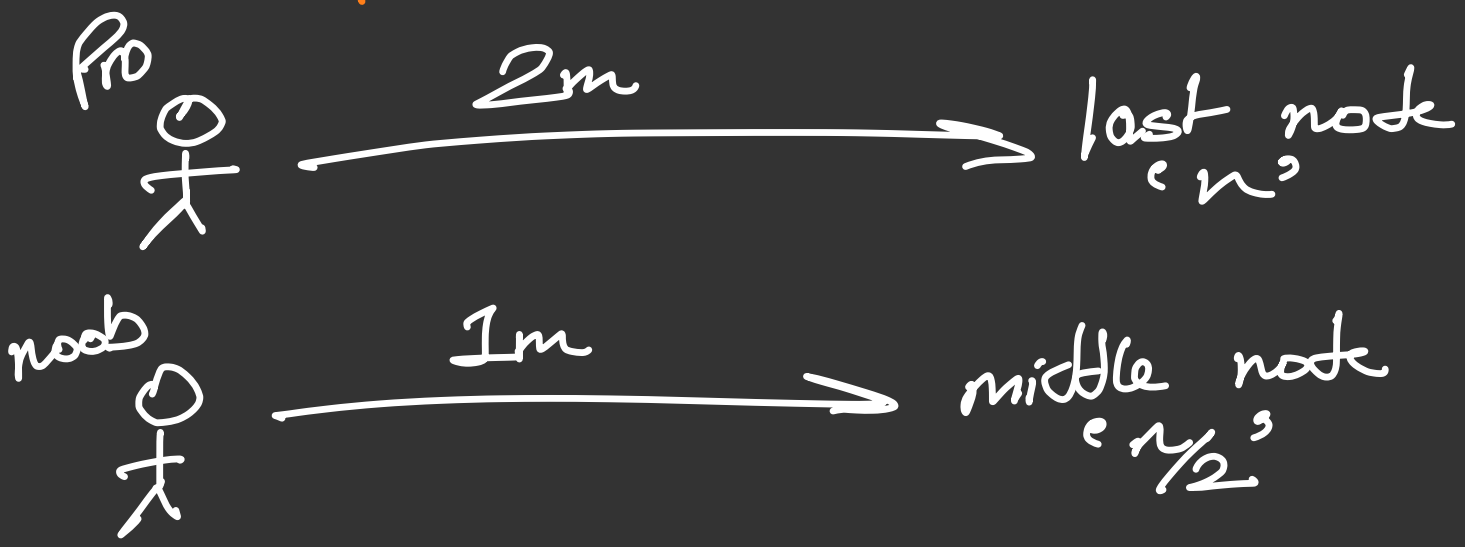
Reverse using Recursion

→ TC = O(n)
→ SC = O(n)

head
temp = head
prev = NULL

```
reverse(head, temp, prev)
{
    if(temp == NULL)
    {
        head = prev;
        return;
    }
    Node *forward = temp -> next;
    temp -> next = prev;
    prev = temp;
    temp = forward;
    reverse(head, temp, prev);
}
```

Middle of Linked List

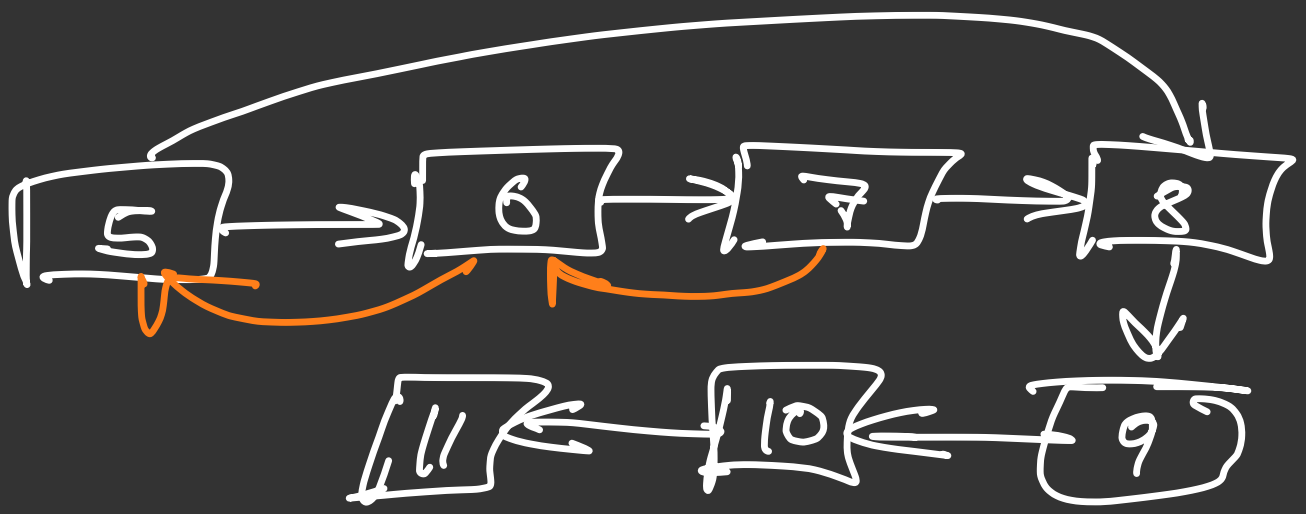


→ middle_LL.cpp

☆ While the first node which increments by 2 will reach the end, second node will reach the middle.

Lecture 46

K-Reverse (DiD iT)



reverse (start, actual head for now, a counter to store previous end, end)

⟶ kReverse.cpp

isCircular

~~Showing~~ TLE ??

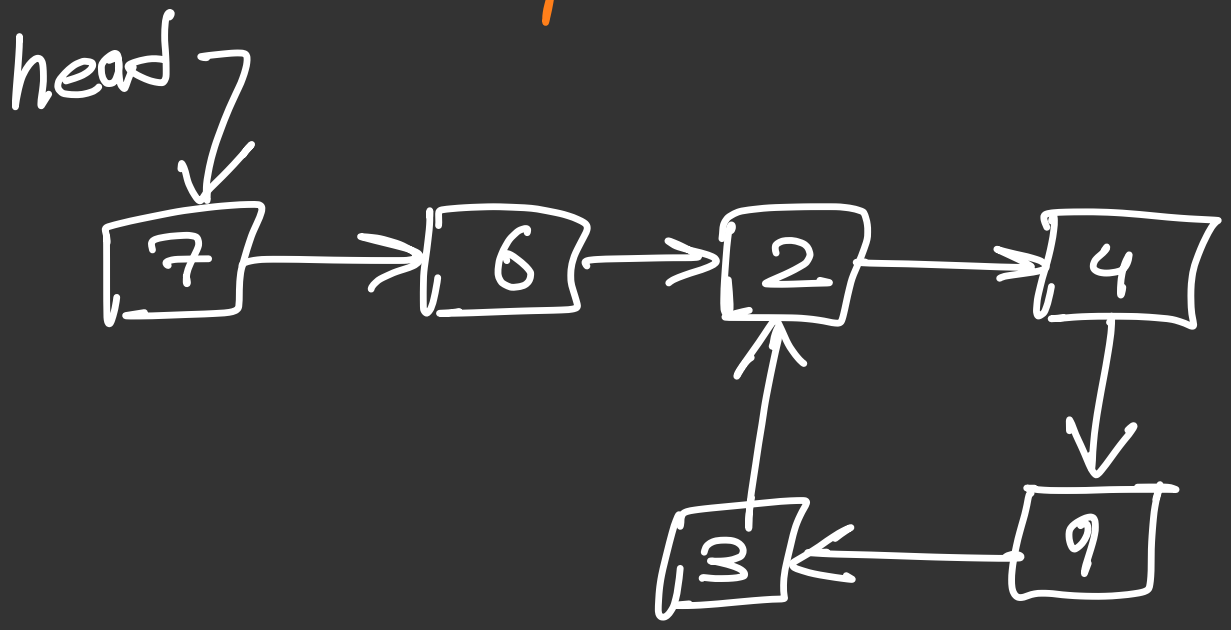
⟶ isCircular.cpp

Lecture 47

→ Detects cycle/loop in LL

→ Remove cycle from LL

→ beginning / start node of loop in LL



Approach 1 → TC = O(n) = SC

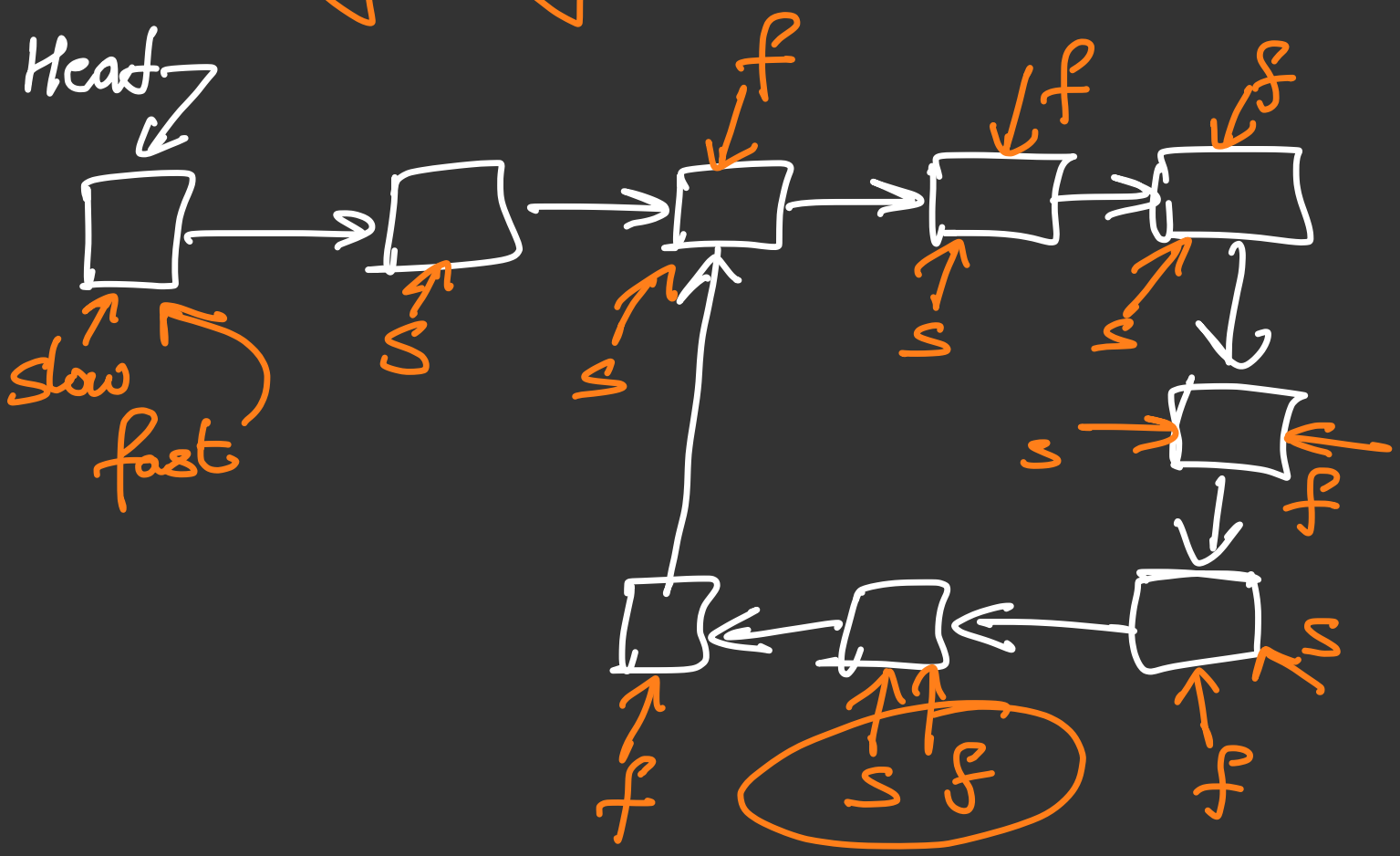
→ via maps, they store key-value pairs

map<Node*, bool> visited;

→ loop-LL.cpp

Approach 2

↳ Floyd's Cycle detection Algorithm



slow = 1 step

fast = 2 step

if (slow == fast)

↳ loop is present.

TC = O(n)

SC = O(1)

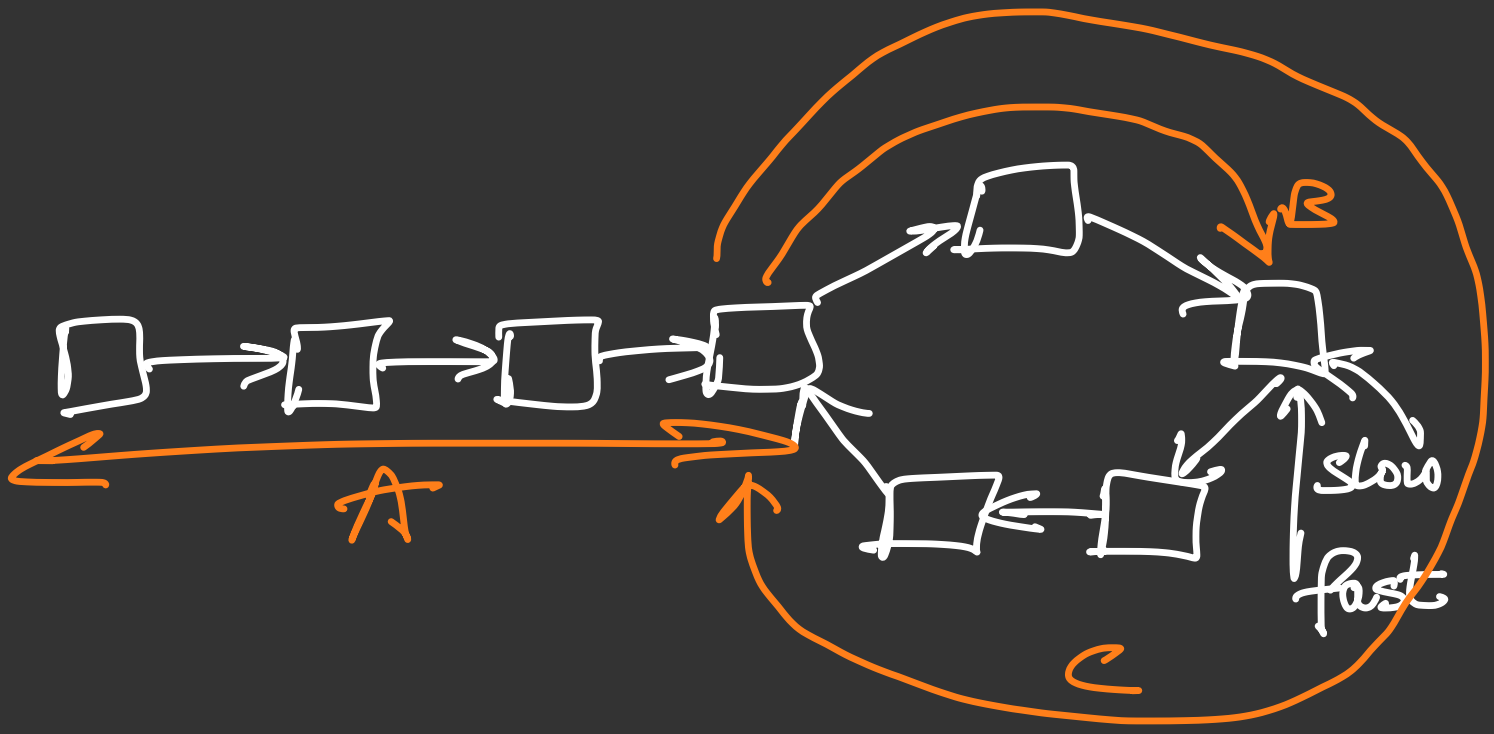
Starting node of the loop?

Approach →

(1) Using FCD node inside loop where slow meets fast

(2) Now make slow equal to head, and now increment both by 1.

→ Where they meet
↳ STARTING NODE of the Loop



Distance by fast pointer
$= 2 \times$ slow pointer

$$A + xC + B = 2 \times (A + yC + B)$$

$$A + xC + B = 2A + 2yC + 2B$$

$$C(x - 2y) = A + B$$

(or)

$$A + B = \underline{k} \times C$$

This should be taken as

$$A + B = C$$

Also, removing the loop after finding the starting node

→ firstNodeOfTheLoop.cpp

→ removingTheLoop.cpp

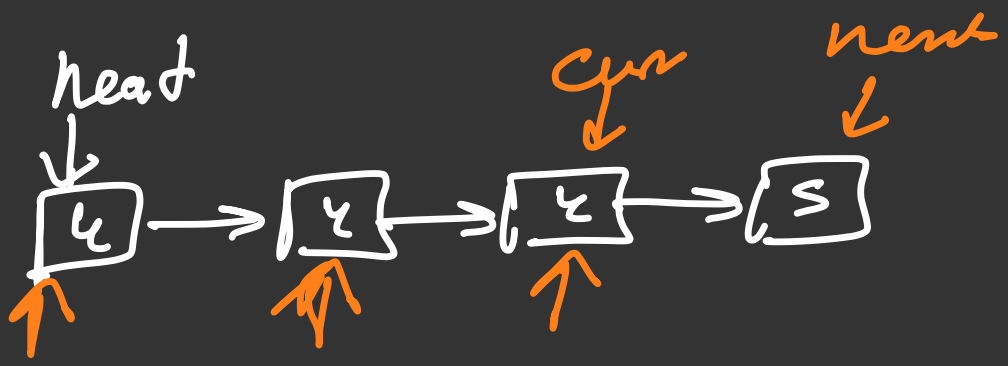
$SC = O(1)$

$TC = O(n)$

Lecture 48

Removing Duplicates

① Sorted LL



→ removeDuplicates_Sorted.cpp

② Unsorted LL

Approach 1:

→ pick an element
→ traverse all the elements after that element and if same delete

↳ TC = O(n²)

Approach 2

↳ Sort the LL

↳ Then duplicate removal in sorted

space ↓, time ↑

↳ O(nlogn)

Approach 3

↳ Use map

↳ key value pair

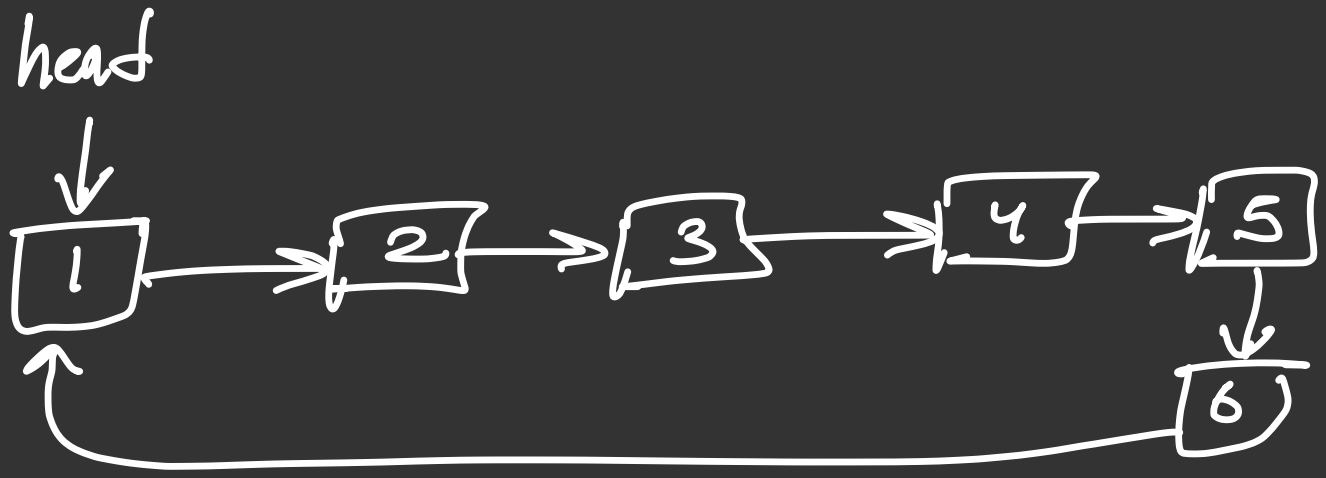
space ↑, time ↓

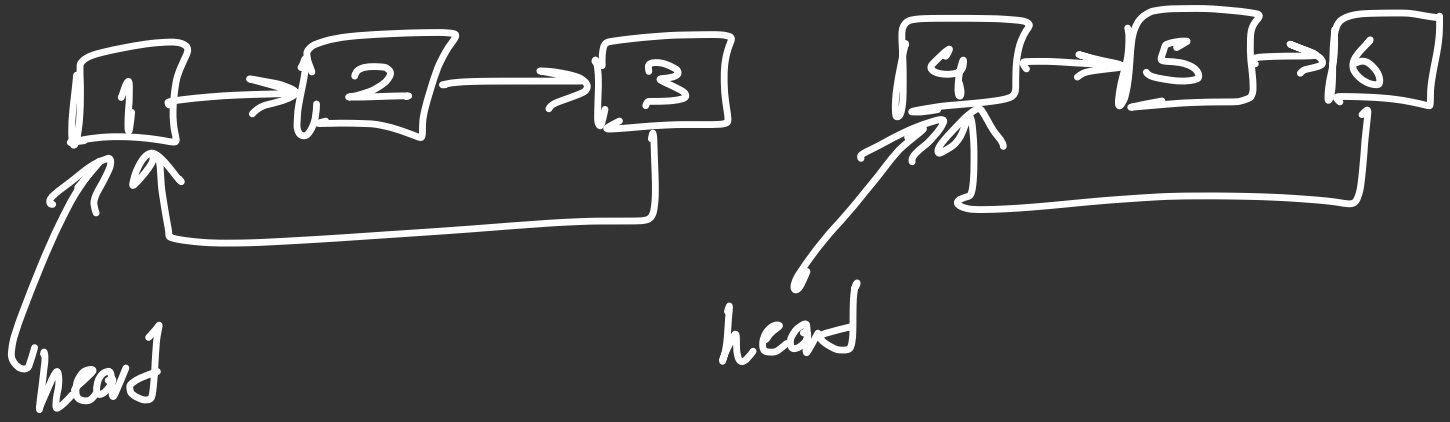
↳ O(n)

→ removeDuplicates_UnSorted.cpp

Homework Question

Split Circular LL into 2 LL

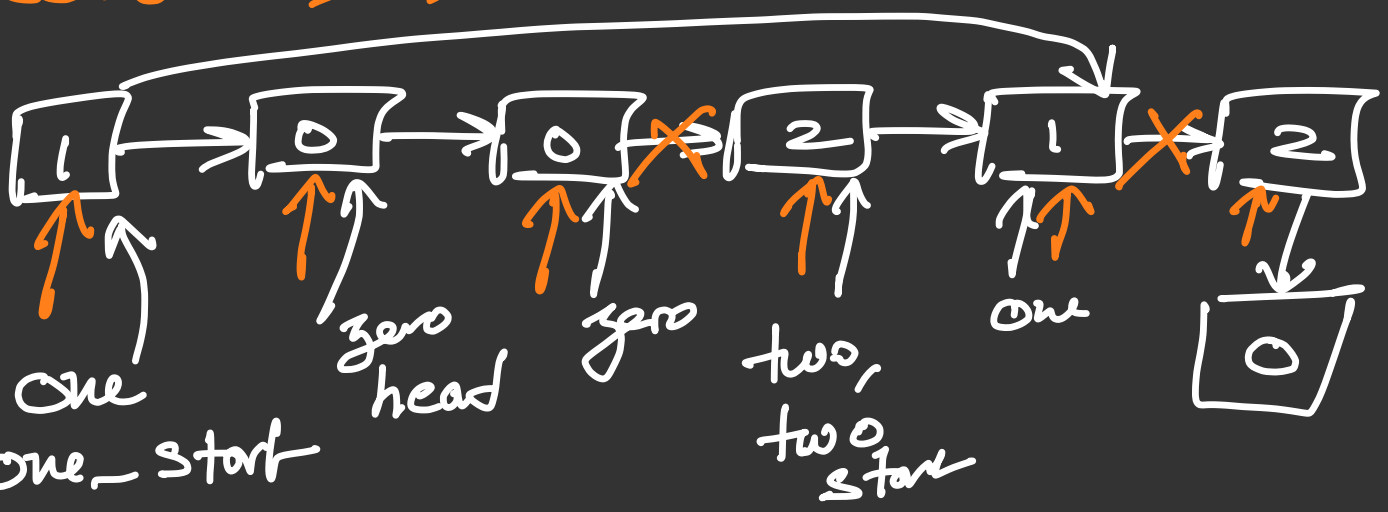




→ splitCircular_LL.cpp

Lecture 49

Sort 0, 1, 2



Approach ⟶

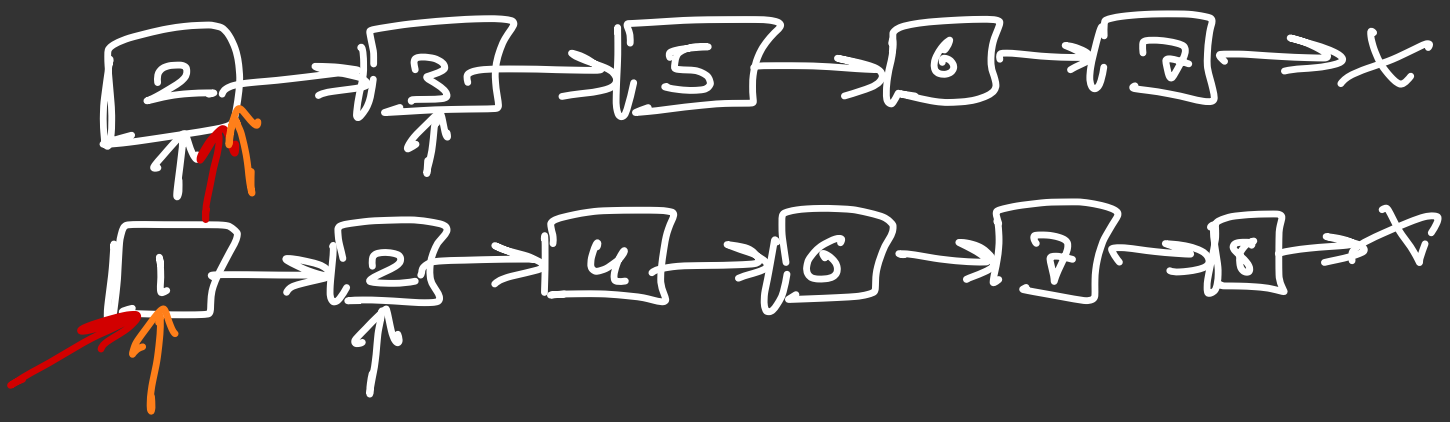
have three different lists

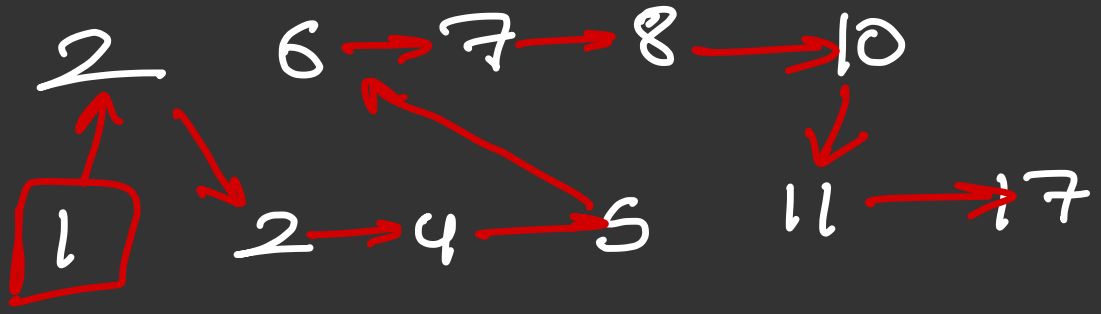
↳ one, two, three

and then join them in the end.

⟶ 0_1_2_LL.cpp

Merge 2 sorted LL

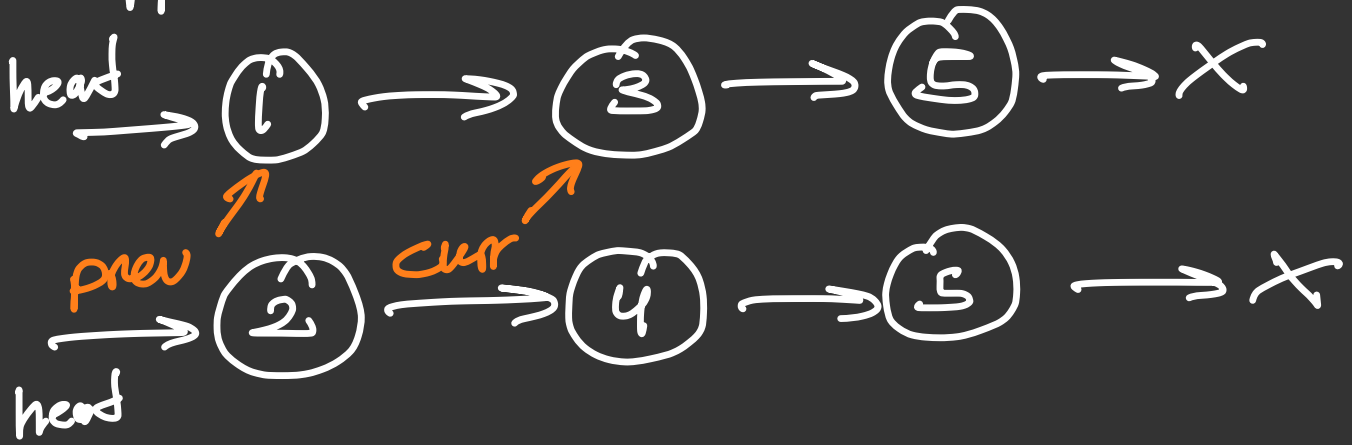




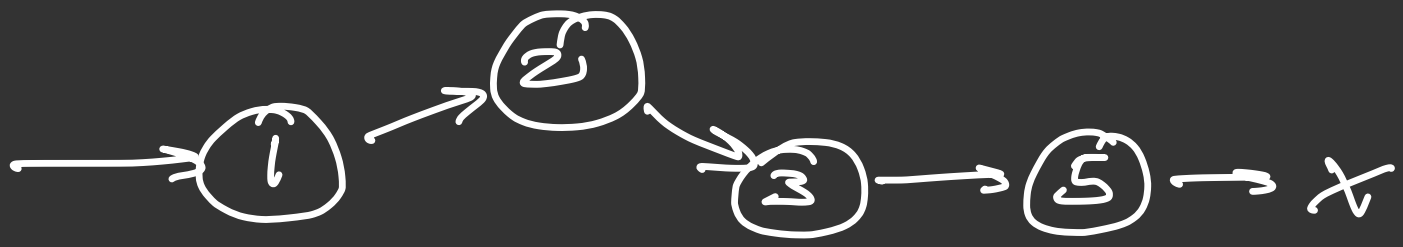
Approach 1

↳ what I did

Approach 2



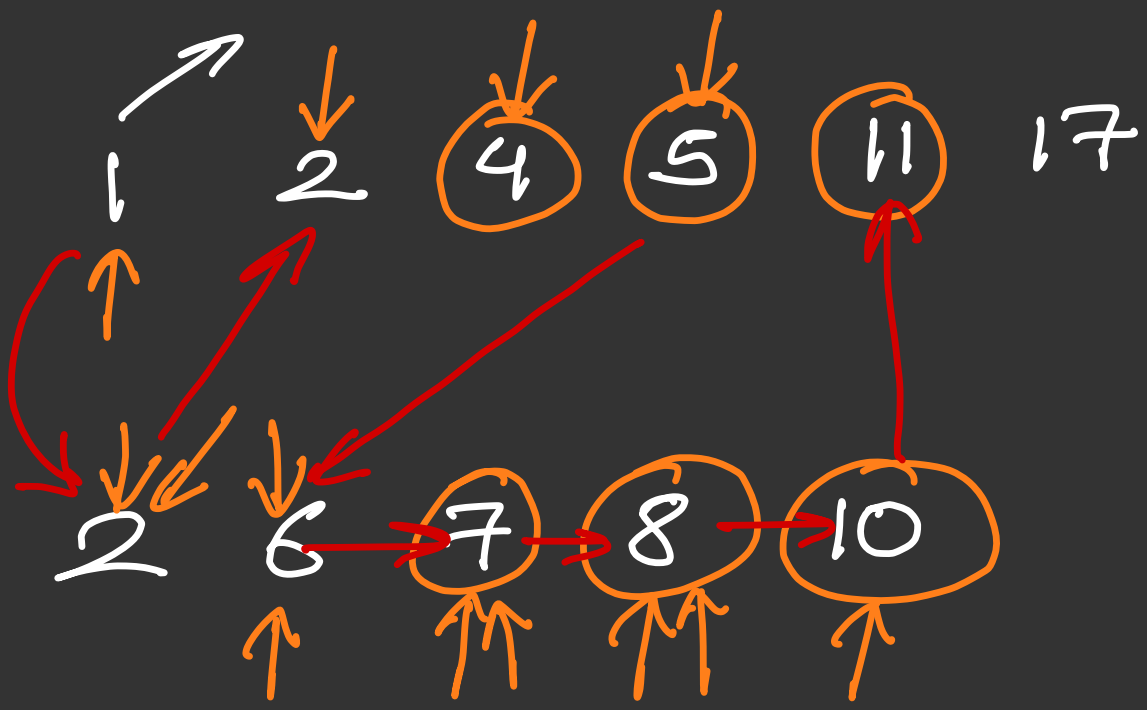
☆ If we can insert [2] in b/w prev and current do it.

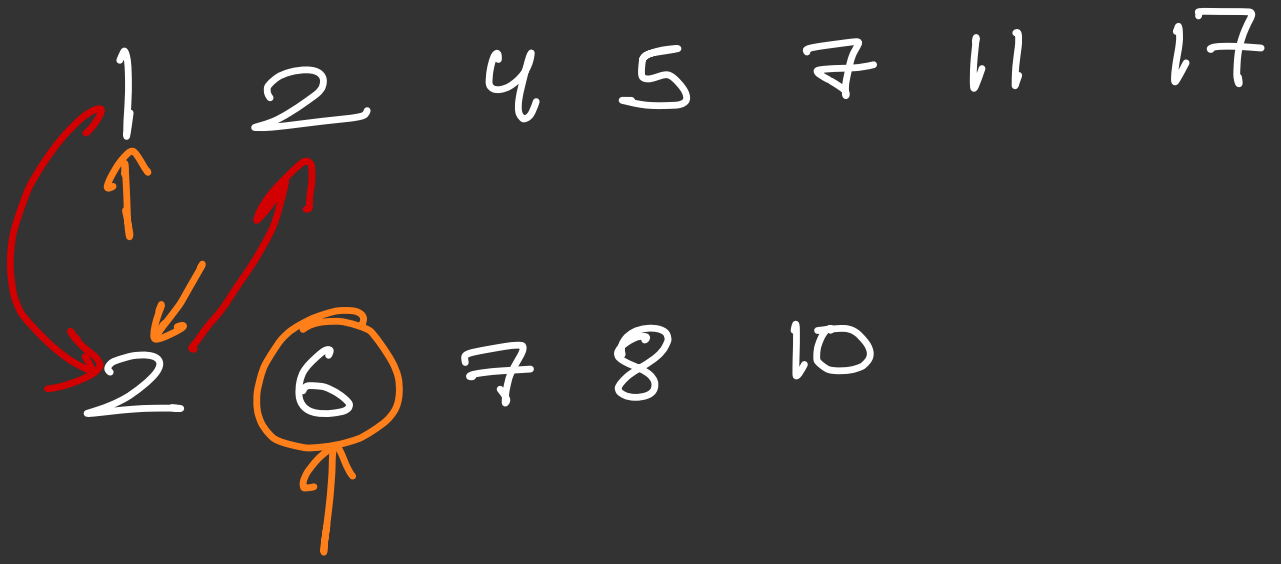


(prev->data) ≤ temp->data ≤ (cur->data)

↳ insert temp in middle

Else, shift one place forward. update cur and prev pointers.





finally, done approach 2 also.

Lecture 50

Check if LL is Palindrome ?

Approach 1

→ Create an array and copy elements of LL and check (vector array)

But, TC = SC = O(N)

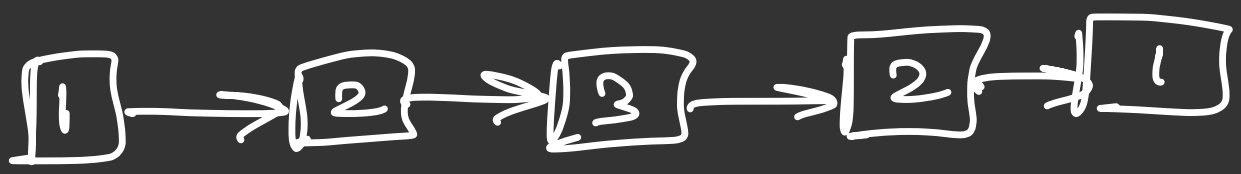
Approach 2

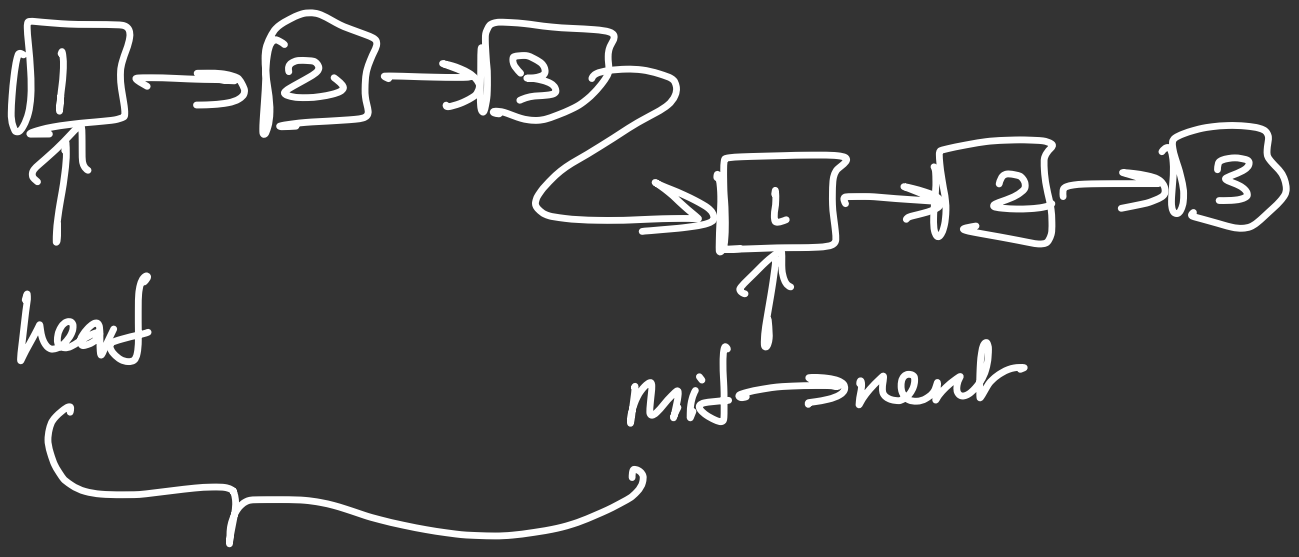
→ with space complexity as O(1)

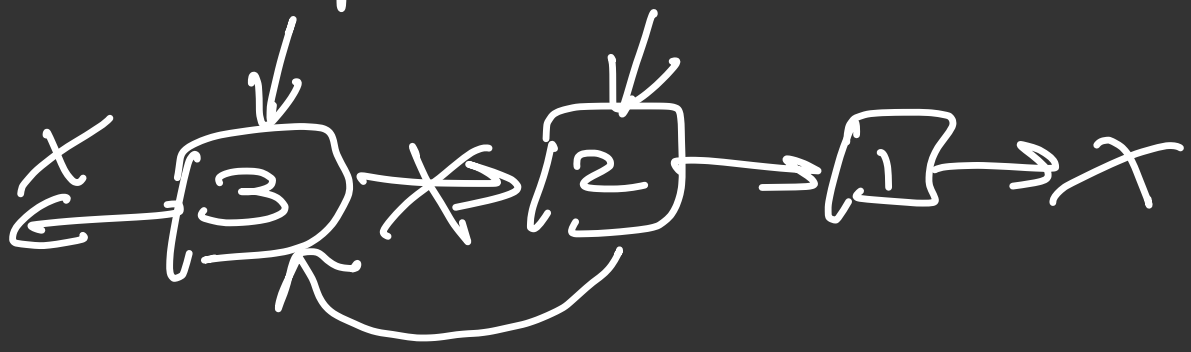
1. Reach mid
2. After reaching mid, reverse the list from there

③ Compare both the halfs.

example :→







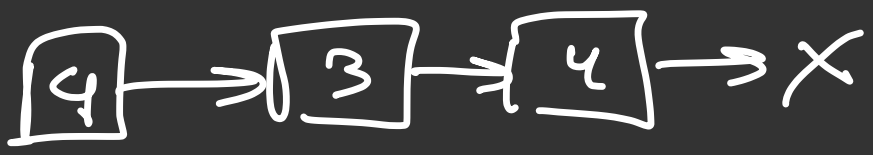
head = NULL

→ palindrome_LL.cpp

Lecture 52

Add 2 numbers using Linked List

example:



$$\begin{array}{r} {}^{1}3\,9\,4\,5 \\ +\;\;4\,3\,4 \\ \hline 4\,3\,7\,9 \\ \hline \end{array}$$

(1) Reverse the lists

(2) Then add

- keep a check of carry
- one number may be of more digits than another.

Example:

```
    1 1 1 1 1 1
  1 4 5 9 6 7 8 4
+ 6 2 7 5 4 6 7 9 9
-----------------
  6 3 2 1 4 3 5 8 3
-----------------
```

→ add2Numbers — LL · qp.

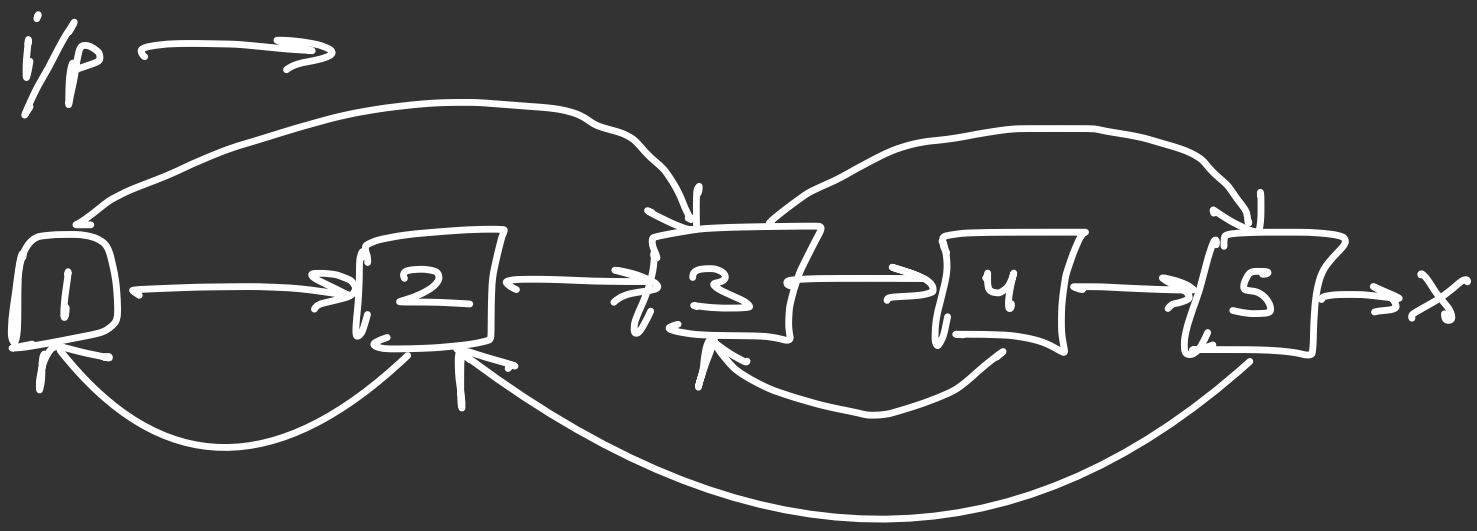
```
        1 1
  2 9 1 2 9 9 2 8 2
            2 8 9
-----------------
  2 9 1 2 9 9 5 7 1
-----------------
```

TC = O(N+M)

SC = O(Max(M,N))

Lecture 52

Clone a LL with Random pointers



→ next pointer

→ random pointer

☆ While cloning, the original LL will pointing to a different address via random pointer, so how will you match that ?

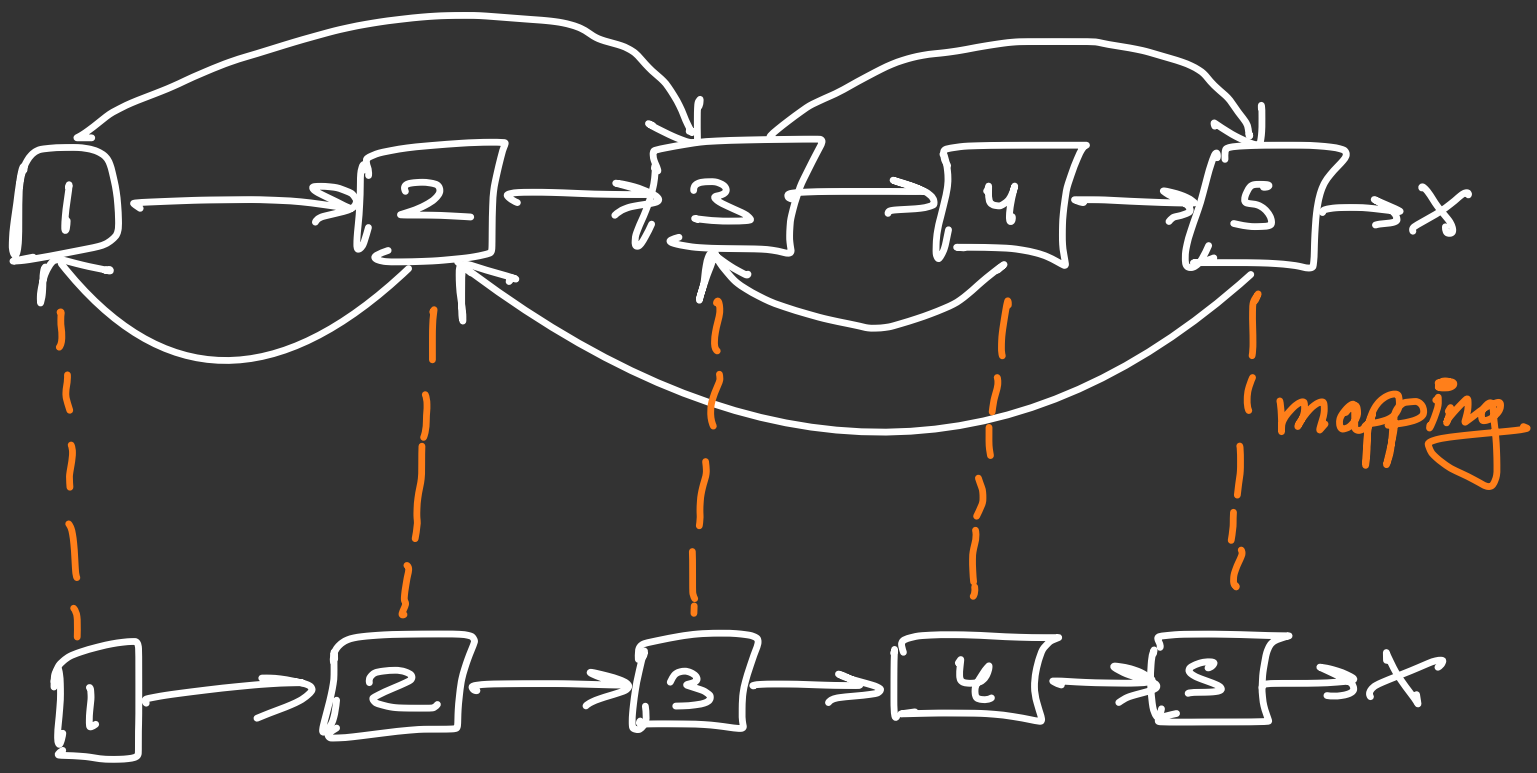
Approach 1 ⟶

1. Create the base LL
2. Now traverse this list and check the distance it takes to traverse from the original node to random pointer node, do the same with copy LL.

$TC = O(n^2)$

Approach 2 ⟶

1. Create the clone base list
2. Mapping of original node with Clone no



③ Now traverse again the LL and via mapping point the null pointers to the nodes respectively

cloneNode → random

= mapping[originalNode → random]

mapping[originalNode]

TC = O(n) = SC

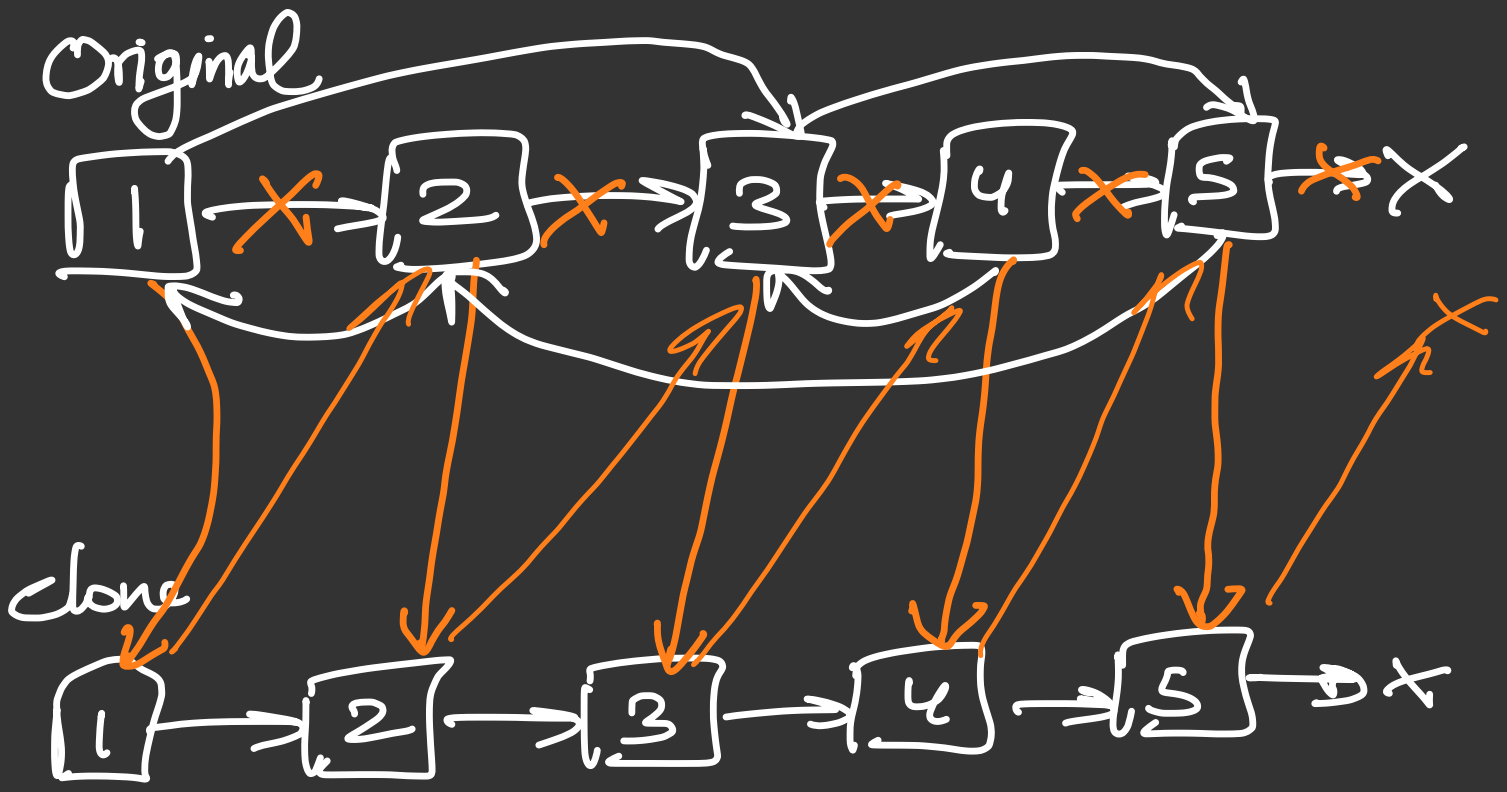
→ cloneA_LL.cpp

Approach 3 →

(To reduce space complexity)

↳ reduce map

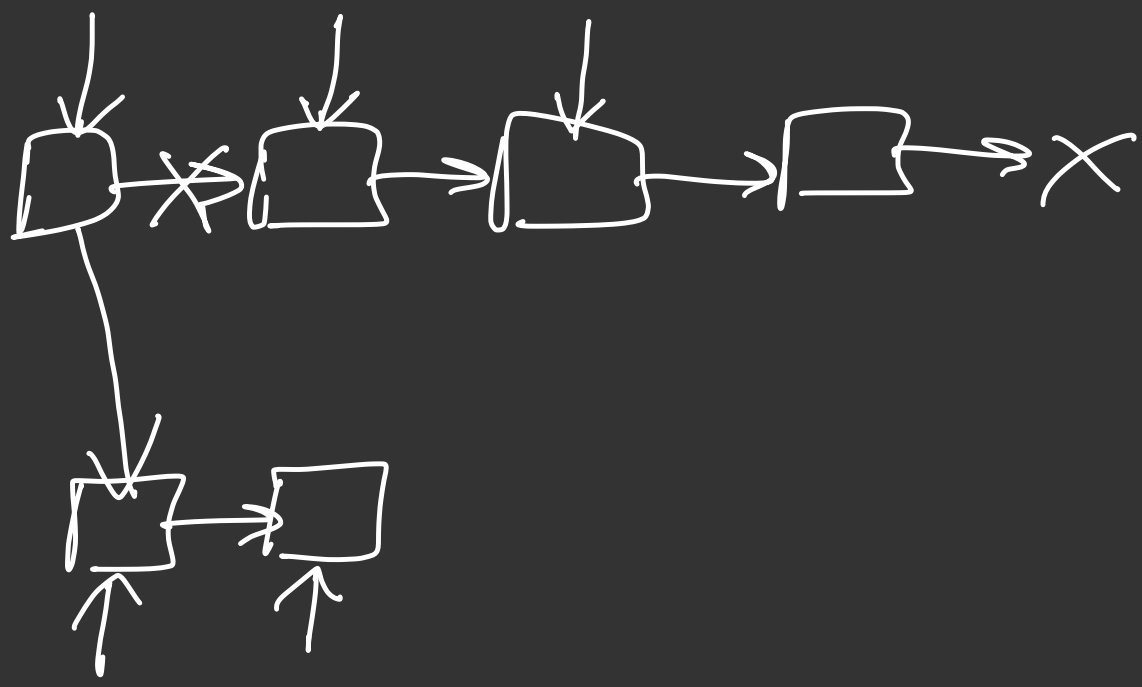
✗ Remove map and change LINKS



→ Random pointers are still pointing to original nodes

→ Once next pointers work is over we use them to create the mapping

→ And now do the rest.



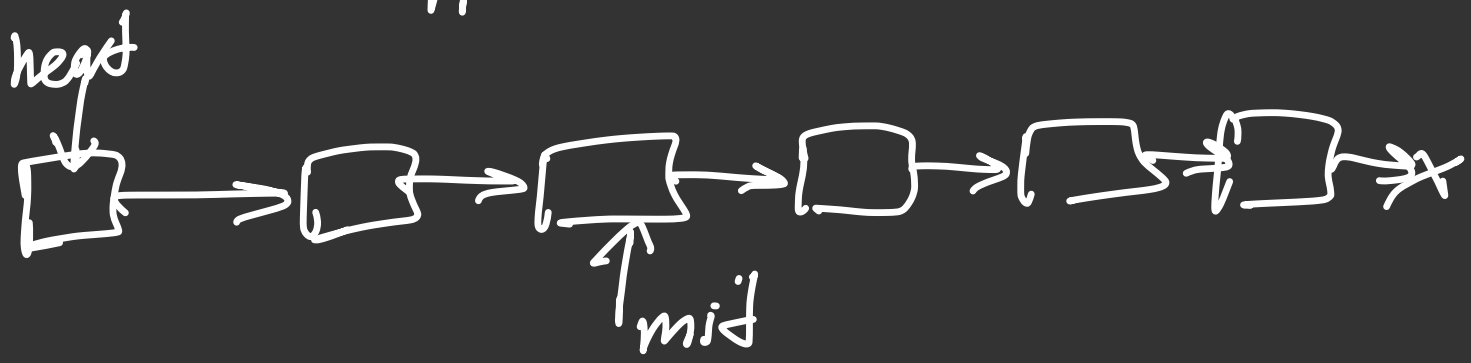
Approach 3 also in
↳ cloneA_4.cpp

Lecture 53

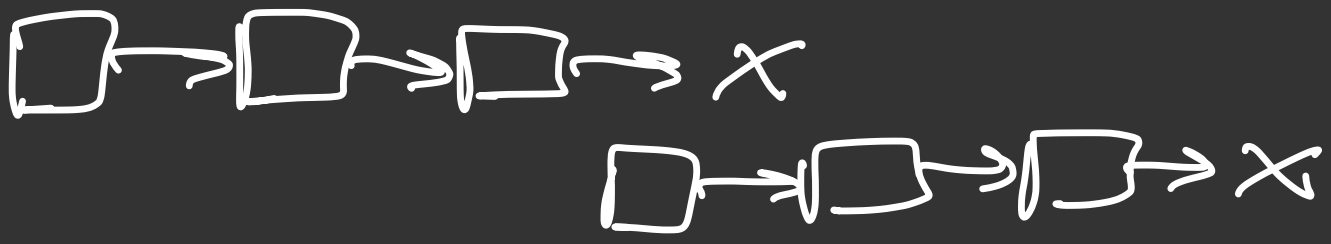
Merge Sort in Linked List

1. Base Condition
   - ↳ when no element or only one element left

2. Now, break the list into two mid ↳ through slow and fast approach




```
head1 = head;
head2 = mid->next;
mid -> next = NULL;
```



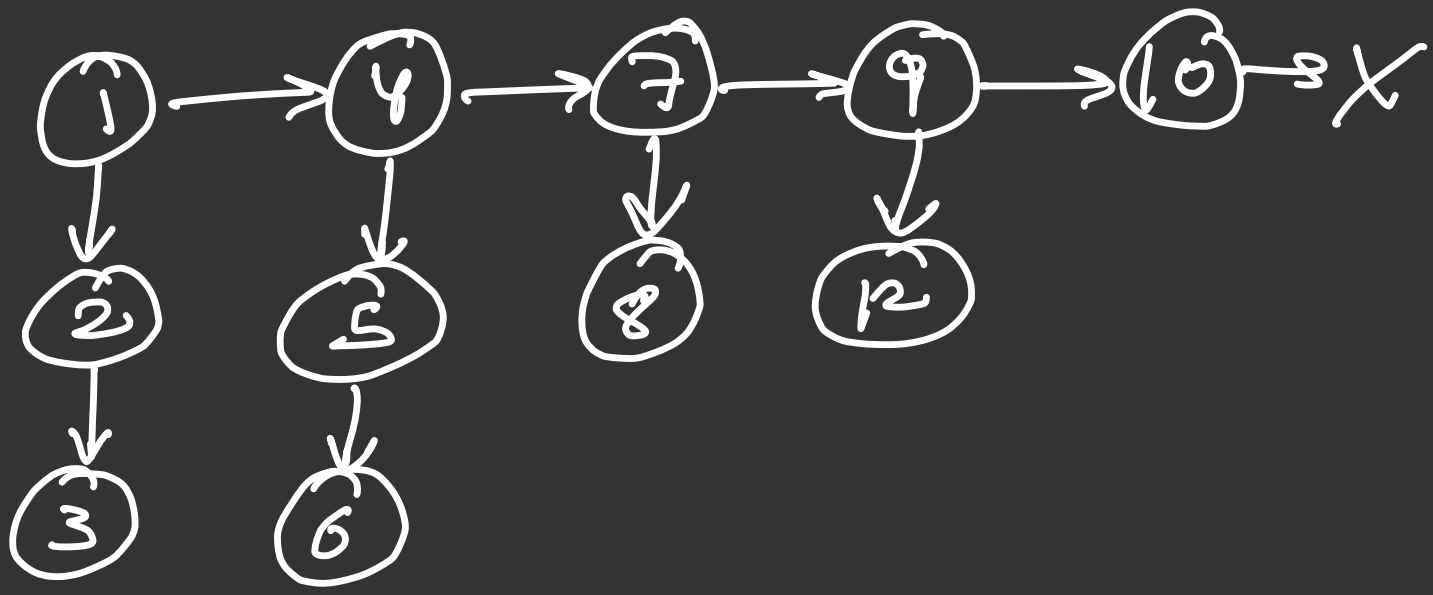
③ Now, sort both of these through recursive calls

④ Now merge finally

⟶ mergeSort_LL.cpp

Flatten a Linked List

example:



→ Node
  ↳ next
  ↳ child

→ Child linked lists are sorted

→ flattenLinkedList.cpp