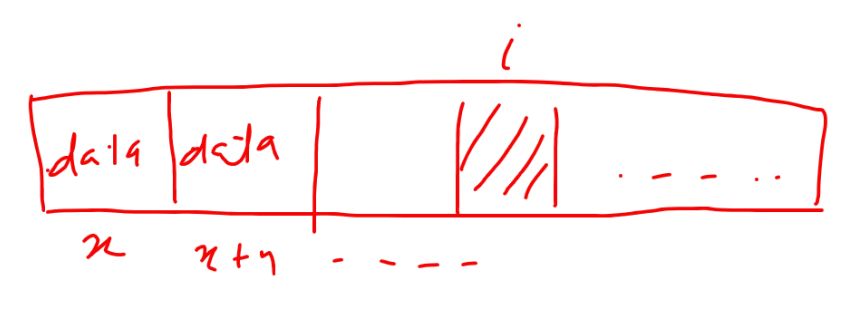
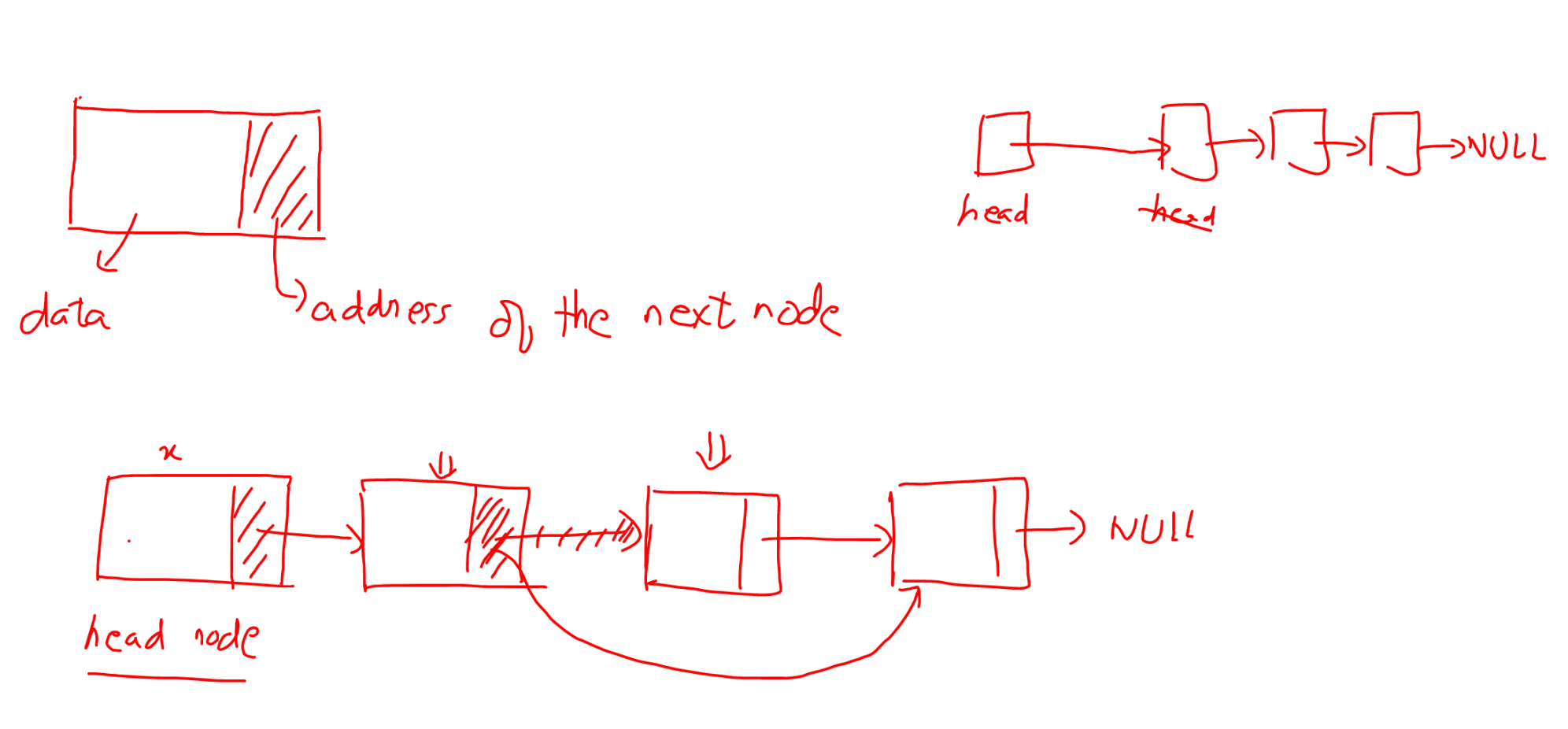
Arrays:



data
data
x
x+1

In L.L. elements are not stored in contiguous memory locations (unlike arrays).



data
address of the next node
head
head
NULL
head node
NULL

Adv. of L.L over arrays:

1) No upper limit of size unlike static arrays.
2) Insertion at the beginning of a L.L takes O(1) time.
3) No shifting of elements is required to delete something from a L.L.

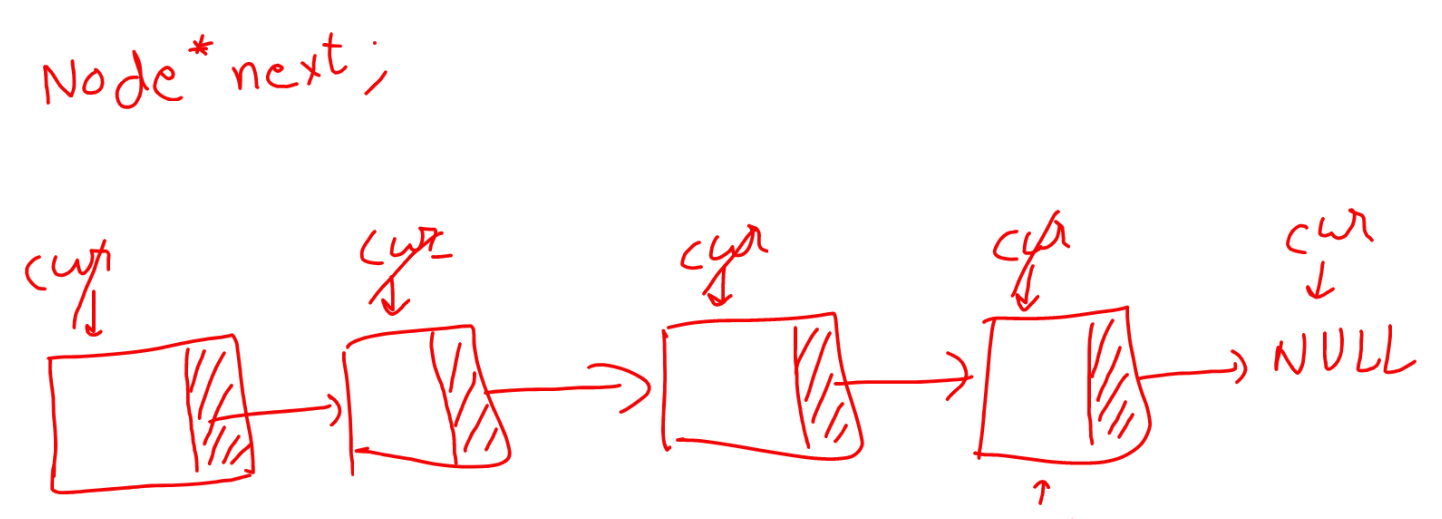
Disadv. of L.L:

1) No contiguous memory allocation leads to no random access. Any element can be accessed only in O(n) time.
2) Requires some extra storage compared to arrays.
3) Not cache-friendly.

C++
x → data
x → next

Java
x. data
x. next

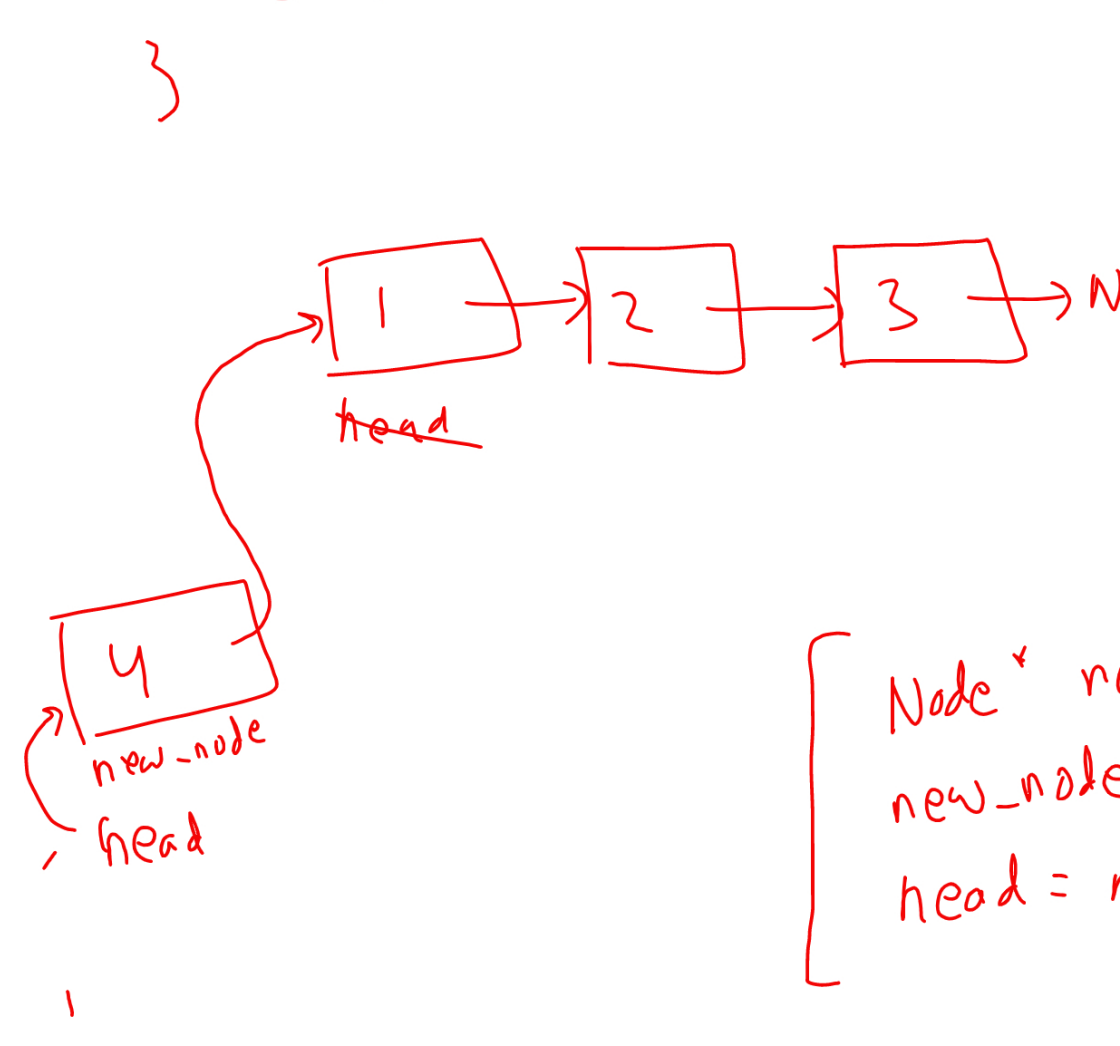
```
class Node {
    public:
        int data;
        Node* next;
};
```



cur
cur
cur
cur
cur
cur
NULL
head

(cur→next != NULL) ✗

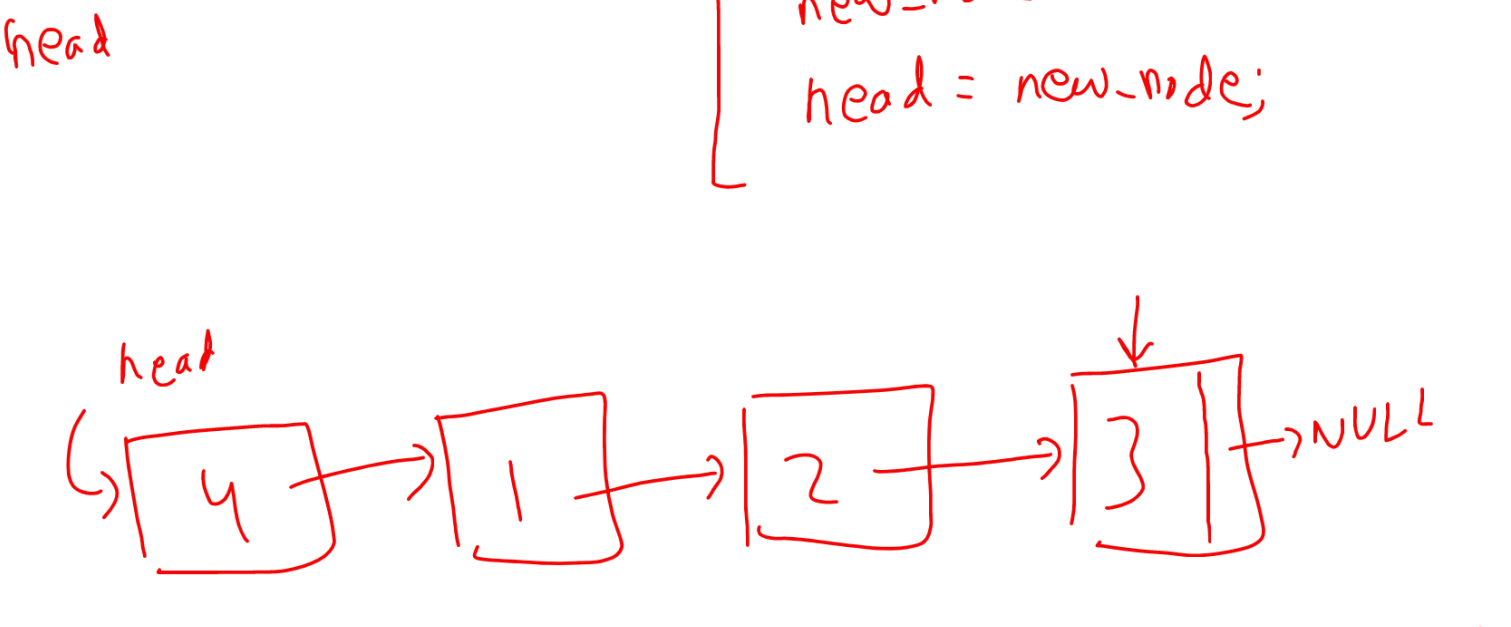
```
cur = head;
while (cur != NULL) {
    // ------
    cur = cur→next;
}
```



1
2
3
NULL
head
4
new_node
head

(d)

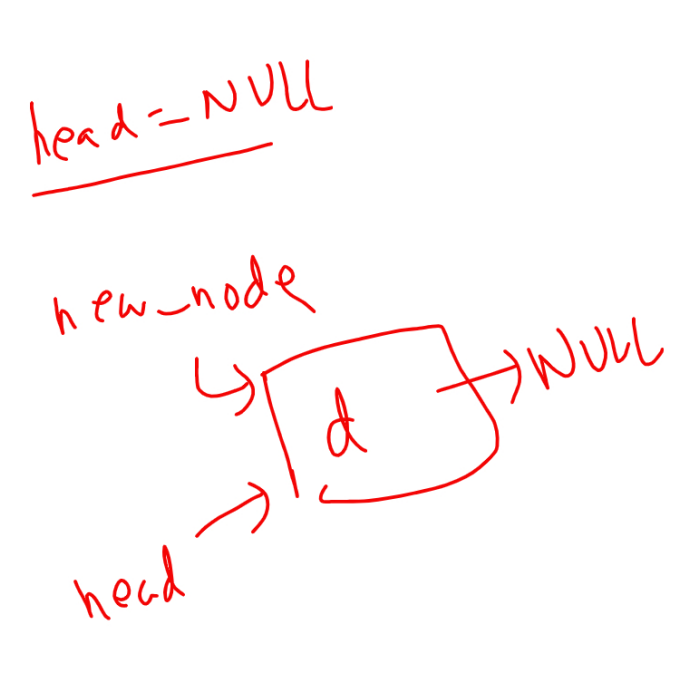
```
Node* new_node = new Node(d);
new_node→next = head;
head = new_node;
```



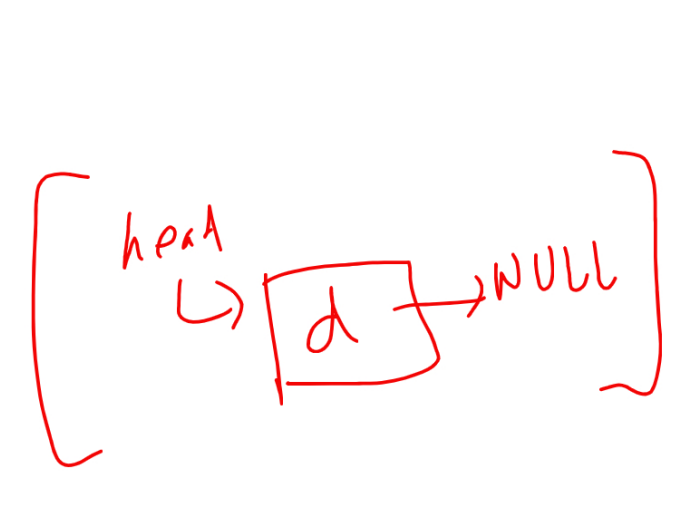
head
4
1
2
3
NULL

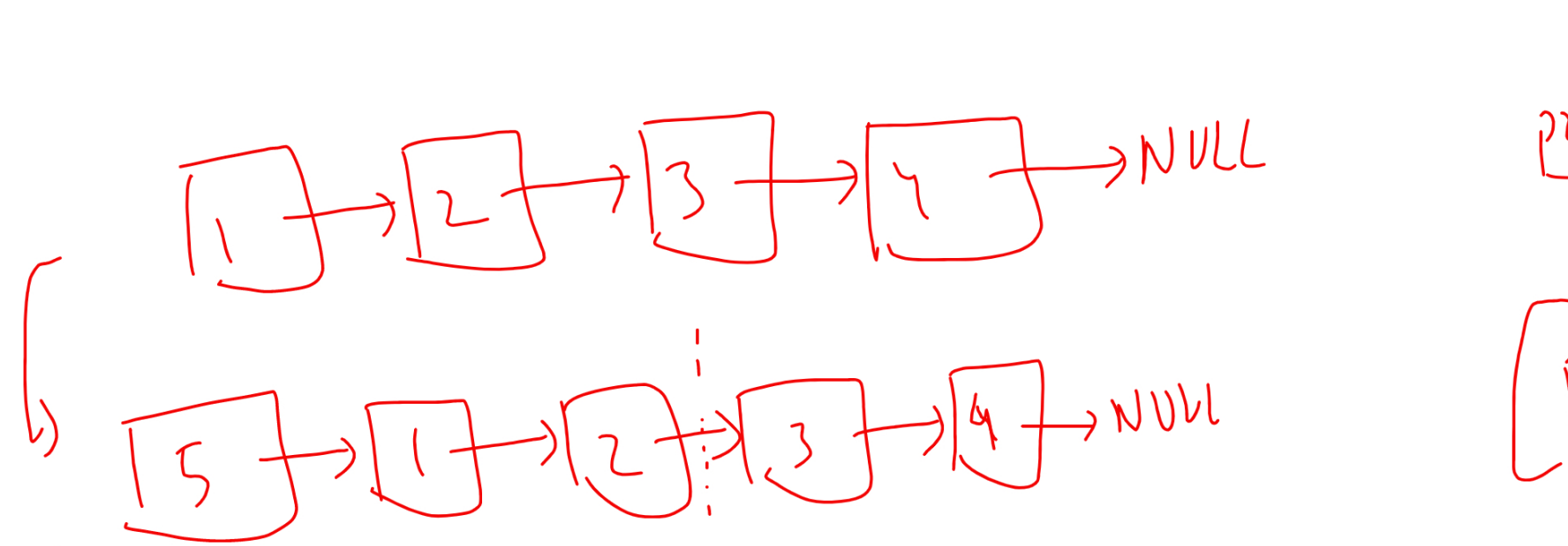
```
while (cur→next != NULL) {
    cur = cur→next;
}
cur→next = new Node(d);
```

head = NULL



new_node
d
NULL
head

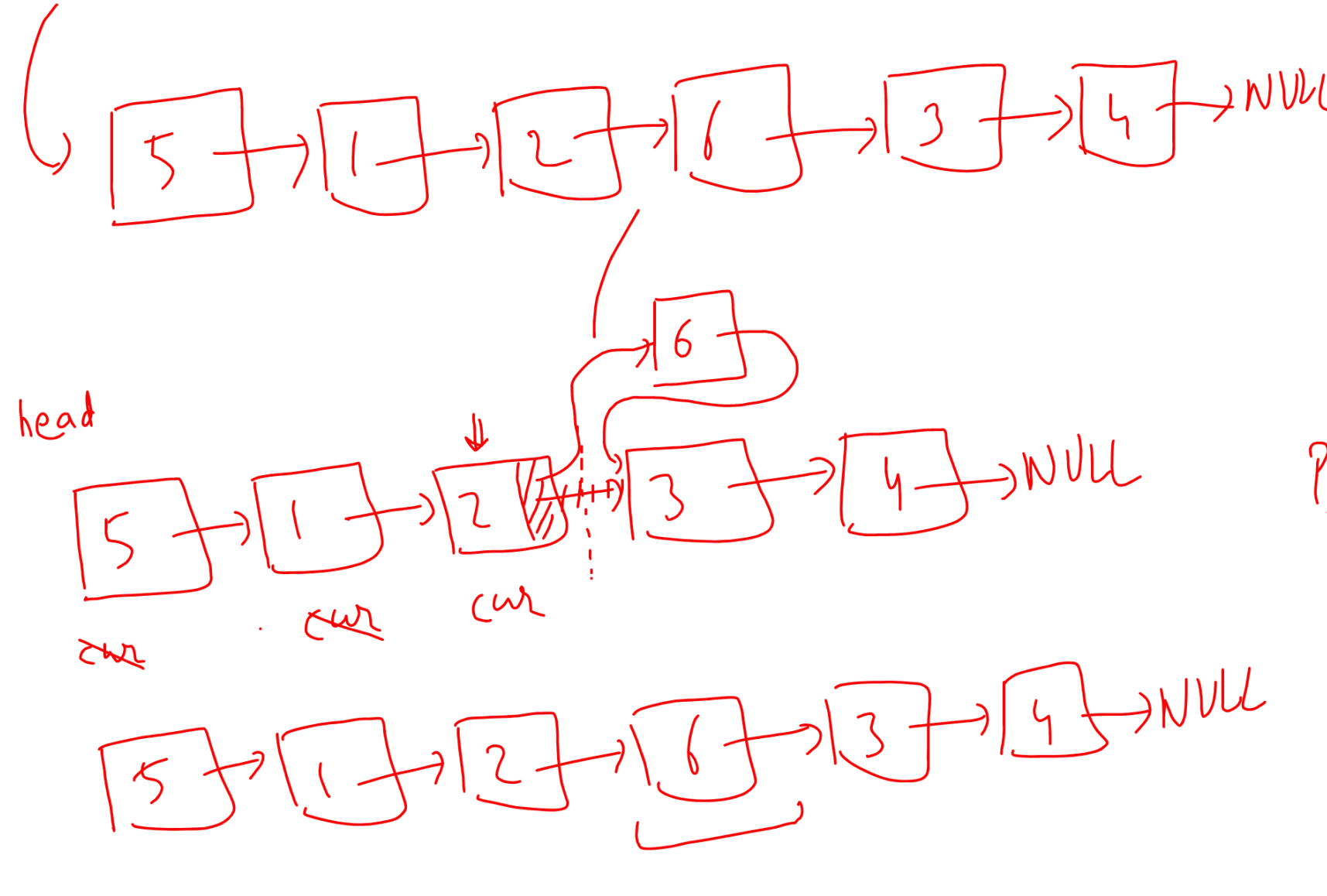




1
2
3
4
NULL
5
1
2
3
4
NULL
5
1
2
6
3
4
NULL

pos = 0, d=5

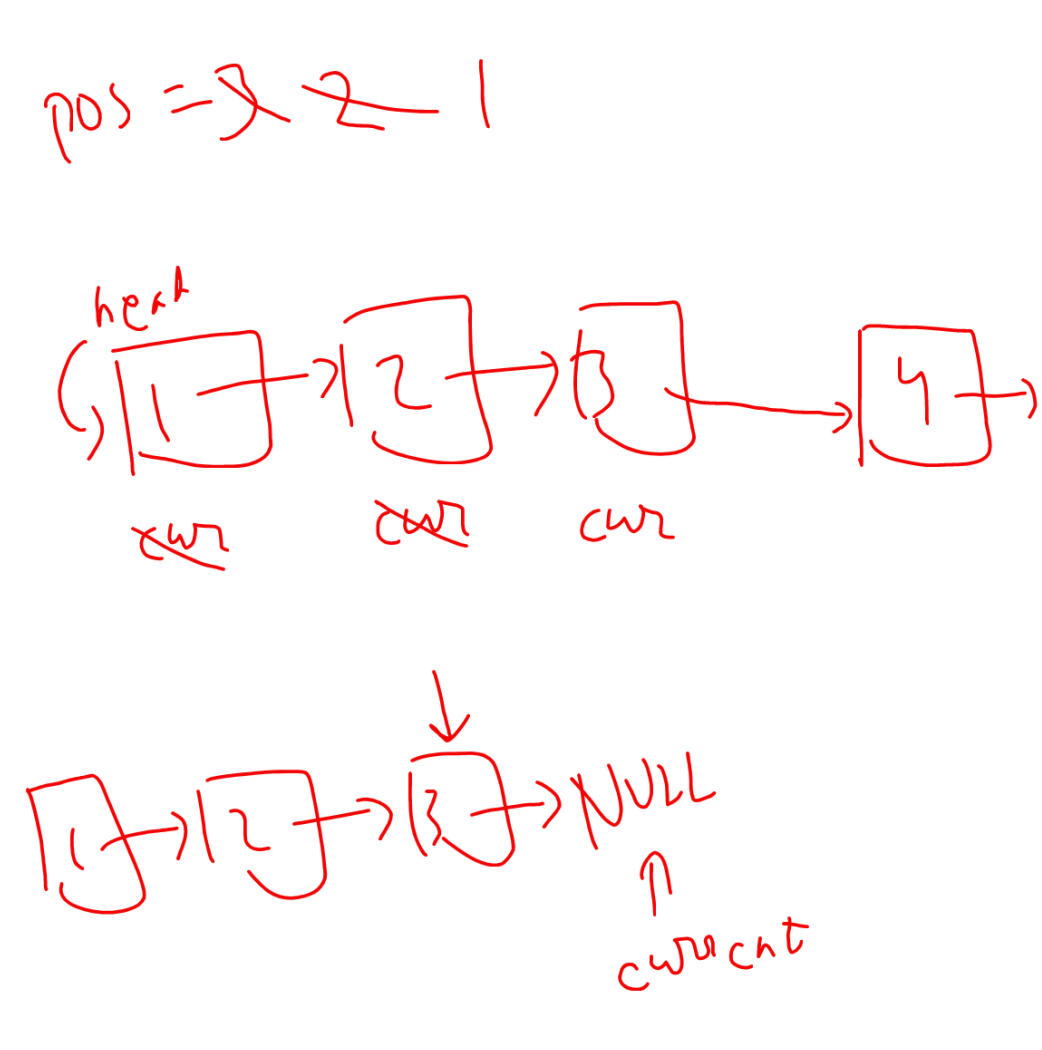
[pos = 3, d=6]



head
6
5
1
2
3
4
NULL
cur
cur
cur
5
1
2
6
3
4
NULL

pos = 3, d = 6

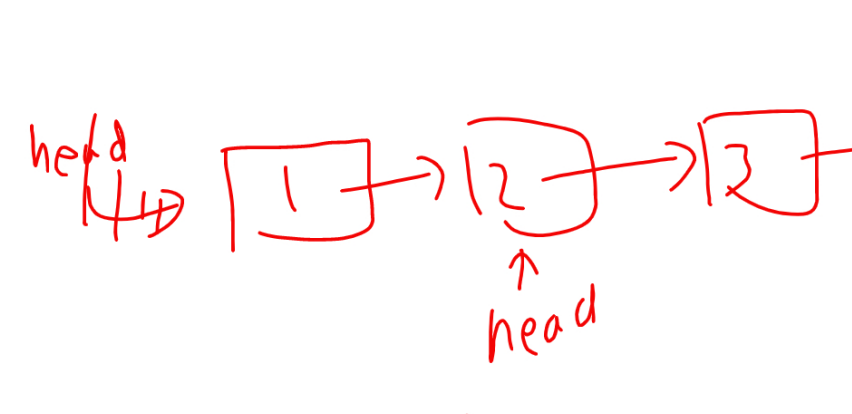
pos = ~~3~~ ~~2~~ 1



head
1
2
3
4
NULL
cur
cur
cur
1
2
3
NULL
current

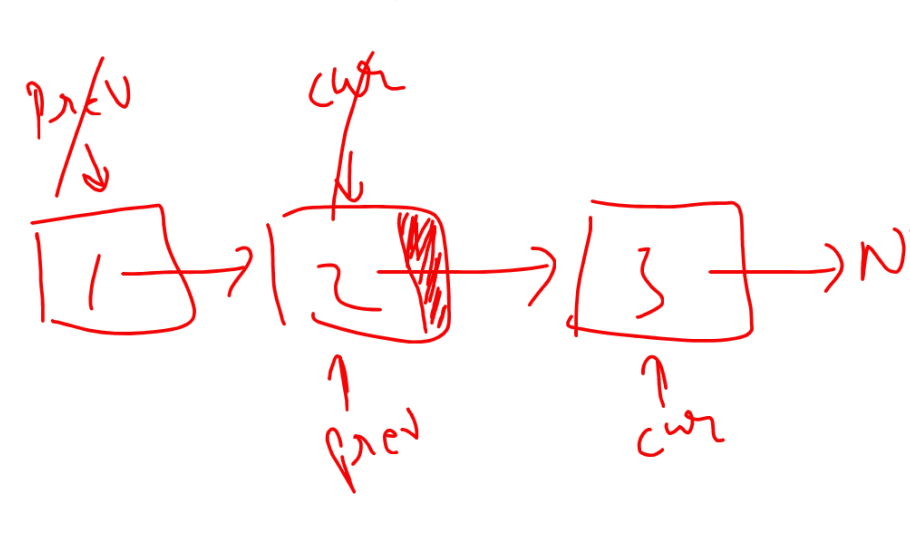
pos = ~~3~~ d = 4
1

pos = 4



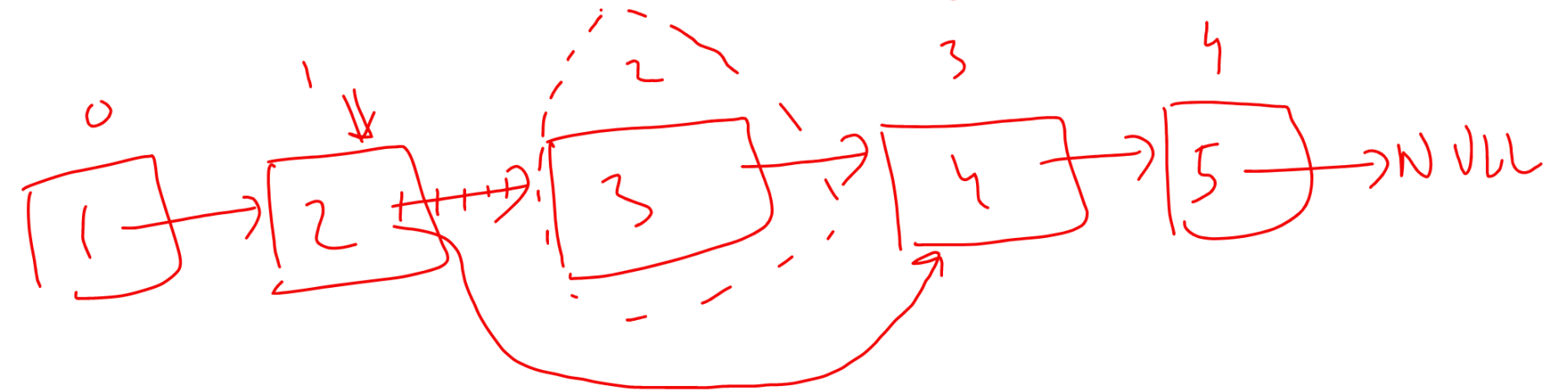
head
1
2
2
NULL
head

head = head→next.



prev
cur
1
2
3
NULL
prev
cur

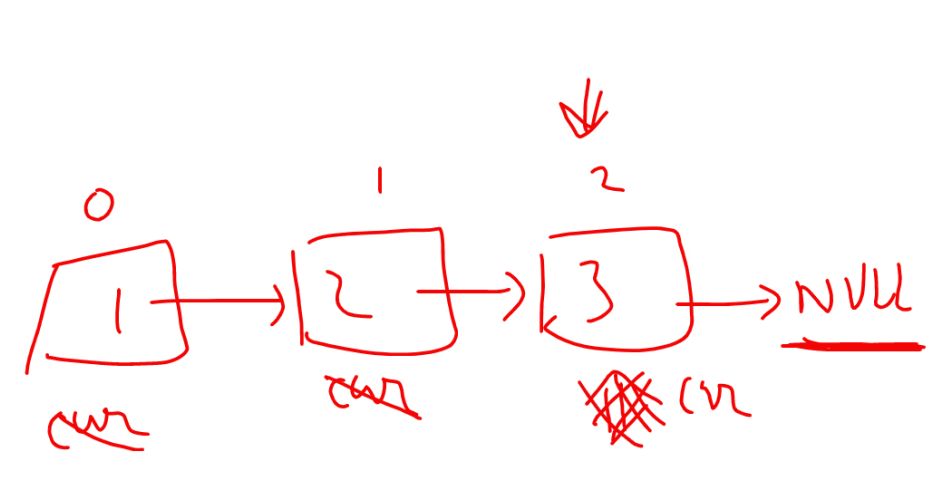
while (cur→next→next != NULL)



0
1
2
3
4
1
2
3
5
NULL

pos = 2

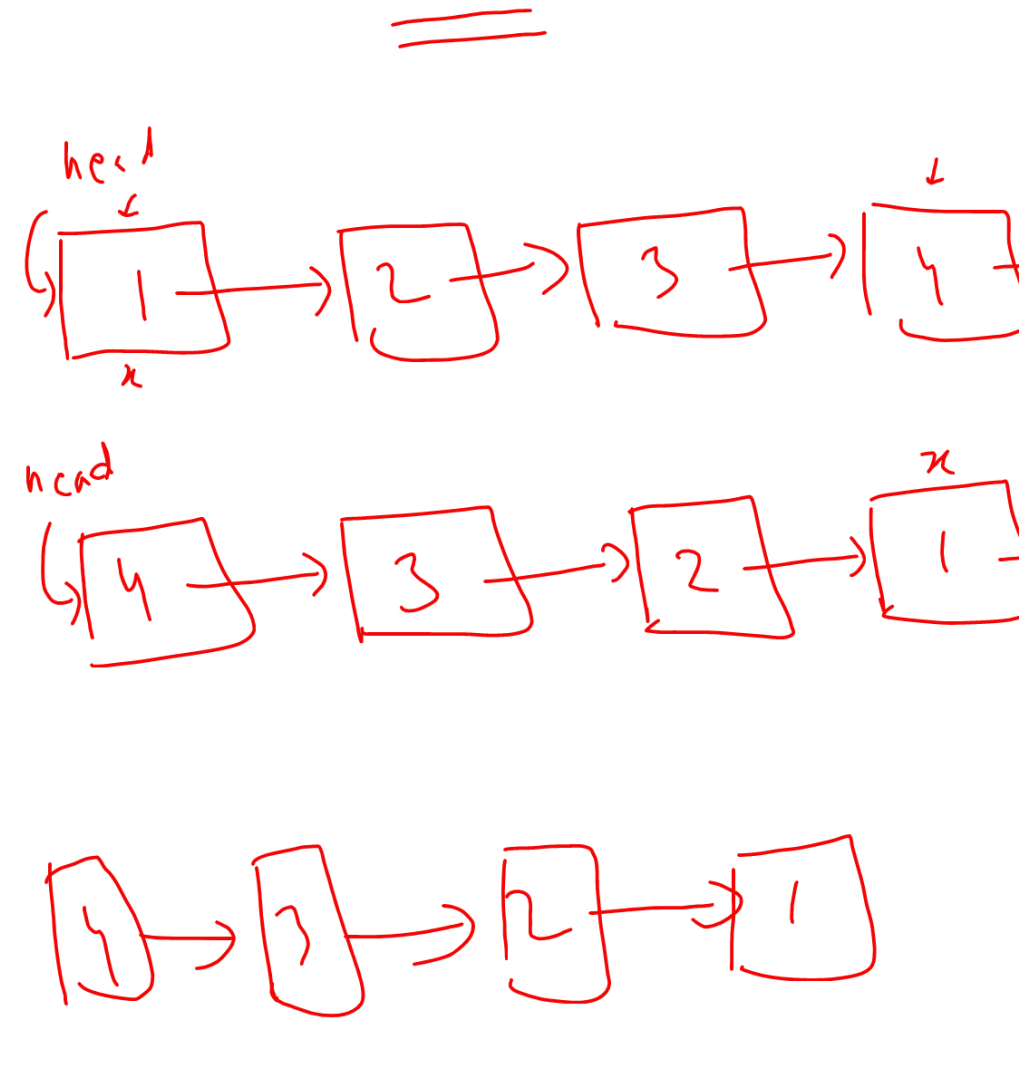
cur = cur→next



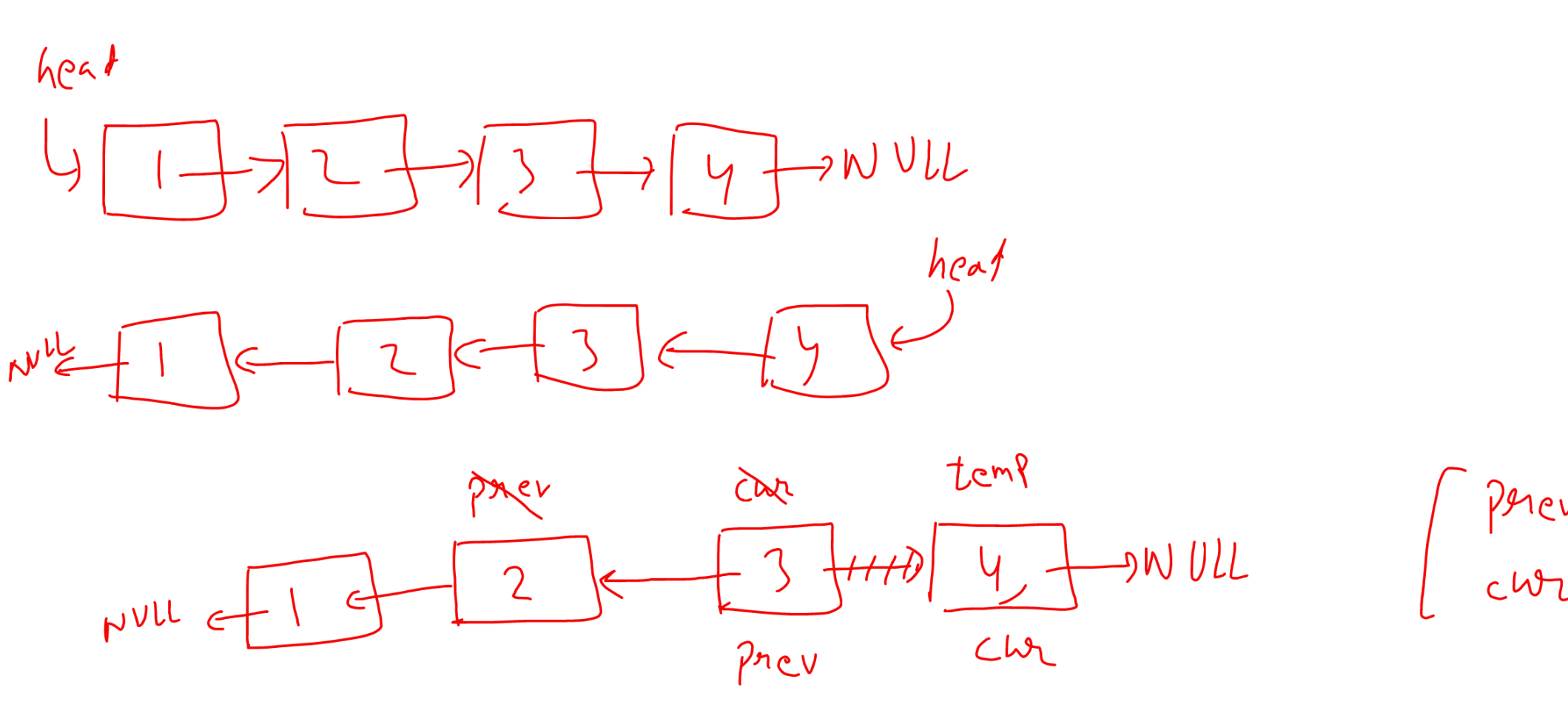
0
1
2
1
2
3
NULL
cur
cur
cur

pos = ~~3~~ ~~2~~ 1

[pos > 2]
don't delete any node



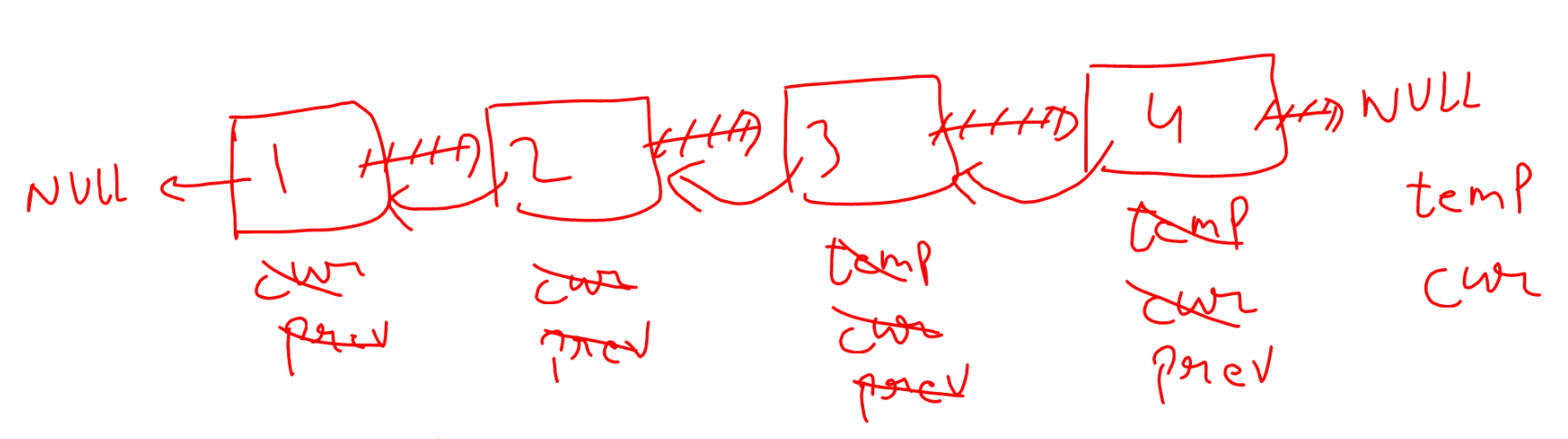
head
1
2
3
1
head
4
3
2
1
1
3
2
1



head
1
2
3
4
NULL
NULL
1
2
3
4
head
prev
cur
NULL
1
2
3
4
NULL
temp
prev
cur

[prev cur]

```
Node* temp = cur→next
cur→next = prev
prev = cur
→ cur = temp
```

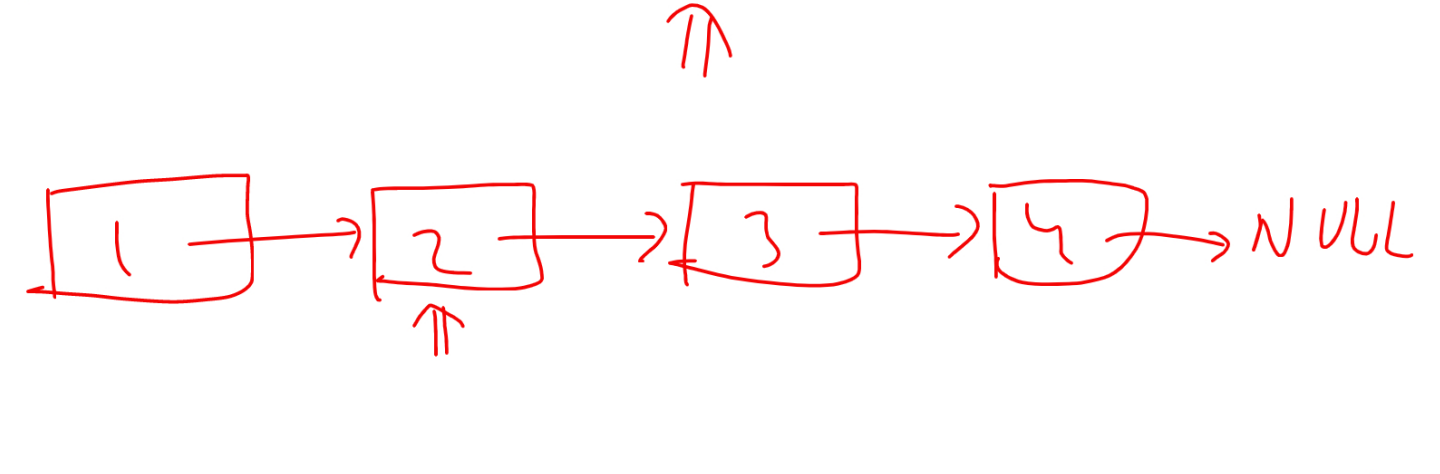


NULL
1
2
3
4
NULL
cur
prev
cur
prev
temp
cur
prev
temp
cur

prev = ~~NULL~~ 1
cur = ~~x~~ 2
temp = 2

```
while (cur != NULL) {
    ------
}
return prev;
```

MIDDLE

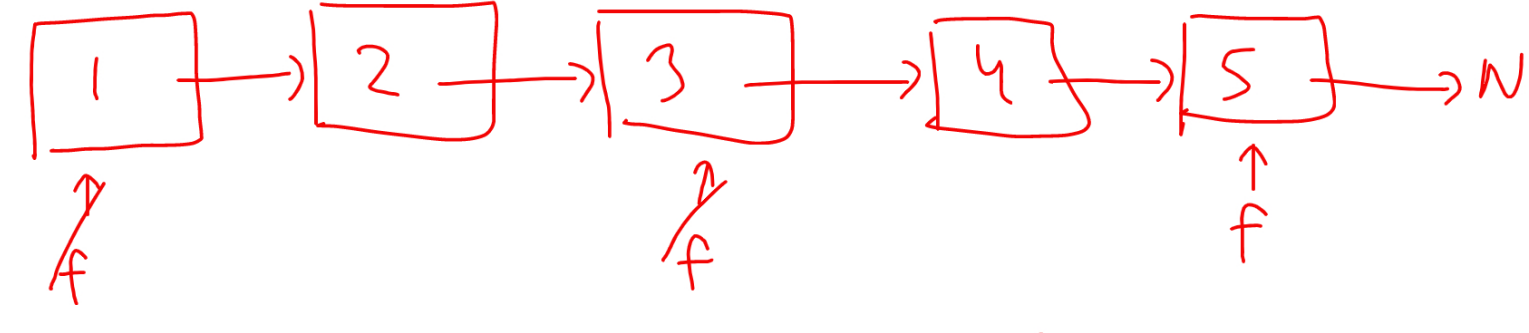


1
2
3
4
5
NULL
1
2
3
4
NULL

#1
→ Traverse & count the nodes.
→ Traverse again & return $(\frac{count}{2})^{th}$ node.

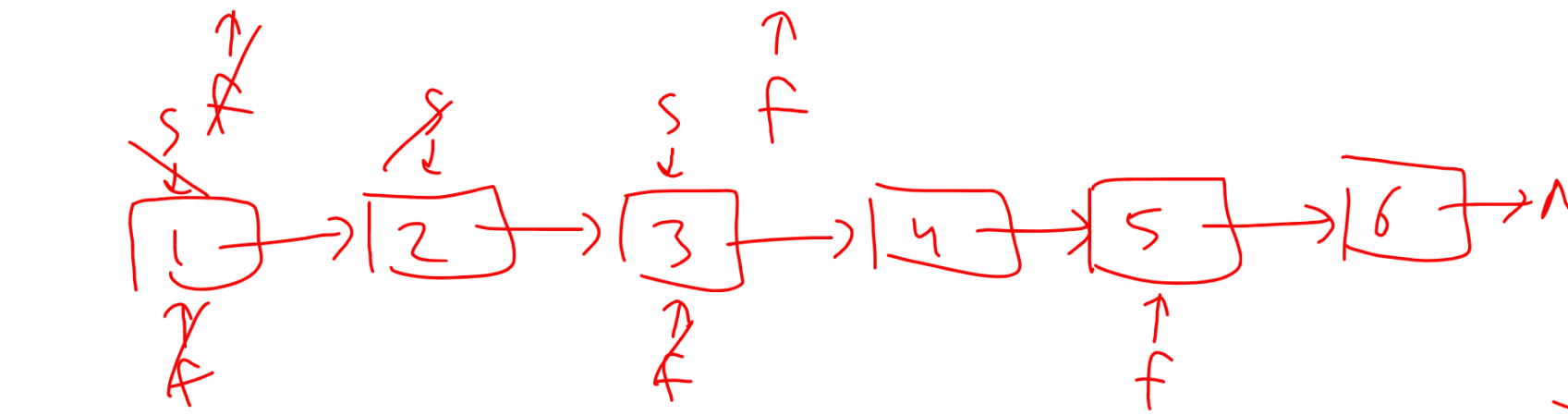
TC: O(n)

#2 Two-pointer approach:



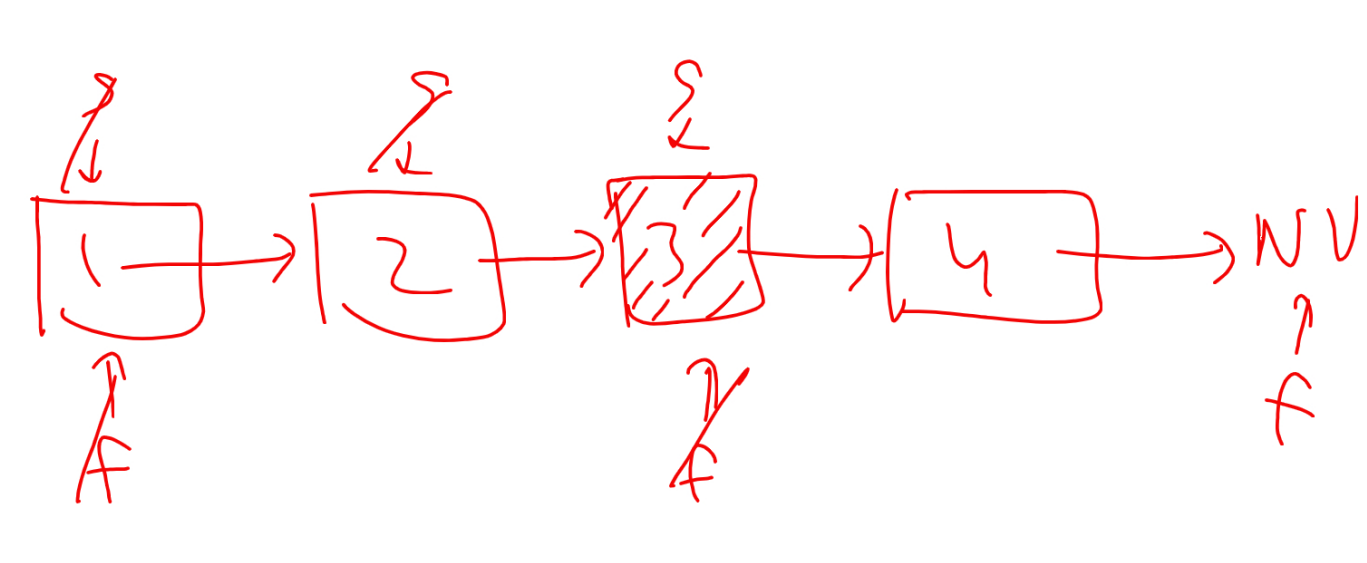
s
s
s
1
2
3
4
5
NULL
f
f
f

slow: moves one step at a time. x
fast: moves two steps at a time. 2x

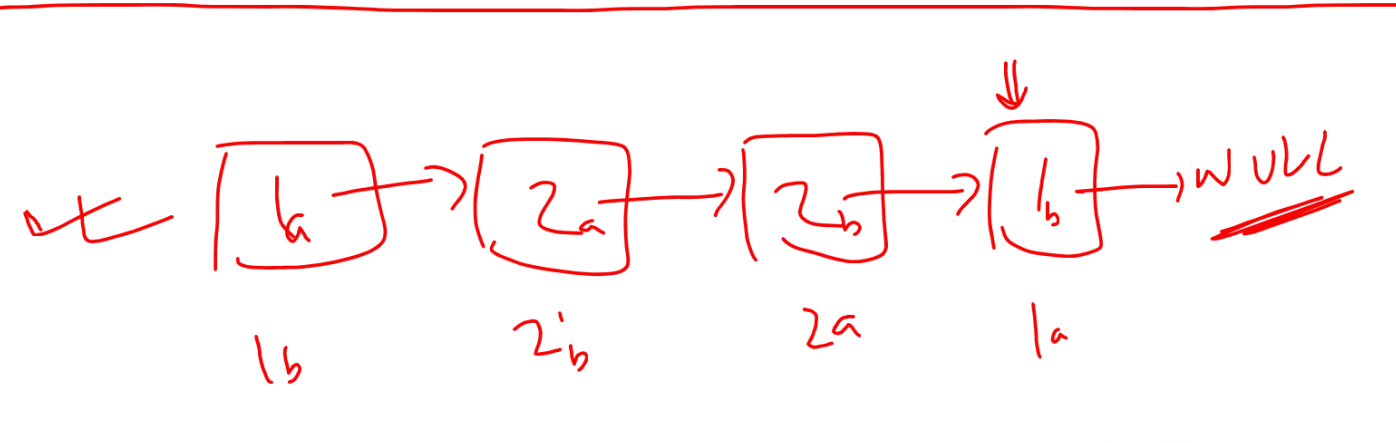


s
s
1
2
3
4
NULL
f
f
s
s
s
1
2
3
4
5
6
NULL
f
f

Stopping condition 1) fast→next == NULL
2) fast→next→next == NULL

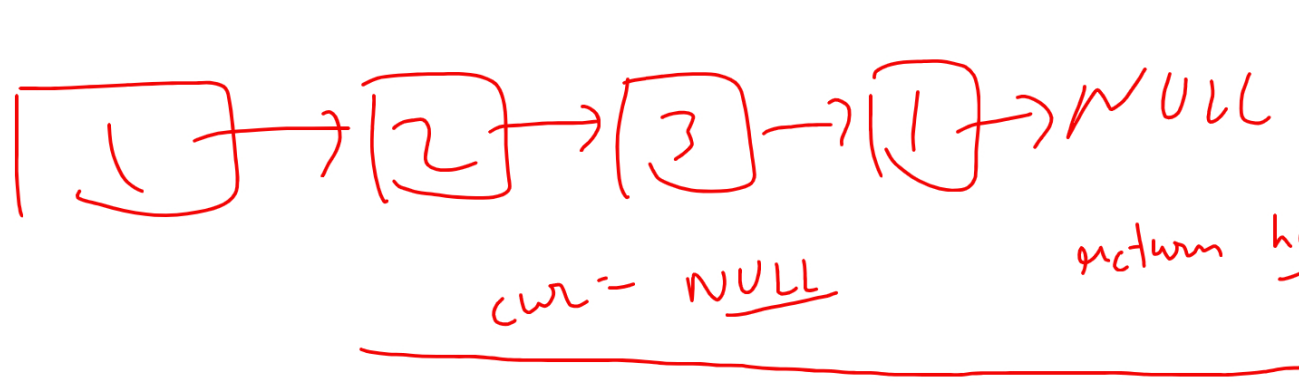


s
s
s
1
2
3
4
NULL
f
f
f



1b
2b
2a
1a
NULL
1b
2b
2a
1a

Palindrome.



NOT a Palindrome.

return head

cur = NULL

cur: 1b returned_node = 1a, return (returned_node→next)

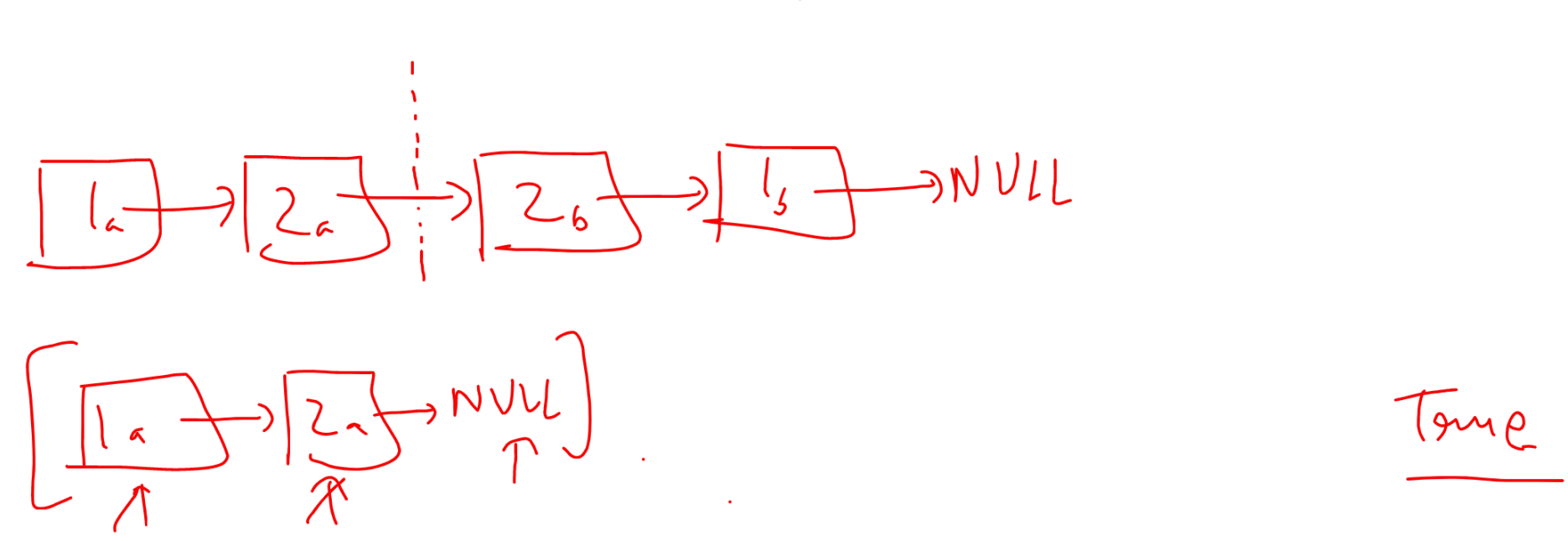
cur: 2b returned_node = 2a, return 2b

Rec. stack

cur: 2a ret_node = 2b, return 1b

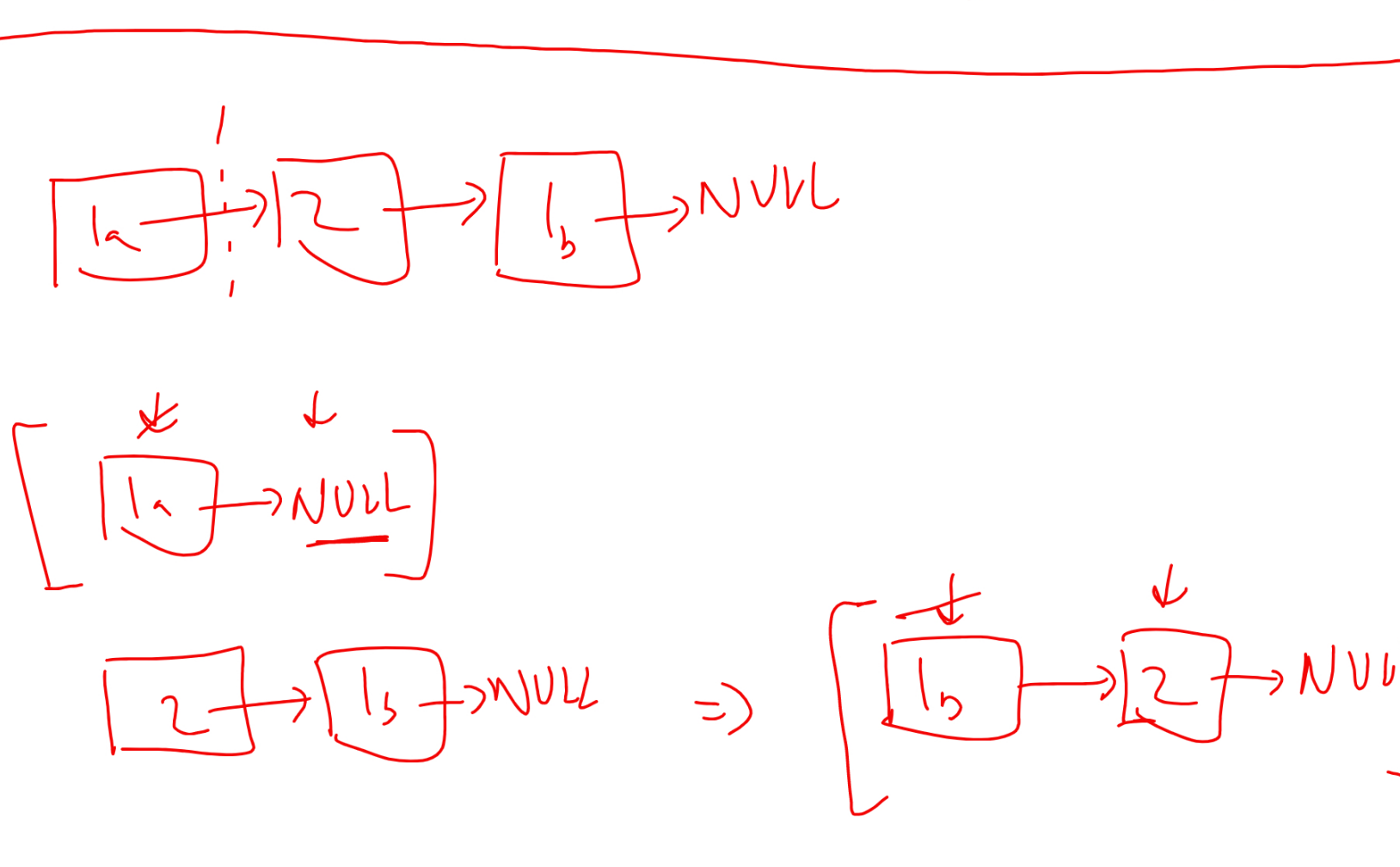
cur: 1a ret_node = 1b, return NULL

return true.



1a
2a
2b
1b
NULL
1a
2a
NULL
2b
1b
NULL
1b
2b
NULL

True



1a
2
1b
NULL
1a
NULL
2
1b
NULL
1b
2
NULL

True

Step-1: Find the middle node & split the list into two halves.

Step-2: Reverse the second half.

Step-3: Compare the 2 halves till at least one of the halves get exhausted.