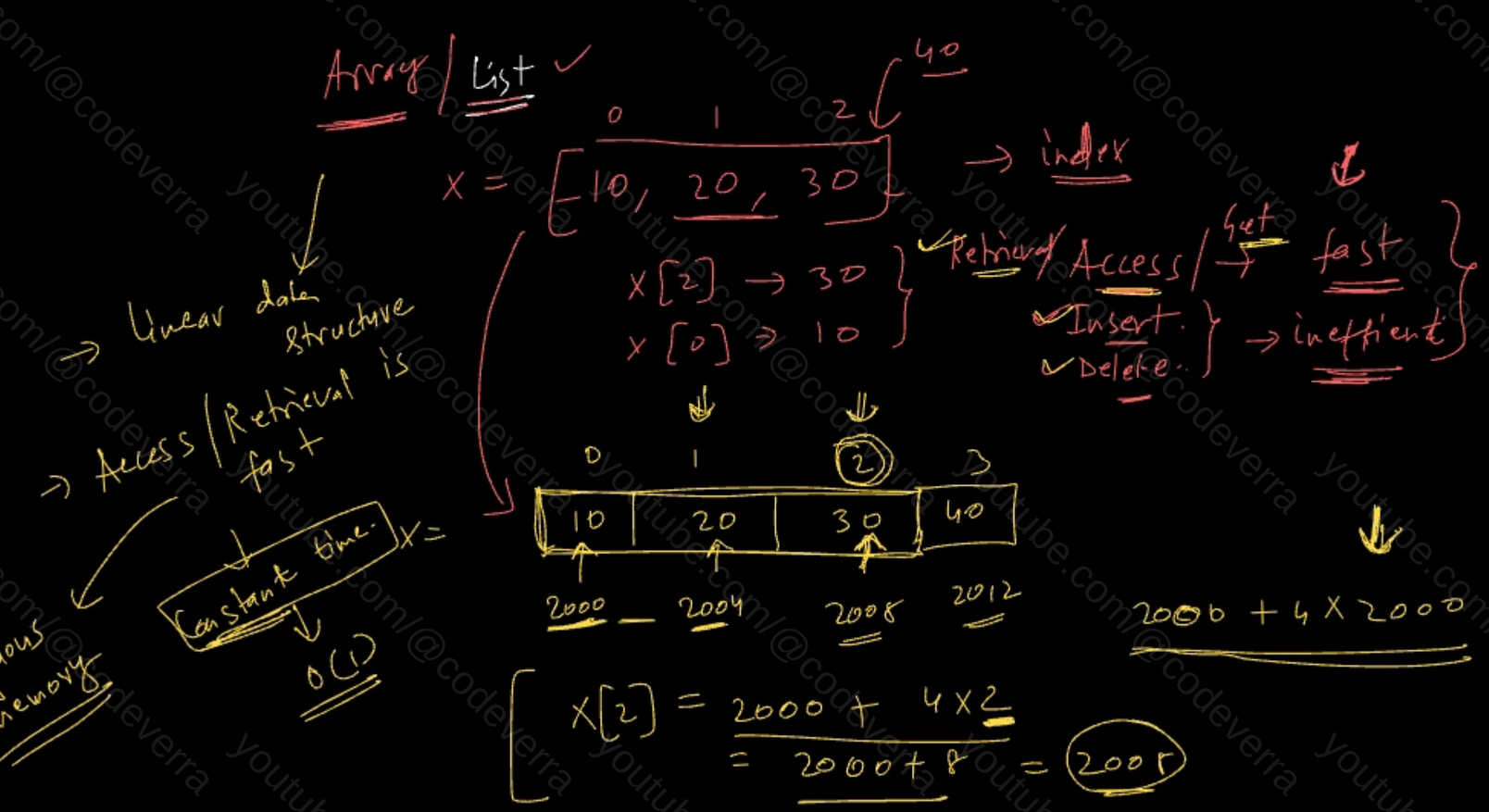
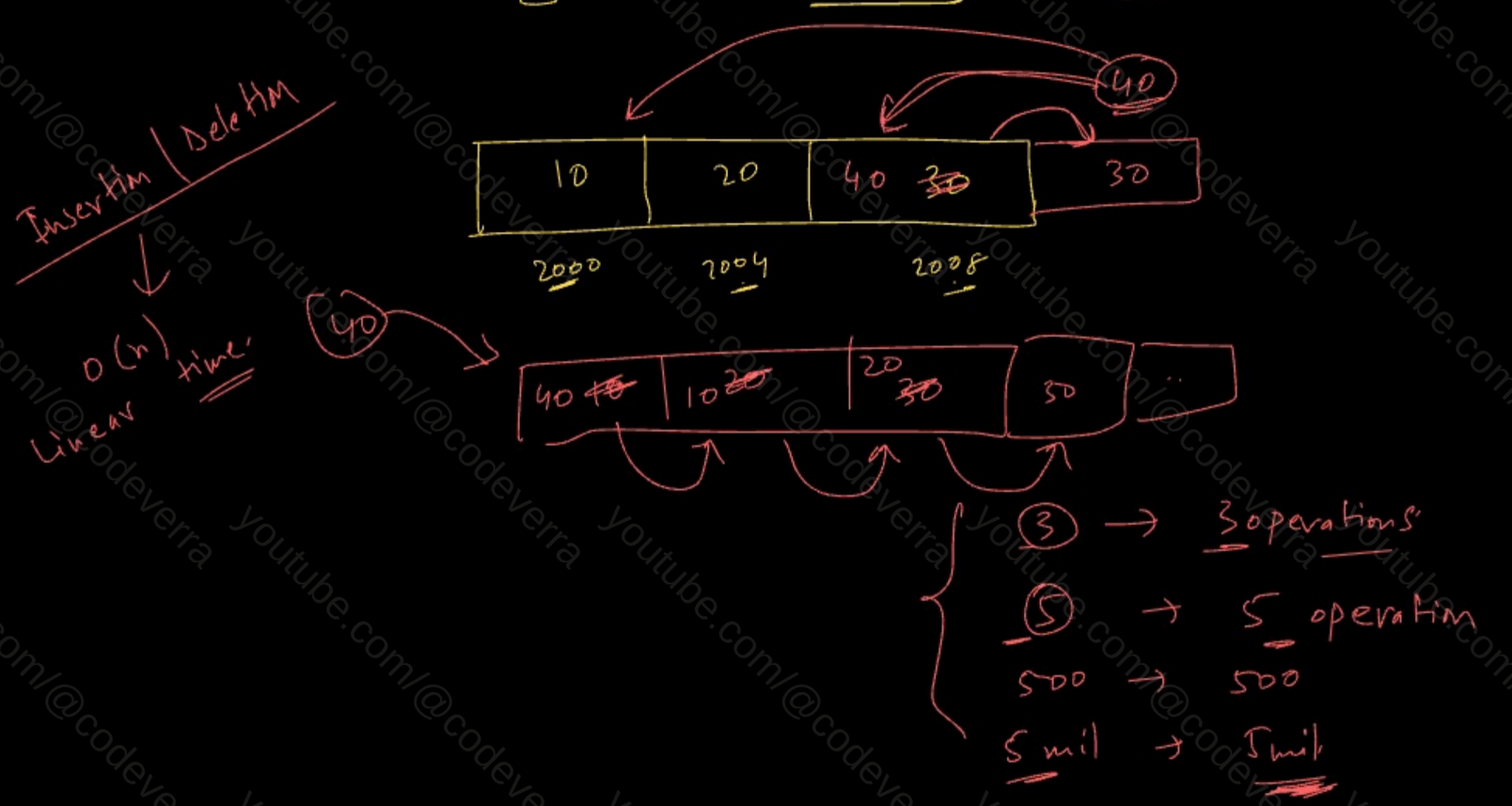
Linked List

Agenda:

1. What is a linked list?
2. Comparison between a linked list and a list(array)
3. Common operations on a linked list
4. Python implementation of a linked list
5. Common interview questions
6. Coding interview questions



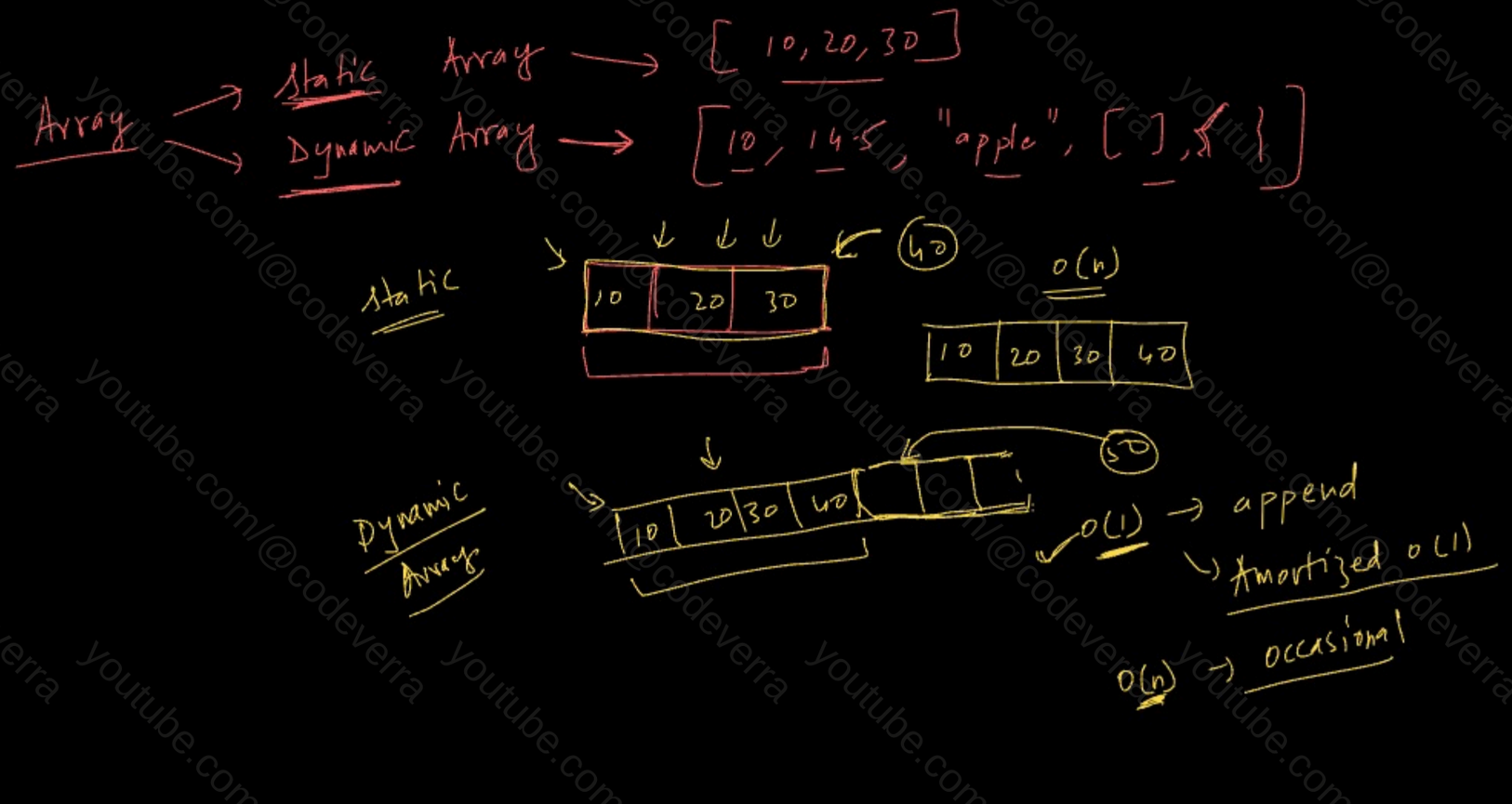


List is linear data structure (sequential arrangement). We can access elements using the index in constant time.
Data is stored in contiguous memory block.

Two types of list : Static array and dynamic array

Common operations on a list:

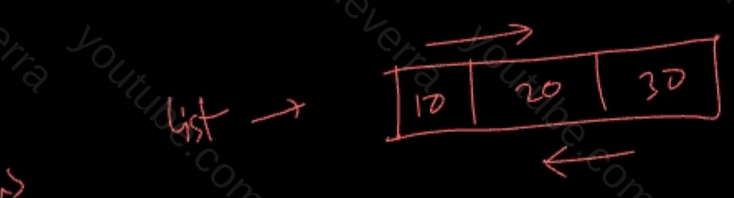
- Accessing a value is very efficient (Constant time O(1)) - because of index.
  Position of any element can be calculated from the first element.
- Inserting or deleting is inefficient (linear time O(n)) - because elements have to be shifted to accomodate new member.

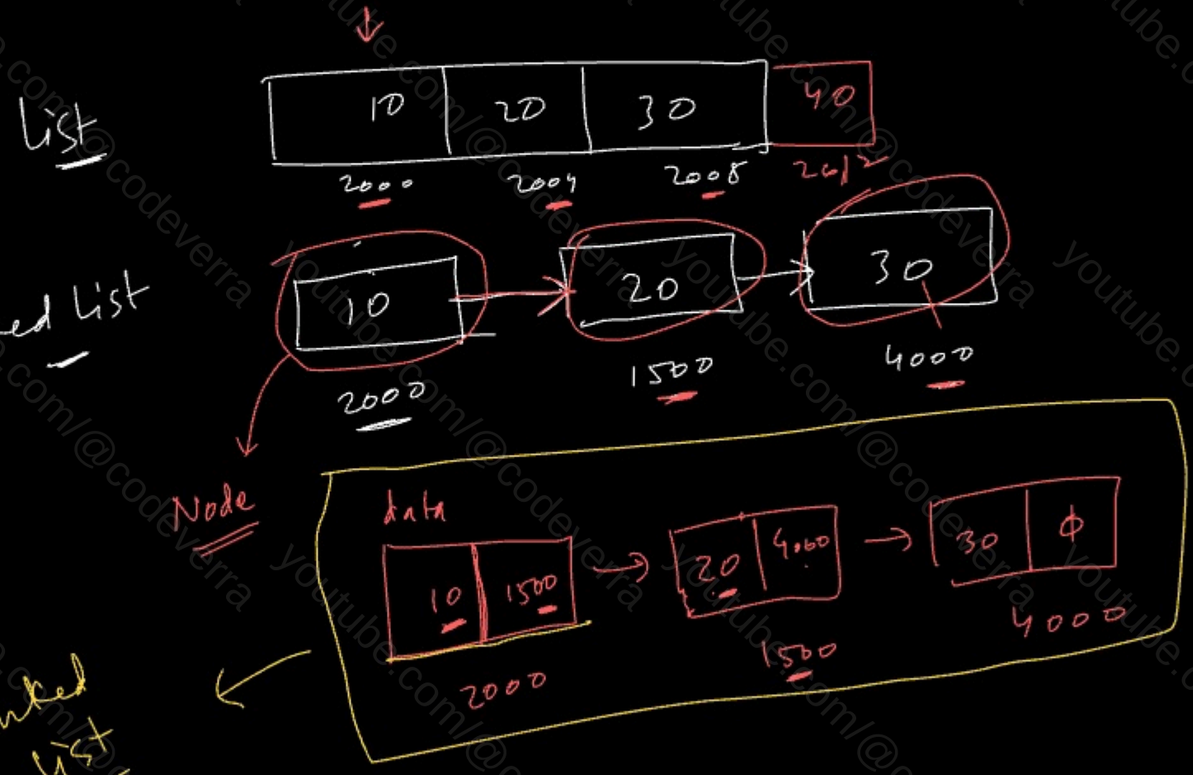


Linked List

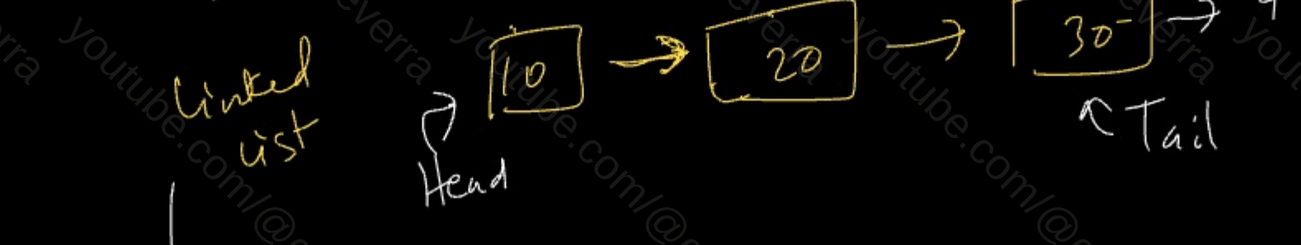
→ Linear data structure
→ sequential arrangement
→ Collection of items

Linear { List, Linked list, Stack, queue.
non Linear { + Graphs, Heap, Tree, Hashmap



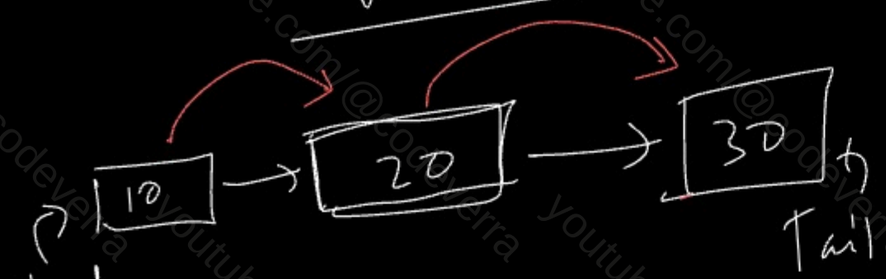


Linked List

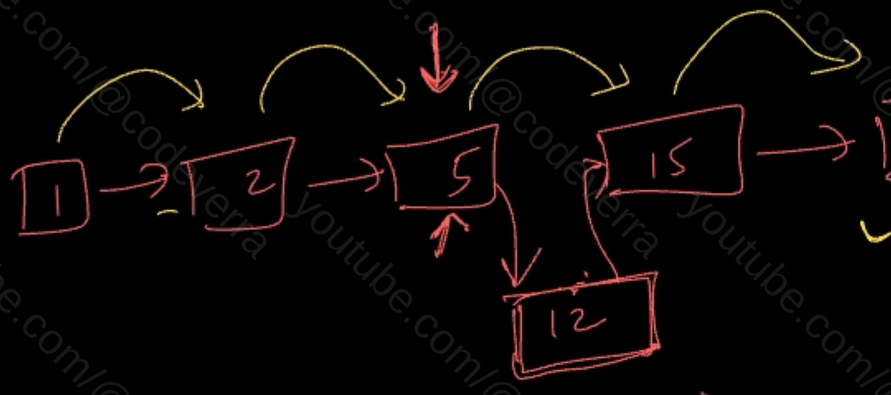


Linked List → stored in scattered way
↳ NOT contiguous
→ Insertion is efficient
→ Accessing is slow

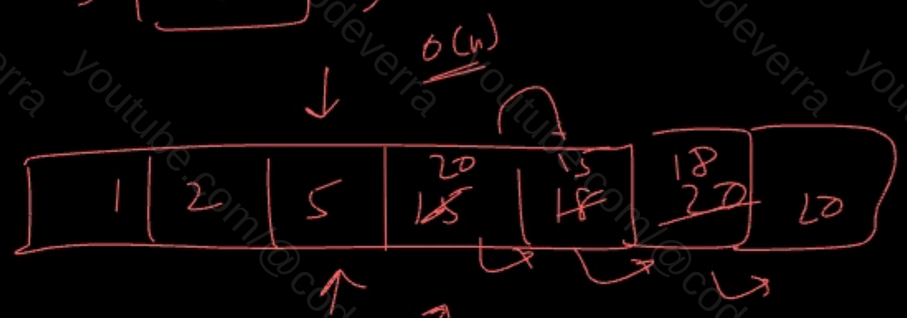
Linked List



① Get → O(n) → Linear time.
② Insertion → O(1) → if reference is known



Insertion (if reference of position is known) → O(1) → Constant time



List

① Contiguous memory

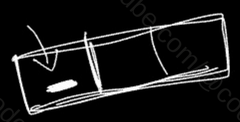
② Accessing is fast → O(1)

③ Insertion / Deletion usually slow

④ Extra memory is allocated for dynamic arrays

actual memory > number of element

⑤ data



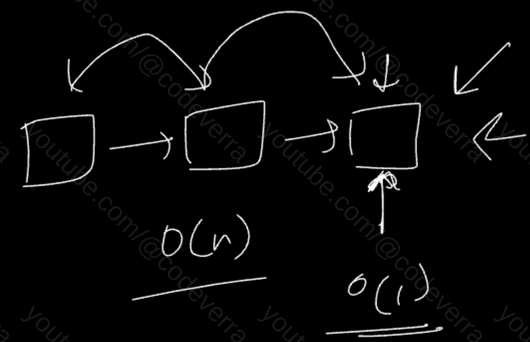
⑥ List is cache friendly → Contiguous memory

Linked List

① Scattered / distributed

② Accessing is slow → O(n)

③ Insertion / Deletion is efficient (if reference is known)



④ flexible. We can change the size of LL anytime.

⑤ Node → data, reference to next

Lightweight data type

} x element
2x memory
3x → in case of DLL
} Space efficiency is low

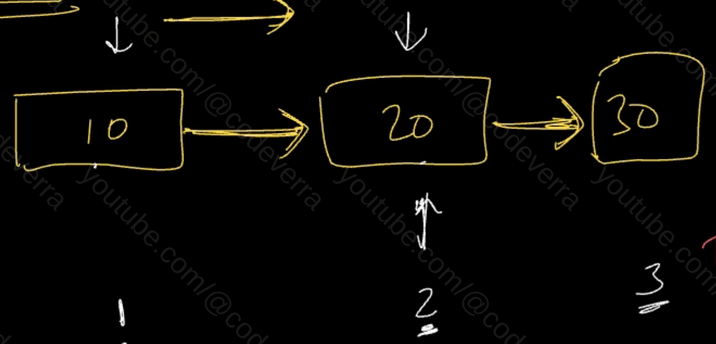
Efficiency / Complexity / Expense } → Time, Space

⑥ Not suitable for CPU caching

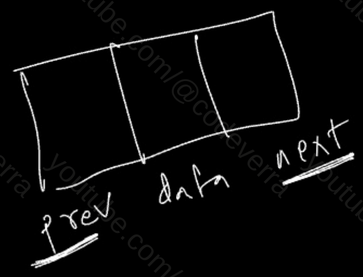
↳ because not contiguous

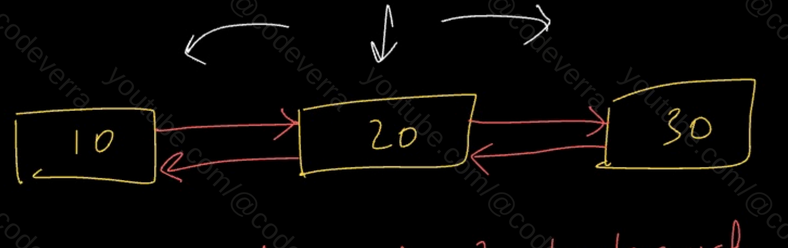
Types of Linked List

① Singly Linked List

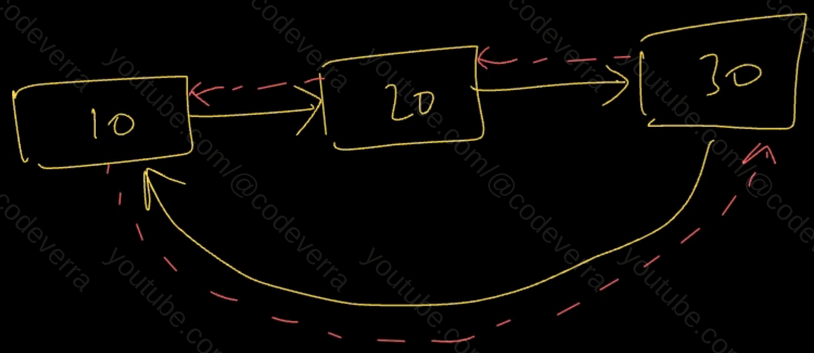


② Doubly Linked List





③ Circular Linked List



Python Implemation of LL

oops

{ class
object
attributes & methods }

[1, 10, [10, 20]]

10 → 20 → 30

class
Node → object → data
→ reference to next node

class
LinkedList → Head → Tail

Linked List

node1 → 10 → node2 → 20 → node3 → 30 → None (0)

Head

Common Operations on Linked List

1. Traverse → Visualize a Linked List
→ Check if a element is present in list (LL) (membership check)
→ Check the length of LL

Head → 10 → 20 → 30 → 40 → None → $ (None)

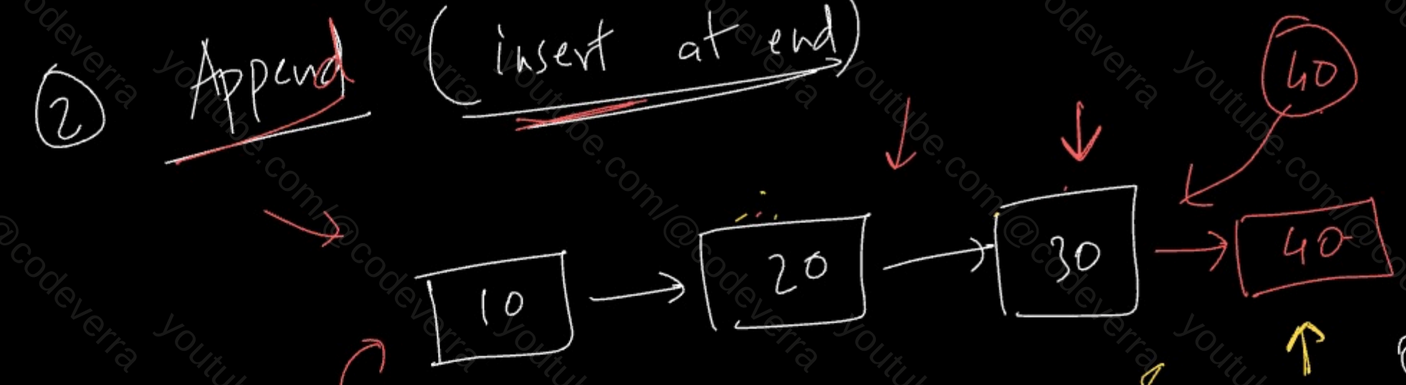
Visualise → [10, 20, 30, 40]

While loop

Traversal algorithm ←

```
Curr = head
while Curr:
    print(Curr.data)
    Curr = Curr.next
```

② Append (insert at end)




```
curr = head
while curr.next
    curr = curr.next
curr.next = value
```

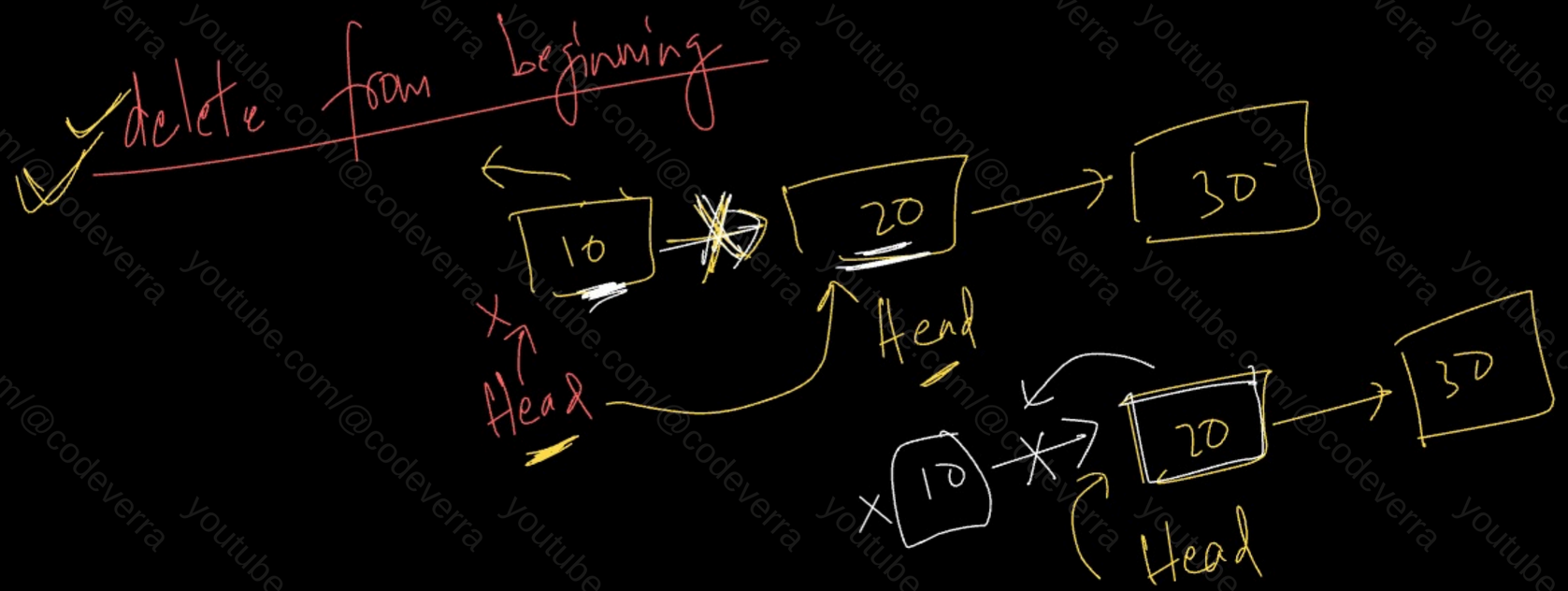
① Find the last node

② point the next to the given value

Algorithm to append to Linked List

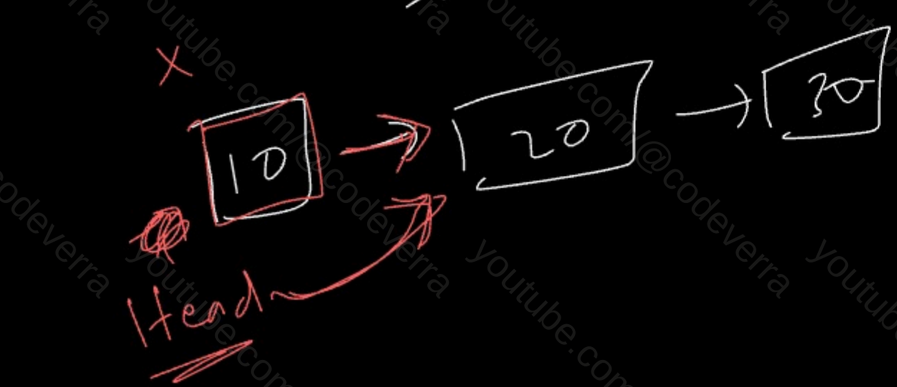
③ Delete

- delete from beginning
- delete at any position
- delete at the end

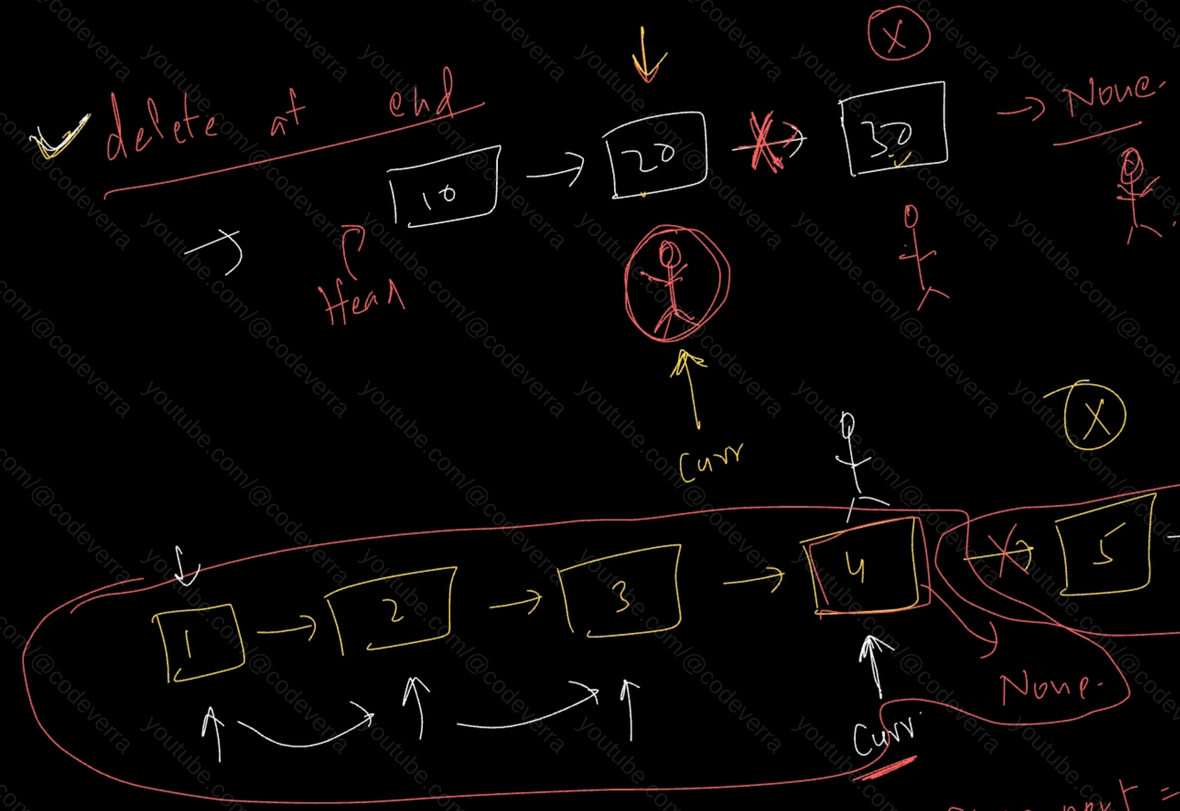


Algorithm

```
curr = head
head = head.next
curr.next = None
```



delete at end

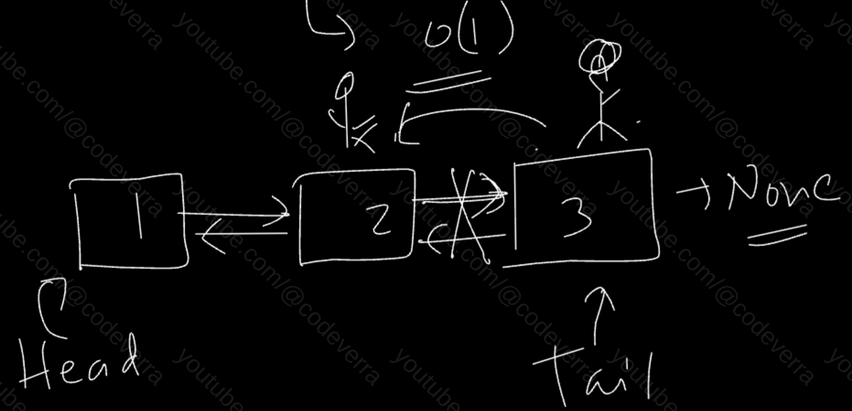


Algorithm to delete from end

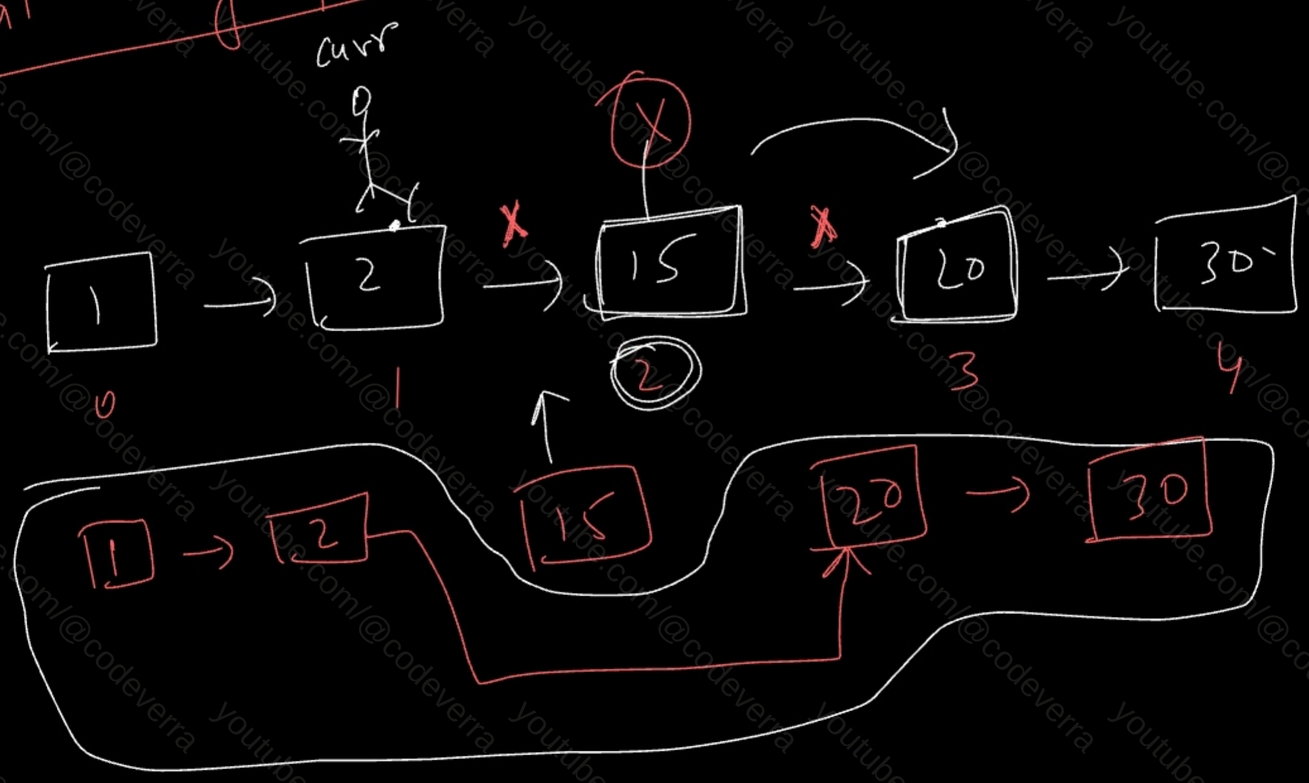
```
curr = head
while curr.next.next :
    curr = curr.next
curr.next = None
```

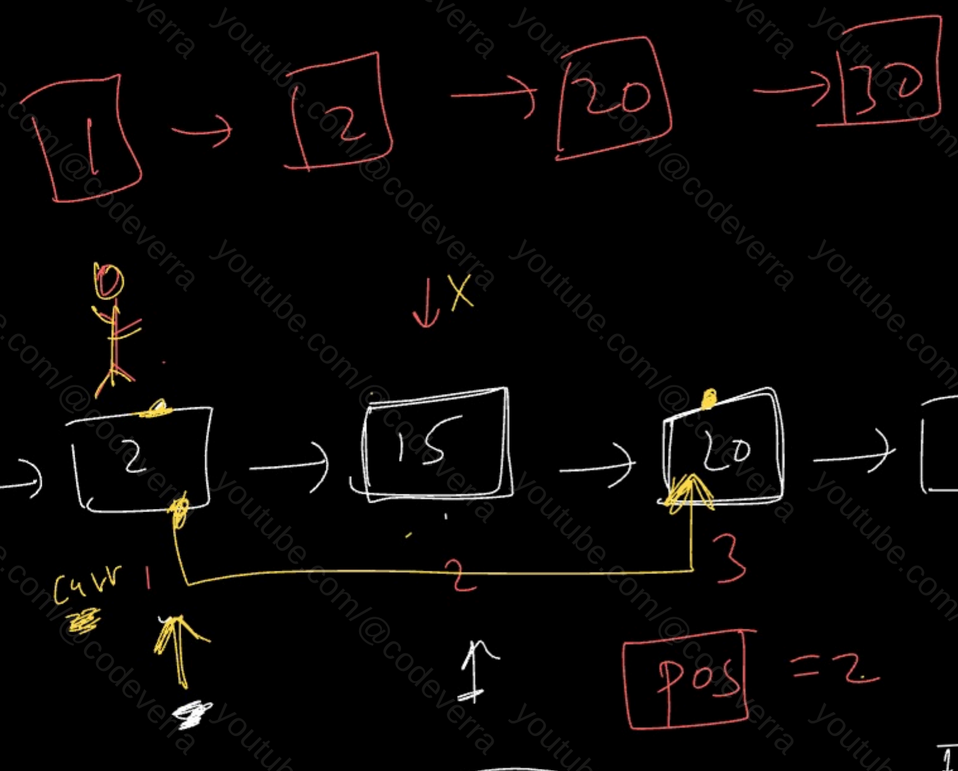
Doubly linked list

→ Tail is known.



delete at any position





Edge Cases

- head is one
- there is only one element
- index invalid

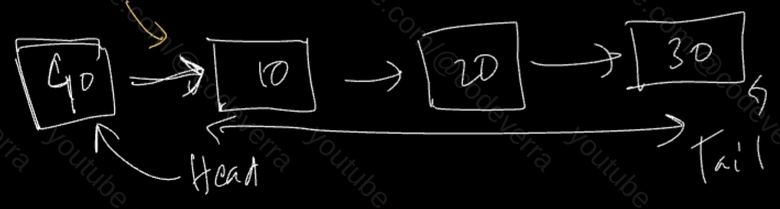
Algorithm to delete @ any position

- curr = head
- for _ in range(pos-1):
  - curr = curr.next
- curr.next = curr.next.next #Variable

④ Insertion

- at start
- at any position
- at end (append)

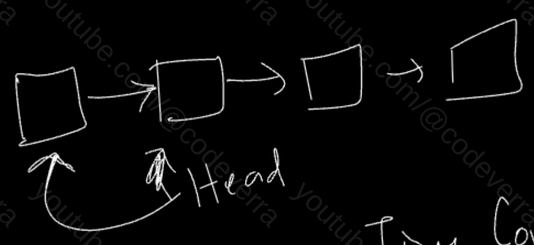
Insert at start

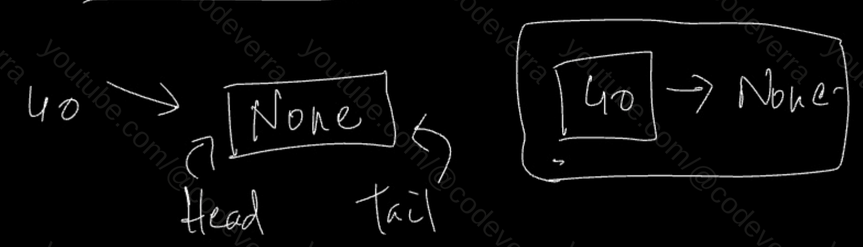



```
node.next = self.head
self.head = node
```

Work even on empty LL

edge case → Linked List is empty

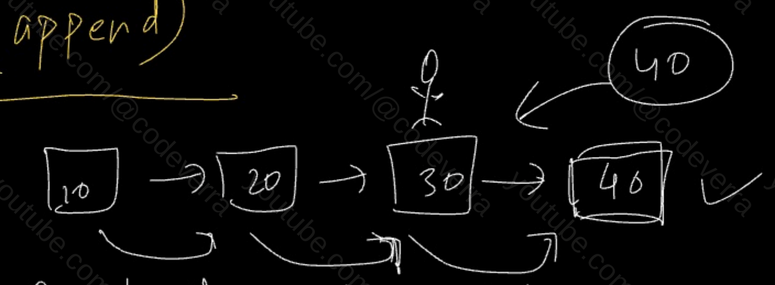




Time Complexity: O(1)
Constant Time

→ self.head = node

insert at end (append)



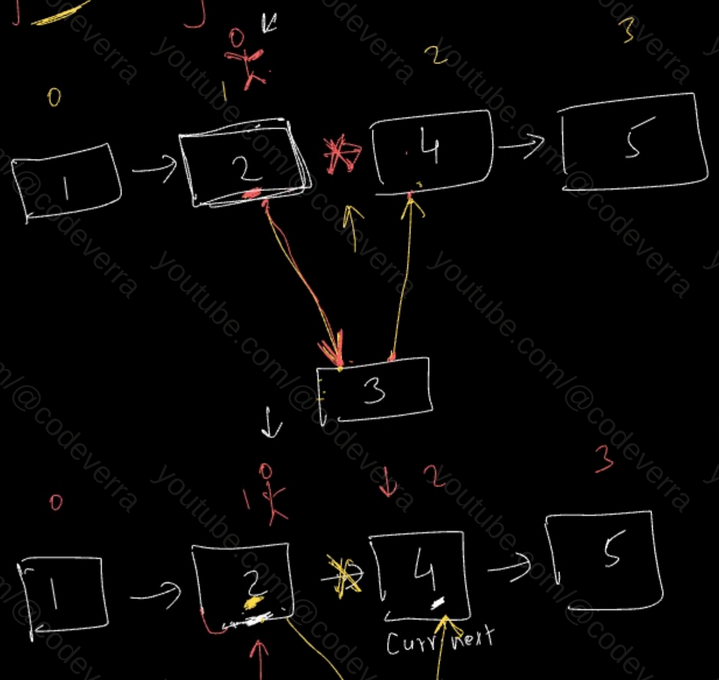
Time Complexity → O(n)

Insert at any position

- Time Complexity
  - O(n)
  - O(1) if reference to node is known

- High level
- Code
- Dry run
- Edge Cases

→ Boundary Conditions



pos → 2
value → 3

node = Node(3)

range(1) → [0]

range(4) → [0,1,2,3]

Code

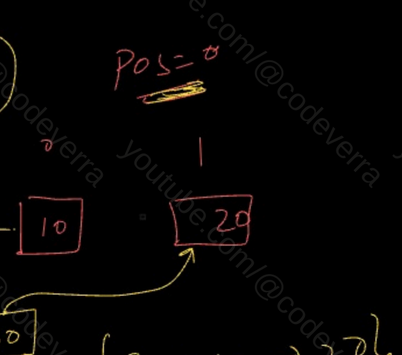
```
Curr = head
for _ in range(pos-1):
    curr = curr.next

node.next = curr.next
curr.next = node
```

Algorithm to insert node at any position

not applicable

pos = 0

insert-at-beginning



size = 3

pos = 0, 1, 2

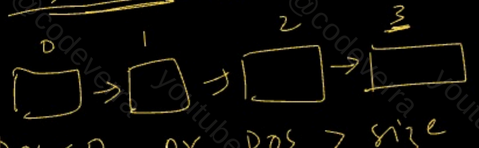
Invalid ← pos < 0 or pos ≥ size

edge cases

pos = 0

if list is empty → ? (Not an edge case) ✗

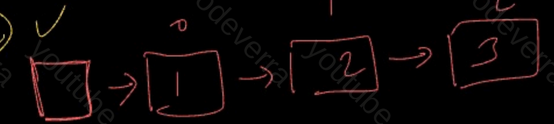
pos should be valid ①

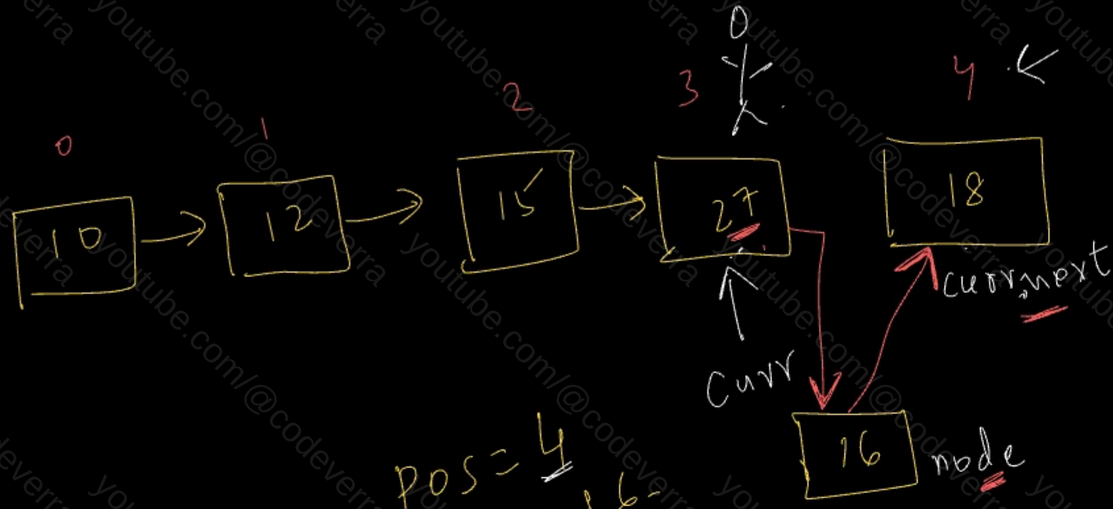


pos < 0 or pos > size

↳ invalid position

pos = 0 ②





pos = 4
value = 16

node = Node(16)

0 1 2 3 4 5
10 → 12 → 15 → 27 → 16 → 18

insert at any position

```
curr = head
for _ in range(pos-1):
    curr = curr.next

node.next = curr.next
curr.next = node
```

range(4-1)
range(3)
[0, 1, 2]

pos < 0 or pos > size
↓
invalid

size = 0
pos = 0
Val = 20

Head → 20 → None
Tail

node.next = head
head = node

20 → None
Head, Tail

Linked List → insert

→ High level / Code (pseudo)
→ Dry run
→ Edge Case Check
→ Final dry run
→ Boundary cases (Empty, pos=0, end)

Final
- if pos < 0 or pos > size: raise Exception
- if pos == 0: insert_at_beginning
- else: node = Node(val)

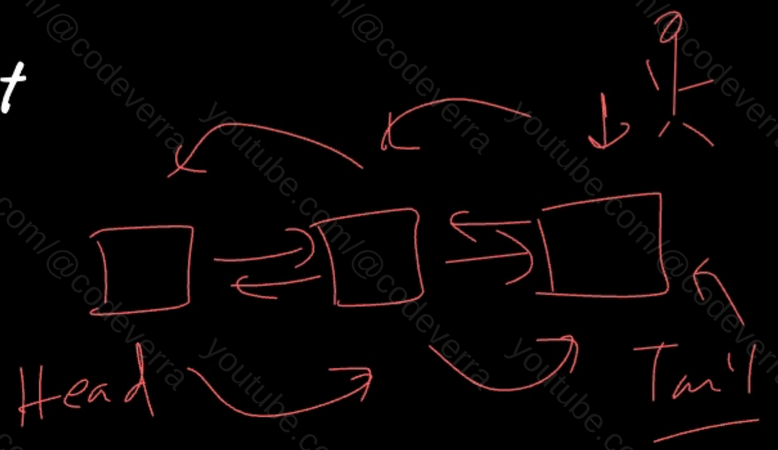
```
curr = head
for _ in range(pos-1):
    curr = curr.next

node.next = curr.next
curr.next = node
```

Summary of common operation on list and linked list

S No	Aspect	Array / List (Dynamic Array)	Linked List (Singly/Doubly)
1	Memory allocation	Contiguous block in memory. Fixed or dynamically resized (reallocation & copying).	Non-contiguous. Each node allocated separately, connected by pointers.
2	Element access	O(1) via index (direct address computation).	O(n) traversal from head (no direct indexing).
3	Insertion at beginning	O(n) (shift all elements).	O(1) (just rewire head pointer).
4	Insertion at end	Amortized O(1) (if capacity available), else O(n) on resize.	O(1) if tail pointer maintained, else O(n) traversal.
5	Insertion in middle	O(n) (shift elements).	O(n) (traverse to position), but actual link update O(1).
6	Deletion at beginning	O(n) (shift all elements left).	O(1) (move head pointer).
7	Deletion at end	Amortized O(1).	O(n) for singly list (traverse to node before last), O(1) for doubly with tail pointer.
8	Deletion in middle	O(n) (shift elements).	O(n) to reach node, O(1) to unlink.
9	Memory overhead per element	None beyond the value itself (plus resizing buffer in dynamic arrays).	Extra space for one (singly) or two (doubly) pointers per node.
10	Cache friendliness	High: contiguous memory improves CPU cache locality.	Poor: nodes scattered in memory, bad cache performance.
11	Resizing	Requires allocating larger block and copying all elements (expensive).	No resizing cost, fully dynamic growth/shrink.
12	Implementation complexity	Simple (built-in in most languages).	More complex (manual node management & pointers).
13	Use cases	Frequent random access, static or predictable size, cache-sensitive tasks.	Frequent insertions/deletions in dynamic or unpredictable size data.

Applications of Linked List:

1. implementing stacks and queues using linked list
2. chaining in hash maps
3. undo/redo is implemented using Linked Lists
4. LRU cache
5. used in graph representations (adjacency list)



Some more additions to the LinkedList class :

1. having dynamic size attribute
2. maintaining a tail pointer
3. magic methods like __str__, __repr__, __len__, __contains__, __iter__ etc
4. two pointer approach in linked list (slow /fast pointers)
5. implementing more methods like pop, search etc
6. converting a list to LL and LL to list
7. doubly linked list
8. using linked list to implement stack and queue

alternate Constructor

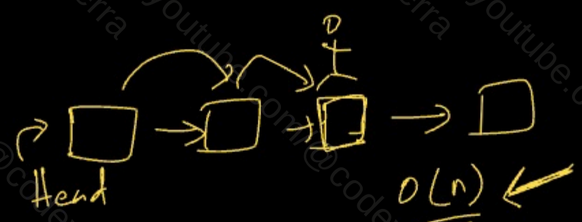
MCQ → Gate / interview

Leetcode Practice Problems:

1. finding middle of a linked list
2. delete nth node from a linked list
3. cycle detection → 2 pointer
4. reverse a linked list →
5. merge two sorted list
6. remove nth node from end

Maintaining a "tail pointer" in linked list

DLL



In a singly linked list, the append (insert at end) operation normally takes O(n) time, since we must traverse the entire list to reach the last node.

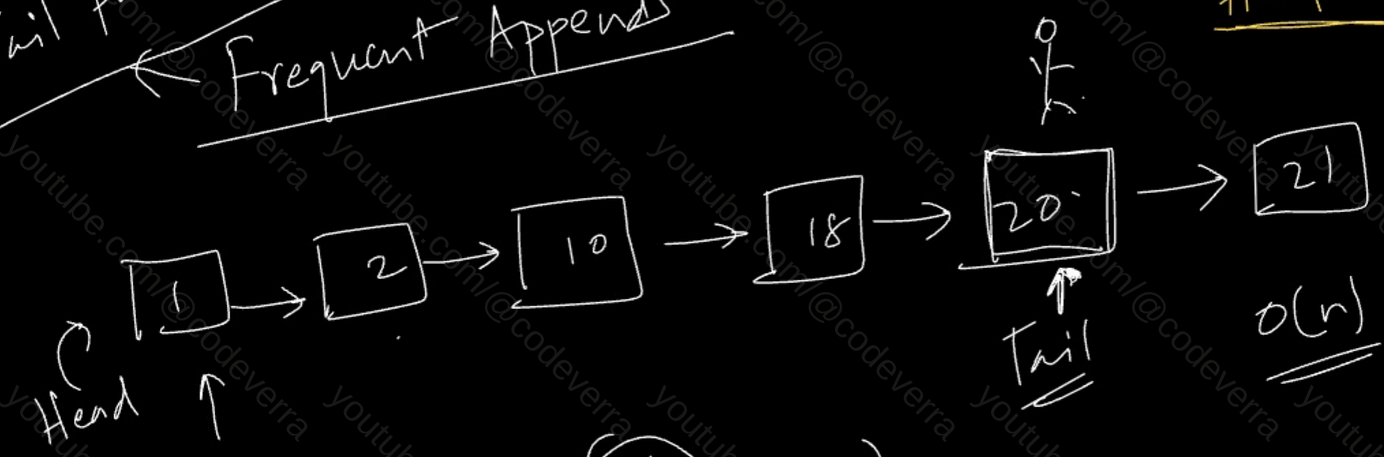
- Suppose you need to design a linked list optimized for frequent appends (e.g., continuously adding new data to the end).
- How can you modify the linked list to make append run in O(1) time instead of O(n)?
- Hint: Maintaining an extra pointer/reference may help.

optimization → DSA interviews

Tail

Insert at beginning
Delete at beginning
} → O(1)
↳ Because of Head pointer

Use Tail pointer ← Frequent Appends



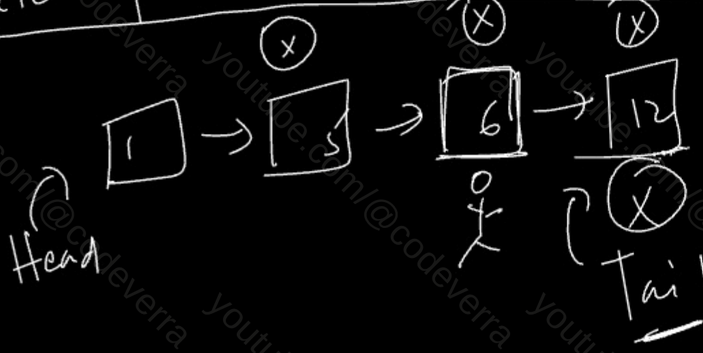
O(n) → O(1)
↓
Using a tail pointer

21

```
node = Node(21)
tail.next = node
tail = node
```

append
O(n) → O(1)

Suppose: ↳ Delete from end frequently O(n)



O(n) → O(1)
↳ Doubly Linked list

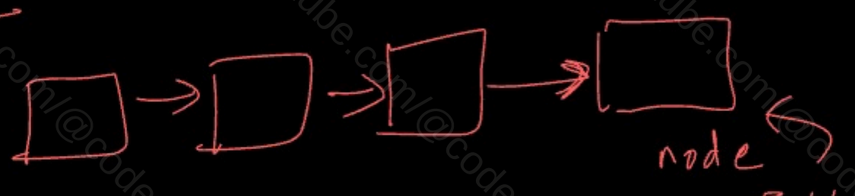
Algorithm for tail node

LinkedList

- head
- size
- tail

append

delete

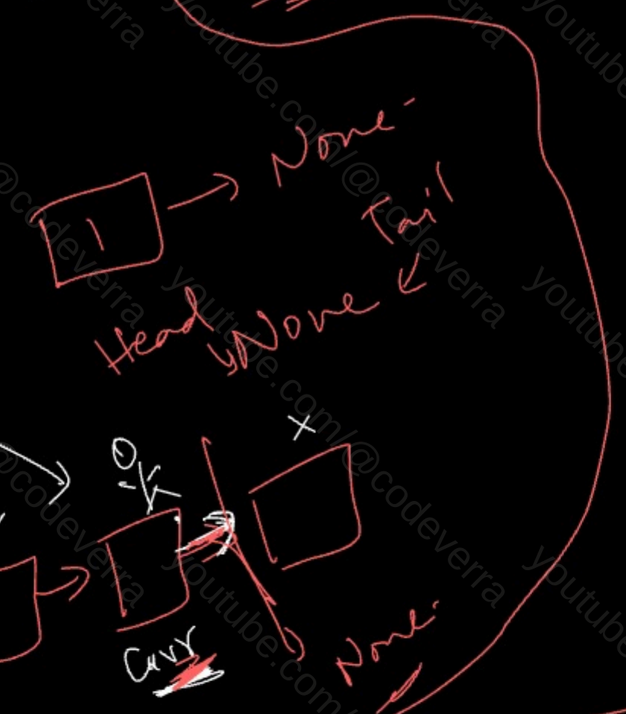


append using tail O(1)

if linkedlist is empty:
head = node
tail = node

else
tail.next = node
tail = node

delete O(n)

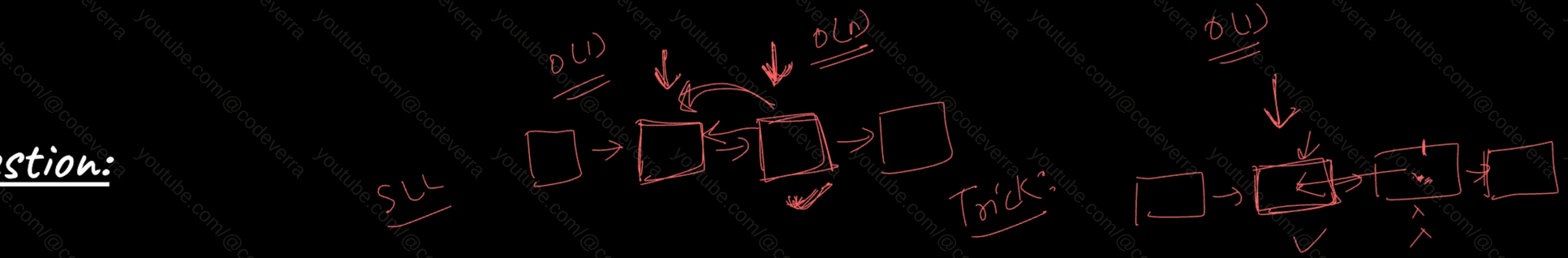


- if linkedlist is empty:
  raise Exception
- if there is only one node:
  head = None
  Tail = None
  size -= 1
  delete_logic7
- else:
  traverse to 2nd last node
  tail = curr
  size -= 1

LRU Cache → Doubly Linked List

- delete from end
- append frequent

→ SLL + tail pointer



Interview Question:

Design a data structure that supports the following operations in the indicated time:

- insert(x) at front: O(1)
- remove(node) given a pointer to the node: O(1)
- get_last() and remove_last(): O(1)

Doubly Linked list

SLL

DLL

DLL + 1 pointer

a) Which linked-list variant would you use? Explain briefly why.

b) Sketch the node layout and write pseudo-code for remove(node).

Doubly Linked List

↳ ① double Connections

next, prev

Head

Tail

insert | delete at both ends

O(1) → DLL

DLL → with tail pointer

O(1)

append ✓

prepend.

delete at beginning

delete at end

Disadvantage:

① Memory is more
prev, next

② Code is more complex
↳ maintain more connections

A doubly linked list is like a singly linked list but each node stores two pointers: next and prev. next points to the successor, prev to the predecessor.
Each node uses extra space (one more pointer) compared to singly linked lists (typically 2x pointer overhead)

We can use DLL when :

- ✓ we need efficient deletion given a node handle (e.g., LRU cache, editor undo/redo). ↓ deletion O(1)
- ✓ we want to traverse both directions easily (e.g., some list-based algorithms).
- ✓ When both ends of the list are hot (both insert/delete at head and tail).

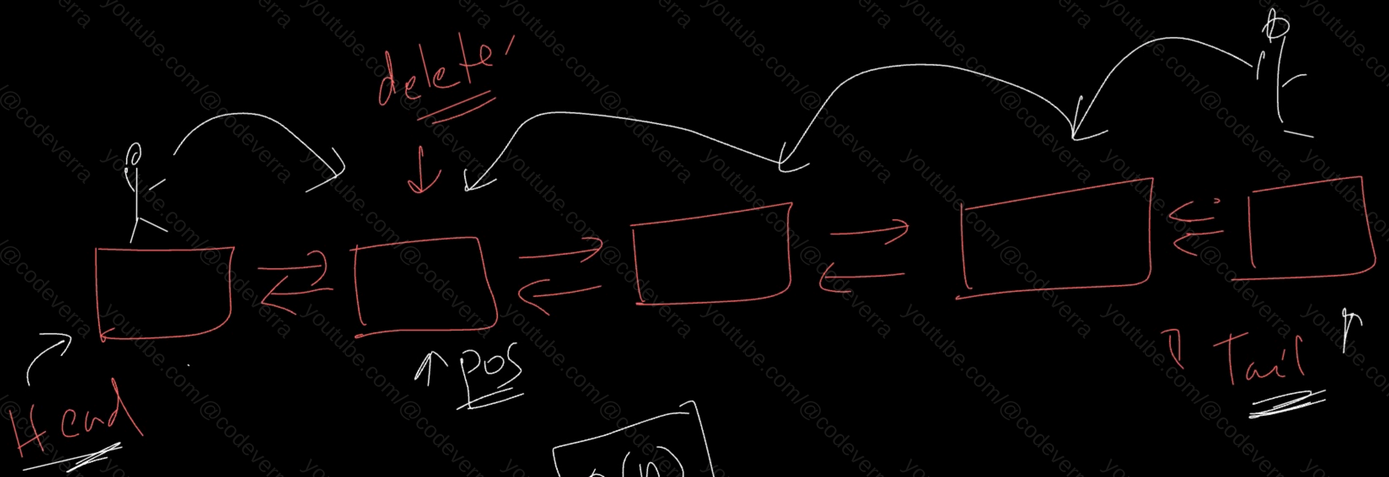
Things to keep in mind about DLL:

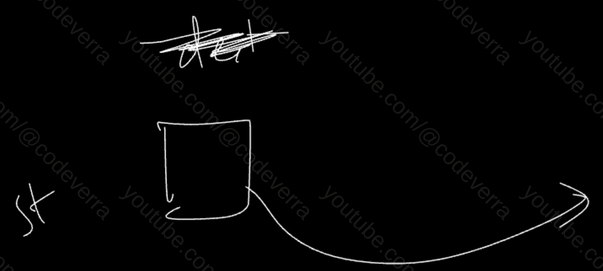
- More memory and slightly more bookkeeping – every insertion/deletion must update both next and prev.
- Slightly more complex implementation; show careful pointer updates in code.

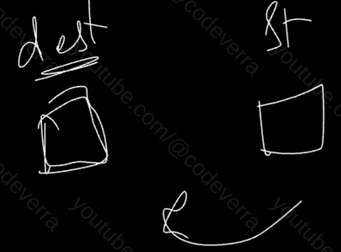
Common operations in Doubly Linked List:

- append(value) : O(1)
- prepend(value) : O(1)
- insert_at_position(pos, value) : O(n)
- delete_from_beginning() : O(1)
- delete_from_end() : O(1)
- delete_at_position(pos) : O(n)
- to_list(), from_list() : O(n)

traverse







When to use DLL:

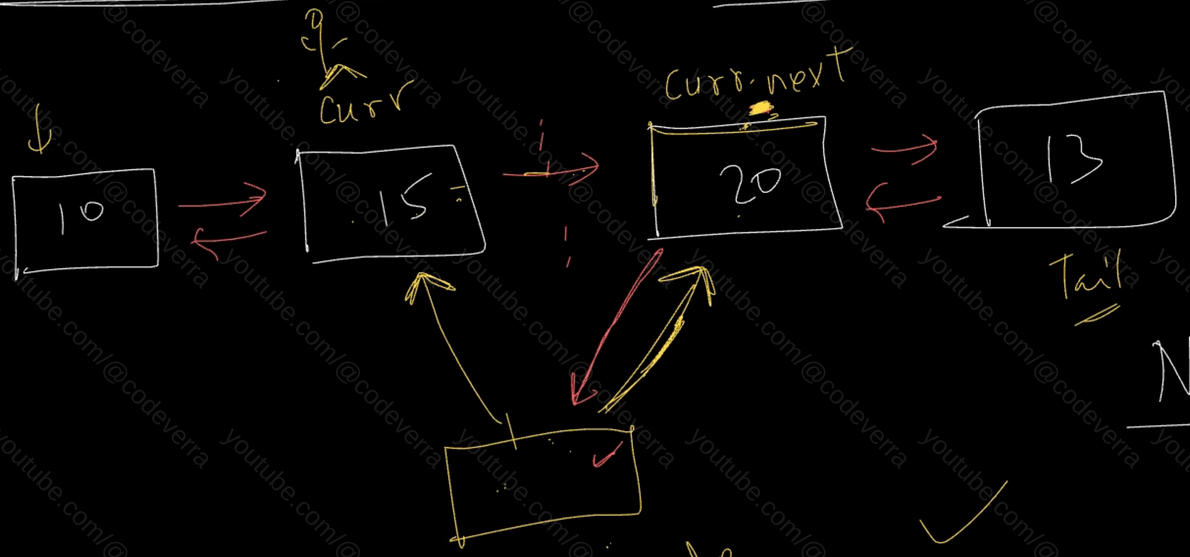
- When frequent deletions are needed given a pointer (like in LRU cache).
- When bi-directional traversal is needed.
- When end operations (append/delete at tail) are common.

When SLL is enough:

- If memory is tight.
- If operations are mostly at the head (like stack).
- If you don't need backward traversal.

	Operation	Singly Linked List (SLL)	Doubly Linked List (DLL)	Notes / Tips
1	Access by index (random access)	O(n)	O(n)	Both must traverse from head (or tail for DLL if optimizing).
2	Search by value	O(n)	O(n)	No improvement; both need traversal.
3	Insert at beginning	O(1)	O(1)	Just update head (and `prev` in DLL).
4	Insert at end (append)	O(n) without tail, O(1) with tail	O(1) with tail	DLL + tail pointer makes append efficient.
5	Insert at arbitrary position (by index)	O(n)	O(n)	Must traverse; DLL may optimize by traversing backward if position near end.
6	Delete at beginning	O(1)	O(1)	SLL: move head. DLL: move head and reset `prev`.
7	Delete at end	O(n) (need second-last node)	O(1) (with tail)	Big advantage of DLL if you need frequent deletions from end.
8	Delete at arbitrary position (by index)	O(n)	O(n)	DLL can traverse from closer end (slightly better average).
9	Delete given node (with reference)	O(n) (need previous node)	O(1)	DLL shines here: `node.prev.next = node.next`, `node.next.prev = node.prev`.
10	Memory usage	Less (1 pointer per node)	More (2 pointers per node)	DLL uses \~2× pointer memory.
11	Traversal	Forward only	Forward and backward	DLL allows reverse traversal directly.
12	Implementation complexity	Simple	More complex	DLL requires careful updates to both `prev` and `next`.

Insert at any position in DLL



Node
- data
- next
- prev

pos → valid

Check if pos is valid

pos < 0 or pos > size → invalid

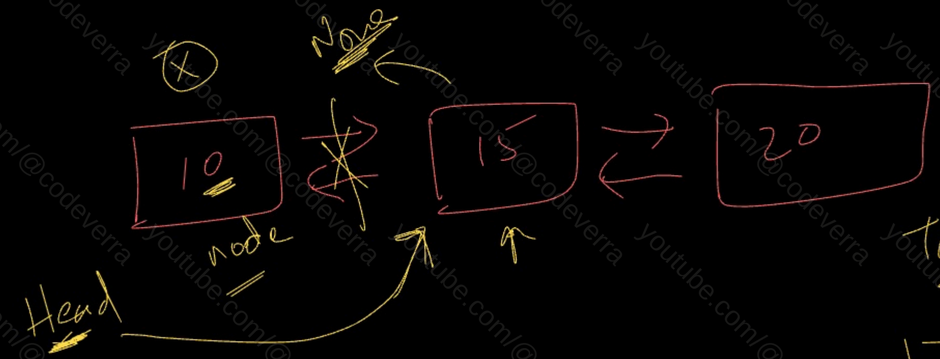
Handling Connections

```
node.next = curr.next
node.prev = curr
curr.next.prev = node
curr.next = node
```

Algorithm to insert

(pop) Delete from beginning in DLL

O(1)



(pop) Delete from beginning in DLL

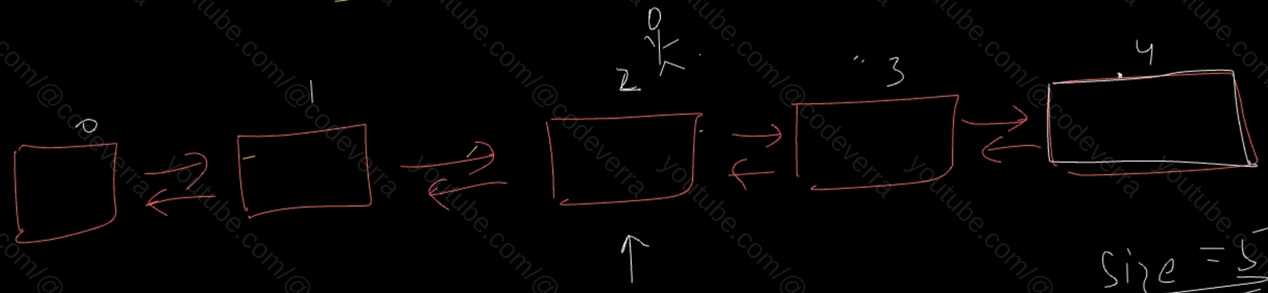
① if LL is empty
↳ Exception

② if size = 1
↳ head = tail = None

head.next = None
↓
there is only one element in the list

③ val = head.data
head = head.next
head.prev = None
↳ return val

Delete from any position in DLL



delete pos = 3

Size = 5

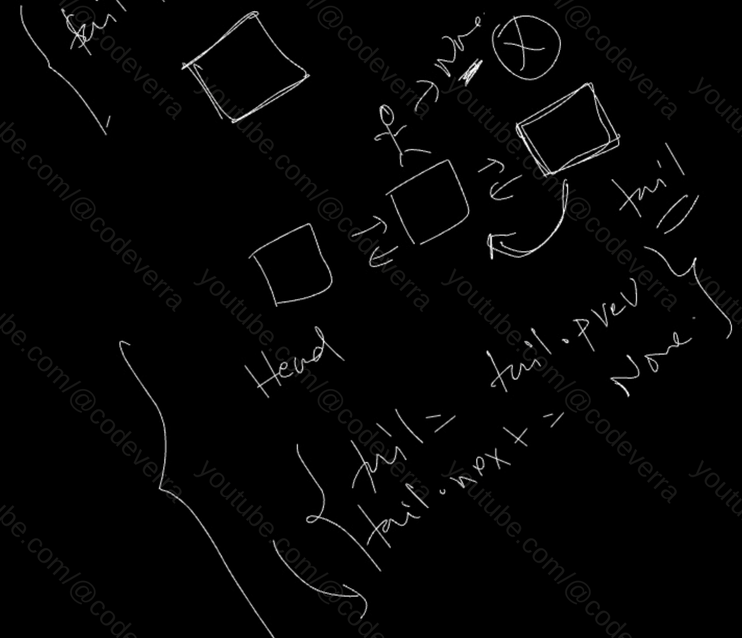
0, 1, 2, 3, 4

if we need to traverse → O(n)

O(1)

reference is given → node ← predecessor node

tail.prev = None



Algorithm

① Check if pos is valid. → raise Exception

invalid ← (pos < 0 or pos >= size)

② if only one element

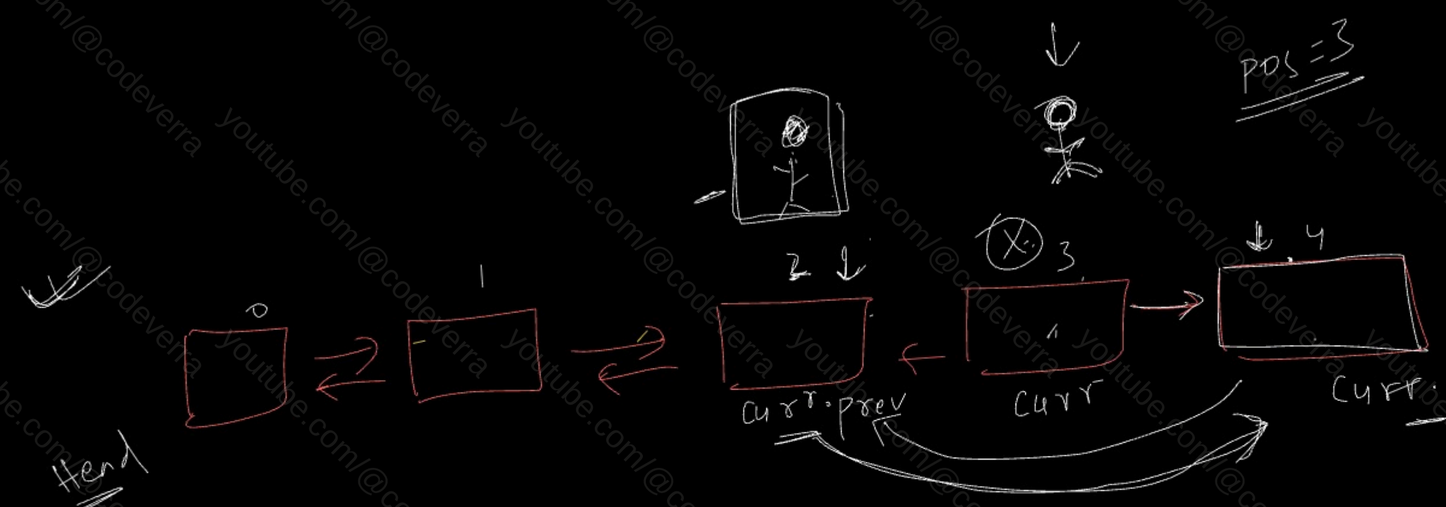
size = 1

head → None

tail

③ pos == 0 → delete from beginning

pos = size - 1 → delete from end



⑤ delete (pop) in DLL

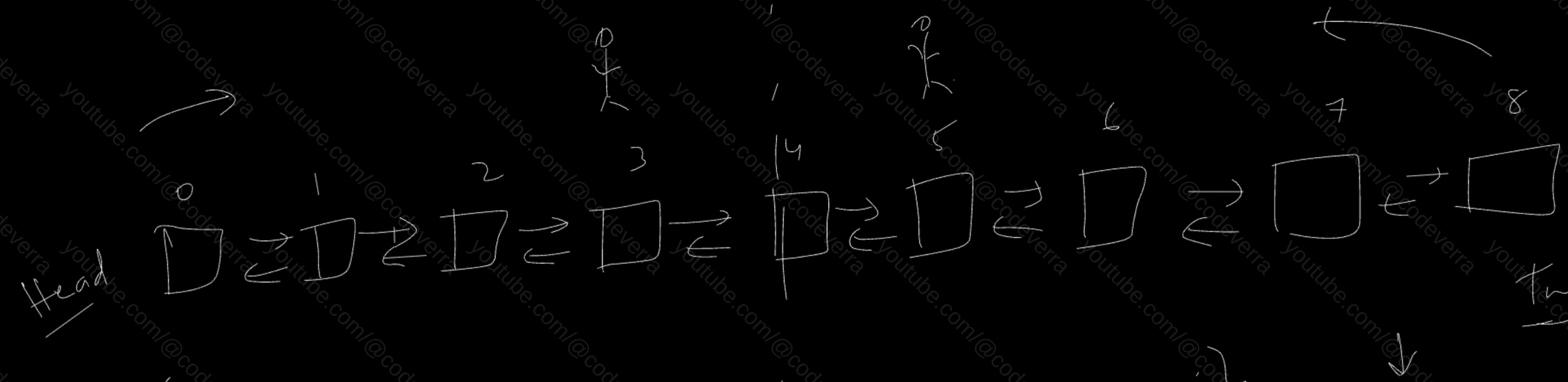
Value = curr.data

curr.prev.next = curr.next

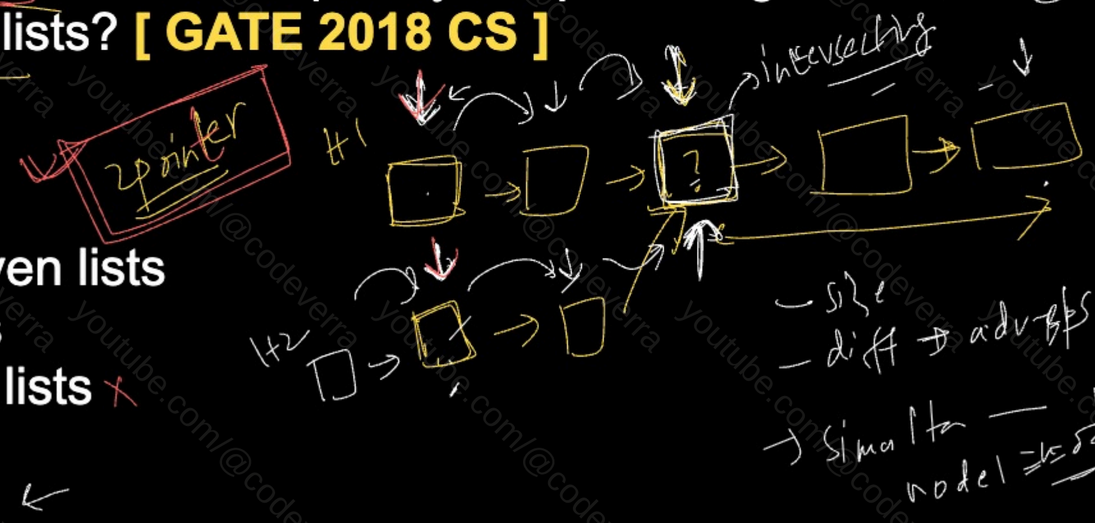
curr.next.prev = curr.prev

size = size - 1

return value

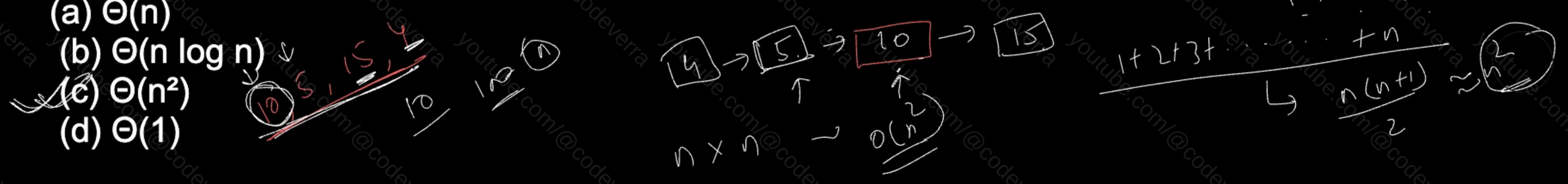


Question 1:
Suppose there are two singly linked lists both of which intersect at some point and become a singly linked list. The head or start pointers of both the lists are known, but the intersecting node and lengths of lists are not known. What is worst-case time complexity of optimal algorithm to find intersecting node from two intersecting linked lists? **[GATE 2018 CS]**



(a) Θ(n*m), where m, n are lengths of given lists
(b) Θ(n²), where m > n and m, n are lengths of given lists
(c) Θ(m + n), where m, n are lengths of given lists
(d) Θ(min(n, m)), where m, n are lengths of given lists

Question 2:
What is the worst-case time complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order? **[GATE 2020 CS]**

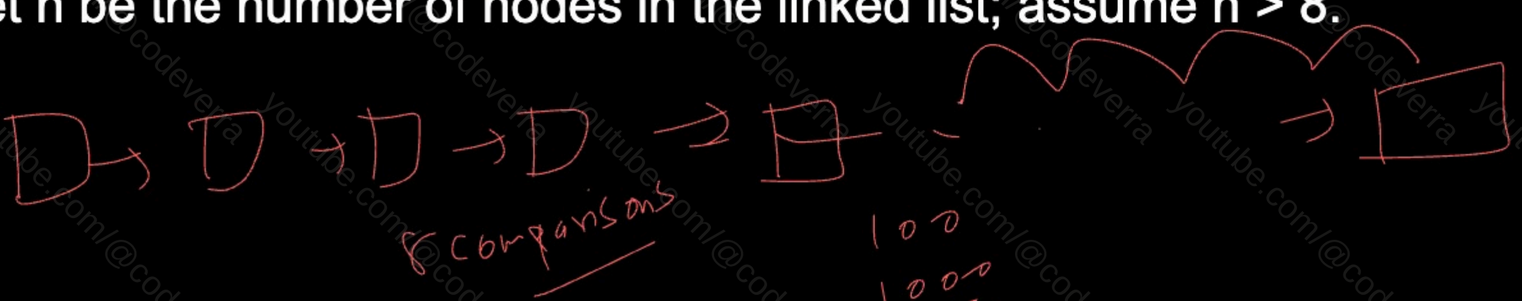


(a) Θ(n)
(b) Θ(n log n)
(c) Θ(n²)
(d) Θ(1)

Question 3:
In worst case, the number of comparisons needed to search a singly linked list of length n for a given element is:

(a) $\log_2 n$
(b) n/2
(c) $\log_2$
(d) n

Question 4:
What are the time complexities of finding the 8th element from beginning and 8th element from end in a singly linked list? Let n be the number of nodes in the linked list; assume n > 8.



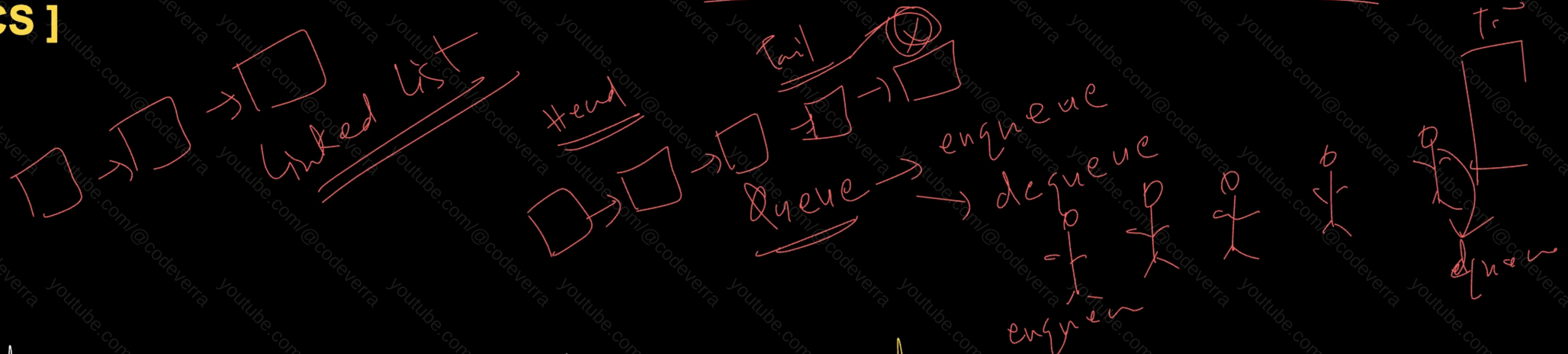
(a) O(1) and O(n)
(b) O(1) and O(1)
(c) O(n) and O(1)
(d) O(n) and O(n)

Question 5:
An unordered list contains n distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is. **[GATE 2015 CS]**

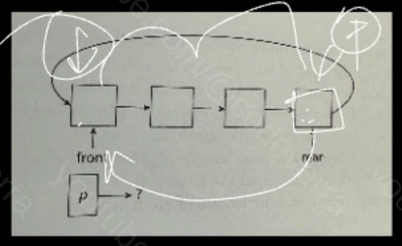
(a) Θ(n log n)
(b) Θ(n)
(c) Θ(log n)
(d) Θ(1)

Question 6:
A queue is implemented using a *noncircular* singly linked list with a head pointer and a tail pointer. n is the number of nodes. enqueue is implemented by inserting a new node at the **head**. dequeue is implemented by deletion of a node from the **tail**. What is the time complexity of the most time-efficient implementation of enqueue and dequeue, respectively, for this data structure? **[GATE 2018 CS]**

(a) Θ(1), Θ(1)
(b) Θ(1), Θ(n)
(c) Θ(n), Θ(1)
(d) Θ(n), Θ(n)



Question 7:
A circularly linked list is used to represent a queue. A single variable p is used to access the queue. To which node should p point such that both the operations enqueue and dequeue can be performed in constant time? **[GATE 2004 CS]**

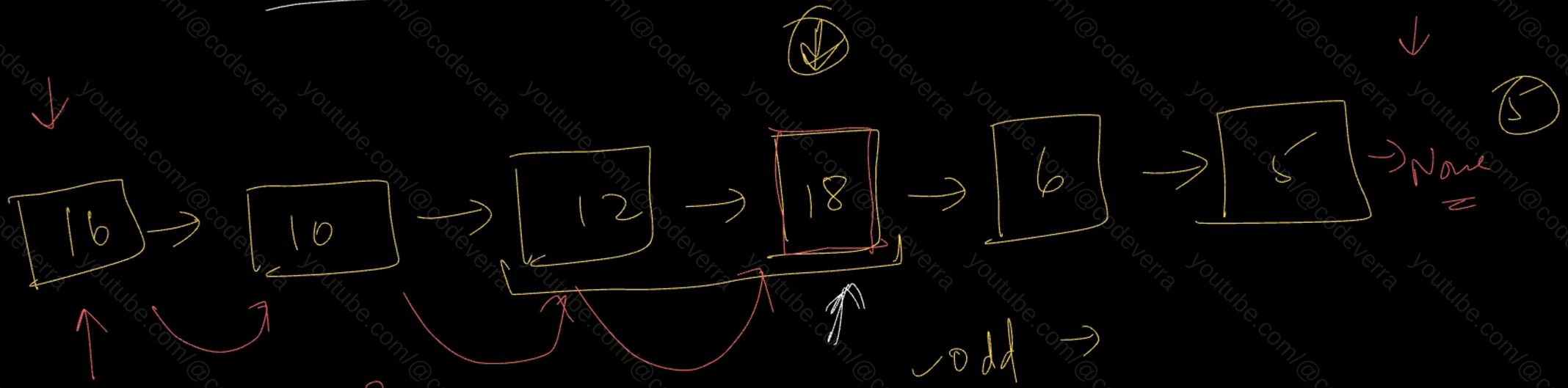


(a) front node
(b) rear node
(c) not possible with a single pointer
(d) node next to front

Leetcode

1. Find middle of linked list → 2 pointer
2. Reverse a linked list →
3. Merge linked lists →

① Find middle of a Linked List



size = 7

size = 6 → size/2 = 3

size = 5 → size/2 = 2

odd →

even →

```
# find size
curr = head
size = 0
while curr:
    count = size + 1
    curr = curr.next

curr = head
for _ in range (size//2):
    curr = curr.next
return curr
```

2 passes

O(n)

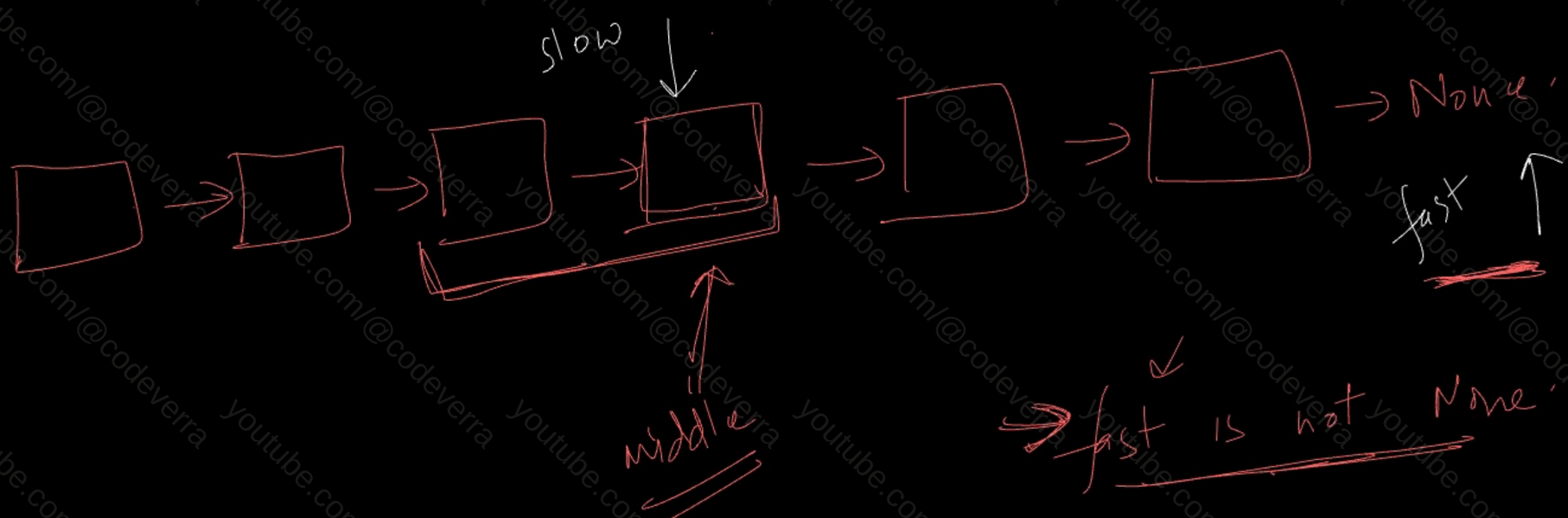
Complexity → O(n)

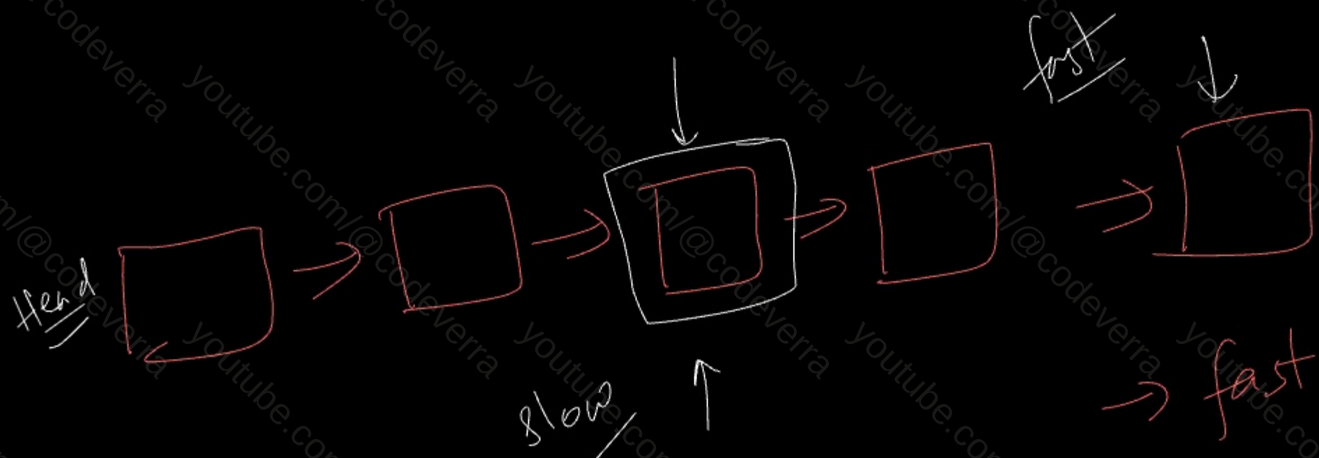
2 passes

2 pointer approach

1 pass

slow / fast
hare / tortoise






```
# initialize fast & slow
fast = head
slow = head
while fast and fast.next:
    slow = slow.next      # 1 step
    fast = fast.next.next # 2 step
return slow
```

SP

O(1)

1 pass

Cycle detection
↓
Floyd

Time Complexity : O(n)
Space Complexity : O(1)

fast, slow

Reverse a Linked List

Reversed Linked List → 12 → 8 → 20 → 15 → 10 → None

Brute Force / optimal approach →

[10, 15, 20, 8, 12]

12 → 8 → 20 → 15 → 10 → None

TC: O(n)
SC: O(n)
} → Linear

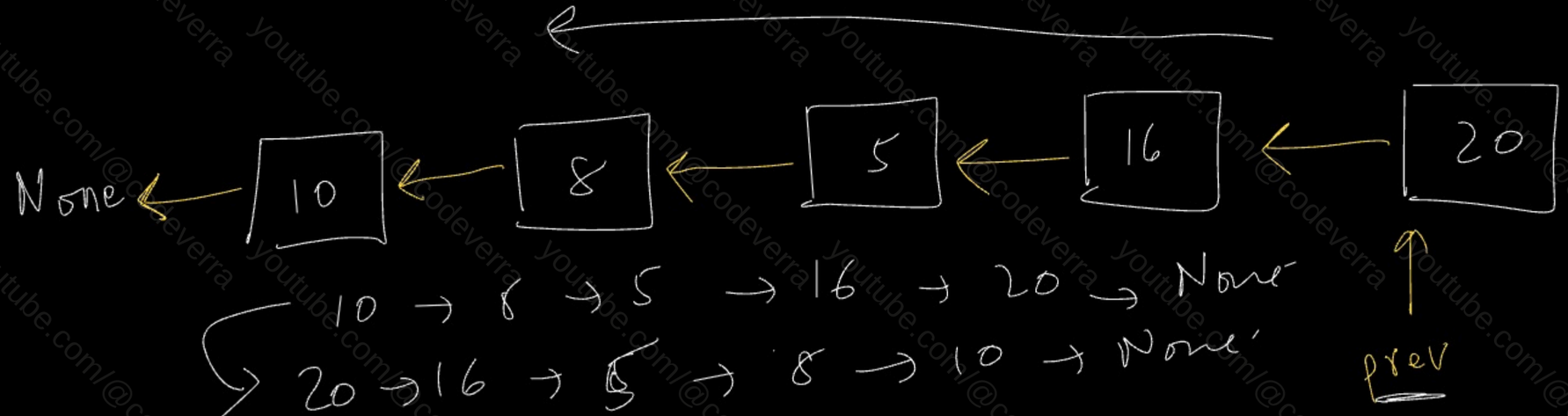
Recursive Approach

iterate

Pointer reversal approach

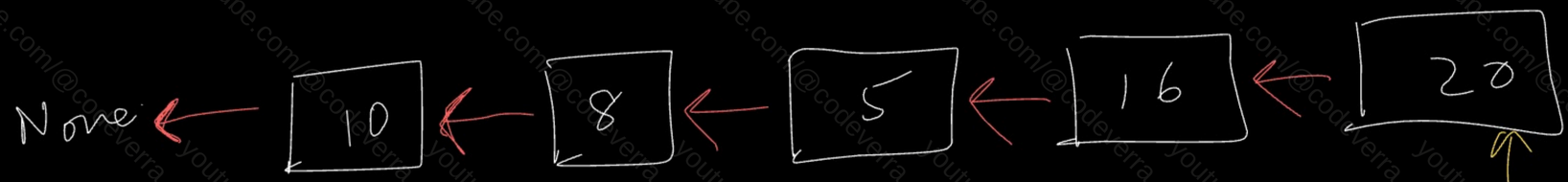
What? → curr → prev

How? → using temp



10 → 8 → 5 → 16 → 20 → None

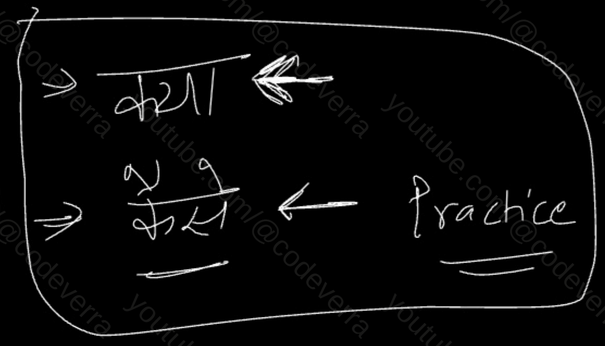
20 → 16 → 5 → 8 → 10 → None



Reverse linked list

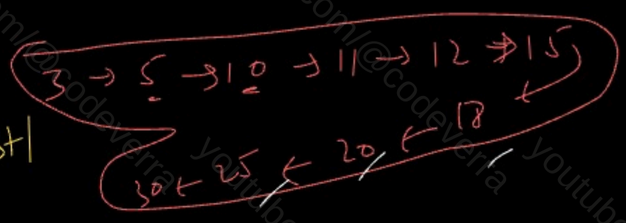
Algorithm

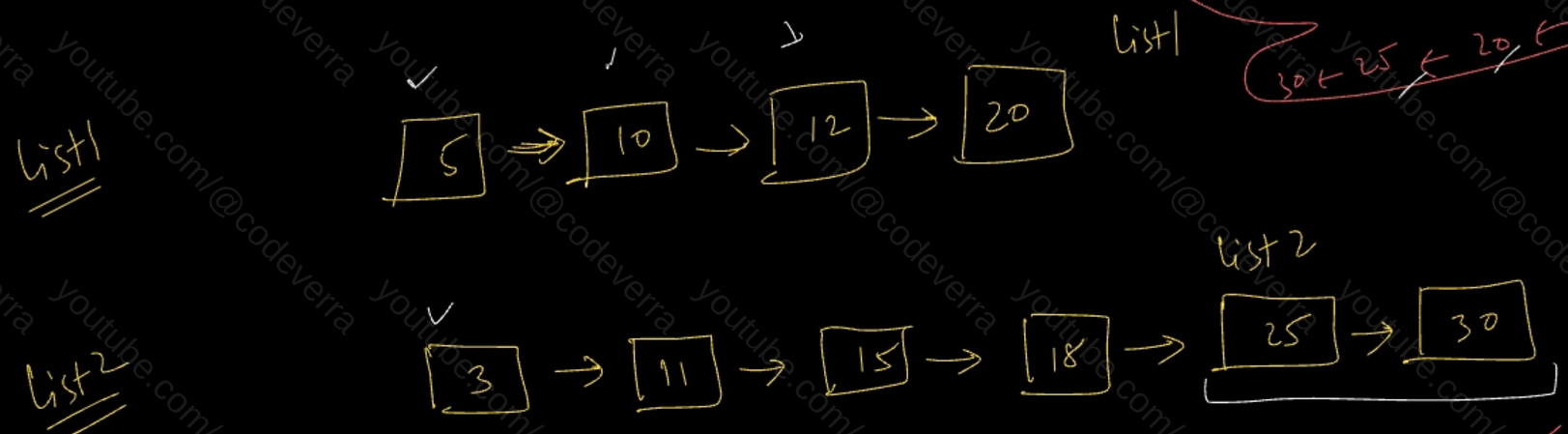
```
prev = None
curr = head
while curr:
    temp = curr.next
    curr.next = prev
    prev = curr
    curr = temp
return prev
```



- Linear → TC → O(n)
- Constant → SC → O(1) → temp, prev, curr

Merge two sorted lists





Edge case
list1 → empty
↳ list2

list2 → empty
↳ list1