MergeSort

→ Divide & conquer

mergeSort(arr, low, high)

T(n)

① Divide the bigger problem into smaller subproblems

if (low < high) {

c ← mid = low + (high - low)/2

T(n/2) ← mergeSort(arr, low, mid)

② conquer

↳ Recursion

mergeSort(arr, mid+1, high)

T(n/2)

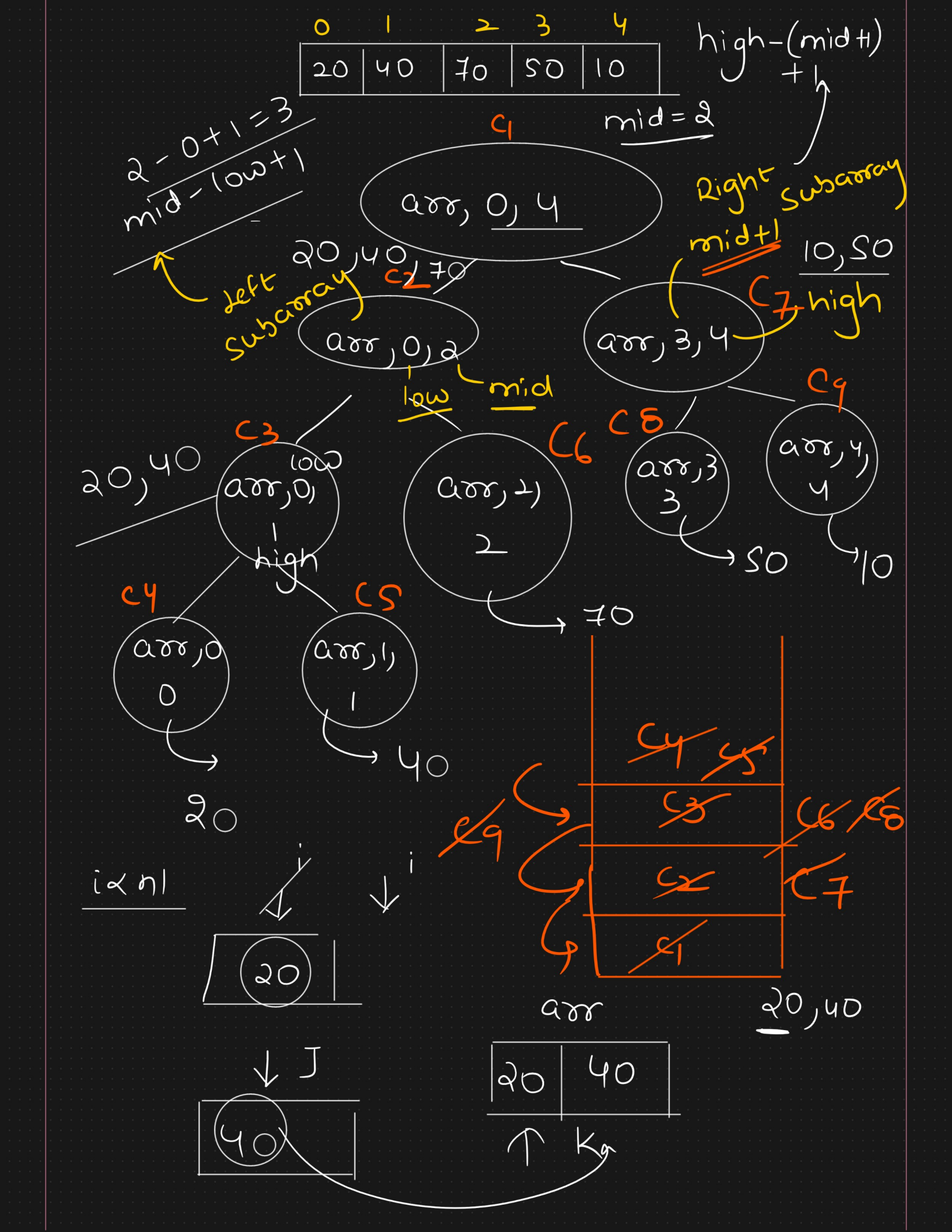
③ Combine

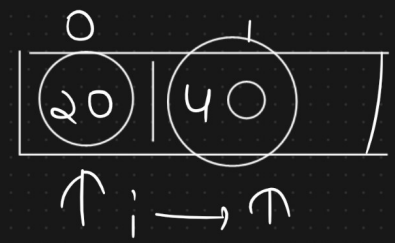
↳ mergeProcedure(arr, low, mid, high)

O(n)

low = 0, high = 4

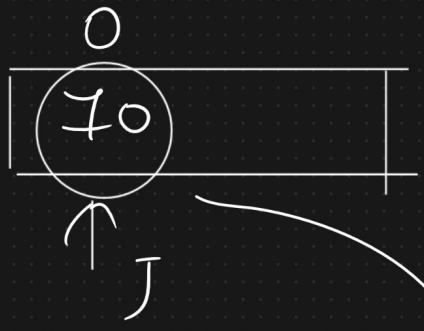
high - mid

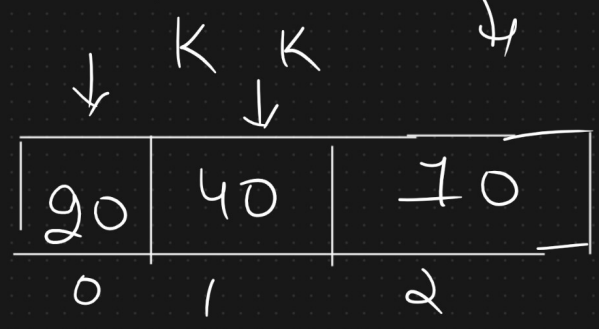

0 1 2 3 4
20 40 70 50 10
high-(mid+1)
+1
mid=2
2-0+1=3
mid-low+1
C1
arr, 0, 4
Right subarray
mid+1
10,50
C7
high
20,40
C2
70
left subarray
arr, 0, 2
arr, 3, 4
low
mid
C3
C6
C8
C9
20,40
low
arr, 0, 1
high
arr, 2, 2
arr, 3, 3
arr, 4, 4
50
10
C4
C5
70
arr, 0, 0
0
arr, 1, 1
1
40
20
i<mid
i
i
20
C4 C5
C3
C6 C8
C9
C2
C7
C1
arr
20,40
J
40
20 40
K

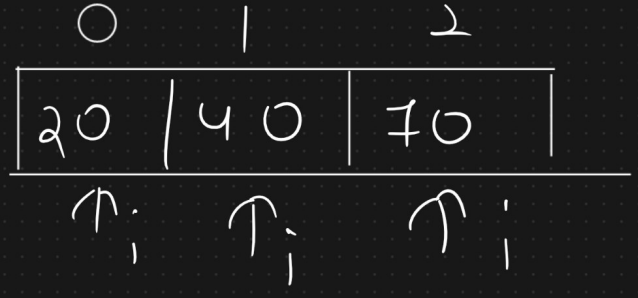


20, 70

4□, 70





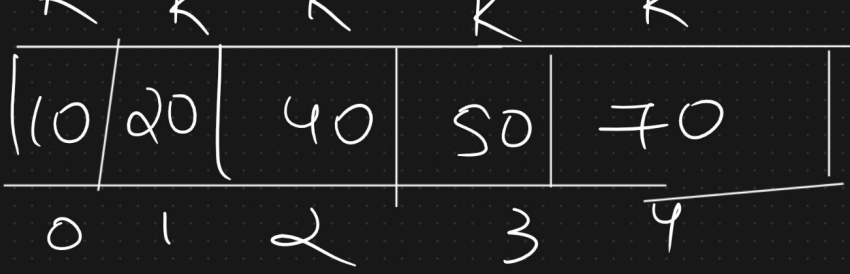


20, 10

20, 50

40, 50

70, 50



→ Sorted array

$$T(n) = 2T(n/2) + n$$

$2T(n/2)$ — Conquer

n — Merge Procedure

→ Master's Theorem / Substitution Method

$$= O(n \log n)$$

→ Best, average, worst

Space complexity (Stack space + Extra space)

→ $O(n)$

Extra space

Outplace sorting algorithm

Linked List

→ Dynamic data Structure

↳ No contiguous storage

Static array ← fixed size

dynamic array ← not any fixed size of an array

ArrayList

Python → only dynamic array

Random access

more frequently Array

↑ Access / Search

↓

Binary Search

↳ O(log n)

Linked List - (more frequently

↳ Insertion, deletion (Linked List)

↳ Skip List

↓

{ Advance DSA

Working of Dynamic Array

Load Factor > 0.8

(1) Size = 10

0	10
1	20
2	40
3	70
4	90
5	100
6	120
7	220
8	
9	

(1) copy

(2) Insertion of new element 270

10 * 2 = 20 (2)

0	10
1	20
2	40
3	70
4	90
5	100
6	120
7	220
8	270
19	

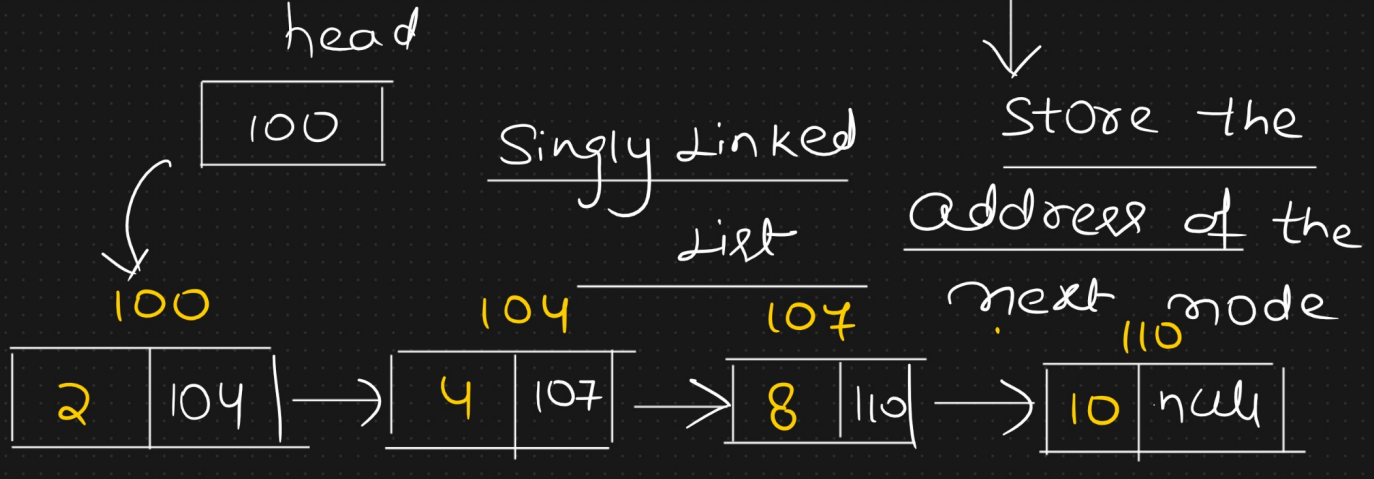
Complete DSA in Python

Linked List

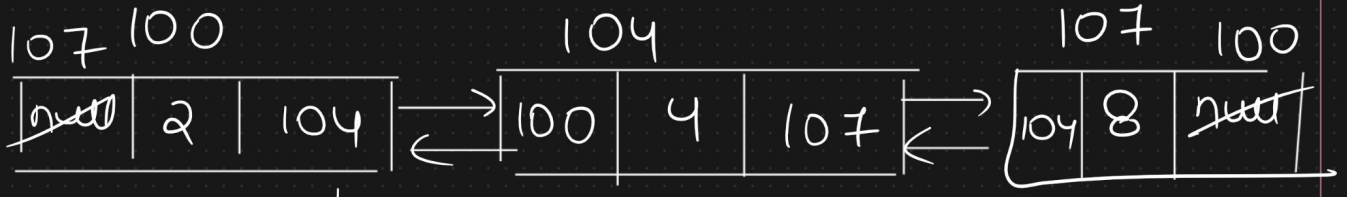
- Operations
  - Insertion
  - Deletion
  - Cycle Detection
  - Reversal
  - Palindrome

Linked List

- node
  - Data
  - Pointer (next)
    - Store the address of the next node



Doubly linked list



Realtime application

↳ Spotify

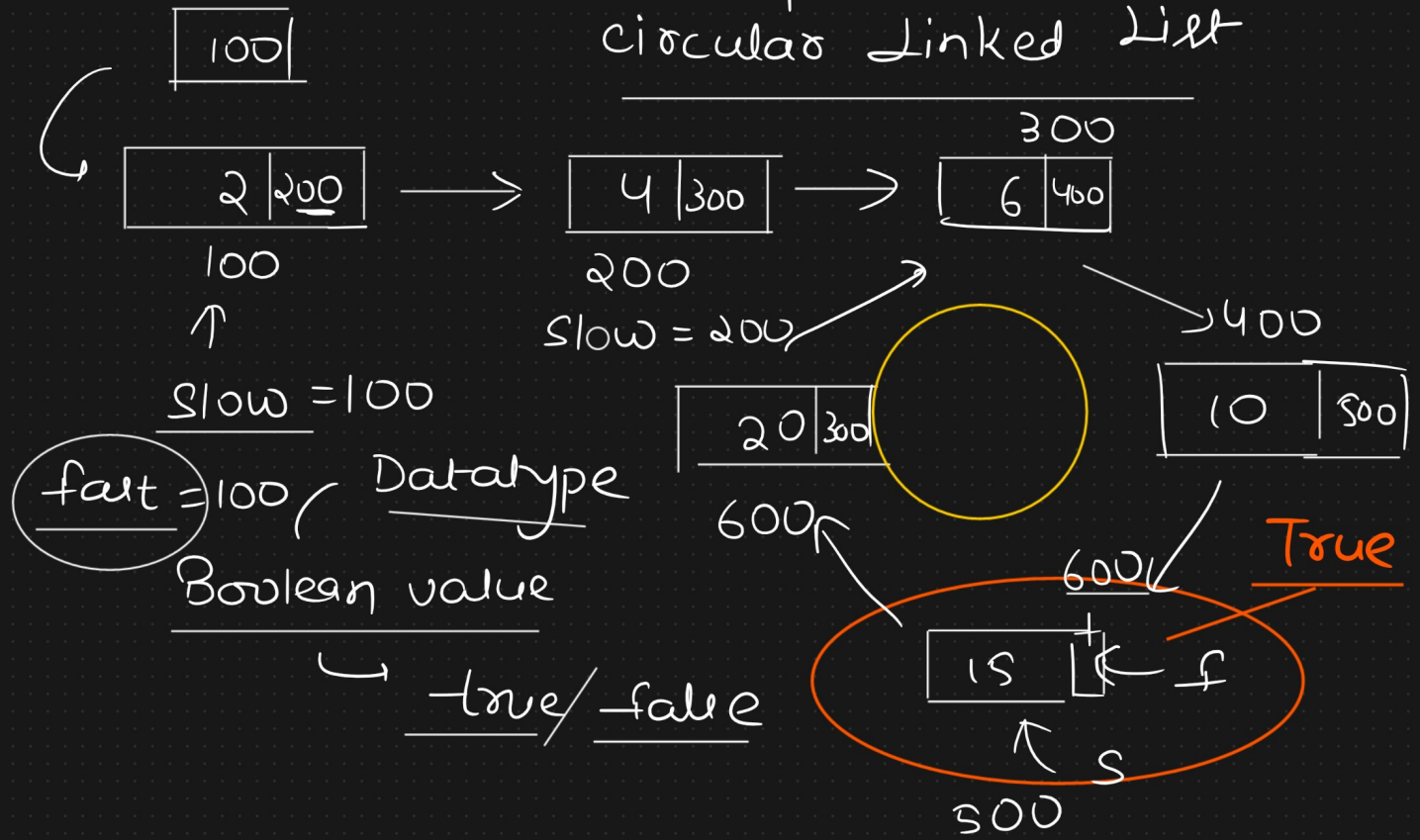
Circular + Doubly

backward songs

forward songs

Slow → one step at a time,
fast → two step at a time

circular linked list



fast = 100 / Datatype

Boolean value

↳ true / false

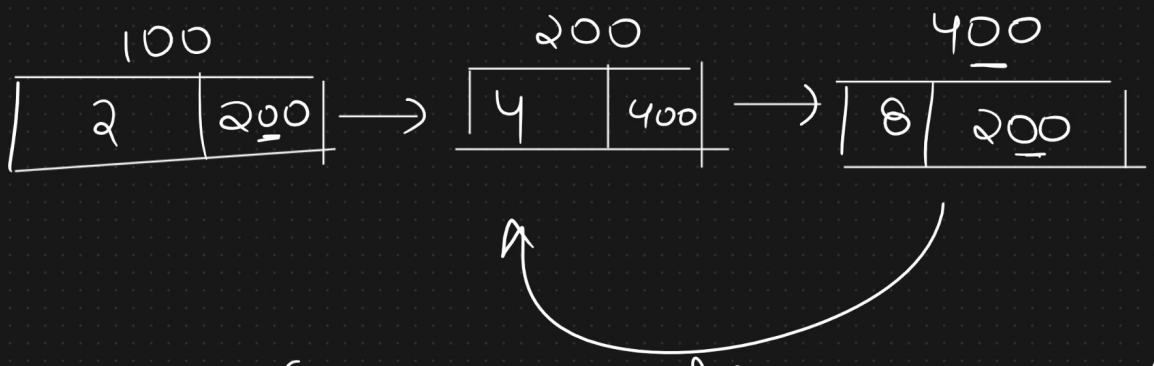
if (slow == fast)

↳ cycle in Linked List

↳ return true

floyd's cycle Detection Algorithm

head



while (slow != null && fast != null && fast.next != null)

slow = ~~100~~ 200 400

fast = ~~100~~ ~~200~~ ~~400~~ 400 400

Same node Point

Loop / cycle in a given Linked List

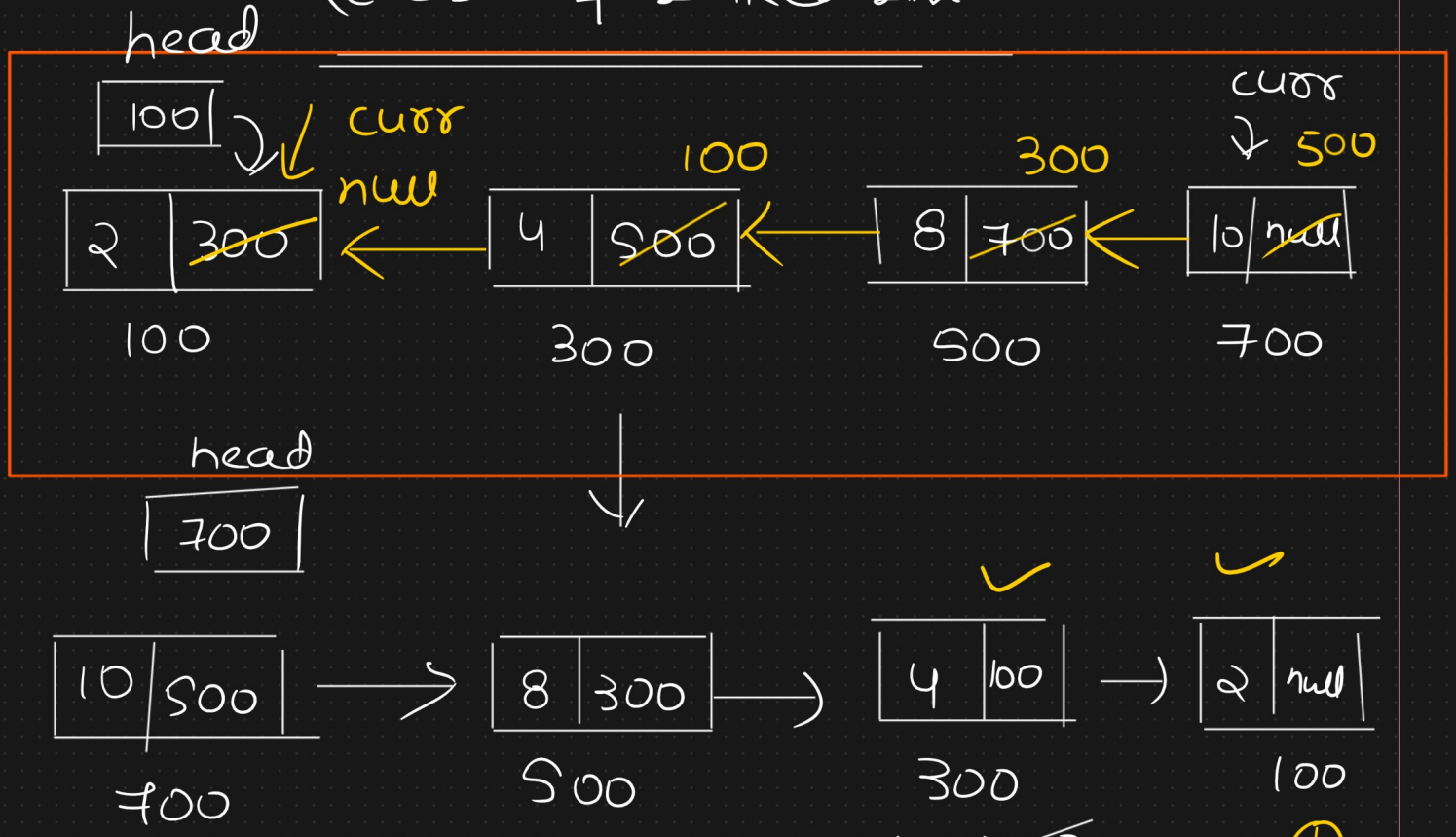
time complexity

↳ $O(n)$

space complexity

↳ $O(1)$

Reversal of Linked List



null 700 500

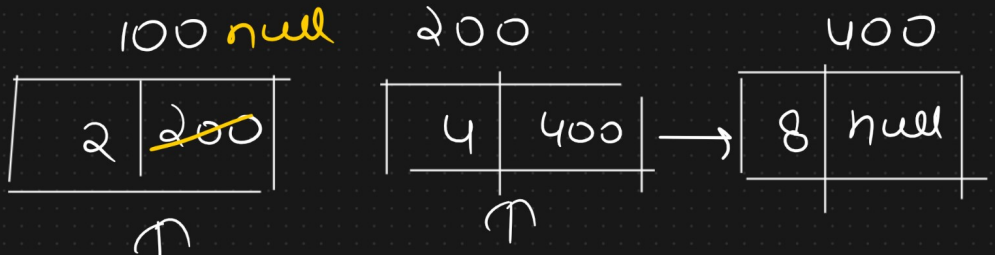
Curr = head = 100 300 null 700

nextPtr = null 300 500 700

Prev = null 100 300 500

O(n)

```
while (curr != null) {
    nextPtr = curr.next;
    curr.next = prev;
    prev = curr;
    curr = nextPtr;
}
head = prev;
return head;
```



Complete LL

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

Palindrome

LinkedList

ArrayList ← 2 Pointers

time complexity = O(n)

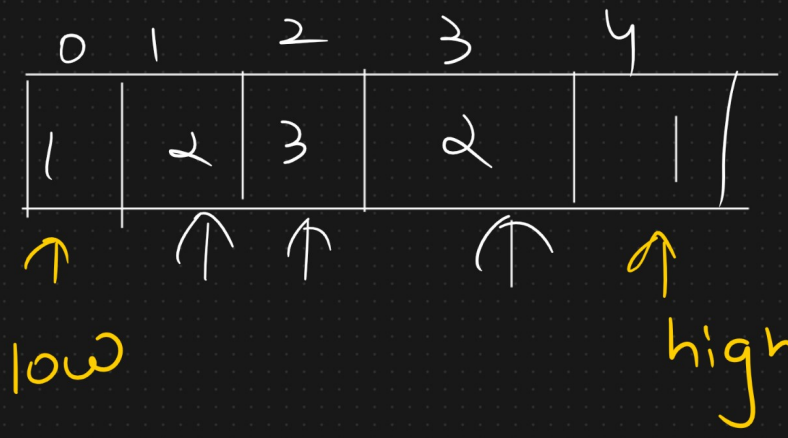
low
high

Approach

low = 0 /

high = ~~3~~ 2

```
while (low < high) {
    if (arr(low) == arr(high)) {
        low = low+1;
        high = high-1;
    }
    return true
}
arr(low) != arr(high) {
    return false }
```



low = ~~0~~ 1 ~~2~~

high = ~~4~~ ~~3~~ 2

Space complexity = O(n)

↳ ~~ArrayList~~

↳ Dynamic Array

Task

time complexity = O(n)

Space complexity = O(1)