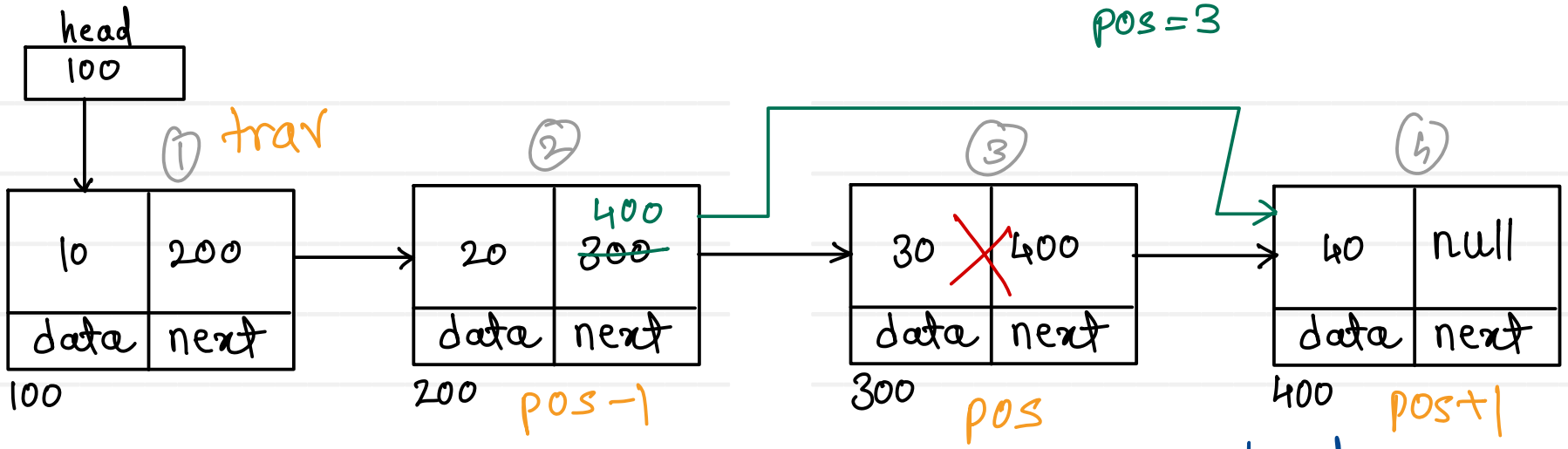
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Singly linear Linked List - Delete position



1. if list is empty
   return
2. if list is not empty
   a. traverse till pos-1 node
   b. add pos+1 node into next of pos-1 node

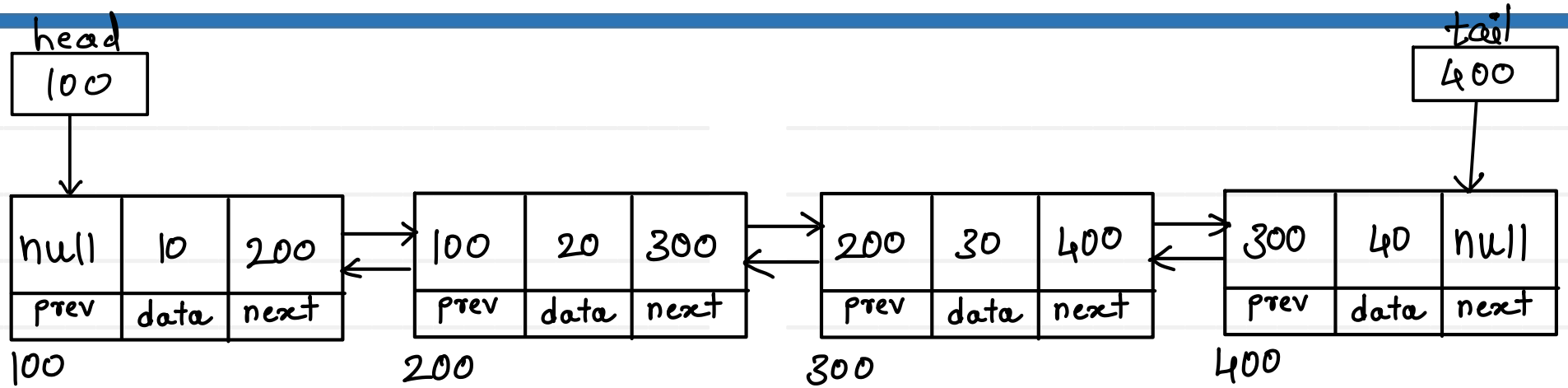
$T(n) = O(n)$

```
trav = head
for(i=1; i<pos-1 ; i++) {
    trav = trav.next;
    if(trav.next == null) return;
}
trav.next = trav.next.next
```

pos=5

trav	i	i<4
100	1	T
200	2	T
300	3	T
400		

Doubly Linear Linked List - Display



Forward traversal

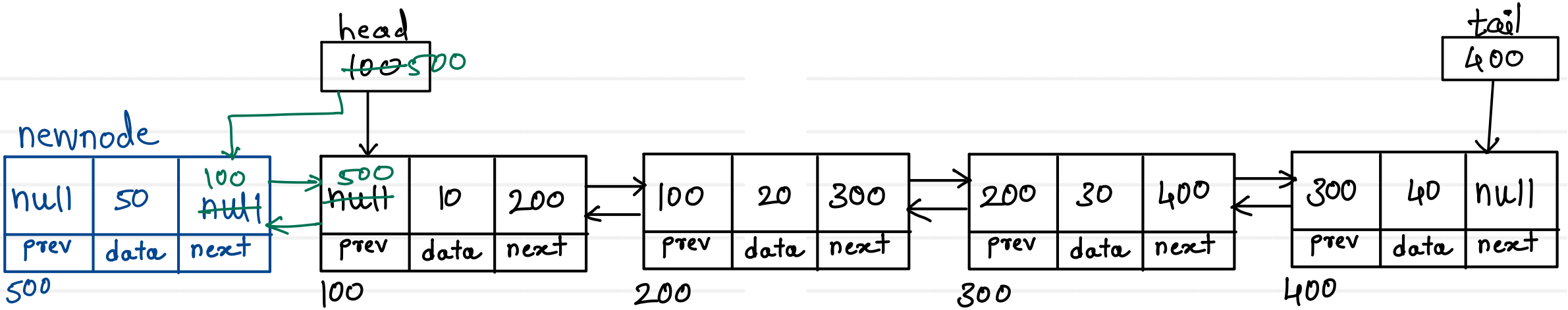
1. create trav & start at head
2. print current node data (trav->data)
3. go on next node (trav = trav->next)
4. repeat above two steps till last node

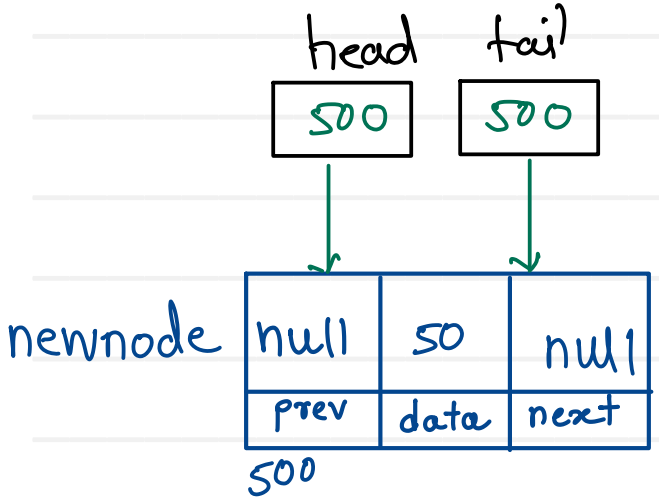
Backword traversal

1. create trav & start at tail
2. print current node data (trav->data)
3. go on prev node (trav = trav->prev)
4. repeat above two steps till first node

$$T(n) = O(n)$$

Doubly Linear Linked List - Add first

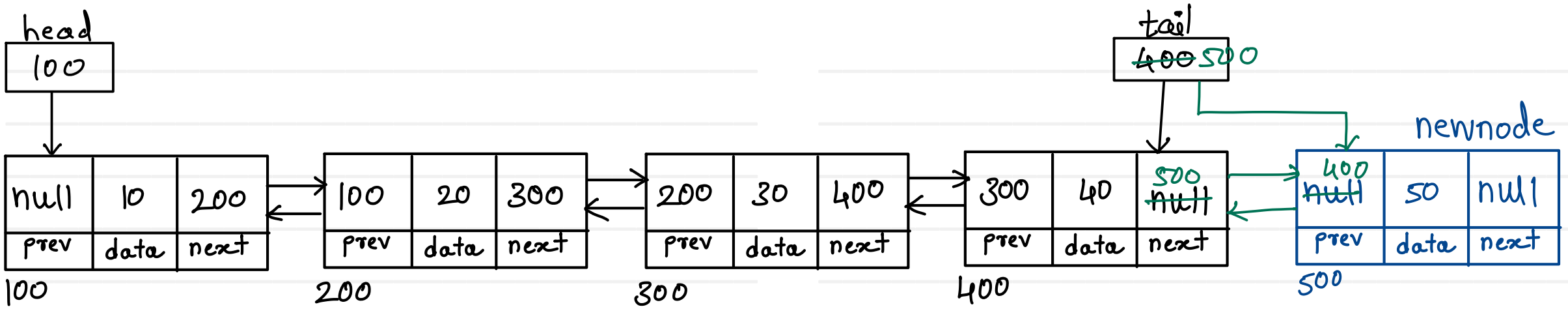


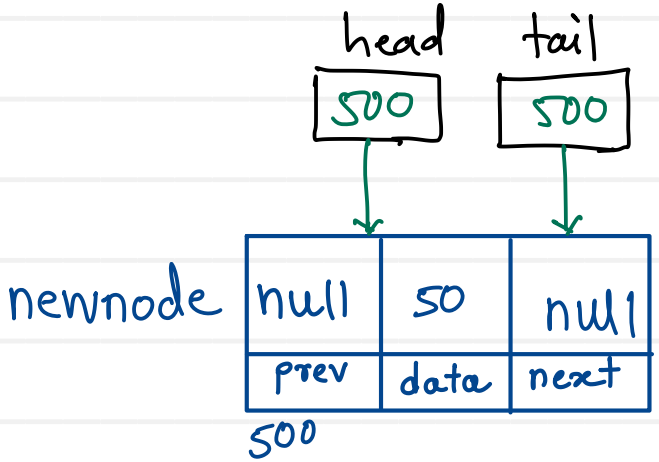


1. Create newnode
2. if empty
   add newnode into head & tail
3. if not empty
   a. add first node into next of newnode
   b. add newnode into prev of first node
   c. move head on newnode

$T(n) = O(1)$

Doubly Linear Linked List - Add last





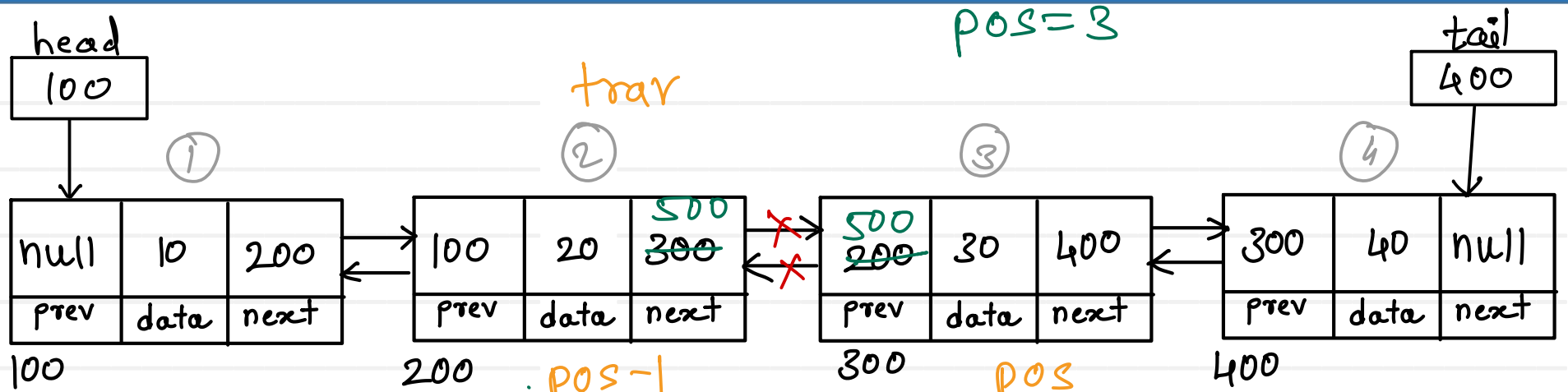
if empty
head=tail=newnode.

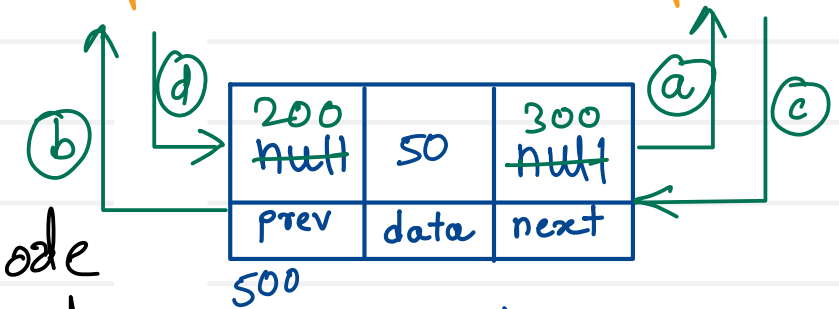
a. add last node into prev of newnode
b. add newnode into next of last node
c. move tail on newnode

$T(n) = O(1)$

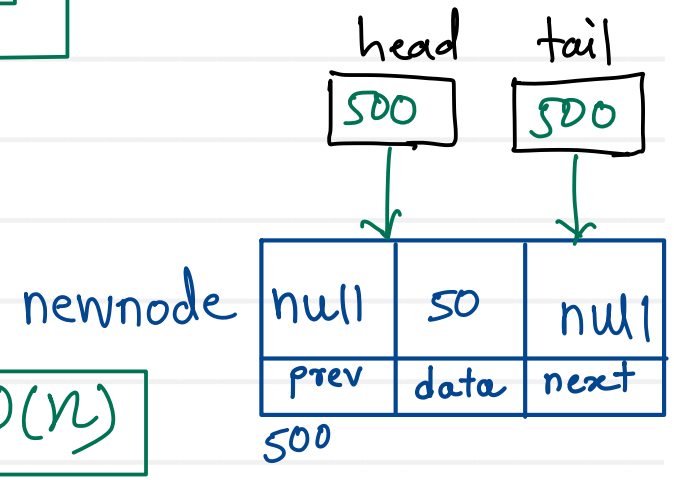
Doubly Linear Linked List - Add position

valid positions :
1 to count+1
invalid positions:
< 1
> count+1



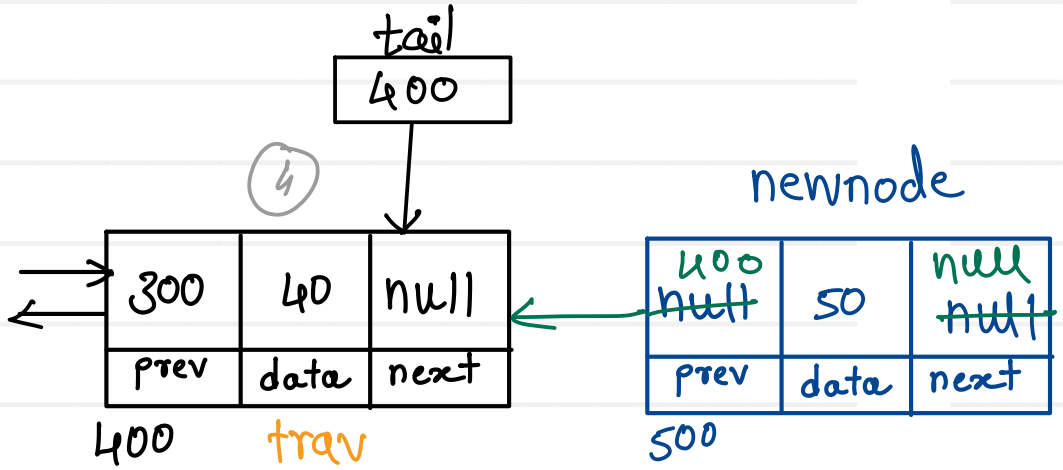


- traverse till pos-1 node
a. add pos node into next of newnode
b. add pos-1 node into prev of newnode
c. add newnode into prev of pos node
d. add newnode into next of pos-1 node



$T(n) = O(n)$

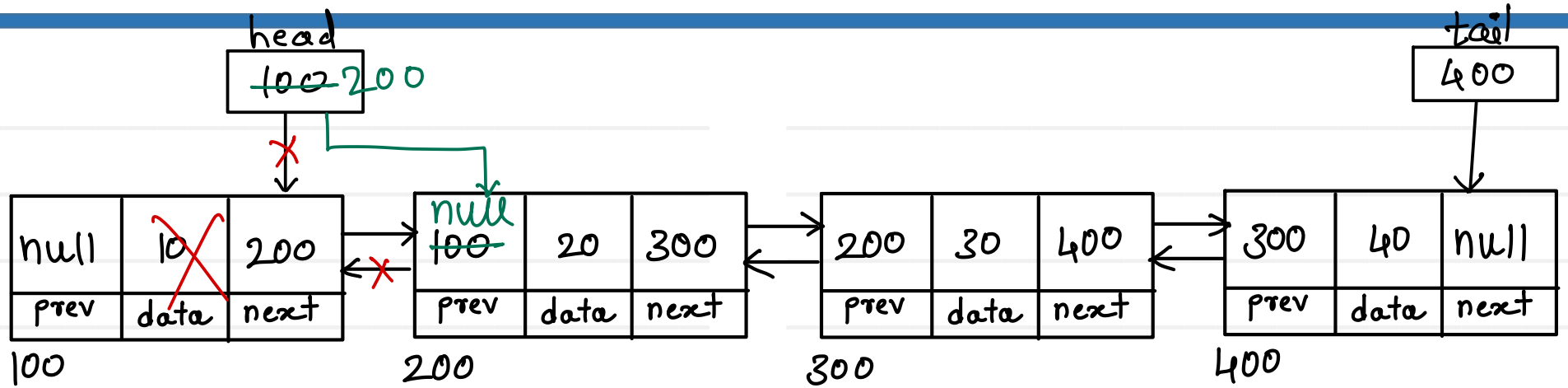
pos = 5



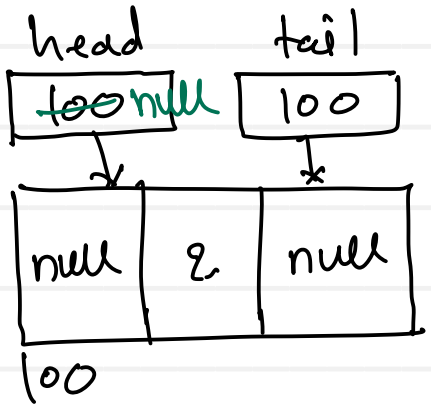

```
newnode.next = trav.next
newnode.prev = trav
trav.next.prev = newnode
trav.next = newnode
```

null pointer exception →

Doubly Linear Linked List - Delete first




```
head = head.next
head.prev = null
```

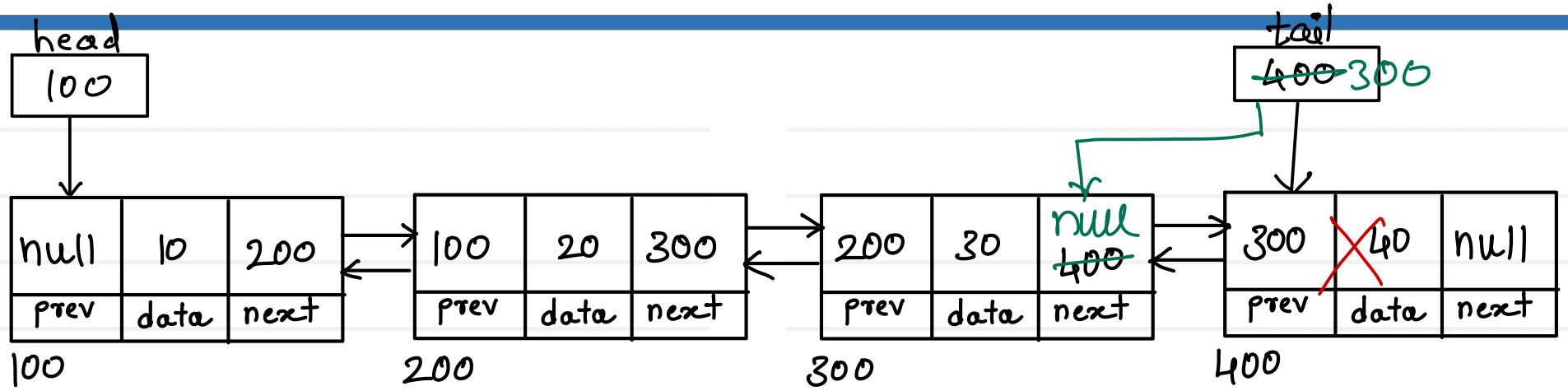



```
if(head == tail)
    head = tail = null;
```

if list is not empty
a. move head on second node
b. add null into prev of second node

$T(n) = O(1)$

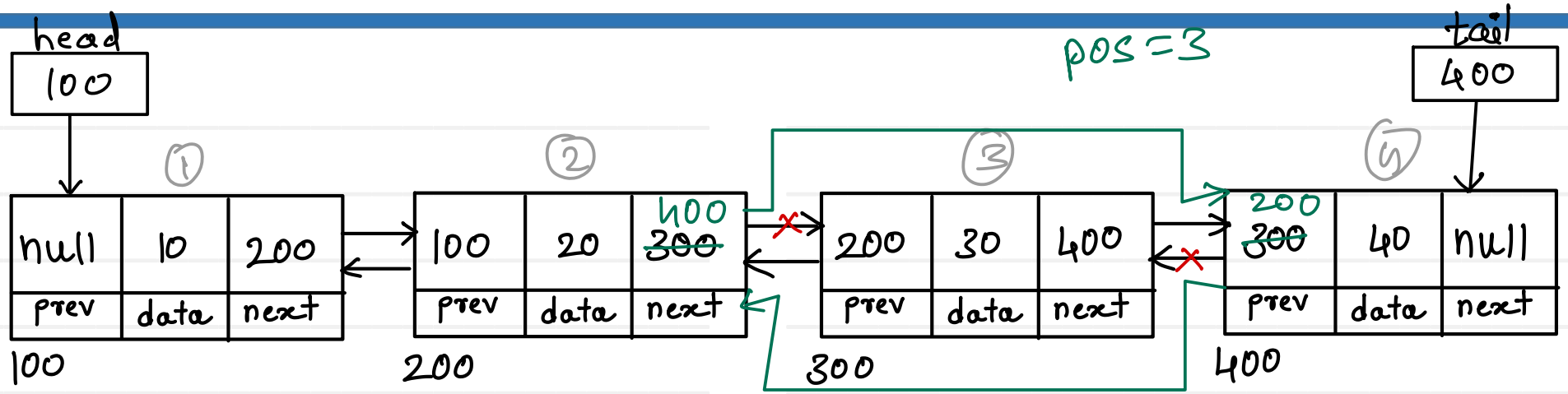
Doubly Linear Linked List - Delete last



if list is not empty
a. move tail on second last node
b. add null into next of second last node

$T(n) = O(1)$

Doubly Linear Linked List - Delete Position



valid positions :
1 to count

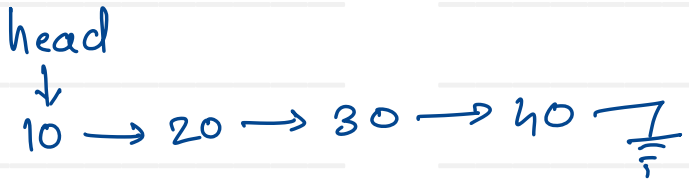
invalid positions :
< 1
> count

a. traverse till pos node
b. add pos+1 node into next of pos-1 node
c. add pos-1 node into prev of pos+1 node

$T(n) = O(n)$

Singly linear linked list - Display reverse

Direct Recursion

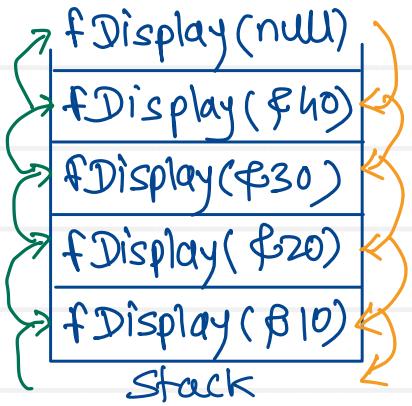


Tail Recursion

```
void fDisplay(Node trav) {
    if (trav == null)
        return;
    sysout(trav.data);
    fDisplay(trav.next);
}
```

$T(n) = O(n)$

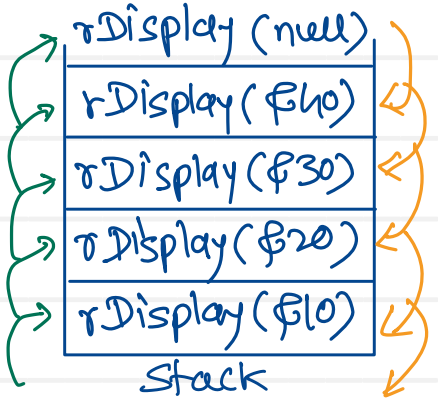
$S(n) = O(n)$



10, 20, 30, 40

Head/Non-tail recursion

```
void rDisplay(Node trav) {
    if (trav == null)
        return;
    rDisplay(trav.next);
    sysout(trav.data);
}
```

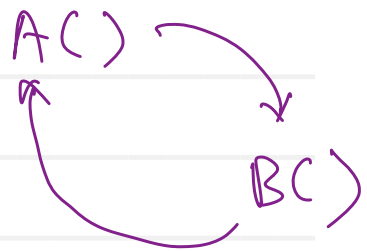


40, 30, 20, 10

Direct Recursion

```
A( ) {
    A();
}
```

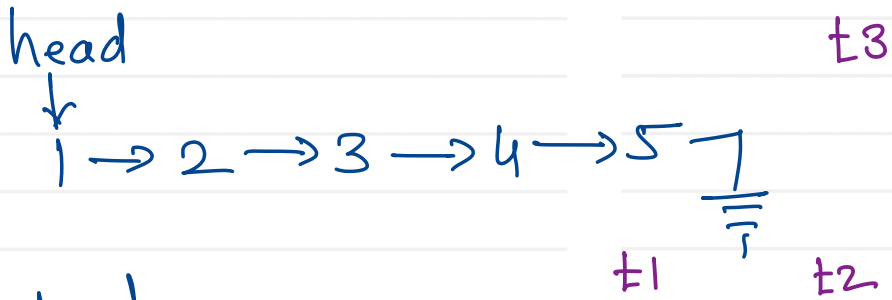
Indirect Recursion



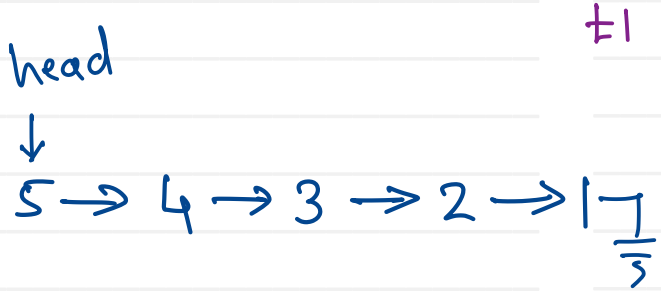
Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]

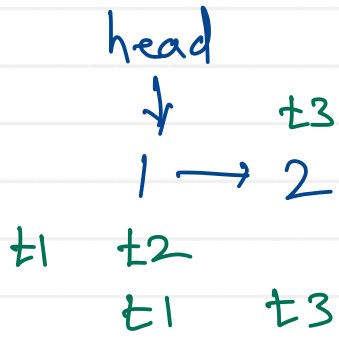


Example 2:
Input: head = [1,2]
Output: [2,1]



Example 3:

Input: head = []
Output: []




```
void reverseList(Node head) {
    Node t1 = null;
    Node t2 = head;
    Node t3;
    while(t2 != null) {
        t3 = t2.next;
        t2.next = t1
        t1 = t2
        t2 = t3
    }
    head = t1;
}
```

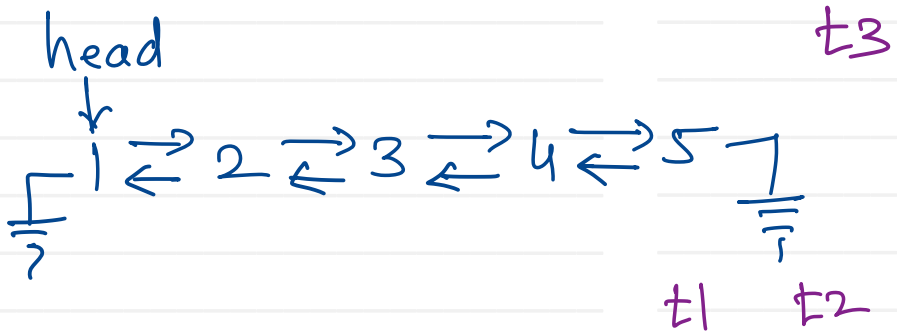
$T(n) = O(n)$

$S(n) = O(1)$

Reverse Linked List

Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]



Example 2:
Input: head = [1,2]
Output: [2,1]

Example 3:

Input: head = []
Output: []

T(n) = O(n)

S(n) = O(1)

```
Node reverseList(head){
    Node t1 = head;
    Node t2 = head.next;
    Node t3;
    t1.next = null;
    while(t2 != null){
        t3 = t2.next;
        t2.next = t1;
        t1.prev = t2;
        t1 = t2;
        t2 = t3;
    }
    head = t1;
    return head;
}
```

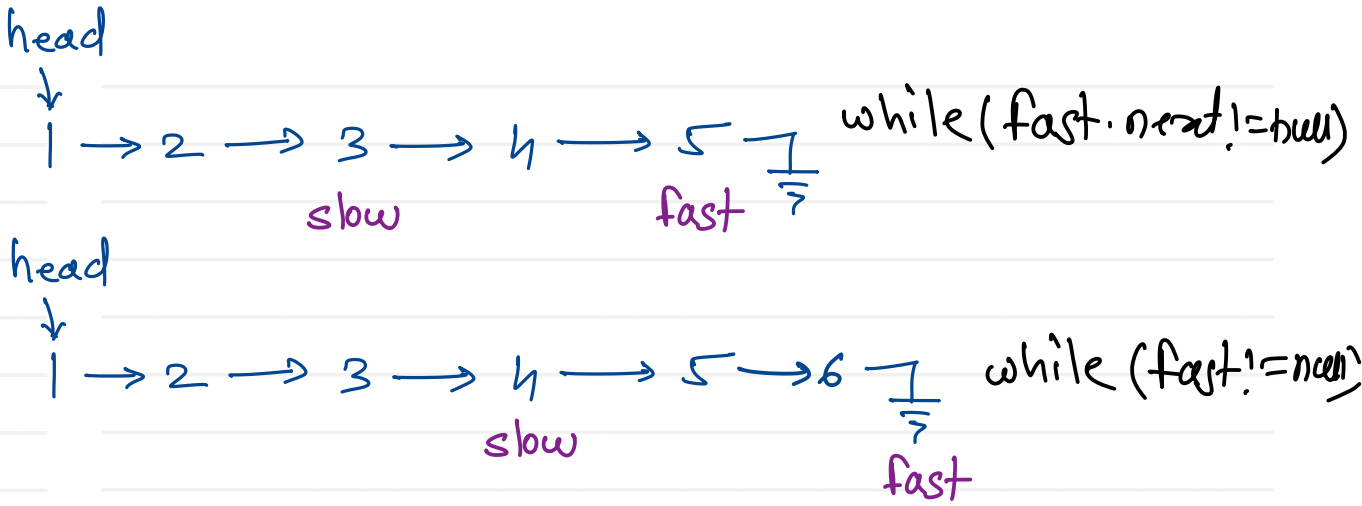
Middle of the Linked List

Given the head of a singly linked list, return the middle node of the linked list.

If there are two middle nodes, return the second middle node.

Example 1:
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.

Example 2:
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.




```
Node findMid(Node head){
    Node fast=head, slow=head;
    while(fast!=null && fast.next!=null){
        fast=fast.next.next;
        slow=slow.next;
    }
    return slow;
}
```

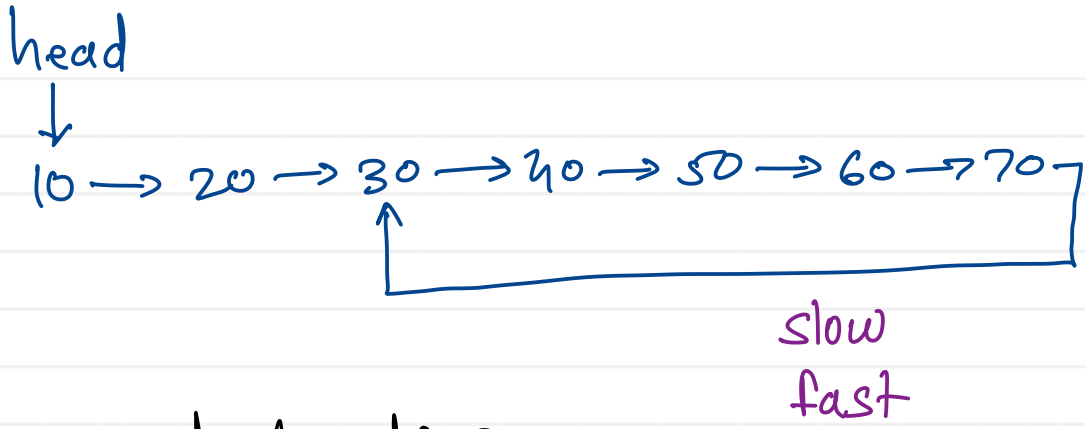
$T(n) = O(n)$

Linked List Cycle (loop)

Given head, the head of a linked list, determine if the linked list has a cycle in it.

Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.



Example 1:
Input: head = [3,2,0,-4], pos = 1
Output: true

Example 2:
Input: head = [1,2], pos = 0
Output: true

Example 3:
Input: head = [1], pos = -1
Output: false

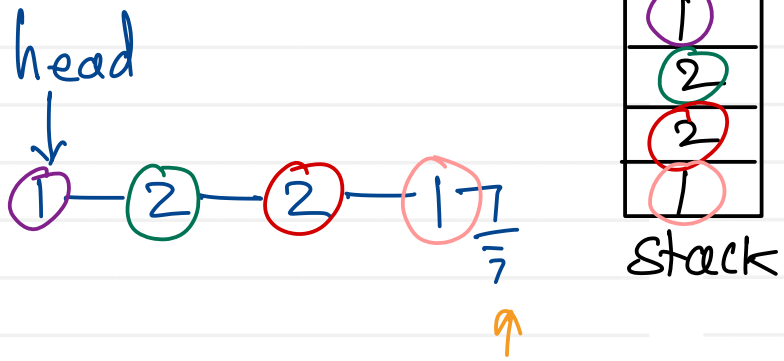
```
boolean hasCycle (Node head) {
    Node fast = head, slow = head;
    while(fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
        if(fast == slow)
            return true;
    }
    return false;
}
```

$T(n) = O(n)$

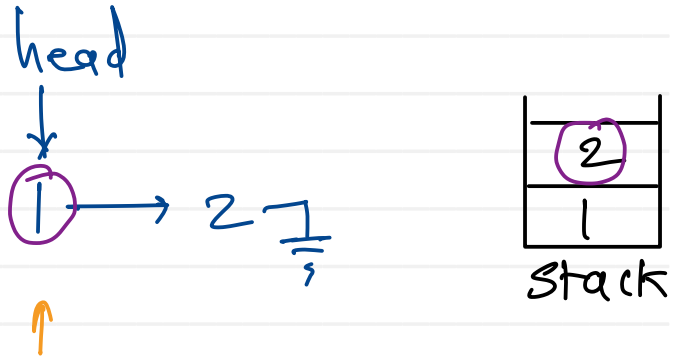
Palindrome Linked List

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:
Input: head = [1,2,2,1]
Output: true



Example 2:
Input: head = [1,2]
Output: false



T(n) = O(n)
S(n) = O(n)

```
boolean isPalindrome(Node head) {
    Stack<Integer> st = new Stack<>();
    Node trav = head;
    while (trav != null) {
        st.push(trav.data);
        trav = trav.next;
    }
    Node trav = head;
    while(!st.isEmpty()) {
        if(trav.data != st.pop())
            return false;
        trav = trav.next;
    }
    return true;
}
```

Linked list - Applications

- linked list is a dynamic data structure because it can grow or shrink at runtime.
- Due to this dynamic nature, linked list is used to implement other data structures like
  1. Stack
  2. Queue
  3. Hash table
  4. Graph

Stack

(LIFO)

1. Add first
   Delete first
2. Add last
   Delete last

Queue

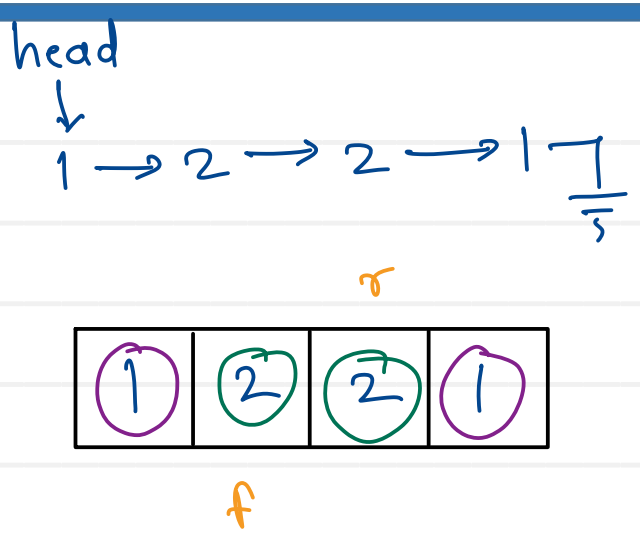
(FIFO)

1. Add first
   Delete last
2. Add Last
   Delete first

Deque

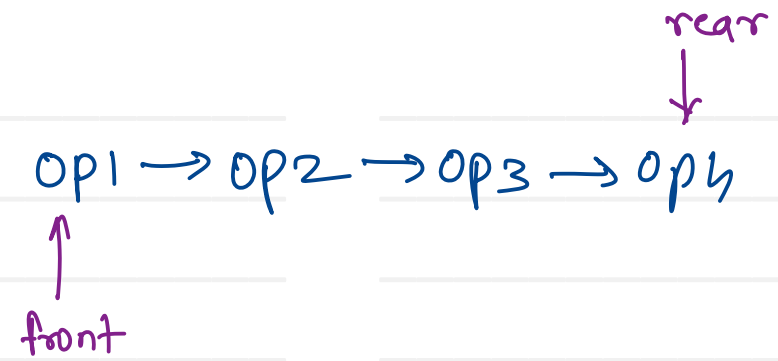
(Double Ended Queue)

1. Input restricted deque
   push is allowed from only one end
2. Output restricted deque
   pop is allowed from only one end

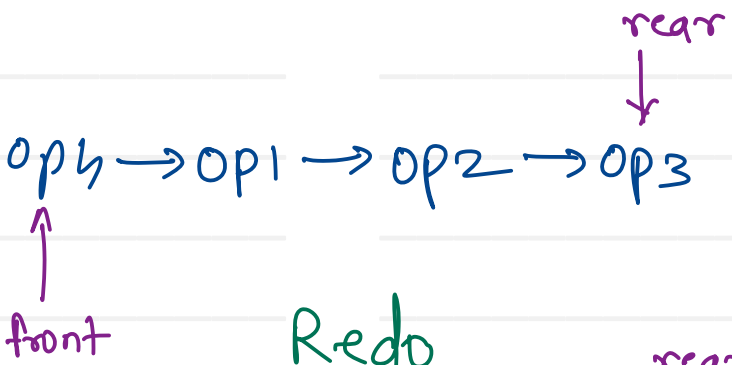


stack :
push : n
pop : n
2n
O(n)

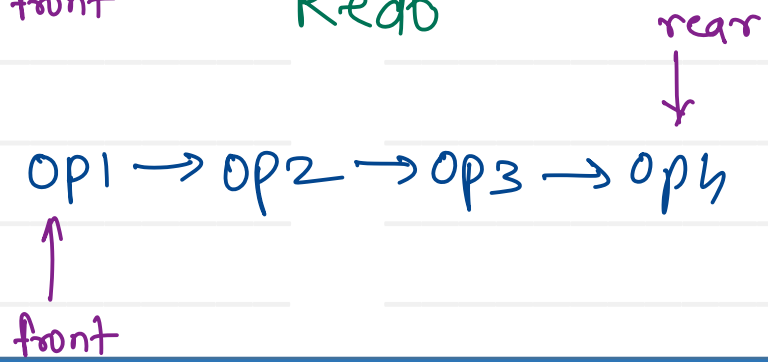
deque :
push : n
pop : 1/2 n
3/2 n
O(n)

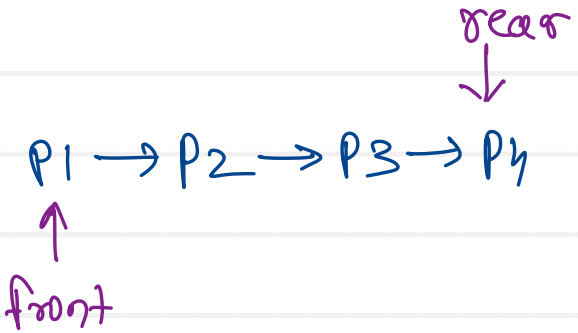


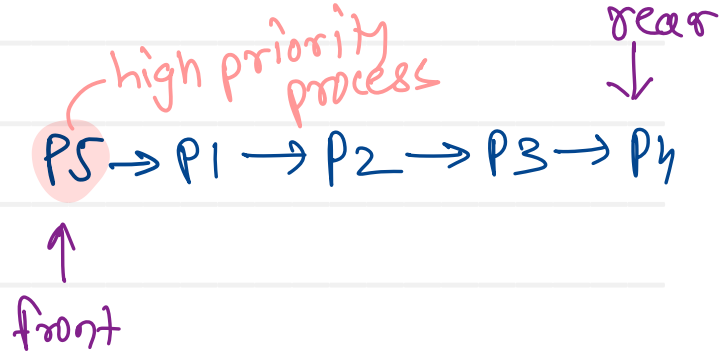
Undo



Redo







Array Vs Linked list

Array	Linked list
• Array space inside memory is continuous	• Linked list space inside memory is not continuous
• Array can not grow or shrink at runtime	• Linked list can grow or shrink at runtime
• Random access of elements is allowed	• Random access of elements is not allowed
• Insert or delete, needs shifting of array elements	• Insert or delete, need not shifting of linked list elements
• Array needs less space	• Linked list needs more space

Three pointers Technique

- Three-Pointer Technique is an extension of the Two-Pointer approach introducing an additional pointer to optimize the traversal of arrays and linked lists.

- The third pointer provides extra flexibility by allowing the algorithm
  - to track or manipulate multiple conditions simultaneously
  - making it useful for solving problems that involve triplets
  - partitioning data
  - maintaining three different states.

- The key idea is to place three pointers within a data structure and update them based on specific conditions. This allows for efficient traversal and decision-making without redundant computations.

Fast and slow pointers Technique

- Middle of the Linked list
- Nth node from the end of the Linked list
- Detect loop in the Linked list
- Find the starting point of loop in the Linked list
- Remove Loop in the Linked List

Two sum

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6

Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6

Output: [0,1]

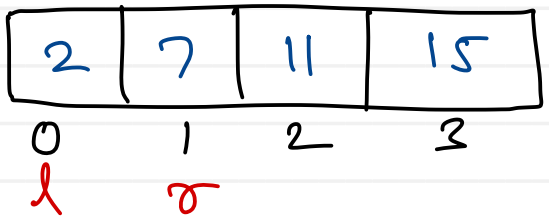
$T(n) = O(n^2)$

$S(n) = O(1)$

```
int [] twoSum( int nums[], int target){
    for( i=0 ; i< nums. length-1 ; i++) {
       for(j=i+1; j<nums.length; j++) {
          if(nums[i]+nums[j] == target)
             return new []{i,j};
       }
    }
    return new []{ };
}
```

Two pointers Technique

- The two-pointer technique is a widely used approach to solving problems efficiently, which involves arrays or linked lists.
- This method involves traversing arrays or lists with two pointers moving at different speeds or in different directions.
- This technique is used to solve problems more efficiently than using a single pointer or nested loops.



target = 9

l	r		
0	3	2+15 = 17	> 9
0	2	2+11 = 13	> 9
0	1	2+7 = 9	= 9

- Find a pair of elements that sum up to a target.
  - Array: [2, 7, 11, 15]
  - Target Sum: 9
- Use two index variables **left** and **right** to traverse from both corners.
  - Initialize: left = 0, right = n – 1
  - Run a loop while **left < right,** do the following inside the loop
- Compute current sum, **sum** = arr[left] + arr[right]
- If the **sum** equals the **target**, we've found the pair.
- If the **sum** is less than the **target**, move the **left** pointer to the right to increase the **sum**.
- If the **sum** is greater than the **target**, move the **right** pointer to the left to decrease the **sum**.

Two sum

sorted

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6
Output: [0,1]

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length-1;
    while(left < right) {
        sum = nums[left] + nums[right];
        if(sum == target)
            return new int[] {left, right};
        else if(sum < target)
            left++;
        else
            right++;
    }
    return new int[] { };
}
```

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com