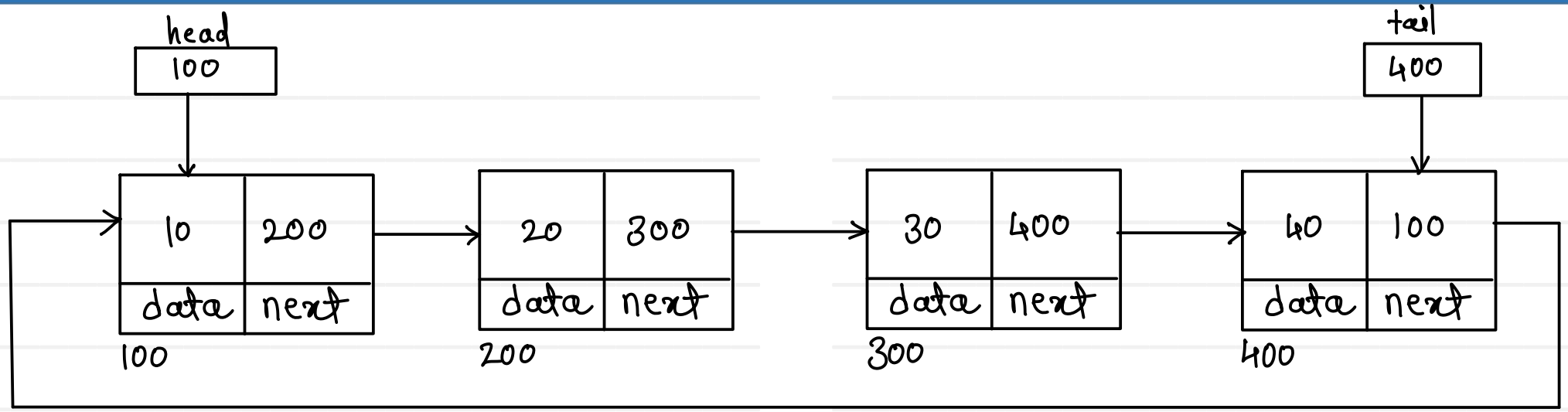
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Singly Circular Linked List - Display

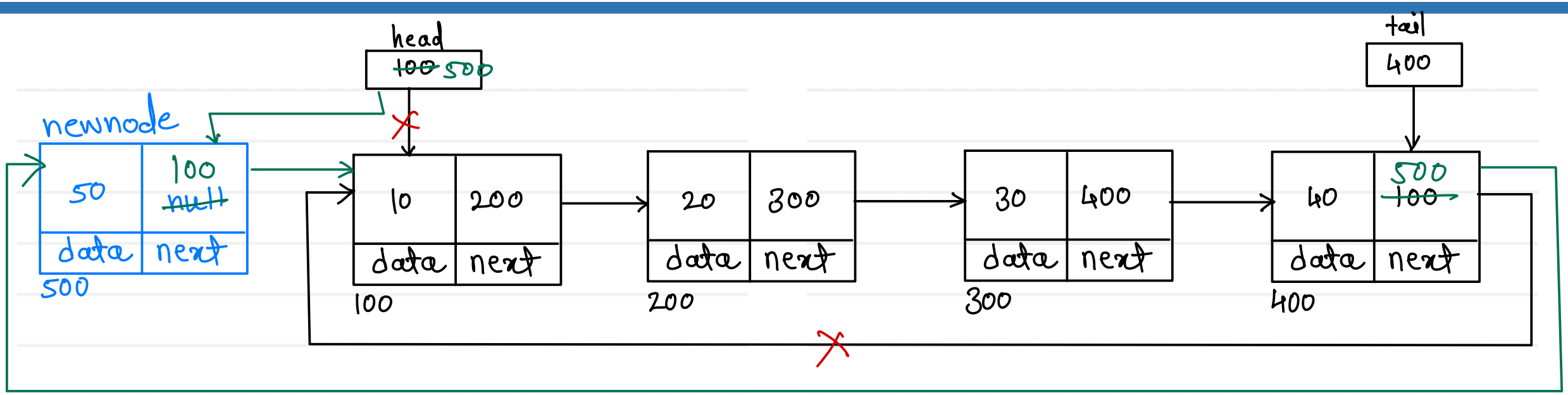


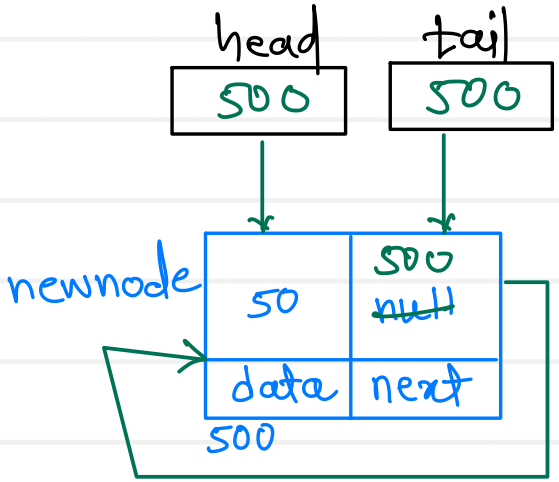
1. create trav & start at first node
2. visit / print current data
3. go on next node
4. repeat above two steps till last node

```
Node trav = head;
do {
    sysout(trav.data);
    trav = trav.next;
} while (trav != head);
```

$T(n) = O(n)$

Singly Circular Linked List - Add first



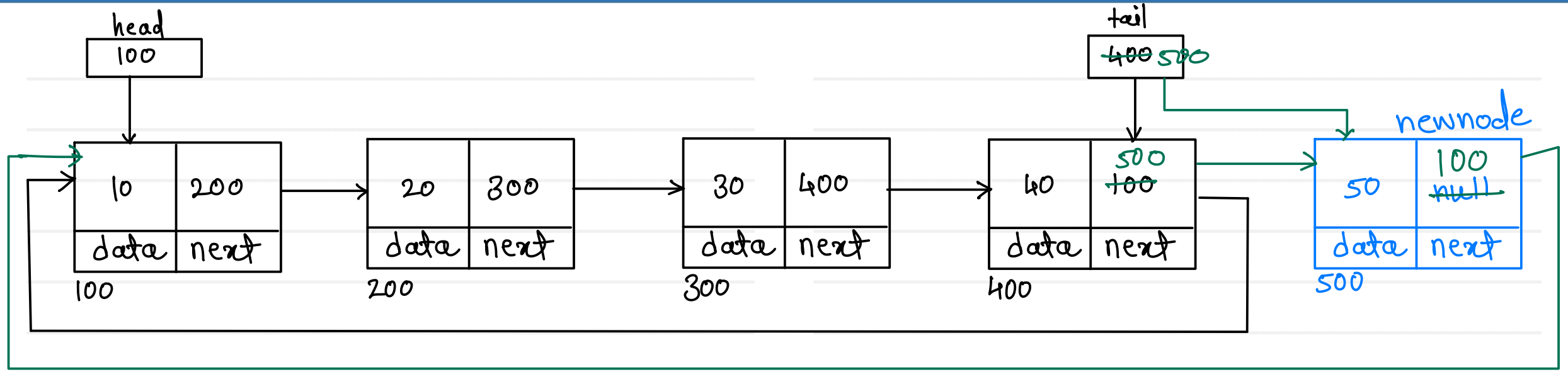


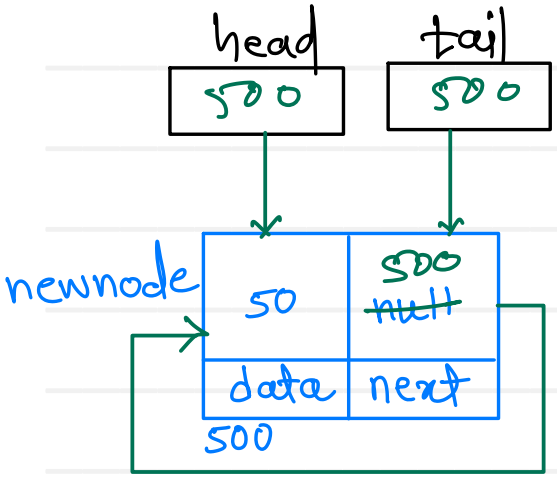
if list is empty
1. add newnode into head & tail
2. make list circular

if list is not empty
1. create new node
2. add first node into next of newnode
3. add newnode into next of last node
4. move head on newnode

$T(n) = O(1)$

Singly Circular Linked List - Add last

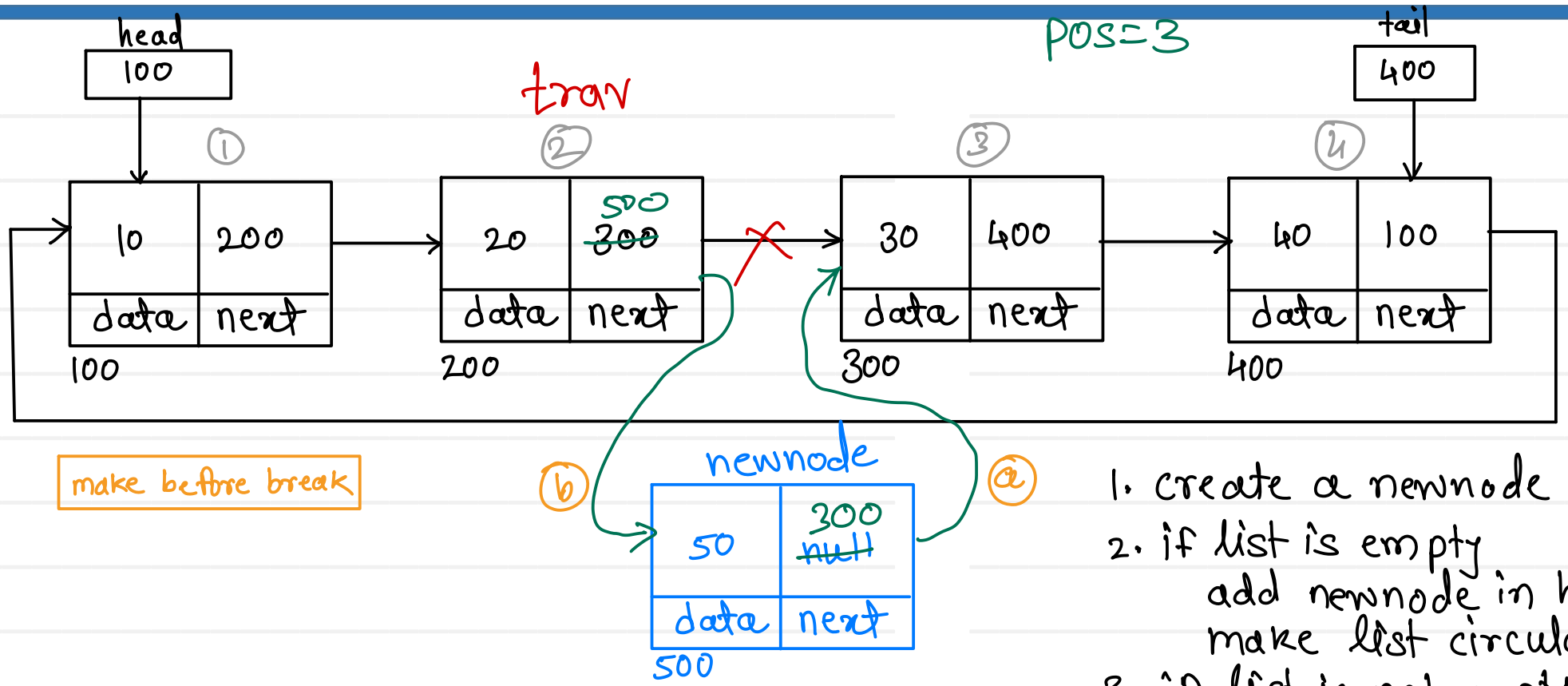




2. add first node into next of newnode
3. add newnode into next of last node
4. move tail on newnode

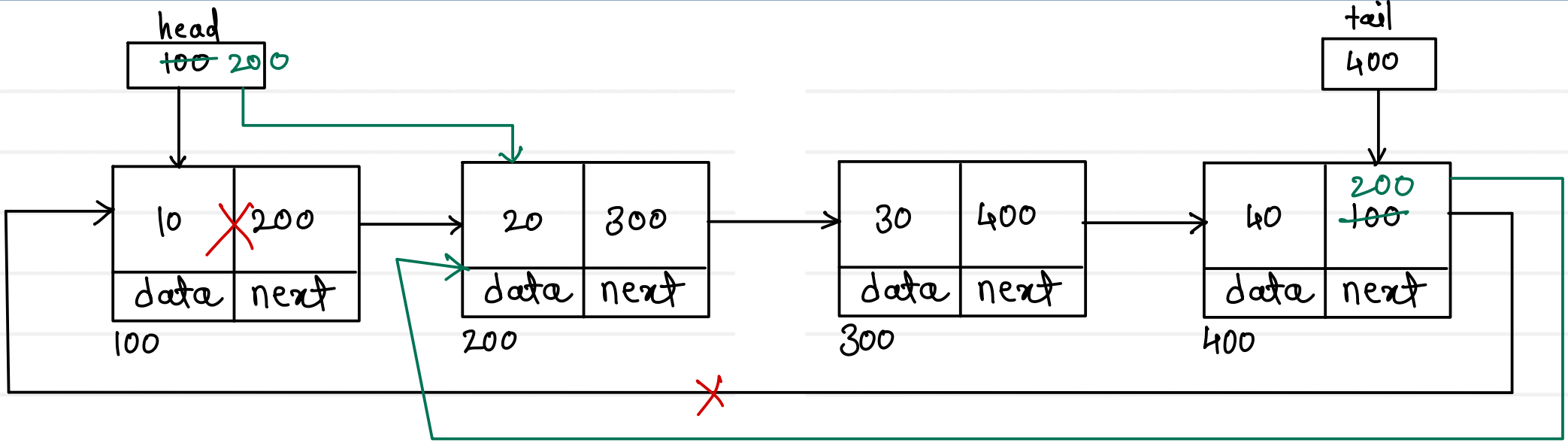
$T(n) = O(1)$

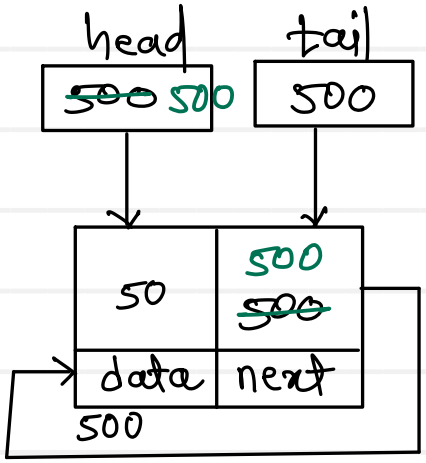
Singly Circular Linked List - Add position



1. create a newnode
2. if list is empty
   add newnode in head & tail.
   make list circular.
3. if list is not empty
   a. traverse till pos-1 node
   b. add pos node into next of newnode
   c. add newnode into next of pos-1 node

Singly Circular Linked List - Delete first



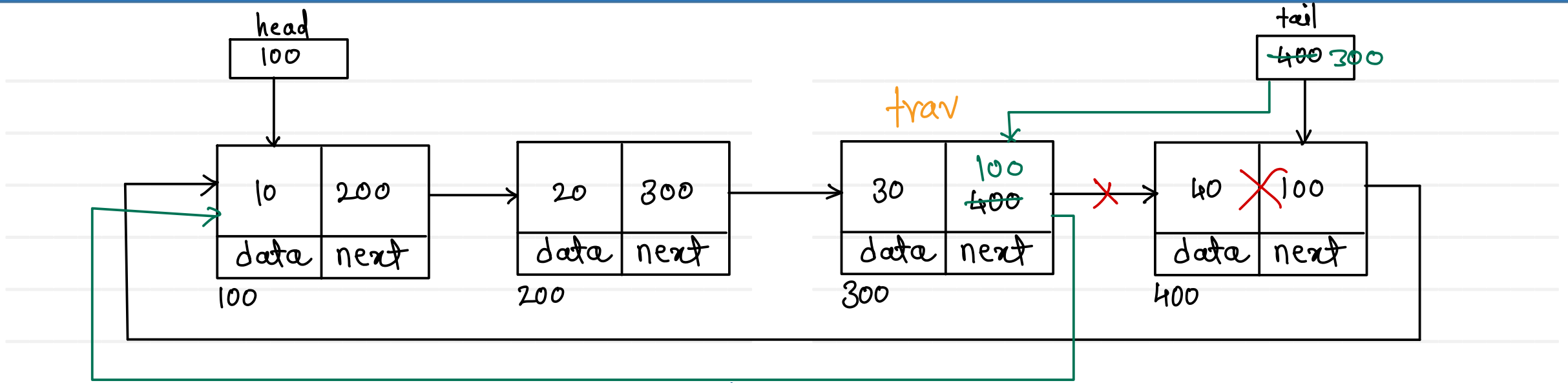


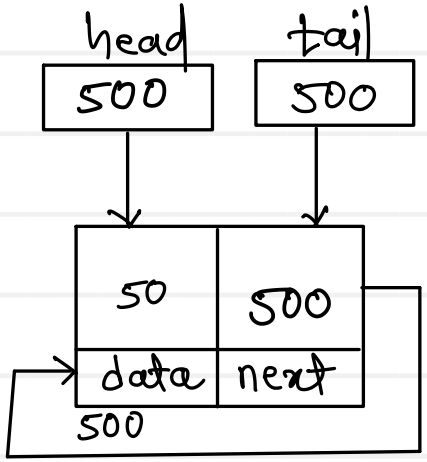
tail.next = head.next
head = head.next

1. if empty, return
2. if single node, head = tail = null
3. if multiple nodes
   a. add seconde into next of last node
   b. move head on second node

$T(n) = O(1)$

Singly Circular Linked List - Delete last






```
while ( trav.next != tail)
```

or

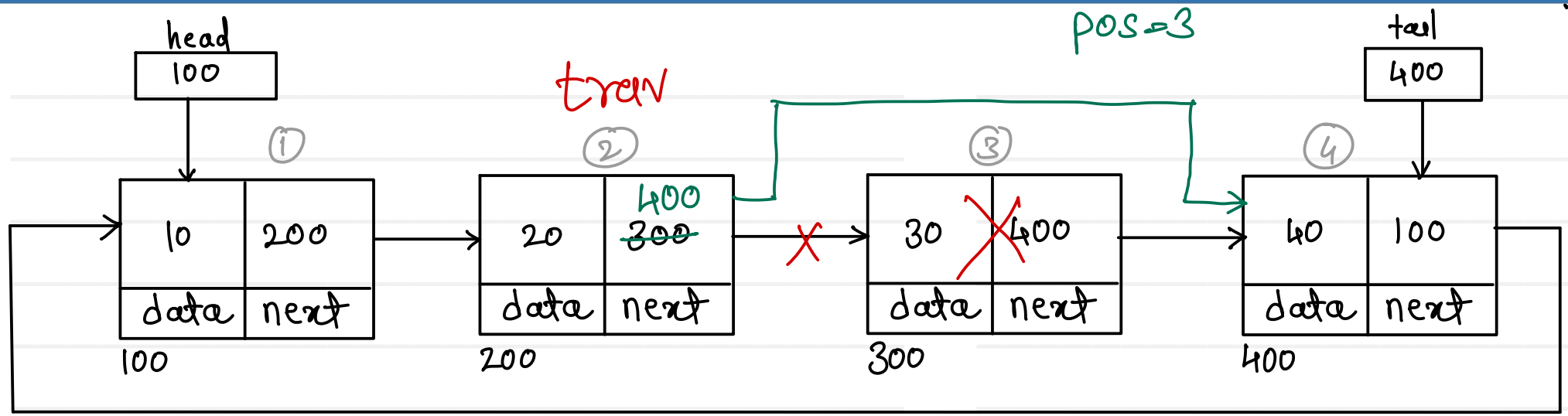
```
while ( trav.next.next != head)
    trav = trav.next;

trav.next = head;
tail = trav;
```

$T(n) = O(n)$

1. if empty, return
2. if single node, head = tail = null
3. if multiple node,
   a. traverse till second last node
   b. add first node into next of second last
   c. move tail on second last node

Singly Circular Linked List - Delete position

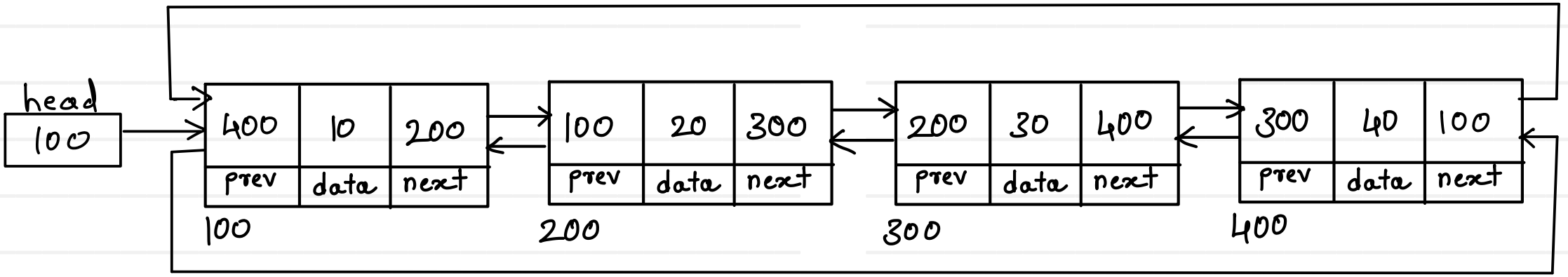


valid positions :
1 to size
invalid positions :
pos < 1
pos > size

1. if list is empty
   return
2. if list has single node
   head = tail = null.
3. if list has multiple nodes
   a. traverse till pos-1 node
   b. add pos+1 node into next of pos-1 node

$T(n) = O(n)$

Doubly Circular Linked List - Display

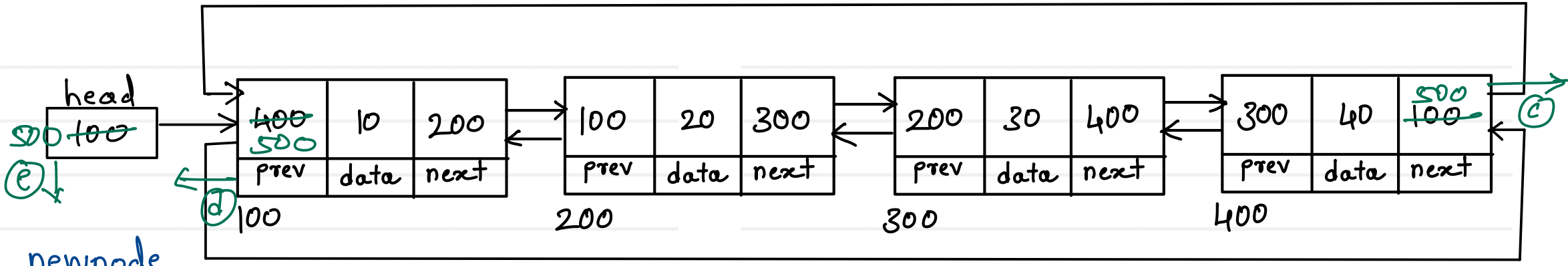


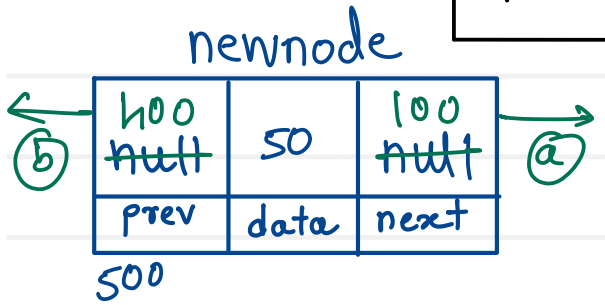
1. Create trav & start at first node
2. print current node data
3. go on next node
4. repeat step 2 & 3 till last node

1. Create trav & start at last node
2. print current node
3. go on prev node
4. repeat step 2 & 3 till first node

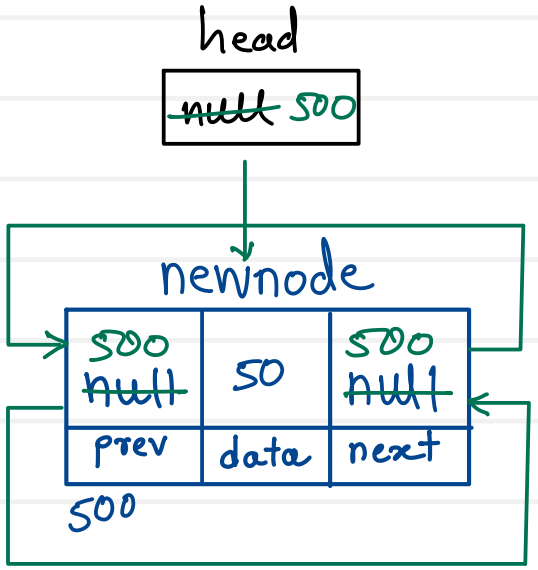
$T(n) = O(n)$

Doubly Circular Linked List - Add first



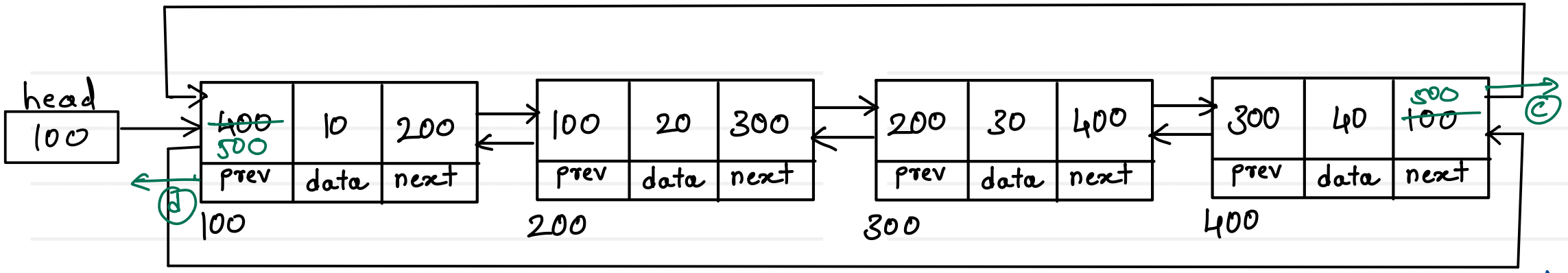


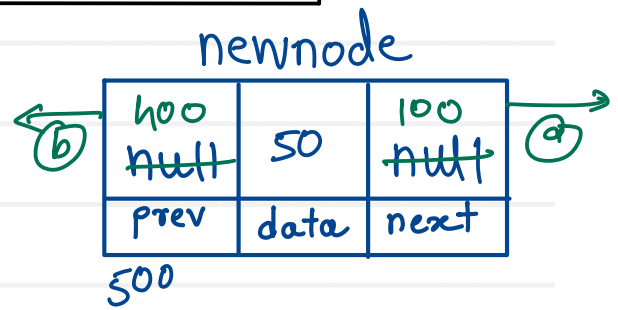
1. Create a newnode
2. if empty
   a. add newnode into head
   b. make list circular
3. if not empty
   a. add first into next of newnode
   b. add last node into prev of newnode
   c. add newnode into next of last node
   d. add newnode into prev of first node
   e. move head on newnode

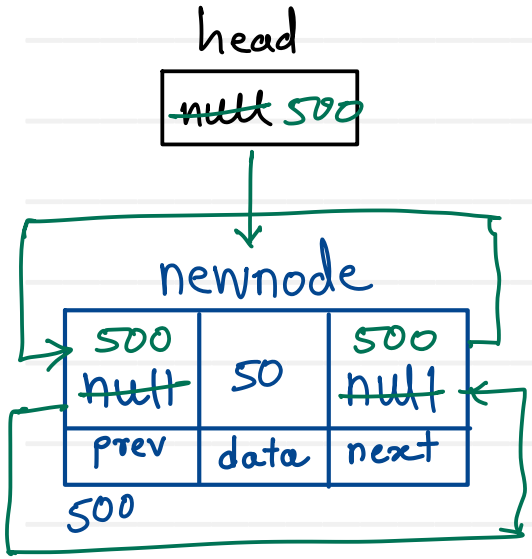


T(n)=O(1)

Doubly Circular Linked List - Add last



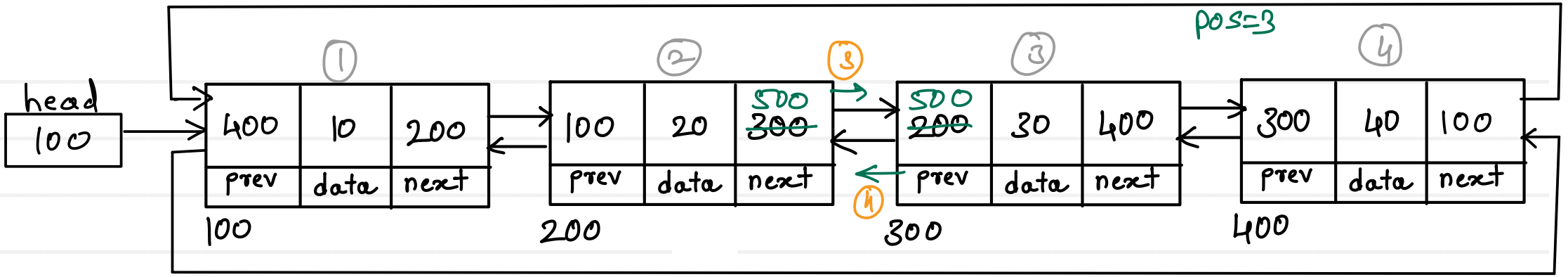




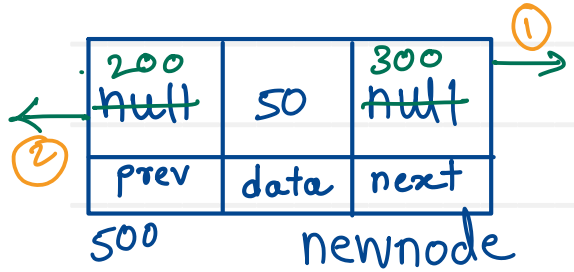
1. Create a newnode
2. if empty
   a. add newnode into head
   b. make list circular
3. if not empty
   a. add first into next of newnode
   b. add last node into prev of newnode
   c. add newnode into next of last node
   d. add newnode into prev of first node

$T(n) = O(1)$

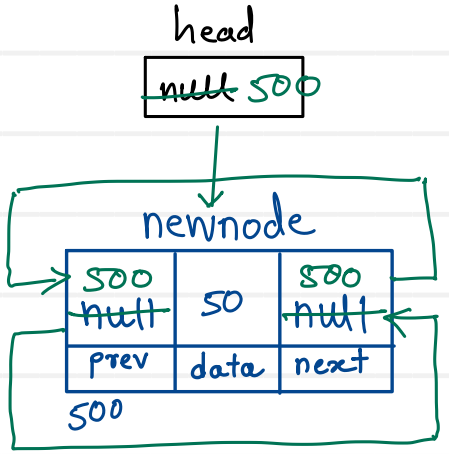
Doubly Circular Linked List - Add position



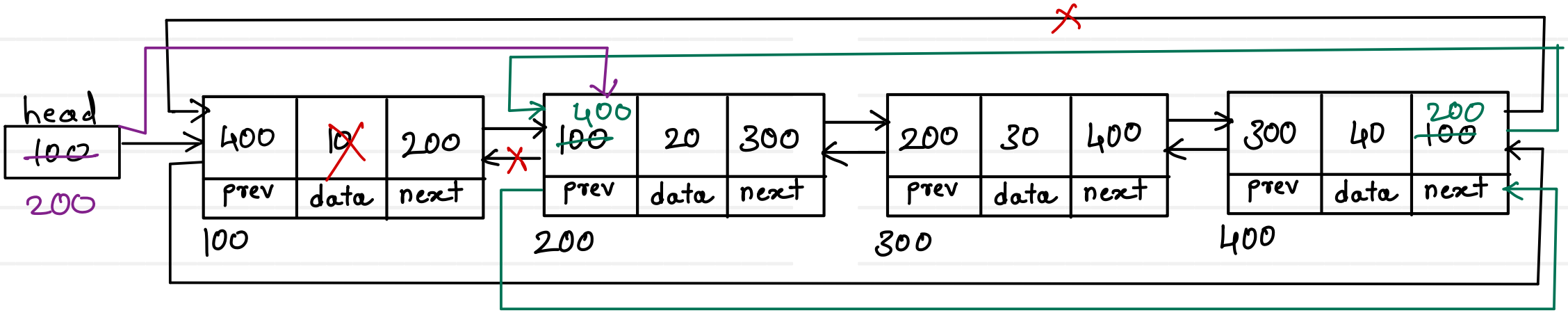
1. create node
2. if list is empty
   a. add newnode into head
   b. make list circular
3. if list is not empty.
   a. traverse till pos-1 node
   b. add pos node into next of newnode
   c. add pos-1 node into prev of newnode
   d. add newnode into next of pos-1 node
   e. add newnode into prev of pos node



$T(n) = O(1)$



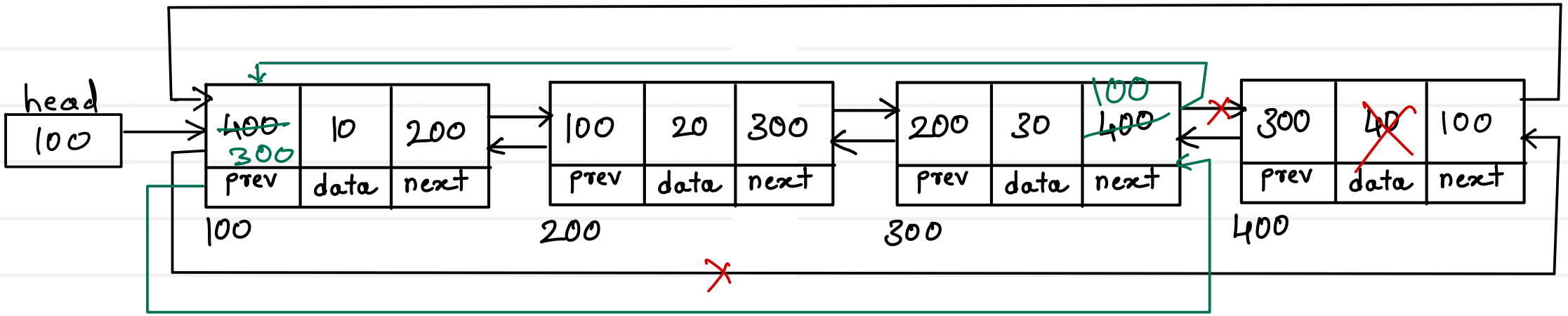
Doubly Circular Linked List - Delete first



1. add second node into next of last node
2. add last node into prev of second node
3. move head on second node

T(n)=O(1)

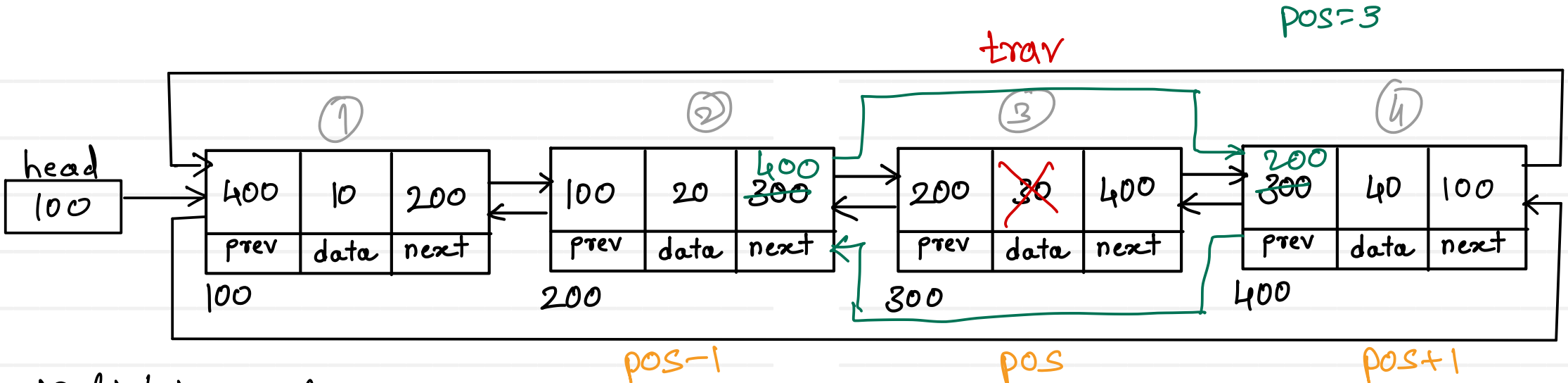
Doubly Circular Linked List - Delete last



1. add first node into next of second last node
2. add second last node into prev of first node

T(n)=O(1)

Doubly Circular Linked List - Delete position



1. if list is empty
   return;
2. if list has single node
   head = tail = null;
3. if list has multiple nodes
   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

trav → pos
trav.prev → pos-1
trav.next → pos+1

$T(n) = O(n)$

Sliding window Technique

- involve moving a fixed or variable-size window through a data structure, to solve problems efficiently.
- This technique is used to find subarrays or substrings according to a given set of conditions.
- This method used to efficiently solve problems that involve defining a **window** or **range** in the input data and then moving that window across the data to perform some operation within the window.
- This technique is commonly used in algorithms like
  - **finding subarrays** with a specific sum
  - **finding the longest substring** with unique characters
  - solving problems that require a fixed-size window to process elements efficiently.

- There are two types of sliding window
  - **Fixed size sliding window**
    - Find the size of the window required
    - Compute the result for 1st window
    - Then use a loop to slide the window by 1 and keep computing the result
  - **Variable size sliding window**
    - increase right pointer one by one till our condition is true.
    - At any step if condition does not match, shrink the size of window by increasing left pointer.
    - Again, when condition satisfies, start increasing the right pointer
    - follow these steps until reach to the end of the array

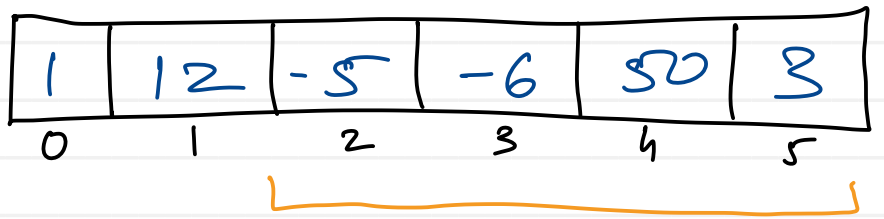
Maximum Average Subarray

You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose length is equal to k that has the maximum average value and return this value. Any answer with a calculation error less than 10-5 will be accepted.

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000
Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75

Example 2:
Input: nums = [5], k = 1
Output: 5.00000



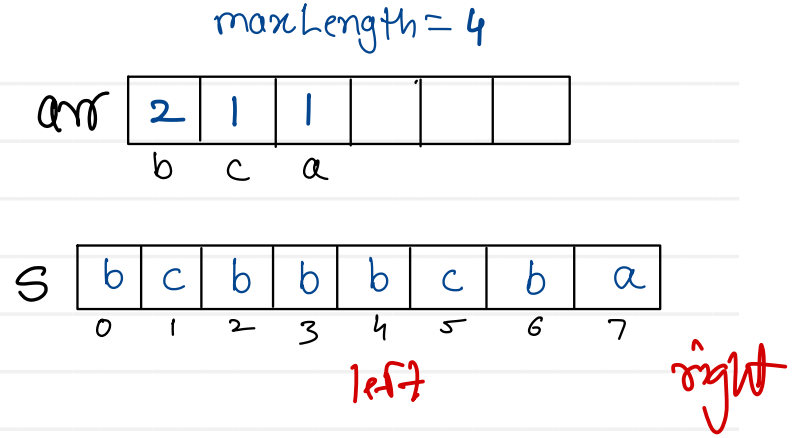
1. maxSum = 1+12-5-6 = 2
2. slide window by 1
3. currSum = currSum − left element of previous window + right element of new window
   = 2 − 1 + 50
   = 51
4. if currSum is greater than maxSum
   maxSum = currSum

Maximum Length Substring With Two Occurrences

Given a string s, return the maximum length of a substring such that it contains at most two occurrences of each character.

Example 1:
Input: s = "bcbbbcba"
Output: 4
Explanation: The following substring has a length of 4 and contains at most two occurrences of each character: "bcbbbcba".

Example 2:
Input: s = "aaaa"
Output: 2
Explanation: The following substring has a length of 2 and contains at most two occurrences of each character: "aaaa".




```
int maxLength = 0;
int start=0, end=0;
int[] arr = new int[26];
for( ; end < s.length() ; end++){
    arr[s.charAt(end) - 'a']++;
    while(arr[s.charAt(end) - 'a']==3){
        arr[s.charAt(start) - 'a']--;
        start++;
    }
    maxLength = Math.max(maxLength, end-start+1);
}
return maxLength;
}
```

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]

HashMap

key	value
2	0

Example 2:

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

Output: [1,2]

HashMap

key	value
3	0
2	1

Example 3:

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

Output: [0,1]

```
int[] twoSum(int[] nums, int target){
    Map<Integer,Integer> tbl = new HashMap<>();
    for(int i=0; i<nums.length; i++){
        if(tbl.containsKey(target-nums[i]))
            return new int[]{tbl.get(target-nums[i]),i};
        tbl.put(nums[i],i);
    }
    return new int[]{};
}
```

6

3

6-3=3

Hashing

Array :
Linear search – O(n)
Binary search – O(log n)

Linked List :
Linear search – O(n)

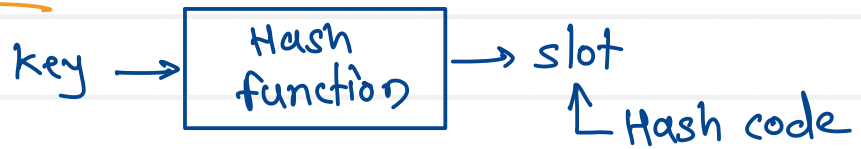
- Hashing is a technique
- implementation of hashing is Hash table

Hash table :
search : O(1)

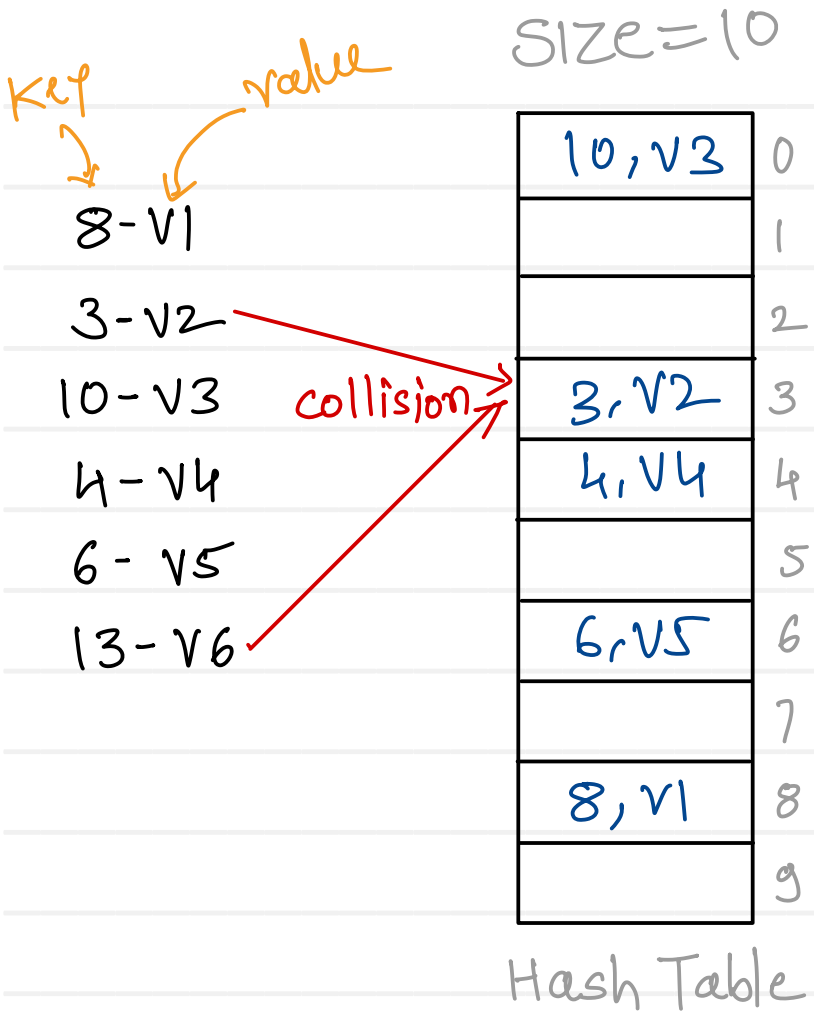
- hashing is a technique in which data can be inserted, deleted and searched in constant average time O(1)
- Implementation of hashing is known as hash table
- Hash table is array of fixed size in which elements are stored in key - value pairs

Array - Hash table
Index - Slot

- In hash table only unique keys are stored
- Every key is mapped with one slot of the table and this is done with the help of mathematical function known as hash function



Hashing



h(k) = k % size ← hash function

h(8) = 8 % 10 = 8
h(3) = 3 % 10 = 3
h(10) = 10 % 10 = 0
h(4) = 4 % 10 = 4
h(6) = 6 % 10 = 6
h(13) = 13 % 10 = 3

insert: O(1)
1. find slot
2. arr[slot] = data

search: O(1)
1. find slot
2. return arr[slot]

remove: O(1)
1. find slot
2. arr[slot] = null

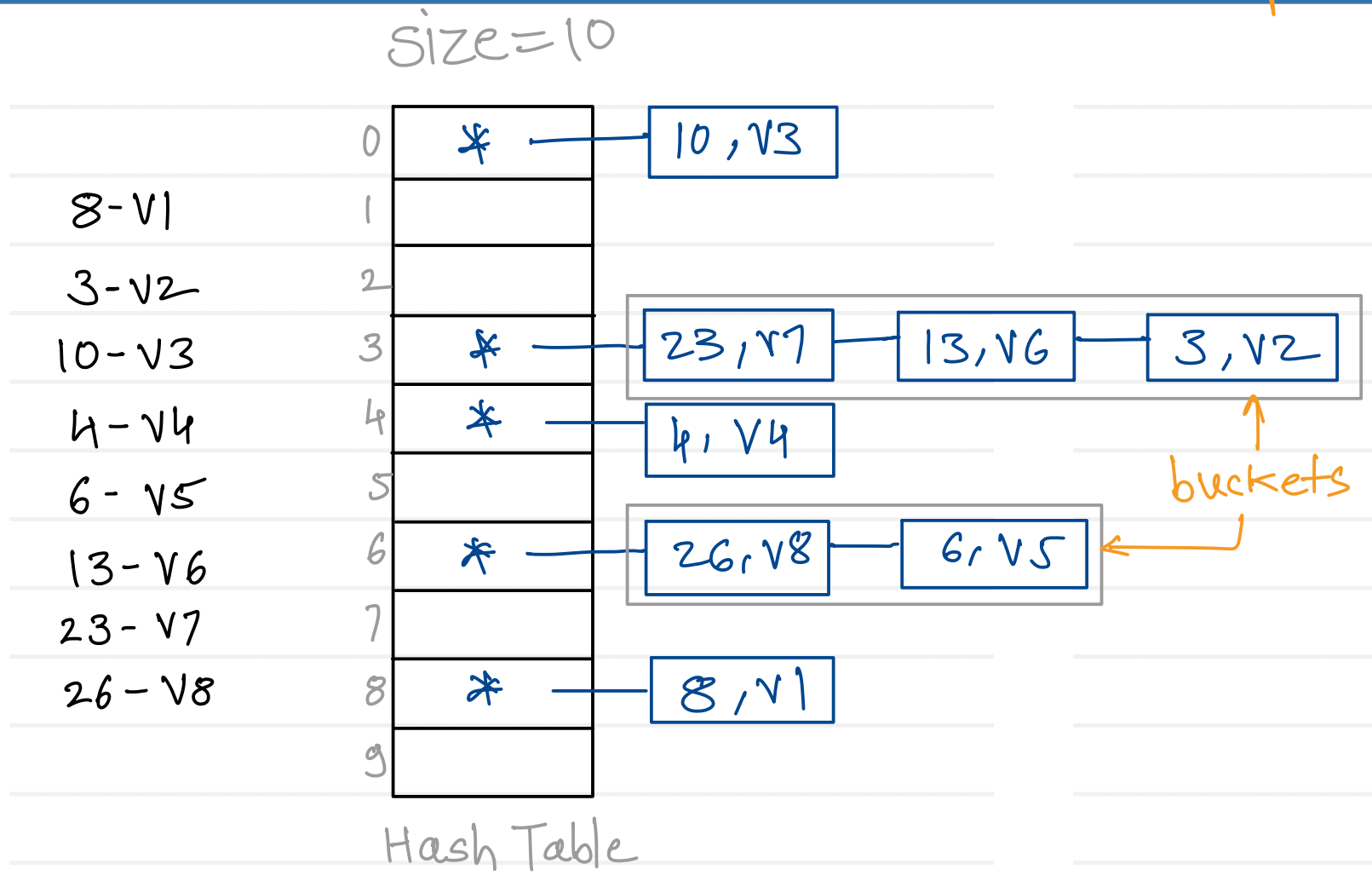
Collision:
it is a situation when two distinct key gives/ yeilds same slot

Collision handling/resolution Techniques
1. Closed Addressing
2. Open Addressing
   i. linear probing
   ii. Quadratic probing
   iii. Double hashing

Closed Addressing / Chaining / Separate Chaining

per slot linked list

h(k) = k % size



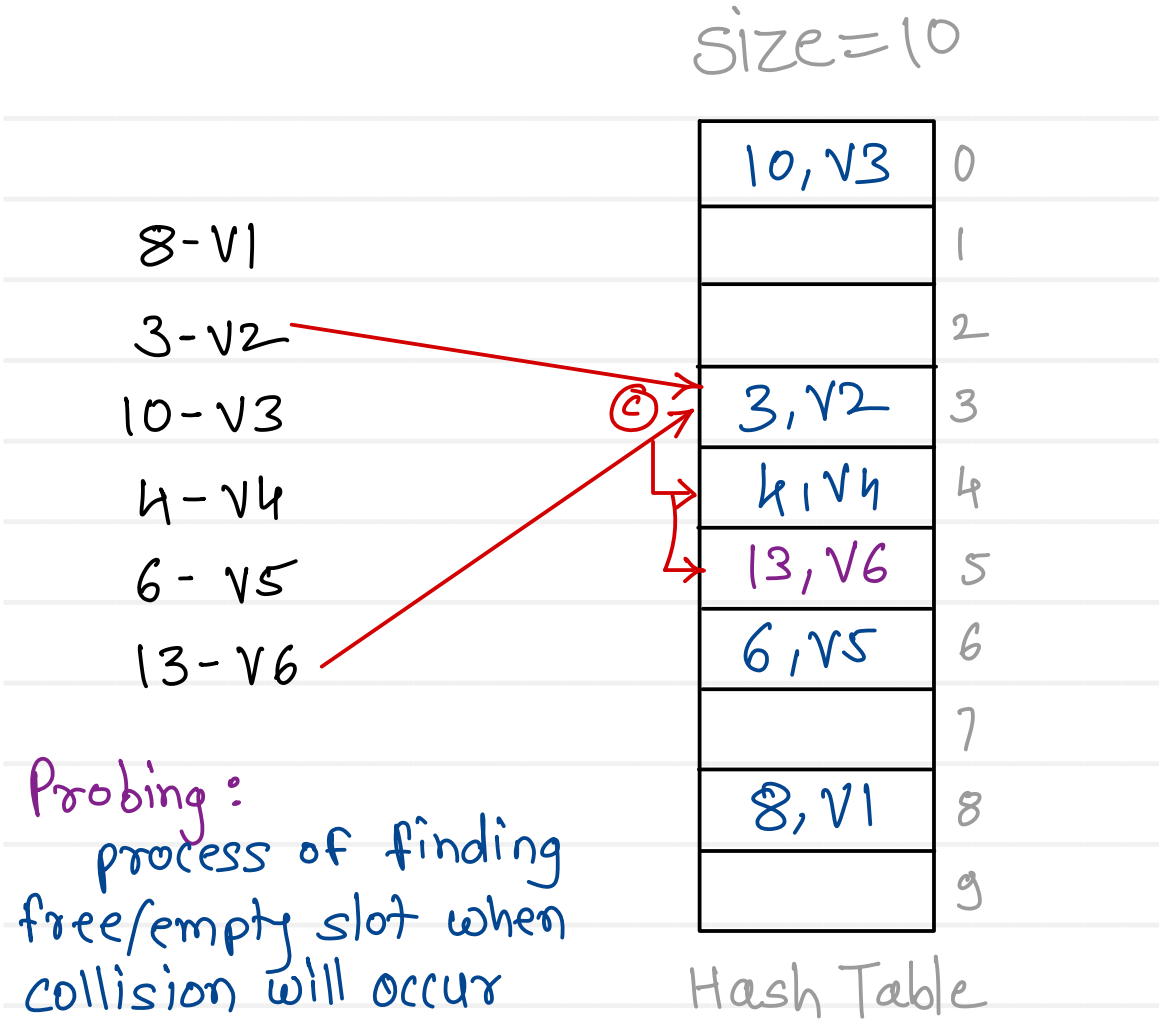
Advantage:

- no limitation on number of key-value pairs

Disadvantages:

- data is stored outside the table
- space requirement is more
- Worst case time complexity is O(n)
  - when multiple keys yield same slot.

Open addressing - Linear probing



Probing :
process of finding free/empty slot when collision will occur

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i ← probe number

where i = 1, 2, 3, …

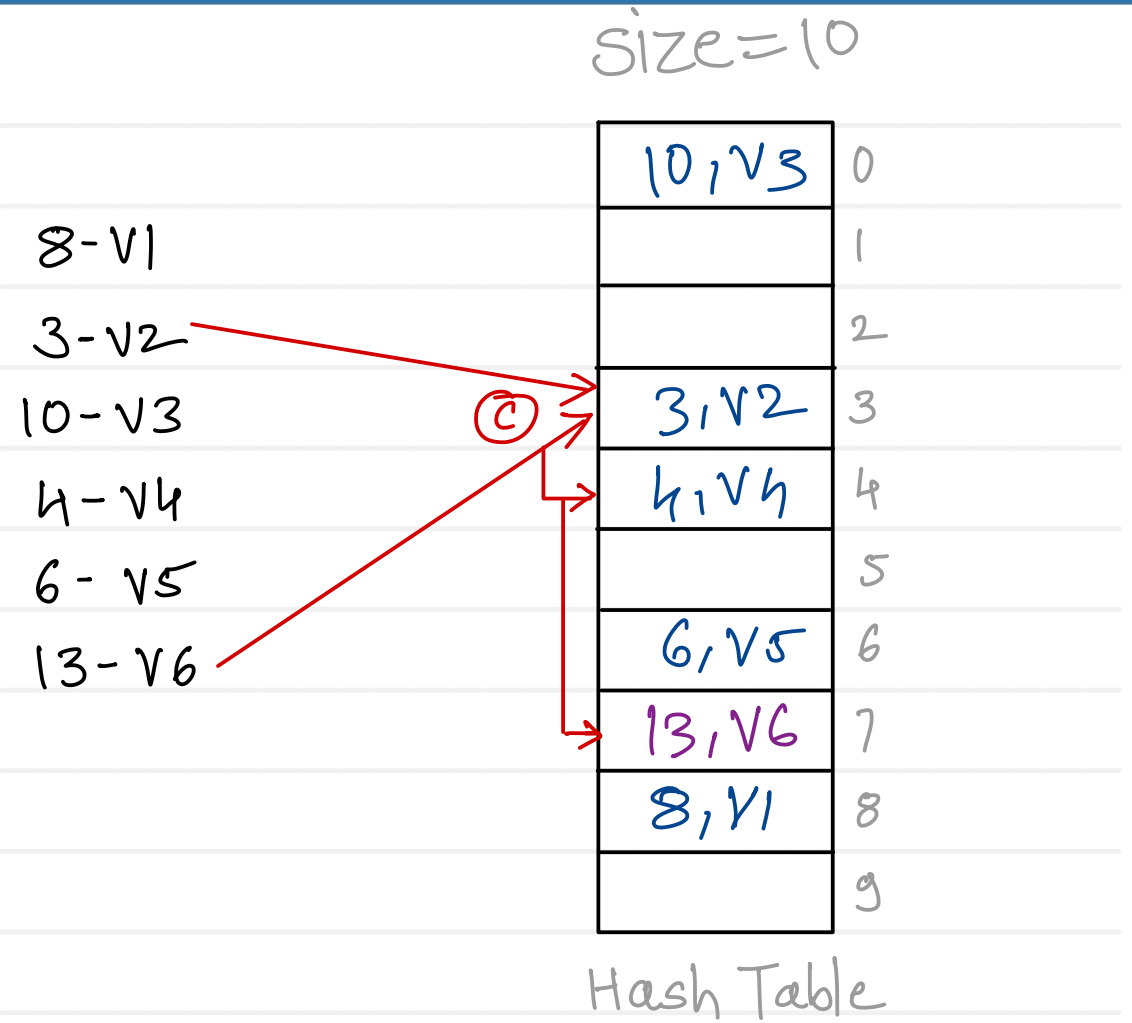
h(13) = 13 % 10 = 3 ©

h(13,1) = [h(13) + f(1)] % 10
= [3 + 1] % 10 = 4 (1st probe) ©

h(13,2) = [3 + 2] % 10 = 5 (2nd probe)

Primary clustering :
- need to take long run of filled slots "near" key position to find empty slot
- near key position table is bulky/ crowded.

Open addressing - Quadratic probing



h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, …

h(13) = 13 % 10 = 3 (c)

h(13,1) = [3 + 1] % 10 = 4 (1st) (c)

h(13,2) = [3 + 4] % 10 = 7 (2nd)

- primary clustering is solved

Secondary clustering:

- need to take long run of filled slots "away" key position to find empty slot

- no garantee of getting free slot for key value pair.

Open addressing - Quadratic probing

size=10

	10, V3	0
8-V1		1
3-V2	23, V7	2
10-V3	3, V2	3
4-V4	4, V4	4
6-V5		5
13-V6	6, V5	6
23, V7	13, V6	7
33, V8	8, V1	8
	33, V8	9

Hash Table

h(k) = k % size

h(k, i) = [h(k) + f(i)] % size

f(i) = i^2

where i = 1, 2, 3, …

h(23) = 23 % 10 = 3 (c)

(1) h(23,1) = [3+1] % 10 = 4 (c)

(2) h(23,2) = [3+4] % 10 = 7 (c)

(3) h(23,3) = [3+9] % 10 = 2

h(33) = 23 % 10 = 3 (c)

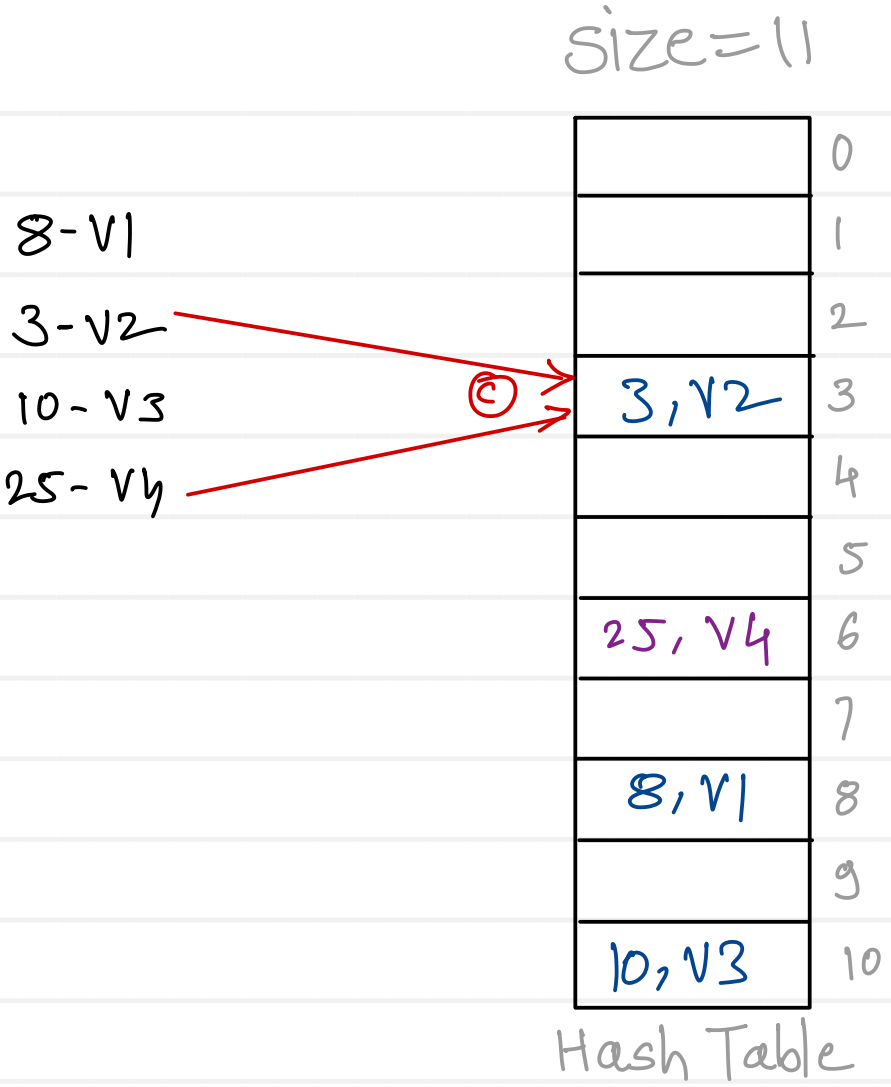
(1) h(33,1) = [3+1] % 10 = 4 (c)

(2) h(33,2) = [3+4] % 10 = 7 (c)

(3) h(33,3) = [3+9] % 10 = 2 (c)

(4) h(33,4) = [3+16] % 10 = 9

Open addressing - Double hashing



h1(k) = k % size

h2(k) = 7 - (k % 7)

h(k, i) = [h1(k) + i * h2(k)] % size

$h_1(8) = 8 \% 11 = 8$

$h_1(3) = 3 \% 11 = 3$

$h_1(10) = 10 \% 11 = 10$

$h_1(25) = 25 \% 11 = 3$ (C)

$h_2(25) = 7 - (25 \% 7) = 3$

(I) $h(25,1) = [3 + 1 * 3] \% 11 = 6$

$h_1(36) = 3$

$h_2(36) = 6$

$h(36,1) = [3 + 1 * 6] \% 11 = 9$

Rehashing

$$\text{Load factor} = \frac{n}{N}$$

(λ)

n - number of elements (key-value) present in hash table
N - number of total slots in hash table

$N = 10$
$n = 6$
$\lambda = \frac{n}{N} = \frac{6}{10} = 0.6$

Hash table is 60 % filled

- Load factor ranges from 0 to 1.
- If n < N Load factor < 1 - free slots are available
- If n = N Load factor = 1 - free slots are not available

- In rehashing, whenever hash table will be filled more than 60 or 70 % size of hash table is increased by twice
- Existing key value pairs are remapped according to new size

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com