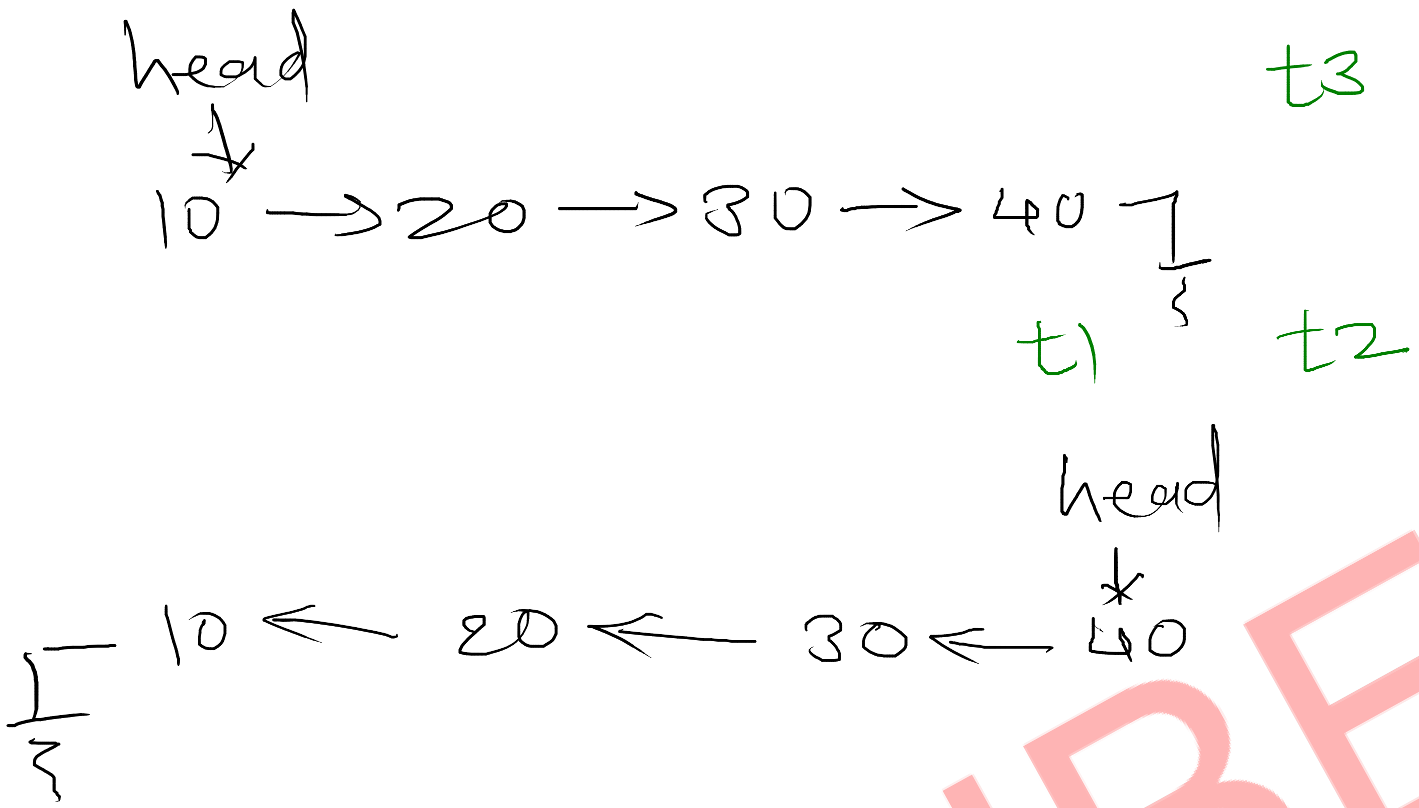
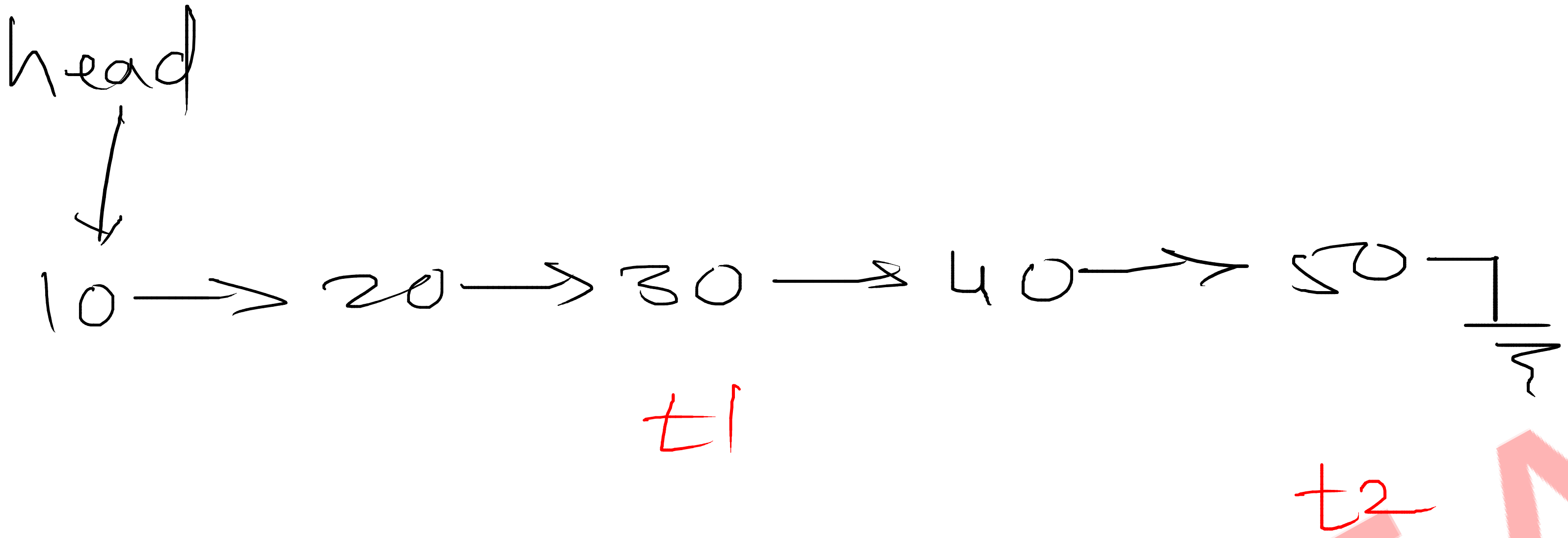
Singly Linear Linked List - Reverse List



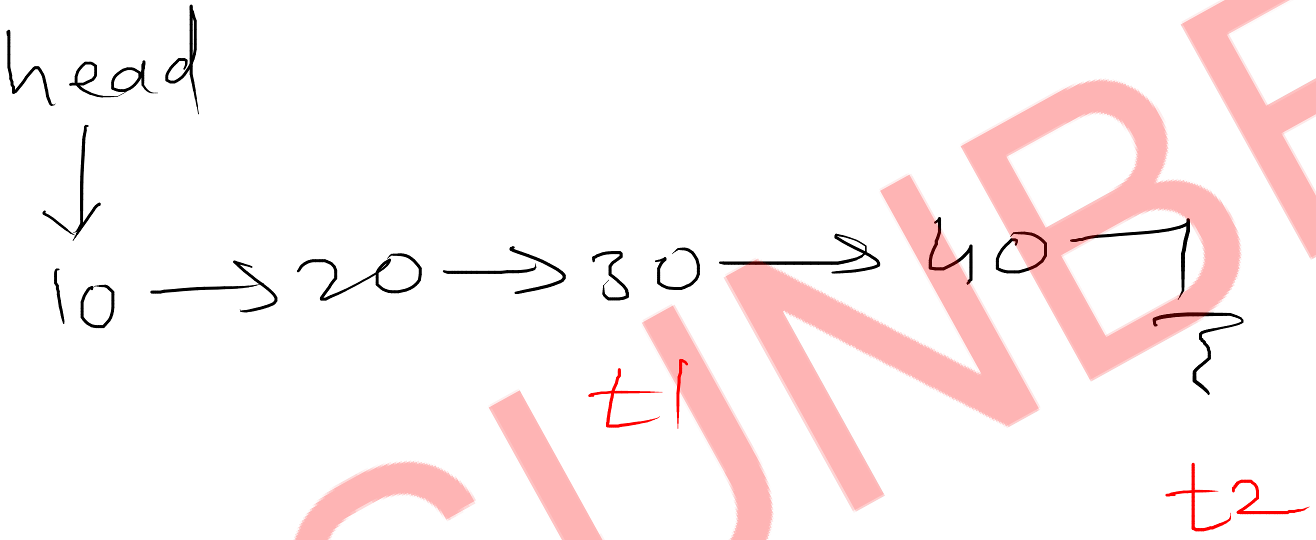
```
void reverseList( ){
    t1 = head;
    t2 = head.next;
    while(t2 != null){
        t3 = t2.next;
        t2.next = t1;
        t1=t2
        t2=t3
    }
    head.next = null
    head = t1;
}
```

$T(n) = O(n)$

Singly Linear Linked List - Find mid



```
t1 = t2 = head;
while(t2.next != null)
{
    t2 = t2.next.next;
    t1 = t1.next;
}
```

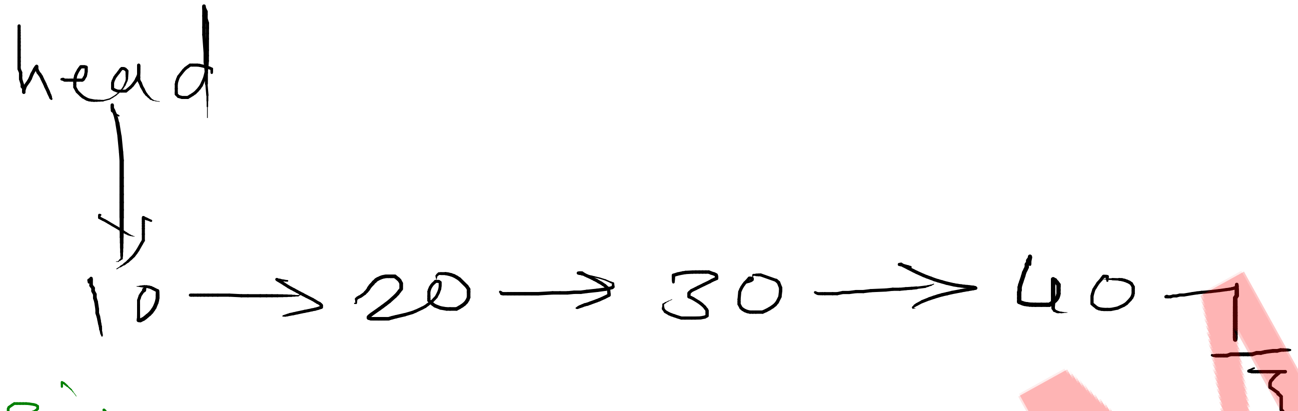


```
t1 = t2 = head;
while( t2 != null && t2.next != null)
{
    t2 = t2.next.next;
    t1 = t1.next;
}
```

$T(n) = O(n)$

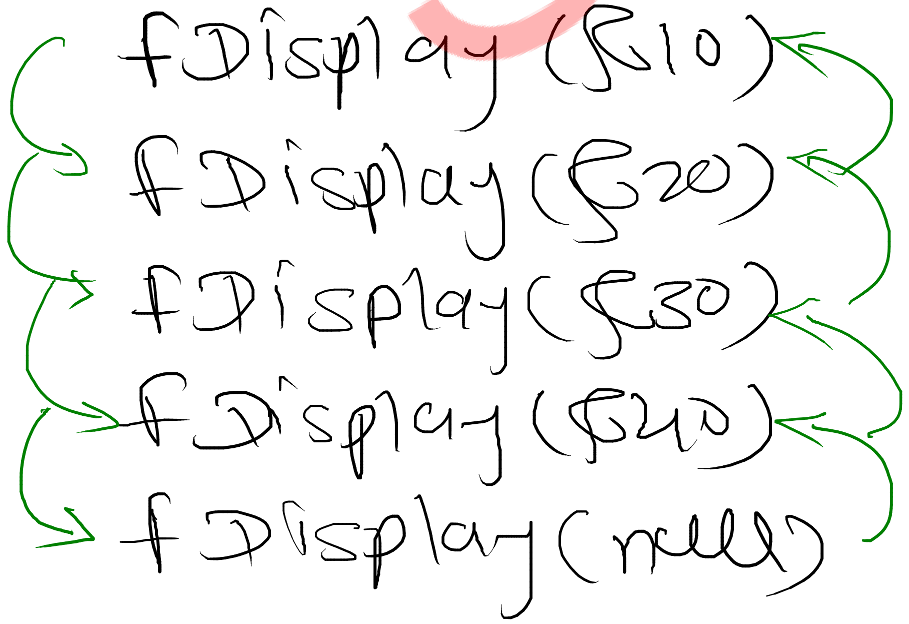
$S(n) = O(1)$

Singly Linear Linked List - Reverse Display



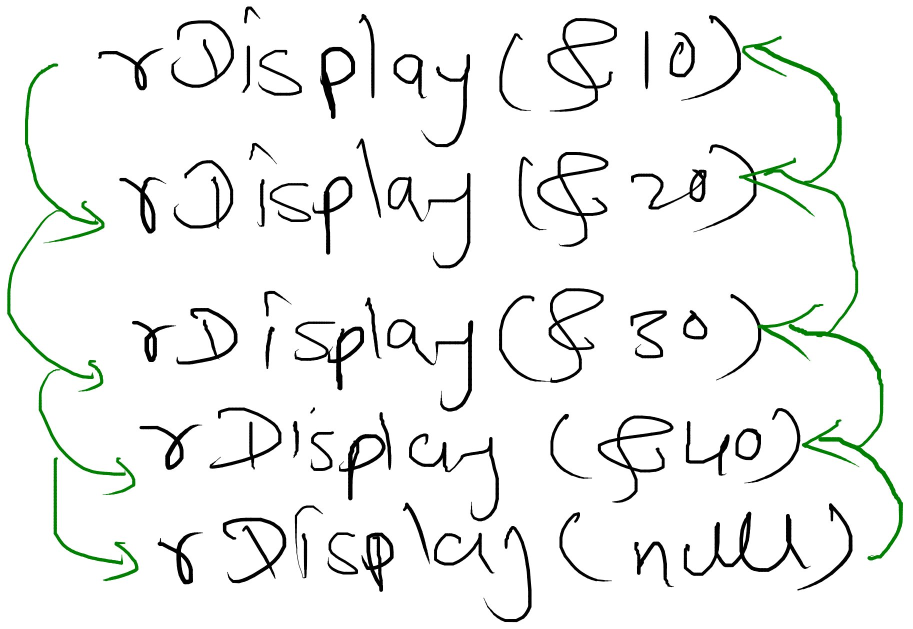
Tail Recursion

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

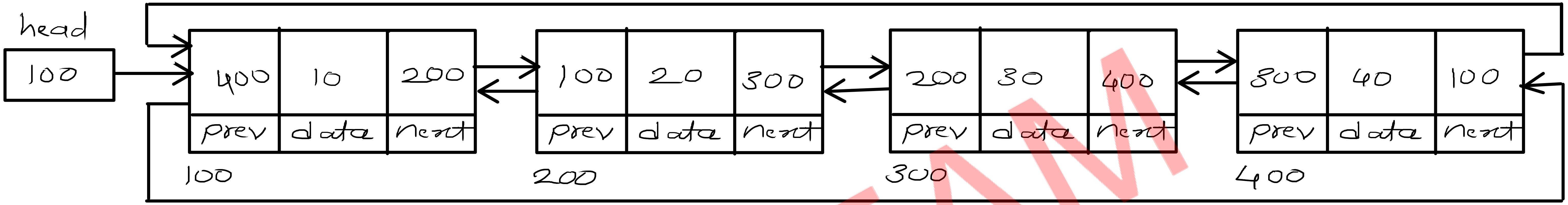


Non-tail recursion

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



Doubly Circular Linked List - Display

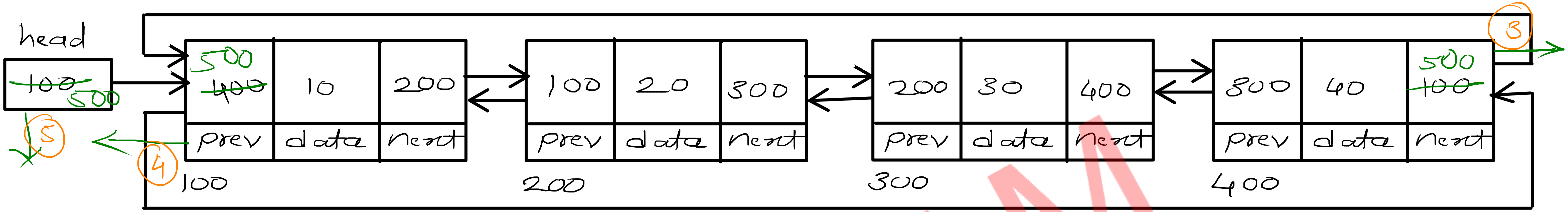


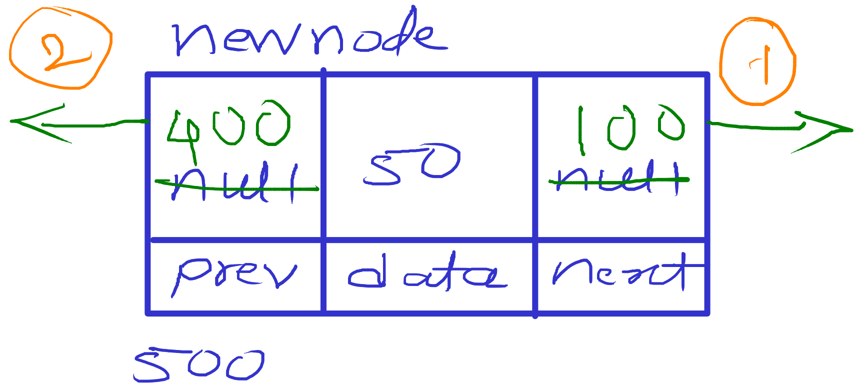
//1. create a trav and start at first node
//2. print data of current node (trav.data)
//3. go on next node
//4. repeat step 2 and 3 till last node

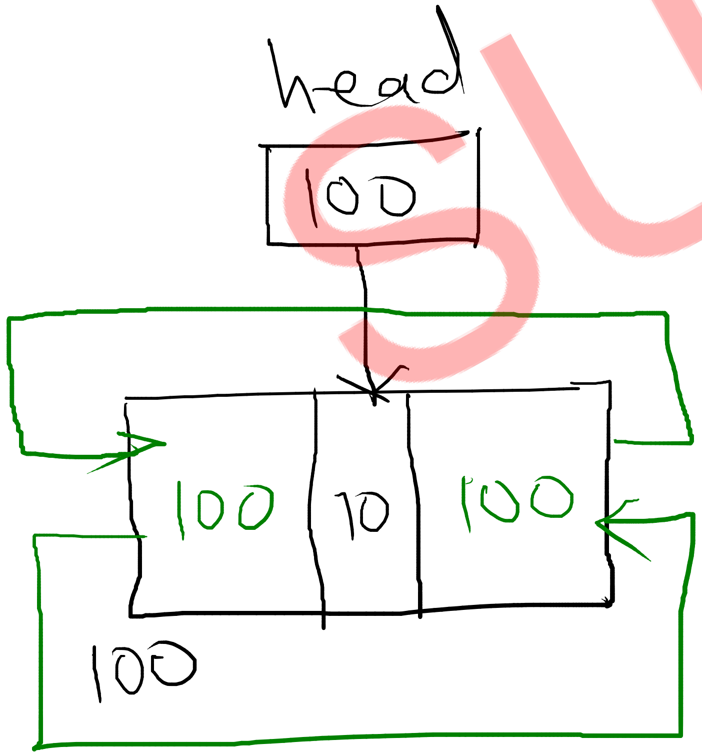
//1. create a trav and start at last node
//2. print data of current node (trav.data)
//3. go on prev node
//4. repeat step 2 and 3 till first node

$T(n) = O(n)$

Doubly Circular Linked List - Add first



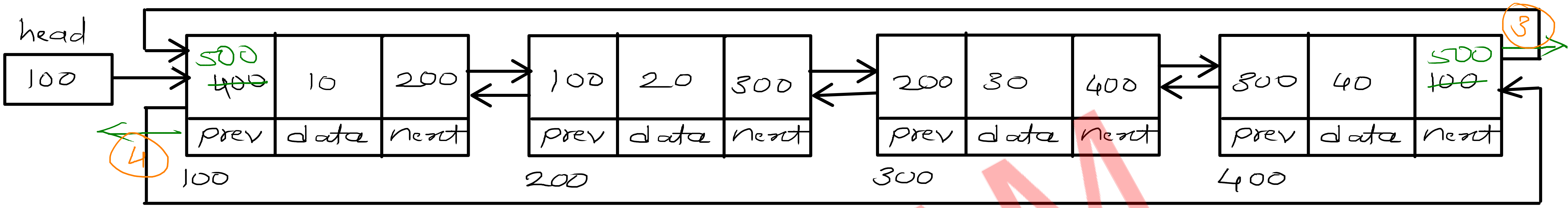




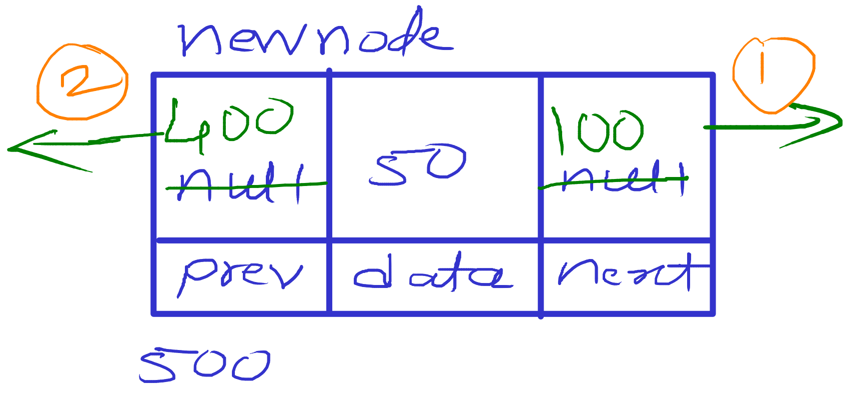
```
//1. create newnode
//2. if list is empty
        //a. add newnode into head
        //b. make list circular
//3. if list is not empty
        //a. add first node into next of newnode
        //b.  add last node into prev of newnode
        //c. add newnode into next of last node
        //d. add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

$T(n) = O(1)$

Doubly Circular Linked List - Add Last



```
//1. creat node
//2. if list is empty
        //a. add newnode into head
        //b. make list circular
//3. if list is not empty
        //a. add first node 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
```



after before

add first → last <-> first

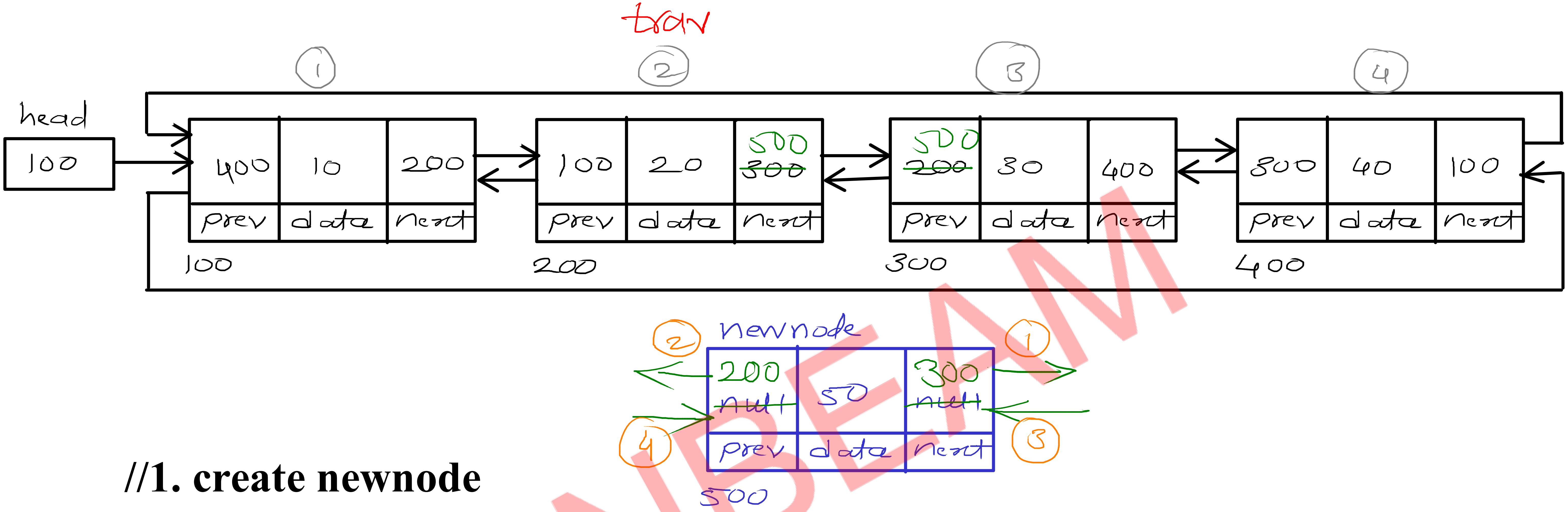
add last → last <-> first

add pos → pos-1 <-> pos

$T(n) = O(1)$

Doubly Circular Linked List - Display

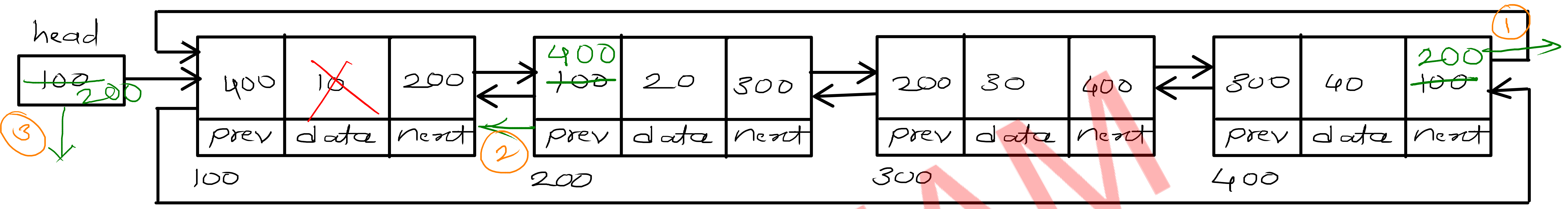
pos=3



```
//1. create newnode
//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 prev of pos node
    //e. add newnode into next of pos-1node
```

$T(n) = O(n)$

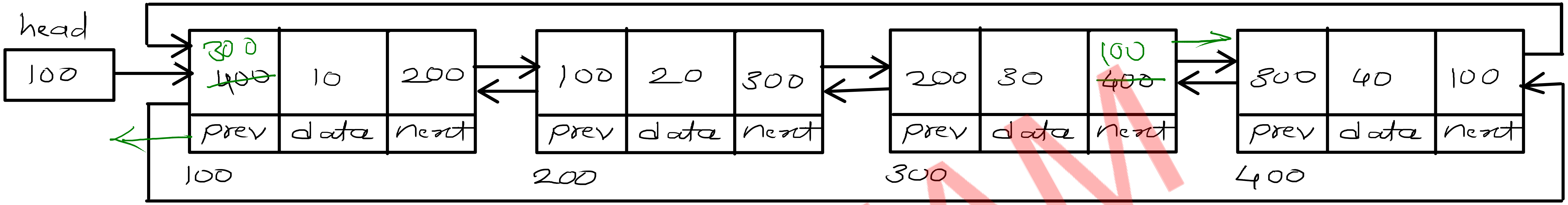
Doubly Circular Linked List - Delete First



```
//1. if list is empty
//2. if list has single node
        // make head = null;
//3. if list has multiple nodes
        //a. add second node into next of last node
        //b. add last node into prev of second node
        //c. move head on second node
```

$T(n) = O(1)$

Doubly Circular Linked List - Delete Last



//1. if list is empty

//2. if list has single node

    //make head = null

//3. if list has multiple nodes

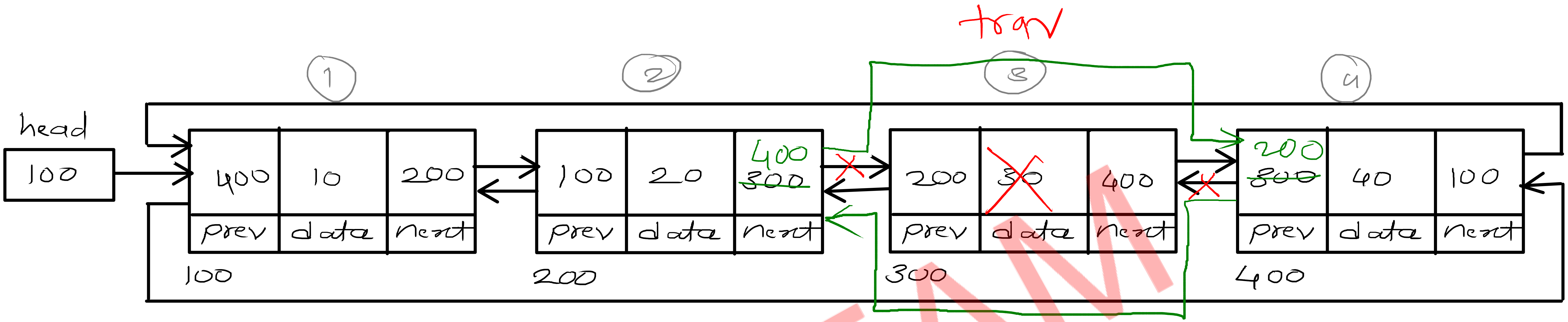
    //a. add first node into next of second last node

    //b. add second last node into prev of first node

$T(n) = O(1)$

Doubly Circular Linked List - Display

pos = 3



//1. if list is empty

//2. if list has single node

    // make head = 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

T(n) = O(n)

Linked List Applications

- linked list is a dynamic data structure
- due to this dynamic nature, liked list is used to implement other data structures like
    - stack
    - queue
    - hash table(seperate chaining)
    - graph (adjacency list)

- OS is also using linked list to maintain processes (job queue, ready queue, waiting queue)

Stack

(LIFO)

top

1. **Add first**
   Delete first

2. **Add last**
   Delete last

Queue

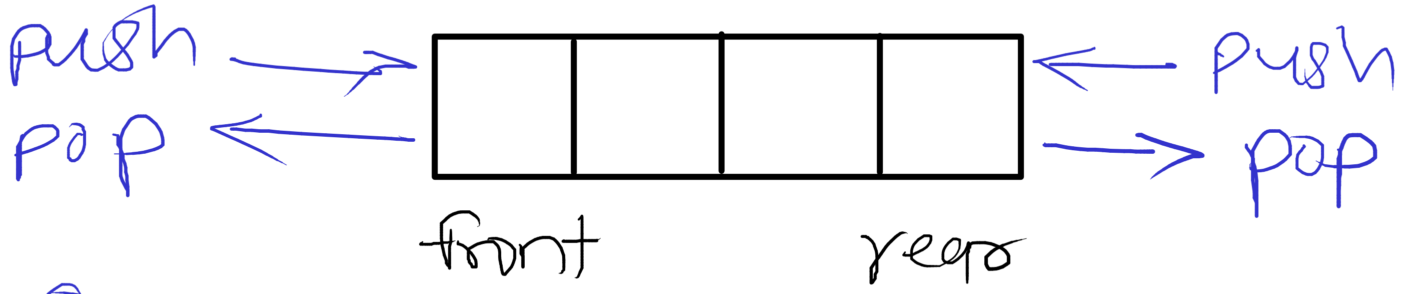
(FIFO)

rear and front

1. **Add first**
   Delete last

2. **Add last**
   Delete first

Deque - (Double Ended Queue)



① push front
Add first

③ push rear
Add last

② pop front
Delete first

④ pop rear
Delete last

Types:
1) Input restricted Deque
2) Output restricted Deque

Array Vs Linked List

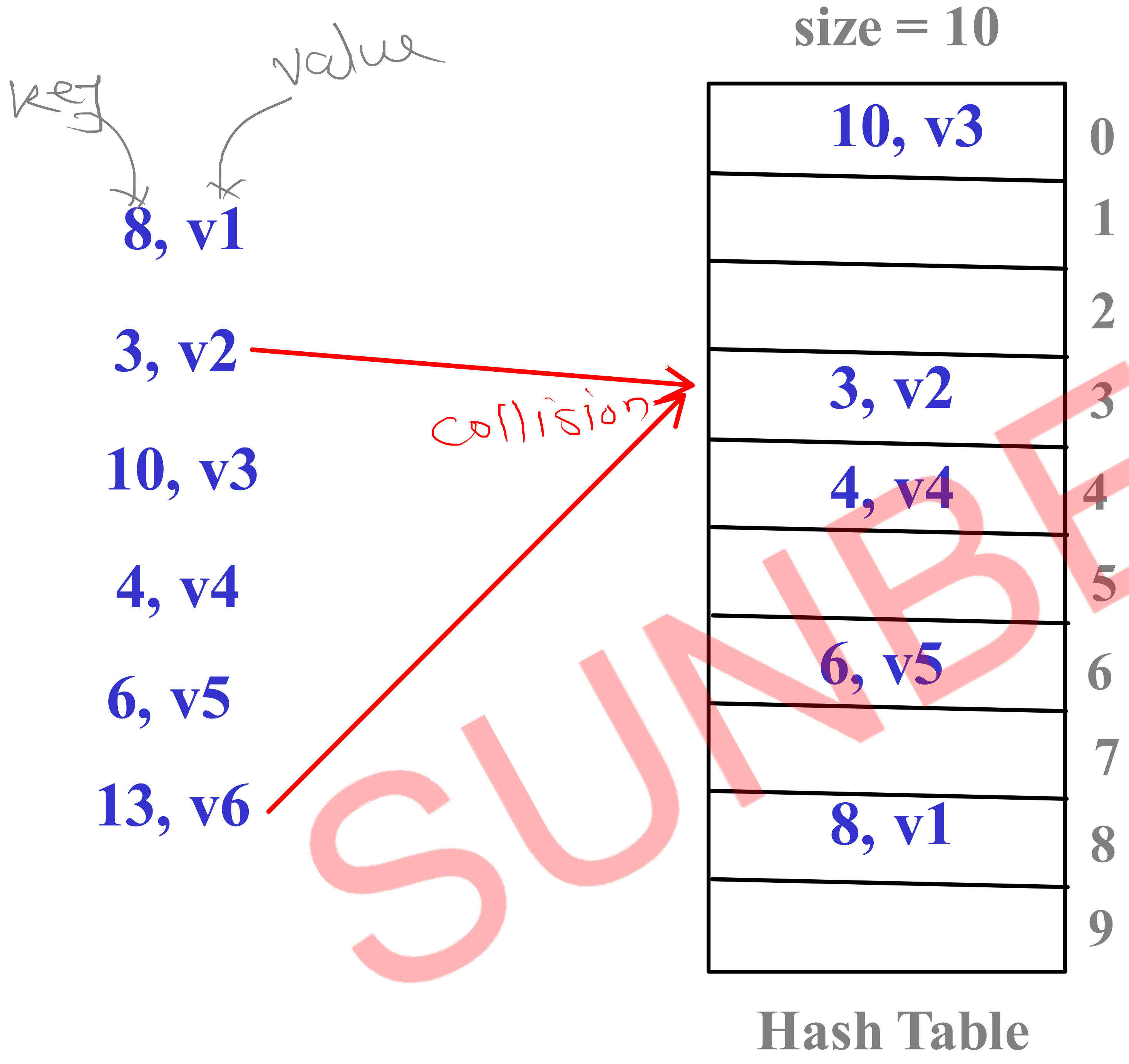
Array	Linked List
1. Array space in memory is contiguous	1. Linked list space in memory is not contiguous
2. Array can not grow or shrink at runtime	2. Linked list can grow or shrink at runtime
3. Random access of elements is allowed	3. Random access of elements is not allowed(sequential)
4. Insert or Delete, needs shifting of array elements	4. Insert or Delete, do not need shifting of nodes
5. Array needs less space	5. Linked lists need more space

Hashing

- hashing is a technique in which data can be inserted, removed and searched in constant avearge time (O(1))
- implementation of this technique is known hash table
- hash table is nothing but fixed size array in which elements are stored in key-value pair
    - Array - hash table
    - index - slot
- keys are always unique but values can be duplicates
- every key is mapped with one slot of the hash table.
- this mapping is done by a mathematical function known as "hash function"

Hashing

h(k) = k % size



8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

size = 10

0	10, v3
1	
2	
3	3, v2
4	4, v4
5	
6	6, v5
7	
8	8, v1
9	

Hash Table

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

Collision:
– when two different key
yield same slots

Add: – O(1)
1) slot = k % size
2) arr[slot] = entry

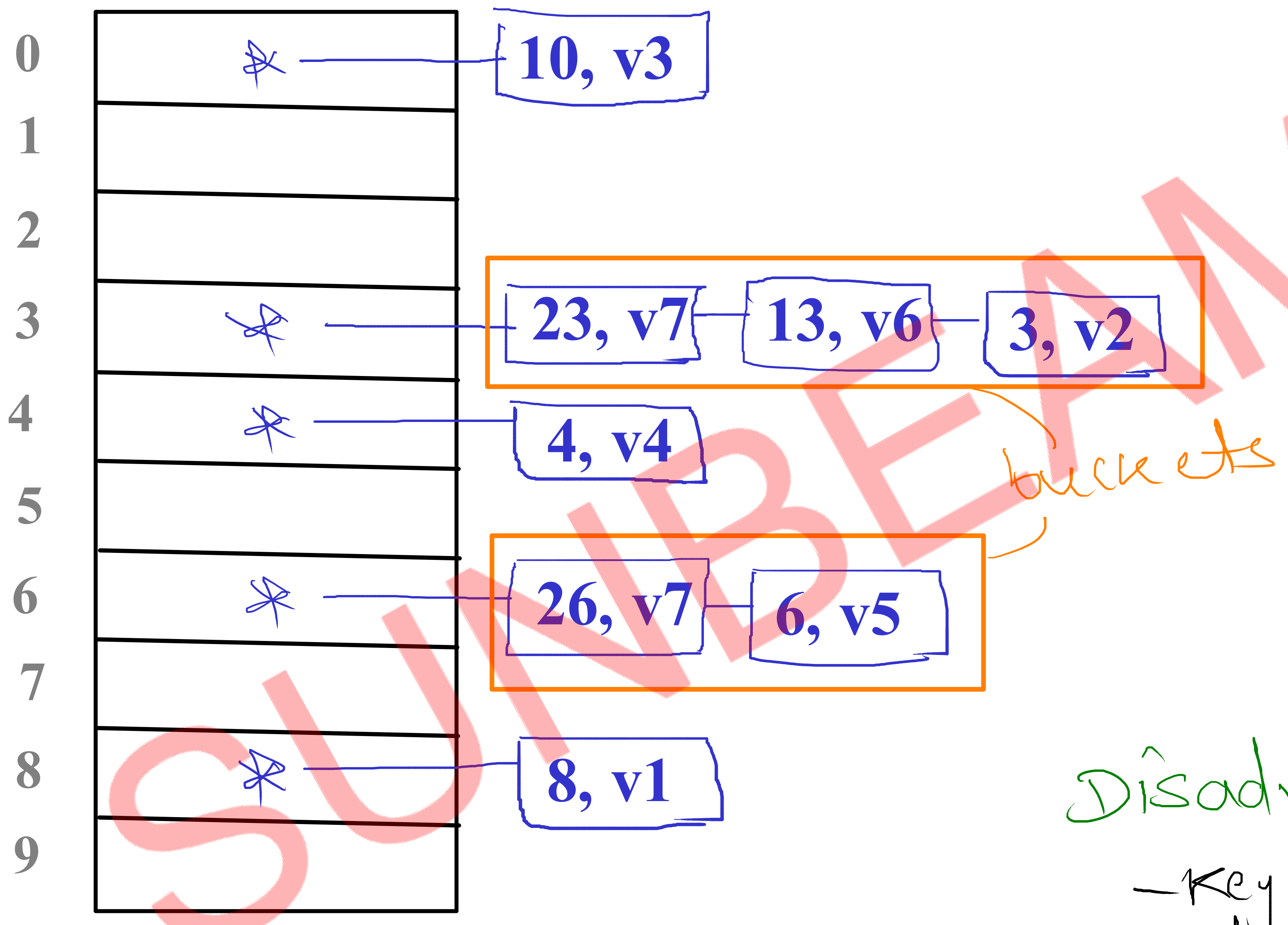
Search: – O(1)
1) slot = k % size
2) return arr[slot]

Delete: – O(1)
1) slot = k % size
2) arr[slot] = null

Closed Addressing/ Seperate Chaining / Chaining

8, v1
3, v2
10, v3
4, v4
6, v5
13, v6
23, v7
26, v7

size = 10



Hash Table

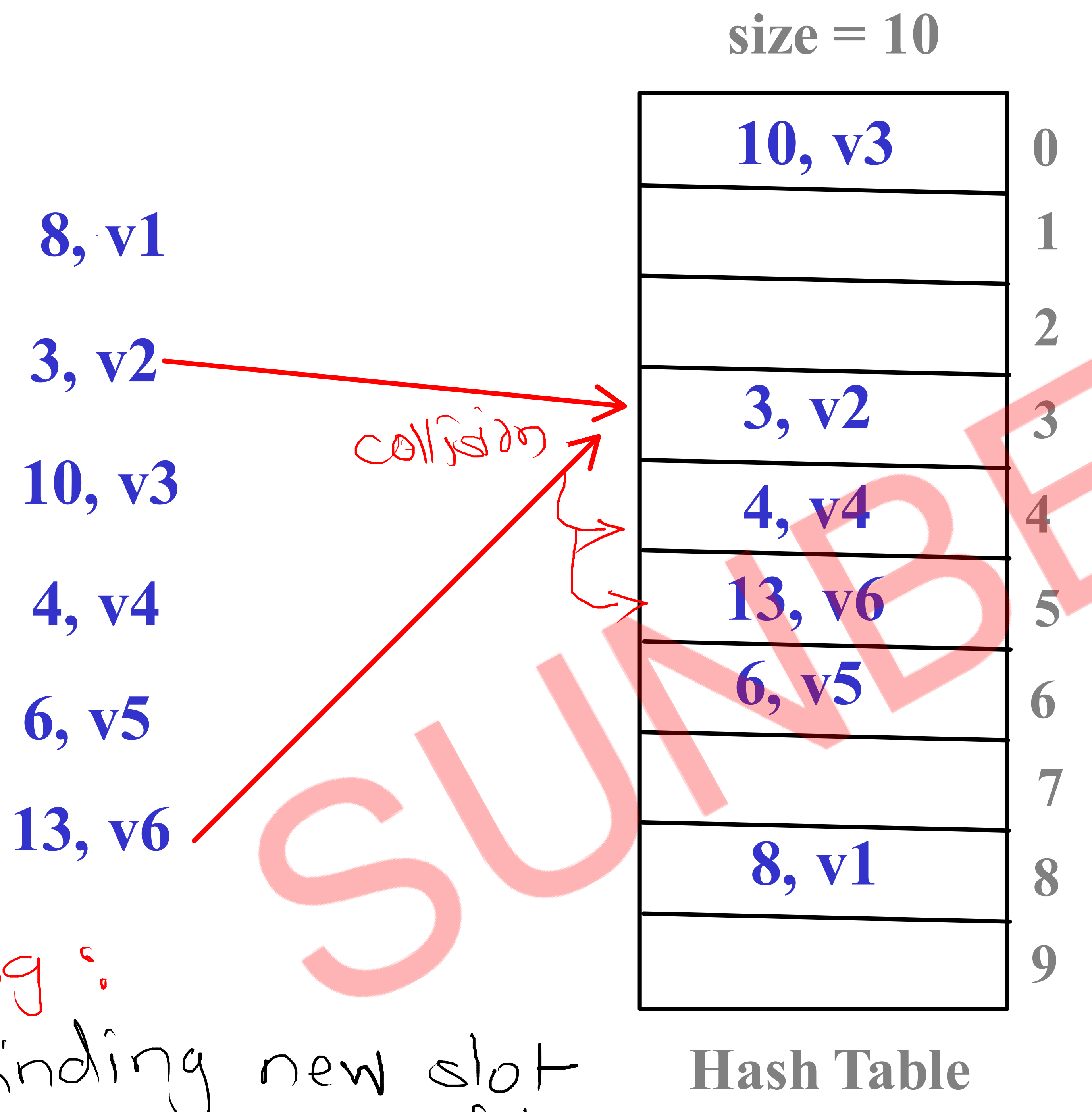
h(k) = k % size

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

Disadvantages

- Key-value pairs are stored outside the table
- will need more space
- worst case
  T(n) = O(n)

Open Addressing - Linear Probing



8, v1
3, v2
10, v3
4, v4
6, v5
13, v6

size = 10

0	10, v3
1	
2	
3	3, v2
4	4, v4
5	13, v6
6	6, v5
7	
8	8, v1
9	

Hash Table

$h(k) = key \ \% \ size$

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

$f(i) = i$

where i = 1, 2, 3, ...

↳ probe number

$h(8) = 8 \% 10 = 8$

$h(3) = 3 \% 10 = 3$

$h(10) = 10 \% 10 = 0$

$h(4) = 4 \% 10 = 4$

$h(6) = 6 \% 10 = 6$

$h(13) = 13 \% 10 = 3$ ⓒ

$h(13,1) = [3 + 1] \% 10$

$= 4$ ⓒ (1^{st} probe)

$h(13,2) = [3 + 2] \% 10$

$= 5$ (2^{nd} probe)

Probing :

- finding new slot whenever collision will occur

Primary clustering :

- it takes long runs of filled slots "near" key position to find empty slot

Open Addressing - Quadratic Probing

8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

size = 10

10, v3	0
	1
	2
3, v2	3
4, v4	4
	5
6, v5	6
13, v6	7
8, v1	8
	9

Hash Table

h(k) = key % size

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

f(i) = i^2

where i = 1, 2, 3, ...

↳ probe number

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 ⓒ

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

= 4 ⓒ (1st probe)

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

= 7 (2nd probe)

- there is no garuntee of getting free slot .

- primary clustering is solved

- secondary clustering
  - it takes long runs of filled slots "away" key position to find empty slot

Open Addressing - Quadratic Probing

size = 10

10, v3	0
	1
23, v7	2
3, v2	3
4, v4	4
	5
6, v5	6
13, v6	7
8, v1	8
33, v8	9

Hash Table

23, v7

33, v8

h(k) = key % size

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

f(i) = i^2

where i = 1, 2, 3, ...

$h(23) = 23 \% 10 = 3$ ⓒ

$h(23,1) = [3+1] \% 10 = 4$ ⓒ (1st)

$h(23,2) = [3+4] \% 10 = 7$ ⓒ (2nd)

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

$h(33) = 33 \% 10 = 3$ ⓒ

$h(33,1) = [3+1] \% 10 = 4$ ⓒ (1st)

$h(33,2) = [3+4] \% 10 = 7$ ⓒ (2nd)

$h(33,3) = [3+9] \% 10 = 2$ ⓒ (3rd)

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

Hashing - Double Hashing

8, v1

3, v2

10, v3

25, v6

size = 11

	0
	1
	2
3, v2	3
	4
	5
25, v6	6
	7
8, v1	8
	9
10, v3	10

Hash Table

$h1(k) = key \% size$

$h2(k) = 7 - (key \% 7)$

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

$h1(8) = 8 \% 11 = 8$

$h1(3) = 3 \% 11 = 3$

$h1(10) = 10 \% 11 = 10$

$h_1(25) = 25 \% 11 = 3$ ⓒ

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

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

$= 6$

Rehashing

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

(λ)

n - Number of elements (key value pairs) in hash table
N - Number of slots in hash table

if $n < N$	Load factor < 1	- free slots are available
if $n = N$	Load factor $= 1$	- no free slots
if $n > N$	Load factor > 1	- can not insert at all

- Rehashing is make the hash table size twice of existing size if hash table is 70 or 75 % full

- In rehashing existing key value pairs are again mapped according to new hash table size