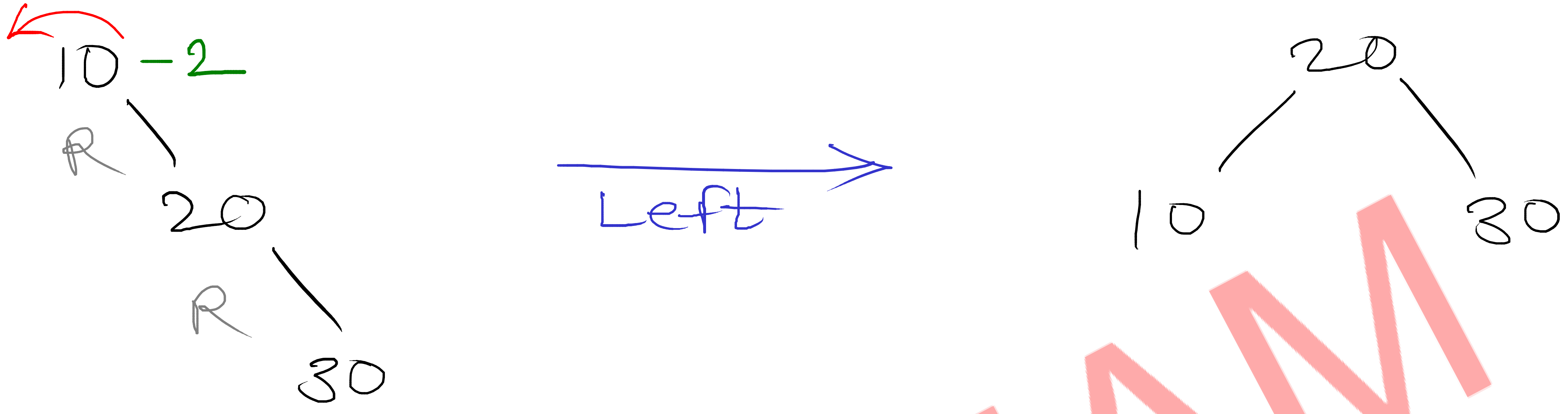
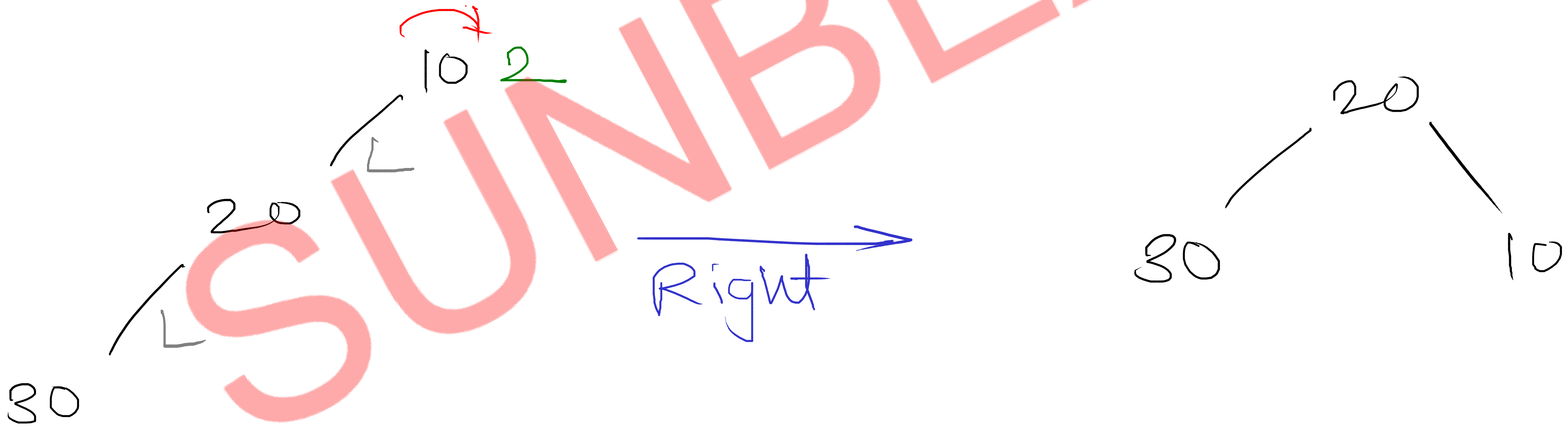
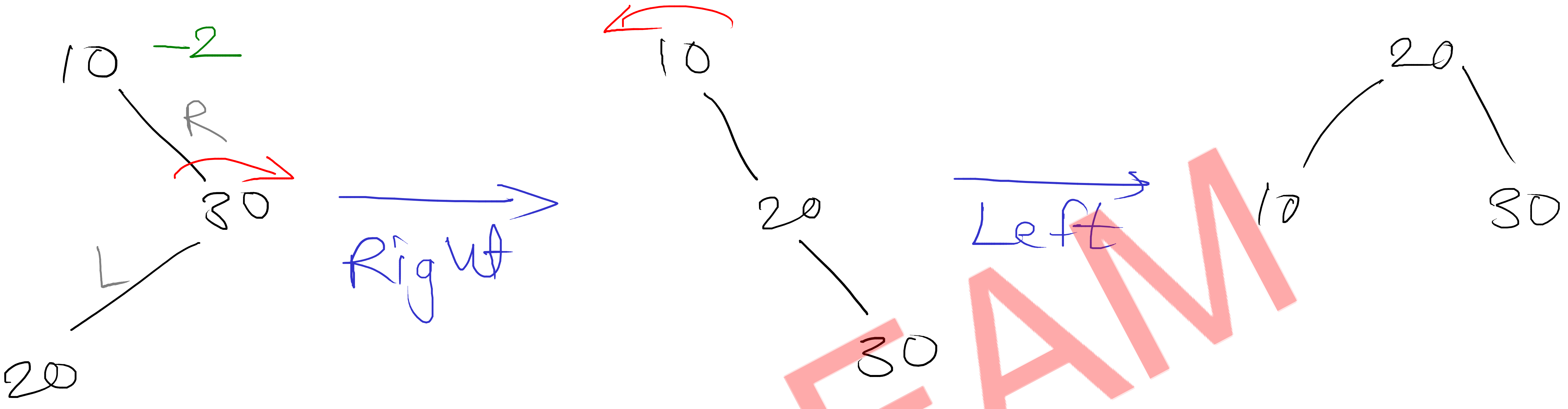
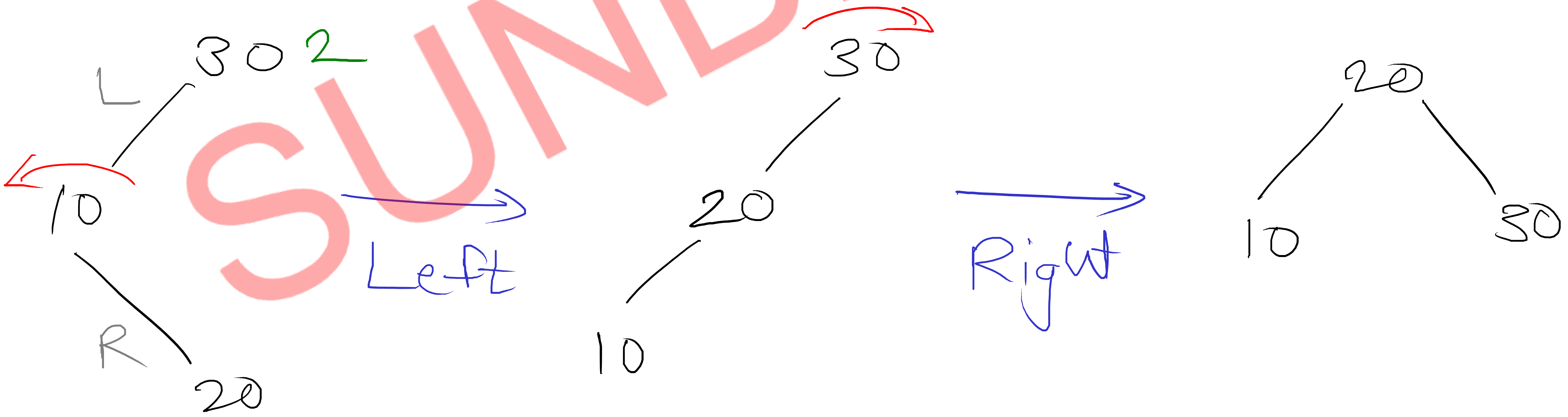
Single Rotations



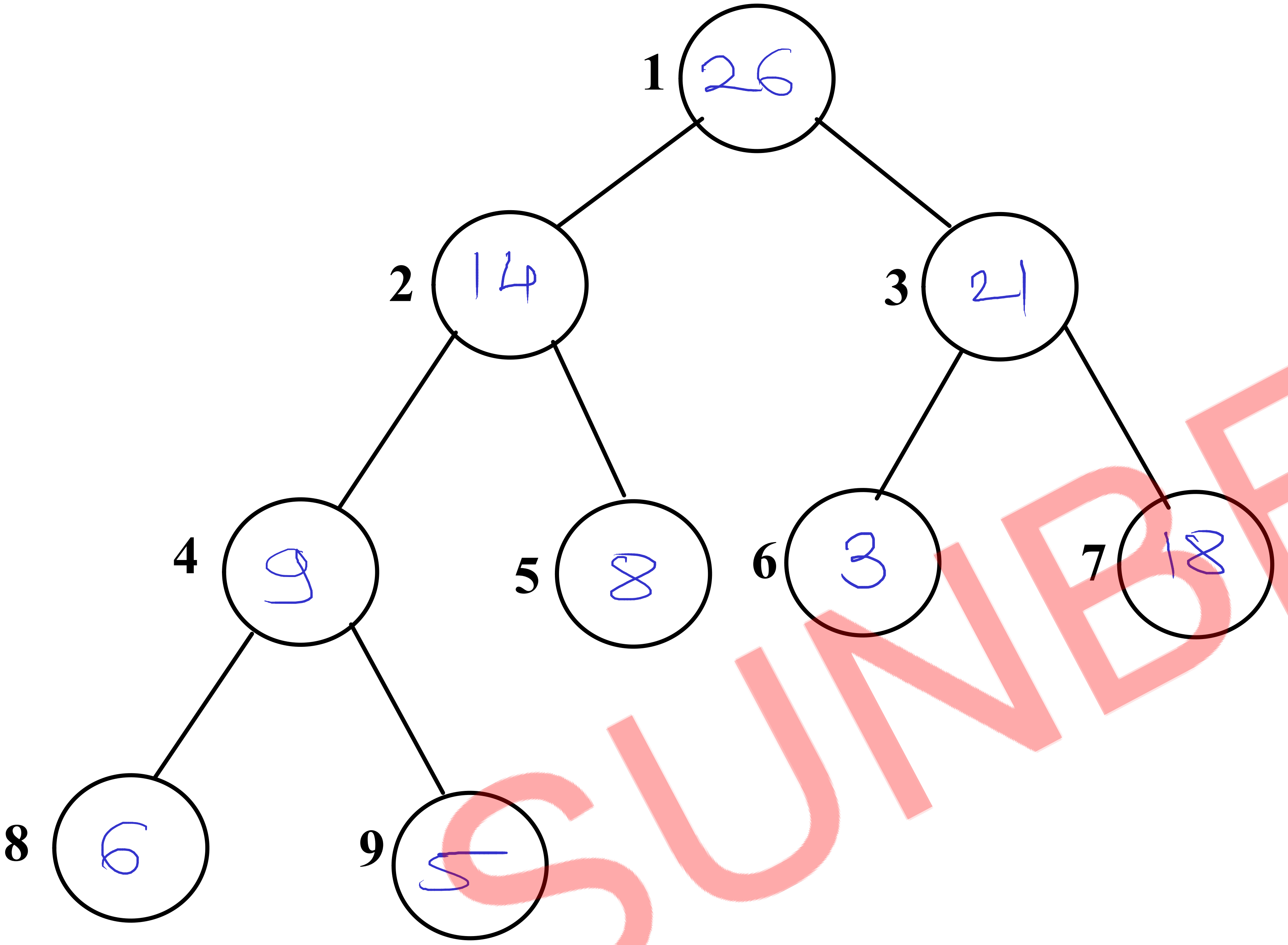


Double Rotations





Heap - Create Heap



6 14 3 26 8 18 21 9 5

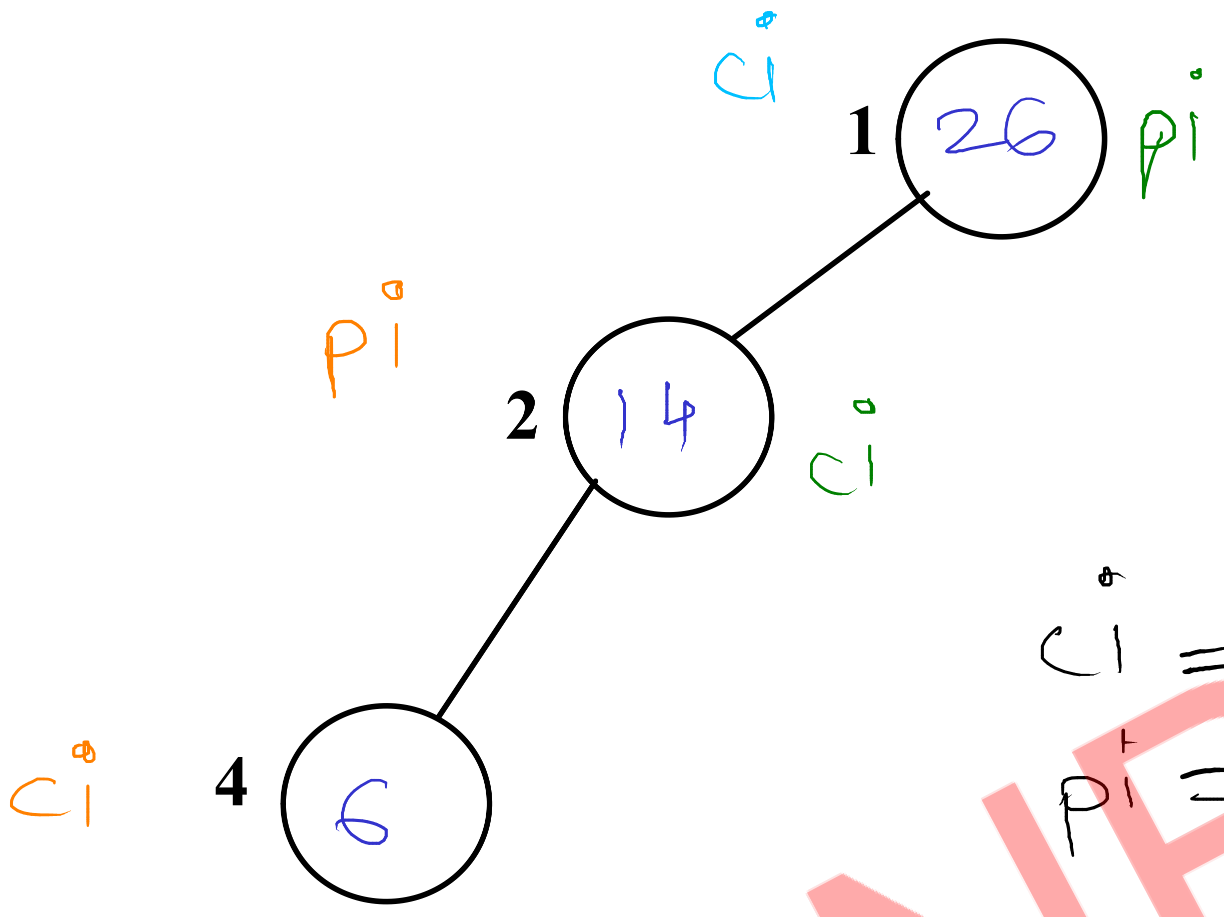
$T(n) = O(\log n)$

26	14	21	9	8	3	18	6	5
1	2	3	4	5	6	7	8	9

1) add new element at first empty location of the heap

2) adjust the position of newly added element by comparing with all its ancestors

$pi = 0$

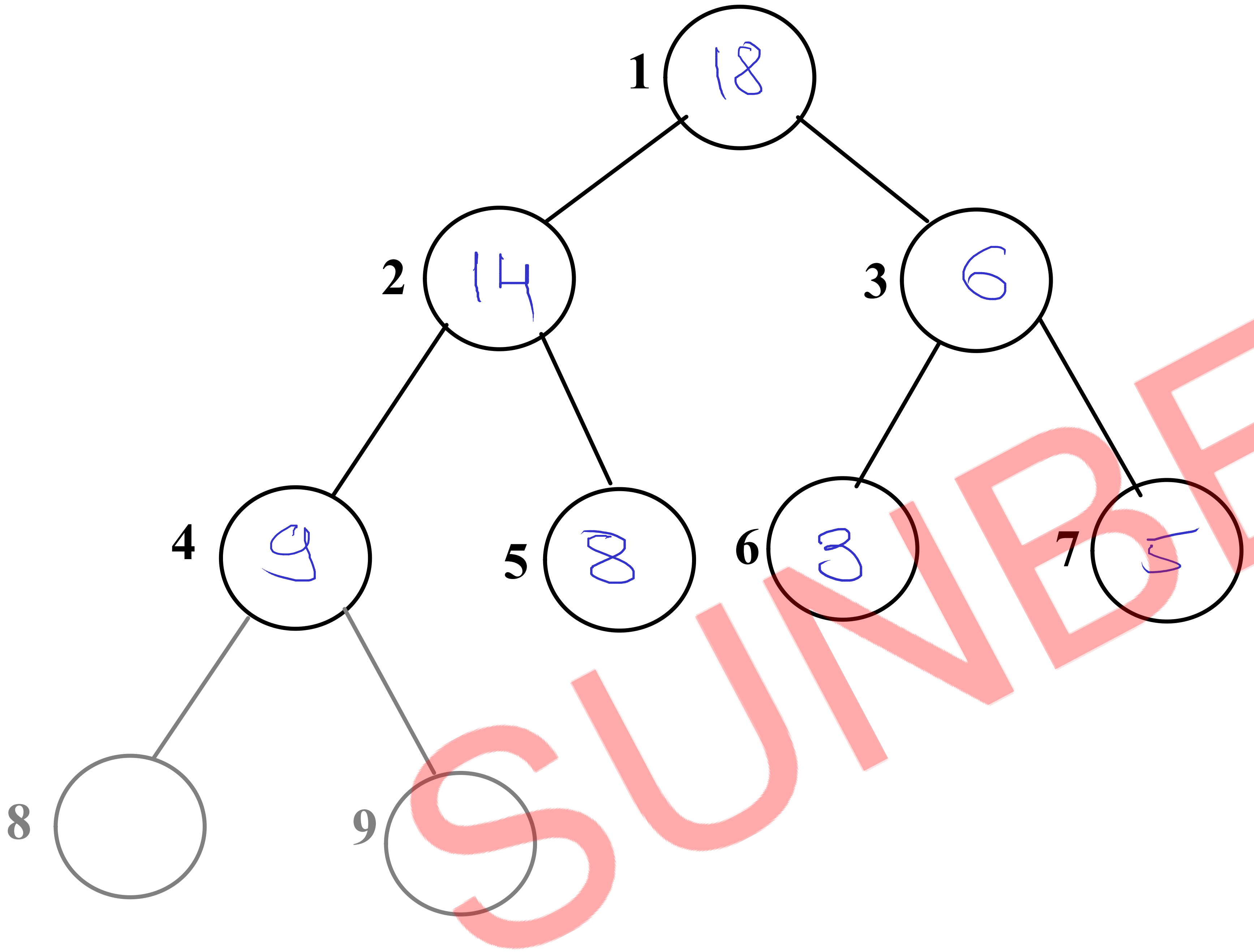


ci	pi
4	2
2	1
1	0

$ci = pi$

$pi = ci/2$

Heap - Delete Heap



6	14	18	9	8	3	5		
1	2	3	4	5	6	7	8	9

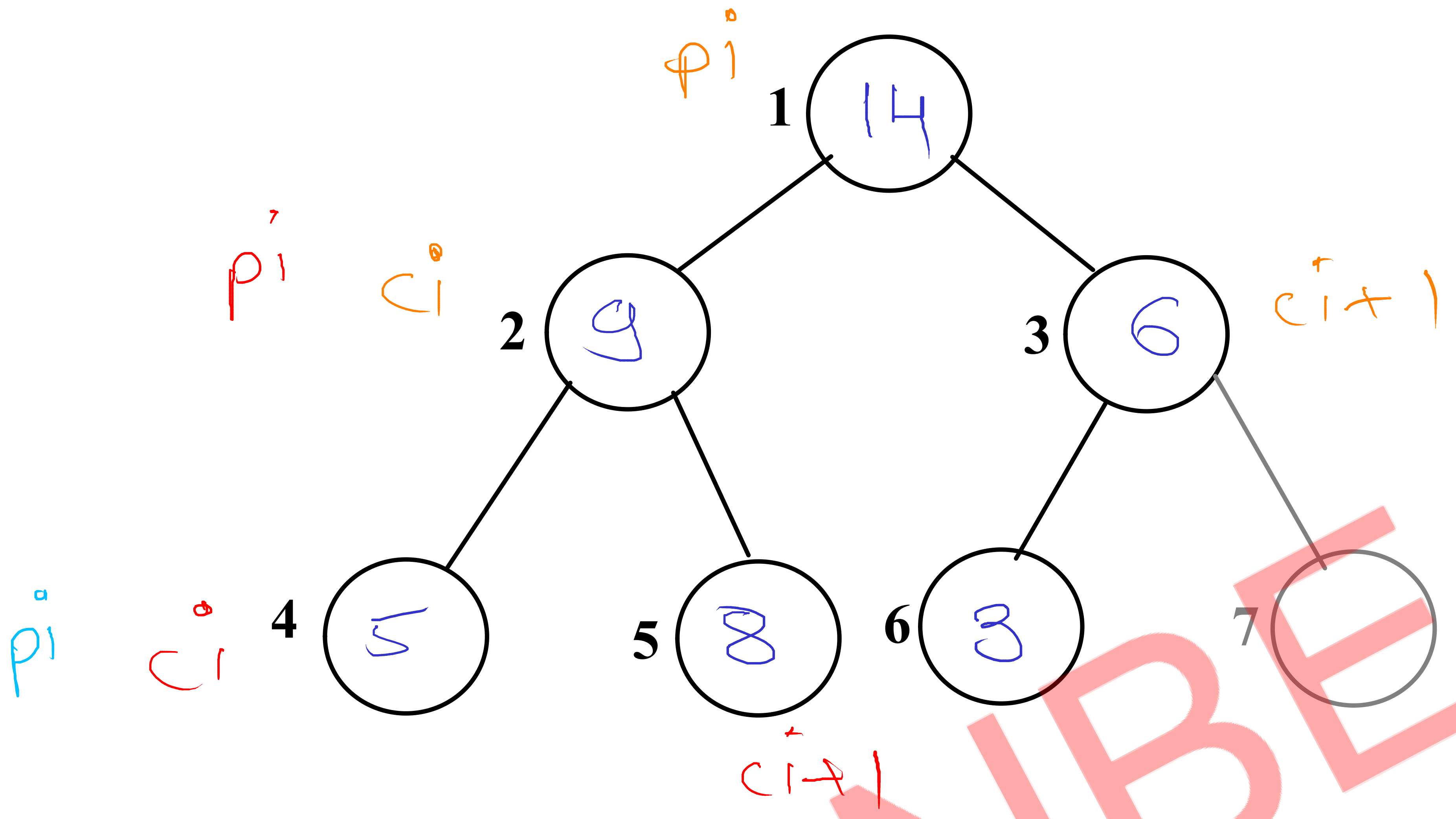
Max = 26

Max = 21

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

1) Delete root element
2) place last element at root's location
3) Adjust the position o it upto leaf nodes.

Max = 18



pi	ci
1	2
2	4
4	8

ci = 8

pi = ci

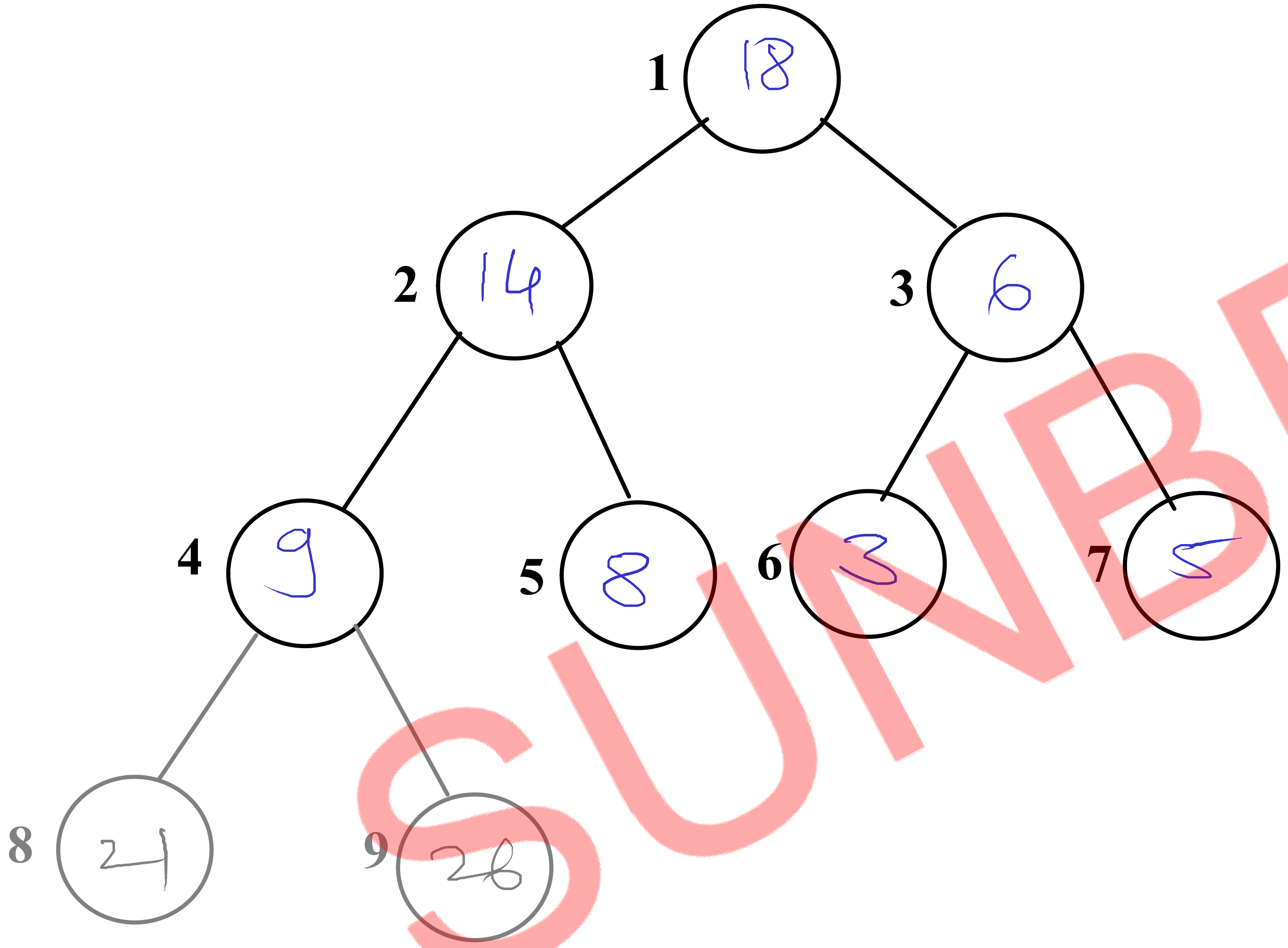
ci = pi * 2

Heap sort

1. create heap of given array → $n \log n$
2. delete all the elements from heap → $n \log n$

$$\frac{\begin{array}{l} n \log n \\ n \log n \end{array}}{2n \log n}$$

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



18	14	6	9	8	3	5	21	26
1	2	3	4	5	6	7	8	9

Array - linear search — O(n)

Binary search — O(log n)

Linked — linear search - O(n)

Binary tree — O(n)

Binary search tree — O(log n)

Hash Table — O(1)

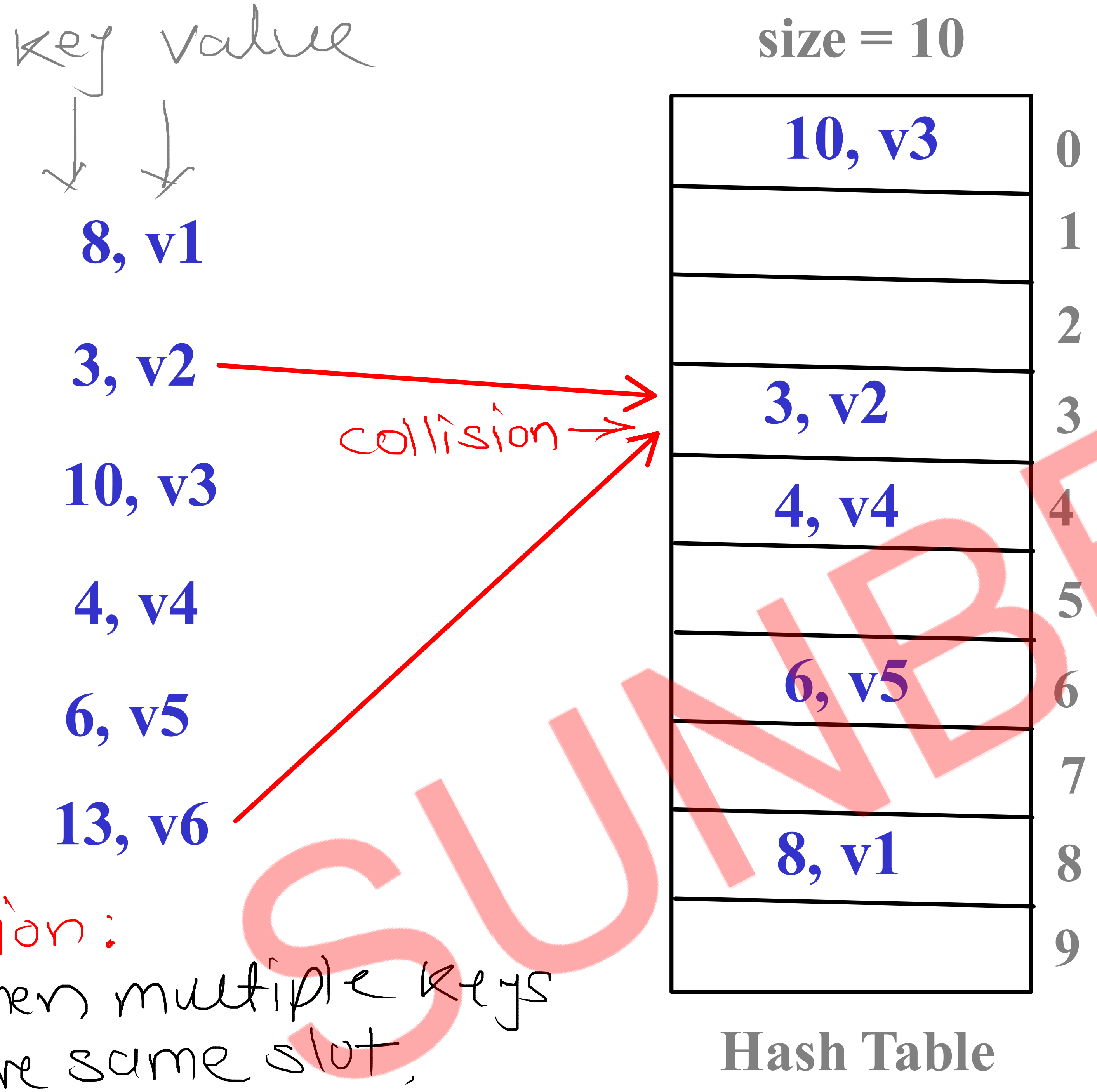
Hashing

- 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

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

Add : O(1)

1) find slot
2) arr[slot] = {key, value}

Search : O(1)

1) find slot
2) return arr[slot]

Delete : O(1)

1) find slot
2) arr[slot] = null;

Collision :

when multiple keys yeild/give same slot.

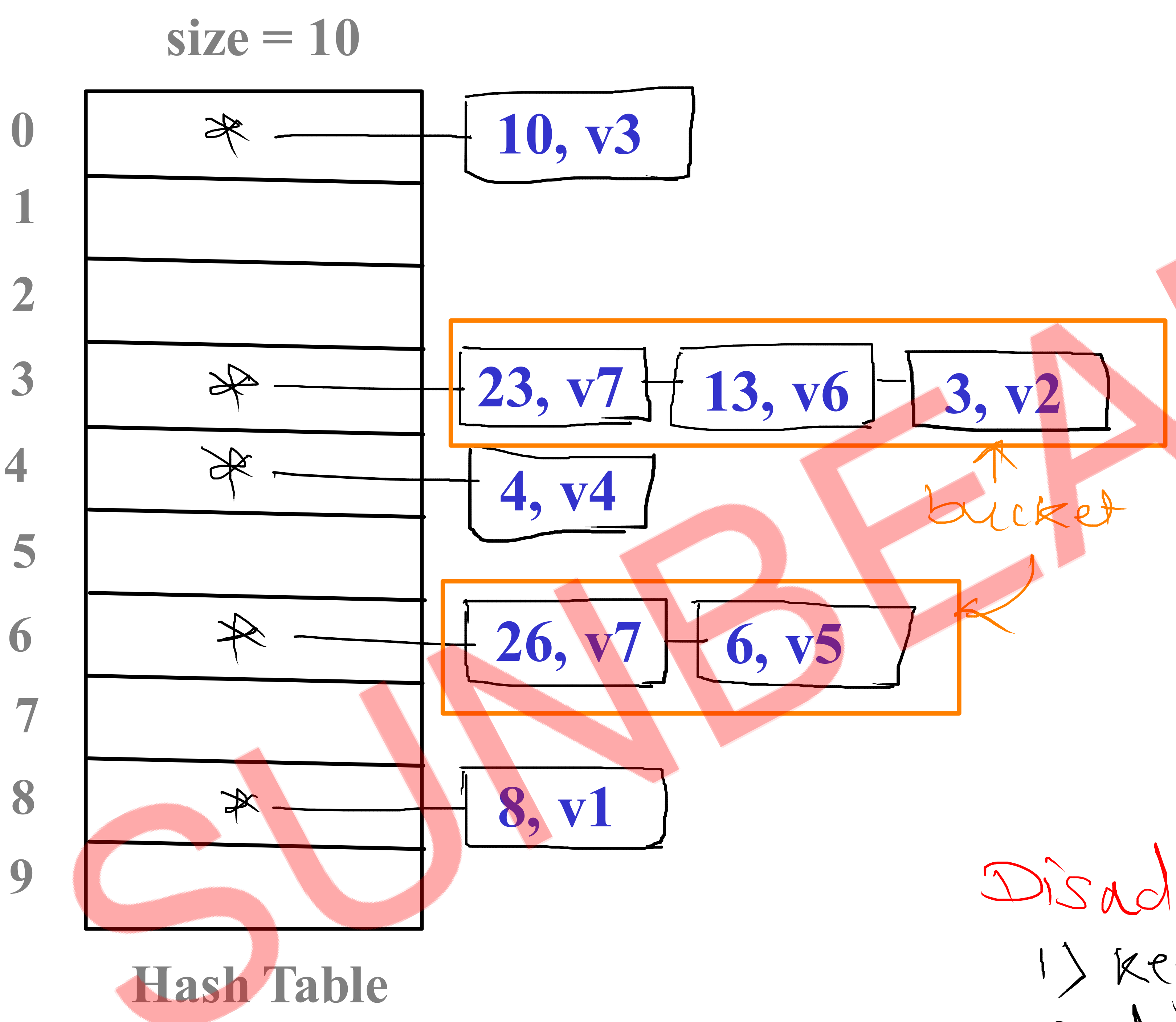
Collision handling techniques :

1) closed Addressing
2) Open Addressing

Closed Addressing/ Seperate Chaining / Chaining (open probing)

h(k) = k % size

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



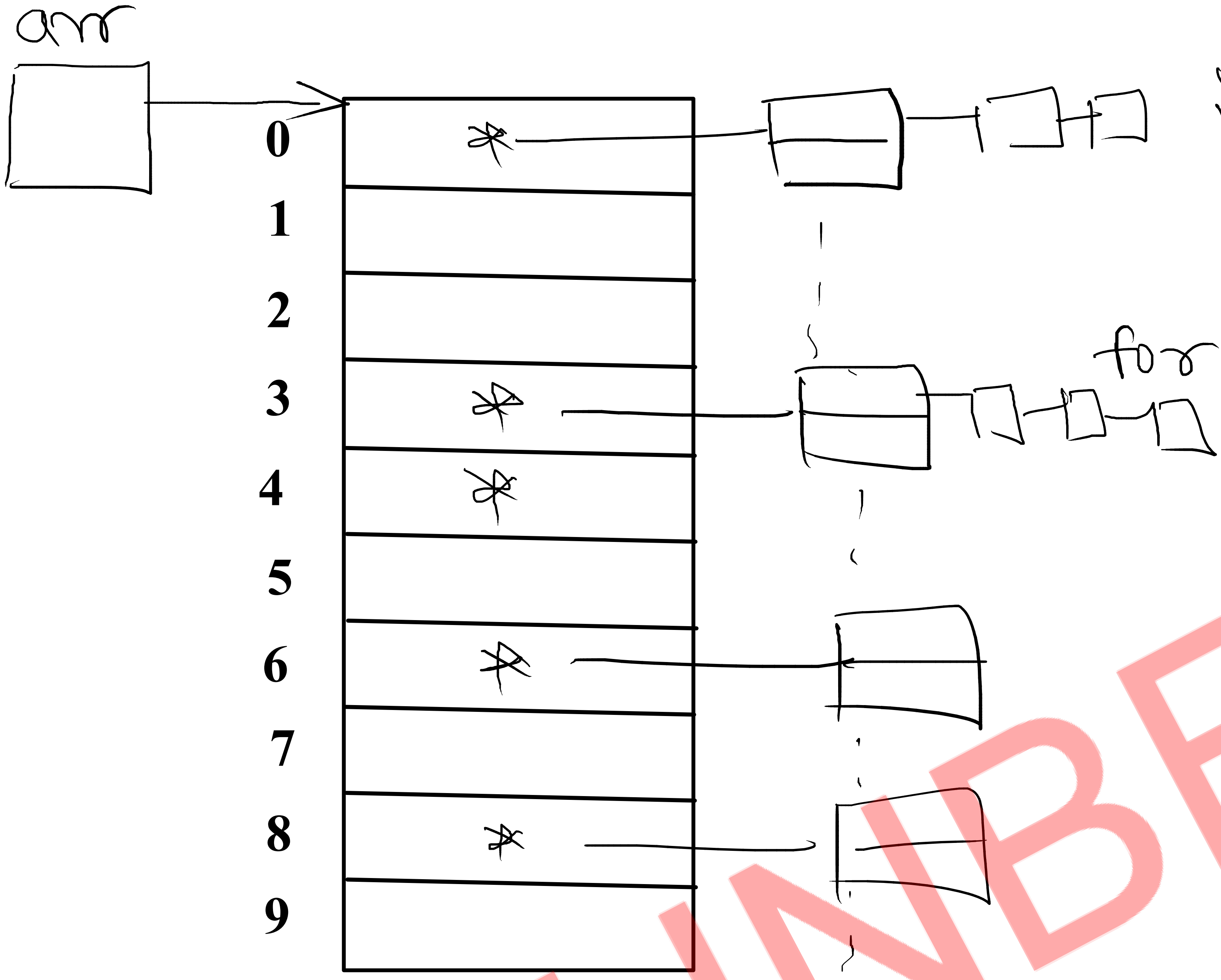
h(8) = 8%10 = 8
h(3) = 3 % 10 = 3
h(10) = 10%10 = 0
h(4) = 4 % 10 = 4
h(6) = 6 %10 = 6
h(13) = 13 %10 = 3
h(23) = 23 % 10 = 3
h(26) = 26 %10 = 6

Disadvantage :

1) key, value pairs are stored outside the table.
2) memory requirement (space) is more.
3) worst case time complexity is O(n) ← when all keys will yeild slot.

Advantage :

- multiple key, value pairs can be stored into hash table



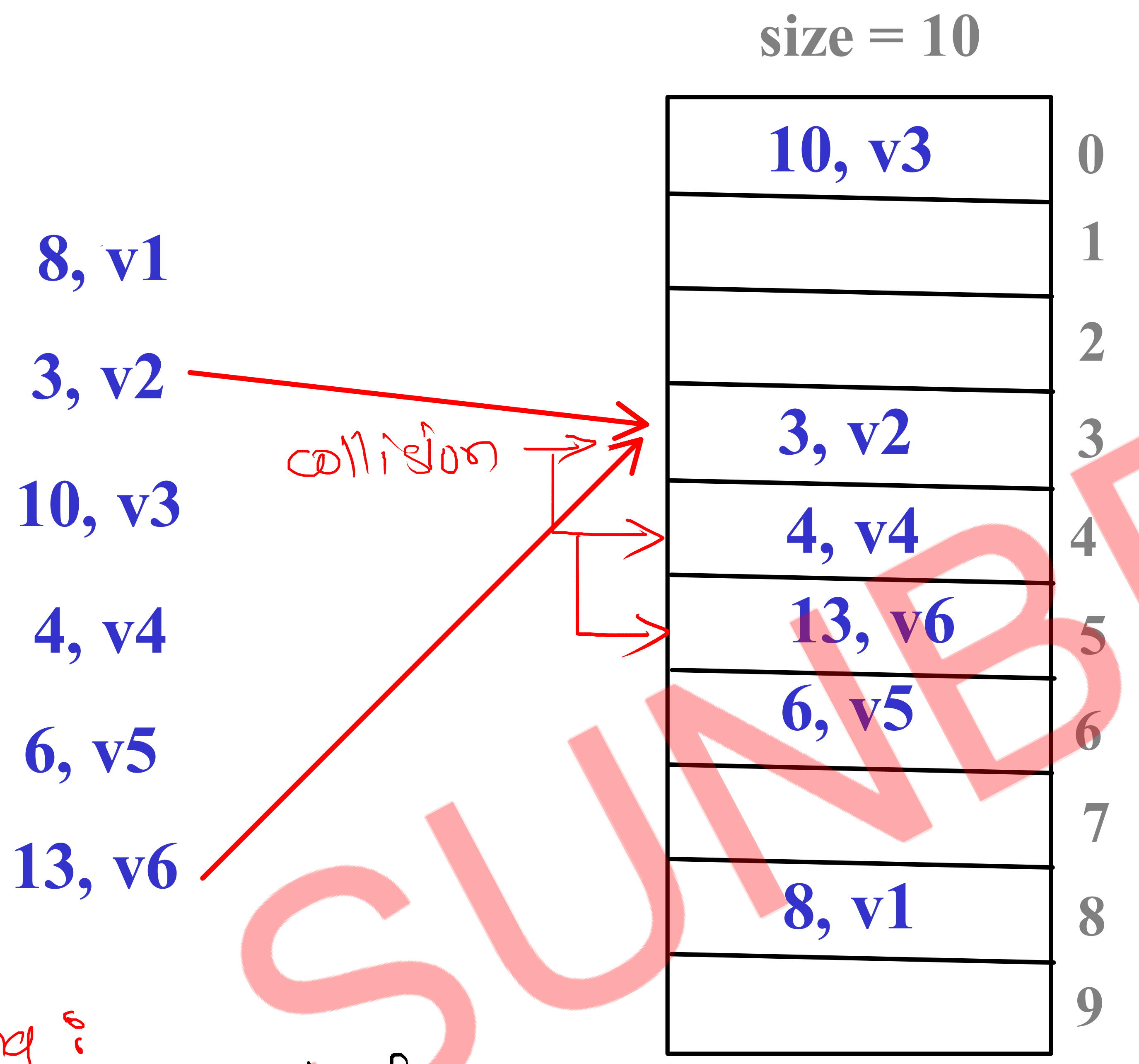
```
int arr[];

List arr[];
for(int i=0 ; i< arr.length ;i++)
arr[i] =new LinkedList();

Entry{
  int key;
  String value;
}
```

Open Addressing - Linear Probing

(closed probing)



Hash Table

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

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

$f(i) = i$

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

↑ probe number

$h(13) = 13 \ \% \ 10 = 3$ ⓒ

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

$= 4$ (1st probe) ⓒ

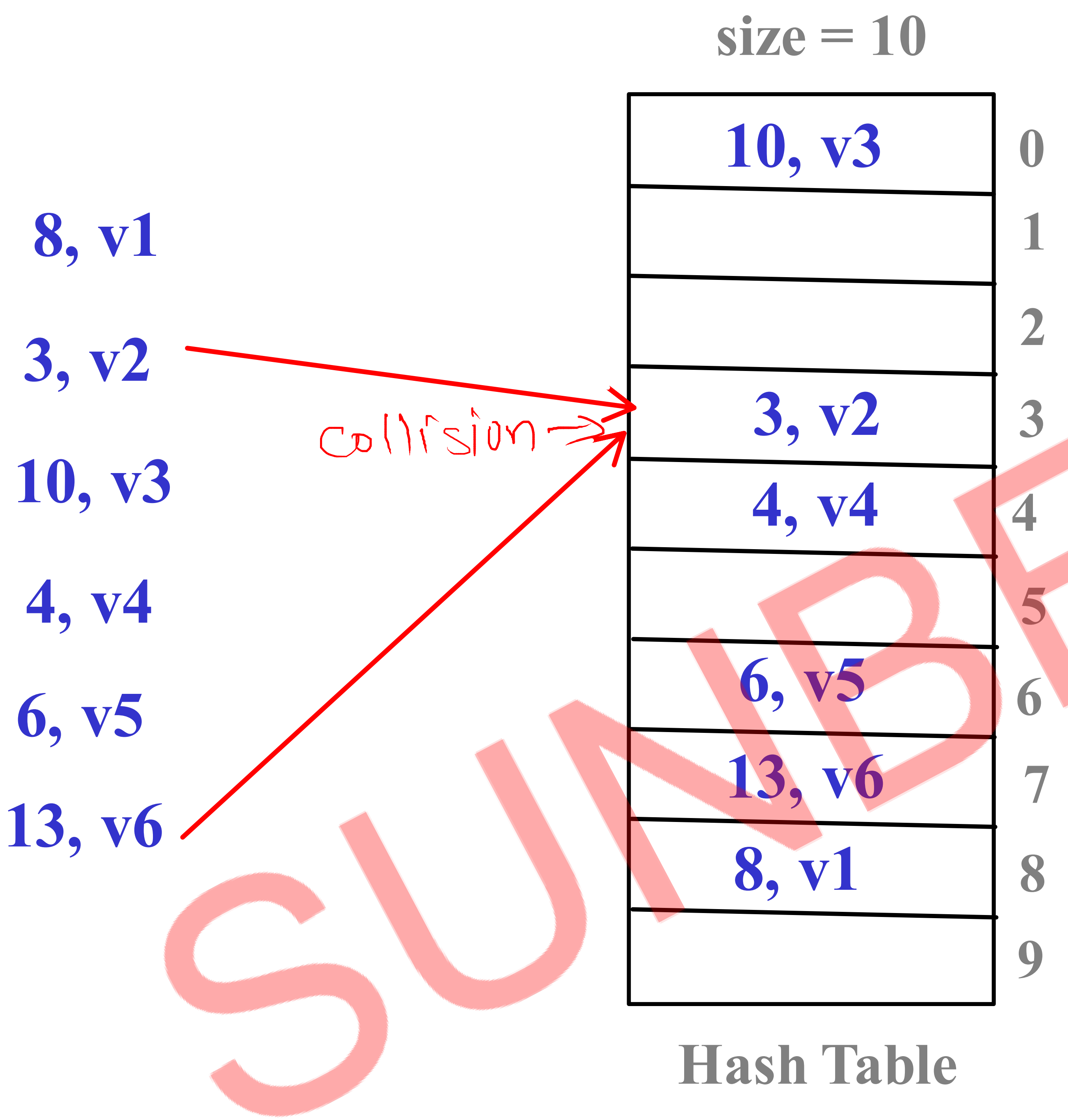
$h(13,2) = [3+2] \ \% \ 10$

$= 5$ (2nd probe)

Probing :
finding next free slot of tbl whenever collision will occur.

Primary clustering :
- needs long run of filled slots to find empty slot "near" key position.

Open Addressing - Quadratic Probing



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

h(k) = key % size

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

f(i) = i^2

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

h(13) = 13 % 10 = 3 ⓒ

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

= 4 (1st probe) ⓒ

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

= 7 (2nd probe)

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 c

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

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

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

h(33) = 33 % 10 = 3 c

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

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

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

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

Secondary clustering :

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

Hashing - Double Hashing

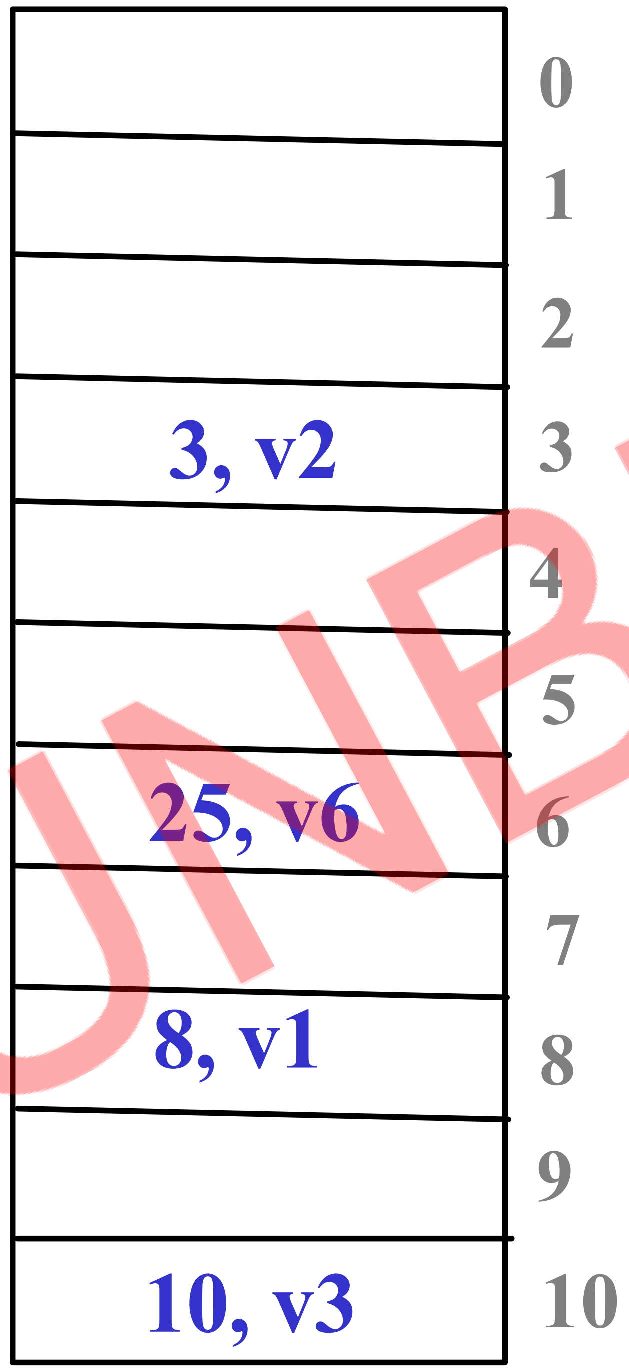
8, v1

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

$h1(k) = key \% size$

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

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

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

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

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

$h_1(25) = 25\%11 = 3$ ⓒ

$h_2(25) = 7-4 = 3$

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

$= 6$ 1st

$h_1(36) = 3$ ⓒ

$h_2(36) = 6$

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

Rehashing

$$\text{Load Factor} = \frac{n}{N} = \frac{6}{10} = 0.6 \rightarrow 60\% \text{ filled}$$

(λ)

n - Number of elements (key value pairs) in hash table - 6

N - Number of slots in hash table - 10

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