Stable sort vs Unstable sort

* Array: [{A, 65}, {B, 90}, {C, 55}, {D, 85}, {E, 55}, {F, 65}]

* Stable sort:

- Equal elements maintains their relative order as in original array -- Guaranteed.

[{C, 55}, {E, 55}, {A, 65}, {F, 65}, {D, 85}, {B, 90}]

e.g. Bubble, Insertion, ...

* UnStable sort:

- Equal elements may not maintain their relative order as in original array.

[{C, 55}, {E, 55}, {F, 65}, {A, 65}, {D, 85}, {B, 90}]

e.g. Selection.

In-place sort vs Out-place sort

*** In-place sort**

- No additional space requires for holding array element.

- Aux Space complexity is O(1)

e.g. Selection, Bubble, Insertion, ...

*** Out-place sort**

- Additional space requires for holding sorted array element.

- Aux Space complexity is O(n) -- without stack space.

e.g. Merge

Searching of data

1. **Array - Linear search**

   $T(n) = O(n)$

2. **Array - Binary search**

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

3. **Linked List - search**

   $T(n) = O(n)$

4. **Binary Tree - search**

   $T(n) = O(n)$

5. **BST - search**

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

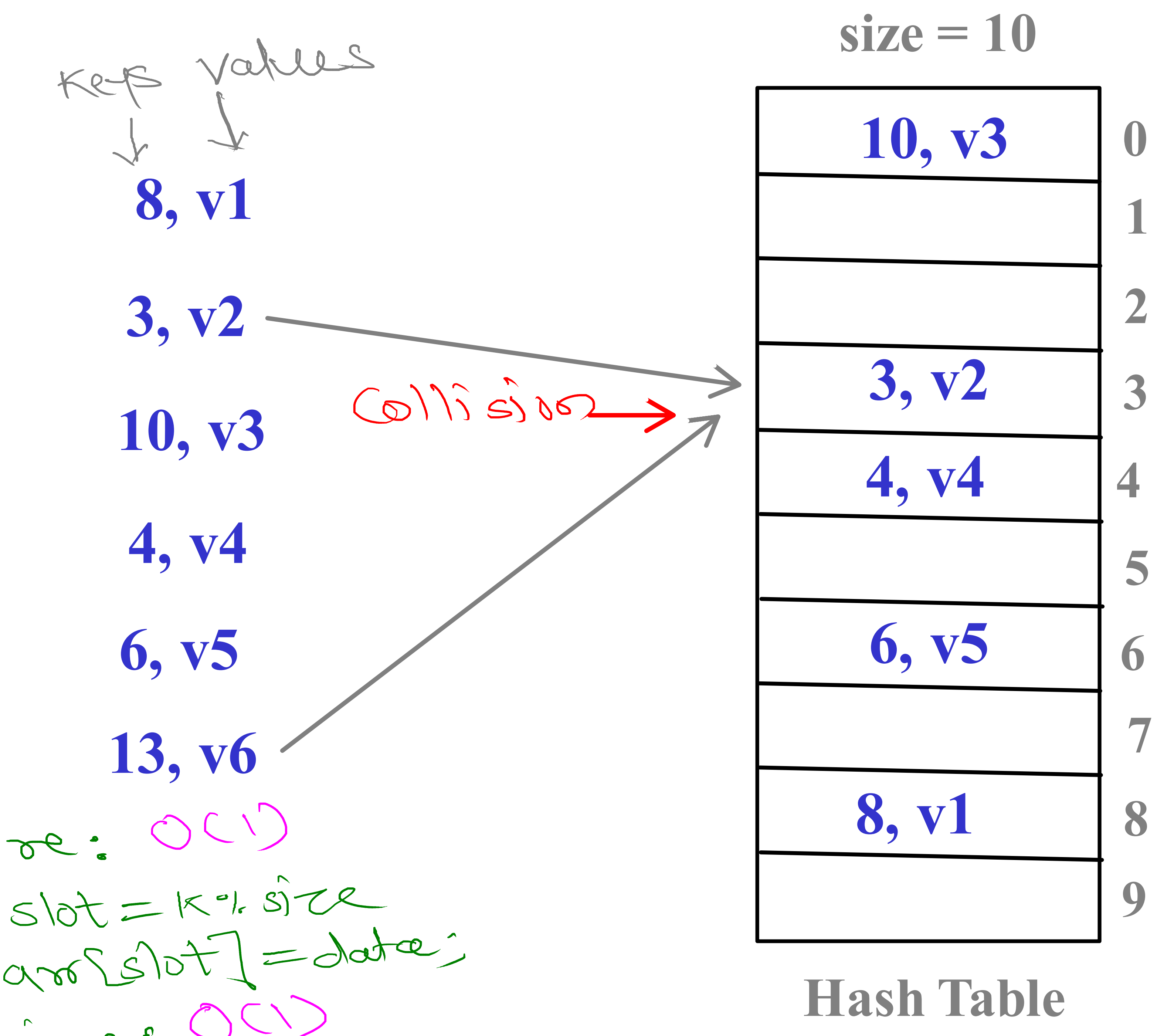
In all these searching options, time is dependent on number of elements in that data structure

- will have variable time complexity for every option

- solution for this is Hashing / Hash Table.

- data will be searched in constant amount of time. $O(1)$

Hashing



size = 10

8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

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

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

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

```
store: O(1)
    slot = k % size
    arr[slot] = data;
retrieve: O(1)
    slot = k % size
    return arr[slot]
search: O(1)
    slot = k % size
    return arr[slot].data;
```

Collision:-
when two different keys yield same slot, it is called as collision

-Whenever collision will occur, next free slot will be find out by any one of the collision handling technique.

Closed Addressing/ Seperate Chaining / Chaining

8, v1

3, v2

10, v3

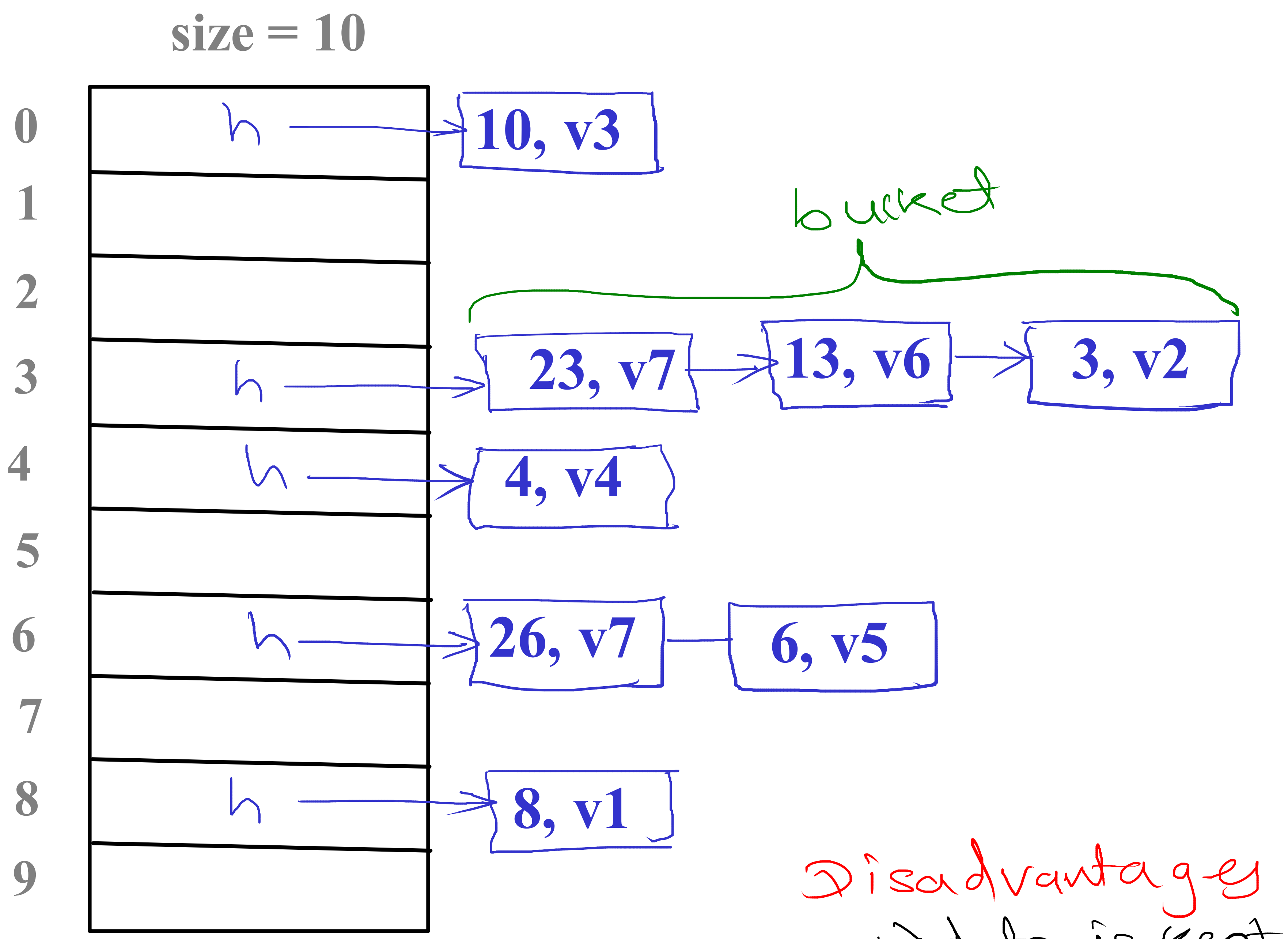
4, v4

6, v5

13, v6

23, v7

26, v7



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

h(13)=18%10=3

h(23)=23%10=3

h(26)=26%10=6

Disadvantages :-

1> data is kept out side the table

2> needs more space

3> one of the linked may grow heavily. in this case operations will not be performed in constant time.

Open Addressing - Linear Probing

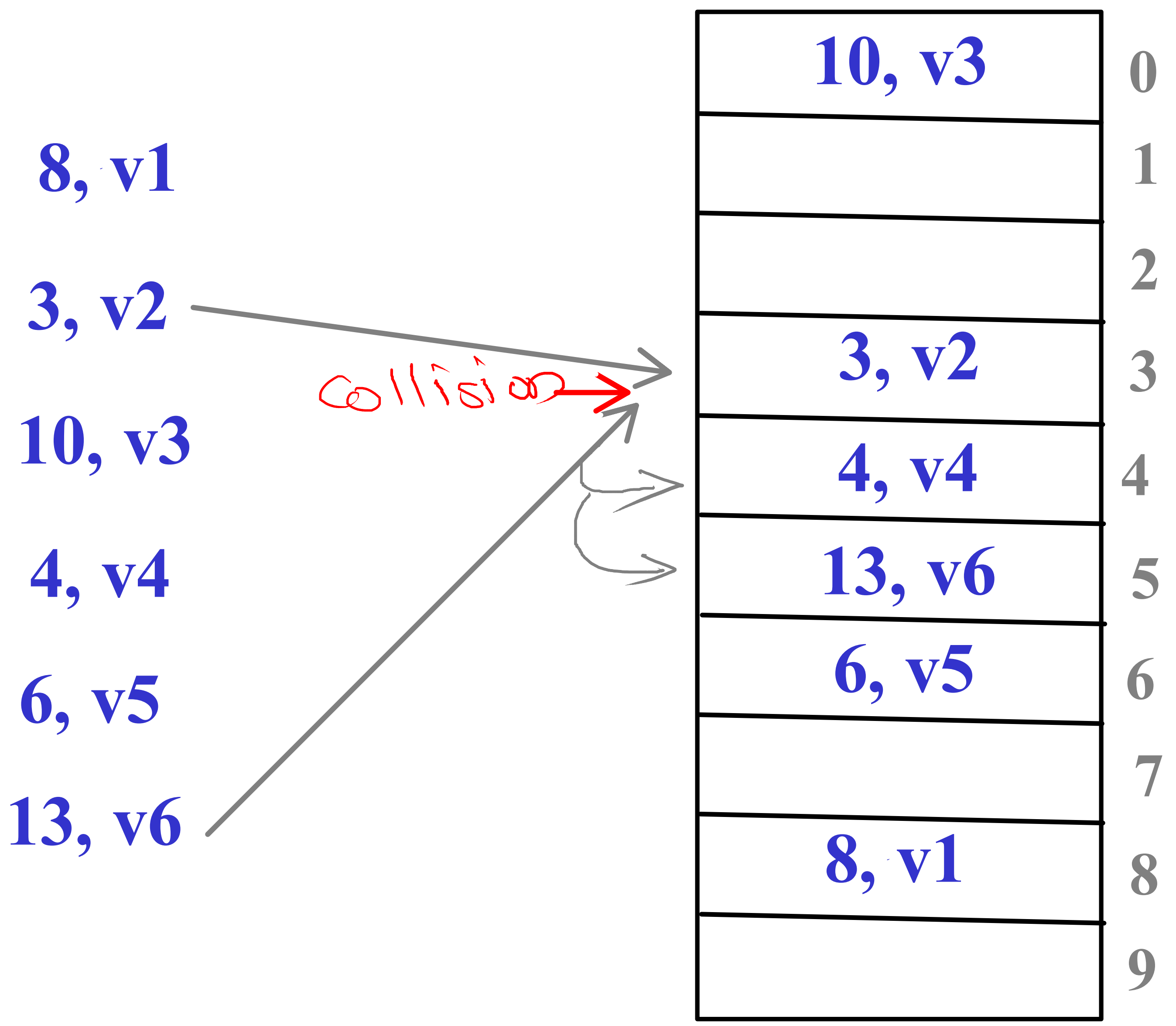
$h(k) = k \ \% \ size$

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

$f(i) = i$

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

size = 10



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

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

$= 4$ (1st probe)(collision)

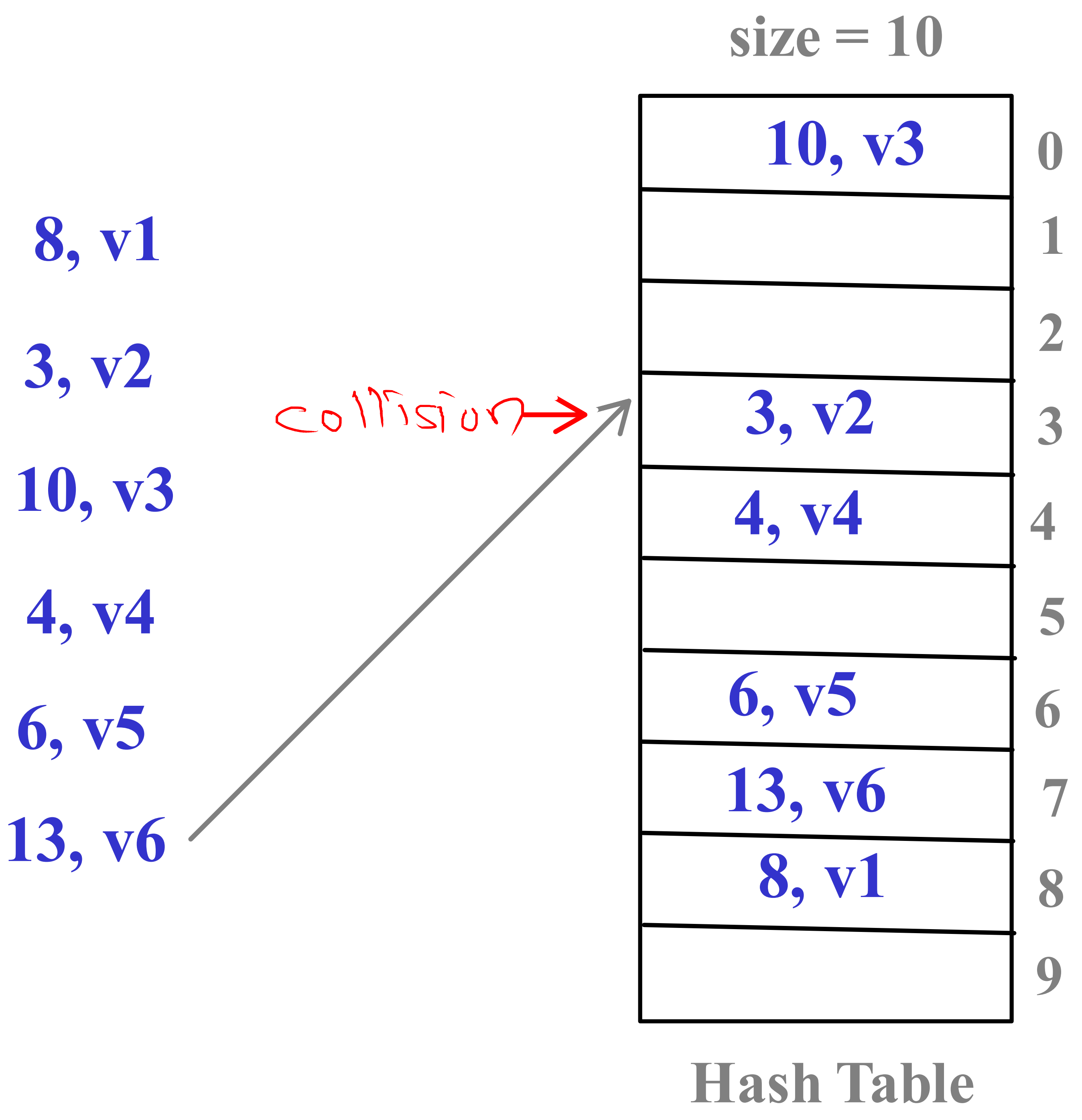
$h(13,2) = [3+2] \% 10$

$= 5$ (2nd probe)

Primary Clustering

it creates long runs of filled slots "near" the hash position of key

Open Addressing - Quadratic Probing



8, v1

3, v2

10, v3

4, v4

6, v5

13, v6

collision

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) = k % size

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

f(i) = i^2

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

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

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

= 4 (1st probe)(collision)

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

= 7 (2nd probe)

Open Addressing - Quadratic Probing

23, v7

33, v8

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

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 (collision)

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

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

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

h(33) = 33 %10 = 3 (collision)

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

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

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

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

Secondary Clustering

it creates long runs of filled slots "away" the hash position of key

- there is not guarantee, for getting free slot to any key

Hashing - Double Hashing

8, v1

3, v2

10, v3

14, v6

size = 11

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

Hash Table

h1(k) = k % size
h2(k) = 7 - (key % 7)

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

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

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

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

$h_1(14) = 14 \% 11 = 3$ (collision)

$h_2(14) = 7 - 0 = 7$

$h(14,1) = [3 + 1 * 7] \% 11$
$= 10$ (1st) (collision)

$h(14,2) = [3 + 2 * 7] \% 11$
$= 6$ (2nd)

- primary as well as secondary clustering is removed
- key value pairss are evenly distributed in table

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 aviable
if n = N	load factor = 1	- no free slots
if n > N	load factor > 1	- can not insert at all

- Rehashing is making the hash table size twice of existing size if hash table is 60 to 70 % full

- In rehashing existing keys are again mapped according to new size of table