Sunbeam Institute of Information Technology
Pune and Karad

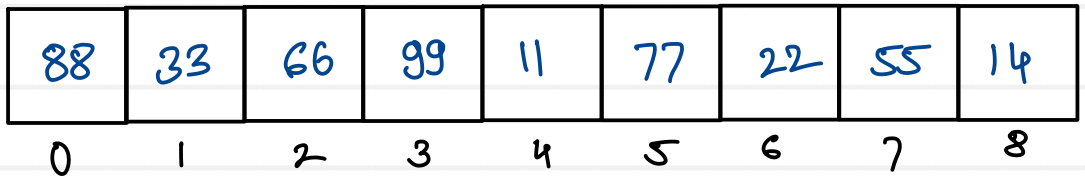
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Linear search (random data)

1. decide/take key from user
2. traverse collection of data from one end to another
3. compare key with data of collection
   - 3.1 if key is matching
     return index/true
   - 3.2 if key is not matching
     return -1/false



Key == arr[i]

77

Key

i = 0, 1, 2, 3, 4, 5

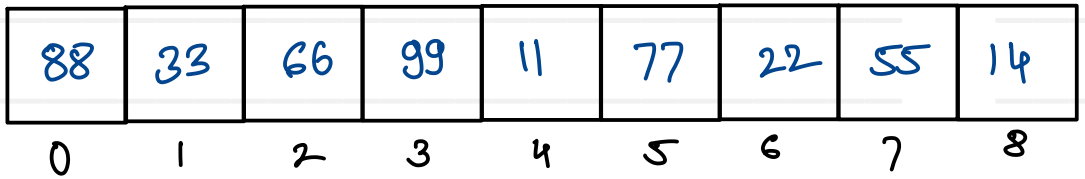
Key is found

89

Key

i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Key is not found



key = 88 → Best case → O(1)

key = 11 → Avg case → O(n)

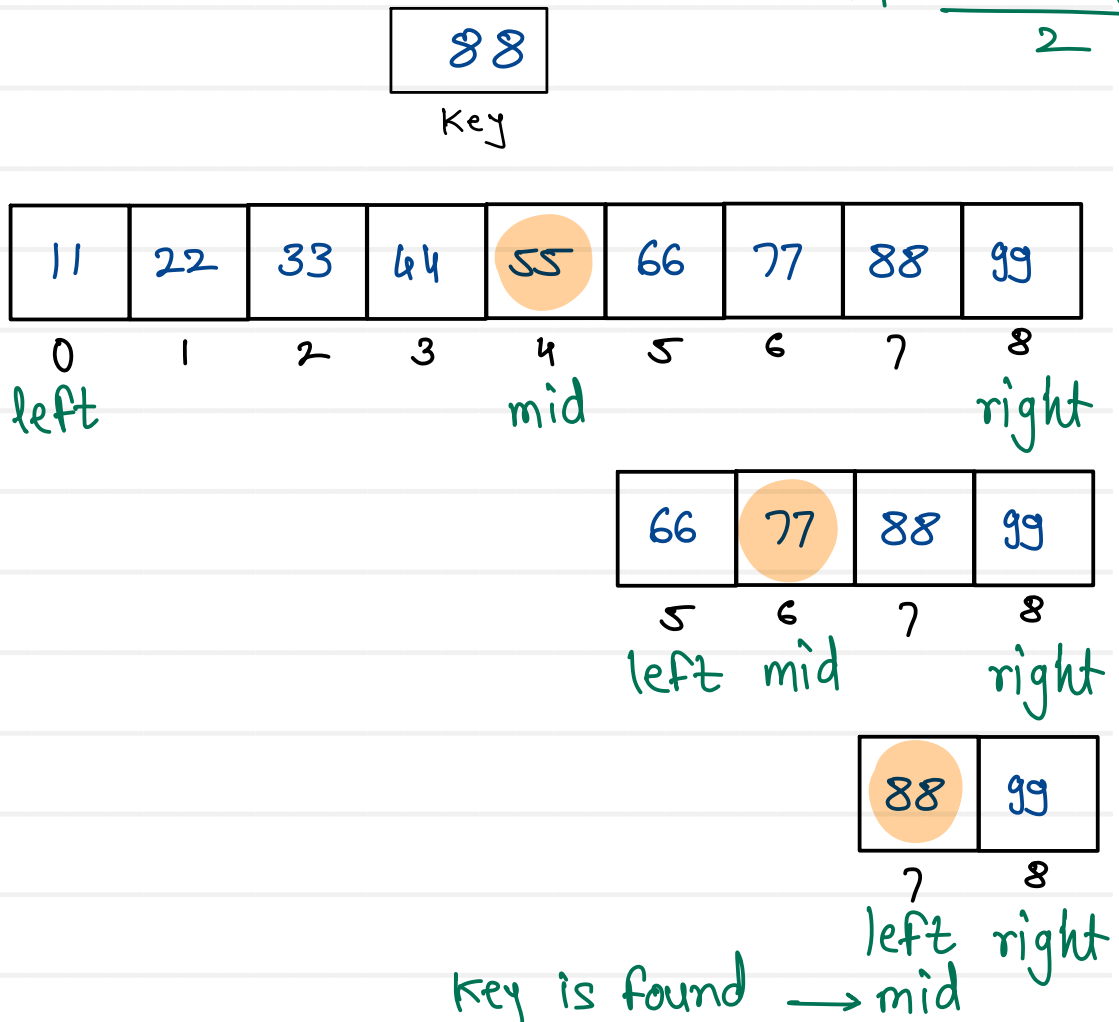
key = 14/100 → Worst case → O(n)

S(n) = O(1)

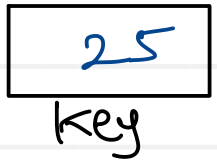
Binary search

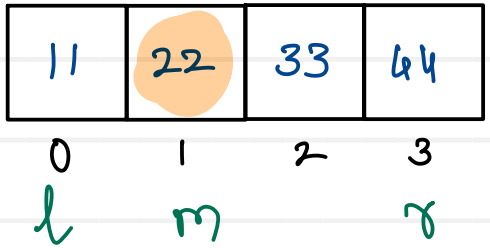
1. take key from user
2. divide array into two parts
   (find middle element)
3. compare middle element with key
3.1 if key is matching
   return index (mid)
3.2 if key is less than middle element
   search key in left partition
3.3 if key is greater than middle element
   search key in right partition
3.4 if key is not matching
   return -1

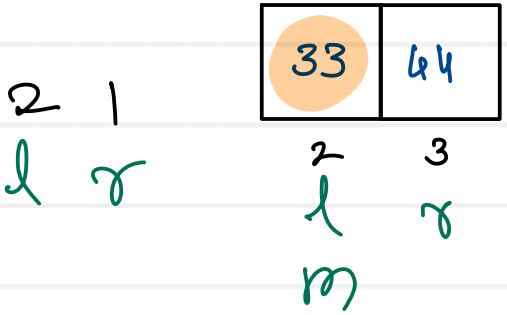
mid = (left + right) / 2



Binary search







valid partition : left <= right
invalid partition : left > right

left partition : left to mid-1
right partition : mid+1 to right

```
l=0 , r=8 , m;
while(l <= r) {
    m = (l+r) /2;
    if( key == arr[m]
        return m;
    else if( key < arr[m])
        right = m-1;
    else
        left = m+1;
}
return -1;
```

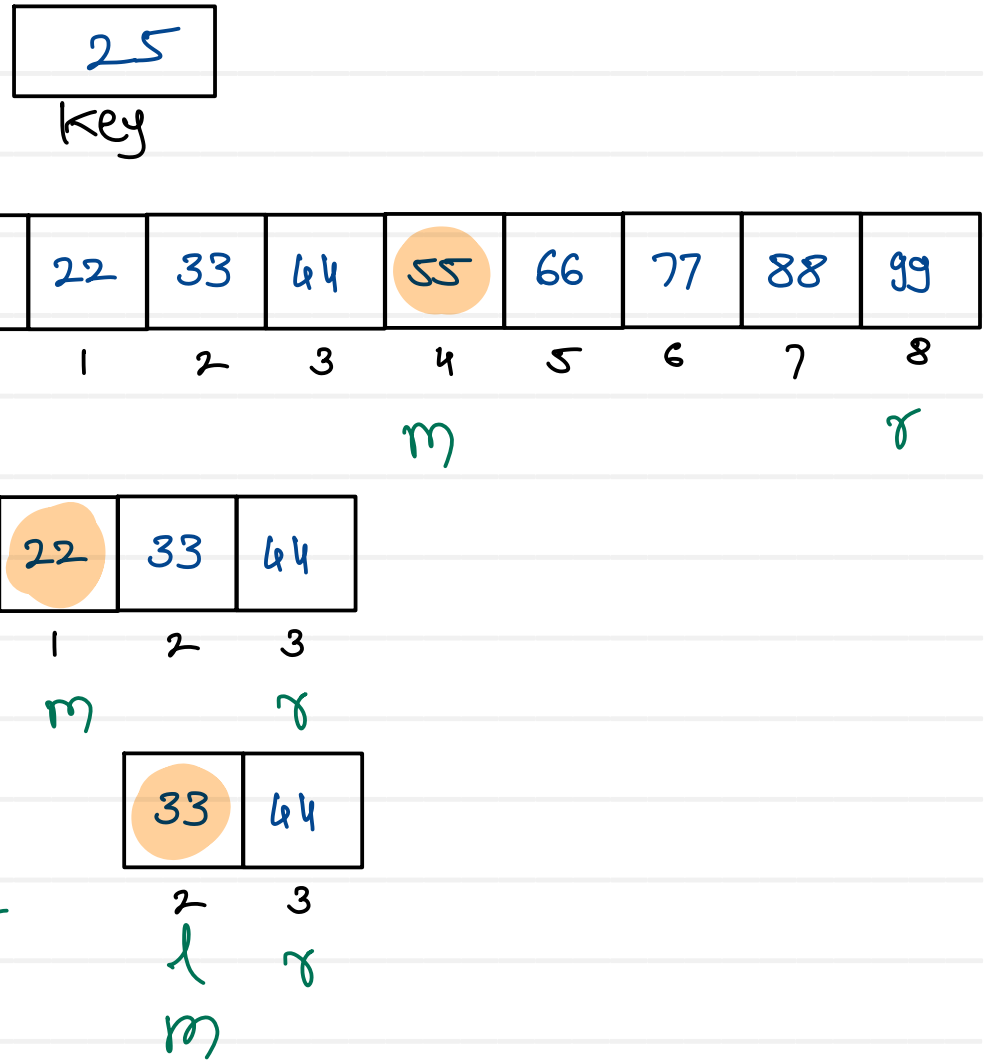
11	22	33	44	55	66	77	88	99
0	1	2	3	4	5	6	7	8

key = 88

l	r	l <= r	m
0	8	T	4
5	8	T	6
7	8	T	7

key = 25

0	8	T	4
0	3	T	1
2	3	T	2
2	1	F	



main()

↓ ↑ -1

binarySearch(arr, 25, 0, 8) m=4

↙ ↗ -1

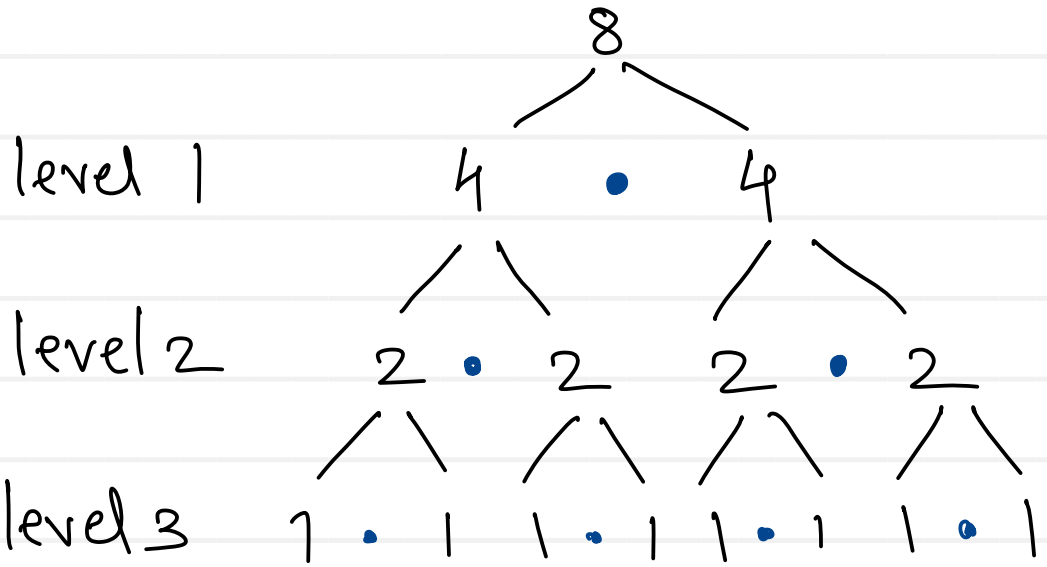
binarySearch(arr, 25, 0, 3) m = 1

↘ ↖ -1

binarySearch(arr, 25, 2, 3) m=2

↙ ↗ -1

binarySearch(arr, 25, 2, 1) X



Best case - first level → O(1)

Avg case - middle level → O(log n)

Worst case - last level / not found → O(log n)

$S(n) = O(1)$

n - no. of elements

l - levels of division

$$2^l = n$$

$$\log 2^l = \log n$$

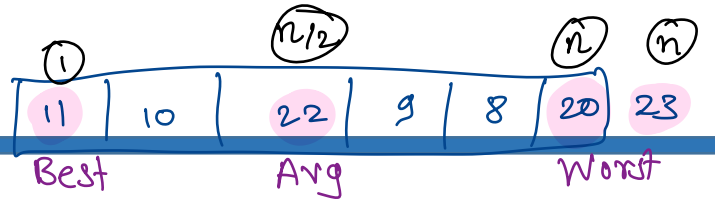
$$l = \frac{\log n}{\log 2}$$

No. of itrs $= l = \frac{\log n}{\log 2}$

$$\text{Time} \propto \frac{1}{\log 2} \log n$$

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

Searching algorithms analysis



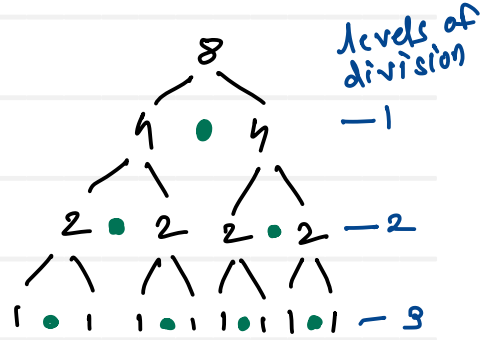
- Time is directly proportional to number of comparisons
- For searching and sorting algorithms, count number of comparisons done

1. Linear search
   - Best case - if key is found at few initial locations → O(1)
   - Average case - if key is found at middle locations → O(n)
   - Worst case - if key is found at last few locations / key is not found → O(n)

S(n)=O(1)

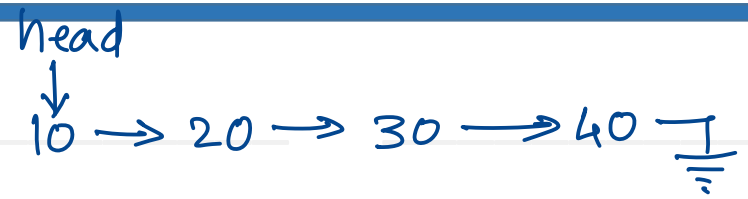
2. Binary search
   - Best case - if key is found at first few levels → O(1)
   - Average case - if key is found at middle levels → O(log n)
   - Worst case - if key is found at last level / not found → O(log n)

S(n)=O(1)



$2^3 = 8 \Rightarrow 2^l = n$ $\quad l = \frac{\log n}{\log 2}$

Singly linear linked list - Display reverse

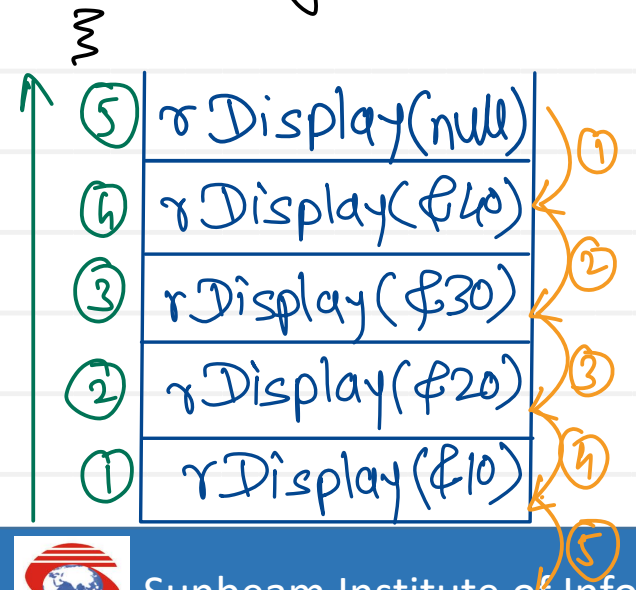


Tail Recursion

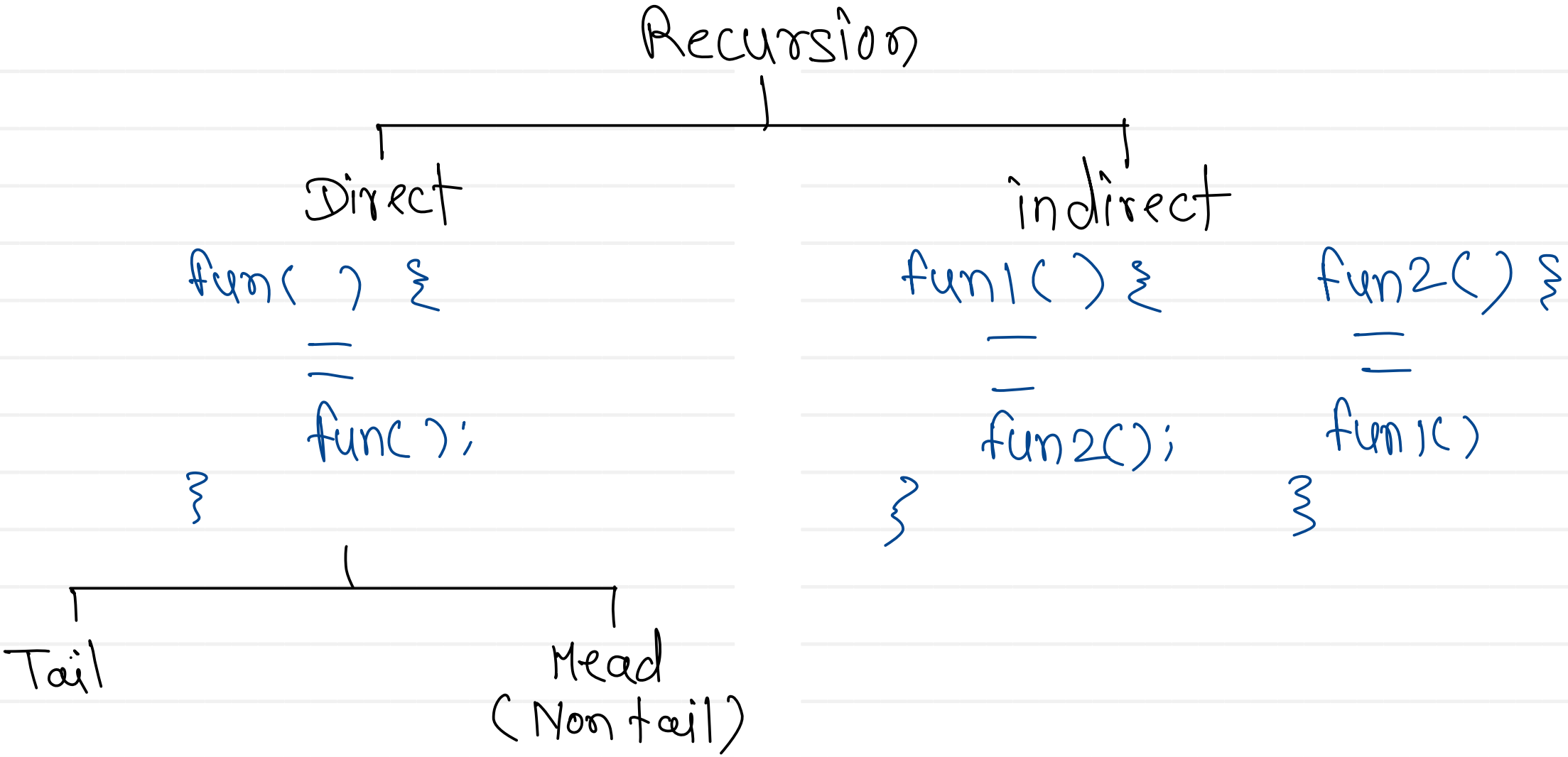
```
void fDisplay(Node trav) {
    if(trav == null)
        return;
    sysout(trav.data);
    fDisplay(trav.next);
}
```

Head Recursion

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



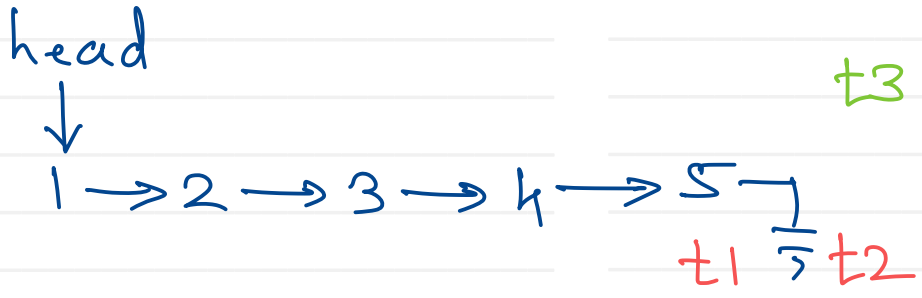
40 30 20 10


Recursion
Direct
indirect
fun( ) {
func();
}
fun1( ) {
fun2();
}
fun2( ) {
fun1()
}
Tail
Mead
(Non tail)

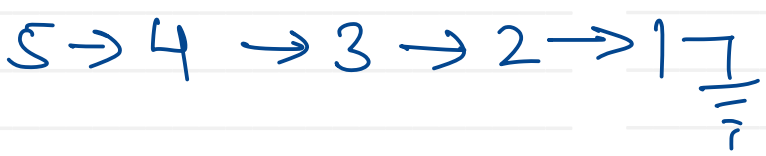
Reverse Linked List

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: []

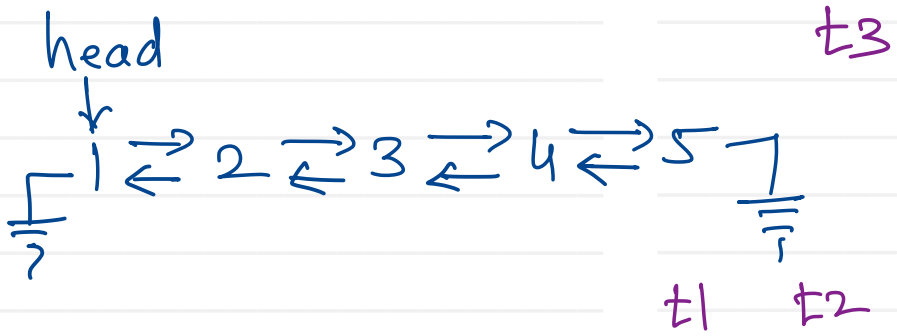
$T(n) = O(n)$
$S(n) = O(1)$

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

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

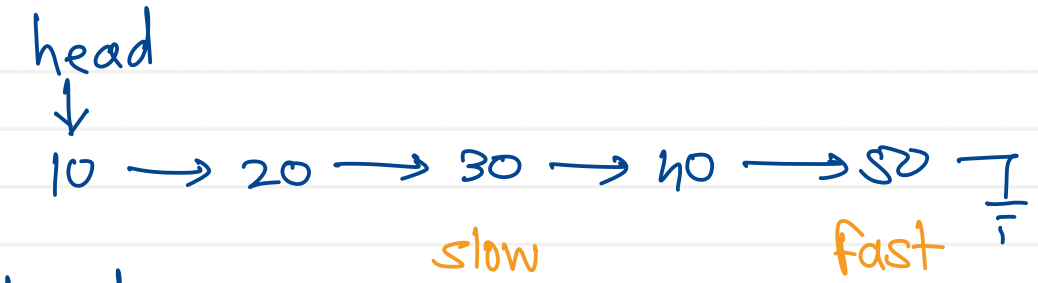
Given the head of a singly linked list, return the middle node of the linked list.

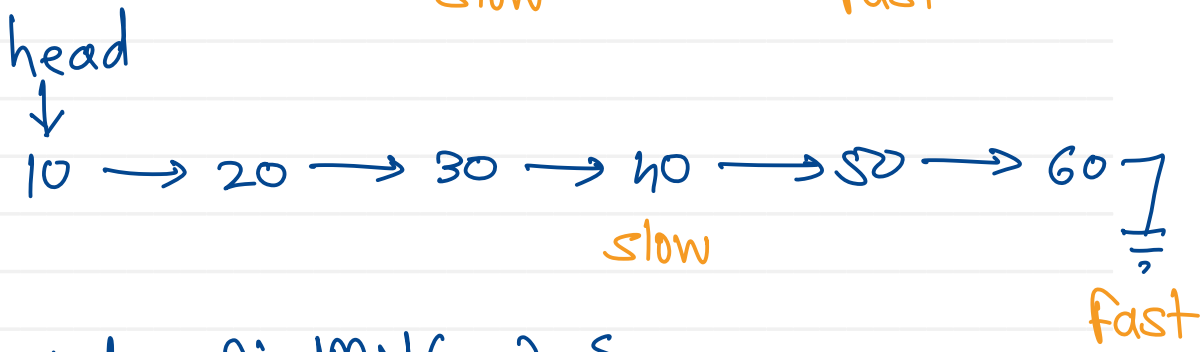
If there are two middle nodes, return the second middle node.

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.

$T(n) = O(n)$
$S(n) = O(1)$






```
Node findMid( ) {
    Node slow = head , fast = head;

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

Linked List Cycle (Floyd cyclic detection)

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.

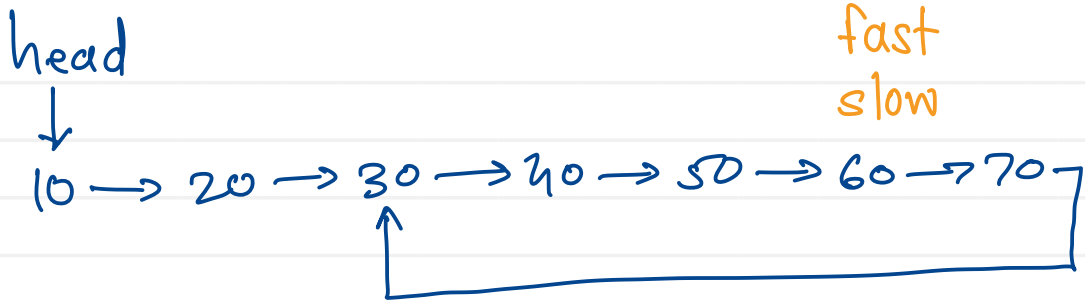
Return true if there is a cycle in the linked list. Otherwise, return false.

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

T(n) = O(n)
S(n) = O(1)



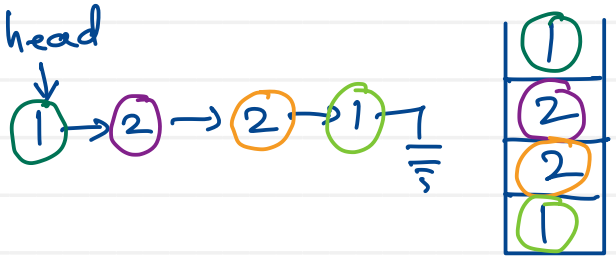

```
boolean hasCycle( ) {
    Node slow = head , fast = head;

    while ( fast != null && fast.next != null)
    {
        fast = fast.next.next;
        slow = slow.next;
        if( fast == slow)
            return true;
    }
    return false;
}
```

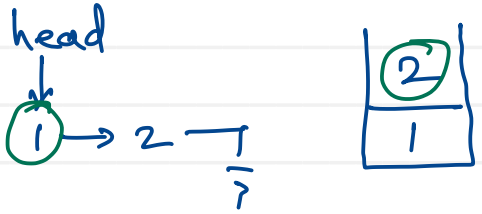
Palindrome Linked List

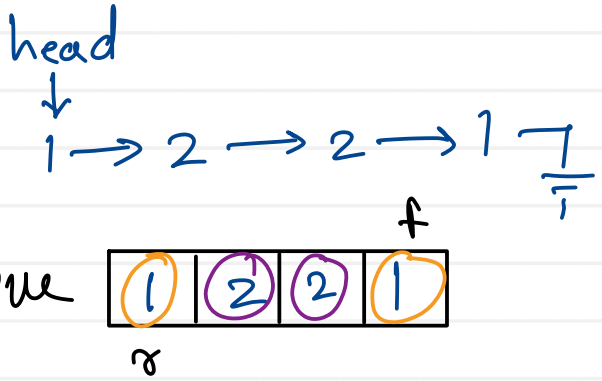
Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

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



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





n
n/2
= 3/2 n

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

total itr = 2n
Time ∝ 2n
T(n) = O(n)
space of stack -> S(n) = O(n)

Hashing

Array : linear search → O(n)
binary search → O(log n)

Linked list : linear search → O(n)

Hashing :
technique which allows to search in constant time O(1)

implementation is called as Hash Table

- 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

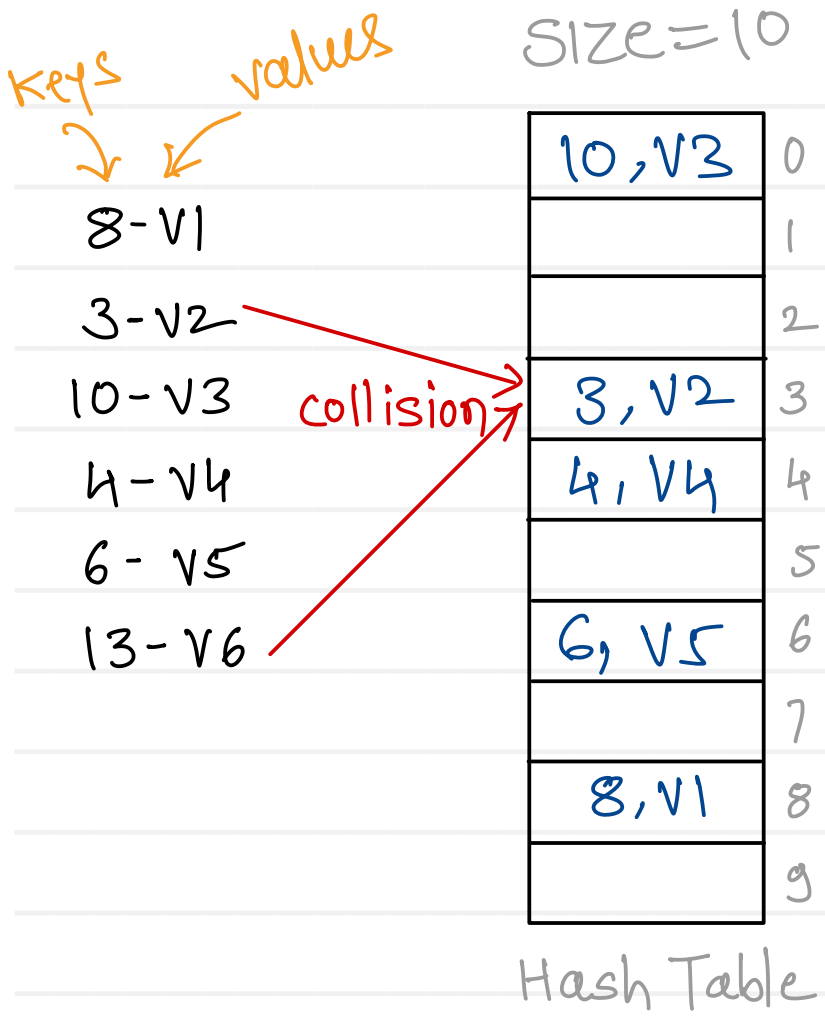
Array - Hash table
Index - Slot

- 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

Hash Functions

1. Division method — k % m
2. Multiplication method - m * (k * A) 0 > A < 1
3. Mid-square method — k^2 & find middle digit
4. Folding method — divide in two part, sum them, % m
5. Cryptographic hashing — MD5, SHA-1 and SHA-256
6. Universal hashing - ((a.k+b) % p) % m
7. Perfect hashing - collision free hash function
   for static set of key.
   - guarantees that no two keys will hash to same value

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

Add: - O(1)
1. slot = h(k)
2. arr[slot] = (key, value)

Search: - O(1)
1. slot = h(k)
2. return arr[slot].value

Delete: - O(1)
1. slot = h(k)
2. arr[slot] = null

Collision :
situation when two different keys give/yield same slot

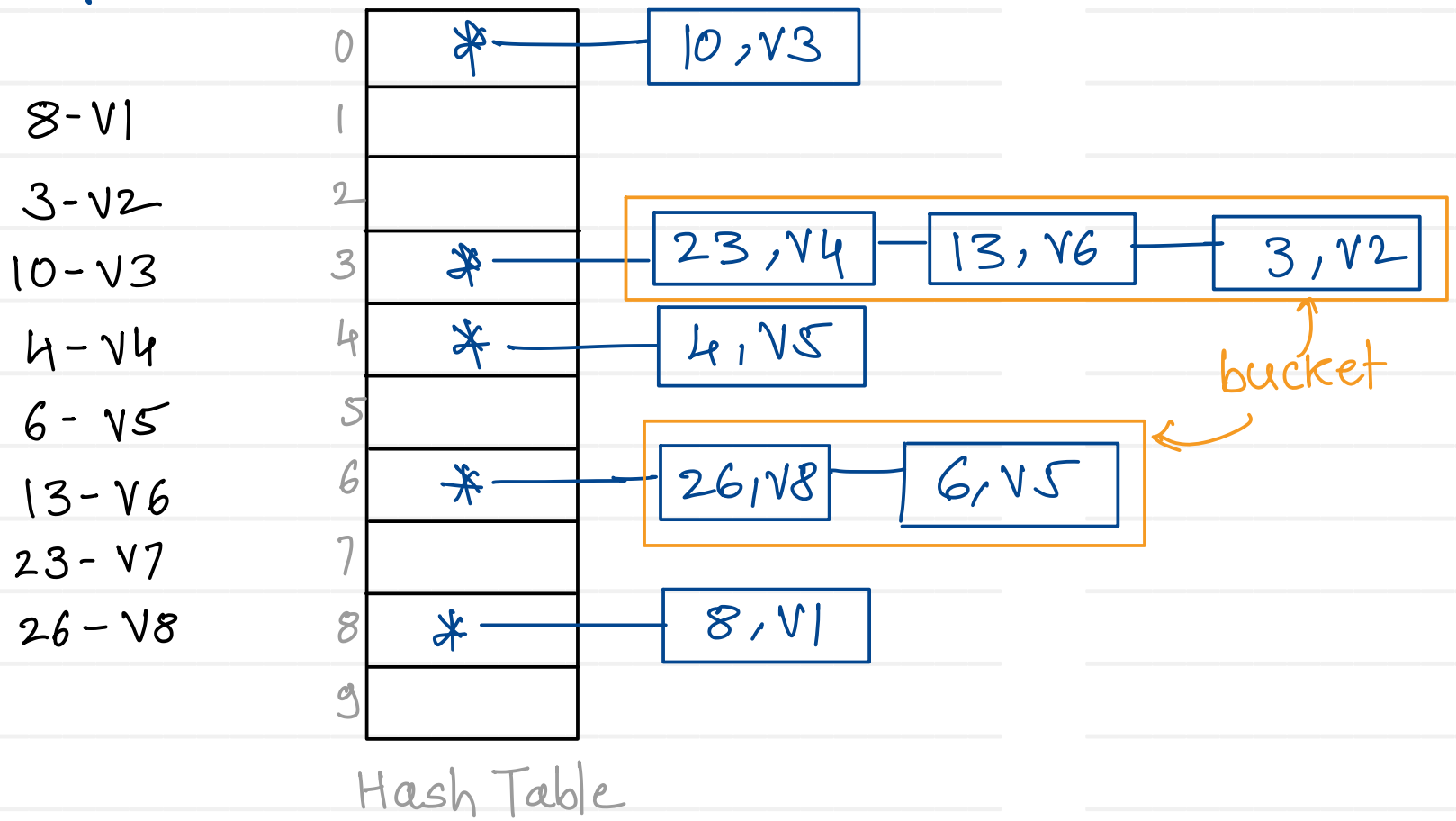
Collision handling techniques :

1. Closed addressing
2. Open addressing
   a. Linear probing
   b. Quadratic probing
   c. Double hashing

Closed Addressing / Chaining / Separate Chaining

open probing

size=10



h(k) = k % size

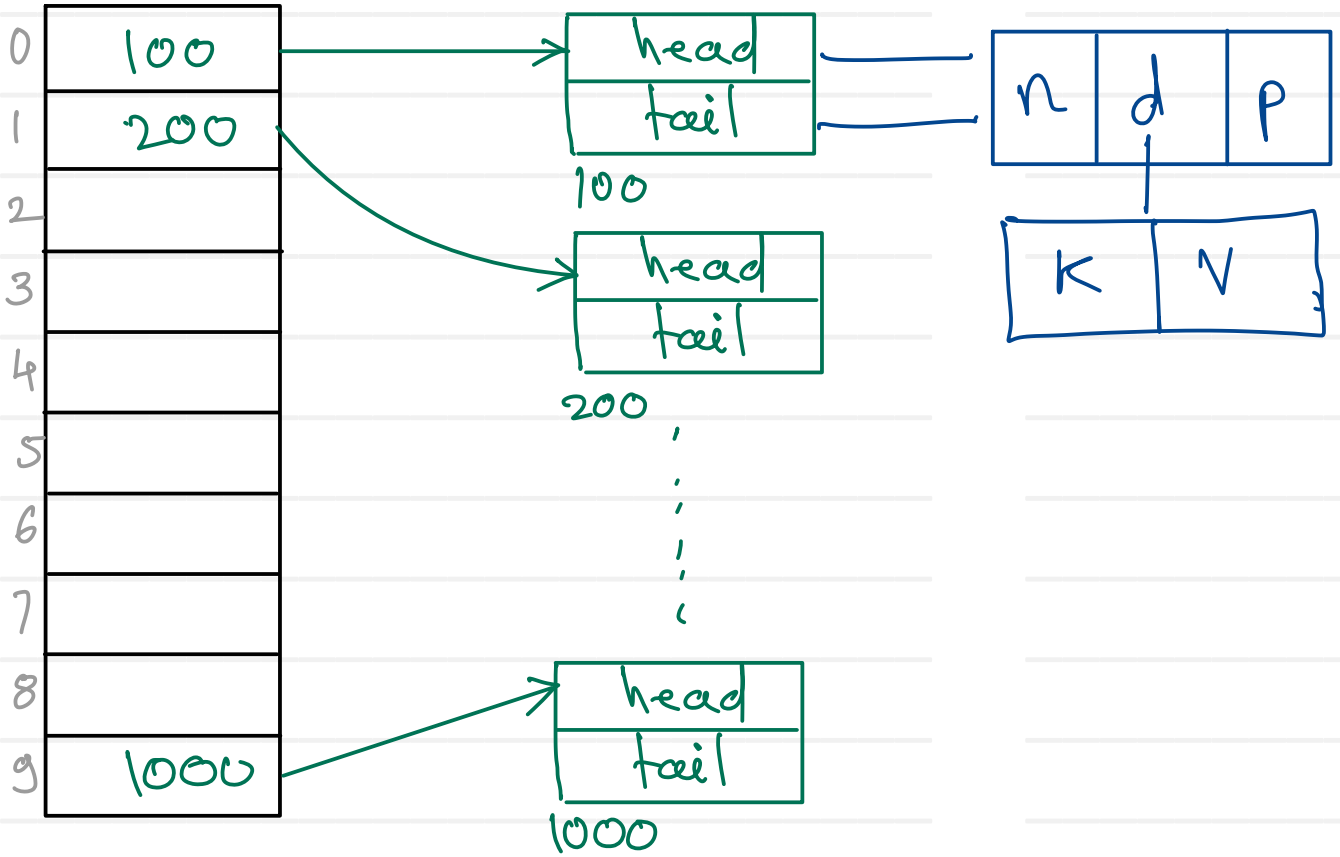
Disadvantage :

1. key-value pairs are stored out side the table
2. space requirement is more
3. worst case time complexity may be O(n)

Advantage :

- no restriction on no. of key-value pairs

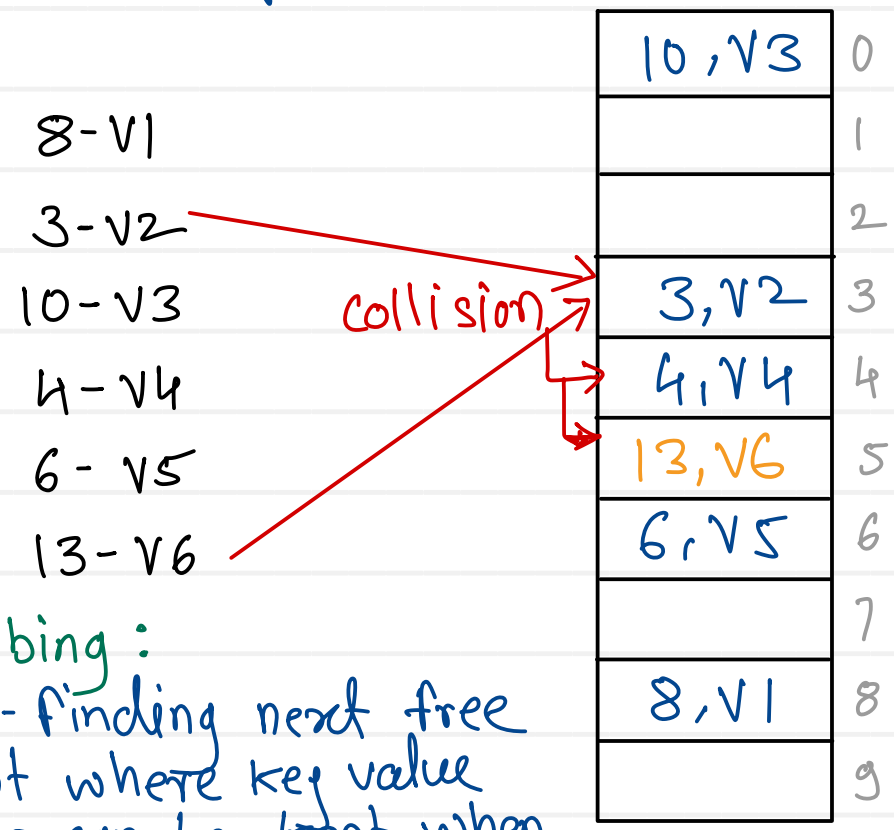
new List[10] new LinkedList();



Open addressing - Linear probing

closed probing

size=10



Hash Table

Probing :
- finding next free slot where key value pair can be kept when collision will occur

h(k) = k % size

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

f(i) = i (i: probe number)

where i = 1, 2, 3, …

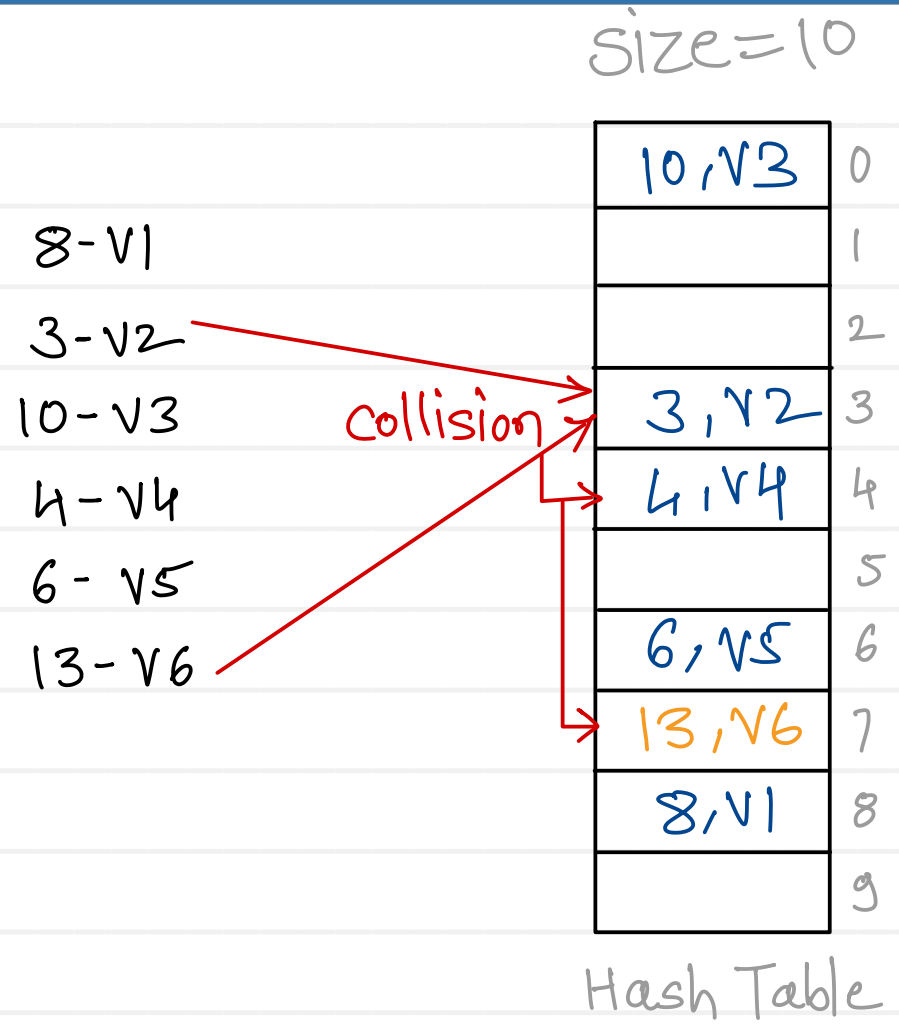
h(13) = 13 % 10 = 3 ©

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

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

Primary clustering :
- near key position table becomes bulky or crowded
- need to take long run of filled slots "near" key position to get empty slot.

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 ☹

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

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

- primary clustering is solved
- there is no guarantee that will get free slot for key-value pair

Open addressing - Quadratic probing

size = 10

8 - V1
3 - V2
10 - V3
4 - V4
6 - V5
13 - V6
23, V7
33, V8

10, V3	0
	1
23, V7	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(23) = 23 % 10 = 3 ©

1st : h(23, 1) = [3 + 1] % 10 = 4 ©

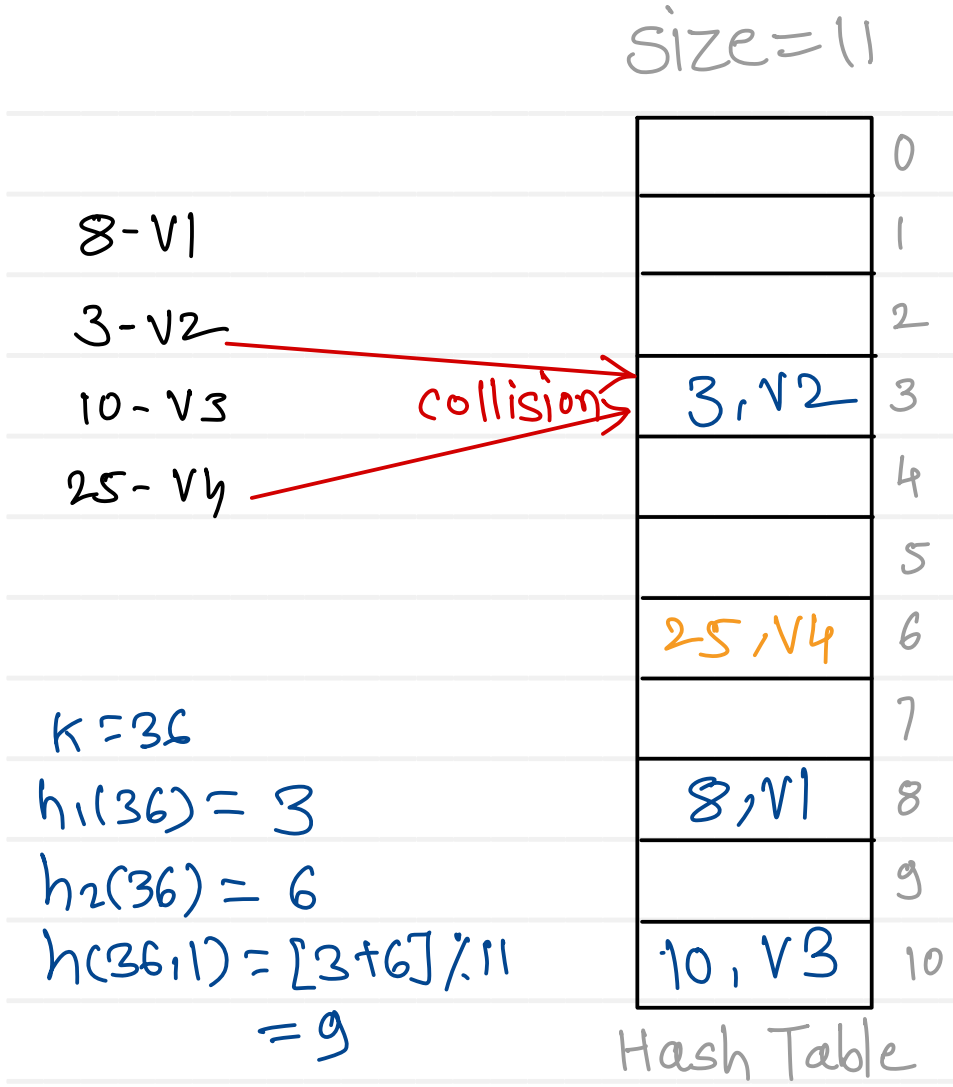
2nd : h(23, 2) = [3 + 4] % 10 = 7 ©

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

Secondary clustering :

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

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)

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

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

Rehashing

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

e.g. $N = 10, n = 7$

$\lambda = \frac{7}{10} = 0.7$ ↓ hash table is 70% filled

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

- 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