Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

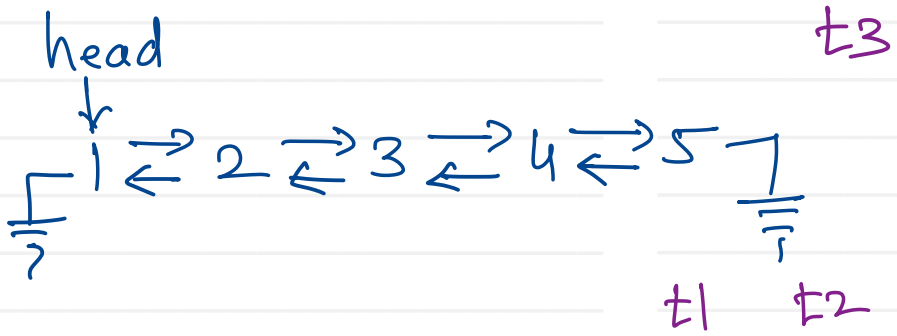
Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

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]

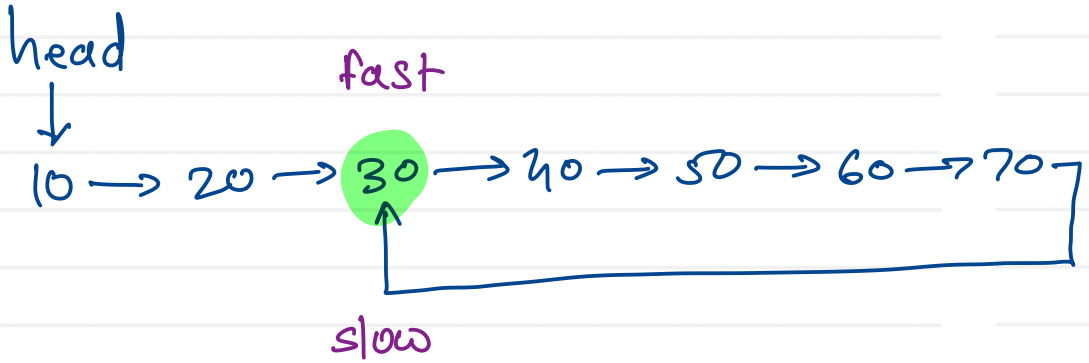
Example 3:

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;
}
```


head
fast
10
20
30
40
50
60
70
slow

Linked list - Applications

- linked list is a dynamic data structure because it can grow or shrink at runtime.
- Due to this dynamic nature, linked list is used to implement other data structures like
  1. Stack
  2. Queue
  3. Hash table
  4. Graph

Stack

(LIFO)

1. Add First
   delete First
2. Add Last
   delete Last

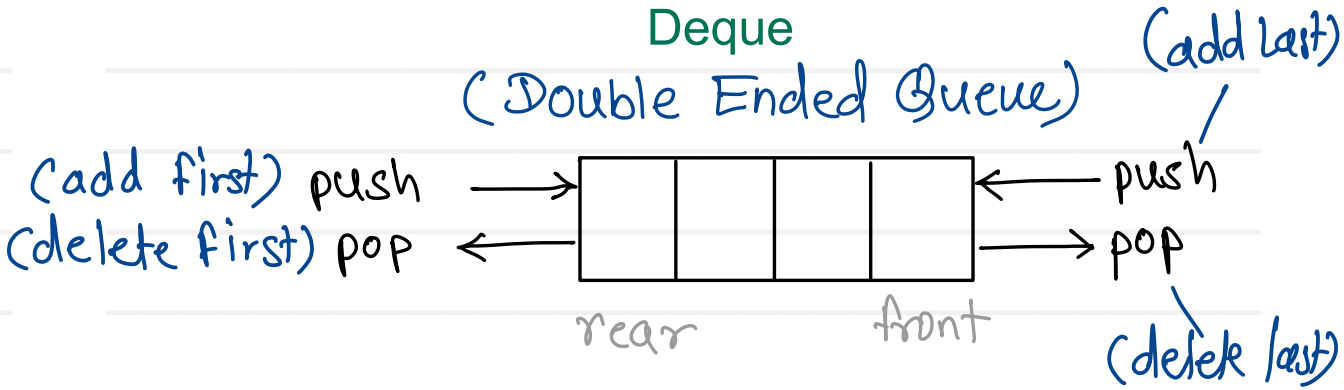
Queue

(FIFO)

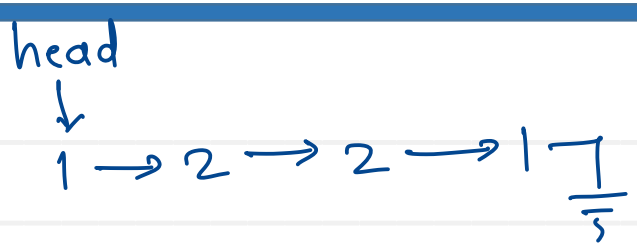
1. Add First
   Delete Last
2. Add Last
   Delete First

Deque

(Double Ended Queue)



1. Input restricted deque
   - one input (push) is closed
2. Output restricted deque
   - one output (pop) is closed



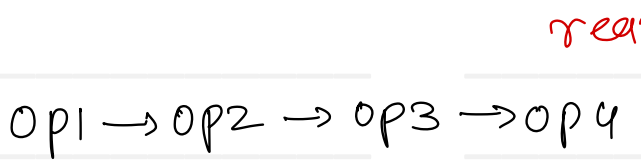
Deque

push = (4) − n + = 2n
pop = (4) − n

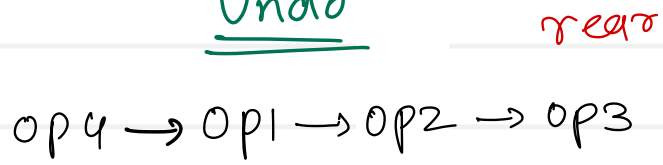
T(n) = O(n)

push = (4) − n + = 3n/2
pop = (2) − n/2

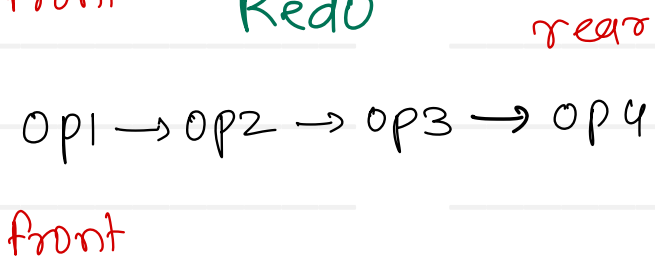
T(n) = O(n)

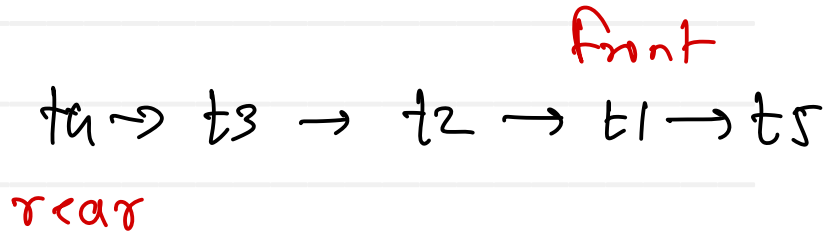


Undo



Redo





Array Vs Linked list

Array

- Array space inside memory is continuous
- Array can not grow or shrink at runtime
- Random access of elements is allowed
- Insert or delete, needs shifting of array elements
- Array needs less space

Linked list

- Linked list space inside memory is not continuous
- Linked list can grow or shrink at runtime
- Random access of elements is not allowed
- Insert or delete, need not shifting of linked list elements
- Linked list needs more space

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

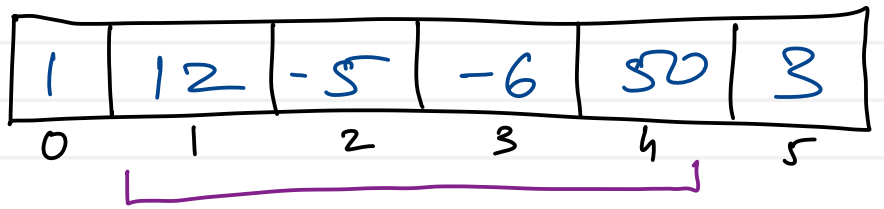
Maximum Average Subarray

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



maxSum = 0, 2, 5] 42

$$maxAvg = \frac{maxSum}{k}$$

maxSum = 1+12-5-6 = 2

= maxSum - left ele + right ele

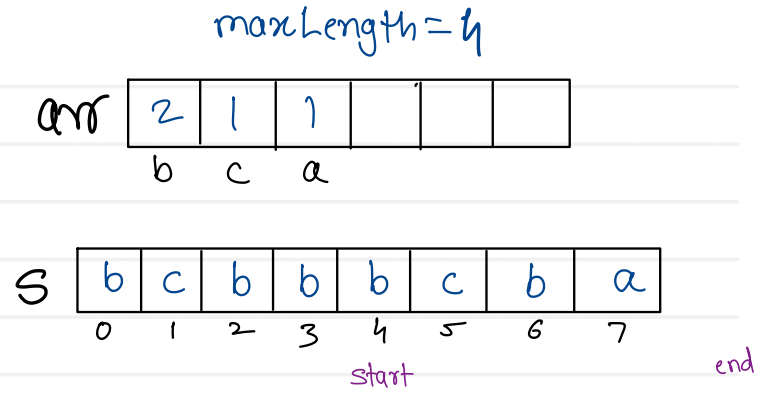
= 2 - 1 + 50 = 51

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;
}
```

Three pointers Technique

- Three-Pointer Technique is an extension of the Two-Pointer approach introducing an additional pointer to optimize the traversal of arrays and linked lists.

- The third pointer provides extra flexibility by allowing the algorithm
  - to track or manipulate multiple conditions simultaneously
  - making it useful for solving problems that involve triplets
  - partitioning data
  - maintaining three different states.

- The key idea is to place three pointers within a data structure and update them based on specific conditions. This allows for efficient traversal and decision-making without redundant computations.

Fast and slow pointers Technique

- Middle of the Linked list
- Nth node from the end of the Linked list
- Detect loop in the Linked list
- Find the starting point of loop in the Linked list
- Remove Loop in the Linked List

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]

Example 2:

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

Output: [1,2]

Example 3:

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

Output: [0,1]

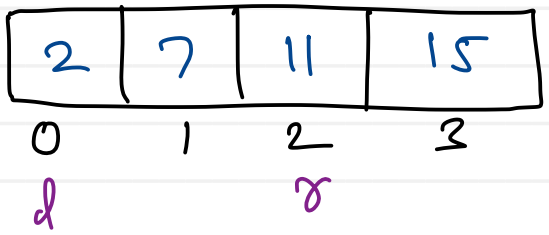
$T(n) = O(n^2)$

$S(n) = O(1)$

```
int[] twoSum(int[] nums, int target) {
    for(i=0; i<nums.length-1; i++){
        for(j=i+1; j<nums.length; j++){
            if(nums[i]+nums[j]==target)
                return new int[]{i,j};
        }
    }
    return new int[]{};
}
```

Two pointers Technique

- The two-pointer technique is a widely used approach to solving problems efficiently, which involves arrays or linked lists.
- This method involves traversing arrays or lists with two pointers moving at different speeds or in different directions.
- This technique is used to solve problems more efficiently than using a single pointer or nested loops.



target = 9

l	r	
0	3	2+15 = 17 > 9
0	2	2+11 = 13 > 9
0	1	2+7 = 9 = 9

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

- Find a pair of elements that sum up to a target.
  - Array: [2, 7, 11, 15]
  - Target Sum: 9
- Use two index variables **left** and **right** to traverse from both corners.
  - Initialize: left = 0, right = n – 1
  - Run a loop while **left < right,** do the following inside the loop
- Compute current sum, **sum** = arr[left] + arr[right]
- If the **sum** equals the **target**, we've found the pair.
- If the **sum** is less than the **target**, move the **left** pointer to the right to increase the **sum**.
- If the **sum** is greater than the **target**, move the **right** pointer to the left to decrease the **sum**.

Two sum

sorted

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]

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6
Output: [0,1]

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length-1;
    while(left < right) {
        sum = nums[left] + nums[right];
        if(sum == target)
            return new int[] {left, right};
        else if(sum < target)
            left++;
        else
            right++;
    }
    return new int[] { };
}
```

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.

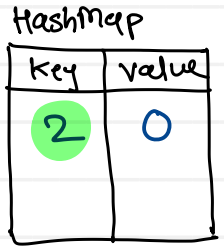
You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

Input: nums = [2,7,11,15], target = 9

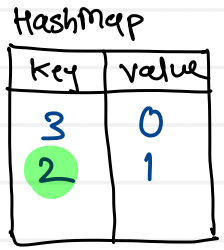
Output: [0,1]



Example 2:

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

Output: [1,2]



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[]{};
}
```

Hashing

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

Linked List :
linear search - O(n)

Hash Table
search - O(1)

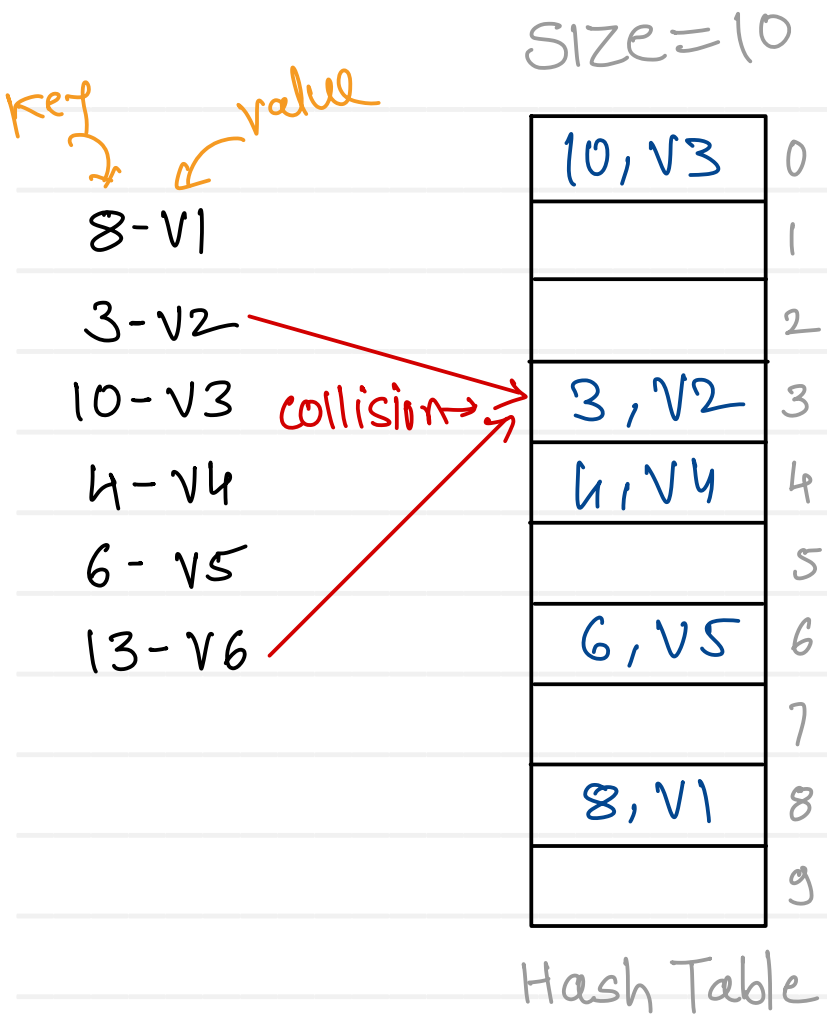
- hashing is a technique in which data can be inserted, deleted and searched in constant average time O(1)
- Implementation of hashing is known as hash table
- Hash table is array of fixed size in which elements are stored in key - value pairs

Array - Hash table
Index - Slot

Associate access

- 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 code

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

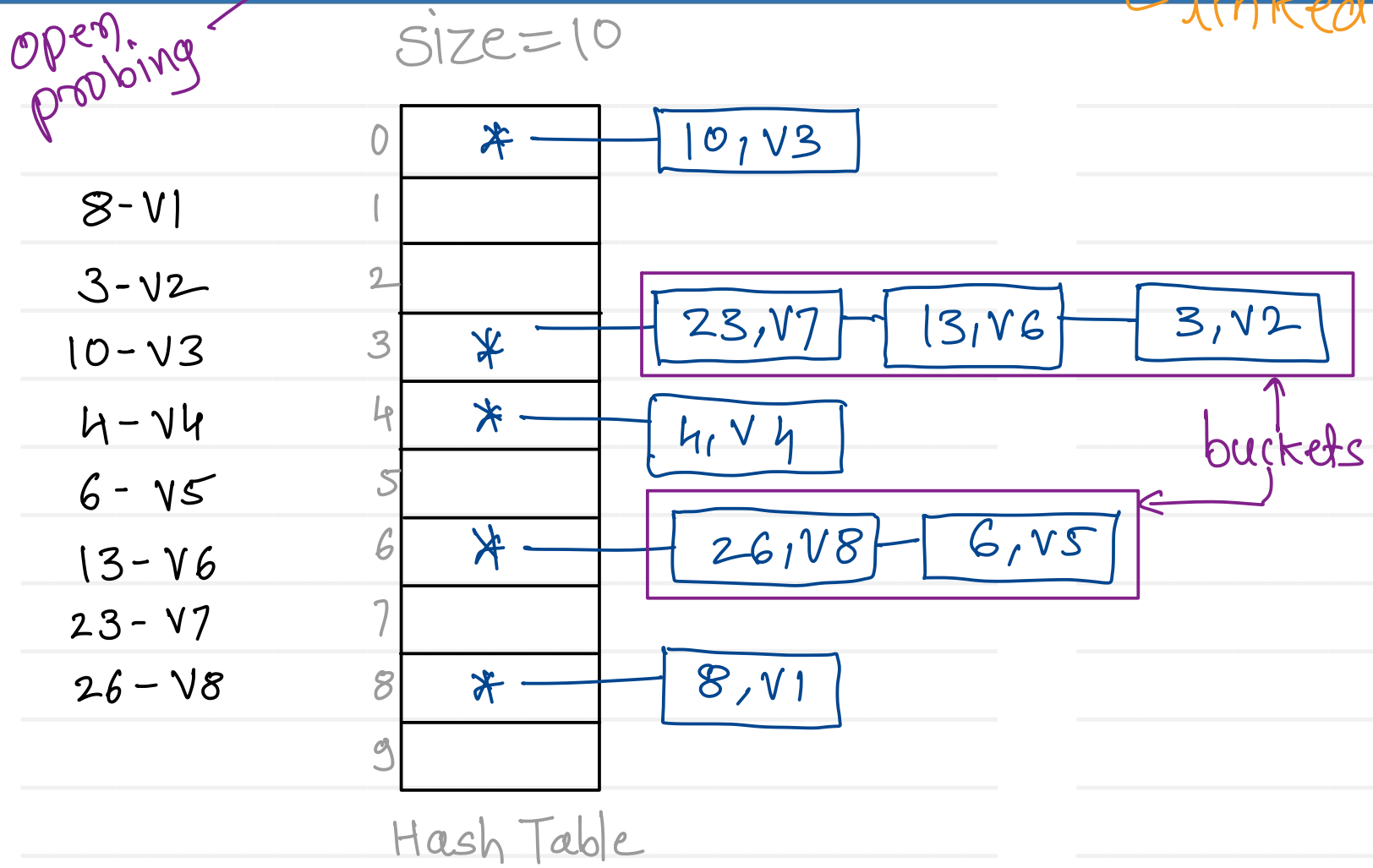
Collision handling/resolution Techniques
1. Closed Addressing
2. Open Addressing
   i. linear probing
   ii. Quadratic probing
   iii. Double hashing

Closed Addressing / Chaining / Separate Chaining

open probing

↳ linked list per slot

h(k) = k % size



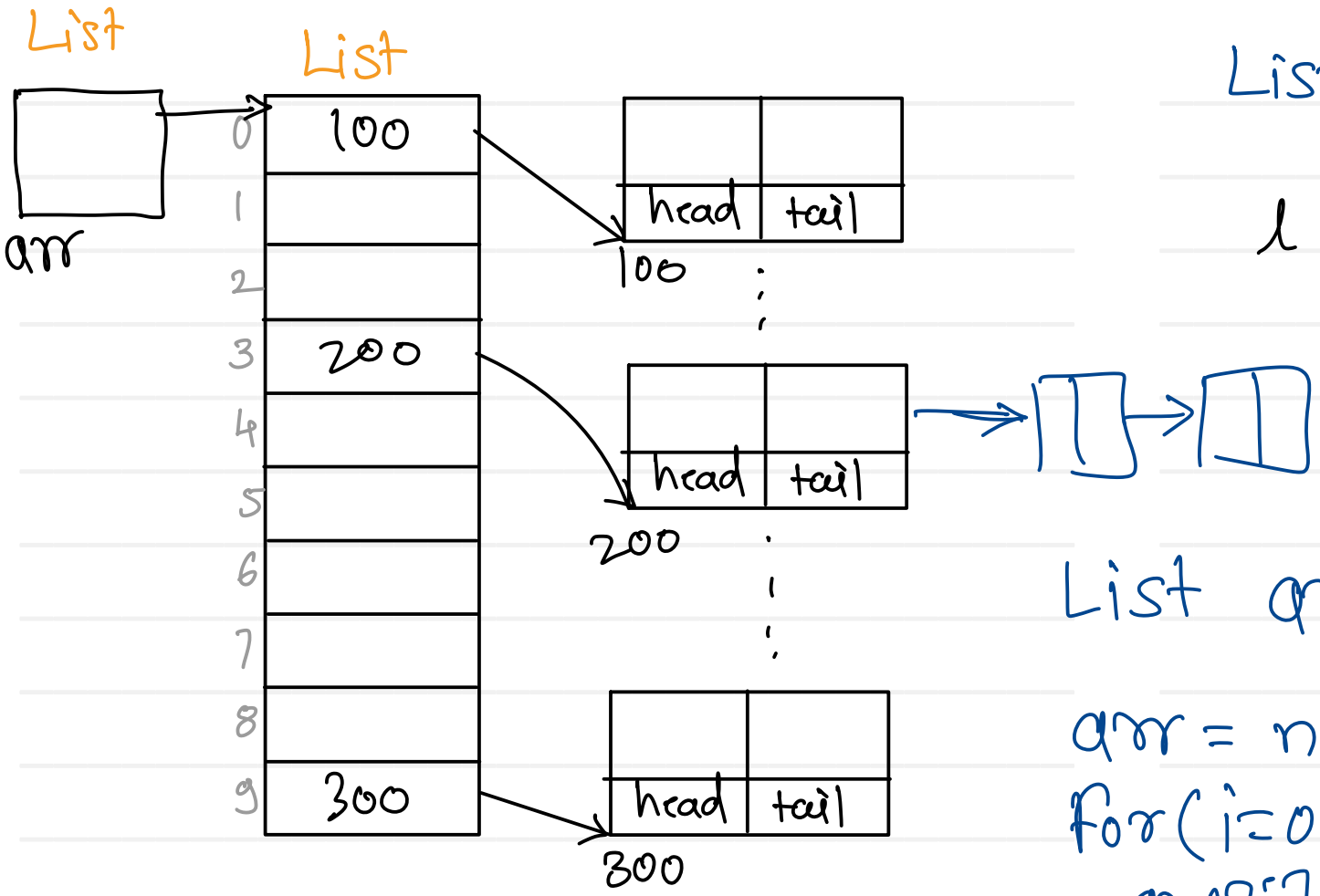
Advantage:

multiple key value pairs can be stored into table.

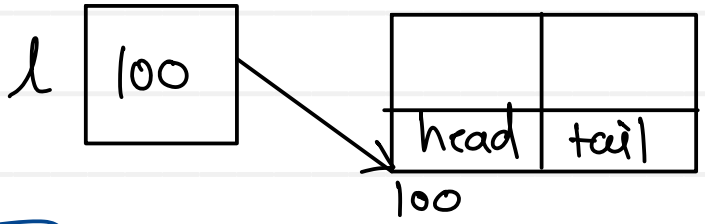
Disadvantages:

1. key value pairs are stored out the table.
2. needs more space
3. Worst case time complexity = O(n)

↳ maximum keys yeild same slot




```
List l = new LinkedList();
```




```
List arr[];
arr = new List[SIZE]
for(i=0; i<SIZE, i++)
  arr[i] = new LinkedList()
```

Open addressing - Linear probing

closed probing

size=10

8-V1

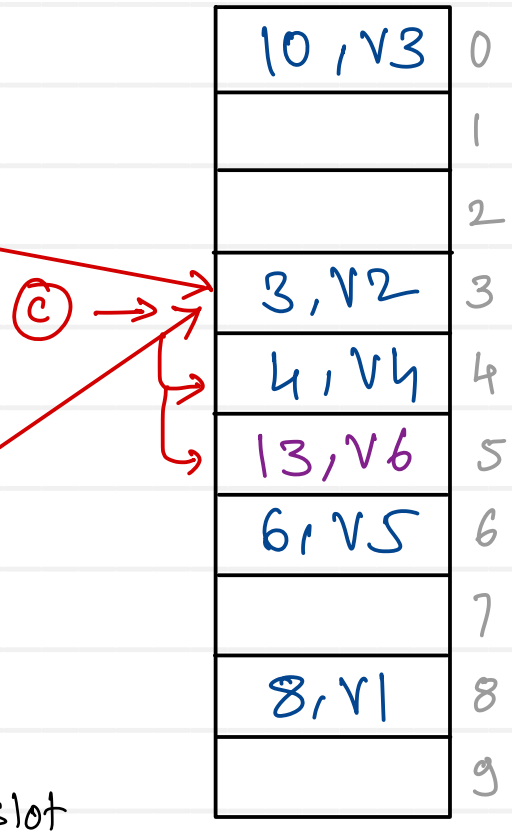
3-V2

10-V3

4-V4

6-V5

13-V6



Probing:
finding next free slot to store key value pair whenever 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 ©

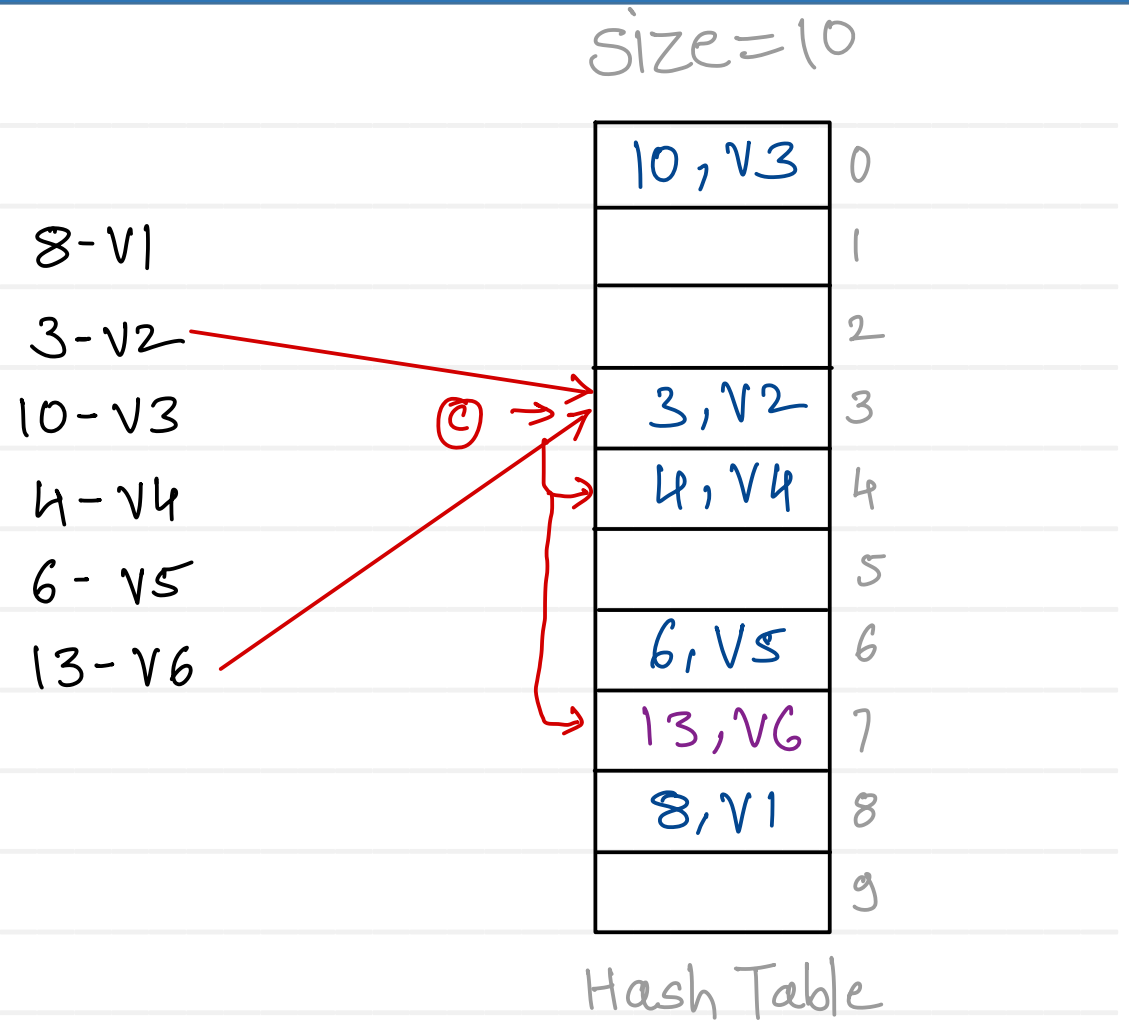
h(13,1) = [3+1] % 10 = 4 (1st probe) ©

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

Primary clustering:

1. table becomes bulky/crowded near key position.
2. to find free slot for key when collision will occur, need to take long run of filled slots "near" key position

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 ©

h(13,1) = [3 + 1] % 10 = 4 (1st) ©

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

- primary clustering is solved

Secondary clustering

- need to take long run to find empty slot "away" key position.

- there is no grantee of getting slot for key.

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
33, V8	9

Hash Table

h(k) = k % size

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

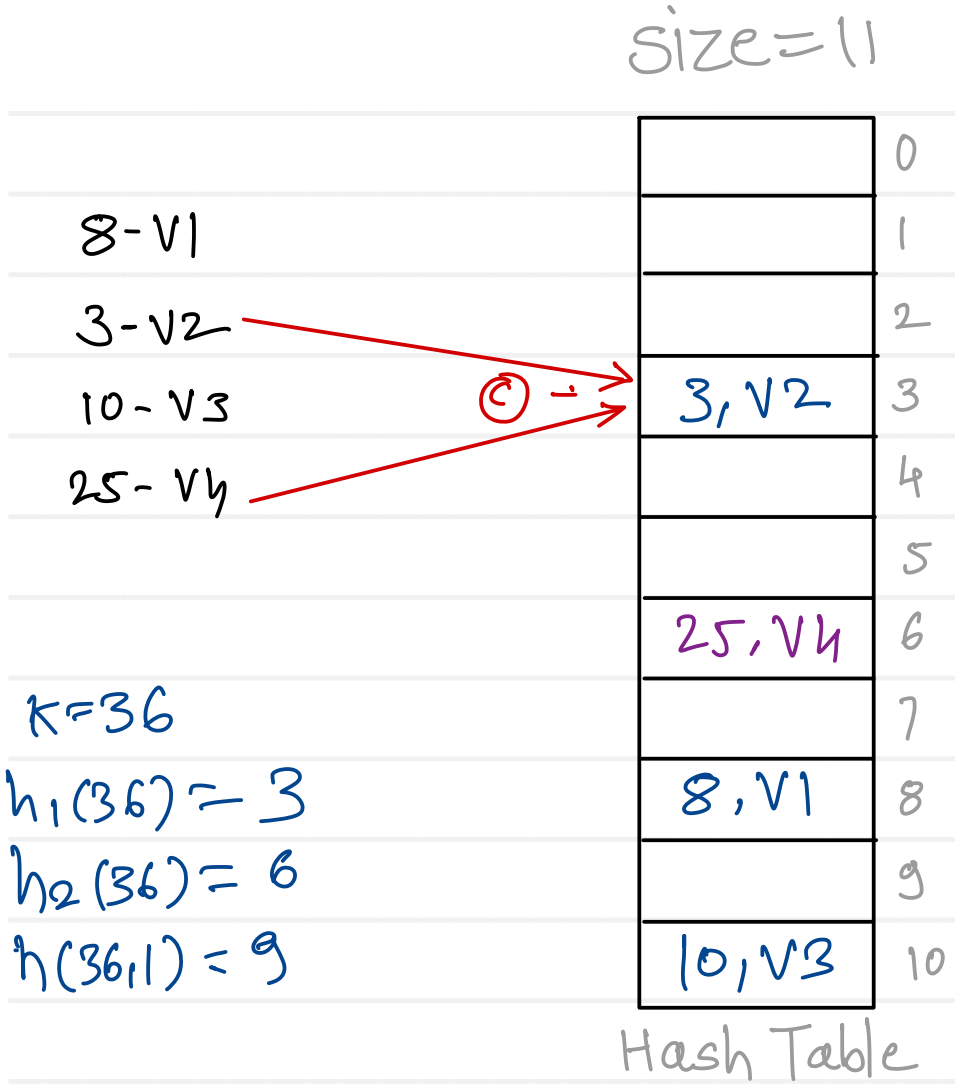
f(i) = i^2

where i = 1, 2, 3, …

h(23) = 23 % 10 = 3 (C)
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) = 3 % 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)
h(33, 4) = [3 + 16] % 10 = 9 (4th)

Open addressing - Double hashing



h1(k) = k % size

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

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

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

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

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

$h_1(25) = 25 \% 11 = 3$ ©

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

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

K=36

$h_1(36) = 3$

$h_2(36) = 6$

$h(36, 1) = 9$

- primary & secondary clustering is removed.

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