Algorithm Design Technique

Brute Force approach

- we check for all the possibilities of input
- need nested loops
- $T(n) = O(n^2), O(n^3) \ldots$

e.g. selection sort,
bubble sort,
insertion sort

Divide and Conquer

- divide bigger problem in small-small problems.
- solve small problems individually
- merge (conquer) solutions together to get final solution.

e.g. Merge sort, Quick sort

Merge Sorted Array

Given two sorted integer arrays nums1 and nums2 in ascending order, and two integers m and n representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in ascending order.

The final sorted array should be stored inside the array nums1. To accommodate this, nums1 has a length of m + n.

Example 1:
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]

Example 2:
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]

Example 3:
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]

As sorted array to be stored in nums1 again, need to start from last elements of the arrays.

before

← i (at index 2 of nums1)

	0	1	2	3	4	5
nums1	1	2	3	0	0	0

← j (at index 2 of nums2)

	0	1	2
nums2	2	5	6

compare ith & jth element, whichever is maximum put it into nums1 from last index

after

← k (at index 5 of nums1)

	0	1	2	3	4	5
nums1	1	2	2	3	5	6

Time complexity:
$\boxed{T(n) = O(n)}$

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

i

0 1 2 3

HashMap

key	value
2	0

Example 2:

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

Output: [1,2]

i

0 1 2

HashMap

key	value
3	0
2	1

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[]{ };
}
```

$T(n) = O(n)$

$S(n) = O(n)$

Contains Duplicate

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1: i

Input: nums = [1,2,3,1]

Output: true

Hashset

$\boxed{1, 2, 3}$

Example 2: i

Input: nums = [1,2,3,4]

Output: false

Hashset

$\boxed{1, 2, 3, 4}$

Example 3:

Input: nums = [1,1,1,3,3,4,3,2,4,2]

Output: true

```
boolean containsDuplicate( int[] nums) {
    Set<Integer> s = new HashSet<>();
    for( int n : nums ) {
        if( s.add(n) == false)
            return true;
    }
    return false;
}
```

$T(n) = O(n)$

Valid Anagram

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:
Input: s = "anagram", t = "nagaram"
Output: true

Example 2:
Input: s = "rat", t = "car"
Output: false

```
boolean isAnagram(char[] s, char[] t){
    int count[26] = {0};

    for(int i=0; s[i] != '\0'; i++)
        count[s[i]-'a']++;

    for(int i=0; t[i] != '\0'; i++)
        count[t[i]-'a']--;

    for(int i=0; i<26; i++)
        if(count[i] != 0)
            return false;
    return true;
}
```

Hashing

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

Linked List : search - $O(n)$

Binary tree : search - $O(n)$

BST : search - $O(\log n)$

Graph : search - $O(n)$

Hashing : 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

unique can be duplicate

Array - Hash table Associative access
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 code - result of mathematical function after supplying given key.

Hashing

size = 10

Key - Value

8 - V1
3 - V2
10 - V3
4 - V4
6 - V5
13 - V6

collision-

0	10, V3
1	
2	
3	3, V2
4	4, V4
5	
6	6, V5
7	
8	8, V1
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 :
when multiple keys yeild (give) same slot collision will occur in hash table

insert : — O(1)
i) find slot = h(key)
ii) arr[slot] = (key, value)

search : — O(1)
i) find slot = h(k)
ii) return arr[slot].value

delete : — O(1)
i) find slot = h(k)
ii) arr[slot] = null;

Collision handling techniques:
1) Closed addressing
2) Open addressing
i) linear probing
ii) Quadratic probing
iii) Double hashing

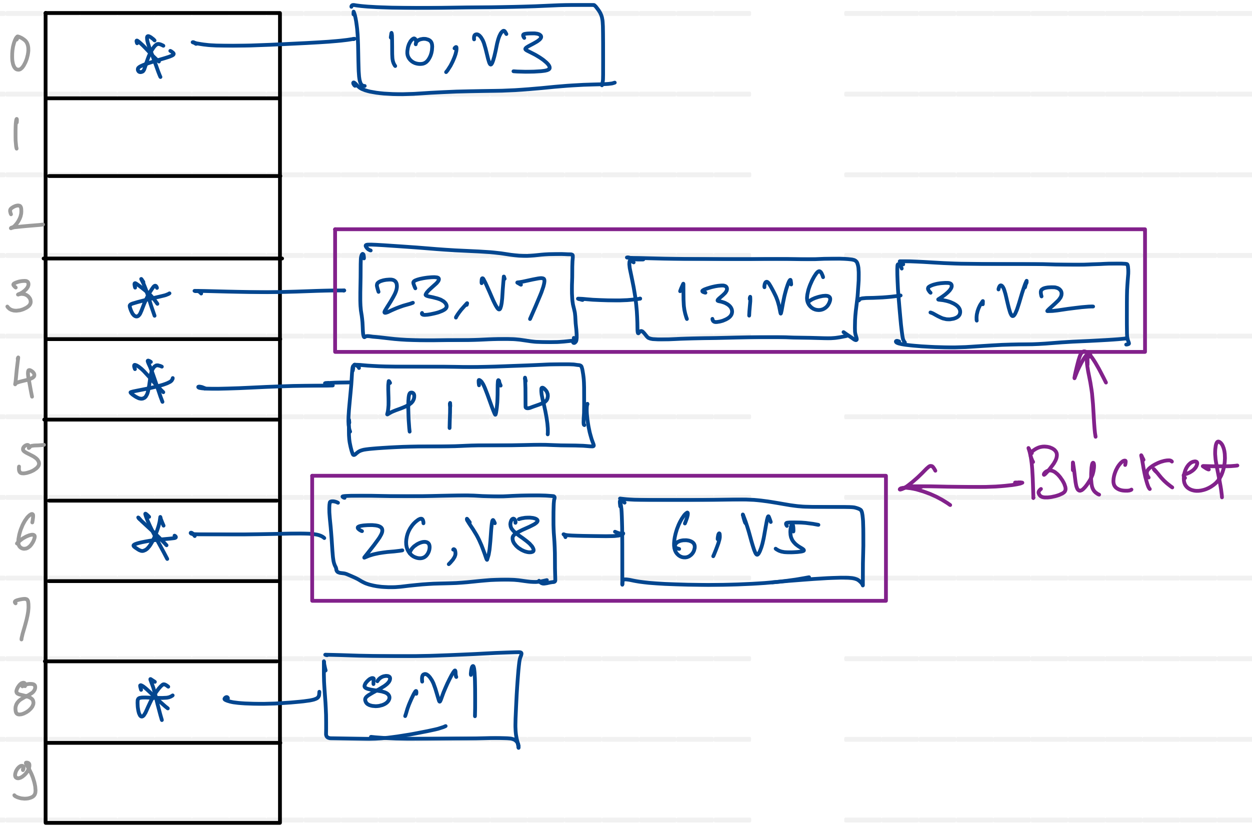
Closed Addressing / Chaining / Separate Chaining

└ linked list per slot

size=10

h(k) = k % size

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

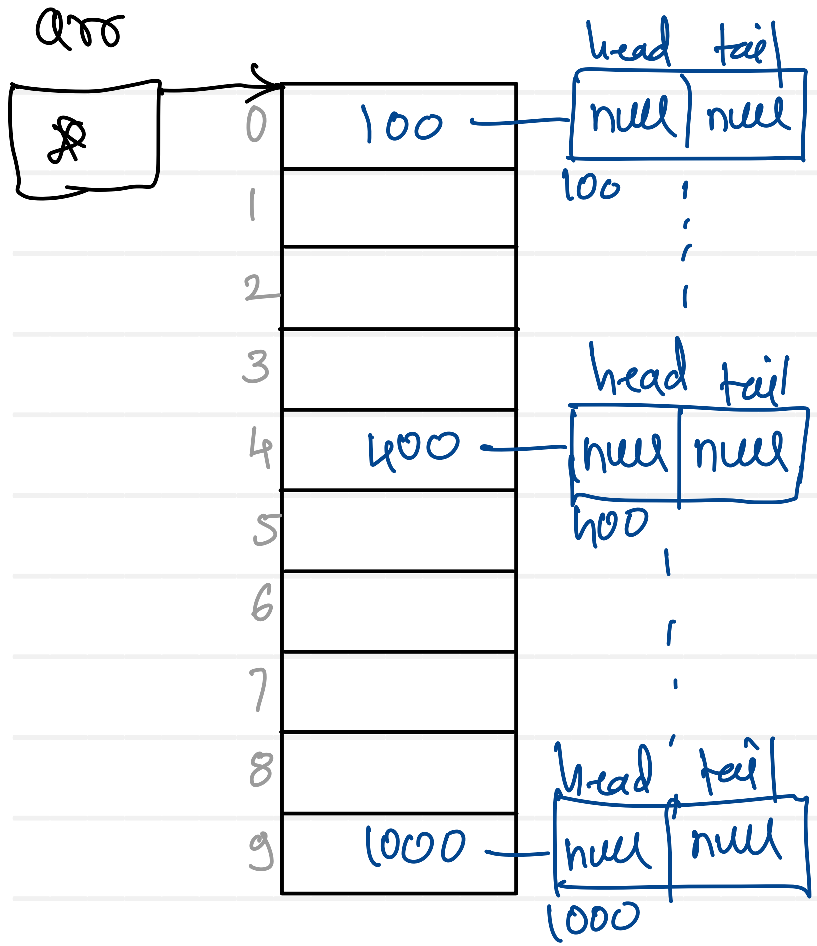


Hash Table

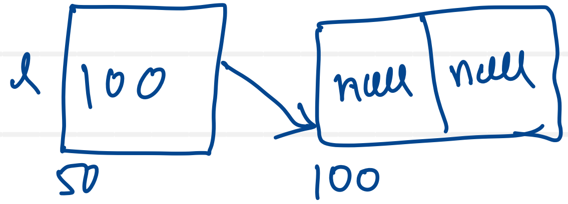
h(13) = 13 % 10 = 3
h(23) = 23 % 10 = 3
h(26) = 26 % 10 = 6

Advantage :
can store multiple key value pairs

Disadvantages :
i. key value pair are stored outside the table
ii. space requirement is more
iii. worst case time complexity is O(n).
(if maximum keys yeild same slot)



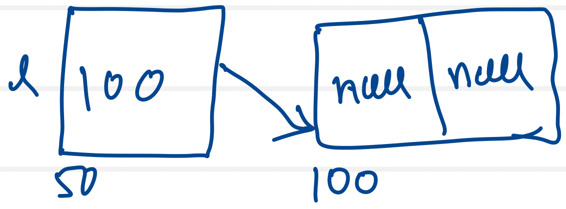

```
List< > l = new LinkedList<>();
```



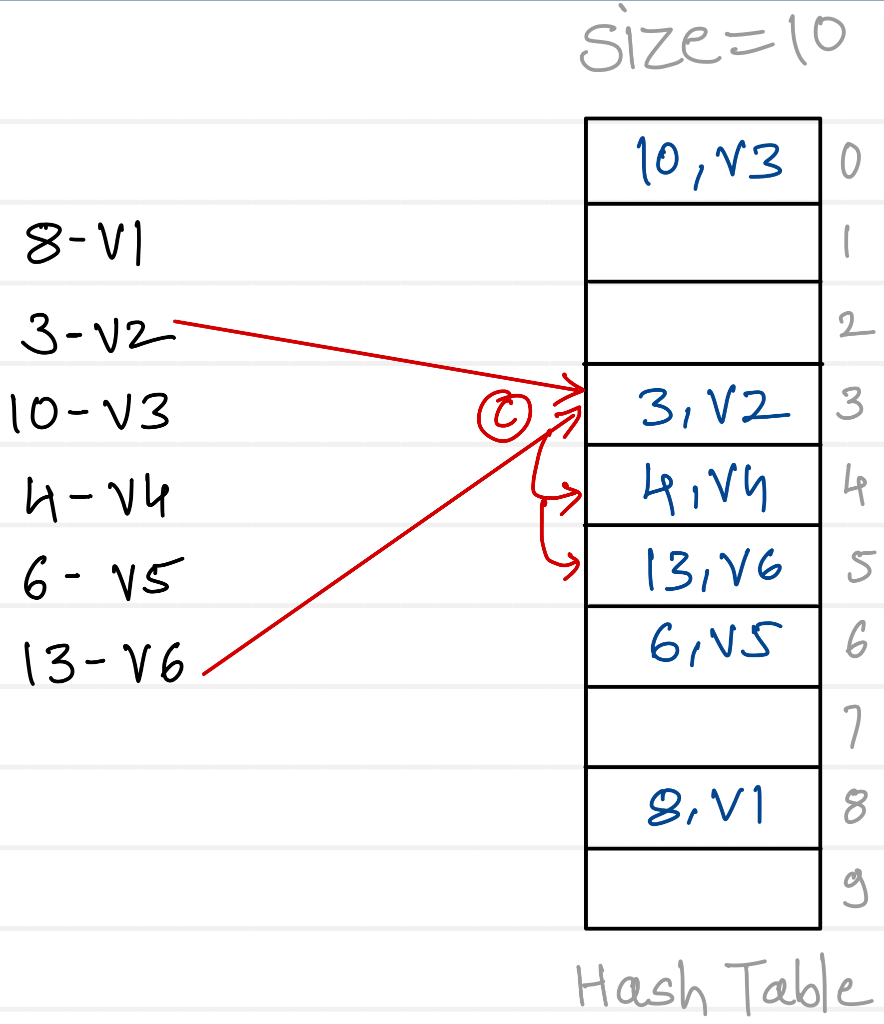

```
arr = new List[size];
for(int i = 0; i < size; i++)
    arr[i] = new LinkedList<>();
```

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

```
class List {
    Node head;
    Node tail;
}
```



Open addressing - Linear probing



h(k) = k % size

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

f(i) = i

where i = 1, 2, 3, … ← probe number

h(13) = 13 % 10 = 3 c

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

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

probing – finding next free slot whenever collision will occur into table.

Primary clustering – need to take long run of filled slots "near" key position to find empty slot.

Open addressing - Quadratic probing

size=10

8-V1
3-V2
10-V3
4-V4
6-V5
13-V6

0	10,V3
1	
2	
3	3,V2
4	4,V4
5	
6	6,V5
7	13,V6
8	8,V1
9	

Hash Table

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

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

$$f(i) = i^2$$

where i = 1, 2, 3, …

$h(13) = 13 \% 10 = 3$ C

$h(13,1) = [3 + 1] \% 10 = 4$ (1st) C

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

- primary clustering is removed.
- no guarantee of getting free slot for given key.

Open addressing - Quadratic probing

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

size=10

0	10, V3
1	
2	23, V7
3	3, V2
4	4, V4
5	
6	6, V5
7	13, V6
8	8, V1
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)

Secondary clustering :
need to take long run of filled slots "away" key position to find empty slot.

Open addressing - Double hashing

size=11

8-V1
3-V2
10-V3
25-V4

0	
1	
2	
3	3, V2
4	
5	
6	25, V4
7	
8	8, V1
9	
10	10, V3

Hash Table

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

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

$h1(8) = 8 \% 11 = 8$
$h1(3) = 3 \% 11 = 3$
$h_1(10) = 10 \% 11 = 10$
$h_1(25) = 25 \% 11 = 3$ c
$h2(25) = 7 - (25 \% 7) = 3$
$h(25,1) = [3 + 1 * 3] \% 11$
$= 6$ (1st probe)

k=36
$h_1(36) = 3$ c
$h_2(36) = 6$
$h(36,1) = 9$

- primary & secondary clustering is removed
- key value pairs are spreaded in table evenly.

Rehashing

Load factor = $\frac{n}{N}$
(λ)

$N = 10$
$n = 6$
$\lambda = \frac{n}{N} = \frac{6}{10} = 0.6$
Hash table is 60 % full (occupied)

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