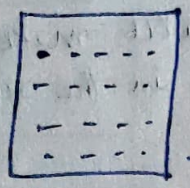
Hash Map:

Why data Structures?

→ To store input for algorithms and to provide output from algorithms.

Operations:

a) Insertion b) Deletion c) Search.



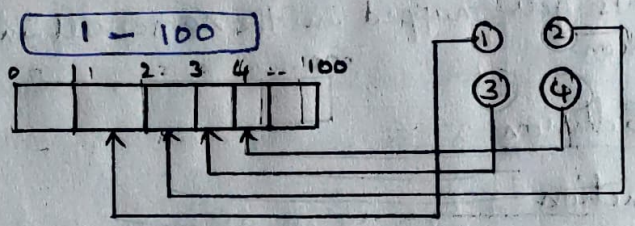
Insertion and Deletion rarely happen as widely.

→ Search is widely used.

- → Unsorted Array O(n)
- → Sorted Array O(nlogn)
- → Linked List O(n)
- → Binary Tree O(n)
- → B.S.T O(n)
- → Balanced B.S.T O(logn)
- → Priority Queue O(n)

→ The best Search complexity till now is O(logn).

D.A.T → Direct Access Table.



a) Insert 6
Go to 6th index and add
O(1)

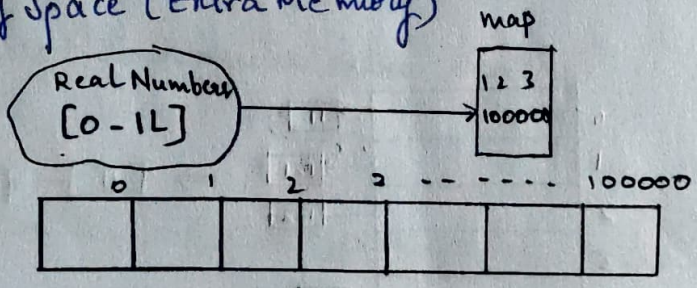
b) Deletion → 6.
O(1)

Advantage:

→ Every operation can be accomplished in O(1)

Disadvantage:

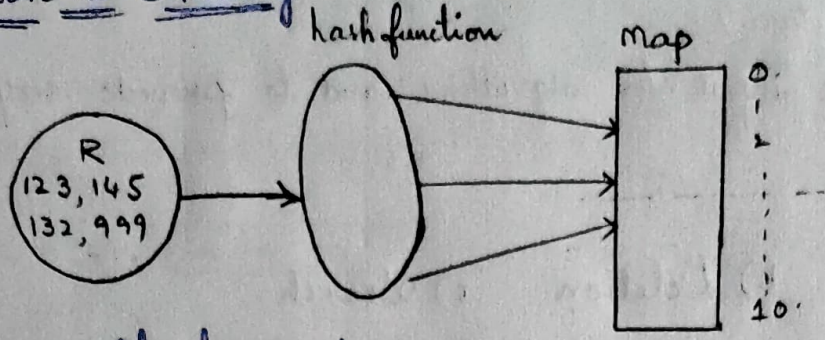
Memory Space (Extra Memory)



→ You want to insert 100000 value then [1, 2, 3, 100000] it is not the best approach, because memory is getting wasted. as we want to insert only 4 values out of 1L size.

→ For small values, D.A.T is good

Introduction to Hashing:



→ We have a pool of numbers, we will make the numbers pass through a function such that the numbers in the real number pool map to some value inside the machine.

hash function ⇒ % 10

121 % 10 ⇒ 1 132 % 10 ⇒ 2

145 % 10 ⇒ 5 999 % 10 ⇒ 9

0	1	2	3	4	5	6	7	8	9
	121	132			145				999

Insertion → O(1), Search → O(1), Deletion → O(1)

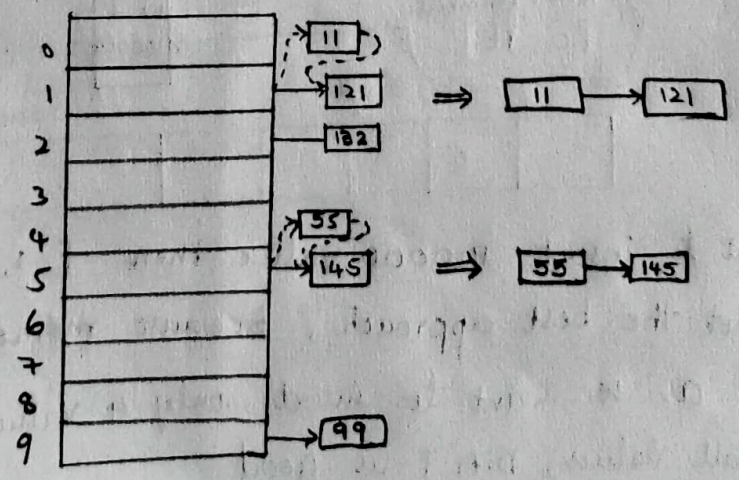
21 % 10 → ①, we get collision.

Collision:
When 2 elements gets mapped in the same cell in two situations is known as collision.

Ways to Handle Collision:

1. Better Hash function
   % 10, Prime number
   → we can decrease the collision but not zero.
2. Chaining
3. Open Addressing a) Linear Probing b) Quadratic Probing c) Double Hashing

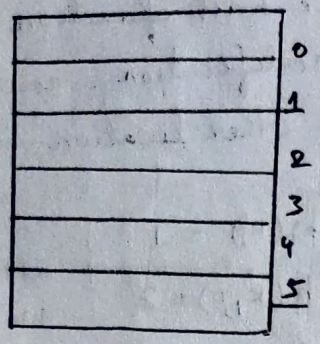
Chaining:



Note: In case of collision, go to that particular index, and make the first node of linked list as the element.

→ Insertion takes O(1)

Search:



5, 15, 25, 35, 45

— — — — —

T(n) = O(n)

Load factor:

$\frac{n}{m}$ → elements in the list

$\frac{n}{m}$ → size of the hash table.

Average Search Time:

$\theta\left(1+\frac{n}{m}\right)$ → load factor

$m = 6, \quad n = 10 \Rightarrow \frac{n}{m} = \frac{10}{6} = 1.6$

L.f:

The value after that the size of the hash map doubles.

Usually the load factor is 75%

i.e once the size reaches 75%, the hash map doubles.

$\frac{6 \times 75}{100} = 4.5$, once it reaches 5, it doubles the size to 12.

Advantage:

most operations in O(1), Collision is taken care of (insertion)

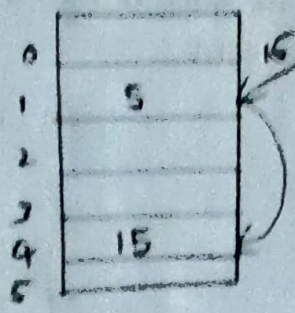
Disadvantage:

need to consider extra space.

Open Addressing:

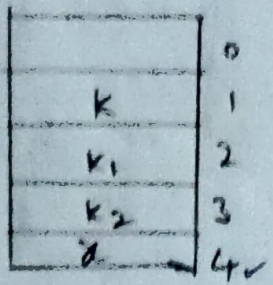
→ It handles the collision without using extra space.

l.f → $0 \leq$ L.f ≤ 1 $\quad \alpha = \frac{n}{m} = \frac{n}{n} = 1$



→ Is already available at that location, 15 cannot be inserted, we would apply the hash function with certain modification, now 15 will be inserted at other location

ex (1)



$h(k) = 1$

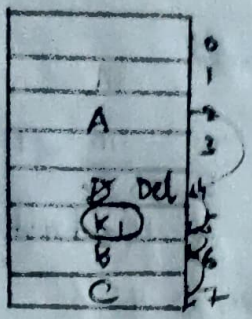
$h(k_1, 0) = 2$ → after applying hash function we are getting index '2'

$h(k_2, 0) = 4$ → collision as 'x' is already present

$h(k_2, 1) = 3$

→ collision Number

ex (2)



$h(k_1, 0) \Rightarrow$ '2' → but it is already occupied by 'A'

$h(k_1, 1) \rightarrow 6$ → This is also occupied

$h(k_1, 2) \rightarrow 7 \rightarrow 11$

$h(k_1, 3) \rightarrow 4 \rightarrow 4$

$h(k_1, 4) \rightarrow 5$

Search:

→ we need to search for k_1, again follow the same hashing function steps and check in that index if the value is equal to k_1, else recompute the hashfunction.

→ Let's say some one deleted 'D' at 4th index and empty value is present. It is understood that it is not gone to that location. So, its better to write "Del" after deleting the element. and it will move further to find k_1.

→ $TC = O(n)$

→ The hash function should be written in such a way that elements occupy uniform slots.

→ So that collison problem decreases and the search operation comes out to $O(1)$.

Linear Probing:

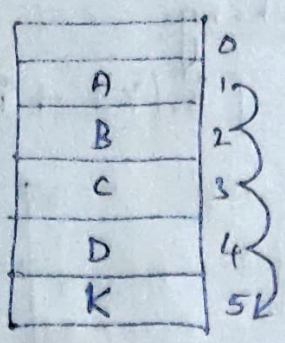
collision number

→ $h'(k,i) \rightarrow (h(k) + i) \% m$

$h(k,1) \rightarrow (h(k)+1) \% m$

$h(k,0) = 1$

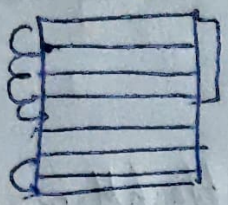
$h(k,1) = 1+1 \rightarrow 2$



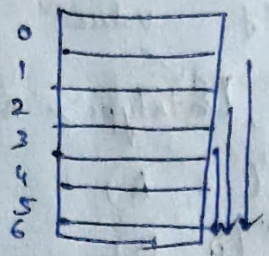
Probing → finding the next location.

→ here it is probing in a linear manner.

Primary Clustering:



⇒ Blocks will get created



It will form blocks the probability of new collison might increase.

Secondary Clustering:

$h(k_1) \Rightarrow 1$

$h(k_1,1) \Rightarrow 2$

$h(k_1,2) \Rightarrow 3$

$h(k_1,3) \Rightarrow 4$

$h(k_2) \Rightarrow 1$

$h(k_2,1) \Rightarrow 2$

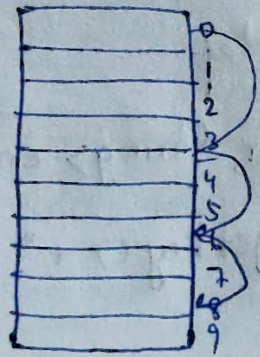
$h(k_2,2) \Rightarrow 3$

$h(k_3,3) \Rightarrow 4$

→ If 2 numbers map to the same number initially then both the numbers follows the same prob sequence.

Quadratic Probing:

$h(k,i) = ((h(k) + (i^2 k + i)) \% m$



$h(k,0) \Rightarrow 0$

$h(k,1) \Rightarrow 4$

$h(k,2) \Rightarrow 6$

$h(k,3) \Rightarrow 8$

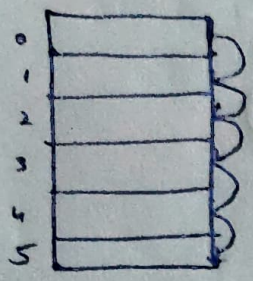
Probed Index is calculated in a Quadratic manner

→ It suffers from Secondary clustering, but not from Primary clustering.

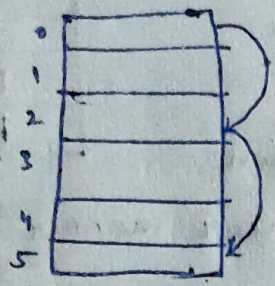
Double Hashing:

$$h(k) = [h_1(k) + i\,h_2(k)] \% n.$$

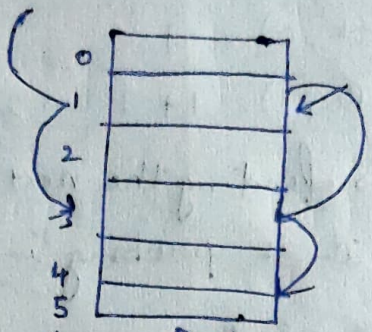
$y = y(mx) + y(mz)$



LP



QP



DH

~~So~~ no Secondary clustering

Syntax & Methods:

key, value.

* `HashMap<DataType, DataType> map = new HashMap<>();`

```
map.put(1, A);
map.put(3, C);
map.put(2, B);
```

→

1	A
3	C
2	B

→ It is an unordered map.

* `LinkedHashMap<Integer, String> map = new LinkedHashMap<>();` (for maintaining the order)

```
map.put(1, A)
map.put(2, B)
map.put(3, C)
```

→

1	A
2	B
3	C

Same order of insertion

* For MultiThreaded Env, where update & Read operations happens simultaneously. Then ConcurrentHashMap should be used.

→ To iterate over a hashMap.

```
for (Map.Entry<Integer, String> entry : map.entrySet()){
    S.O.println(entry.getKey() + " " + entry.getValue());
}
```

Two Sum

[2, 7, 11, 15] Target = 9 , o/p: [0, 1]

[3, 2, 4] Target = 6 , o/p: [1, 2]

→ Algo:

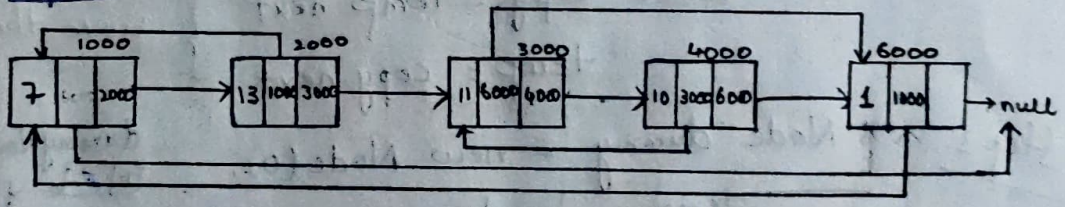
1. check if target - arr[i] is available in the map, If yes return the indices of num's index, i & i

2. If not, add the number arr[i] and its index to the map.

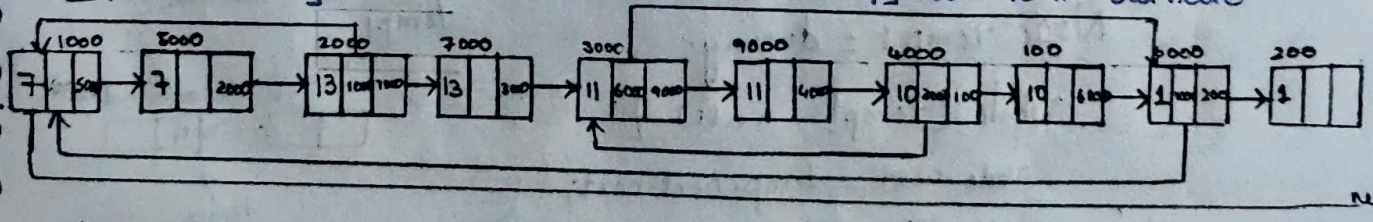
```
Map<Integer, Integer> map = new HashMap<>();
for(int i=0; i<arr.length; i++){
    int num = target - arr[i];
    if(map.containskey(num)){
        return new int[]{ map.get(num), i};
    }
    else{
        map.put(num[i], i);
    }
}
return new int[]{};
```

Copy List with Random Pointer:

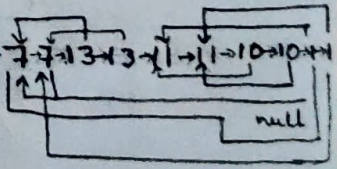
Without Extra Space:



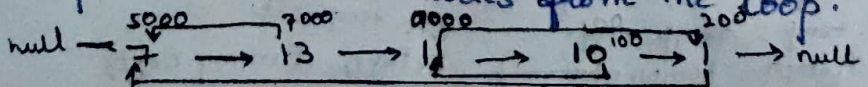
Step 1: Creating new nodes and point the copy node to the old node.



Step 2: For the new nodes, point to the random nodes.

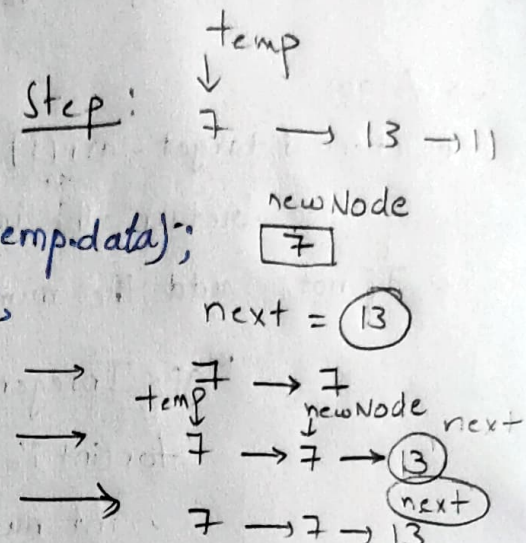


Step 3: Separate all the new nodes from the loop.

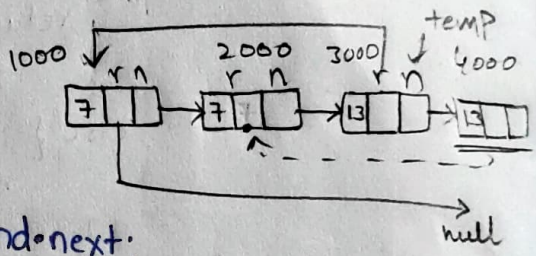


```
Node copyList(Node head){
    if(head == null){
        return null;
    }
    Node temp = head;
    while(temp != null){
        Node newNode = new Node(temp.data);
        Node next    = temp.next;
        temp.next    = newNode;
        newNode.next = next;
        temp         = next;
    }
    // Pointing the random pointer
    temp = head;
    while(temp != head){
        if(temp.rand != null){
            temp.next.rand = temp.rand.next;
```

Step1:



Step2:



(13).□ = 13□.next

= 1000.next

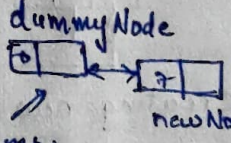
= 2000

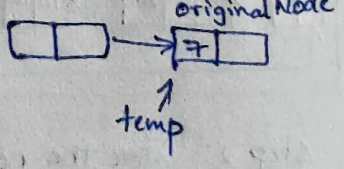
```
        }
        Node copy = temp.next;
        temp = copy.next;
    }
```

} move the temp to copy's next

Step3:

```
Node dummy = new Node(0);
temp = head;
Node temp1 = dummy;
while(temp != null){
    Node next = temp.next.next;
    Node copy = temp.next;
    temp1.next = copy;
    temp.next = next;
    temp = temp.next;
    temp1 = temp1.next;
}
return dummy.next;
```

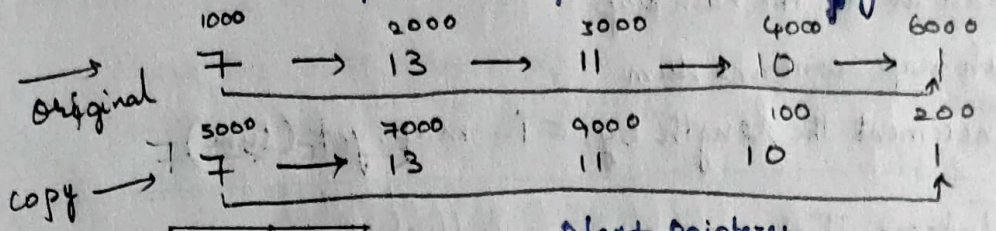




(13) (7) (7) So temp = 13 So temp1 = 7

With Extra Space:

1) Create a copy of each node.
2) Store the original node as key and copy node as value.
3) create next & random pointers for copy nodes.



key	value
1000	5000
2000	7000
3000	9000
4000	100
6000	200

Next pointers:

map.get(temp).next = map.get(temp.next)

map.get(1000).next = map.get(1000.next)

5000.next = map.get(2000)

= 7000

Random Pointers:

map.get(temp).rand = map.get(temp.rand)

map.get(1000).rand = map.get(1000.rand)

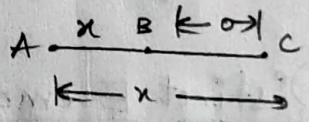
5000.rand = map.get(6000)

= 200

* Longest SubArray With Sum Zero:

[15, -2, 2, -8, 1, 7, 10, 23]

0 1 2 3 4 5 6 7



key	value
0	-1
15	0
13	1
7	3
8	4
~~15~~	~~5~~
25	6
48	7

← at start

At index 3 (15-8) →

At index 6 (15+10) →

At index 2 ⇒ Sum ⇒ 13+2 = 15

length = i - map.get(Sum)

= 2 - map.get(15)

= 2 - 0 = 2

At index 5 ⇒ Sum = 15

length = i - map.get(sum)

= 5 - 0

= 5 ✓

So longest SubArray = 5

Algo:

→ Create a hashmap to hold → SumUpTillNow → (occurence of that sum)

→ Iterate over the array,

If hash map does not contain the sum
Add it to the hash map.

If hash map contains sum.
Increment the length by = i - map.get(sum)

```
Map<Integer, Integer> map = new HashMap<>();
int length=0, sum=0;
map.put(0, -1);
for(int i=0; i< arr.length; i++){
    sum += arr[i];
    if ( map.contains(sum)) {
        length = i - map.get(sum);
    }
    else {
        map.put(sum, i);
    }
}
return length;
```

** Distinct Elements in Window of Size k:

[2 5 5 6 3 2 3], k=4

o/p: [3, 3, 4, 3]

→ Sliding Window

→ HashMap (Element ⇒ Count)

At first Window arr[i] = 2

2 → 1
5 → 2
6 → 1

→ hashmap size → (3)

→ check the count of arr[i] in the map, because we will be discarding this element and moving to the next window, so we should decrement the count.

a) If arr[i] count == 1 then remove the entry

b) If arr[i] count > 1, then decrement.

At 2nd window

5 5 6 3

5 → 2 (no update in value)
6 → 1 (no update in value)
3 → 1

map size → (3)

a) arr[i] = 5

arr[i] count > 1

So decrement count.

So map will be

5 → 1
6 → 1
3 → 1

At 3rd window:

5, 6, 3, 2

5 → 1
6 → 1
3 → 1
2 → 1

map size → 4

After decrementing arr[i]'s count map will be

6 → 1
3 → 1
2 → 1

At 4th window

6, 3, 2, 3

6 → 1
3 → 2
2 → 1

map size → (3)

Algo:

1) Initialize 2 pointers i, j = 0
2) Iterate the array until j < arr.length
3) Add the elements & its count in the map
4) If the window size is Reached get the hash map size.
5) Remove the entry of index start if count is 1
6) decrement the value of entry if count is > 1.

```
Map<Integer, Integer> map = new HashMap<>();
ArrayList<Integer> list = new ArrayList<>();
    int i=0;  int j=0;
      while(j < n){
        if( !map.contains(A[j]){
          map.put(A[j], 1);
        }
       else{
          map.put (A[j], map.get(A[j])+1);
        }
      if(j-i+1 == K){
        list.add (map.size());
       if(map.get(A[i]) == 1){
        map.remove[A[i]);
       } else{
          map.put (A[i])map.get(A[i])-1);
        }
       i++;
      }
      j++;
  }
```

Group Anagrams :- ["eat", "tea", "tan", "ate", "nat", "bat"]
1 2 3 4 5 6

→ Based on Key, we need to group the strings.

→ Before grouping, we need to check if the key is available in the map.

→ Key in the Map is a "map"<character, Integer>

1. eat → frequencyMap e→1, a→1, t→1
4. ate a→1, t→1, e→1
2. tea → frequency map t→1, e→1, a→1 → This map is already available add it to the list

[eat, tea, ate], [bat]

3. tan [tan, nat] t→1, a→1, n→1 → not available in map, so create a new list

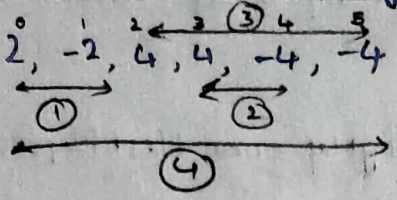
```
Map< Map<Character, Integer>, List<String> > map = new HashMap<>();
    for (String str : strs) {
Map<Character, Integer> frequencyMap = new HashMap<>();
for (int i=0; i < str.length(); i++){
    char ch = str.charAt(i);
    if (!frequencyMap.containsKey(ch)){
        frequencyMap.put(ch, 1);
    }
    else {
      frequencyMap.put(ch, frequencyMap.get(ch)+1);
    }
  }
  if (!map.containsKey(frequencyMap)){
    List<String> list = new ArrayList<>();
        list.add(str);
       map.put(frequencyMap, str);
    }
    else {
    List<String> list=map.get(frequencyMap);
        list.add(str);
    }
}
List<List<String>> result= new ArrayList<>();
for( List<String> values: map.values() ){
        result.add(values);
  }
return result;
```

map

1) e-1 a-1 t-1 [eat, tea, ate]
2) t-1 a-1 [tan, nat]
3) b-1 a-1 t-1 [bat]

[[eat, tea, ate], [tan, nat], [bat]]

Count of Zero Sum SubArray:



map

0	→ ✗ 2
2	→ 1
4	→ 1
8	→ 1

ans = 0

At index ⓪:- sum = 2
2 is not available in map.
ans = ⓪

At index ①:- sum = 2 + arr[i]
= 2 - 2 = 0
'0' is available in map.
ans += map.get(0) ⇒ ans += 1
ans = ① , increment the value of 'i' as we found '0' again

At index ②:- sum = 0 + 4 = 4, 4 is not available in map.
add (4, 1) in map.

At index ③:- sum = 4 + 4 = 8, 8 is not available in map
add (8, 1) in map.

At index ④:- sum = 8 - 4 = 4, 4 is available in map.
ans += map.get(4)
ans = 1 + 1 = ②

At index ⑤:- sum = 4 - 4 = 0 is available in map.
ans += map.get(0)
ans = 2 + 2 = ④

```
int ans = 0;
Map<Integer, Integer> map = new HashMap<>();
int sum = 0;
map.put(0,1);                    for(int i = 0; i < arr.length; i++) {
                                     sum += arr[i];
if(!map.contains(Key(sum))){
    map.put(sum, i);
}
else {
    ans += map.get(sum);
    map.put(sum, map.get(sum)+1);
}
}
return ans;
```