Given an array of size N and Q queries. In each query, an element is given, check whether its present in array.

Q = 3

A[] = <2, 4, 11, 15, 6, 8, 14, 9>

K	
4	True
14	True
19	False

Brute Force: For every query, loop through array to check presence.

TC: O(N*Q) SC: O(1)

Optimized: Create an array and mark presence of an element against that particular idx.

A[] = <2, 4, 11, 15, 6, 8, 14, 9>

max idx = 15 new ar. size → 16

int dat[16] = <0> Direct Access Table

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	ϕ 1	0	ϕ 1	0	ϕ 1	0	ϕ 1	ϕ 1	0	ϕ 1	0	0	ϕ 1	ϕ 1

```
for (i=0 ; i<N ; i++)
        dat [ arr[i] ] =1
```

Advantages of DAT

TC of insertion, deletion and search - O(1)

Issues with such representation

1. Space wastage

A[] = <23, 60, 37, 90>

int dat[91]

Only 4 elements are marked out of 91

2. dat [max ele +1]

A[] = $<10^9, 10^{10}, 10^{20}>$

Max dat or any [] → size → $10^5 - 10^6$

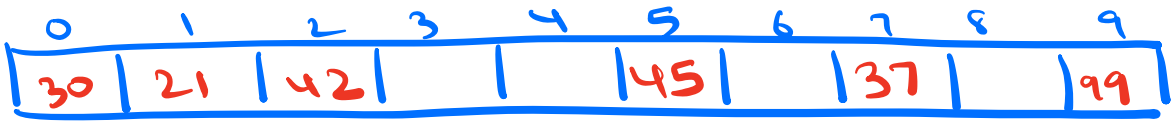
3. Storing values other than positive integers

How to overcome issues provided we retain advantages?

Let's say we have restriction of creating only array of size 10

A[ ] = <21, 42, 37, 45, 99, 30>

How to mark presence of elements?



Element $\xrightarrow{\text{Map}}$ Idx

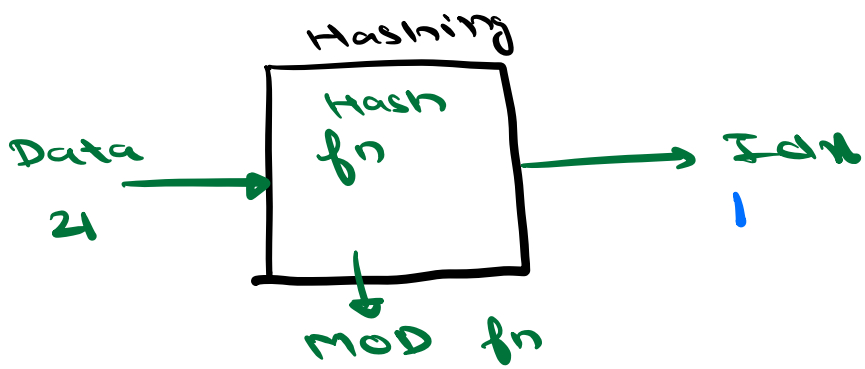
ele	idx
21	% 10 → 1
42	% 10 → 2
37	% 10 → 7
45	% 10 → 5
99	% 10 → 9
30	% 10 → 0

number % 10 → [0 … 9]

21 (Info)

↓

1 (Idx / Hash value)



- Hashing is a process where we pass our data through Hash function which gives us hash value (index) to map our data to.
- In this case, hash function is MOD. Usually, more complex hash functions are used
- DAT → Hash Table

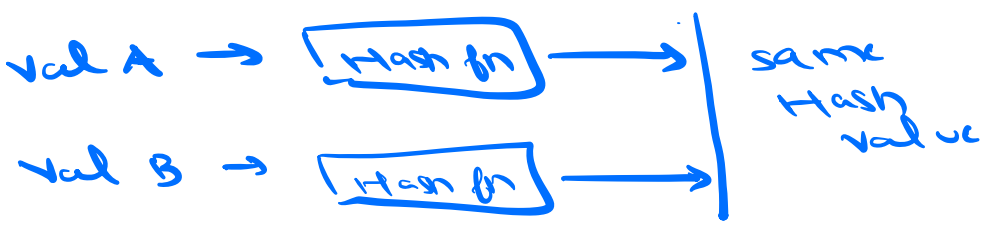
Issues with Hashing

A[ ] = <21, 42, 37, 45, 99, 30>

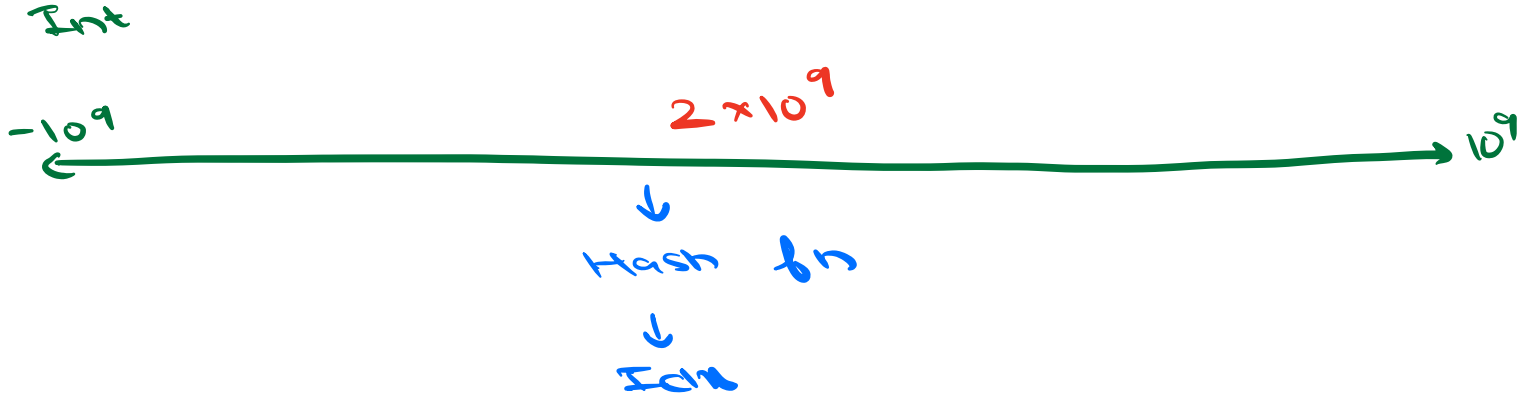
data		idx		data		idx
21	→	1		37	→	7
31	→	1		77	→	7

(idx: %10)

Collision → when 2 diff. values map to same hash value



Can we avoid collision completely?



Pigeonhole principle

5 pigeons

5

Reduce collision → increase size of hash table

de
x → id
2

y →

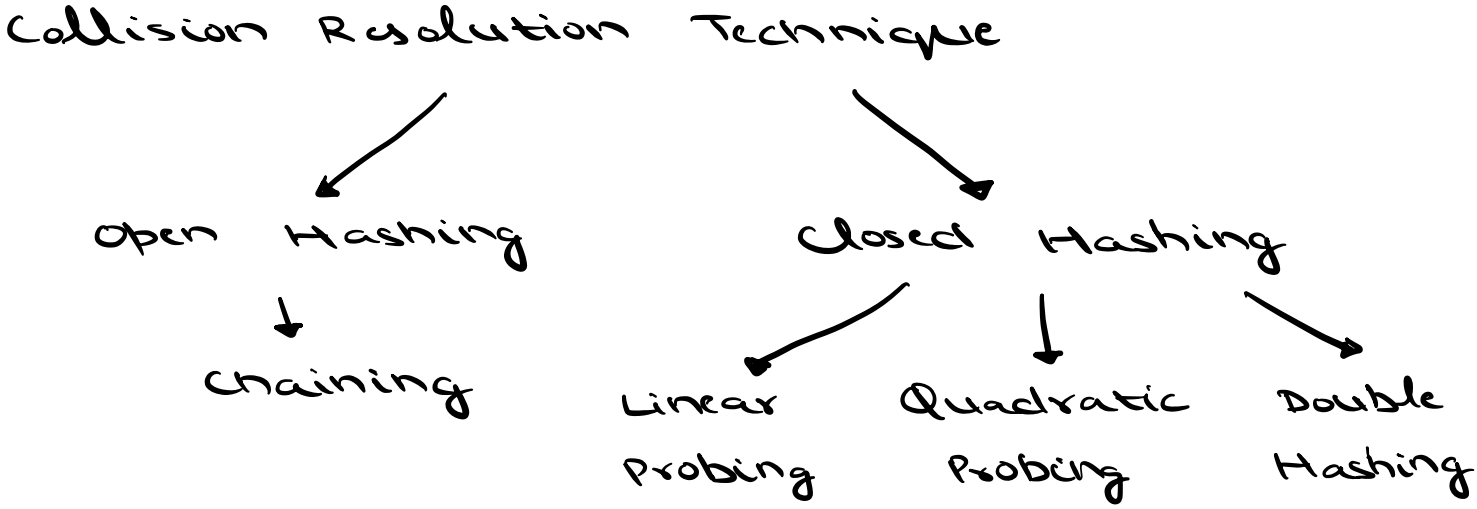
y → 2 $= \frac{1}{10} = 0.1$

10%.

de
x id
2

y

$\frac{1}{20} = 5\%$.

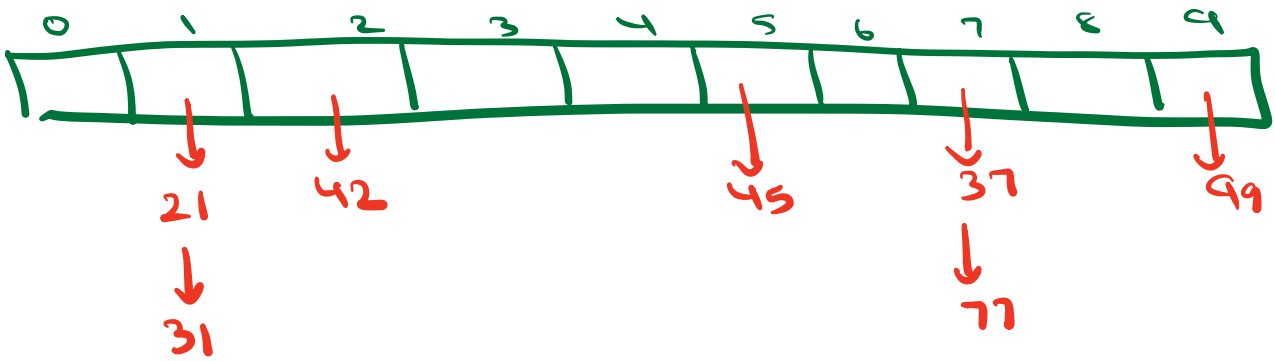


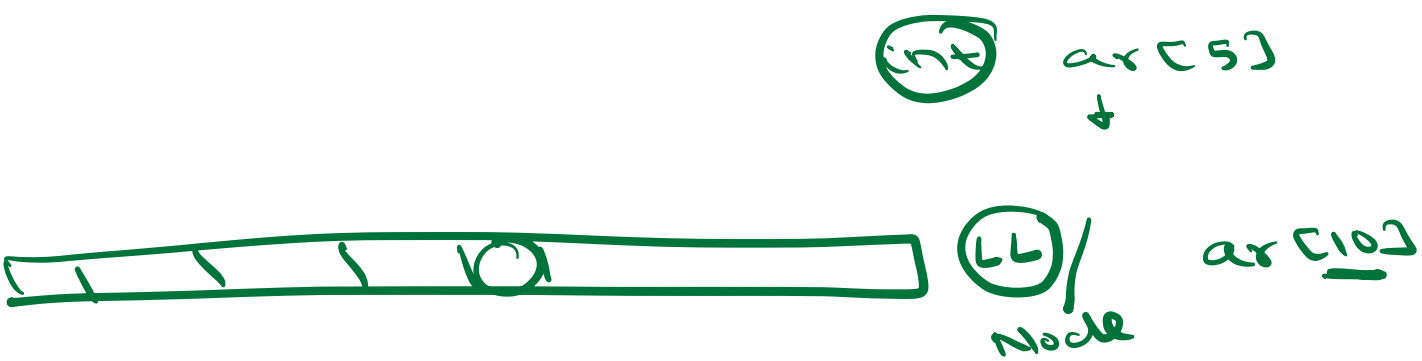
Chaining → LL at every idx

A[ ] = <21, 42, 37, 45, 99, 31, 77>

21, 31, 41, 51, 61, 71, 81, 91

Hash table → 10





Chaining is a technique used in data structures, particularly hash tables, to resolve collisions. When multiple items hash to the same index, chaining stores them in a Linked List or another data structure at the index.

TC of insertion

1. Pass given element to Hash function, it'll return an index.
2. Add element to Linked List at that index.

Insert at head → O(1)

Insert at tail → O(N)

TC of Deletion and Searching

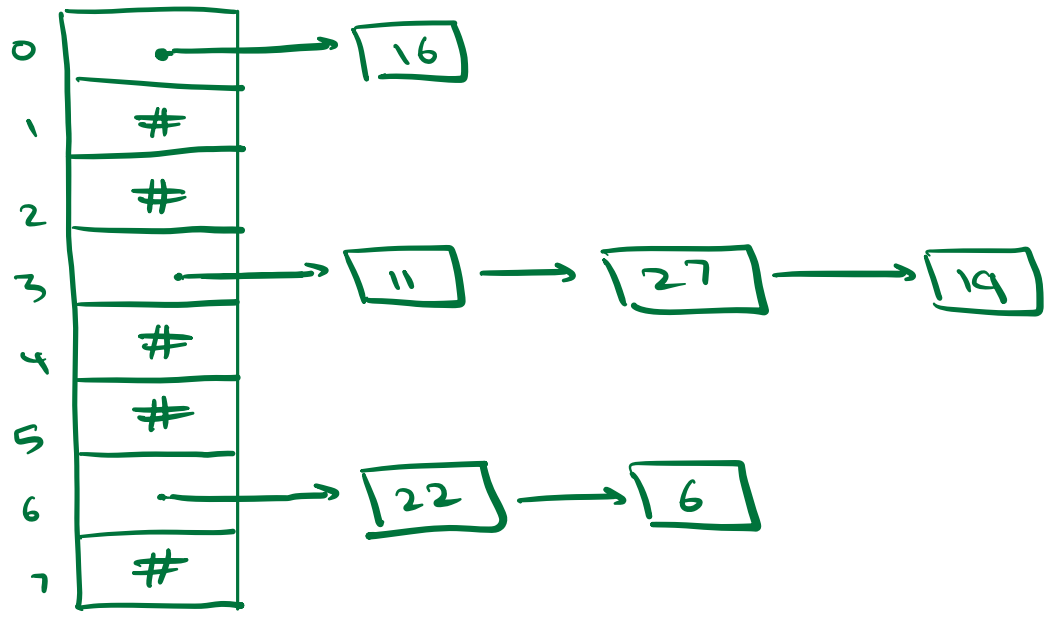
↓ O(N)

↓ O(N)

TC on avg. is less than <= ↗

What is lambda (λ)

$$\lambda = \frac{\text{no. of elements inserted}}{\text{size of array}}$$



size of array → 8

No of elements → 6

$$\lambda = \frac{6}{8} = 0.75$$

8 idx → 6 elements

1 idx → 0.75 de

0 → O O — O(2)

1 → O O — O(2)

2 → O O — O(2)

3 → O O

4 → O O

5 → O O

6 → O O

7 → O O

array size → 8

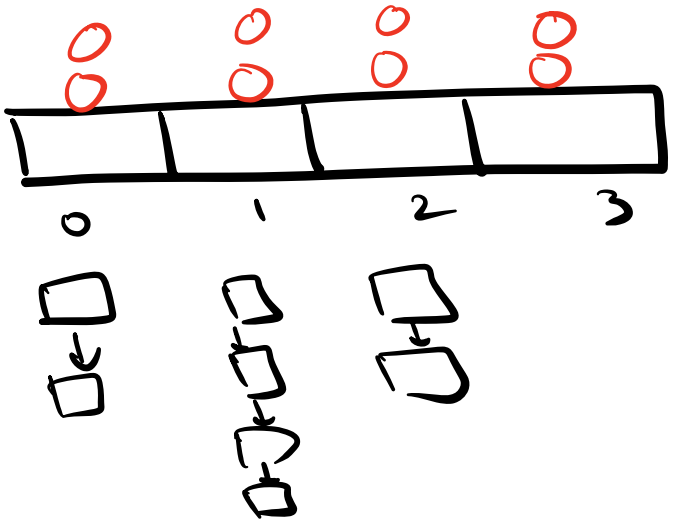
No. of elements → 16

$$\lambda = \frac{16}{8} = 2$$

@ idx → 16 de

1 idx → 2 de (avg)

λ → At an idx, avg no. of elements



$$\lambda = \frac{8}{4} = 2$$

$$\lambda = \frac{9}{4} > 2$$

size of array → 8

Threshold = 2

Total elem = 10

$\lambda = \frac{10}{8} = 1.25$

0 → 1.75 elements

1

2

elem = 14

$\lambda = \frac{14}{8} = 1.75$

elements = 16

$\lambda = \frac{16}{8} = 2$

elements → 17th

$\lambda = \frac{17}{8} = 2.125$ → Threshold 2

HT → 8

New HT

↓

16

Let's say predefined threshold is 0.7, if load factors exceeds this value, then we rehash the table.

Rehashing

1. Create new hash table with double size of original hash table.
2. Redistribute existing elements to their new positions in hash table using new hash function.
3. $\lambda_{new} = \frac{17}{16} = 1. \_\_\_ \leq 2$ (Given Threshold)

10:27

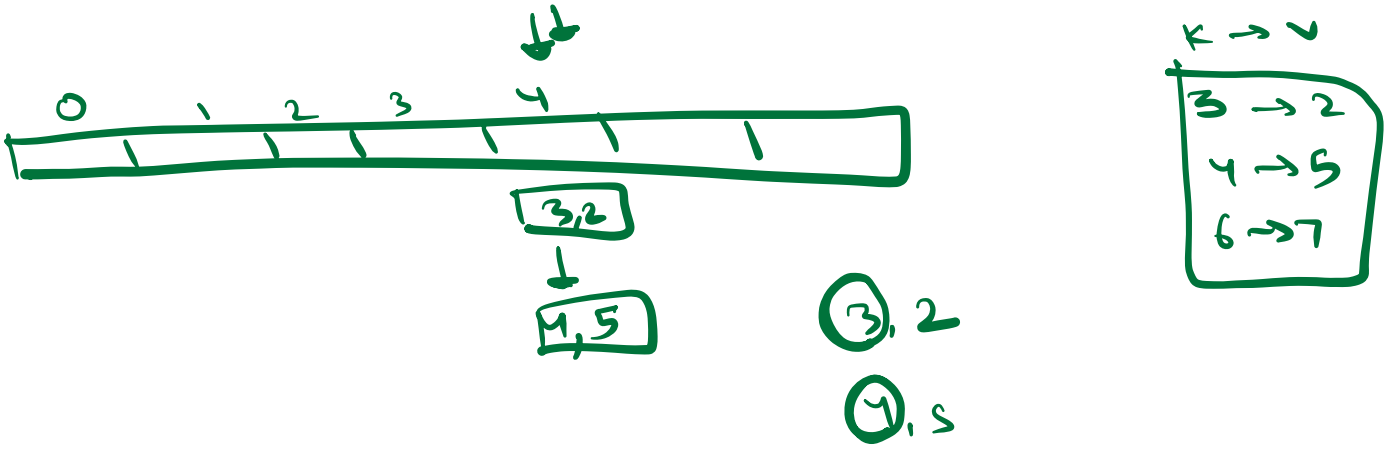
Code Implementation

ArrayList <Integer>
<String>

```
class ArrayList <T> {
    T get(int idx) {
    }
}
```

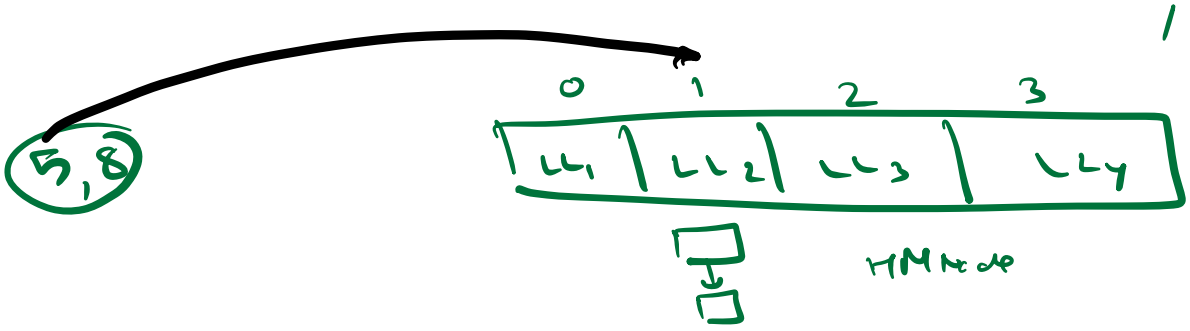
HashMap <Integer, String> hm = new

```
class HashMap <K, V> {
    private class HMNode {
        K key
        V value
        public HMNode(K keys, V values) {
            key = keys    value = values
        }
    }
    private LinkedList <HMNode> [] buckets
    private int size
}
```



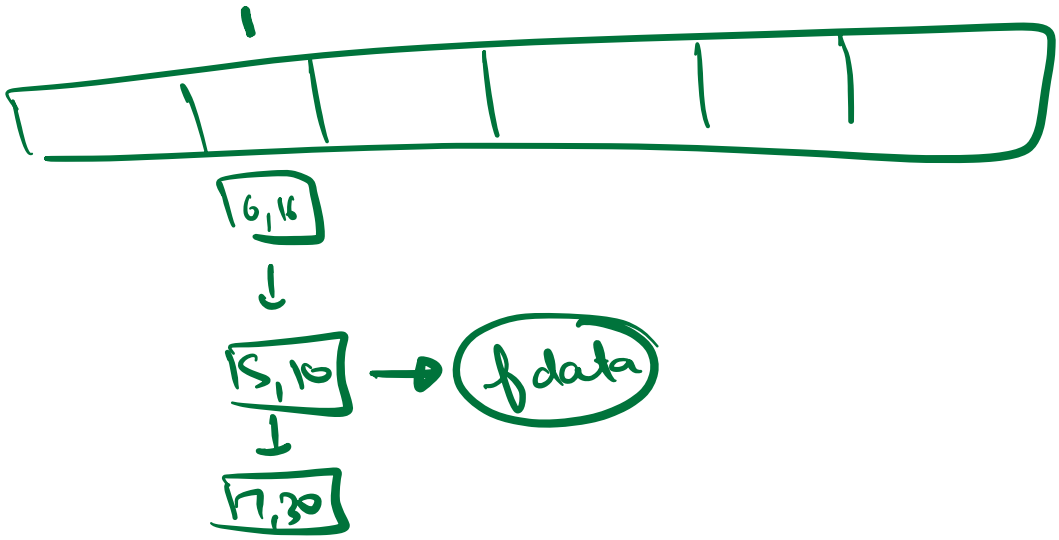

```
public Hashmap() {
    buckets = new LinkedList[4]
    for (i=0 ; i < buckets.len ; i++) {
        buckets[i] = new LinkedList<>()
    }
}
```

Array of LL




```
public void put (K key, V value) {
    int idx = hashfunction(key)
    HMNode fdata = foundat(buckets[idx], key)

    if (fdata == NULL) {
        HMNode hm = new HMNode()
        hm.key = key
        hm.value = value
        buckets[idx].addLast(hm)
        size ++
    }
    else {
        fdata.value = value
    }
    double lambda = size * 1.0 / buckets.len
    if (lambda > 2.0)
        rehash()
}
```



8

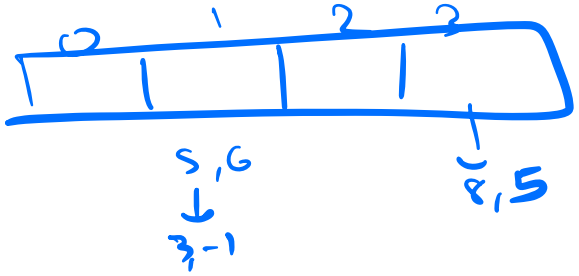
```
public V get (K key) {
    int idx = hashfunction(key)
    HMNode fdata = foundat ( buckets[idx], key)
    if (fdata == NULL) {
        return NULL
    }
    else {
        return fdata.value
    }
}
```

```
public V remove (K key) {
    int idx = hashfunction (key)
    HMNode fdata = found at ( buckets [idx], key)
    if (fdata == NULL) {
        return NULL
    }
    else {
        val = fdata.value
        buckets [idx] . remove (fdata)
        size --
    }
    return val
}
```

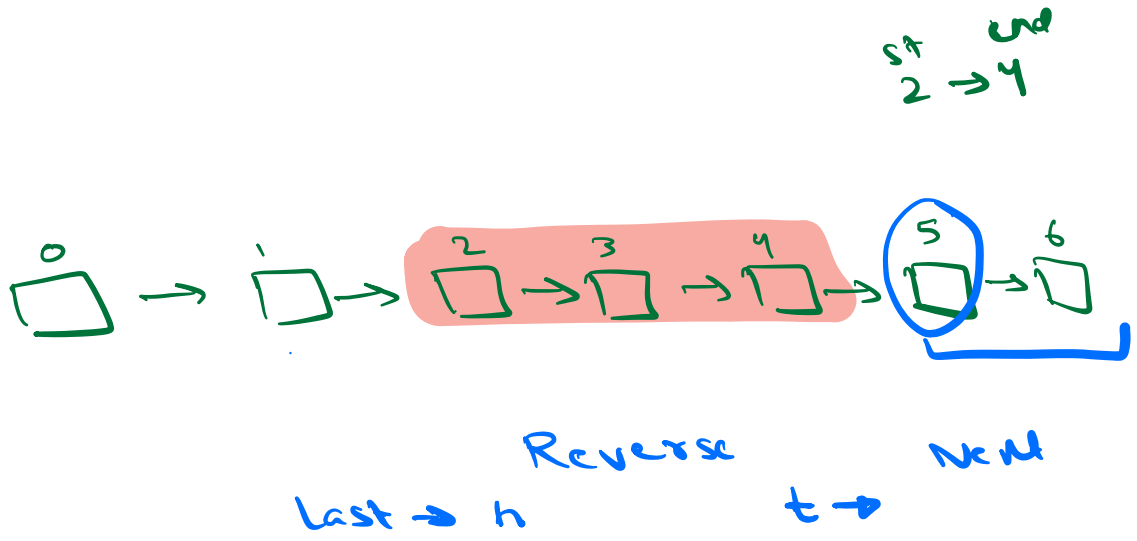
```
public boolean containsKey (K key) {
    int idx = hashfunction (key)
    HMNode fdata = found at ( buckets [idx], key)
    if (fdata == NULL)
        return false
    else
        return true
}
```

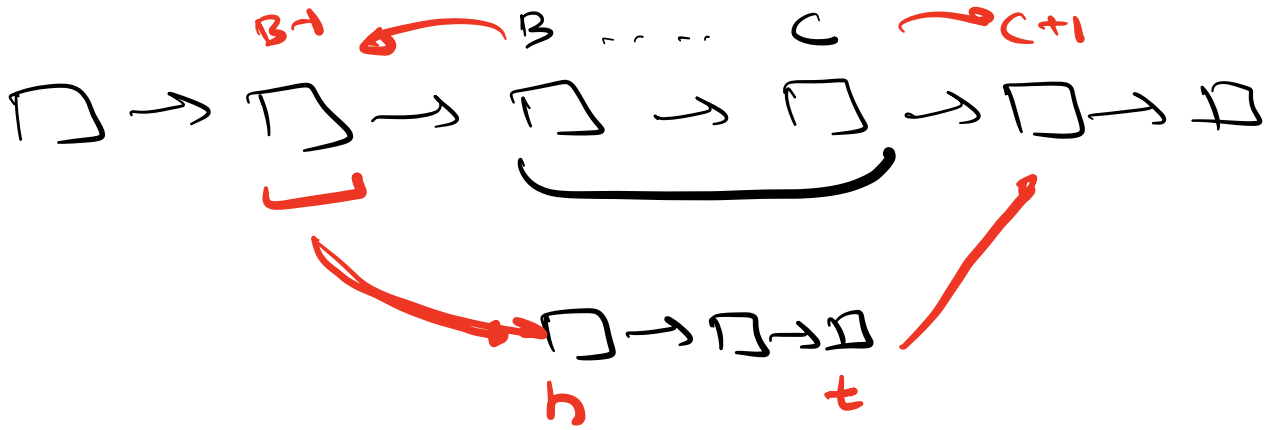
```
private int hashfunction(K key){
    int idx = key.hashCode()
    idx = Maths.abs(idx) % buckets.len
    return idx
}
```

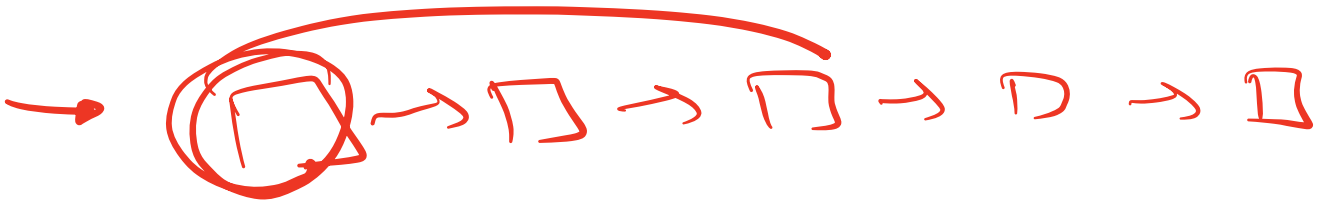
```
private HMNode foundAt
        (LinkedList<HMNode> list, K key){
    for (i=0 ; i < list.size() ; i++){
        HMNode temp = list.get(i)
        if (temp.key == key)
            return temp
    }
    return NULL
}
```



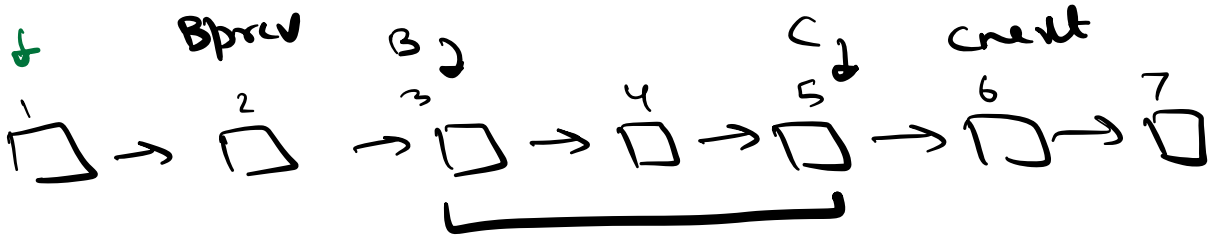
```
private void rehash() {
    LinkedList<HMNode>[] oldbuckets = buckets;
    buckets = new LinkedList[2 * oldbuckets.len];
    for (i=0; i < buckets.len; i++) {
        buckets[i] = new LinkedList<>();
    }
    for (i=0; i < oldbuckets.len; i++) {
        for (j=0; j < oldbuckets[i].size(); j++) {
            HMNode temp = oldbuckets[i].get(j);
            put(temp.key, temp.value);
        }
    }
}
```







Node Bprev =



while (cnt != B) {

B =
C =

}

while (cnt != C){

}

reversal

cnext = c.next

Bprev.next = prev

∴ next = Cnext