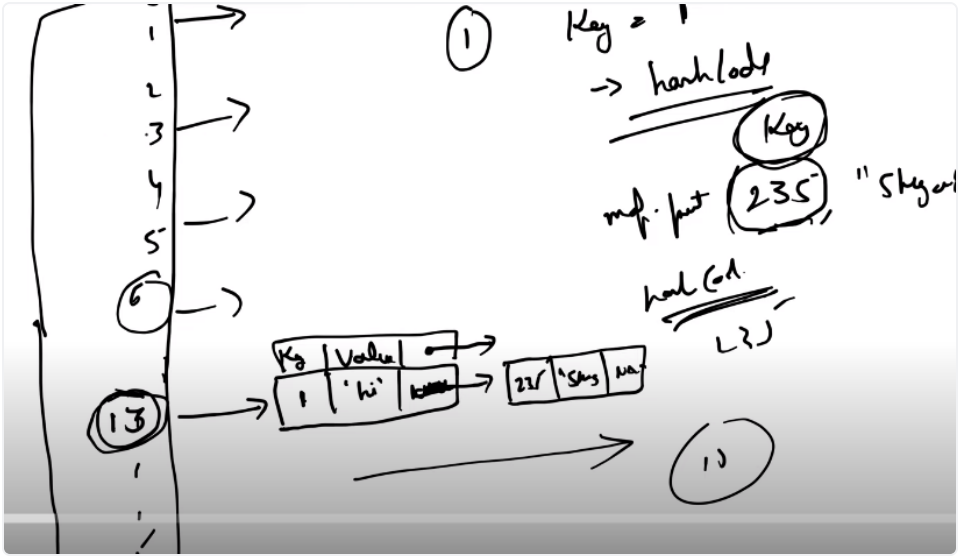
Hashmap Internal Implementation in java | Hashmap in java | Implementing your HashMap in Java

Map<Integer,String> mp = new Hasmap<>();
mp.put(1,"saurav");

- Default size[16]

  it always in 2 pow(0,1,2,3,4,56)
  [2^0, 2^1, 2^2, 2^3, 2^5...]
  For less collision.

- when we apply put function with key value pair. Internally for corresponding key a HASH CODE will be generated from that HASH CODE and index is obtained, on that particular index a linked list for each and every key value pair which is store in the form Node[key, value, next]



- Load Factor
- Collision
- Internal Implementation

Why Maximum capacity of hash map is 1<<30

```
    private static final int MAXIMUM_CAPACITY = 1<<30;
```

- Integer size 4 byte
- 1byte = 8 bit's
- 4byte = 32 bit's
- range for the same -2^31 to 2^31-1
- 1 bit is for signed bit
- Hash map size always 2 to the power something
- could we take Hash Map Size = 2^31 ???.
- You can maximum take 2^31 -1 but hash map size always in 2^ something
- So, we can not take 2^31-1 as maximum size .
- We have to take 2^30 as max size of hash map.

Is user want's to define custom capacity for Hash map.
let's size=4
>>> unsigned right shift operator

- Most significant bit is used as signed bit
- on positive number there is no difference between in >>> and >>, both act the same.
- >> when we work with right shift operator it does not touches the signed bit or Most significant bit , it always work with rest of bit.
- >>> Unsigned right shift operator consider Most significant bit or signed bit .

MUJHE SAMJH ME NHI AYA HAIN >>>unsigned right shift operator
let's capacity = 7 =2^3 = cap

- so we need to find in 2^ [pow]

7=>

32	16	8	4	2	1
0	0	0	1	1	1

cap = cap-1=>6

32	16	8	4	2	1

0	0	0	1	1	0

- here taking capacity =7 and doing cap -1 it doesn't make much difference.
- Here we have 7 and we want make it in the form of 2^3 which is 8.
- Here we are doing cap -1 because we want to make most significant bit is 0

For better understanding let's take example of
Capacity = cap=15

32	16	8	4	2	1
0	0	1	1	1	1

n=cap-1 = 14

32	16	8	4	2	1
0	0	1	1	1	0

n |= n>>>1 => n= n | n>>>1

n= 14

32	16	8	4	2	1
0	0	1	1	1	0

n>>>1 = 7

32	16	8	4	2	1
0	0	0	1	1	1

OR OPERATION BETWEEN [n=n | n>>>1]

	32	16	8	4	2	1
N	0	0	1	1	1	0
N>>>1	0	0	0	1	1	1

OR operation	0	0	1	1	1	1

AND NEW n= 15

32	16	8	4	2	1
0	0	1	1	1	1

n = n | n>>>2

	32	16	8	4	2	1
N	0	0	1	1	1	1
n>>>1	0	0	0	1	1	1
n>>>2	0	0	0	0	1	1

	32	16	8	4	2	1
n	0	0	1	1	1	1
n>>>2	0	0	0	0	1	1
OR operation	0	0	1	1	1	1

AND NEW n= 15

code:

```
package System_Design.LLD.Hashmap_internal_implementation;

public class MyHashMap <K,V>{
    private static final int INITIAL_SIZE = 1<<4;//16
    private static final int MAXIMUM_CAPACITY = 1<<30;

    public Entry[] hashTable;

    public MyHashMap(){
        hashTable = new Entry[INITIAL_SIZE];
    }

    public MyHashMap(int capacity){
        int tableSize = tableSizeFor(capacity);
        hashTable = new Entry[tableSize];
```

```
    }

    private int tableSizeFor(int capacity) {
        int n = capacity-1;
        n |= n>>>1;
        n |= n>>>2;
        n |= n>>>4;
        n |= n>>>8;
        n |= n>>>16;
        return (n<0) ? 1 : (n>=MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY:n+1;
    }

    class Entry<K,V>{
        public K key;
        public V value;
        public Entry next;
        Entry(K k, V v){
            key=k;
            value=v;
        }
    }

    public void put(K key, V value){
        int hashCode = key.hashCode() % hashTable.length;
        Entry node = hashTable[hashCode];

        if(node==null){
            Entry newNode = new Entry(key,value);
            hashTable[hashCode] = newNode;
        }else{
            Entry prevNode = node;
            while(node!=null){
                if(node.key==key){
                    node.value=value;
                    return;
                }
                prevNode=node;
                node=node.next;
            }

            Entry newNode = new Entry(key,value);
            prevNode.next = newNode;

        }

    }

    public V get(K key){
        int hashCode = key.hashCode() % hashTable.length;
        Entry node = hashTable[hashCode];
```

```
        while (node!=null){
            if(node.key.equals(key)){
                return (V) node.value;
            }
            node = node.next;
        }
        return null;
    }

    public static void main(String[] args){
        MyHashMap<Integer,String> mp = new MyHashMap<>();
        mp.put(1,"Hi, ");
        mp.put(2,"My");
        mp.put(3,"Name");
        mp.put(4,"Is");
        mp.put(5,"Saurav");
        mp.put(6,"Suman");
        mp.put(7,"and");
        mp.put(8,"I am Happy");
        String value = mp.get(8);
        System.out.println(value);
    }

}
output:
I am Happy

Process finished with exit code 0
```

Contract B/W Hash code and Equals
Hash code

- If one or same object or key , it doesn't matter how many time we are gonna apply hashcode(), value always will be the same.
- If we have different object's or key it could return same hashcode value. [May or May not be]

Equals

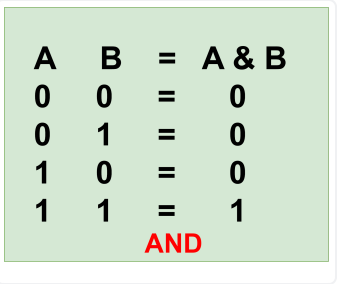
- obj1==obj2 hashcode always same.
- obj1 != obj2 hashcode could be the same

What ever the method we are using to find hash code in our current program there is lot's of chances for collision.

In java HashMap

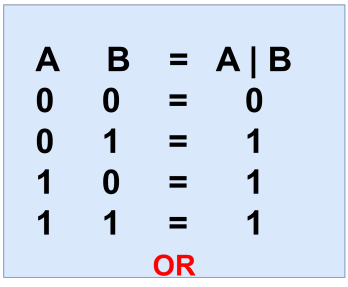
```
/**
 * Computes key.hashCode() and spreads (XORs) higher bits of hash
 * to lower.  Because the table uses power-of-two masking, sets of
 * hashes that vary only in bits above the current mask will
 * always collide. (Among known examples are sets of Float keys
 * holding consecutive whole numbers in small tables.)  So we
 * apply a transform that spreads the impact of higher bits
 * downward. There is a tradeoff between speed, utility, and
 * quality of bit-spreading. Because many common sets of hashes
 * are already reasonably distributed (so don't benefit from
 * spreading), and because we use trees to handle large sets of
 * collisions in bins, we just XOR some shifted bits in the
 * cheapest possible way to reduce systematic lossage, as well as
 * to incorporate impact of the highest bits that would otherwise
 * never be used in index calculations because of table bounds.
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

& [Bitwise AND]



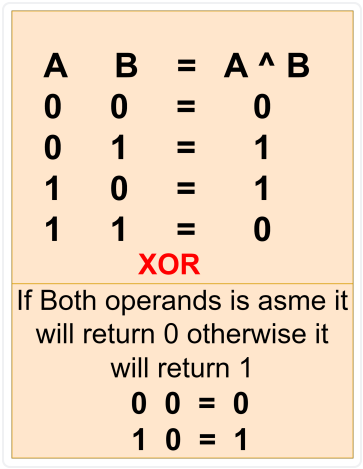
here three 0 and one 1

| [Bitwise OR]



here three times 1 and one times 0

^ [Bitwise XOR]



here two times come 0 and again two times comes 1
so, Probability increases as compare to [and], [or]
So, Probability is 1/2
so collision will decreases.

Load Factor 0.75f;

- hash map capacity full by 75% then we are going resize our hash map.

TREEIFY_THRESHOLD :

- in java 8 if linked list node count increases to 8 or more
- Worst case time complexity O(N)
- the linked list converted into **balanced binary tree or red black tree**
- Worst case time complexity O(log N)

```
/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;
```