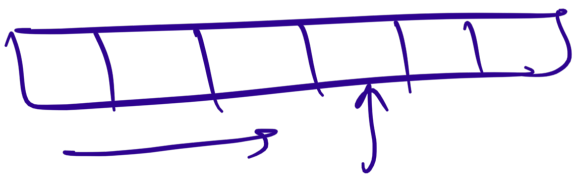
Internal Working of HashMap

Hashing ← HashMap

K → V

mapping

Array

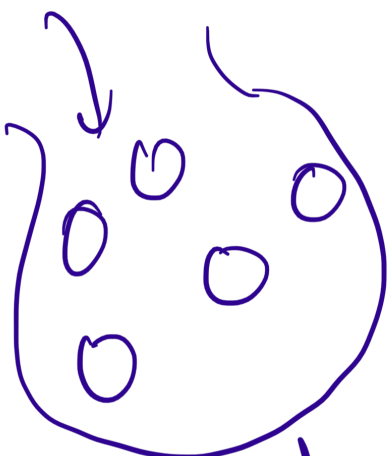


Insert → O(1)

Search → O(1)

Delete → O(1)

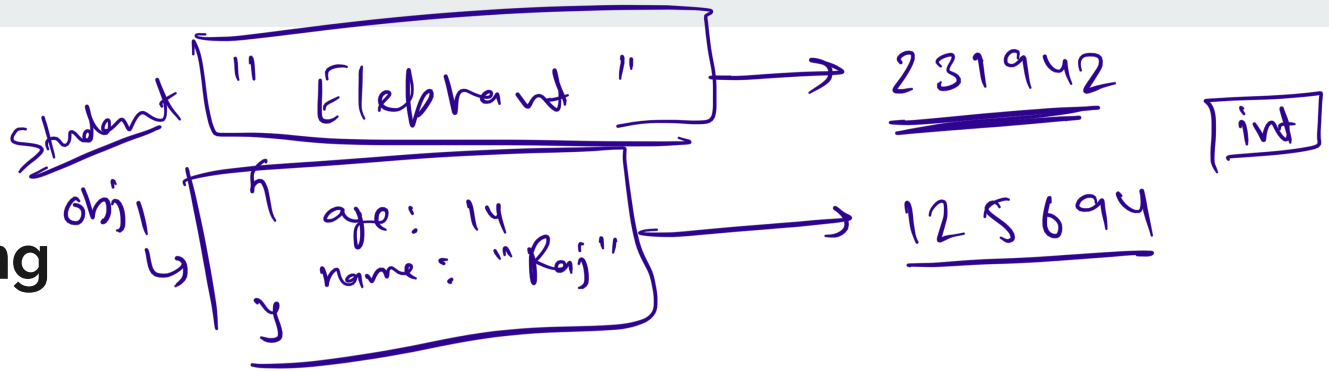
Contigous.



Un-ordered

Non - contigous

Hashing



- The process of assigning a unique value to an object or attribute using an algorithm, which enables quicker access, is known as hashing.
- It's necessary to write the hashCode() method properly for better performance of HashMap.
- hashCode in Java is a function that returns the hashcode value of an object on calling. It returns an integer or a 4 bytes value which is generated by the hashing algorithm.

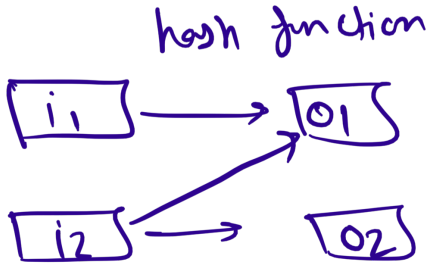
h(obj1) =
h(obj2)

hashCode()
equals()

Integer →
String → Key

hashCode and equals contract

- During the execution of the application, if hashCode() is invoked more than once on the same Object then it must consistently return the same Integer value
- If two Objects are equal, according to the equals(Object) method, then hashCode() method must produce the same Integer on each of the two Objects.
- If two Objects are unequal, according to the equals(Object) method, It is not necessary the Integer value produced by hashCode() method on each of the two Objects will be distinct.



"Cat" → C → 67, A → 65

"Act" → T → 80

Hashing of Strings

```
int hash = 7;
for (int i = 0; i < strlen; i++) {
    hash = hash*31 + charAt(i);
}
```

((7*31 + 67) * 31 + 65)*31 + 80

93 42 56

((67)*31 + 65) *31 + 80

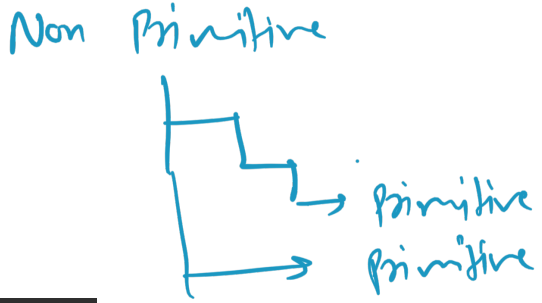
h(i1) = h(i2)

Collision.

Hashing of Custom Class Objects

```java
class Boy {
    String name;
    int age;

    List<Integer> marks;

    @Override
    public int hashCode() {
        return Objects.hash(age, marks, name);
    }
```

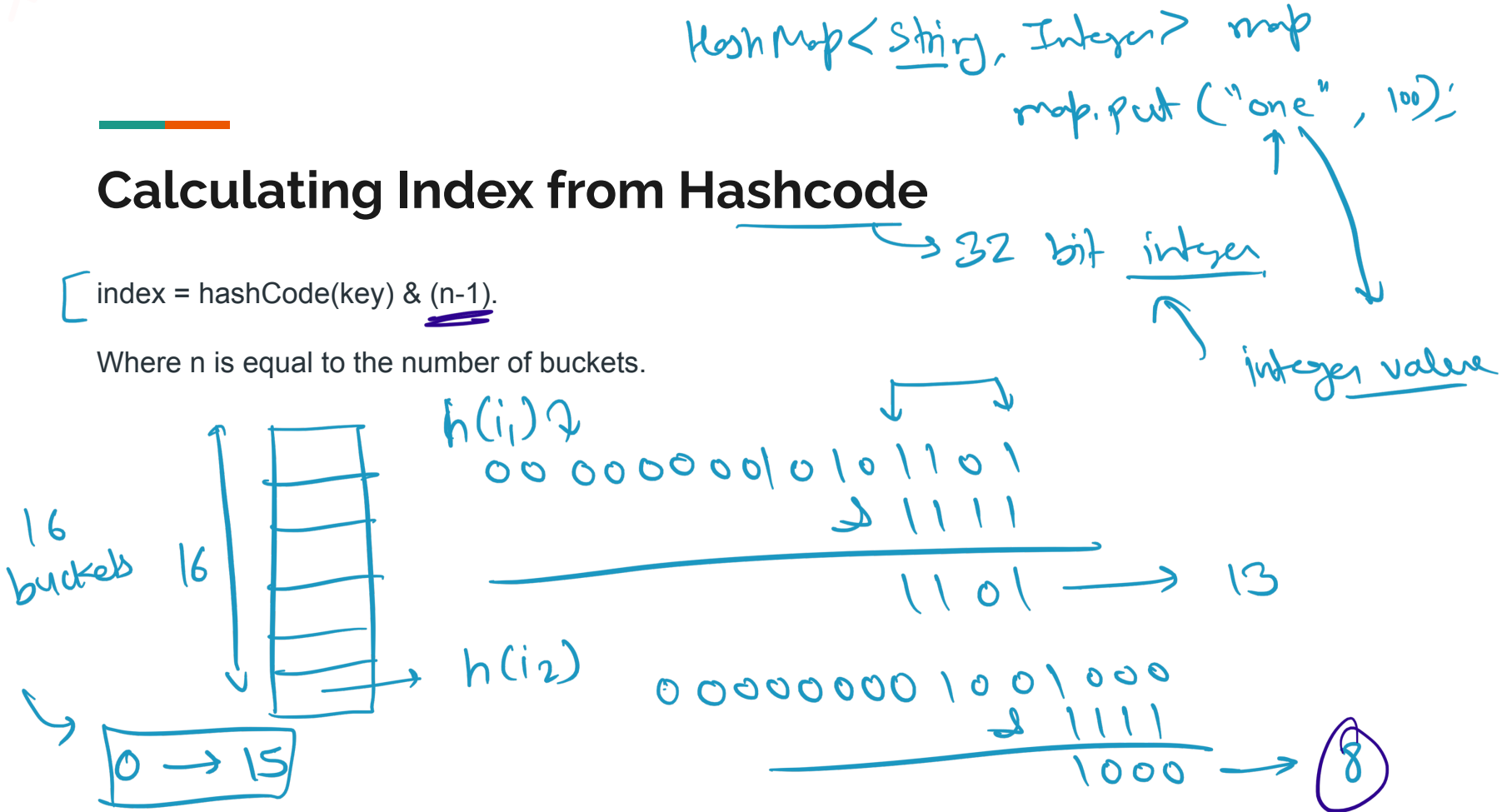
```java
public static int hashCode(Object[] a) {
    if (a == null)
        return 0;

    int result = 1;

    for (Object element : a)
        result = 31 * result + (element == null ? 0 : element.hashCode());

    return result;
}
```



Calculating Index from Hashcode

index = hashCode(key) & (n-1).

Where n is equal to the number of buckets.



h(i3)

0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 0
& 1 1 1 1
1 0 0 0 → 8

Class Node<K, V> {

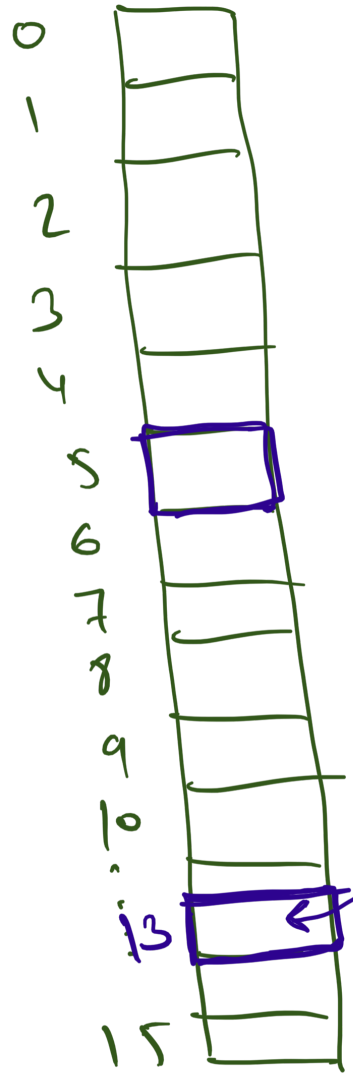
data [K key
V value
int hash

next [Node <K, V> next;

}

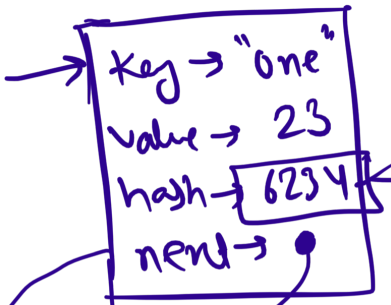
Internal Java HashMap

1. HashMap uses its static inner class Node<K,V> for storing map entries. That means each entry in hashMap is a Node. Internally HashMap uses a hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket where the new entry can be added.
2. HashMap uses multiple buckets and each bucket points to a Singly Linked List where the entries (nodes) are stored.
3. Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket(singly linked list).
4. If there already exists a key with the same hashCode, then the equals() method is used on the keys. If the equals method returns true, that means there is already a node with the same key and hence the value against that key is overwritten in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.
5. If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added to the bucket found.



HashMap<String, Integer> map

map.put("one", 23);



Node

h("one") —assumed→ 6234

hashcode("one")

→ index

hashTable[13] =

6234

↓

0000000000101011 0

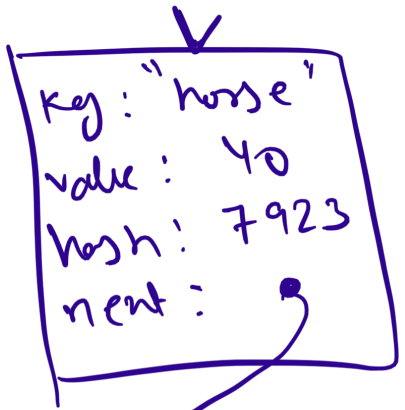
& 1111

1101 → 13

Node hashTable[] = new Node[16];

map.put("horse", 40);

map.put("one", 2);



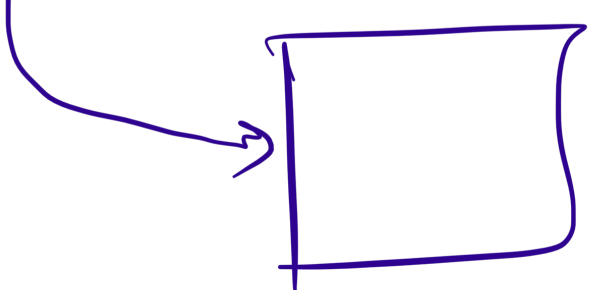
index → 13

equals

"one".equals("horse") → True.

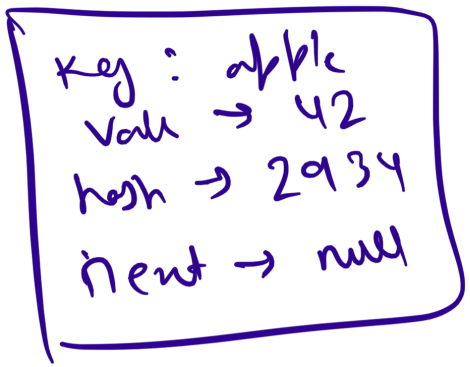
→ false.

map.put("orange", 73);



index → 13

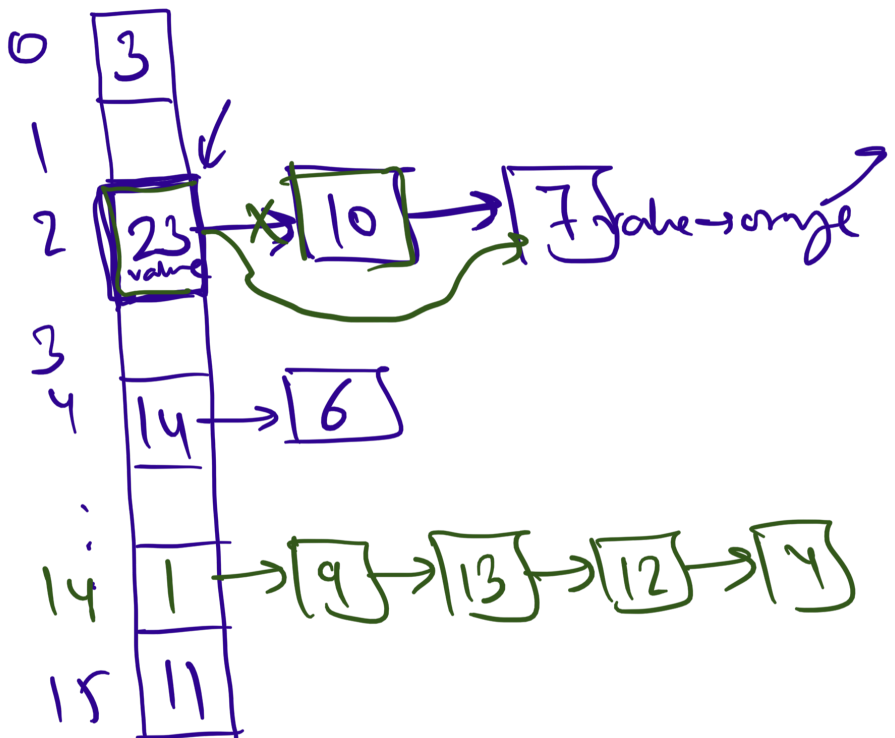
map.put("apple", 42);



index ⟶ 5

How To get(key)?

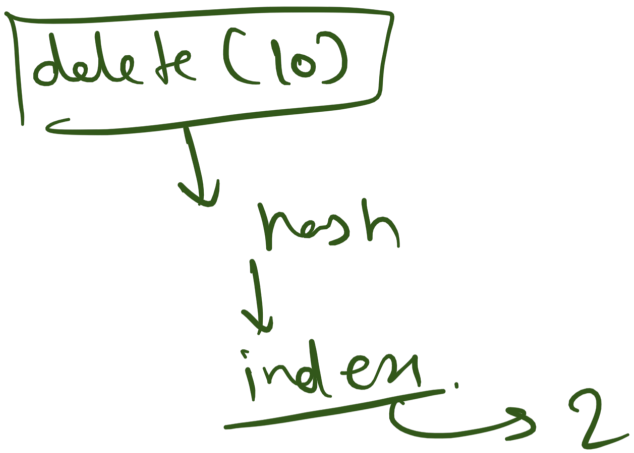
```
Map<Integer, String> map
map.put(23, "Apple")
map.put(14, "egg");
```

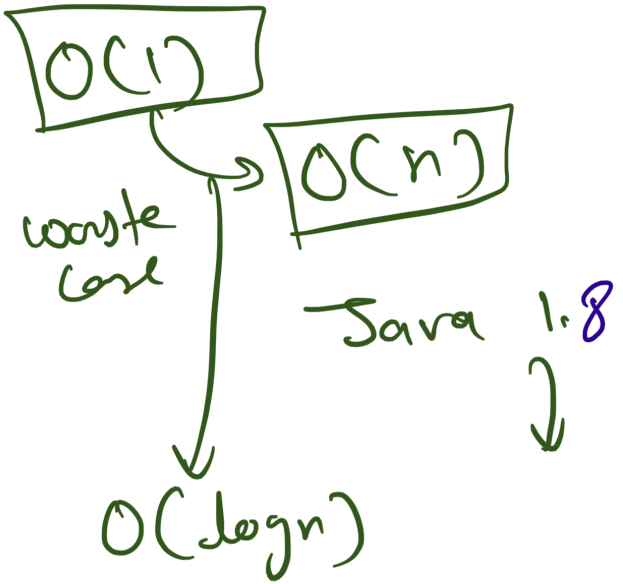


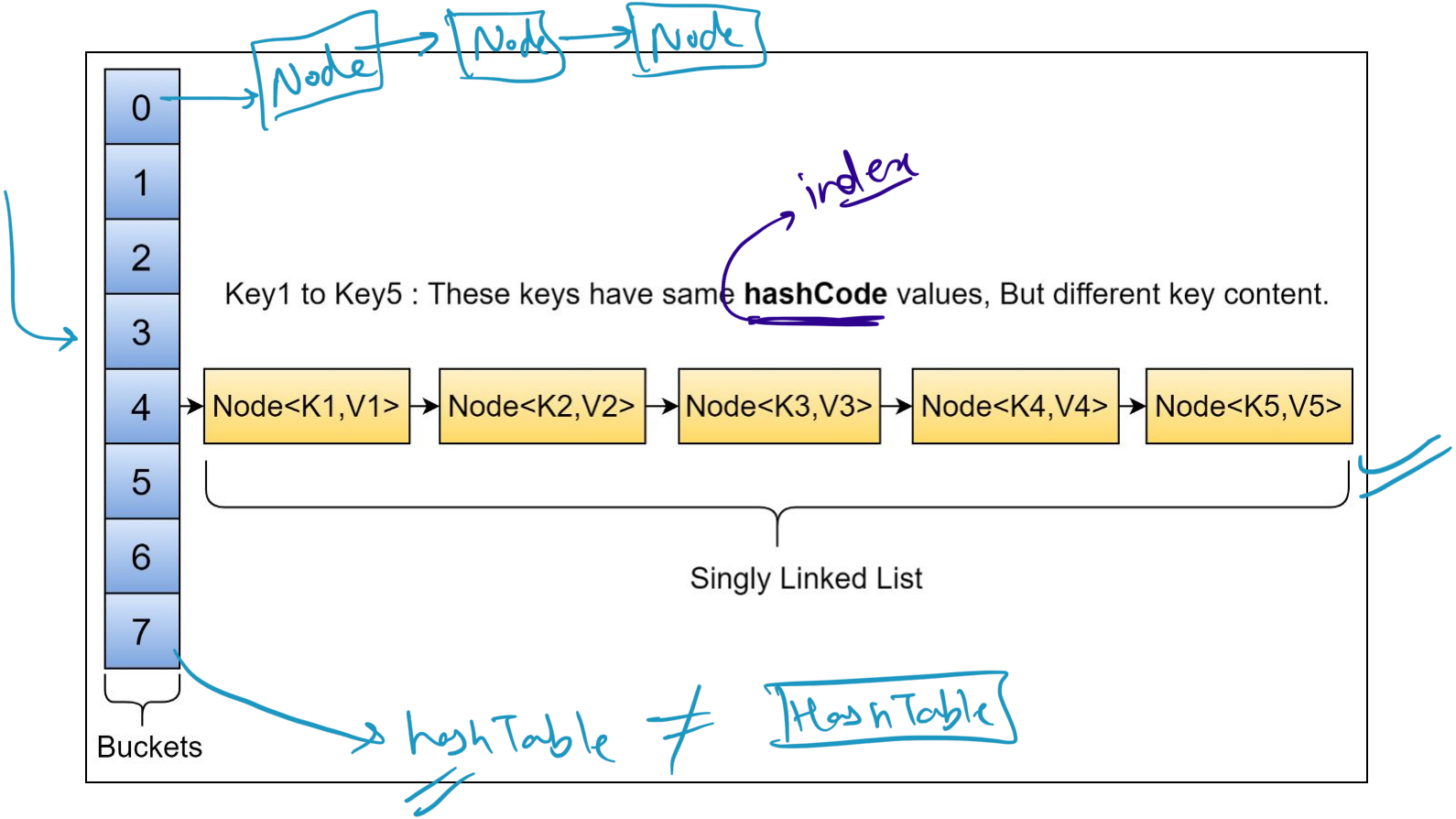

```
map.get(7)
    ↳ "orange"
```

1) Calculate hash
   h(7)
2) find index → 2

How to delete ?






Node
Node
Node
0
1
2
3
4
5
6
7
index
Key1 to Key5 : These keys have same **hashCode** values, But different key content.
Node<K1,V1>
Node<K2,V2>
Node<K3,V3>
Node<K4,V4>
Node<K5,V5>
Singly Linked List
Buckets
HashTable ≠ HashTable

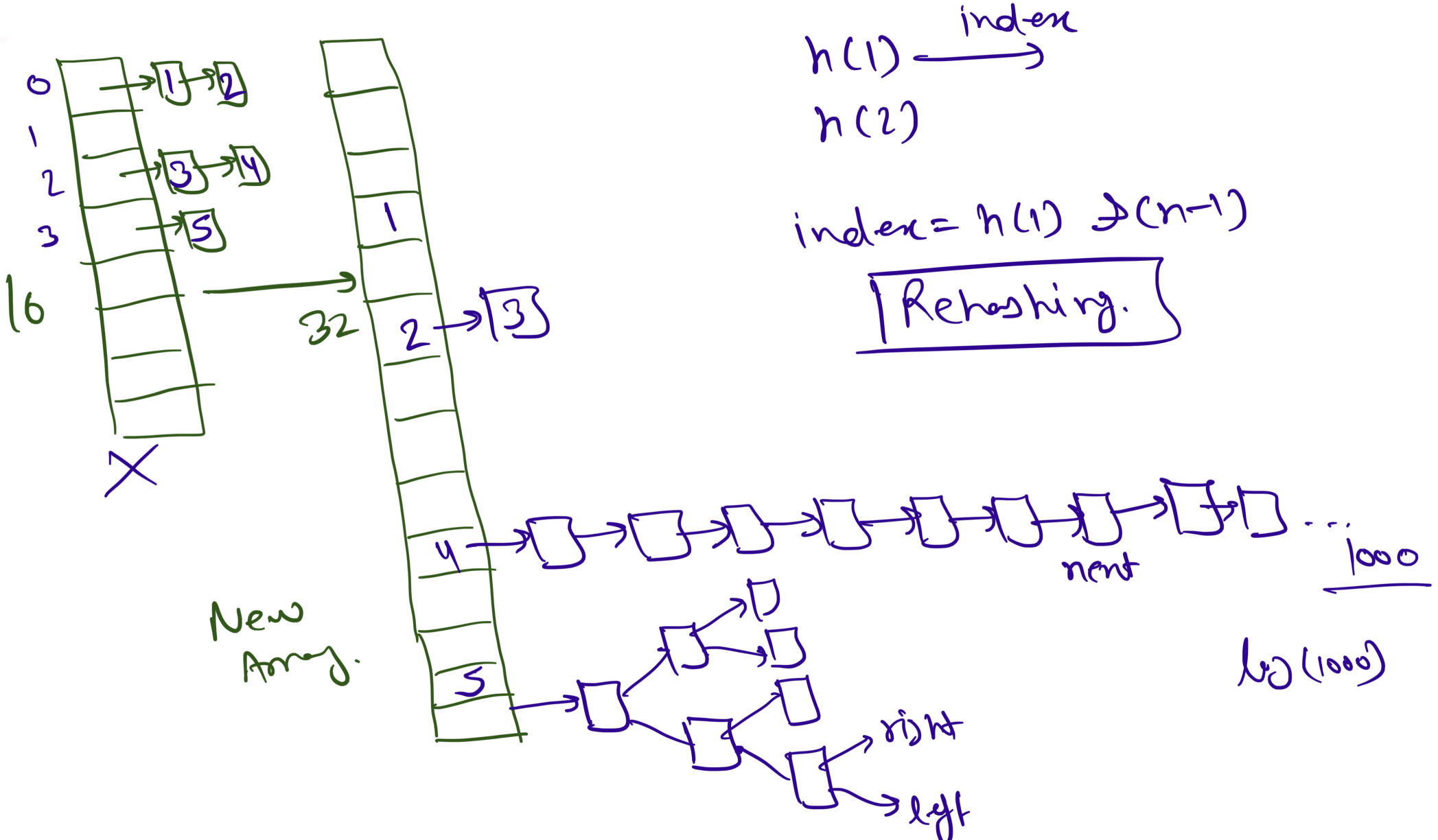
Load factor & Rehashing

Whenever the number of entries in the hashmap crosses the threshold value then the bucket size of the hashmap is doubled and **rehashing** is performed and all already existing entries(nodes) of the map are copied and new entries are added to this increased hashmap.

```
Threshold value = Bucket size * Load factor
```

Eg. If the bucket size is 16 and the load factor is 0.75 then the threshold value is 12.

16 0.5 → above 8


h(1) → index
h(2)
index = h(1) % (n-1)
Rehashing.
0
1
2
3
16
32
New Array.
next
1000
log(1000)
right
left

```
1       Map<String, Long> myPhoneBook = new HashMap<>();
2       //Bucket size 16 and LD=0.75
3       Map<String, Long> myPhoneBook = new HashMap<>(64);
4       //Bucket size 64 and LD=0.75
5       Map<String, Long> myPhoneBook = new HashMap<>(128, 0.90f);
6       //Bucket size 16 and LD=0.9
7       Map<String, Long> myPhoneBook = new HashMap<>(parentsPhoneBook);
8       //Bucket copy the parentsPhoneBook to myPhoneBook
```

Time Complexity

Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a thresold is crossed (static final int TREEIFY_THRESHOLD = 8;). The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup.

- In a fairly distributed hashMap where the entries go to all the buckets in such a scenario, the hashMap has O(1) time for search, insertion, and deletion operations.
- In the worst case, where all the entries go to the same bucket and the singly linked list stores these entries, O (n) time is required for operations like search, insert, and delete.
- In a case where the threshold for converting this linked list to a self-balancing binary search tree(i.e. AVL/Red black) is used then for the operations, search, insert and delete O(log(n)) is required as AVL/Red Black tree has a max length of log(n) in the worst case.

Practice:

- interviewbit → hosting → Intermediate to Advanced
- Leet code → hosting → Intermediate to Advanced
- GfG
  - School
  - Easy
  - Medium.

GeeksforGeeks.