Hashing is been introduced to improve the searching capability & avoiding the duplicates.

On Hashing we can search the element in Constant time O(1) in average case.

In worst case it would be taking O(logn).

Hashing Data Structures

Set

Set allows only value.
Does not allow duplicates.

Map

Map allows Key, value Pairs.
Allows only unique keys.
Allows Duplicate values.
Map replaces the value when the key is present.

How Hashing works ?

Simple, in hashing we try to understand the behaviour of a data then we derive a hash logic so that we can link similar group of data.

Ultimately this Hash Logic makes an important role here.

In hashing similar group of data can be represented by Bucket.

In simple first we decide, how many buckets we need, then we try to link data to buckets based on their hash logic.

Hash Collisions : There is always high possibility that different sets of data , can point to same bucket, it does not mean that they are equal. This is called Hash Collision, to support this we map LinkedList to the bucket.

Hashing Data Structures

Set

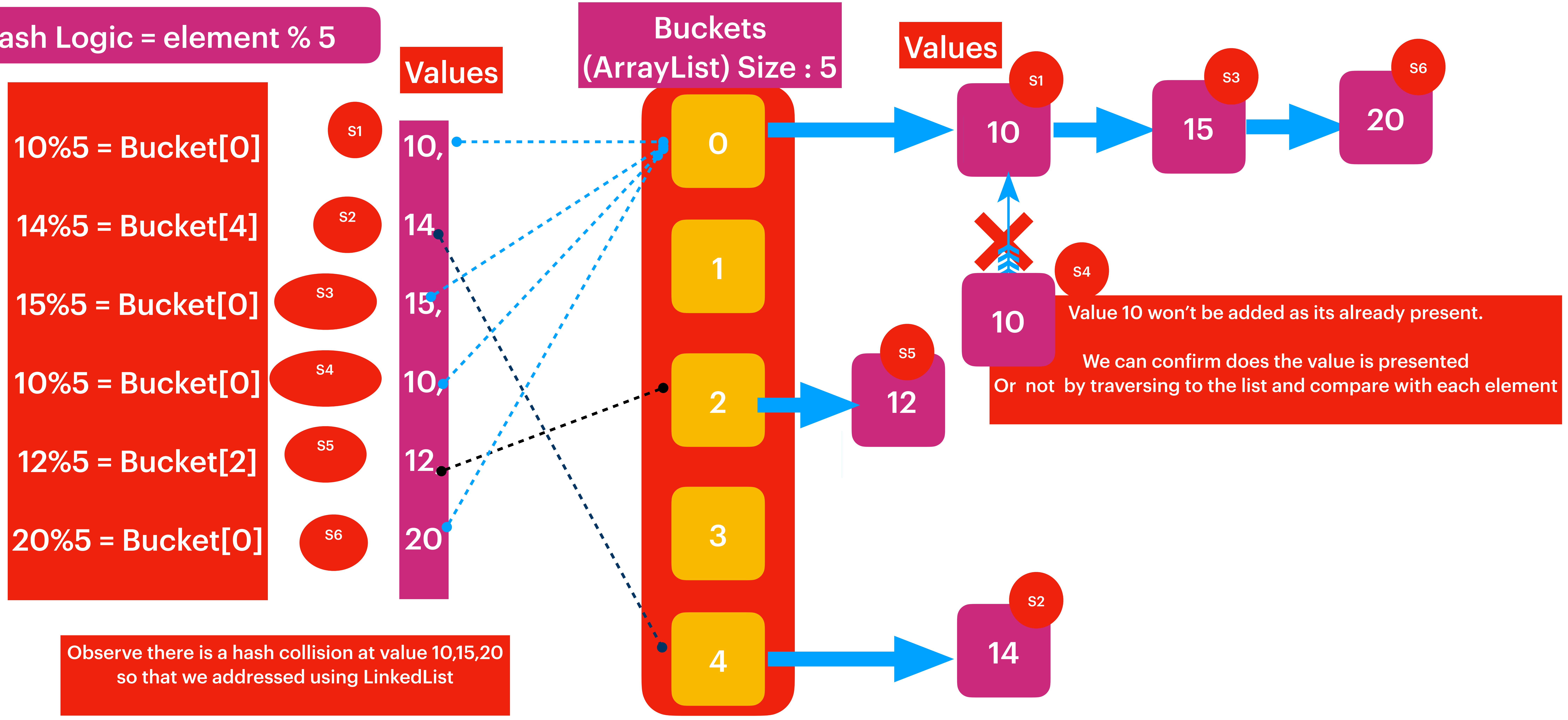
Set allows only value.
Does not allow duplicates.

Map

Map allows Key, value Pairs.
Allows only unique keys.
Allows Duplicate values.
Map replaces the value when the key is present.

Finally, in hashing data structure
First we understand the behaviour of a data, we decide how many initial buckets we need then derive the hash logic, this hash logic is going to represent a bucket.
As we know that there is a hash collision, while adding element to avoid duplicates we compare then add .

Hash Logic = element % 5



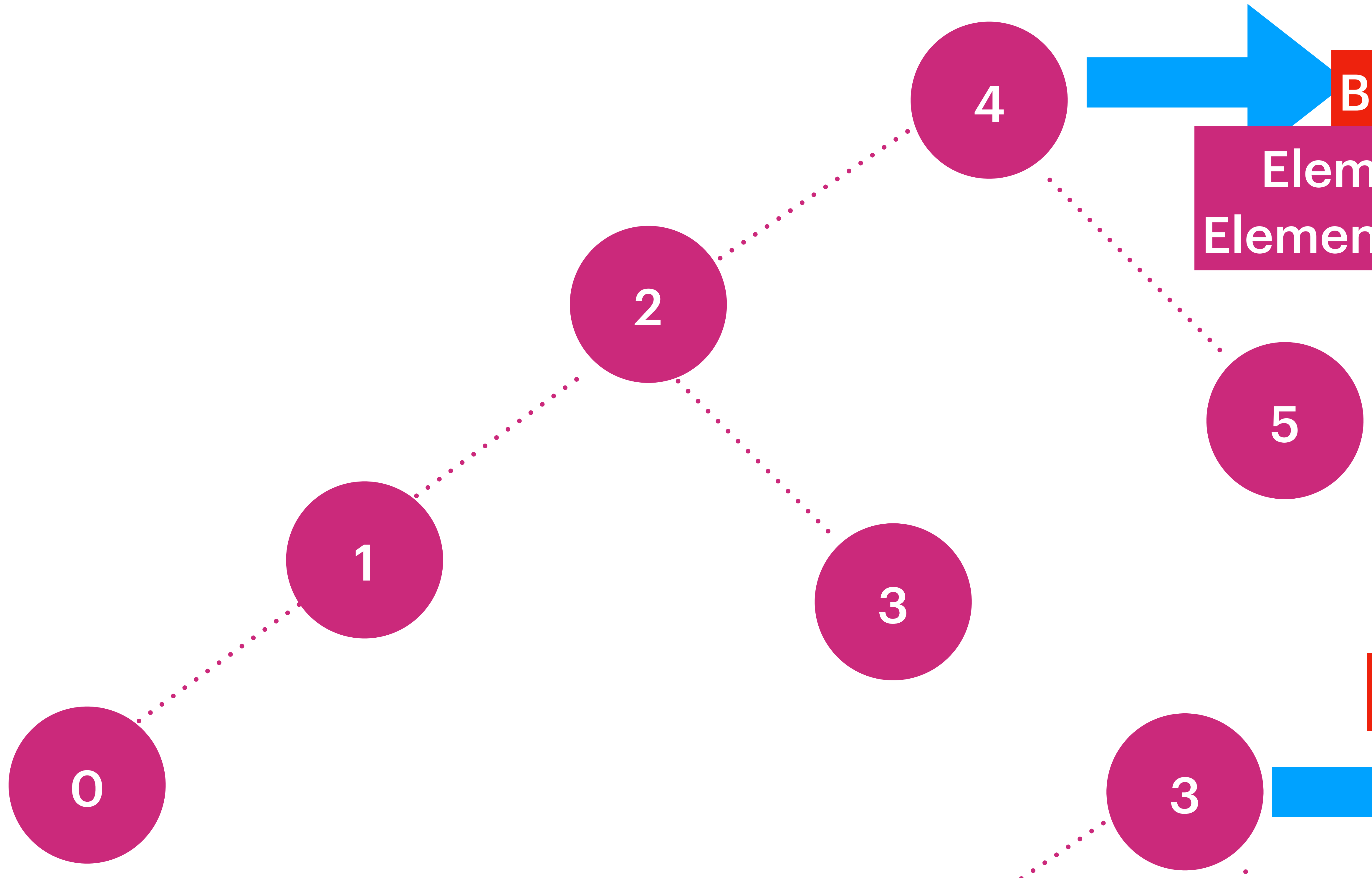
Value 10 won't be added as its already present.

We can confirm does the value is presented Or not by traversing to the list and compare with each element

Observe there is a hash collision at value 10,15,20 so that we addressed using LinkedList

A bad hash logic or bad dataset can result to pointing of all the elements to single bucket. To address this after reaching threshold limit we would convert LinkedList to Height based Balanced Binary Search Tree (AVL), which enables us to add/search/delete element in O(logn) time.

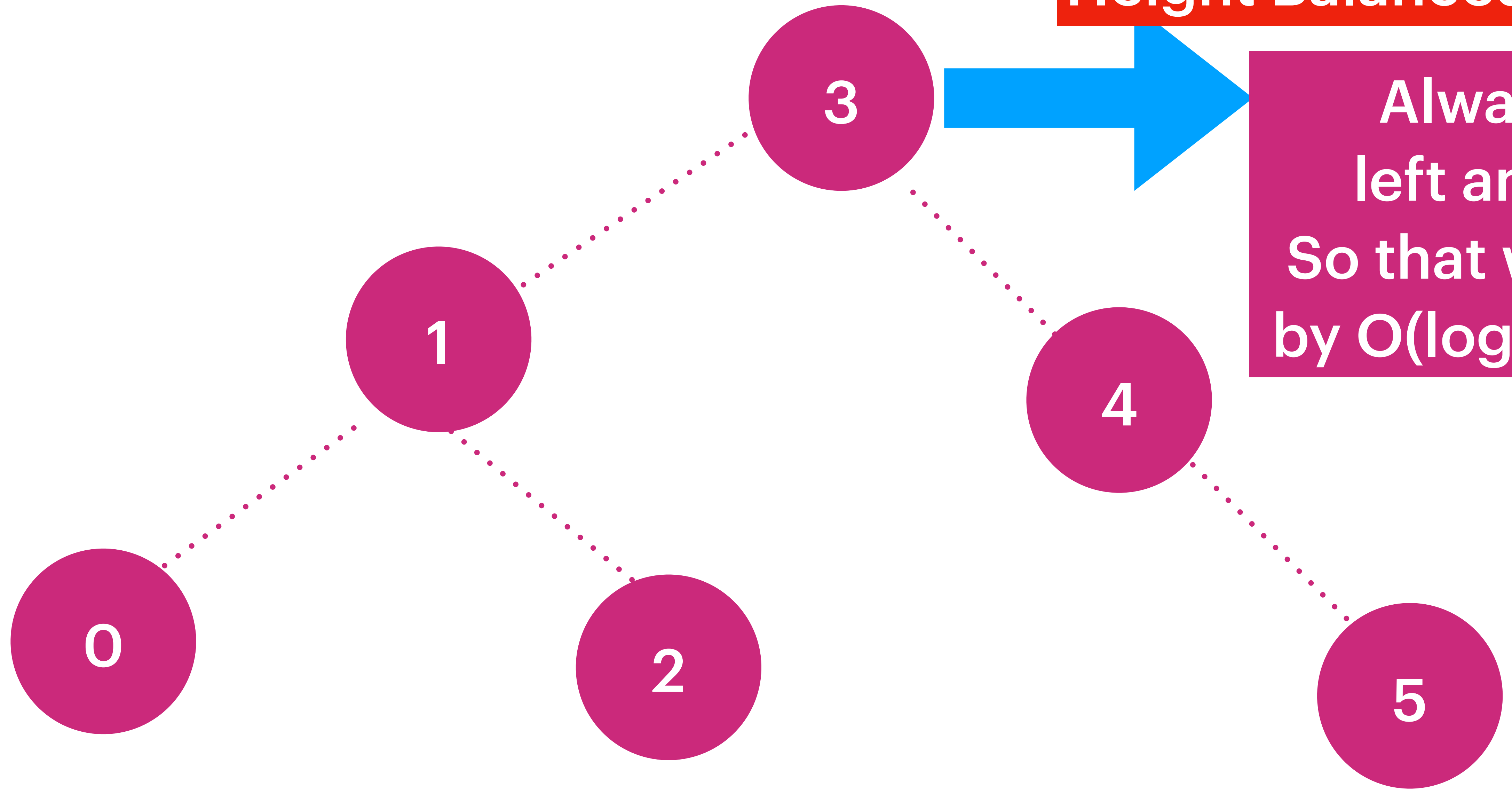
Time Complexity :
Insertion Node : Avg Case O(1) / Worst Case O(logn)
Delete Node : Avg Case O(1) / Worst Case O(logn)
Search : Avg Case O(1) / Worst Case O(logn)



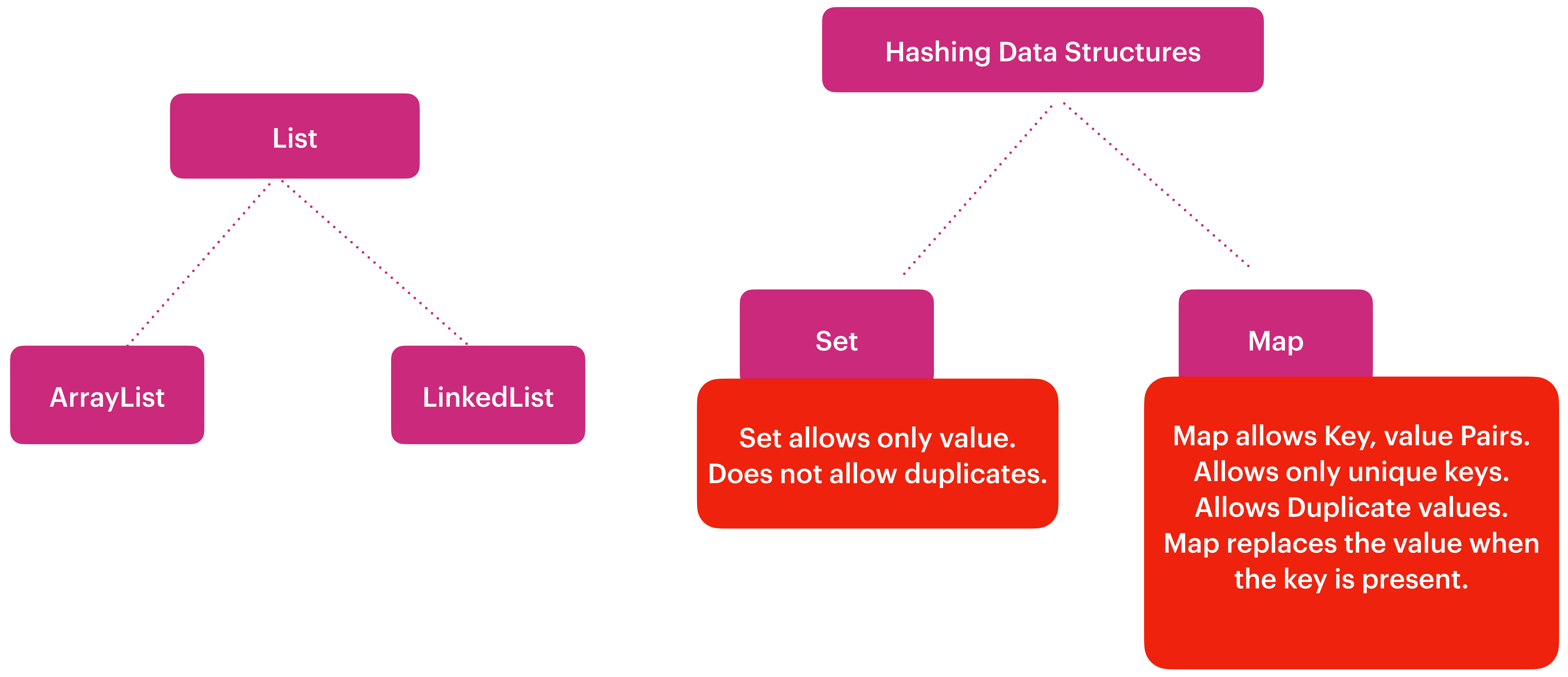
Binary Search Tree

Elements less than the root are on left side,
Elements greater than the root are on right side.

Height Balanced Binary Search Tree



Always height diff between
left and right subtrees is <= 1.
So that we can add/delete/search
by O(logn) time even in worst case.



Set allows only value.
Does not allow duplicates.

Map allows Key, value Pairs.
Allows only unique keys.
Allows Duplicate values.
Map replaces the value when the key is present.

Importance Of LoadFactor In Hashing

Hash Logic = element % size

Initial bucketSize = 5
LoadFactor = 0.75

6 => 6%5 => Bucket(1)
7 => Bucket(1)
8 => Bucket(3)
9 => Bucket(4)

↓

Reached 75% capacity
= Math.round(5 * 0.75) = 4

↓ add(11)

Rehash all the elements

↓

bucketSize = size * 2 = 10

6%10 = Bucket(6)
7%10 = Bucket(7)
8%10 = Bucket(8)
9%10 = Bucket(9)
11%10 = Bucket(1)

LoadFactor can speak about when to rehash the Data Structure.
Usually we setup LoadFactor to 75%.
It means as and when 75% of the buckets were occupied
then we double the bucket size.
Rehashing of each element is needed as and when we double the bucket size.
Rehashing is always costly operation so that choosing bucket size will always gives best performance.
As per documentation => size = actualSize/loadFactor ,
Assume as per your requirement you need 5 buckets then
size = Math.round(5/0.75) = 7

From Documentation :
An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.
As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the HashMap class, including get and put). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operations will ever occur.