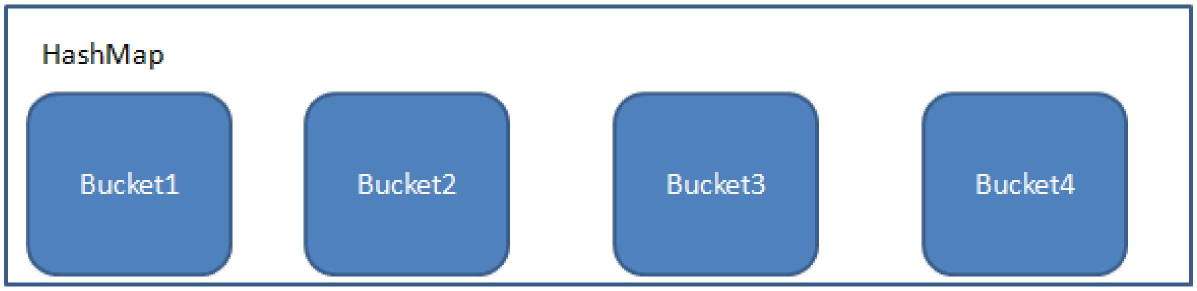
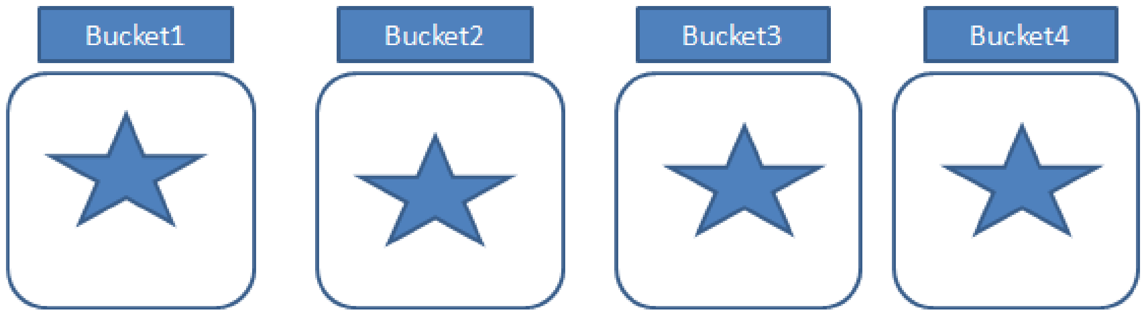
HashMap Performance Improvements in Java 8

HashMap contains a certain number of buckets. It uses hashCode to determine which bucket to put these into. For simplicity's sake imagine it as a modulus.

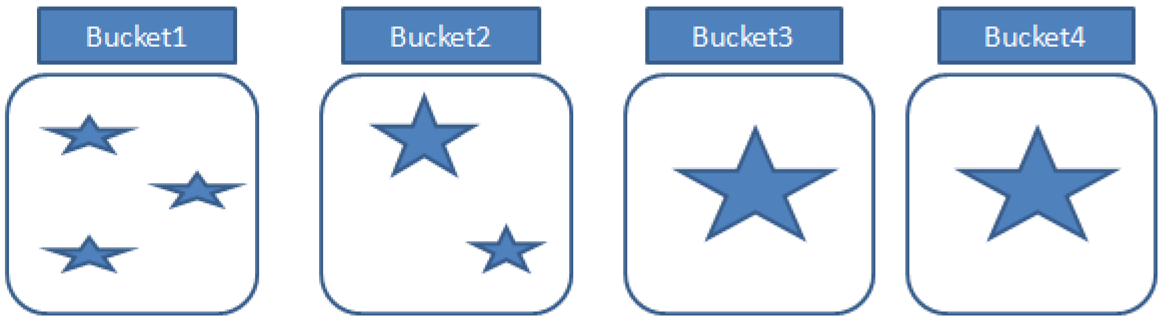
If our hashcode is 123456 and we have 4 buckets, 123456 % 4 = 0 so the item goes in the first bucket, Bucket 1.



If you have strong hashcode algorithm then elements in haspMap dispersed equally in the buckets



If you have poor hashcode algorithm then elements in haspMap then collision happens and more than one elements stored in the same bucket using LinkedList



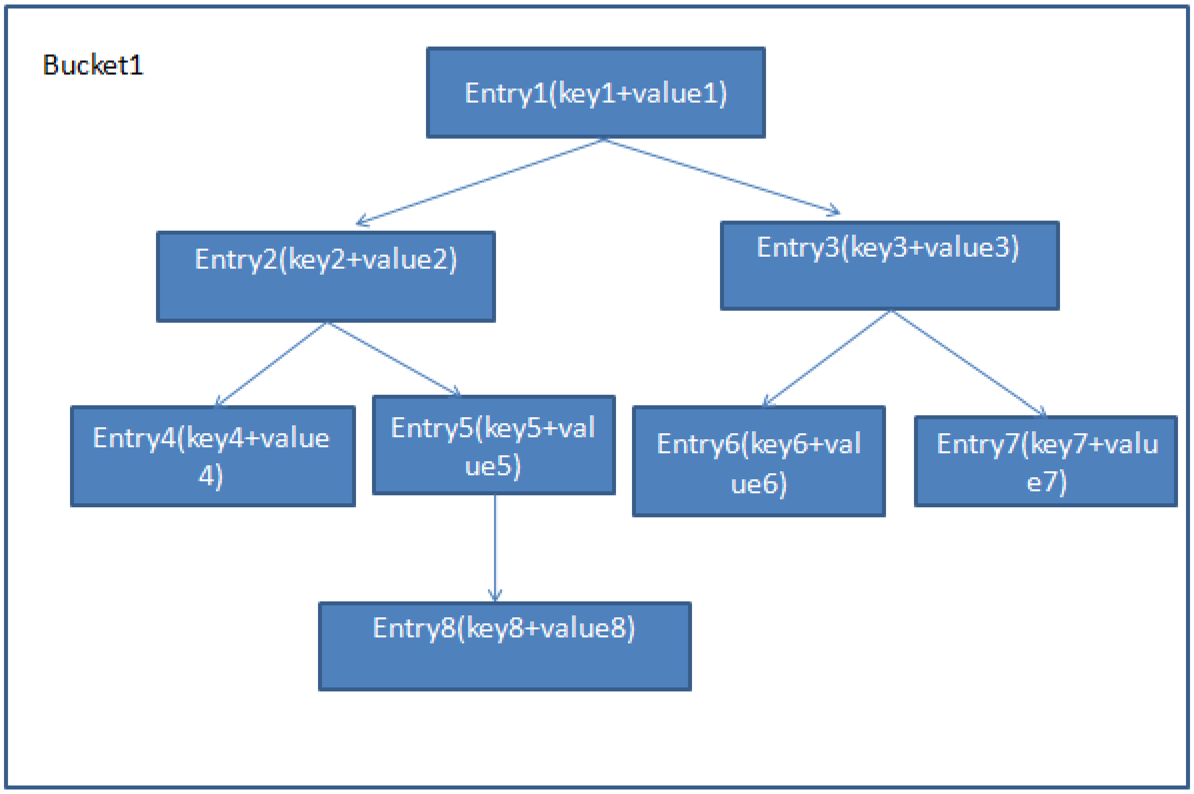
The less even this distribution is, the further we're moving from O(1) operations and the closer we're moving towards O(n) operations.

In Java 8, To improve the performance of HashMap during the collisions, below three properties introduced

```
static final int TREEIFY_THRESHOLD = 8;
static final int UNTREEIFY_THRESHOLD = 6;
static final int MIN_TREEIFY_CAPACITY = 64;
```

In Java 8, HashMap elements use balanced trees instead of linked lists under certain circumstances now.

The implementation of Hashmap in Java 8 tries to mitigate this by organizing some buckets into trees rather than linked lists if the buckets become too large. This is what TREEIFY_THRESHOLD = 8 is for. If a bucket contains more than eight items, it should become a tree.



It is first sorted by hash code. If the hash codes are the same, it uses the compareTo method of Comparable if the objects implement that interface, else the identity hash code.

If entries are removed from the map, the number of entries in the bucket might reduce such that this tree structure is no longer necessary. That's what the UNTREEIFY_THRESHOLD = 6 is for.

If the number of elements in a bucket drops below six, we might as well go back to using a linked list.

Finally, there is the MIN_TREEIFY_CAPACITY = 64.
When a hash map grows in size, it automatically resizes itself to have more buckets. If we have a small hash map, the likelihood of us getting very full buckets is quite high, because we don't that have many different buckets to put stuff into. It's much better to have a bigger hash map, with more buckets that are less full. This constant basically says not to start making buckets into trees if our hash map is very small - it should resize to be larger first instead.

- Best Case: O(1). Hashcode of all the keys are distinct, then get and put operations run in O(1) i.e. constant time (independent of the number of keys in the map).
- Worst Case (Bins): O(n). Hashcodes of all keys are same then same operations can take O(n) time (n = number of keys in the map).
- Worst Case (Trees): O(log(n)). Hashcodes of all keys are same then same operations can take O(log(n)) time (n = number of keys in the map).