Lecture 13

# Hashing, HashMap and HashSet in Java

**Last time:**

We implemented a very simple HashTable using linear probing. Its hash function is also simple since the keys are positive integers and we assumed that it would be random.

```
key % tableLength;
```

But, having random keys is not always the case. To handle non-random keys, we talked about a few general rules for a better hash function.

- **Don't Use Non-Data**: The keys should be squeezed as much as it could. For example, category code has to be changed to be from 0 to 15. Also, the checksum should be removed because it is a derived number from other information and does not add any more information.
- **Use All the Data**: Other than the non-data values, we need to use all the data values. Don't just use the first four digits, etc.
- **Use a Prime Number for the Modulo Base**: Which means the table array length should be a prime number. This can prevent clusters from happening. For example, if the table size is 50, then all the multiples of 50 in a dataset will be hashed into the same index.
- **Use Folding**: Another reasonable hash function involves breaking the keys into groups of digits and adding (mixing up) the groups.

It is easy for you to find a huge literature on the subject. So, I encourage you to explore if you are curious about it.

Today, we will look into hashing in Java.

## hashCode() method

The hashCode() method in Java is implemented in the Object class.

What does it mean?
Each class in Java inherits it!

Basically, the hashCode() method returns a numeric representation of an object.

```
Integer intObj = new Integer(8722);
String strObj = "8722";
System.out.println("hashCode for integer: " + intObj.hashCode());
System.out.println("hashCode for string: " + strObj.hashCode());
```

This will print
hashCode for integer: _____
hashCode for string: _____

For example, hashCode method in the String class uses the following formula.

```
s.charAt(0)* 31^(n-1) + s.charAt(1)*31^(n-2) + … + s.charAt(n-1)
```

For example,
“cats”

Now, why 31?

What does this tell us?

## HashMap

The Map interface in Java is not an extension of the Collection interface. It starts off its own hierarchy.

The interface describes a mapping from keys to values, without duplicate keys, by definition.

As a result, maps provide a good way of searching for an object on the value of another.

Other than the fact that it is key and value mapping implementation, HashMap is a hashtable as we discussed so far.

Here are some code snippets from HashMap class (Java 7).

```
/**
 * The table, resized as necessary. Length MUST Always be a power
of two.
 */
….. Entry<K,V>[] table;

public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    threshold = (int)(DEFAULT_INITIAL_CAPACITY *
DEFAULT_LOAD_FACTOR);
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}
```

```
static class Entry<K,V> ……{
    final K key;
    V value;
    Entry<K,V> next;
    int hash;
    Entry(int h, K k, V v, Entry<K, V>  n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }
}
```

```
public V put(K key, V value) {
    if (key == null)
        return putForNullKey(value);
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key ||
key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}

static int indexFor(int h, int length) {
    return h & (length-1);
}

void addEntry(int hash, K key, V value, int bucketIndex) {
    Entry<K, V> e = table[bucketIndex];
    table[bucketIndex] = new Entry<>(hash, key, value, e);
    if (size++ >= threshold)
        resize(2*table.length);
}
```

## Why Java HashMap uses the power-of-two as the length of its array?

## Analysis of indexFor and hash methods

Can you see how the HashMap class tries to resolve collisions (open addressing or separate chaining)?

Let's try to use the HashMap for a small application, word frequency counter!

Major methods of the HashMap in java are:

- put(key, value): associates the value with the key in the map
- get(key): returns the value of the key or null.
- keySet(): returns a set view of the keys in the map
- values(): returns a collection view of the values in the map

For keySet() and values() methods, we can use an Iterator to traverse the returned values.

**A simple application: implementing the frequency count of words**

```
/*
 * You can try to tune initial capacity and load factor.
 * default values are 16 and 0.75 respectively.
 */
Map<Sting, Integer> freqOfWords = new HashMap<String,
Integer>(10, 0.65f);

String[] words = "coming together is a beginning keeping together
is progress working together is success".split(" ");

for (String word : words) {
    Integer frequency = freqOfWords.get(_________________);
    if (frequency == null) {
        frequency = 1;
    } else {
        frequency++;
    }
    freqOfWords.put(_____________________, _____________________);
}
```

How do we iterate all the words and print each per one line?

```
Iterator<String> itr = freqOfWords._________________.iterator();

while (itr._________________) {
    System.out.println(itr._________________);
}
```

Or

```
for (String word : freqOfWords._____________________) {
    System.out.println(word);
}
```

## HashSet

Java provides HashSet class which is a regular set in which all objects are distinct.

HashSet's major methods are:

- add(key) : adds a key to the set.
- contains(key) : returns true if the set has that key
- iterator() : returns an iterator over the elements

HashSet class implements the Set interface **using HashMap instance**.

Look at some simplified source code snippets of the HashSet class.

```
public class HashSet<E> implements Set<E> {
    private … HashMap<E, Object> map;
    // Dummy value to associate with an Object in the backing Map
    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }

    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
}
```

What do you see here?
It creates HashMap instance and it uses HashMap's put method to add a new element but with dummy object as its value.

And, it adds, or puts, a new element only if it is not already present in the map.

How does it check an object is a duplicate or not?

It stores and retrieves elements by using a hash function that converts elements into an integer.

Let's try to use the HashSet for a small application, distinct word counter!

```
Set<String> distinctWords = new HashSet<String>();
String[] words = "coming together is a beginning keeping together
is progress working together is success".split(" ");

for (String word: words) {
    distinctWords.add(word);
}

System.out.println("There are " + distinctWords.size() + "
distinct words.");
System.out.println("They are: " + distinctWords);
```

How would you iterate and print words one per line?