Hashmaps

Introduction to Hashmaps

Suppose we are given a string or a character array and asked to find the maximum occurring character. It could be quickly done using arrays. We can simply create an array of size 256, initialize this array to zero, and then, simply traverse the array to increase the count of each character against its ASCII value in the frequency array. In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called hashmaps.

In hashmaps, the data is stored in the form of keys against which some value is assigned. Keys and values don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example: The given string array is:

str[] = {"abc", "def", "ab", "abc", "def", "abc"}

Hashmap will look like as follows:

Key (datatype = string)	Value (datatype = int)
“abc”	3
“def”	2
“ab”	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: *One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.*

The values are stored corresponding to their respective keys and can be invoked using these keys. To insert, we can do the following:

- hashmap.put(key, value)

The functions that are required for the hashmaps are(using templates):

- **insert(k key, v value):** To insert the value of type v against the key of type k.
- **get(k key):** To get the value stored against the key of type k.
- **remove(k key):** To delete the key of type k, and hence the value corresponding to it.

To implement the hashmaps, we can use the following data structures:

1. **Linked Lists:** To perform insertion, deletion, and search operations in the linked list, the time complexity will be O(n) for each as:
    - For insertion, we will first have to check if the key already exists or not, if it exists, then we just have to update the value stored corresponding to that key.
    - For search and deletion, we will be traversing the length of the linked list.

2. **BST:** We will be using some kind of a balanced BST so that the height remains of the order O(logN). For using a BST, we will need some sort of comparison between the keys. In the case of strings, we can do the same. Hence, insertion, search, and deletion operations are directly proportional to the height of the BST. Thus the time complexity reduces to O(logN) for each.
3. **Hash table:** Using a hash table, the time complexity of insertion, deletion, and search operations, could be improved to O(1) (same as that of arrays). We will study this in further sections.

Inbuilt Hashmap

In Java, we have two types of hashmaps:

- **Treemap** (uses BST implementation)
- **Hashmap** (uses hash table implementation)

Note: Both have similar functions and similar ways to use, differing only in the time complexities. The time complexity of each of the operations(insertion, deletion, and searching) in the **Treemap** is O(logN), while in the case of **Hashmap**, they are O(1).

Hashmap:

- Import hashmap library

```
import java.util.HashMap;
```

- Syntax to declare:

```
HashMap<datatype_for_keys, datatype_for_values> name;
```

- Operations performed:
    1. **Insertion:** Suppose, we want to insert the string “abc” with the value 1 in the hashmap named *ourmap*, there are two ways to do so:
        - Simply create a pair of both and insert in the map using .**insert** function. Syntax:

```
HashMap<String, Integer> ourmap = new HashMap<>();
```

```
ourmap.put("abc", 1);
```

2. **Searching:** Suppose we want to find the value stored in the hashmap against key *"abc"*, there are two ways to do so:
    - As did in insertion like arrays, the same way we can figure out the value stored against the corresponding key. Syntax:

```
int value = ourmap.get("abc");
```

Note: If we try to access a key that is not present in the unordered_map, then there are two different outcomes:

➢ If we are accessing the value using .get() function, then we will get an error specifying that we are trying to access the value that is not present in the map.

But what if we want to check if the key is present or not on the map? For that, we will be using the .count() function, which tells if the key is present in the map or not. It returns false, if not present, and true, if present

Syntax:

```
boolean isPresent = ourmap.containsKey("ghi");
```

Note: *We can also check the size of the map by using .size() function, which returns the number of key-value pairs present on the map.*

Syntax:

```
int size_of_map = ourmap.size();
```

3. Deletion: Suppose we want to delete the key *"abc"* from the map, we will be using .remove() function.

Syntax:

```
ourmap.remove("abc");
```

Remove Duplicates

Problem statement: Given an array of integers, we need to remove the duplicate values from that array, and the values should be in the same order as present in the array.

For example: Suppose the given array is arr = {1, 3, 6, 2, 4, 1, 4, 2, 3, 2, 4, 6}, answer should be {1, 3, 6, 2, 4}.

Approach: We will add unique values to the vector and then return it. To check for unique values, start traversing the array, and for each array element, check if the value is already present in the map or not. If not, then we will insert that value in the vector and update the map; otherwise, we will proceed to the next index of the array without making any changes.

Let's look at the code for better understanding.

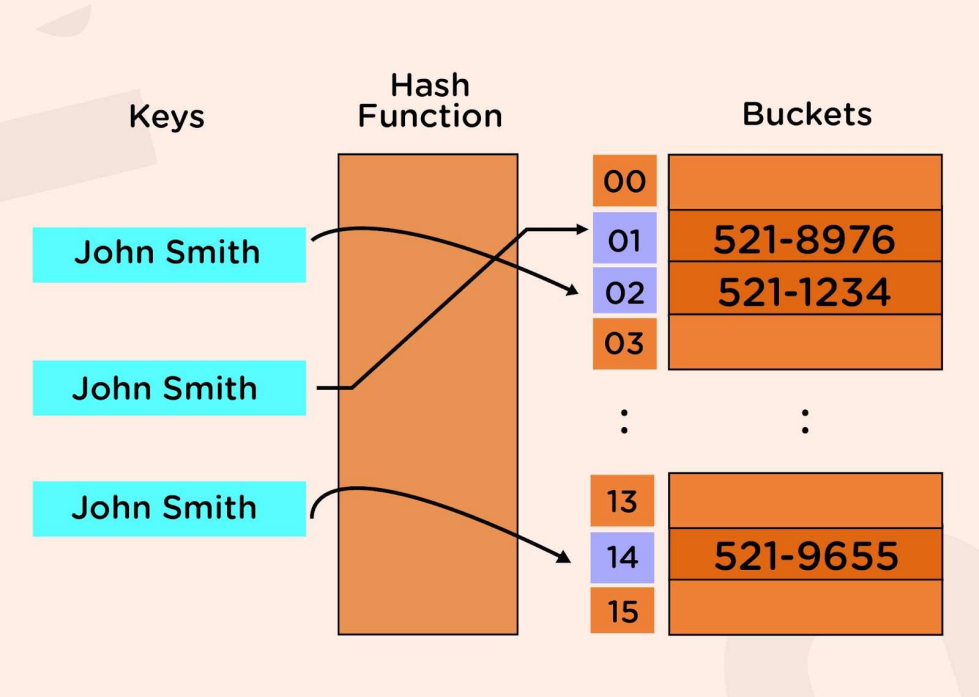
```
public ArrayList<Integer> removeDuplicates(ArrayList<Integer> a, int size){
      // to store the unique elements.
      ArrayList<Integer> output = new ArrayList<>();

      HashMap<Integer, Boolean> seen = new HashMap<>();
      for (int i=0; i<size; i++) {   // traversing the array
            if (seen.containsKey(a[i])) // using .containsKey() function to
                continue;       //check if the value has already occurred.
            else {
                seen.put(a[i], true);  // If not, then updating the map
                output.add(a[i]); // and inserting in map.
            }
      }
      return output;
}
```

Bucket Array and Hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Till now, we have seen that arrays are the fastest way to extract data as compared to other data structures as the time complexity of accessing the data in the array is O(1). So we will try to use them in implementing the hashmaps.

Now, we want to store the key-value pairs in an array, named as a **bucket array**. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a **hash function**. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following the image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using hashcode. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (% bucket_size).

One example of a hash code could be: (Example input: "abcd")

"abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0)
Where p is generally taken as a prime number so that they are well distributed.

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = "ab" and s2 = "cd". Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

Compression_function1 = 292 % 2 = 0

Compression_function2 = 298 % 2 = 0

This means they both lead to the same index 0.

This is known as a **collision**.

Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as **separate chaining**.

On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair over that index. If not, then will we find an alternate position for the same. To find the alternate position, we can use the following:

$h_i(a) = hf(a) + f(i)$

Where hf(a) is the original hash function, and f(i) is the i-th try over the hash function to obtain the final position $h_i(a)$.

To figure out this f(i), following are some of the techniques:

1. **Linear probing:** In this method, we will linearly probe to the next slot until we find the empty index. Here, f(i) = i.

2. **Quadratic probing:** As the name suggests, we will look for alternate i^2 positions ahead of the filled ones, i.e., $f(i) = i^2$.
3. **Double hashing:** According to this method, f(i) = i * H(a), where H(a) is some other hash function.

In practice, we generally prefer to use separate chaining over open addressing, as it is easier to implement and is also more efficient.

Let's now implement the hashmap of our own.

Hashmap implementation - Insert

As discussed earlier, we will be implementing separate chaining. We will be using value as a template and key as a string as we are required to find the hash code for the key. Taking key as a template will make it difficult to convert it using hash code.

Let's look at the code for the same.

```
public class MapNode<K, V> {
	K key;
	V value;
	MapNode<K, V> next;

	public MapNode(K key, V value) {
		this.key = key;
		this.value = value;
	}
}
--------------------------------------------------------------------------

class Map <K, V> {
	ArrayList<MapNode<K, V>> buckets;
	int size;
	int numBuckets;
	public Map() {
		numBuckets = 5;
		buckets = new ArrayList<>();
		for (int i = 0; i < numBuckets; i++) {
			buckets.add(null);
		}
	}
```

```
	private int getBucketIndex(K key) {
		int hashCode = key.hashCode();
		return hashCode % numBuckets;
	}

	public void insert(K key, V value) {
		int bucketIndex = getBucketIndex(key);
		MapNode<K, V> head = buckets.get(bucketIndex);
		while (head != null) {
			if (head.key.equals(key)) {
				head.value = value;
				return;
			}
			head = head.next;
		}
		head = buckets.get(bucketIndex);
		MapNode<K, V> newElementNode = new MapNode<K, V>(key , value);
		size++;
		newElementNode.next = head;
		buckets.set(bucketIndex, newElementNode);
		double loadFactor = (1.0*size)/numBuckets;
		if (loadFactor > 0.7) {
			rehash();
		}
	}
}
```

Hashmap implementation - Delete and search

Refer to the code below and follow the comments in it.

```
public V removeKey(K key) {
	int bucketIndex = getBucketIndex(key);
	MapNode<K, V> head = buckets.get(bucketIndex);
	MapNode<K, V> prev = null;
	while (head != null) {
		if (head.key.equals(key)) {
			size--;
			if (prev == null) {
				buckets.set(bucketIndex, head.next);
			} else {
				prev.next = head.next;
			}
			return head.value;
		}
		prev = head;
```

```
            head = head.next;
      }
      return null;
}

public V getValue(K key) {
      int bucketIndex = getBucketIndex(key);
      MapNode<K, V> head = buckets.get(bucketIndex);
      while (head != null) {
            if (head.key.equals(key)) {
                  return head.value;
            }
            head = head.next;
      }
      return null;
}
```

Time Complexity and Load Factor

Let's define specific terms before moving forward:

1. n = Number of entries in our map.
2. l = length of the word (in case of strings)
3. b = number of buckets. On average, each box contains (n/b) entries. This is known as **load factor** means b boxes contain n entries. We also need to ensure that the load factor is always less than 0.7, i.e.,

   (n/b) < 0.7, this will ensure that each bucket does not contain too many entries in it.
4. To make sure that load factor < 0.7, we can't reduce the number of entries, but we can increase the bucket size comparatively to maintain the ratio. This process is known as **Rehashing**.

This ensures that time complexity is on an average O(1) for insertion, deletion, and search operations each.

Rehashing

Now, we will try to implement the rehashing in our map. After inserting each element into the map, we will check the load factor. If the load factor's value is greater than 0.7, then we will rehash.

Refer to the code below for better understanding.

```
public int size() {
	return size;
}

public double loadFactor() {
	return (1.0 * size)/numBuckets;
}

private void rehash() {
	System.out.println("Rehashing: buckets"+ numBuckets+" size " + size);
	ArrayList<MapNode<K, V>> temp = buckets;
	buckets = new ArrayList<>();
	for (int i = 0; i < 2*numBuckets; i++) {
		buckets.add(null);
	}
	size = 0;
	numBuckets *= 2;
	for (int i = 0; i < temp.size(); i++) {
		MapNode<K, V> head = temp.get(i);
		while (head != null) {
			K key = head.key;
			V value = head.value;
			insert(key, value);
			head = head.next;
		}
	}
}
```

Note: While solving the problems, use the in-built hashmap only.