Hashing and HashMaps

Objectives

- Understand the setup and utility of hash based data structures
- Utilize and implementation of the Map interface.

For-Each Loop

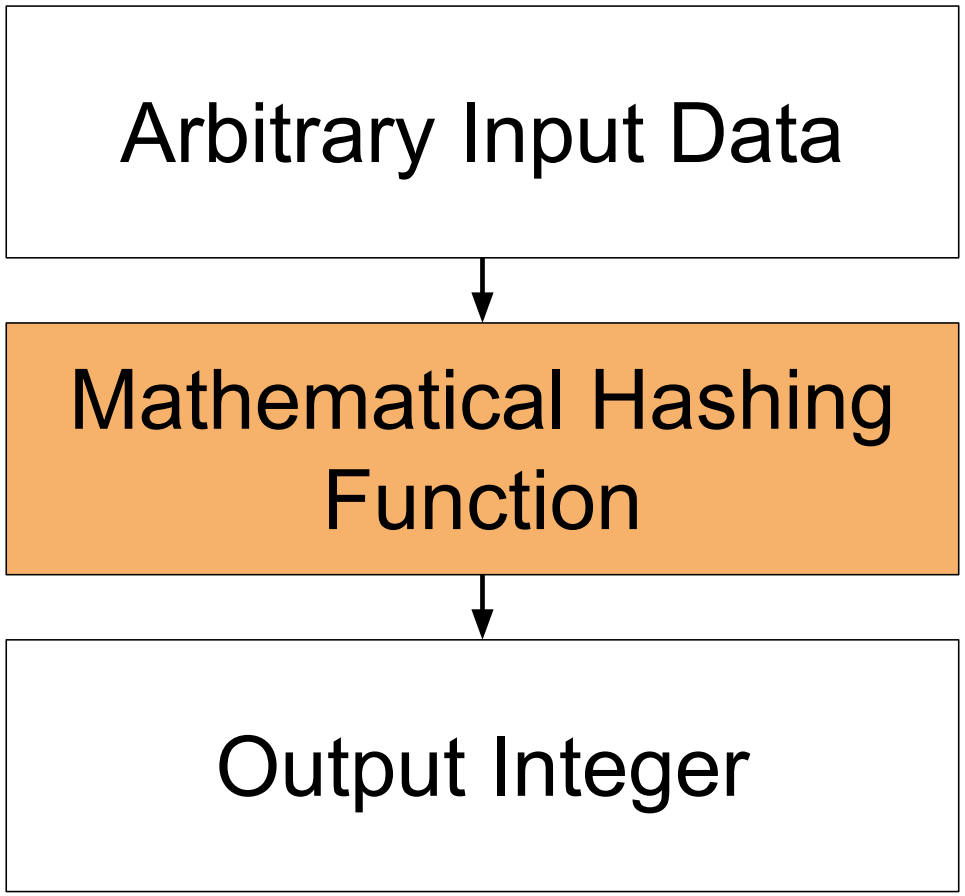
Imagine we have a variable named nums that is an ArrayList of integers. We have two ways of printing them.

```
for(int i = 0; i < nums.size(); i++){
  Integer num = nums.get(i);
  System.out.println(num);
}
```

```
for(Integer num: nums){
    System.out.println(num);
}
```

Both of the above code segments do the same thing

Hashing in Principle



Hashing Values in Java

```
String str1 = "Hello";
System.out.println(str1.hashCode());
```

```
69609650
```

```
Double num = 14.5;
System.out.println(num.hashCode());
```

```
1076690944
```

Key point: The HashValue of the values "Hello" and 14.5 are *always* the same (determinitic).

Hashing Custom Objects in Java

```
class RandomDataContainer<E>{
    E data;
    RandomDataContainer(E data){
        this.data = data;
    }
}
RandDataContainer<String> rdc = new
    RandDataContainer("Hello");
System.out.println(rdc.hashCode());
```

```
???
```

- **The Issue:** The output of this will change in between each run.
- How do we make it deterministic?

Hashing Custom Objects in Java

```
class RandomDataContainer<E>{
    E data;
    RandomDataContainer(E data){
        this.data = data;
    }

    @Override
    public int hashCode(){
        return data.hashCode();
    }
}
RandDataContainer<String> rdc = new RandDataContainer("Hello");
System.out.println(rdc.hashCode());
```

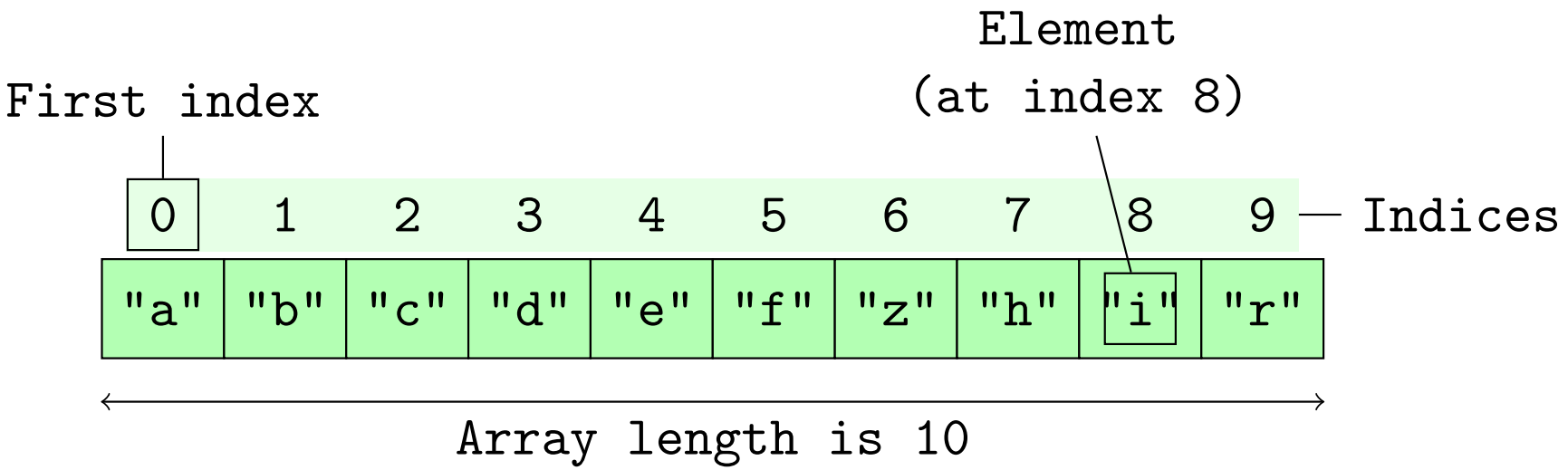
```
69609650
```

- **Key Point:** If we want an object's hashcode to be deterministic, manually override it and provide your own!

Object Hash Purpose

Key Point: The hashCode is used convert arbitrary objects into an integer.

Index Lookup vs Hash Lookup

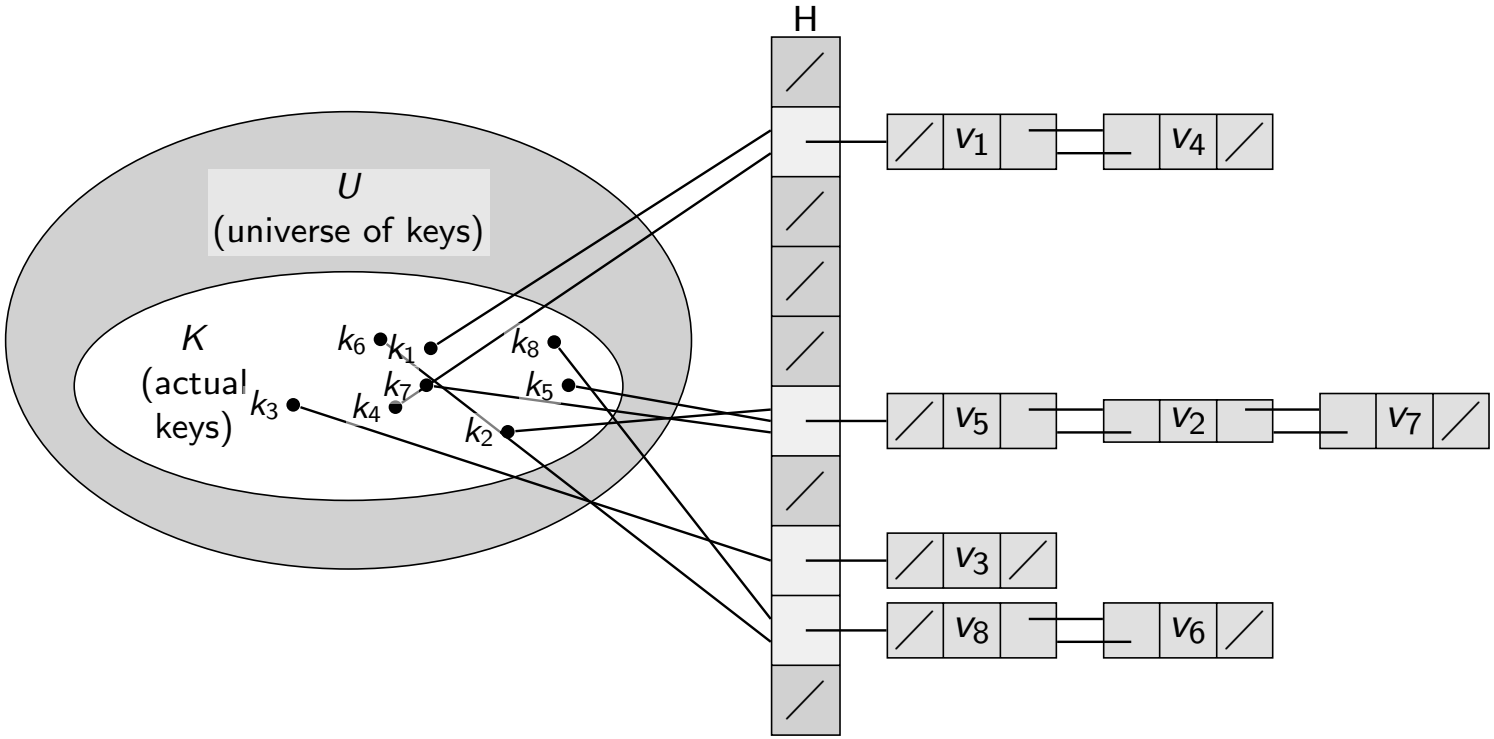


- Lookup time-complex *O(1)* if we know the index of the thing that we are looking for.
- If we have to search it's $O(N)$ if we use an array or $O(log_2(N)$ if we switch to a tree.
- We can to better!

Hashing to "Index"

1. Use hashing + more math to convert a "key" to an integer.
2. Use that integer to access an array!
3. Creates the notion of key-value pairs.
4. Key is the "index", value is the thing at that index.

Index Lookup vs Hash Lookup



- The hashing algorithm produces a finite set of keys: U
- Our hashtable will contain a subset of those: K
- k_n is a key that we use to index into a list.
- H is a list of lists where each item in the list is the hash code

Possible Uses

- Store phone numbers:
  - key: Name
  - value: Number
- Address book:
  - key: Name
  - value: Address
- Dictionary:
  - key: Word
  - value: Dictionary

Key Takeaway: We have key values pairs and we use to key to modify/access the value, just like an index in an array.

Map Interface Method

```
Map<K, V> map = // ..

Map<String, Integer> map = // ..
Map<Integer, List<String>> map = // ..
```

Map Interface Method

```
Map<K, V> map = // ..

Map<String, Integer> map = // ..
Map<Integer, List<String>> map = // ..
```

1. put(K key, V value): Puts a key-value pair into the map.
2. putAll(Map<K, V> map): Puts all of the key–value pairs into a map.

Map Interface Method

```
Map<K, V> map = // ..

Map<String, Integer> map = // ..
Map<Integer, List<String>> map = // ..
```

1. put(K key, V value): Puts a key-value pair into the map.
2. putAll(Map<K, V> map): Puts all of the key–value pairs into a map.
3. get(K key): Gets the value associated with a key in a map.
4. getOrDefault(K key, V defaultValue): Gets the value associated with a key. If that key doesn't exist, *returns the defaultValue*.

Map Interface Method

```
Map<K, V> map = // ..

Map<String, Integer> map = // ..
Map<Integer, List<String>> map = // ..
```

1. put(K key, V value): Puts a key-value pair into the map.
2. putAll(Map<K, V> map): Puts all of the key–value pairs into a map.
3. get(K key): Gets the value associated with a key in a map.
4. getOrDefault(K key, V defaultValue): Gets the value associated with a key. If that key doesn't exist, *returns the defaultValue*.
5. containsKey(K key): Returns true if the map contains the key and false otherwise.

Map Interface Method

```
Map<K, V> map = // ..

Map<String, Integer> map = // ..
Map<Integer, List<String>> map = // ..
```

1. put(K key, V value): Puts a key-value pair into the map.
2. putAll(Map<K, V> map): Puts all of the key–value pairs into a map.
3. get(K key): Gets the value associated with a key in a map.
4. getOrDefault(K key, V defaultValue): Gets the value associated with a key. If that key doesn't exist, *returns the defaultValue*.
5. containsKey(K key): Returns true if the map contains the key and false otherwise.
6. remove(K key): Removes the key value pair referenced by `key`.

Iterating over Keys and Values

```
//iterates over a maps keys
for(K key: map.keySet()){
    System.out.println(item);
}

//Iterativng over a map's values
for(V value: map.values()){
    System.out.println(item);
}
```

- map.values(): Gets you the collection of values present in the map.
- map.keys(): Gets you the collection of keys in the dictionary.

Iterating over Keys-Values

```
for(Map.Entry<K, V> item: map.entrySet()){
    System.out.println("Key:" + item.getKey());
    System.out.println("Value:" + item.getValue());
}
```

- entrySet(): Gives us a set of Map.Entry<K, V> pairs (i.e., Set<Map.Entry<K, V>).
- getKey(): Gets you the key from a Map.Entry.
- getValue(): Gets you the value from a Map.Entry.