

Solutions

5.1 Sets and Maps as Abstractions

Reference example

```
Set<String> names = new HashSet<String>();
names.add("Ada"); names.add("Ada");
Map<String, Integer> counts = new HashMap<String, Integer>();
counts.put("Ada", 2); counts.put("Ada", 3);
System.out.println(names.size());
System.out.println(counts.get("Ada"));
```

5.1.1 - Explain

Choose a set or a map for unique visitor identifiers and for visit counts by identifier.

Use a set for unique membership. Use a map from identifier to count when each key needs an associated total.

5.1.2 - Write code

Count occurrences of a, b, a and print the count of a.

Update the value for each encountered key.

```
Map<String,Integer> counts = new HashMap<String,Integer>();
for (String word : new String[]{"a","b","a"})
    counts.put(word, counts.getOrDefault(word,0)+1);
System.out.println(counts.get("a"));
```

Expected output

2

Solutions

5.2 Hash Functions

Reference example

```
int capacity = 7;  
System.out.println(Math.floorMod(17, capacity));  
System.out.println(Math.floorMod(24, capacity));
```

5.2.1 - Explain

Do equal hash values prove two keys are equal?

No. Collisions are possible. A hash-based collection must use its equality rule as well as the hash.

5.2.2 - Explain

Why must a key equality rule and its hash calculation agree?

Equal keys must produce compatible hashes so lookups search the correct location. Mutating a stored key in a way that changes those properties can break retrieval.

Solutions

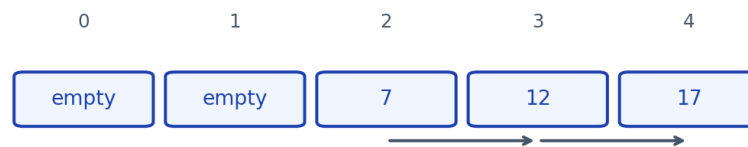
5.3 Collision Resolution

Reference example

Capacity: 5
Keys inserted: 7, 12, 17
Bucket index: $\text{key} \bmod 5$
All three initially map to bucket 2

Linear probing after three insertions

All keys initially hash to index 2



Probe to later available slots, using the same rule for lookup.

5.3.1 - Explain

Under linear probing in an empty capacity-5 table, where do 7, 12, and 17 land?

They initially hash to 2. With linear probing, they occupy indices 2, 3, and 4 in that insertion order.

5.3.2 - Write code

Insert 7, 12, and 17 into an empty capacity-5 table using linear probing. Use null to mark an unused slot and print the table.

All three hash to 2; wrap around with remainder if a later position is needed.

```
Integer[] table = new Integer[5];
for (int key : new int[]{7,12,17}) {
    int index = Math.floorMod(key, table.length);
    while (table[index] != null) index = (index + 1) % table.length;
    table[index] = key;
}
System.out.println(Arrays.toString(table));
```

Expected output

[null, null, 7, 12, 17]

Solutions

5.4 Load Factor and Resizing

Reference example

Entries: 6
Buckets: 8
Load factor: $6 / 8 = 0.75$
After expanding to 16 buckets: entries remain 6

5.4.1 - Explain

Find the load factor for 9 entries in 12 buckets, then after expansion to 24 buckets.

The load factors are $9/12 = 0.75$ and $9/24 = 0.375$.

5.4.2 - Explain

Why is copying entries to the same numeric indices insufficient when a table changes capacity?

Bucket assignment depends on capacity. Recompute the destination under the new table size so lookup and insertion remain consistent.