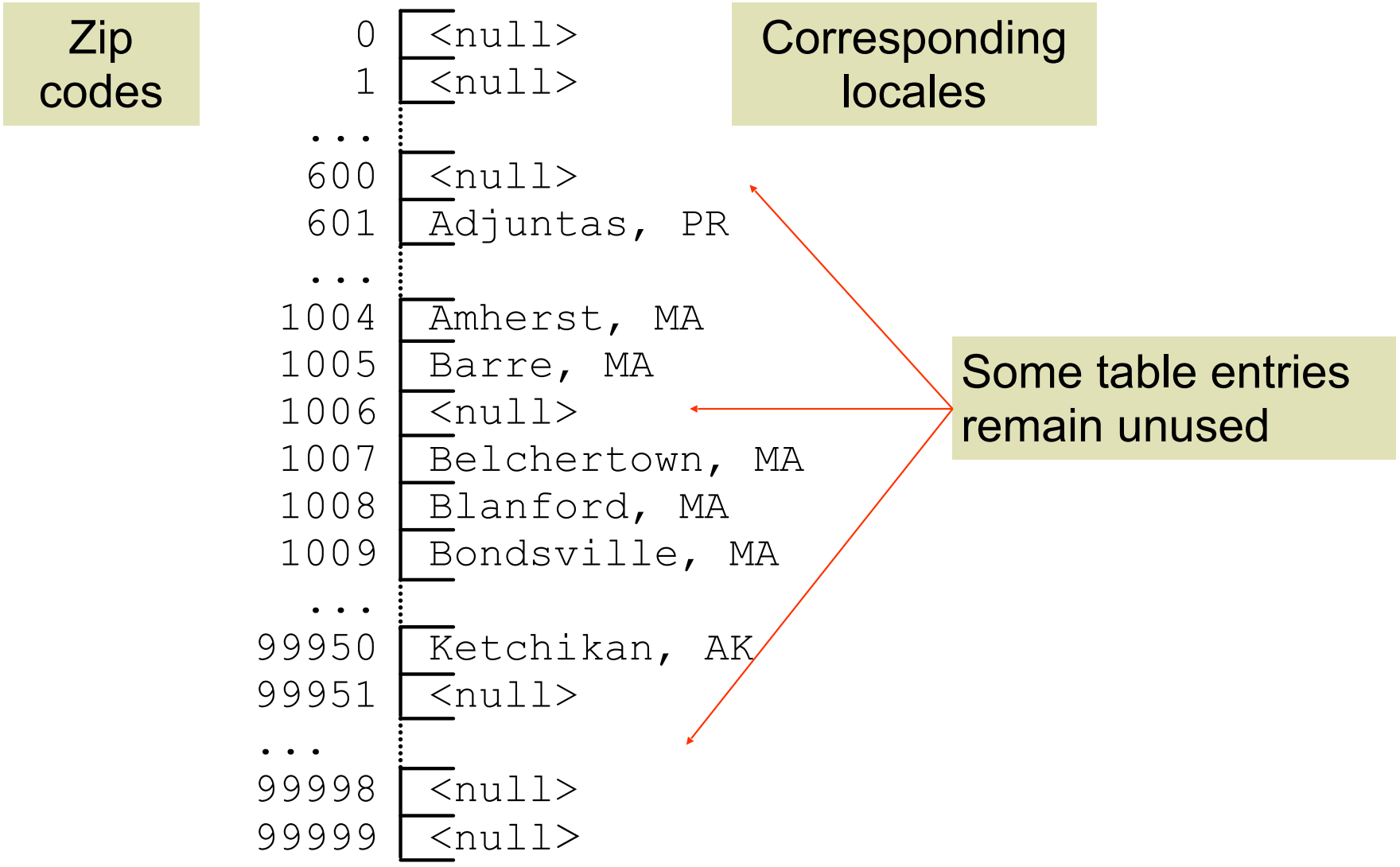
Lookup Tables

- A lookup table is an array that helps to find data very quickly.
- The array stores references to data records (or some values).
- A data record is identified by some key.
- The value of a key is directly translated into an array index using a simple formula.

Lookup Tables (cont'd)

- Only one key can be mapped onto a particular index (no *collisions*).
- The index that corresponds to a key must fall into the valid range (from 0 to array.length−1).
- Access to data is "instantaneous" (*O*(1)).

Lookup Tables: Example 1

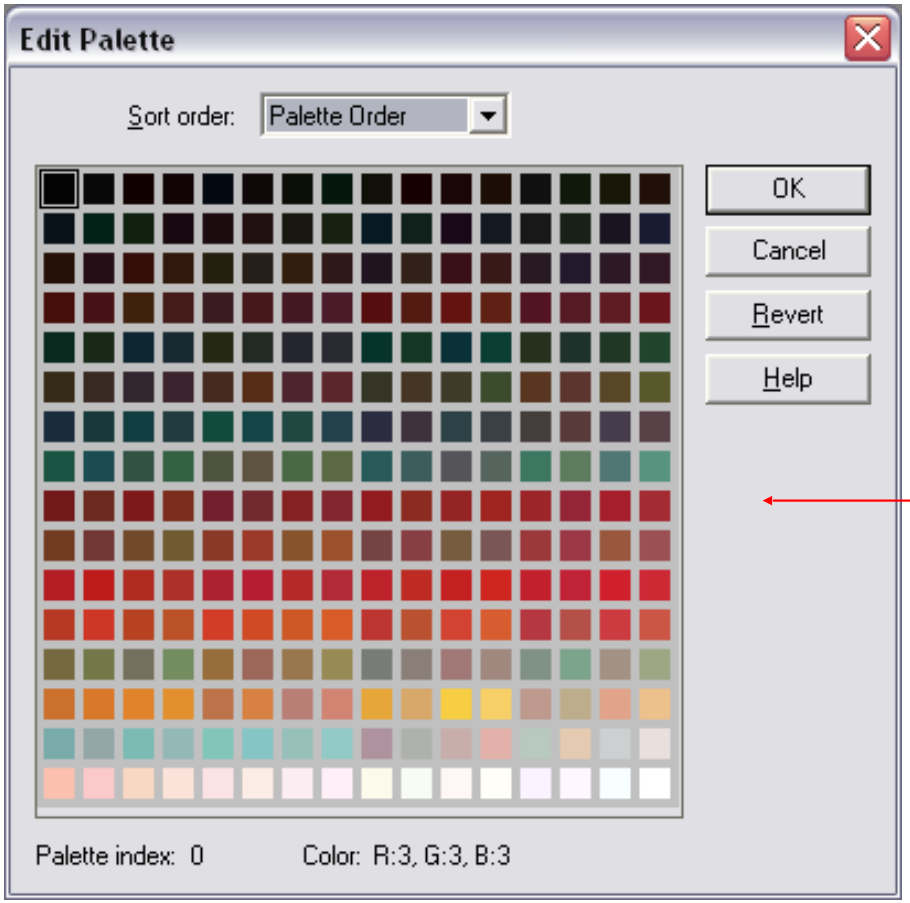


Lookup Tables: Example 2

```
private static final int [ ]  n_thPowerOf3 =
        { 1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683 };
...

   // precondition: 0 <= n < 10
   public int powOf3 (int n)
   {
       return n_thPowerOf3 [ n ];
   }
```

Lookup Tables: Example 3



256 colors used in a particular image; each of the palette entries corresponds to a triplet of RGB values

Applications of Lookup Tables

- Data retrieval
- Data compression and encryption
- Tabulating functions
- Color mapping

Hash Tables

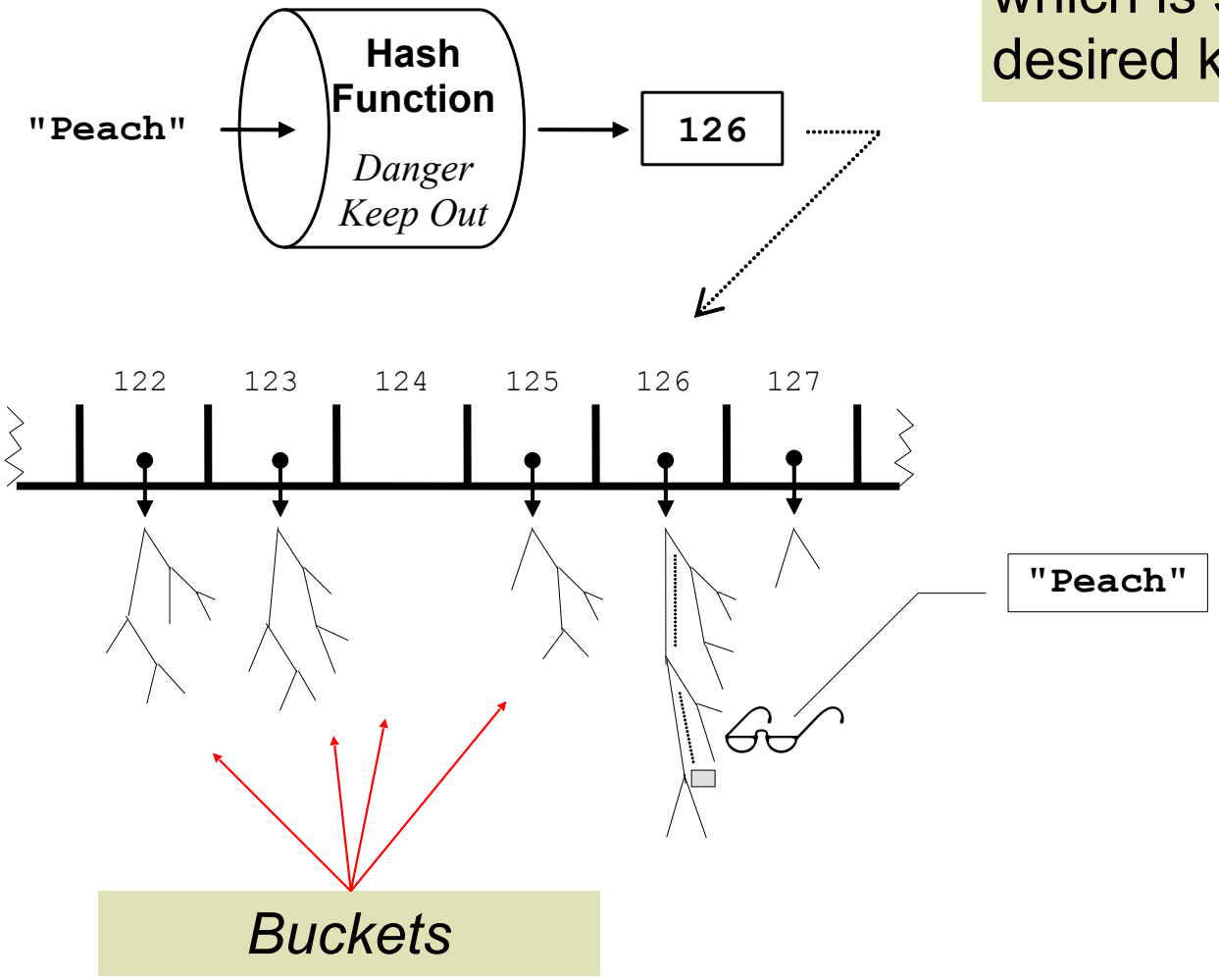
- A hash table is similar to a lookup table.
- The value of a key is translated into an array index using a *hash function*.
- The index computed for a key must fall into the valid range.
- The hash function can map different keys onto the same array index — this situation is called a *collision*.

Hash Tables (cont'd)

- The hash function should map the keys onto the array indices randomly and uniformly.
- A well-designed hash table and hash function minimize the number of collisions.
- There are two common techniques for resolving collisions: *chaining* and *probing*.

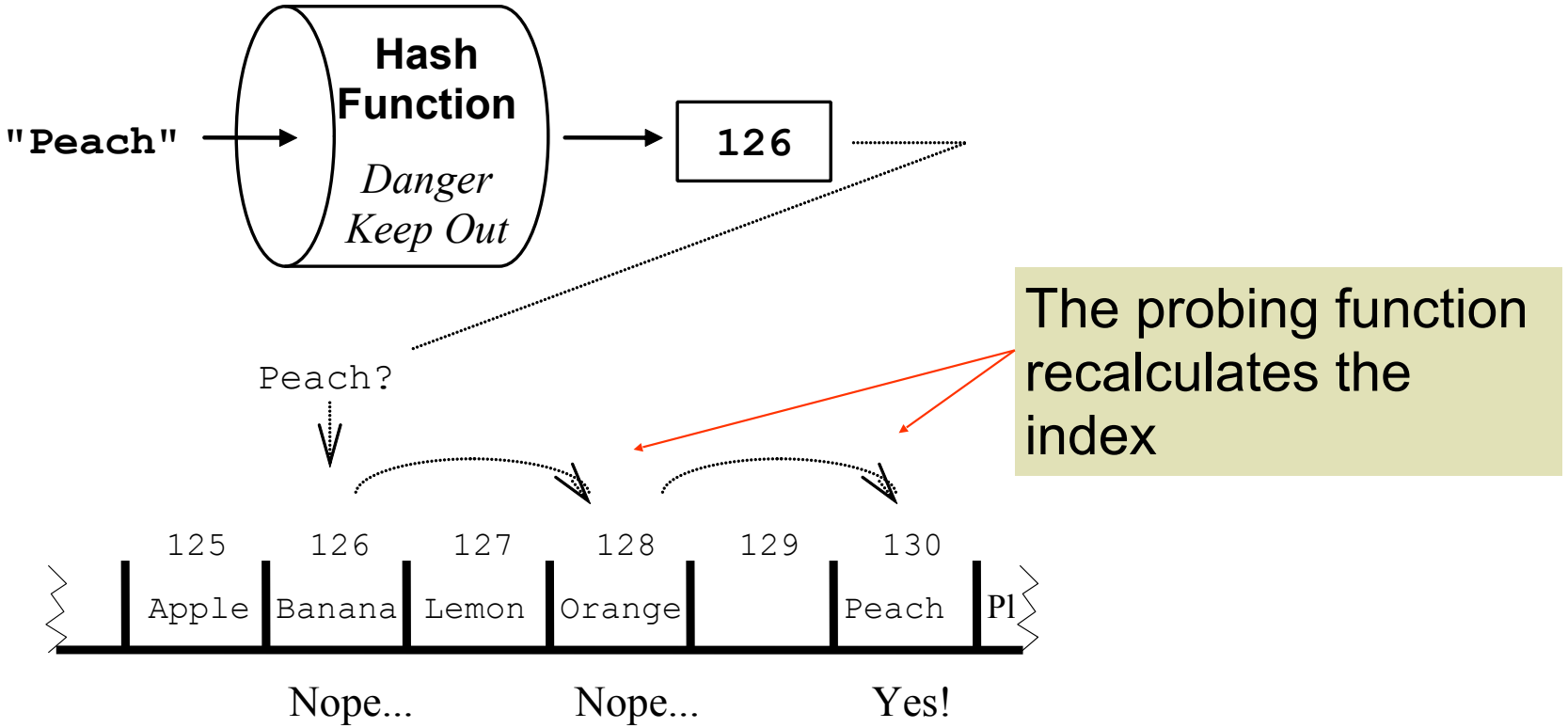
Chaining

Each element in the array is itself a collection, called a *bucket* (a list or a BST), which is searched for the desired key



Probing

If the place where we want to store the key is occupied by a different key, we store the former in another location in the same array, computed using a certain probing formula



The probing function recalculates the index

java.util.HashSet<*E*> and **java.util.HashMap<*K*,*V*>** Classes

- These classes implement the **Set<*E*>** and **Map<*K*,*V*>** interfaces, respectively, using hash tables (with chaining).
- This implementation may be more efficient than **TreeSet** and **TreeMap**.

hashCode Examples

- For String:

$$hashCode = s_0 \cdot 31^{n-1} + s_1 \cdot 31^{n-2} + \ldots + s_{n-1}$$

  - (where s_i is Unicode for the *i*-th character in the string)

- For Person:

```
public int hashCode ( )
 {
     return getFirstName( ).hashCode( ) +
                getLastName( ).hashCode( );
}
```

Consistency

- **HashSet** / **HashMap** first use **hashCode**, then **equals**.
- **TreeSet** / **TreeMap** use only **compareTo** (or a comparator)
- For consistent performance, these methods should agree with each other:
  - x.equals (y) $\Leftrightarrow$ x.compareTo (y) == 0
  - x.equals (y) $\Rightarrow$ x.hashCode( ) == y.hashCode( )