Implementation of Linear Probing (continued)

- Helping method for locating index:

```
private int findIndex(long key)
 // return -1 if the item with key 'key' was not found
 {
   int index = h(key);
   int probe = index;
   int k = 1; // probe number
   do {
     if (table[probe]==null) {
       // probe sequence has ended
       break;
     }
     if (table[probe].getKey()==key)
       return probe;
     probe = (index + step(k)) % table.length; // check next slot
     k++;
   } while (probe!=index);

   return -1; // not found
 }
```

Implementation of Linear Probing (continued)

- Find and Deleting the item:

```
public T find(long key)
 {
   int index = findIndex(key);
   if (index>=0)
     return (T) table[index];
   else
     return null; // not found
 }

 public T delete(long key)
 {
   int index = findIndex(key);
   if (index>=0) {
     T item = (T) table[index];
     table[index] = AVAILABLE; // mark available
     return item;
   } else
     return null; // not found
 }
```

Open addressing: Quadratic Probing

- Designed to prevent *primary clustering*.
- It does this by increasing the step by increasingly large amounts as more probes are required to insert a record. This prevents clusters from building up.
  - In quadratic probing the step is equal to the *square* of the probe number.
  - With linear probing the step values for a sequence of probes would be {1, 2, 3, 4, etc}. For quadratic probing the step values would be {1, 2^2, 3^2, 4^2, etc}, i.e. {1, 4, 9, 16, etc}.
- **Disadvantage of this method:**
  - After a number of probes the sequence of steps repeats itself (remember that the step will be probe number^2 *mod* the size of the hash table). This repetition occurs when the probe number is roughly half the size of the hash table.
  - **Secondary clustering**.

Open addressing: Quadratic Probing

- **Disadvantage of this method:**
  - After a number of probes the sequence of steps repeats itself. => It fails to insert a new item even if there is still a space in the array.
  - **Secondary clustering:** the sequence of probe steps is the same for any insertion. Secondary clustering refers to the increase in the probe length (that is the number of probes required to find a record) for records where *collisions* have occurred (the keys are mapped to the **same** value). Note that this is not as large a problem as *primary clustering*.

- However, it is important to realize that in practice these two issues are not significant, given a large hash table and a good hash function it is extremely unlikely that these issues will affect the performance of the hash table, unless it becomes nearly full.

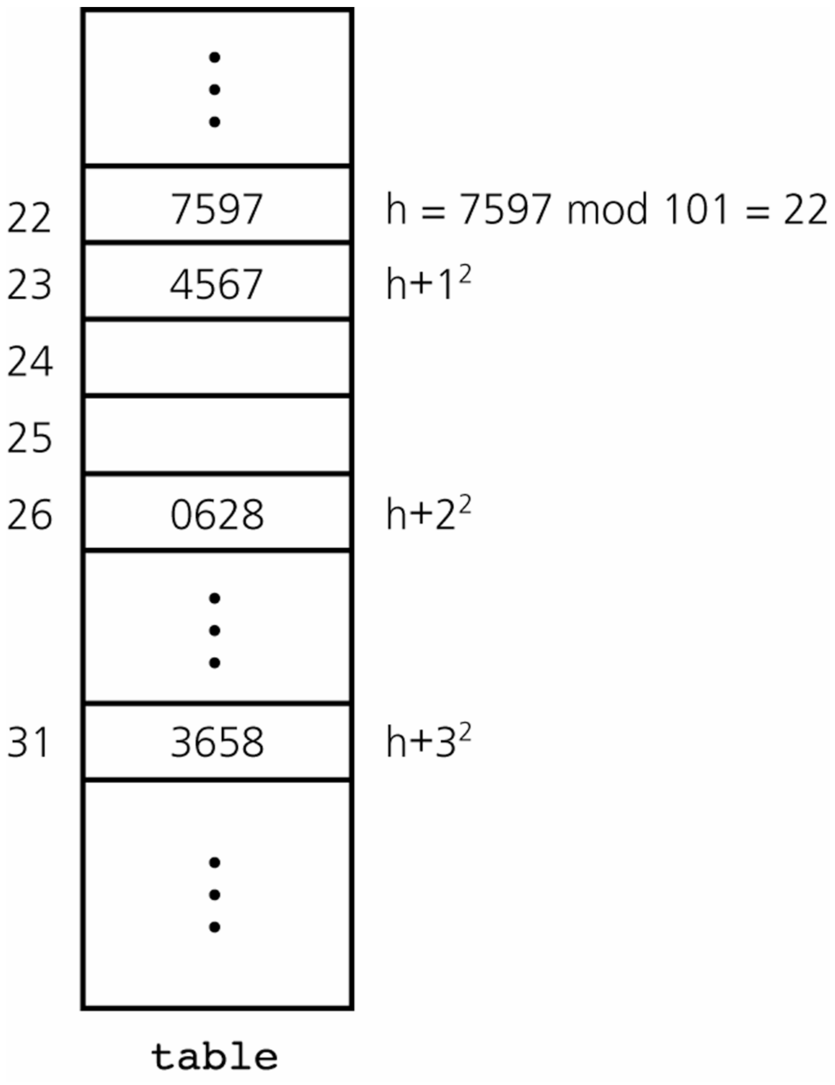

Figure: Quadratic probing with $h(x) = x \bmod 101$

Implementation

- It’s enough to modify the step helping method:

```
private int step(int k) // step function
  {
      return k*k // quadratic probing
  }
```

Open addressing: Double Hashing

- Double hashing aims to avoid both *primary* and *secondary clustering* and is guaranteed to find a free element in a hash table as long as the table is not full. It achieves these goals by calculating the step value using a **second hash function** *h'*.

$$step(k) = k.h'(key)$$

- This new hash function *h'* should:
  - be different from the original hash function (remember that it was the original hash function that resulted in the collision in the first place) and,
  - not result in zero (as original index + 0 = original index)

Open addressing: Double Hashing

- The second hash function is usually chosen as follows:

$$h'(key) = q - (key \% q),$$

where q is a prime number q<N (N is the size of the array).

- **Remark**: It is important that the size of the hash table is a ***prime*** number if double hashing is to be used. This guarantees that successive probes will (eventually) try every index in the hash table before an index is repeated (which would indicate that the hash table is full).
  - For other hashings (and for q) we want to use prime numbers to eliminate existing patterns in the data.

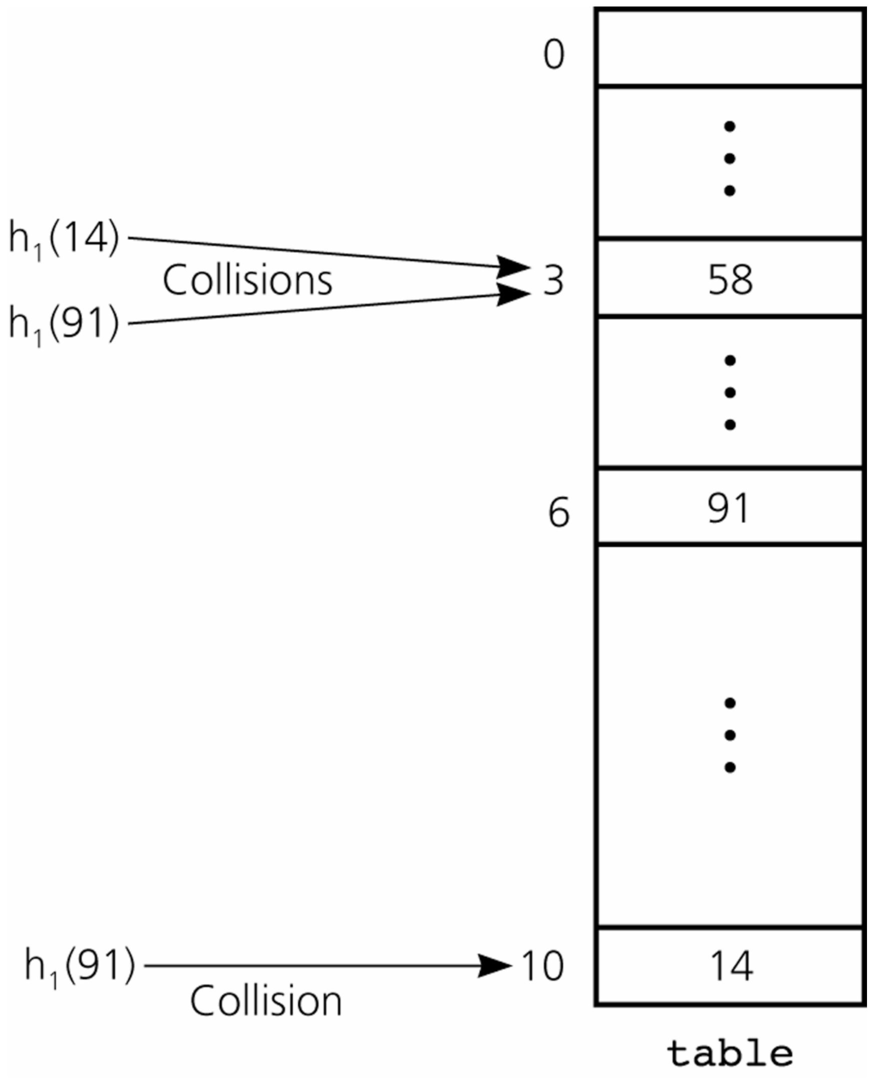

Figure: Double hashing during the insertion of 58, 14, and 91

Double Hashing – Implementation

```
public class DoubleHashTable<T extends KeyedItem>
implements HashTableInterface<T>
{
  private KeyedItem[] table;
  // special values: null = EMPTY, T with key=0 = AVAILABLE
  private static KeyedItem AVAILABLE = new KeyedItem(0);
  private int q; // should be a prime number

  public DoubleHashTable(int size,int q)
  // size: should be a prime number;
  //       recommended roughly twice bigger
  //       than the expected number of elements
  // q: recommended to be a prime number, should be smaller
   than size
  {
    table = new KeyedItem[size];
    this.q=q;
  }
```

Double Hashing – Implementation

```
private int h(long key) // hash function
 // return index
 {
   return (int)(key % table.length); // typecast to int
 }

 private int hh(long key) // second hash function
 // return step multiplicative constant
 {
   return (int)(q - key%q);
 }

 private int step(int k,long key) // step function
 {
   return k*hh(key);
 }
```

Double Hashing – Implementation

- Call `step(k,item.getKey())`instead of `step(k)`

```
public void insert(T item) throws HashTableFullException
 {
   int index = h(item.getKey());
   int probe = index;
   int k = 1;
   do {
     if (table[probe]==null || table[probe]==AVAILABLE) {
       // this slot is available
       table[probe] = item;
       return;
     }
     probe = (index + step(k,item.getKey())) % table.length; // check
  next slot
     k++;
   } while (probe!=index);
   throw new HashTableFullException("Hash table is full.");
 }
```

Open addressing performance

- The performance of a hash table depends on the **load factor** of the table.
- The **load factor** α is the *ratio of the number of data items to the size of the array*.
- Of the three types of open addressing *double hashing* gives the **best** performance.
- Overall, open addressing works very well up to load factors of around **0.5** (when 2 probes on average are required to find a record). For load factors greater than 0.6 performance declines dramatically.

Rehashing

- If the load factor goes over the safe limit, we should increase the size of the hash table (as for dynamic arrays). This process is called **rehashing**.
- *Comments:*
  - we cannot just double the size of the table, as the size should be a prime number;
  - it will change the main hash function
  - it's not enough to just copy items
- Rehashing will take time O(N)

Dealing with Collisions (2nd approach): Separate Chaining

- In separate chaining the hash table consists of an array of **lists**.
- When a collision occurs the new record is added to the list.
- Deletion is straightforward as the record can simply be removed from the list.
- Finally, separate chaining is less sensitive to load factor and it is normal to aim for a load factor of around 1 (but it will work also for load factors over 1).

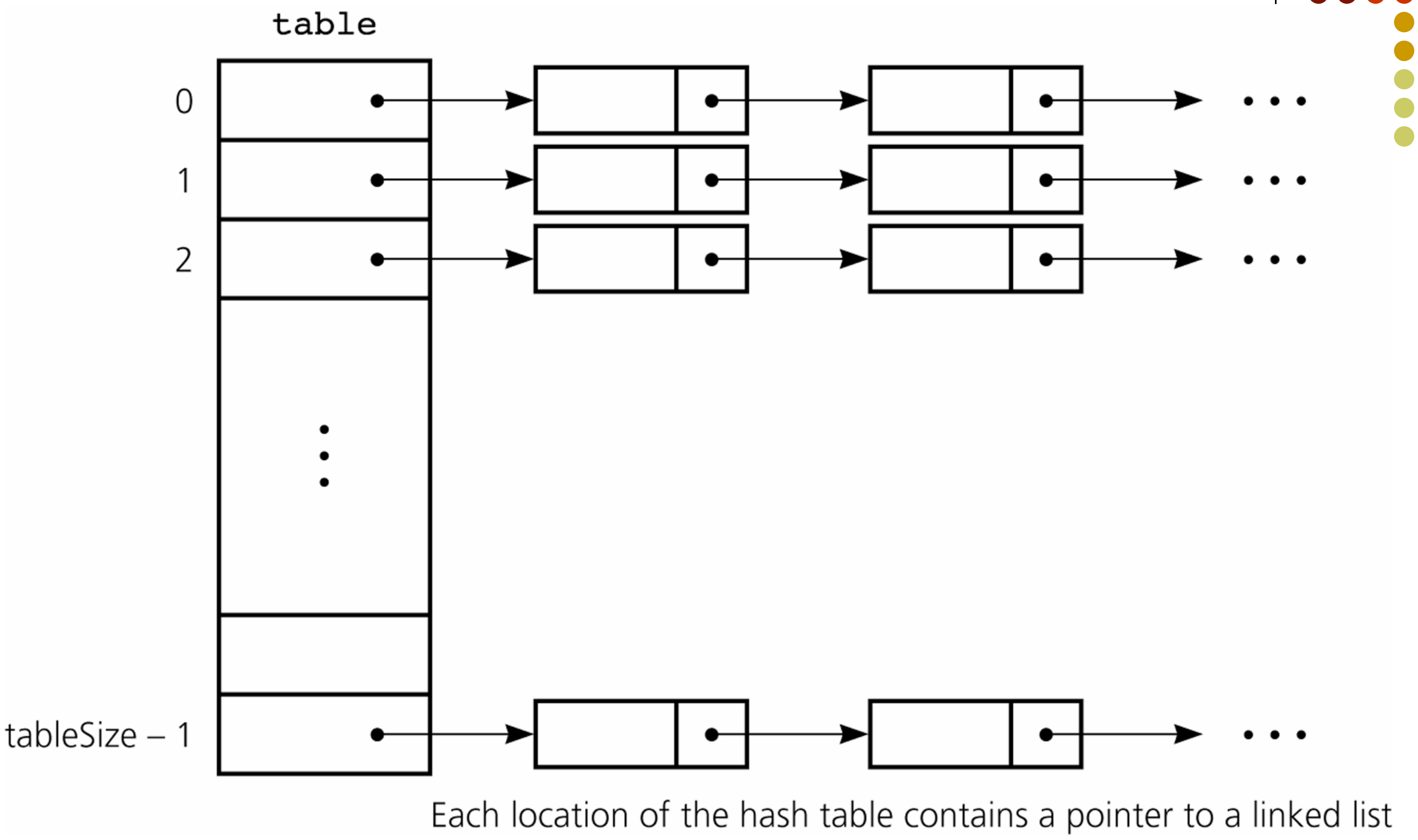
table
0
1
2
tableSize – 1
Each location of the hash table contains a pointer to a linked list

Figure: Separate chaining (using linked lists).

If array-based implementation of list is used: they are called **buckets**.