Hash Tables

Comparison-Based Search

- Linear Search (O(n))
- Binary Search (O(logn))

Direct Mapping

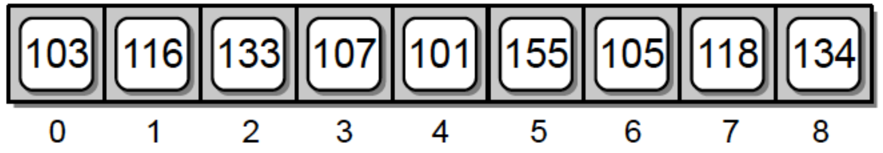

Figure 11.1: A collection of product codes stored in an array.

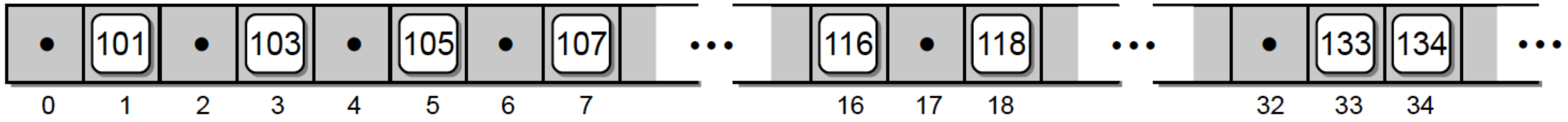

Figure 11.2: Storing a collection of product codes by direct mapping.

Hashing

- What if the key can be any integer value?
- **Hashing** is the process of mapping a search key to a limited range of array indices with the goal of providing direct access to the keys.
- The keys are stored in an array called a **hash table** and a **hash function** is associated with the table.

Hash Function

- Hash function converts or maps the search keys to specific entries in the table.
- Suppose we have the following set of keys:

  765, 431, 96, 142, 579, 226, 903, 388

and a hash table, T, containing M = 13 elements. We can define a simple hash function h() that maps the keys to entries in the hash table:

h(key) = key % M

Hash Function

- Applying the hash function to key 765 yields a result of 11 (765%13 = 11), which indicates 765 should be stored in element 11 of the hash table.
- Likewise, if we apply the hash function to the next four keys in the list, we find:

h(431) => 2	h(96) => 5
h(142) => 12	h(579) => 7

- all of which are unique index values.

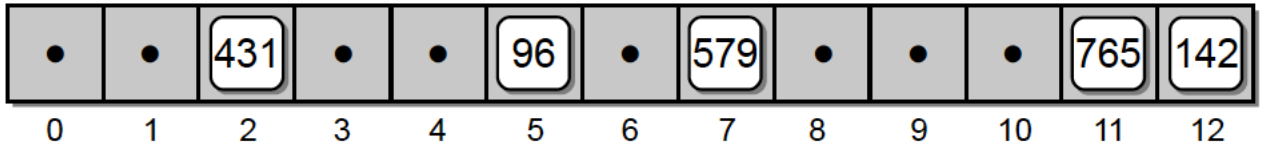

Figure 11.3: Storing the first five keys in the hash table.

Collision

- Consider what happens when we attempt to add key 226 to the hash table.
- The hash function maps this key to entry 5, but that entry already contains key 96.
- The result is a **collision**, which occurs when two or more keys map to the same hash location.

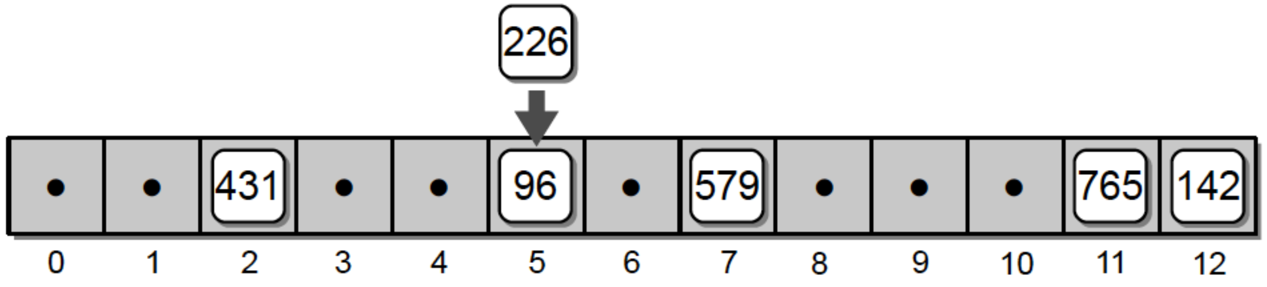

Figure 11.4: A collision occurs when adding key 226.

Linear Probing

- If two keys map to the same table entry, we must resolve the collision by **probing** the table to find another available slot.
- The simplest approach is to use a **linear probe**, which examines the table entries in sequential order starting with the first entry immediately following the original hash location.

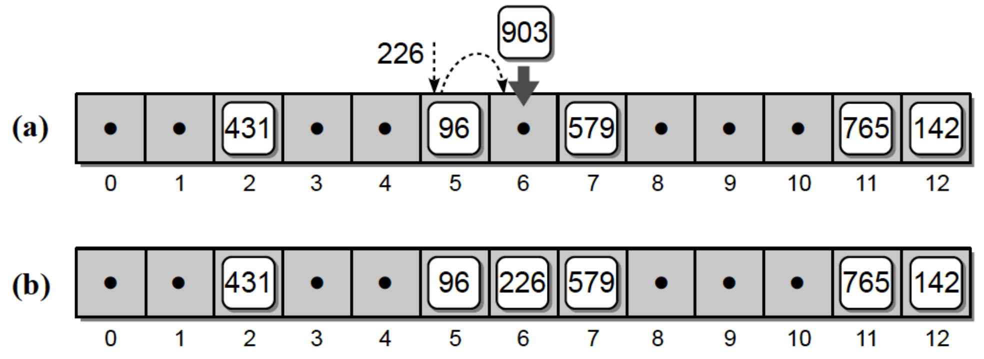

Figure 11.5: Resolving a collision for key 226 requires adding the key to the next slot.

Linear Probing

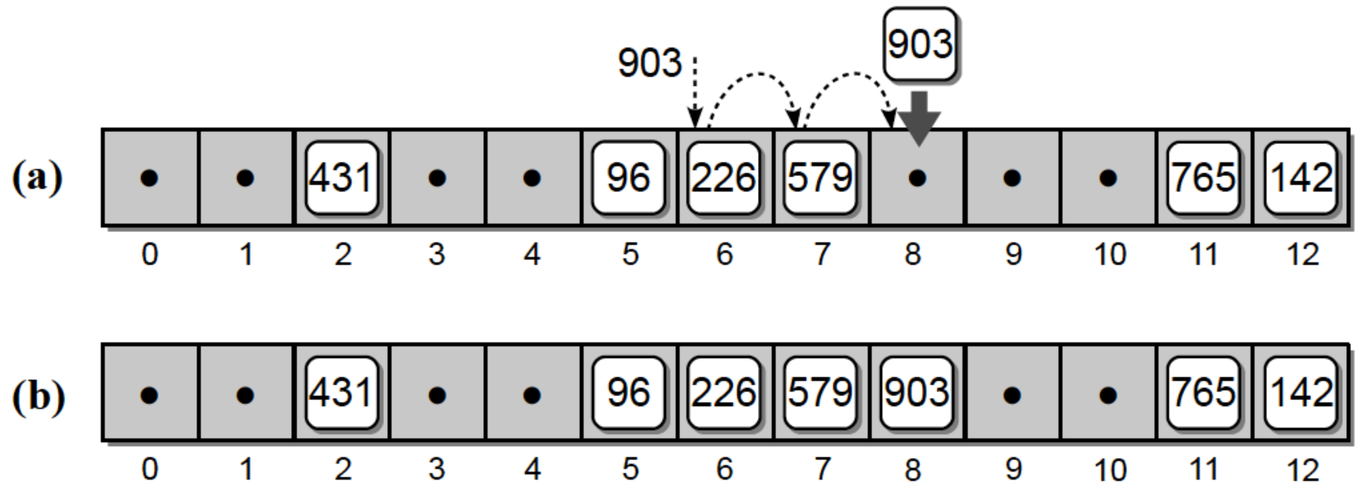

Figure 11.6: Adding key 903 to the hash table: (a) performing a linear probe; and (b) the result after adding the key.

Linear Probing

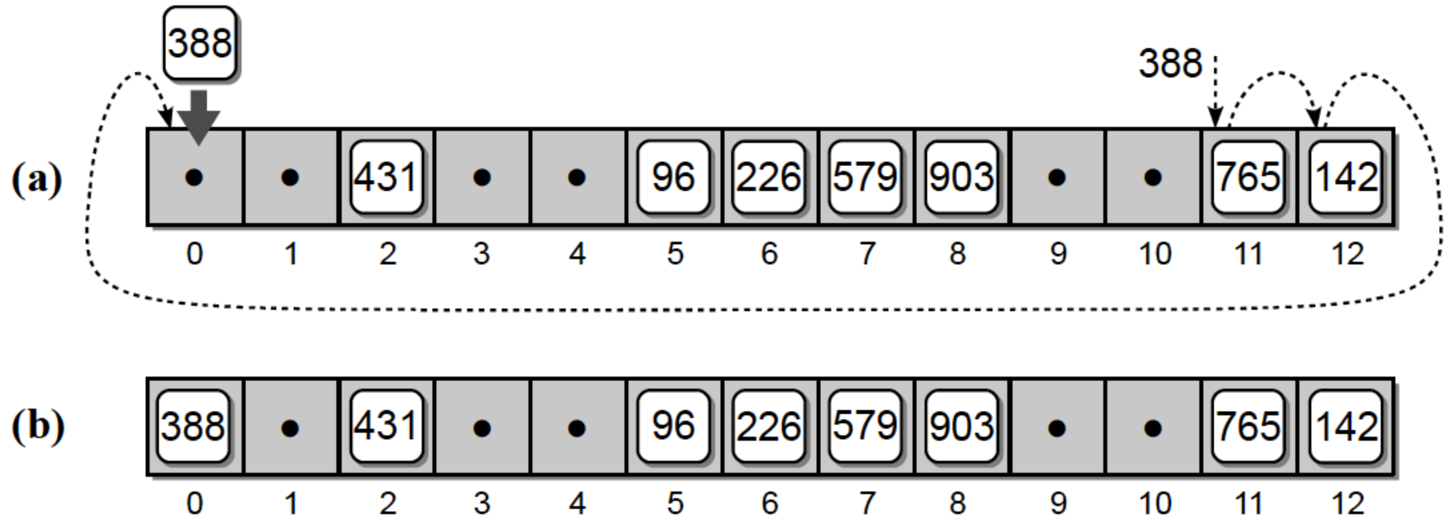

Figure 11.7: Adding key 388 to the hash table: (a) performing a linear probe; and (b) the result after adding the key.

Searching

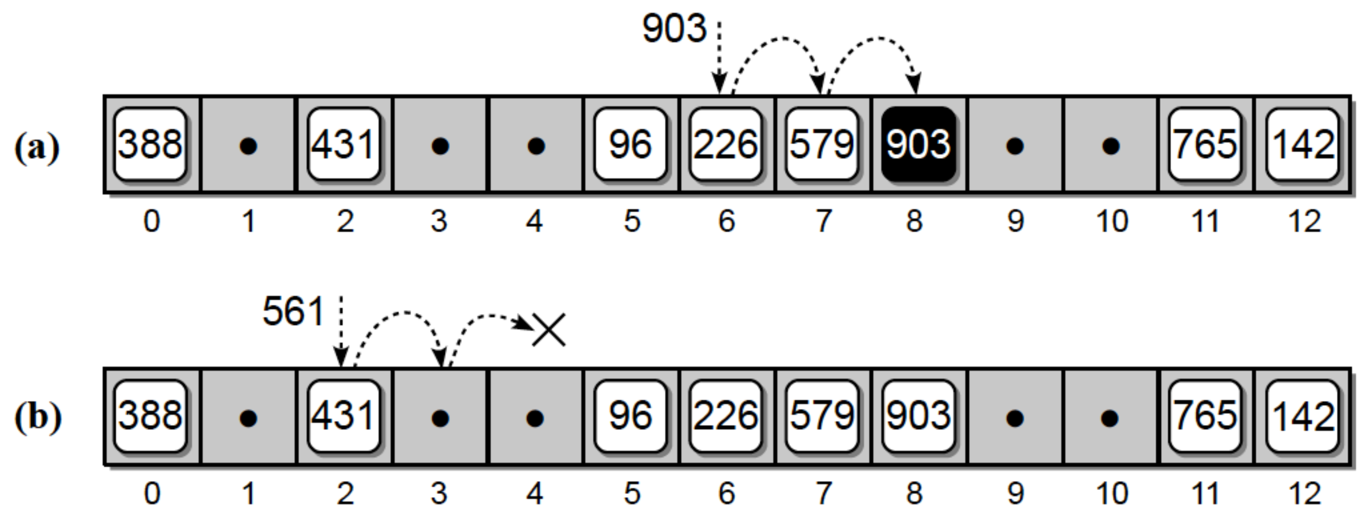

Figure 11.8: Searching the hash table: (a) a successful search for key 903 and (b) an unsuccessful search for key 561.

Deleting

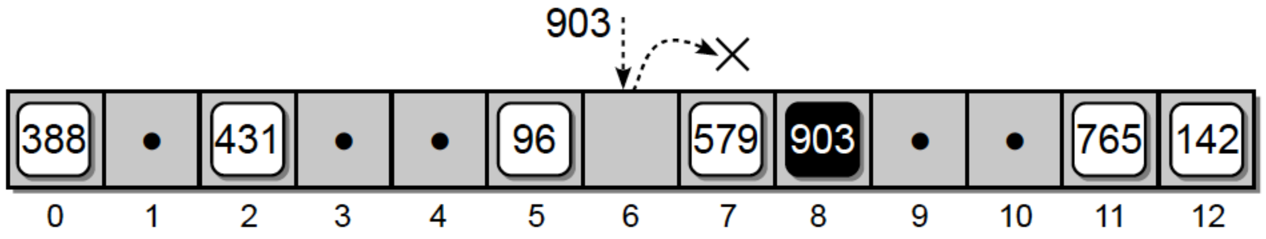

Figure 11.9: Incorrect deletion from the hash table.

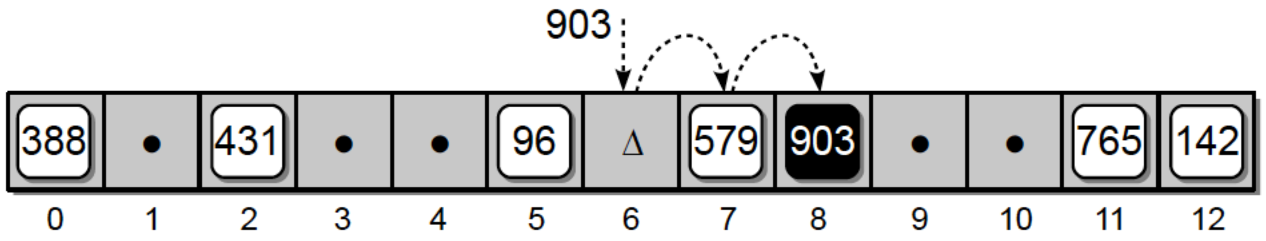

Figure 11.10: The correct way to delete a key from the hash table.

Clustering

- As more keys are added to the hash table, more collisions are likely to occur.
- Since each collision requires a linear probe to find the next available slot, the keys begin to form clusters.
- As the clusters grow larger, so too does the probability that the next key added to the table will result in a collision.
- As the clusters grow larger, so too does the length of the search needed to find the next available slot.
- We can reduce the amount of primary clustering by changing the technique used in the probing.

Modified Linear Probe

- When probing to find the next available slot, a loop is used to iterate through the table entries.
- The order in which the entries are visited form a probe sequence.
- The linear probe searches for the next available slot by stepping through the hash table entries in sequential order.
- The next array slot in the probe sequence can be represented as an equation:

slot = (home + i) % M

where i is the ith probe in the sequence, i = 1; 2; : : :M - 1.

home is the home position, which is the index to which the key was originally

mapped by the hash function.

Modified Linear Probe

- The use of the linear probe resulted in six collisions in our hash table of size M = 13:

h(765) => 11	h(579) => 7
h(431) => 2	h(226) => 5 => 6
h(96) => 5	h(903) => 6 => 7 => 8 => 9
h(142) => 12	h(388) => 11 => 12 => 0

- When the keys are inserted in the order:

  765, 431, 96, 142, 579, 226, 903, 388

Modified Linear Probe

- We can improve the linear probe by skipping over multiple elements instead of probing the immediate successor of each element.
- This can be done by changing the step size in the probe equation to some fixed constant c:

  slot = (home + i*c) % M

- Suppose we use a linear probe with c = 3 to build the hash table using the same set of keys. This results in only two collisions as compared to six when c = 1

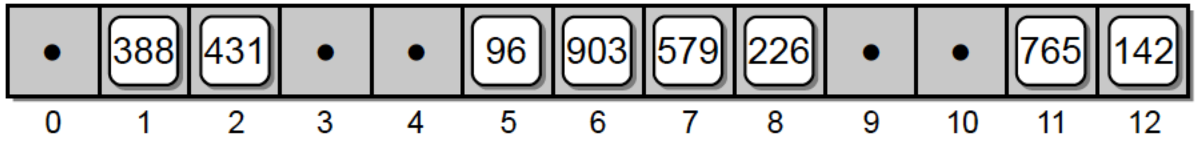

Figure 11.11: The hash table using a linear probe with $c = 3$.

Quadratic Probing

- A better approach for reducing primary clustering is with the use of quadratic probing, which is specified by the equation:

  slot = (home + i^2) % M

- Quadratic probing eliminates primary clustering by increasing the distance between each probe in the sequence.

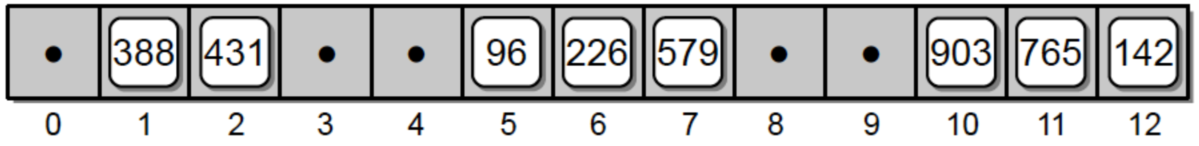

Figure 11.12: The hash table using a quadratic probe.

Quadratic Probing

- While the number of collisions has increased, the primary clustering has been reduced.
- In practice, quadratic probing typically reduces the number of collisions but introduces the problem of secondary clustering.
- **Secondary clustering** occurs when two keys map to the same table entry and have the same probe sequence.
- For example, if we were to add key 648 to our table, it would hash to slot 11 and follow the same probe sequence as key 388.
- Finally, there is no guarantee the quadratic probe will visit every entry in the table. But if the table size is a prime number, at least half of the entries will be visited.

Double Hashing

- Secondary clustering occurs because the probe equation is based solely on the original hash slot.
- A better approach for reducing secondary clustering is to base the probe sequence on the key itself.
- In double hashing, when a collision occurs, the key is hashed by a second function and the result is used as the constant factor in the linear probe:

  slot = (home + i * hp(key)) % M

Double Hashing

- While the step size remains constant throughout the probe, multiple keys that map to the same table entry will have different probe sequences.
- To reduce clustering, the second hash function should not be the same as the main hash function and it should produce a valid index in the range $0 < c < M$.
- A simple choice for the second hash function takes the form:

    hp(key) = 1 + key % P

  where P is some constant less than M.

Double Hashing

- Suppose we define a second hash function:

  hp(key) = 1 + key % 8

  and use it with double hashing to build a hash table from our sample keys.

- To ensure every table entry is visited during the probing, the table size must be a prime number.

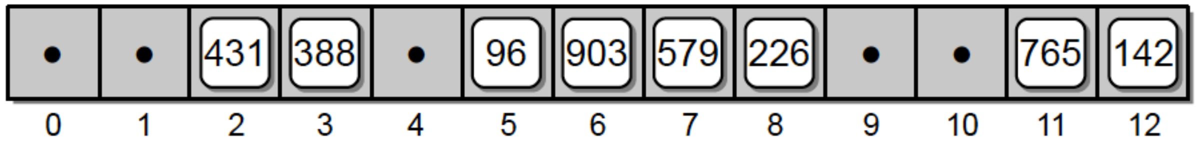

Figure 11.13: The hash table using double hashing.

Rehashing

- How do we decide how big the hash table should be?
- With a hash table, we create a new array larger than the original, but we cannot simply copy the contents from the old array to the new one.
- Instead, we have to rebuild or rehash the entire table by adding each key to the new array as if it were a new key being added for the first time.

Rehashing

- The ratio between the number of keys in the hash table and the size of the table is called the **load factor**.
- In practice, a hash table should be expanded before the load factor reaches 80%.

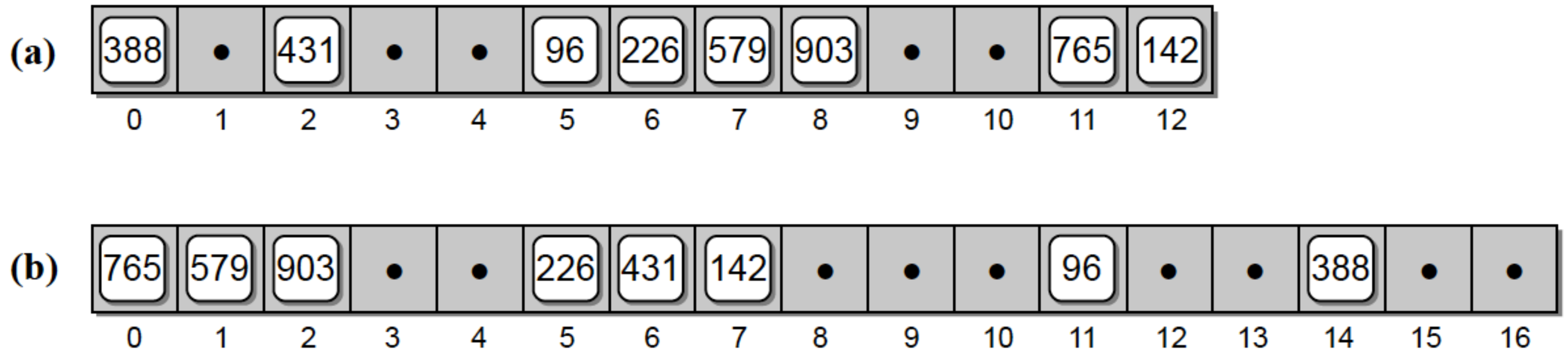

Figure 11.14: The result of enlarging the hash table from 13 elements to 17.

Efficiency Analysis

- The efficiency of the hash operations depends on the **hash function**, the **size of the table**, and the **type of probe** used to resolve collisions.
- Assume there are n elements currently stored in the table of size m.
- In the worst case, the probe has to visit every entry in the table, which requires O(m) time.
- Hashing is very efficient in the average case. The average case assumes the keys are uniformly distributed throughout the table.

Efficiency Analysis

- When using a linear probe, the average number of comparisons required to locate a key in the hash table for a successful search is: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

- and for an unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$

Efficiency Analysis

- When using a quadratic probe or double hashing, the average number of comparisons required to locate a key for a successful search is: $\frac{-\log(1-\alpha)}{\alpha}$

  - and for an unsuccessful search: $\frac{1}{(1-\alpha)}$

Separate Chaining

- We can eliminate collisions entirely if we allow multiple keys to share the same table entry.
- To accommodate multiple keys, **linked lists** can be used to store the individual keys that map to the same entry.
- The linked lists are commonly referred to as **chains** and this technique of collision resolution is known as **separate chaining**.

Separate Chaining

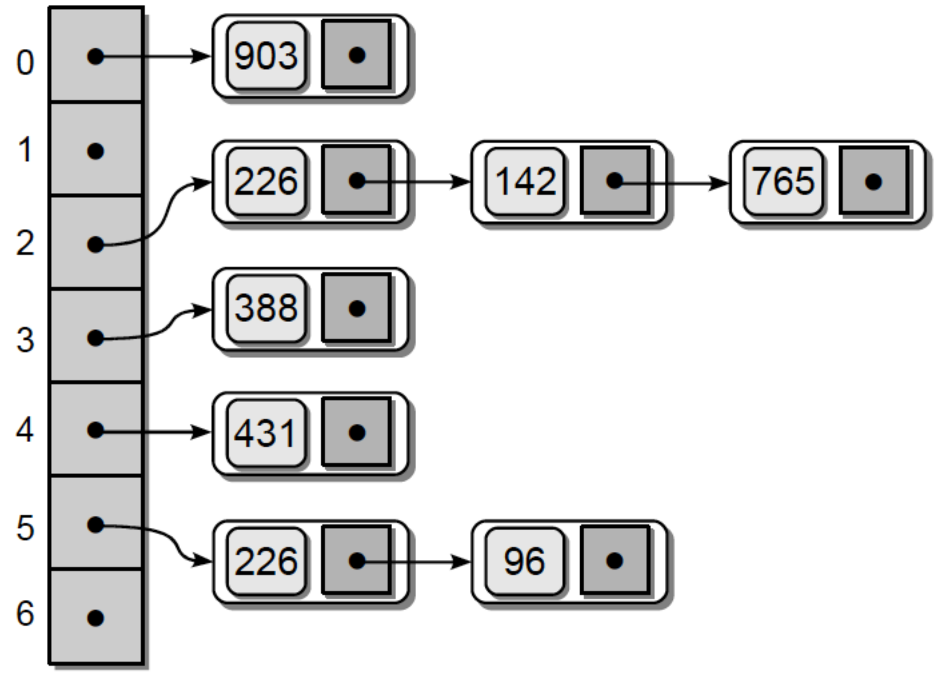

Figure 11.15: Hash table using separate chaining.

- Searching: After mapping the key to an entry in the table, the corresponding linked list is searched to determine if the key is in the table.
- Deleting: The key is again mapped in the usual way to find the linked list containing that key. After locating the list, the node containing the key is removed from the linked list just as if we were removing any other item from a linked list. Since the keys are not stored in the array elements themselves, we no longer have to mark the entry as having been filled by a previously deleted key.

Separate Chaining

- Separate chaining is also known as **open hashing** since the keys are stored outside the table.
- The term **closed hashing** is used when the keys are stored within the elements of the table as described in the previous section.

Hash Function

- The efficiency of hashing depends in large part on the selection of a good hash function.
- The computation should be simple in order to produce quick results.
- The resulting index cannot be random. When a hash function is applied multiple times to the same key, it must always return the same index value.
- If the key consists of multiple parts, every part should contribute in the computation of the resulting index value.
- The table size should be a prime number, especially when using the modulus operator. This can produce better distributions and fewer collisions as it tends to reduce the number of keys that share the same divisor.

The HashMap Abstract Data Type

- Python's dictionary is implemented using a hash table with closed hashing.
- For the implementation of the HashMap ADT, we are going to use a hash table with closed hashing and a double hashing probe.
- We can start with a relatively small table (M = 7) and allow it to expand as needed by rehashing each time the load factor is exceeded.
- As we saw earlier, a load factor between 1/2 and 2/3 provides good performance in the average case. For our implementation we are going to use a load factor of 2/3.

Listing 11.1 The `hashmap.py` module.

```python
# Implementation of the Map ADT using closed hashing and a probe with
# double hashing.
from arrays import Array

class HashMap :
   # Defines constants to represent the status of each table entry.
  UNUSED = None
  EMPTY = _MapEntry( None, None )

   # Creates an empty map instance.
  def __init__( self ):
    self._table = Array( 7 )
    self._count = 0
    self._maxCount = len(self._table) - len(self._table) // 3

   # Returns the number of entries in the map.
  def __len__( self ):
    return self._count

   # Determines if the map contains the given key.
  def __contains__( self, key ):
    slot = self._findSlot( key, False )
    return slot is not None

  # Adds a new entry to the map if the key does not exist. Otherwise, the
  # new value replaces the current value associated with the key.
 def add( self, key, value ):
   if key in self :
     slot = self._findSlot( key, False )
     self._table[slot].value = value
     return False
   else :
     slot = self._findSlot( key, True )
     self._table[slot] = _MapEntry( key, value )
     self._count += 1
     if self._count == self._maxCount :
       self._rehash()
     return True

  # Returns the value associated with the key.
 def valueOf( self, key ):
   slot = self._findSlot( key, False )
   assert slot is not None, "Invalid map key."
   return self._table[slot].value

  # Removes the entry associated with the key.
 def remove( self, key ):
   ......

  # Returns an iterator for traversing the keys in the map.
 def __iter__( self ):
   ......

  # Finds the slot containing the key or where the key can be added.
  # forInsert indicates if the search is for an insertion, which locates
  # the slot into which the new key can be added.
 def _findSlot( self, key, forInsert ):
    # Compute the home slot and the step size.
   slot = self._hash1( key )
   step = self._hash2( key )

    # Probe for the key.
   M = len(self._table)
   while self._table[slot] is not UNUSED :
     if forInsert and \
        (self._table[slot] is UNUSED or self._table[slot] is EMPTY) :
       return slot
     elif not forInsert and \
        (self._table[slot] is not EMPTY and self._table[slot].key == key) :
       return slot
     else :
       slot = (slot + step) % M
  # Rebuilds the hash table.
  def _rehash( self ) :
     # Create a new larger table.
    origTable = self._table
    newSize = len(self._table) * 2 + 1
    self._table = Array( newSize )

     # Modify the size attributes.
    self._count = 0
    self._maxCount = newSize - newSize // 3

     # Add the keys from the original array to the new table.
    for entry in origTable :
      if entry is not UNUSED and entry is not EMPTY :
        slot = self._findSlot( key, True )
        self._table[slot] = entry
        self._count += 1

  # The main hash function for mapping keys to table entries.
  def _hash1( self, key ):
    return abs( hash(key) ) % len(self._table)

  # The second hash function used with double hashing probes.
  def _hash2( self, key ):
    return 1 + abs( hash(key) ) % (len(self._table) - 2)

# Storage class for holding the key/value pairs.
class _MapEntry :
  def __init__( self, key, value ):
    self.key = key
    self.value = value
```