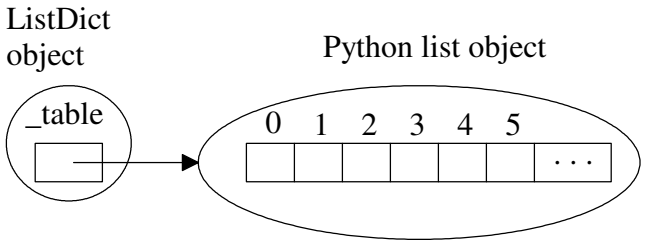
METHOD	WHAT IT DOES
`d = ListDict()`	Creates and returns an empty dictionary.
`d.__getitem__(key)`	Same as `d[key]`. Returns the value associated with `key` if `key` exists, or returns `None` otherwise.
`d._setitem__(key, value)`	Same as `d[key] = value`. If `key` exists, replaces its associated value with `value`; otherwise, inserts a new `key/value` entry.
`d.pop(key)`	Removes the `key`/value entry and returns the associated value if `key` exists, or returns `None` otherwise.
`d.__contains__(key)`	Same as `key in d`. Returns `True` if `key` is a key in `d`, or returns `False` otherwise.
`d.__len__()`	Same as `len(d)`. Returns the number of entries currently in `d`.
`d.__iter__()`	Returns an iterator on the keys of `d`. Supports a `for` loop with `d`. Keys are visited in an unspecified order.
`d.__str__()`	Same as `str(d)`. Returns a string containing the string representations of the key/value pairs in `d`.

[TABLE 19.3] The interface of the `ListDict` class



1. Consider the following `ListDict` class implementation.

```
class Entry(object):
    """A key/value pair."""

    def __init__(self, key, value):
        self.key = key
        self.value = value

    def __eq__(self, other):
        return self.key == other.key

    def __str__(self):
        return str(self.key) + ":" + str(self.value)
```

```
class ListDict(object):
    """A list-based implementation of a dictionary."""

    def __init__(self):
        self._table = []

    def __getitem__(self, key):
        """Returns the value associated with key or
        returns None if key does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table[index].value
        except:
            return None

    def pop(self, key):
        """Removes the entry associated with key and
        returns its value or returns None if key
        does not exist."""
        entry = Entry(key, None)
        try:
            index = self._table.index(entry)
            return self._table.pop(index).value
        except:
            return None

    def __setitem__(self, key, value):
        """Inserts an entry with key/value if key
        does not exist or replaces the existing value
        with value if key exists."""
        entry = Entry(key, value)
        try:
            index = self._table.index(entry)
            self._table[index] = entry
        except:
            self._table.append(entry)

    # The methods __len__(), __str__(), keys(),
    # __contains__, and values() are exercises
```

a) Explain how the __getitem__ method looks up a key?

b) What is the average-case theta notation of each operation?

__getitem__(self, key):

pop(self, key):

__setitem__(self, key, value):

__contains__(key):

2. Hashing Motivation and Terminology:

a) Sequential search of an array, linked list follows the same search pattern for any given target value being searched for, i.e., scans the array from one end to the other, or until the target is found.

If *n* is the number of items being searched, what is the average and worst case theta notation for a search?

average case $\Theta($)

worst case $\Theta($)

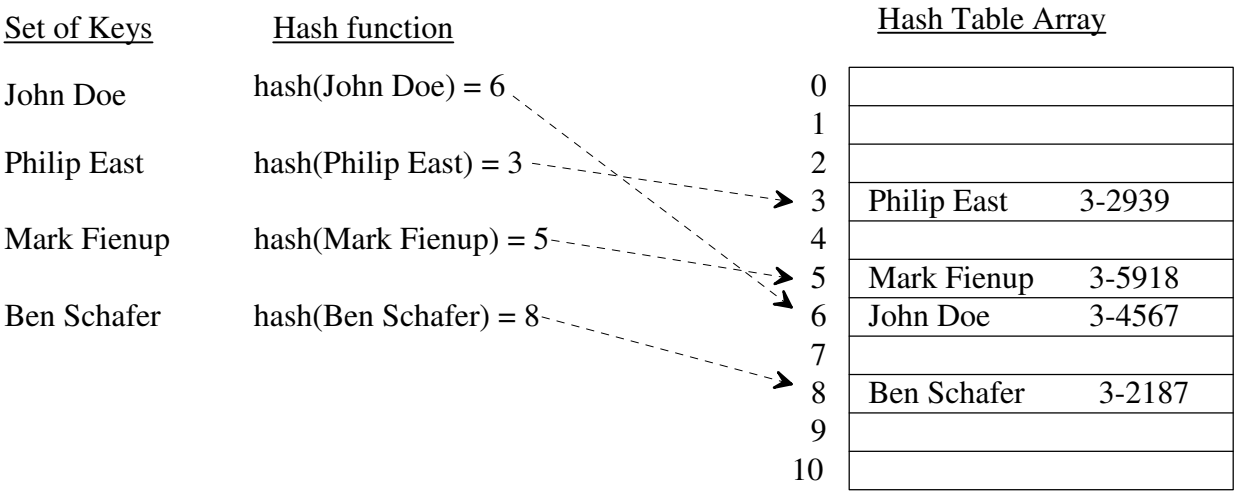
b) Similarly, binary search of a sorted array or AVL tree always uses a fixed search strategy for any given target value. For example, binary search always compares the target value with the middle element of the remaining portion of the array needing to be searched.

If *n* is the number of items being searched, what is the average and worst case theta notation for a search?

average case $\Theta($)

worst case $\Theta($)

Hashing tries to achieve average constant time (i.e., $O(1)$) searching by using the target's value to calculate where in the array (called the *hash table*) it should be located, i.e., each target value gets its own search pattern. The translation of the target value to an array index (called the target's *home address*) is the job of the *hash function*. A *perfect hash function* would take your set of target values and map each to a unique array index.



a) If *n* is the number of items being searched and we had a perfect hash function, what is the average and worst case theta notation for a search?

average case $\Theta($)

worst case $\Theta($)

3. Unfortunately, perfect hash functions are a rarity, so in general two or more target values might get mapped to the same hash-table index, called a *collision*.

Collisions are handled by two approaches:

- *chaining, closed-address*, or *external chaining*: all target values hashed to the same home address are stored in a data structure (called a *bucket*) at that index (typically a linked list, but a BST or AVL-tree could also be used). Thus, the hash table is an array of linked list (or whatever data structure is being used for the buckets)
- *open-address* with some *rehashing* strategy: Each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.

4. Consider the following `HashDict` class implementation using an externally chained linked list for each bucket.

```
from arrays import Array

class HashEntry(Entry):

   def __init__(self, key, value, next):
      Entry.__init__(self, key, value)
      self.next = next

class HashDict(object):
   """A hashing implementation of a dictionary."""

   DEFAULT_CAPACITY = 3

   def __init__(self, capacity = None):
      if capacity is None:
         self._capacity = HashDict.DEFAULT_CAPACITY
      else:
         self._capacity = capacity
      self._table = Array(self._capacity)
      self._size = 0
      self._priorEntry = None
      self._foundEntry = None
      self._index = None

   def __contains__(self, key):
      """Returns True if key is in the dictionary or
          False otherwise."""
      self._index = abs(hash(key)) % self._capacity
      self._priorEntry = None
      self._foundEntry = self._table[self._index]
      while self._foundEntry != None:
         if self._foundEntry.key == key:
            return True
         else:
            self._priorEntry = self._foundEntry
            self._foundEntry = self._foundEntry.next
      return False

   def __getitem__(self, key):
      """Returns the value associated with key or
          returns None if key does not exist."""
      if key in self:
         return self._foundEntry.value
      else:
         return None

   def pop(self, key):
      """Removes the entry associated with key and
          returns its value or returns None if key
      does not exist."""
      if not key in self:
         return None
      else:
         if self._priorEntry is None:
            self._table[self._index] = self._foundEntry.next
         else:
            self._priorEntry.next = self._foundEntry.next
         self._size -= 1
         return self._foundEntry.value
```

a) Explain how the __getitem__ method looks up a key?

b) What is the average-case theta notation of each operation? (Let α be *load factor* (n/Array size))

__getitem__(self, key):

pop(self, key):

__setitem__(self, key, value):

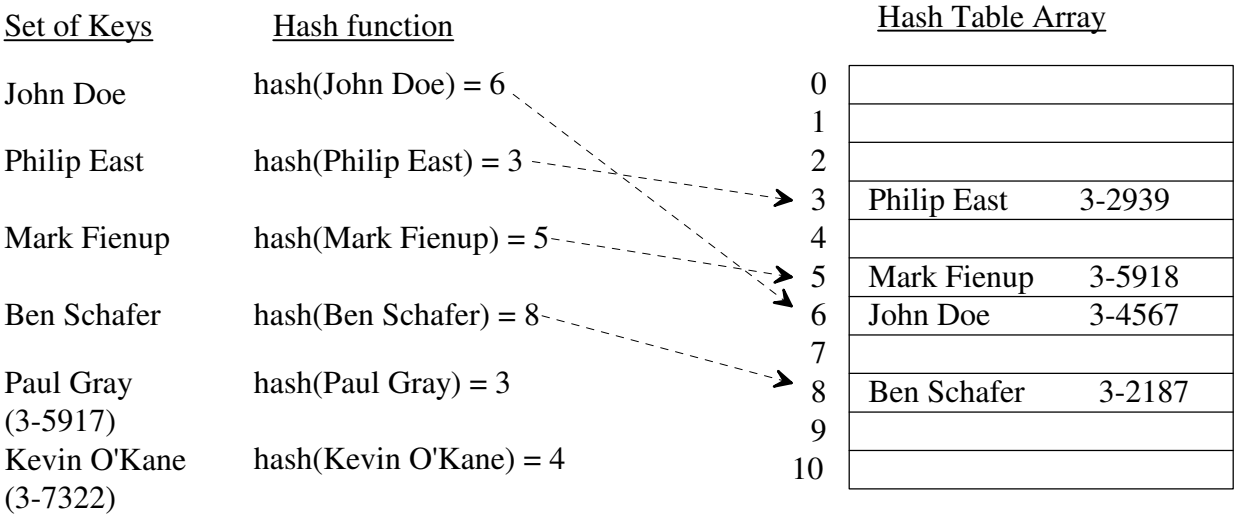
__contains__(key):

```
   def __setitem__(self, key, value):
      """Inserts an entry with key/value if key
          does not exist or replaces the existing value
          with value if key exists."""
      if not key in self:
         newEntry = HashEntry(key, value,
                              self._table[self._index])
         self._table[self._index] = newEntry
         self._size += 1
         return None
      else:
         returnValue = self._foundEntry.value
         self._foundEntry.value = value
         return returnValue

   def __len__(self):
      return self._size

   def __str__(self):
      result = "HashDict: capacity = " + \
               str(self._capacity) + ", load factor = " + \
               str(len(self) / float(self._capacity))
      for i in xrange(self._capacity):
         rowStr = ""
         entry = self._table[i]
         while entry != None:
            rowStr += str(entry) + " "
            entry = entry.next
         if rowStr != "":
            result += "\nRow " + str(i) + ": " + rowStr
      return result
   # The methods keys() and values() are exercises
```

5. Consider the following examples using *open-address* approach with some *rehashing* strategy to handle collisions. In open-address approaches each hash table home address holds at most one target value. The first target value hashed to a specify home address is stored there. Later targets getting hashed to that home address get rehashed to a different hash table address. A simple rehashing strategy is *linear probing* where the hash table is scanned circularly from the home address until an empty hash table address is found.

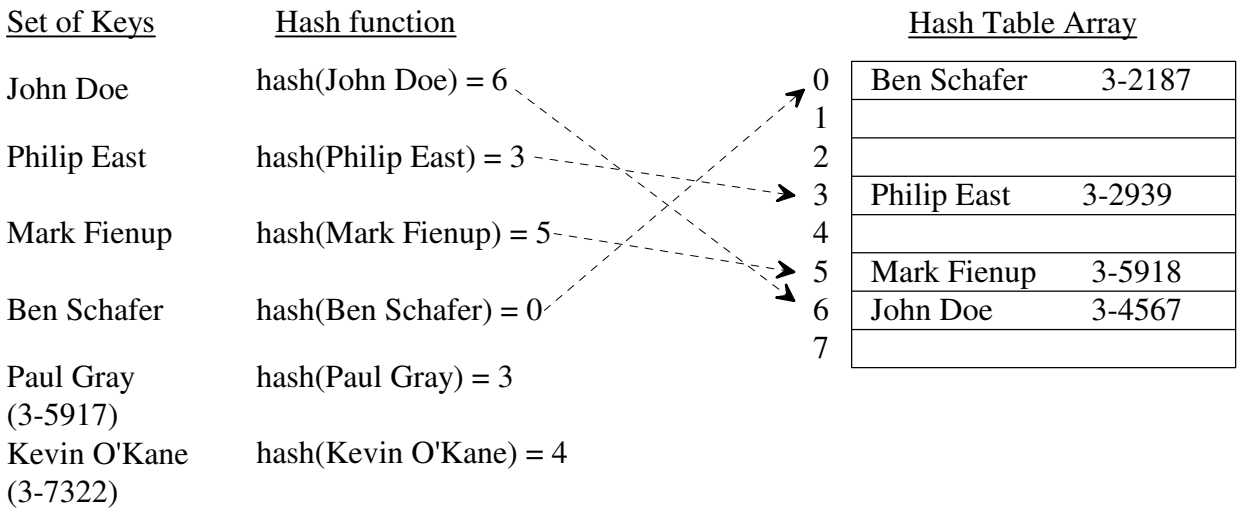


a) Assuming open-address with linear probing where would Paul Gray and Kevin O'Kane be placed?

b) Common rehashing strategies include the following.

Rehash Strategy	**Description**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)
quadratic probing	Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size

c) Assume quadratic probing, insert "Paul Gray" and "Kevin O'Kane" into the hash table.



d) Assume double hashing, insert "Paul Gray" and "Kevin O'Kane" into the hash table.

Set of Keys	Hash function
John Doe	hash(John Doe) = 6
Philip East	hash(Philip East) = 3
Mark Fienup	hash(Mark Fienup) = 5
Ben Schafer	hash(Ben Schafer) = 0
Paul Gray (3-5917)	hash(Paul Gray) = 3 rehash_offset(Paul Gray) = 1
Kevin O'Kane (3-7322)	hash(Kevin O'Kane) = 4 rehash_offset(Kevin O'Kane) = 3

Hash Table Array

0	Ben Schafer	3-2187
1		
2		
3	Philip East	3-2939
4		
5	Mark Fienup	3-5918
6	John Doe	3-4567
7		

e) For the above double-hashing example, what would be the sequence of hashing and rehashing addresses tried for Kevin O'Kane if the table was full? (home address + (rehash attempt #) * offset) % (hash table size) would be:

(4 + (rehash attempt #) * 3) % 8

Rehash Attempt #	0	1	2	3	4	5	6	7	8	9	10
Address											

f) Indicate whether each of the following rehashing strategies suffer from primary or secondary clustering.

- *primary clustering* - keys mapped to a home address follow the same rehash pattern
- *secondary clustering* - rehash patterns from initially different home addresses merge together

Rehash Strategy	**Description**	**Suffers from:**	
		primary clustering	**secondary clustering**
linear probing	Check next spot (counting circularly) for the first available slot, i.e., (home address + (rehash attempt #)) % (hash table size)		
quadratic probing	Check a square of the attempt-number away for an available slot, i.e., (home address + ((rehash attempt #)^2 +(rehash attempt #))/2) % (hash table size), where the hash table size is a power of 2		
double hashing	Use the target key to determine an offset amount to be used each attempt, i.e., (home address + (rehash attempt #) * offset) % (hash table size), where the hash table size is a power of 2 and the offset hash returns an odd value between 1 and the hash table size		

6. Consider the following `HashTable` class implementation.

```
""" File: hashtable.py  Case study for Chapter 19.  """

from arrays import Array

class HashTable(object):
   "Represents a hash table."""

   EMPTY = None
   DELETED = True

   def __init__(self, capacity = 29,
                hashFunction = hash,
                linear = True):
      self._table = Array(capacity, HashTable.EMPTY)
      self._size = 0
      self._hash = hashFunction
      self._homeIndex = -1
      self._actualIndex = -1
      self._linear = linear
      self._probeCount = 0

   def insert(self, item):
      """Inserts item into the table
      Preconditions: There is at least one empty cell or
      one previously occupied cell.
      There is not a duplicate item."""
      self._probeCount = 0
      # Get the home index
      self._homeIndex = abs(self._hash(item)) % len(self._table)
      distance = 1
      index = self._homeIndex

      # Stop searching when an empty cell is encountered
      while not self._table[index] in (HashTable.EMPTY,
                                       HashTable.DELETED):

         # Increment the index and wrap around to first
         # position if necessary
         if self._linear:
            increment = index + 1
         else:
            # Quadratic probing
            increment = self._homeIndex + distance ** 2
            distance += 1
         index = increment % len(self._table)
         self._probeCount += 1

      # An empty cell is found, so store the item
      self._table[index] = item
      self._size += 1
      self._actualIndex = index
```

Let α be the load factor. The average probes with **linear probing** for insertion or unsuccessful search is:

$$\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\alpha)^2}\right)\right)$$

The average probes with linear probing for successful search is:

$$\left(\frac{1}{2}\right)\left(1+\left(\frac{1}{(1-\alpha)}\right)\right)$$

If $\alpha = 0.8$, what is the average probes for:

a) unsuccessful search?

b) successful search?

c) Why is an unsuccessful search worse than a successful search?

```
def search(self, item):
   """Search for item in the table."""
   self._probeCount = 0
   # Get the home index
   self._homeIndex = abs(self._hash(item)) % len(self._table)
    distance = 1
   index = self._homeIndex

   # Stop searching when an empty cell is encountered
   while not self._table[index] in (HashTable.EMPTY,  item):

      # Increment the index and wrap around to first
      # position if necessary
      if self._linear:
         increment = index + 1
      else:
         # Quadratic probing
         increment = self._homeIndex + distance ** 2
         distance += 1
      index = increment % len(self._table)
      self._probeCount += 1

   # An empty cell is found, so return None
   if self._table[index] == HashTable.EMPTY:
      return None
   else:
      self._actualIndex = index
      return self._table[index]

# Methods __len__(), __str__(), loadFactor(), homeIndex(),
# actualIndex(), and probeCount() are exercises.
```

7. Let α be the load factor. The average probes with **quadratic probing** for insertion or unsuccessful search is:

$$(\frac{1}{1-\alpha}) - \alpha - \log_e(1-\alpha)$$

The average probes with quadratic probing for successful search is:

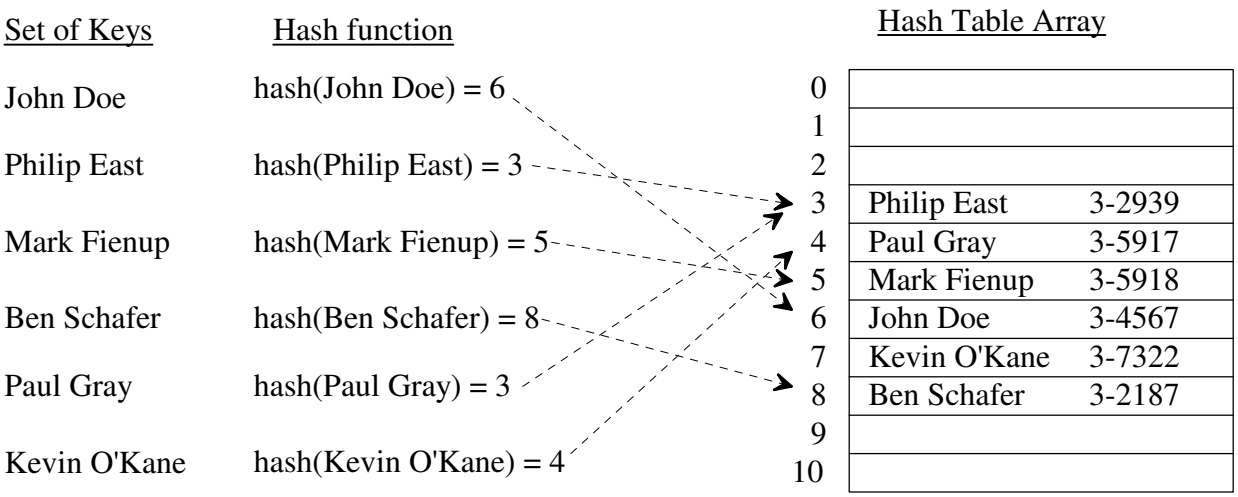
$$1 - (\frac{\alpha}{2}) - \log_e(1-\alpha)$$

Consider the following table containing the average number probes for various load factors:

Probing Type	**Search outcome**	**Load Factor**				
		0.25	**0.5**	**0.67**	**0.8**	**0.99**
Linear Probing	unsuccessful	1.39	2.50	5.09	13.00	5000.50
	successful	1.17	1.50	2.02	3.00	50.50
Quadratic Probing	unsuccessful	1.37	2.19	3.47	5.81	103.62
	successful	1.16	1.44	1.77	2.21	5.11

a) Why do you suppose the "general rule of thumb" in hashing tries to keep the load factor between 0.5 and 0.67?

5. Allowing deletions from an open-address hash table complicates the implementation. Assuming linear probing we might have the following



a) If "Mark Fienup" is deleted, how will we find Kevin O'Kane?

b) How might we fix this problem?

Below from: http://research.cs.vt.edu/AVresearch/hashing/index.php

Quadratic Probing:
Another probe function that eliminates primary clustering is called **quadratic probing**. Here the probe function is some quadratic function $p(K, i) = c_1 i^2 + c_2 i + c_3$ for some choice of constants c_1, c_2, and c_3.

The simplest variation is p(K, i) = i2 (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$). Then the ith value in the probe sequence would be $(h(K) + i^2)$ mod M. Under quadratic probing, two keys with different home positions will have diverging probe sequences. For example, given a hash table of size M = 101, assume for keys k_1 and k_2 that and $h(k_2) = 30$ and $h(k_2) = 29$. The probe sequence for k_1 is 30, then 31, then 34, then 39. The probe sequence for k_2 is 29, then 30, then 33, then 38. Thus, while k_2 will probe to k_1's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Try inserting numbers for yourself, and demonstrate how the probe sequences for diverge by inserting these numbers into a table of size 16: 0, 16, 32, 15, 31.

Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on the probe sequence. Using $p(K, i) = i^2$ gives particularly inconsistent results. For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, this means that the record cannot be inserted at all! For example, if our hash table has three slots, then records that hash to slot 0 can probe only to slots 0 and 1 (that is, the probe sequence will never visit slot 2 in the table). Thus, if slots 0 and 1 are full, then the record cannot be inserted even though the table is not full! A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $p(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $p(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

Double Hashing
Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the name given to the the situation when keys share substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value. This is because function p ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called secondary clustering.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, h_2. Thus, the probe sequence would be of the form $p(K, i) = i * h_2(K)$. This method is called double hashing.

A good implementation of double hashing should ensure that all of the probe sequence constants are relatively prime to the table size M. This can be achieved easily. One way is to select M to be a prime number, and have h_2 return a value in the range $1 <= h_2(K) <= M-1$. Another way is to set $M = 2^m$ for some value m and have h_2 return an odd value between 1 and 2^m.