1. 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 a 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.

a) 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

Rehashing strategies			
Strategies	**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		

b) Assume quadratic probing, 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) = 3
Kevin O'Kane (3-7322)	hash(Kevin O'Kane) = 5 rehash_offset(Kevin O'Kane) = 7

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	

c) 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		

2. When writing a general purpose templated Hash Map/Table class, why might we need to to pass the hashing function as a parameter to the constructor?

```
template <class K, class V>
class LinkedHashEntry {
public:
      K key;
      V value;
      LinkedHashEntry<K, V> *next;

      LinkedHashEntry(K newKey, V newValue) {
            key = newKey;
            value = newValue;
            next = NULL;  }
};

template <class K, class V>
class HashMap {

private:
      LinkedHashEntry<K, V> **table;
      int tableSize;
      int numItems;
      int (*hashFunc)(const K &);

public:
      HashMap(int size, int (*hashFn)(const K &));
      ~HashMap();
      bool contains(K key);
      V get(K key);
      void put(K key, V value);
      void remove(K key);
      int size() const;
      void clear();
      void display();
      double loadFactor() const;
};
```

3. How would you construct a HashMap object that stored integer keys with C++ string values?