<u>Tries, Hash Table Problems</u>

1) Use the following hash function to insert the given elements into the hash table below. Use **<u>quadratic probing</u>** to resolve any collisions. You may assume that the correct table size (in this case, 10) is always passed to the function with the key that is being hashed.

```
     int hash(int key, int table_size)v{
          int a = (key % 100) / 10;
          int b = key % 10;

          return (a + b) % table_size;
     }
```

Keys to insert (one by one, in the following order): 2555, 1523, 5893, 800, 956

0	1	2	3	4	5	6	7	8	9

2) There are two hash functions that take in strings as input shown below. Each returns an integer in between 0 and 1,000,002. (Note: 1,000,003 is a prime number.) Which of these two is a better hash function? Explain the weakness in the other function.

```
int f1(char* str) {

    int i = 0, res = 0;
    while (str[i] != ‘\0’) {
        res = (256*res + (int)(str[i]))%1000003;
        i++;
    }
    return res;
}

int f2(char* str) {

    int i = 0, res = 0;
    while (str[i] != ‘\0’) {
        res = (res + (int)(str[i]))%1000003;
        i++;
    }
    return res;
}
```

3) Consider inserting the following words into an initially empty trie, where nodes are created ONLY if they are necessary. Note that an empty trie is a NULL pointer and has zero nodes. How many nodes will be created after all these words are inserted, including the root node?

cat, part, do, art, cart, car, dog, dart, ark, park

4) Write a recursive function that takes the root of a trie, *root*, and a single character, *alpha*, and returns the number of strings in the trie that contain that letter. You may assume that letter is a valid lowercase alphabetic character (‘a’ through ‘z’).

Note that we are *not* simply counting how many times a particular letter is represented in the trie. For example, if the trie contains only the strings “apple,” “avocado,” and “persimmon,” then the following function calls should return the values indicated:

countStringsWithLetter(root, 'p') = 2
countStringsWithLetter(root, 'm') = 1

The TrieNode struct and function signature are given below. You may assume that the variable numwords accurately stores the number of valid words stored (# of nodes within the trie with flag set to 1) in all trie structs.

```
typedef struct TrieNode {
    struct TrieNode *children[26];
    int numwords;
    int flag; // 1 if the string is in the trie, 0 otherwise
} TrieNode;

int countStringsWithLetter(TrieNode *root, char alpha) { //fill in }}
```

5) Write a function that takes the root of a trie (*root*) and returns the number of strings in that trie that **end** with the letter *q*. The *count* member of the node struct indicates how many occurrences of a particular string have been inserted into the trie. So, a string can be represented in the trie multiple times. If a string ending in *q* occurs multiple times in the trie, all occurrences of that string should be included in the value returned by this function.

The node struct and function signature are given below. You must write your solution in a **single** function. You cannot write any helper functions.

```
typedef struct TrieNode {
   struct TrieNode *children[26];

   // Indicates how many occurrences of a particular string
   // is found in the trie.
   int count;
} TrieNode;

int ends_with_q_count(TrieNode *root) { // fill in }
```