You may work in pairs for this assignment. If you choose to work with a partner, make sure only one of you submits a solution, and you paste a copy of the Partners Template that contains the names and PIDs of both students at the beginning of the file you submit.

You will submit your solution to this assignment to the Curator System (as `HW02P1` and `HW02P2`). Your solution must be either a plain text file (e.g., NotePad++) or a typed MS Word document; submissions in other formats will not be graded.

Partial credit will only be given if you show relevant work.

Part I Hash Table Basics

1. Consider a hash table consisting of $M = 11$ slots, and suppose nonnegative integer key values are hashed into the table using the hash function `h1()`:

```
int h1 (int key) {
  int x = (key + 7) * (key + 7);
  x = x / 16;
  x = x + key;
  x = x % 11;
  return x;
}
```

a) [5 points] Suppose that collisions are resolved by using linear probing. The integer key values listed below are to be inserted, in the order given. Show the home slot (the slot to which the key hashes, before any probing), the probe sequence (if any) for each key, and the final contents of the hash table after the following key values have been inserted in the given order:

Key Value	Home Slot	Probe Sequence
43		
23		
1		
0		
15		
31		
4		
7		
11		
3		

Final Hash Table:

Slot	0	1	2	3	4	5	6	7	8	9	10
Contents											

b) [5 points] Suppose that collisions are resolved by using quadratic probing, with the probe function:

$$\left(k^2 + k\right) / 2$$

The integer key values listed below are to be inserted, in the order given. Show the home slot (the slot to which the key hashes, before any probing), the probe sequence (if any) for each key, and the final contents of the hash table after the following key values have been inserted in the given order:

Key Value	Home Slot	Probe Sequence
43		
23		
1		
0		
15		
31		
4		
7		
11		
3		

Final Hash Table:

Slot	0	1	2	3	4	5	6	7	8	9	10
Contents											

c) [5 points] Suppose that collisions are resolved by using double hashing (see the course notes), with the secondary hash function `Reverse(key)`, which reverses the digits of the key and returns that value; for example, `Reverse(7823) = 3287`.

The integer key values listed below are to be inserted, in the order given. Show the home slot (the slot to which the key hashes, before any probing), the probe sequence (if any) for each key, and the final contents of the hash table after the following key values have been inserted in the given order:

Key Value	Home Slot	Probe Sequence
43		
23		
1		
0		
15		
31		
4		
7		
11		
3		

Final Hash Table:

Slot	0	1	2	3	4	5	6	7	8	9	10
Contents											

2. Consider implementing a hash table for an application in which we will build an initial hash table by inserting a substantial collection of records. After this, we expect that the number of insertions and the number of deletions performed to be roughly the same, although there may be long runs of consecutive insertions or consecutive deletions. Furthermore, the table will use a probe strategy to resolve any collisions that occur during insertion, and therefore we will "tombstone" cells from which a record has been deleted.

a) [10 points] If we implement the hash table described above, then when we search for a record, we cannot conclude the record is not in the table until we have found an empty cell in the table, not just a tombstone. (We will ensure that the table never reaches the state that there are no empty cells.) Explain carefully why the search cannot stop when a tombstone is encountered.

But, as more and more deletions occur, the number of tombstones will affect the performance of the hash table, since those cells are essentially locked into an unusable state. Therefore, we could adopt the following strategy on insertion. If we pass a tombstoned cell during an insertion search, we will remember the first such cell we have seen, and once the insertion search concludes at an empty cell, we will place the new record into that tombstoned cell.

b) [10 points] If we implement insertion as described above, would searches for absent records (as described in part a) still need to proceed until an empty cell is found? Justify your conclusion carefully.

3. Suppose you are implementing a hash table and are trying to choose between using a probing strategy and using chaining (each slot is actually a linked list that can hold as many records as needed). You will use the same hash function and hash table size no matter which strategy you select.

a) [10 points] A *primary collision* occurs when two records map to the same home slot in the table. Will the number of primary collisions be lower if you choose probing or if you choose chaining? Justify your conclusion carefully.

b) [10 points] One objection to using chaining is that you'll have to perform a linear traversal of the linked list to find a record, and linear traversals are slow. Considering that, and assuming that a small but significant number of primary collisions will occur, would searching be more efficient if you chose linear probing instead of chaining? Justify your conclusion carefully.

c) [10 points] Reconsider part b under the assumption that you'd choose quadratic probing as an alternative to chaining.

Part II A Hashing Experiment

4. [35 points] One way to compare the performance of several hash functions is to simulate the addition of a fixed collection of records to a hash table, using each hash function, and counting the number of hits each slot in the table receives. From that information, we can construct a histogram showing how many slots received one hit, how many received two hits, and so forth. We can also calculate the average number of steps that a search would take.

   For instance, given a collection of 1000 words and a hash function that computes a nonnegative integer from a string, we might use a table size of 2000 and discover that:

# of hits	# of slots receiving that # of hits
0	1042
1	932
2	15
3	7
4	3
5	1

   Assume that each slot in the hash table uses a linked list to hold the elements that collide in that slot. From that information, we would then calculate that the average number of comparisons in a search would be about 1.064, since the histogram above implies that:

   - 958 words would require 1 step (932 + 15 + 7 + 3 + 1)
   - 26 words would require 2 steps
   - 11 words would require 3 steps
   - 4 words would require 4 steps
   - 1 words would require 5 steps

   Two files of strings are provided on the course website:

`WordData.txt`	111,444 English words (somewhat loosely)
`CAFeatures.txt`	97,497 names of geographic features related to California

   Your task is to compare the performance of four different hash functions on the two files:

`elfhash()`	classic elfhash function employing complex folding
`sfold()`	string hasher employing more complex folding
`FNVHash()`	another string hasher
`DEKHash()`	Donald Knuth's hash function from *The Art of Computer Programming Vol3*

   Java code for each of the four functions is provided in the course notes or on the course website. You will implement driver code to apply each of the hash functions to all of the strings in each of the latter two files and compute the results needed to complete the table given below. The results for the first file are included below to help you validate your logic.

   In accordance with common guidelines for measuring performance, you will consider table sizes that are multiples of the number of strings being hashed.

   For each of the supplied data files, you'll complete a table like the one shown below. The values I obtained for the `WordData.txt` file and the `sumnshift()` function from the notes are shown.

```
Keys:  WordData.txt   111444 strings

Hash fn   load     avg     worst    % < log N
-----------------------------------------------
sumnshift 1.0    1.5411      9        100%
          1.2    1.4547      8        100%
          1.4    1.3633      8        100%
          1.6    1.3230      7        100%
          1.8    1.2815      7        100%
          2.0    1.3216      7        100%
```

The average number of steps is computed in the manner described in the example given on the previous page.

The worst case is the maximum number of steps that would be needed to find a word.

The last value (% < log N) is the percentage of the given strings that would be found in fewer than log N steps, where N is the number of strings that were hashed. Don't expect this to be 100% in all cases.

You should note that it is NOT necessary to implement a fully-functional hash table in order to solve this assignment. In fact, it may be counter-productive to do so. You may, and should, take advantage of the standard library.

Format your results in tables that look just like the example given above… failure to do that will irritate the person who evaluates your results (and therefore result in deductions).