Project 2 (C++): You are to implement a hash table for information storage and retrieval. Data store in the hash table are 5 digits positive integers.

- Bucket size, B, will be get from argv[2].
- Hash function, hashInt (data), is given below.
- Hash table, an array of linked list.
- The input to your program is a text file contains a list of pairs {<op data >} where op is either + or - or ?; + means insert data, - means delete data, and ? means information retrieval; data is a five digits positive integer and it is the key passes to the hash function to get the bucket index for information storage and retrieval.

****

What your program will perform:

1. Read the input one pair at a time: <op data> // see the input data for examples.
2. if op is +, get the index from hashInt (data), then, go to hashTable[index] to perform insertion process
   If op is -, get the index from hashInt (data), then, go to hashTable[index] to perform deletion process
   If op is ?, get the index from hashInt (data), then, go to hashTable[index] to perform information retrieval process
3. Run your program twice, first with bucket size = 11 and next with bucket size 19.
4. Include in your hard copy *.pdf file:
   - cover page
   - source code
   - outFile1 with bucket size = 11
   - outFile2 with bucket size = 11
   - outFile1 with bucket size = 19
   - outFile2 with bucket size = 19

****

Language: C++

****

Project points: 10 pts

Due Date: Soft copy (*.zip) and hard copies (*.pdf):

+1 (11/10 pts): early submission, 2/18/2022, Friday before midnight
-0 (10/10 pts): on time, 2/21/2022 Monday before midnight
-1 (9/10 pts): 1 day late, 2/22/2022 Tuesday before midnight
-2 (8/10 pts): 2 days late, 2/23/2022 Wednesday before midnight
(-10/10 pts): non submission, 2/23/2022 Wednesday after midnight

*** Name your soft copy and hard copy files using the naming convention as given in the project submission requirement discussed in a lecture and is posted in Blackboard.

*** All on-line submission MUST include Soft copy (*.zip) and hard copy (*.pdf) in **the same email attachments** with correct email subject as stated in the email requirement; otherwise, your submission will be rejected.

****

I. Inputs:

a) inFile (argv [1]): A text file contains a list of pairs {<op data >}

For example,

```
+       21005
+       24650
-       21002
+       98173
?       24650
?       12345
+       49178
+       61099
-       98173
:
```

b) BucketSize (argv [2]): first run use 11, the second run uses 19

****

II. output:

outFile1 (argv [3]): Prints as program stated and prints
the final result of the hash table: from 0 to bucketSize-1, one linked list per text line.
For example (let B be the bucketSize):

hashTabel [0]: (-9999, next's data) → (data, next's data) → ….
hashTabel [1]: (-9999, next's data) → (data, next's data) → ….
:
:
hashTabel [B-1]: (-9999, next's data) → (data, next's data) → ….

For example, if index is 5 and the linked list in hashTable [5] is
(-9999) → (12345) → (21005) → (61099) →….
Then you will print:
hashTabel [5]: (-9999, 12345) → (12345, 21005) → (21005, 61099) →(61099, 48879) →….

outFile2 (argv[4]): For all debugging prints in the program.

****

III. Data structure:

****

- listNode class
  - (int) data
  - (listNode *) next

  methods:
  - constructor (data) //create a node with given data
  - printNode (node) // use the format:
    (node's data, next node's data) →
- HTable class
  - (char) op // either '+' or '- or '?'
  - (int) data
  - (int) bucketSize
  - (listNode**) hashTable // An array of linked list (in ascending order with respect to data), size of bucketSize,

  method:
  - constructor (...) // dynamically allocates (listNode**) hashTable, size of bucketSize,
    // where each hashTable[i] point to a dummy node: (-9999, null), to do so,
    //first, you need to declare the pointer array of pointers of listNode*, i.e.,
    // listNode** hashTable = new listNode*[bucketSize]
    // Then, you need to use a for-loop to get a dummy node for each hashTable[i] to point to.

  - (int) hashInt (data) // This is your hash function for getting the index of hashTable
    // Given the data, the method returns the 'index' between 0 to bucketSize-1, by first
    // computes the sum of each digit of data ***times*** the position (from right to left) of the digit, then
    // returns mod (sum, bucketSize).
    // For example, say data = 36587
    // sum ← 1*7 + 2*8 + 3*5 + 4*6 + 5*3
    // On your own!!!

  - informationProcessing (…) // see algorithm below.
  - (listNode *) findSpot (…) // see algorithm below.
  - hashInsert (…) // see algorithm below.

- hashDelete (…) // see algorithm below.
- hashRetrieval (…) // see algorithm below.
- printList (index, outFile1) // print the linked list of hashTable [index], use the format given in the above.
- printHashTable (…) // output the entire hashTable, call printList (…), index from 0 to bucketSize -1.

****
IV. Main (…) // The following methods may contain typos or bugs, report any such to Dr. Phillips and TA.
****

Step 1: inFile ←open input file using argv [1]
bucketSize ← argv [2]
outFile1, outFile2 ← open from argv [3] and argv [4]
Step 2: hashTable ← using constructor to establish the hashTable
Step 3: informationProcessing (inFile, outFile1. outFile2)
Step 4: printHashTable (outFile1)
Step 5: close all files

****
V. informationProcessing (inFile, outFile1. outFile2)
****

Step 0: outFile2 ← print message: "*** enter informationProcessing method." // debugging print
Step 1: op ← get from inFile
data ← get from inFile
Step 2: outFile2 ← print op and data (with description stating what you are printing) // debugging print
Step 3: index ← hashInt (data)
outFile2 ← print index (with description stating what you are printing) // debugging print
Step 4: printList (index, outFile1) // with description stating which bucket of hashTable you are printing before operation
Step 5: if op == '+'
hashInsert (index, data, outFile1, outFile2)
else if op == '-'
hashDelete (index, data, outFile1, outFile2)
else if op == '?'
hashRetrieval (index, data, outFile1, outFile2)
else
outFile1 ← "op is an unrecognize operation!"
Step 6: outFile1 ← print “After one op” op
printList (index, outFile1)
Step 7: repeat step 1 to step 6 until inFile is empty.

****
VI. hashInsert (index, data, outFile1, outFile2)
****

Step 0: outFile2 ← print message: "*** enter hashInsert method." // debugging print
Step 1: Spot ← findSpot (index, data)
Step 2: if Spot’s next != null *and* Spot’s next’s data == data
outFile1 ← “*** Data is already in the hashTable, no insertion takes place!”
else
newNode ← using constructor to get a listNode with data
newNode’s next ← spot’s next
spot’s next ← newNode
Step 4: outFile2 ← " Returning after hashInsert operation … " // debugging print

```
*****************************************
VII.  hashDelete (index, data, outFile1, outFile2)
*****************************************
Step 0: outFile2 ← print message: "** Enter hashDelete method." // debugging print
Step 1: Spot ← findSpot (index, data)
Step 2: if Spot's next != null *and* Spot's next's data == data
                    temp ← Spot's next // the node after Spot is to be deleted
                    Spot's next ← temp's next
                    temp's next ← null
          else
                    outFile1 ← print message: "*** Warning, the data is *not* in the database!"

Step 3: outFile2 ← " Returning after hashDelete operation … " // debugging print

*****************************************
VIII.  hashRetrieval (index, data, outFile1, outFile2)
*****************************************
Step 0: outFile2 ← print message: "** Enter hashRetrieval." // debugging print
Step 1: Spot ← findSpot (index, data)
Step 2: if Spot's next != null *and* Spot's next's data == data
                    outFile1 ← print message: " *** Yes, the data is in the hashTable!"

          else
                    outFile1 ← print message: " *** No, the data is *not* in the hashTable!"

Step 3: outFile2 ← " Returning after hashRetrieval operation … " // debugging print

*****************************************
IV. (listNode*) findSpot (index, data)
*****************************************
Step 1: Spot ← hashTable[index] // points to dummy node
Step 2: if Spot's next != null *and*  Spot's next's data < data
                    Spot ← Spot's next
Step 3: repeat Step 2 until condition failed
Step 4:  return Spot
```