Project 3: You are to implement a version of radix sort that can sort a file contains all positive integers or a file contains mixture of positive and negative integers. The algorithm was taught in class and is given in the lecture note 3.1.

****

Language: C++

****

Project points: 10 pts

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

-0 2/28/2021 Sunday before midnight

-1 for 1 day late: 3/1/2021 Monday before midnight

-2 for 2 days late: 3/2/2021 Tuesday before midnight

-10/10: 3/2/2021 Tuesday before midnightYou

*** Name your soft copy and hard copy files using the naming convention as given in the "Project Submission Requirements" discussed in a lecture and is posted in Google Classroom.

*** 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.

Run your program twice on data1 and data2. Include in your hard copy *.pdf file the following:

- Cover page.
- Draw illustration of Radix-sort as shown in lecture note for data1 and data2. (No hand drawing!!!)
  -1 pt without these two drawings!!
- Program source code
- outFile1 from data1
- outFile2 from data1
- outFile1 from data2
- outFile2 from data2

****

I. Input: one input txt file. // -1 for hard code file name.

inFile (use argv[1]): a text file contains a list of integers (may contain negative numbers).

****

II. Outputs: There will be two output files. // -1 for hard code file names.

a) outFile1 (argv[2]): print the result of the sorted data, one number per text-line.

b) outFile2 (use argv [3]): Print all other outputs, to help you debugging!

****

III. Data structure:

- listNode class: friend of LLStack, LLQueue, RadixSort

  Reuse codes from project 1 (see project 1 specs).
  // Add or delete or modify methods if deem necessary.

- LLStack class: friend of RadixSort

  Reuse code from project 1 (see project 1 specs).
  // Add or delete or modify methods if deem necessary.

- LLQueue class: friend of RadixSort class

  Reuse codes from project 1 (see project 1 specs).
  // Add or delete or modify methods if deem necessary.
  // make modification of printQueue () and add a new method, printData () as below:

  - printQueue (whichTable, index, outFile2)
    // Print to outFile2 the entire linked list Queue of hashTable[whichTable][index].
    // For example, if whichTable is 1 and index is 6, then print
    Table [1][6]: (-9999, 18) → (18, 36) → (36, 72)..... → (613, NULL) → NULL

- A RadixSort class:
    - (int) tableSize // set to 10 for sorting numbers.
    - hashTable[2][tableSize] (LLQueue) // 2 arrays (size of 10) of linked list queues with dummy nodes.
        // Initially, each hashTable[i][j]'s head and tail points to a dummy node.
    - (int) data
    - (int) currentTable // either 0 or 1
    - (int) previousTable // either 0 or 1
    - (int) numDigits // the number of digit in the largest integer that controls the number of iterations of Radix sort
    - (int) offSet // the absolute value of the largest negative integer in the data;
        // the offSet will add to each data before radix sort and subtract afterward.
    - (int) currentPosition // The digit position of the number while sorting.
    - (int) currentDigit

Methods:
    - constructor () // Creates hashTable[2][tableSize]. On your own!
        // Use loops to create LLQueue for each hashTable[i][j], i = 0 to 1 and j = 0 to 9, where
        // each hashTable[i][j] points to a dummy node and initially, head and tail point to dummy node.
    - firstReading (…) // Read from input file; determine the largest and smallest integers in the file
        // and establishes offset. See algorithm below.
    - loadStack (…) // Constructs a linked list stack from the data in inFile. See algorithm below.
    - RSort (…) // Performs Radix sort; sorts from right-to-left. See algorithm below.
    - moveStack(…) // Moves all nodes on the stack to the first hash table. See algorithm below
    - (int) getLength (data) // Determines and returns the length of a given data. On your own!
        //** suggestion: convert data to string to get the length.
    - (int) getDigit (data, position) //Determines and returns the digit at the position of data. On your own!
        //** suggestion: convert data to string to get the digit then convert digit back to int.
        //** Reminding: string indexing is from left to right, when converting to string, the digit you want is
        // at the position of the string counting from right.
    - printTable (whichTable, outFile2) // On your own!
        // Call printQueue () for each none empty queue in hashTable[whichTable].
    - printSortedData (whichTable, outFile1) On your own!
        // Print each none empty queue in hashTable[whichTable], one data per text line;
        //*** make sure to subtract offSet before printing the data.

    For example: if whichTable is 1 and index is 6 and data in the queue as in the above, then print to outFile1
    18
    36
    72
    :
    :
    613

    *** You may add methods if deem necessary.

****
IV. main(…) // Do not hard code file names!!
****
Step 0: inFile ← open the input file (via argv[1])
        outFile1 ← open outFile1 (via argv[2])
        outFile2 ← open outFile2 (via argv[3])
        hashTable[2][tableSize] ← create by RadixSort constructor
Step 1: firstReading (inFile, outFile2)
Step 2: close inFile
Step 3: inFile ← open the input file // open the file second time
Step 4: S ← loadStack (inFile, outFile2)
Step 5: printStack (S, outFile2)
Step 6: RSort (S, outFile1, outFile2)
Step 7: close all files

```
*****************************
V. firstReading (inFile, outFile2)
*****************************
Step 0: outFile2 ← "*** Performing firstReading"
          negativeNum ← 0
          positiveNum ← 0
Step 1:  data ← read from inFile
         If data < negativeNum
                   negativeNum ← data
          If data > positiveNum
                   positiveNum ← data
Step 2: repeat step 1 until inFile is empty
Step 3: negativeNum < 0
                   offSet ← abs (negativeNum)
          else     offSet ← 0
Steo 4: positiveNum ← positiveNum + offset
         numDigits ← getLength (positiveNum)
Step 4: outFile2 ← print positiveNum, negativeNum, offSet, numDigits (with captions)

*****************************
VI. (LLStack) loadStack (inFile, outFile2)
*****************************
Step 0: outFile2 ← "*** Performing loadStack"
Step 1:  S ← create a new stack
Step 2: data ← read a data from inFile
          data += offSet // for simplicity, we add offset even if it is zero.
          newNode ←create a new listNode with data
          push (S, newNode)

step 3: repeat step 2 until inFile is empty
step 4: return S

*****************************
VII. RSort (S, outFile1, outFile2)
*****************************
Step 0: outFile2 ← "*** Performing RSort"
Step 1:  currentPosition ← 0 // the first digit/position from the right of the data.
          currentTable ← 0
Step 2:  moveStack (S, currentPosition, currentTable) // see the algorithm below
Step 3:  printTable (hashTable[currentTable])
Step 4:  currentPosition++
          currentTable ←1
          previousTable ← 0
          currentQueue ← 0
Step 5: // moving nodes from previous table to current table, process queues sequentially.
          newNode ← deleteQ (hashTable[previousTable][currentQueue])
          hashIndex ← getDigit (newNode's data, currentPosition)
          insertQ (hashTable[currentTable][hashIndex], newNode)
          // add newNode at the tail of the queue -- hashTable[currentTable][hashIndex]
step 6:  repeat steps 5 until hashTable[previousTable][currentQueue] is empty.
Step 7: currentQueue ++ // process the next queue in the previous hashTable
 Step 8: repeat step 5  to step 7 until currentQueue >= tableSize - 1
                          // finish moving all queues from current table.
 Step 9: printTable(currentTable, outFile2)
Step 10: previousTable ←currentTable
          currentTable ← mod (currentTable + 1, 2)
          currentQueue ← 0
          currentPosition++
```

Step 11: repeat step 5 to step 10 while currentPosition < numDigits
Step 12: printSortedData (previousTable, outFile1)

****

VIII. moveStack (S, currentPosition, currentTable)

****

Step 0: outFile2 ← "*** Performing moveStack"
Step 1: // move nodes from stack to hashTable[0]
- newNode ← pop (S)
- hashIndex ← getDigit (newNode's data, currentPosition)
// get the currentPosition of the data in the node, make sure it returns a single digit

- insertQ (hashTable[currentTable][hashIndex], newNode)
// add newNode at the tail of the queue at hashTable[currentTable][hashIndex]

Step 2: repeat step 1 until S is empty