Project 3 (Java): You are to implement Radix Sort for string.

*** What you have to do ***
- Run your program with data1 to produce outFile1 and outFile2
- Run your program with data2 to produce outFile1 and outFile2

Your hard copy includes:

- cover page
- source code
- outFile1 of data1
- outFile2 of data1
- outFile1 of data2
- outFile2 of data2

****
Language: (Java)
****
Project points: 10 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):
9/17/2020 Thursday before midnight
+1 9/13/2020 Sunday before midnight
-1 for 1 day late: 9/18/2020 Friday before midnight
-2 for 2 days late: 9/19/2020 Saturday before midnight
-10/10 : after 9/19/2020 Saturday after midnight

*** Follow "Project Submission Requirement" to submit your project.

****
I. Input: inFile (args[0]): a text file contains words (strings).
****
II. Outputs: There will be two output files.
a) outFile1 (args[1]): for the result of the sorted data.
b) outFile2 (args[2]): for observations
****
III. Data structure:
****
- A RSort class:
    - listNode class:
        Re-use code from your project 1, with the following modifications:
        - change the data type to string
        - change the data in dummy node to "dummyNode"
        - add a method, printNode (…) as follows:

        printNode (node) prints the node.data and the node.next.data in the format as below:
            (node.data, node.next.data) →
      *** you may make other changes if deem necessary.
    - LLStack class:

        Re-use code from your project 1, with the following modifications:
        - change the data in dummy to "dummyNode"
        - delete methods that are not used in this project.
        *** you may make other changes if deem necessary.

- LLQueue class

  Re-use code from your project 1, with the following modifications:

  - change the data in dummy to “dummyNode”
  - change data to string data type
  - make modification of printQueue as below
  - printQueue (whichTable, index, outFile2)

    // Print to outFile2 the entire linked list Queue at the given index of hashTable including the dummy node, use the format given below:

    Table [whichTable][index]: (dummyNode, data1) → ($data_1$, $data_2$)...... → ($data_j$, NULL) → NULL

    For example, if whichTable is 0 and index is 4:

    Table [0][4]: (dummyNode, Sammy) → (Sammy, Beth) → (Beth, Pam) → (Pam, John)....... → (Sean, NULL) → NULL

- hashTable[2][256] (LLQueue)

  // 2 arrays (size of 256) of linked list queues with dummy nodes at the head of each Q.
  // Initially, each hashTable[i][j].head and tail pointing to the dummy node.

- data (string)
- currentTable (int) // either 0 or 1
- nextTable (int) // either 0 or 1
- longestStringLength (int)

  // the length of the longest word (string) in the data file

- currentPosition (int)

Methods:

- constructors (…)

  // Creates two hash tables, arrays of LLQueues, size of 256. Needs to create a new LLQueue with a dummy node for all two hashTable{i][j], 256 buckets

- firstReading (inFile)

  // It opens and reads all data to determine
  // the longest string in the input file.
  // See algorithm below.

- loadStack (inFile) // build a stack from the data in inFile. See algorithm below.
- moveStack(. . .) // move all nodes on the stack to the first hash table. See the algorithm given below
- tableIndex (…) // which index of the given hash table
- getChar (…) // returns the character at the currentPosition of the string in the node
- padString (data) // Write this method on your own!!!!

  // if the data is shorter than the longestStringLength, padded the data string with blanks in the back; so that all data will have the same string length as the longestStringLength

- printTable (…, outFile2) // Call printQueue to print **<u>only those none empty Queues in the table</u>**!

  For example if the current Table is 0, and the only none empty queues are 4, 6, 9, and 20, then print as follows:

  Table [0][4]: (dummyNode, $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][6]: (dummyNode, $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][9]: (dummyNode, $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][20]: (dummyNode, $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL

- printSortedData (…, outFile1) // on your own. Output the data of each node in hashTable[nextTable], sequentially, from 1st queue to the last queue.

IV. Main(…)

```
Step 0: inFile ← open the input file
        outFile1 ← open outFile1  // for the output of sorted data
        outFile2 ← open outFile2  // for observations
        use constructor to create two hash tables of  LLQueue where each hashTable[i][j] linked list queue with
         a dummy node, and let the head and tail point to the dummy node.

Step 1: firstReading (inFile) // see algorithm below
Step 2: close inFile
Step 3: inFile ← open the input file // open the file second time
step 4: S ← loadStack (inFile) // see algorithm below
Step 5: printStack (S, outFile2) // Print caption!!! Say what you are printing
Step 6: currentPosition ←  longestStringLength -1 // Sort from right to left of the paddedData.
        currentTable ← 0

Step 7: moveStack (currentPosition, currentTable) // move all nodes on the stack to
                        // the first hash table.  See the algorithm below
Step 8: - currentPosition - -
        - nextTable ← mod (currentTable + 1, 2)
        - currentQueue ← 0

Step 9: // moving  nodes from currentTable to nextTable, process queues in the current table sequentially.
        node <-- deleteHead (hashTable[currentTable][currentQueue])
         chr <--  getChar (node, currentPosition) // i.e., the character at the currentPosition of node's data
        hashIndex <-- (int) chr  or atoi (chr) // cast chr from asci to integer
        addTail (hashTable[nextTable][hashIndex], node) //
        // add the node at the tail of the queue at hashTable[nextTable][hashIndex]

Step 10:  repeat steps 9 until the currentQueue is empty // finish moving all node in currentQueue.

Step 11: currentQueue ++ // process the next queue in the current hashTable
Step 12: repeat step 9  to step 11  while currentQueue < tableSize
                        // finish moving all queues from the current table.
Step 13: printTable((hashTable[nextTable], outFile2) // to outFile2
Step 14:  currentTable ← nextTable
Step 15: repeat step 8 to step 14 while currentPosition  >= 0
Step 16: PrintSortedData (hashTable[nextTable], outFile1)
Step 17: close all files
```

V.   firstReading (inFile)

```
Step 0: longestStringLength ← 0
Step 1:  data← read a word from inFile
Step 2:  If  length of data  >  longestStringLength
              longestStringLength ← length of data
Step 3: repeat step 1 to step 2 until inFile is empty
```

****

VI.   (LLStack) loadStack (inFile)

****

```
Step 0:  S ← Use LLStack constructor to establish a LLStack

Step 1: S.top ← null  // initialize top points to null
Step 2: data ← read a string from inFile
          // YOU MUST READ ONE STRING At A  TIME!! -2pt otherwise

Step 3: paddedData ← padString (data)
Step 4: newNode ←create a new listNode for paddedData
Step 5: push (newNode) <-- push newNode onto the top of the stack
                  newNode.next ← top
                  top ← newNode
step 6: repeat step 2 – step 5 until inFile is empty
step 7: return S
```

****

VII.   moveStack (S, currentPosition, currentTable)

****

```
Step 1: node <-- pop from the top of the stack S
          // move each listNode from stack to hashTable[0]

step 2: chr <--  getChar (node, currentPosition) // i.e., the character at the currentPosition of node’s data

step 3: hashIndex <-- (int) chr  // cast chr from asci to integer

step 4: addTail (hashTable[currentTable][hashIndex], node)
            // add the node at the tail of the queue at hashTable[currentTable][hashIndex]

Step 5: repeat step 1 to step 4 until stack is empty
```