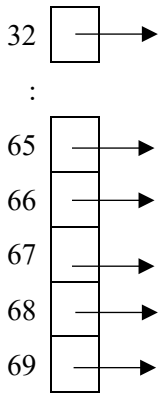
Project 3 (C++): You are to implement the Radix-sort that sorts strings. In this project, we allow duplicates.

*** Please read the specs below entirely three times to understand the specs before asking any question.

What you need to do:

1) You will be given 3 date files: data1, data2 and data3 to test your program; data1 contains a few simple strings where each string only contains A, B, C, D, and E of various length; data2 contains small number of strings, and data3 contains a larger set of strings.

2) Illustration: (You may hand draw the illustration on paper or use computer drawing.)
   - On the top of an 8 X 11 paper, write the strings in **data1,** all in one line horizontally.
   - Eyeball to find the longest string in data1 and count the length of the longest string.
   - Pad any string shorter than the longest string with blank space (use '_' as blank) **on the right side** of each short strings, so that all strings have equal length, and write the padded strings below the original string listing.
   - Next, draw a hash table with only 6 entries, as shown below: 32 for blank, 65 for A, 66 for B, 67 for C, 68 for D and 69 for E.



   - Then, apply Radix-sort to the padded strings one character at one iteration (from right to left), insert each string at the back of the queue of the bucket according to the ascii value of the character; draw one hash table for each iteration. (Although the program only uses 2 hash tables.)
   - When done with sorting, write the result from the last hash table (stings should be sorted) below the hash tables.
   - Include your drawing in the Illustration section of your hard copy (after the cover page). If you hand draw the illustration, take a snap shot of it.

3) Implement your program according to the specs given below.
4) Debug your program until your program passes compilation.
5) Run and debug you program using data1 until your program produces the same hash table and same sorted strings as in your drawing.
6) Run your program two more times using data2, and data3. Check the intermediate hashTable in the logFile to see if your program produces corrected hash table and at the end, it produces the sorted strings.

==========

Include in your hard copy (pdf file)
- a cover page (include only algorithm steps in main() function.)
- your illustration. (-2 if omitted.)
- source code
- data1 // with caption: *** Below is data1 ***
- outFile from data1 // with caption:*** Below is outFile for data1 ***
- logFile from data1 // with caption: *** Below is logFile for data1***. Print only 3 pages if more.
- data2 // with caption: *** Below is data2 ***
- outFile from data2 // with caption:*** Below is outFile for data2 ***
- logFile from data2 // with caption: *** Below is logFile for data2***. Print only 3 pages if more.
- data3 // with caption: *** Below is data3 ***
- outFile from data3 // with caption:*** Below is outFile for data3 ***
- logFile from data3 // with caption: *** Below is logFile for data3***. Print only 3 pages if more.

****
Language: C++
****
Project name: Radix-sort for strings
Project points:10 pts
Due Date: (10/10) on time: 9/26/2025 Friday before midnight (11:59pm).
(-10/10) non-submission: 9/26/2025 Friday after midnight

*** Name your soft copy and hard copy files using the naming convention given in Project Submission Requirements.

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

Email subject: (323.mw or 323.tth) first name last name <Project 3: Radix-sort for strings (C++)>

*** Inside the email body includes:
- Your answer to the 5 questions given in your email body. (-1 if not include.)
- Screen recoding link. (-1 if not submit screen recording).

========== Specs ====== -2 if you hard-code file name ==========

****
I. Input: a) inFile (use argv [1]): a text file contains a list of strings not in any particular format.

****
II. Outputs:
a) outFile (argv [2]): as program dictates.
b) logFile (argv [3]: as program dictates.

****
III. Data structure:

- listNode class
  - (string) data
  - (listNode*) next // set to null

  Methods:
  - constructor (…) // As needed.
  - printNode (node) // using the following format
    (node.data, node.next.data) →
    (fall, leaves) →

- LLQ class // linked list queue
  - (listNode*) head // points to dummy node.
  - (listNode*) tail // initially, tail ← head

  Methods:
  - constructor(...) // **Re-use code from your project 1**
  - insertQ (…) // **Re-use code from your project 1**
  - (listNode*) deleteQ (…) // **Re-use code from your project 1**
  - (bool) isEmpty (…) // **Re-use code from your project 1**
  - printQueue (whichTable, index, oFile) // oFile can be outFile or logFile.
    // Print to oFile the entire LLQ in hashTable[whichTable][index].
    // For example, if whichTable is 1 and index is 6, then print
    Table [1][6]: ("dummy", Azrail) → (Azrail, David) → (David, John)..... → (Zach, NULL) → NULL

- A RadSort class:
  - (int) tableSize // set to 256.
  - (LLQ) hashTable [2][256] // 2 hash table, where each table is an 1D arrays (size of 256) of linked list queues
    // with dummy node.
  - (string) data
  - (int) currentTable // either 0 or 1

- (int) previousTable // either 0 or 1
- (int) longestStringLength // the length of the longest word (string) in the data file
- (int) currentIndex // The character at the index of the string that is currently used in sorting.

Methods:

- constructors (…) // Creates hashTable[2][tableSize], an 2-D arrays of LLQ, size of 256;
  // Use loops to create LLQueue for each hashTable[i][j], i is either 0 or 1 and j = 0 to 255,
  // hashTable[i][j].head ← new listNode ("dummy", null); hashTable[i][j].tail ← hashTable[i][j].head.
- -firstReading (…)
  // It opens and reads all data to determine the longest string in the input file. **See algorithm below**.
- populateFirstTable (…) **// See algorithm below.**
- (string) padString (data) // if a data is shorter than the longestStringLength, padded the data string with blanks
  //in the end of the string (from the right); data will have the same length as the longestStringLength.
  // returns the padded data. **On your own.**
- printTable (…) // **On your own.**
  // Print the queue of the current table, call printQueue to print (**Print <u>only those none empty queues //in the table)</u>** to file (maybe outFile or logFile, as program dictates.)
  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]: ("dummy", $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][6]: ("dummy", $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][9]: ("dummy", $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
  Table [0][20]: ("dummy", $data_1$) → ($data_1$, $data_2$)... → ($data_j$, NULL) → NULL
- RadixSort (…) // Performs Radix sort. **See algorithm below**.
- printSortedData (…) // Output the data (only data, without printing "dummy") in hashTable [Table],
  //sequentially, from 1st queue to the last queue, print 10 strings per text-line. **On your own.**

****
IV. main (…)
****
Step 0: check argc count is correct; check all files, one-by-one, to see if each file can be opened.
inFile, outFile, logFile ← open from argv [1], argv [2], argv [3]
hashTable[2][tableSize] ← establish and initialize as given in the above.
Step 1: firstReading (inFile, logFile)
Step 2: close inFile
reopen inFile.
Step 3: RadixSort (inFile, outFile, logFile)
Step 4: close all files

****
V. firstReading (inFile, logFile)
****
Step 0: logFile ← "*** Entering first Reading ()"
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 while inFile is not empty
Step 4: logFile ← "*** longestStringLength =" // write the value.
Step 5: logFile ← "*** Leaving firstReading ()"

```
*****************************
VI. RadixSort (inFile, outFile, logFile)
*****************************
Step 0: logFile ← " *** Entering RadixSort ()"
Step 1: populateFirstTable ((inFile, outFile, logFile)
Step 2:  currentIndex -- // go to the next character of data toward left
         previousTable ←currentTable
         currentTable ++
         currentTable ← mod (currentTable, 2)
Step 3: logFile ← "*** after swap tables, currentIndex = ; currentTable = , previousTable, =" // print values

        // The Step 4 to Step 5 below is moving data from one queue of the previous table to current table.
Step 4: tableIndex ← 0
Step 5: newNode ← deleteQ (hashTable [previousTable][ tableIndex])
         if newNode != null
                  tempData ← newNode's data
                  hashIndex ← (int) tempData [currentIndex]
                  insertQ (hashTable[currentTable][hashIndex], newNode) // Insert newNode at the tail of queue.
Step 6:  repeat steps 5 until hashTable [previousTable][tableIndex] is empty.
         // Finish move all nodes from one queue in the previous table.

Step 7: tableIndex ++ // continue to process the next queue in the previous hashTable
Step 8: repeat step 4 to step 7 until tableIndex >= tableSize // tableSize is 256, stop at 255.
                           // finish moving all nodes in all queues from previous table to current table.
Step 9: logFile ← "*** Below is the previousTable = **" // write value
         printTable (hashTable[previousTable], logFile)
         logFile ← "*** Below is the currentTable = **" // write value
         printTable (hashTable[currentTable], logFile) // print to logFile.

Step 10: repeat step 2 to step 8 until currentIndex < 0   // continue to the next character until data[0] is processed.

Step 11: outFile ← "*** At the end of Radix sort.  Below is the currentTable = **" // write value
          printSortedData (hashTable[currentTable], outFile)
Step 12: logFile ← "Leaving Radix Sort"

************************************
VI.  populateFirstTable (inFile, outFile, logFile)
************************************
Step 0: logFile ← "*** Entering populateFirstTable ()"
        (string) data
Step 1:  currentIndex←  longestStringLength – 1 // the right most character position of data.
         currentTable ← 0 // the first hashTable
Step 2: data ← read one string from inFile
Step 3: paddedData ← padString (data)
Step 4: newNode ←get a new listNode (paddedData, null)
         hashIndex ← (int) paddedData [currentIndex] //get the ascii value of the charactor
         logFile ← "*** paddedData =  ; currentIndex = ; hashIndex =  ; currentTable =" // print values.
          insertQ (hashTable[currentTable][hashIndex], newNode)
            // insert newNode at the tail of the queue; i.e., at the tail of hashTable[currentTable][hashIndex]

Step 5: repeat Step 2 to Step 4 until inFile is empty
Step 6: logFile ← "*** Finish insert all paddedData into the hashTable [0], the hashTable shown below"
         printTable (hashTable[currentTable], logFile) // print to logFile.
         printTable (hashTable[currentTable], outFile) // print to outFile with caption.
Step 7: logFile ← "*** Leaving populateFirstTable ()"
```