Project 4: (C++) In this project, you are to implement the complete Huffman coding scheme, from compute frequency to text compression and decompression. (Start early on this project!)

Summary of this project:

1) Opens the input text file and computes the characters counts.

2) Constructs the Huffman linked list based on the character counts.

3) Constructs Huffman binary tree, and construct Huffman code.

4) At this point, you have Huffman code array (for encoding) and Huffman binary tree (for decoding.)

5) Closes the input file.

6) Asks the user if he/she wants to compress a text file: ('Y' for yes, 'N' for no.)
   if 'N', exit the program.
   if 'Y' do the following.

7) Asks the user for the name of a text file to be compressed (from console).

8) Opens the text file to be compressed.

9) Calls Encode (…) method to perform compression on the text file using the Huffman code table, and outputs the result to a compressed text file.

10) The name of the compressed file is to be created at run-time, using the original file name with an extension "_Compressed". For example, if the name of the file is "textData1", the name of the compressed file should be "textData1_Compressed". (This can be done simply using string concatenation.)

11) Close the compressed file.

12) To make sure your encoding method works correctly, your program will re-open the compressed file (after it is closed) and call Decode(…) method to perform the de-compression, using the Huffman binary tree. Your program outputs the de-compressed result to a de-compressed text file.

13) The name of the de-compressed file is to be created at run-time, using the original file name with an extension "_deCompressed". For example, if the name of the original text is "textData1", then the name of the de-compressed file should be "textData1_deCompressed".

14) Closed the compressed file and the de-compressed file.
   // after this step your directory should have these files: textData1, textData1_Compressed, and textData1_deCompressed.

15) Repeat 7) to 14) until user type "N" to exit the program.

16) In addition to the input file that you use to compute character counts, you will be provided with two data files: textData1 and textData2 to test your encoding and de-coding of your program.

17) Include in your hard copies PDF file:
   a) Print the input text file //
   b) Print debugFile.
   c) Print textData1, textData1_compressed, textData1_deCompresssed.
   d) Print textData, textData _compressed, textData _deCompresssed

****
Language: C++
Project points: 14 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

9/26/2020 Saturday before midnight
+1 9/22/2020 Tuesday before midnight
-1 for 1 day late: 9/27/2020 Sunday before midnight
-2 for 2 days late: 9/28/2020 Monday before midnight
-14/14 : after 9/28/2020 Monday after midnight
-7/14: does not pass compilation
0/14: program produces no output
0/14: did not submit hard copy.

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

****
I. Input (argv[1]): A text file contains English language.
****

II. Outputs:
****
a) deBugFile to be created at run-time, NOT from argv[]
b) CompressedFile to be created at run-time, NOT from argv[]
c) De-CompressFile to be created at run-time, NOT from argv[]

****
III. Data structure:
****
- A HuffmanCoding class

- A treeNode class
  - chStr (string)
  - frequency (int) //
  - code (string)
  - left (treeNode *)
  - right (treeNode *)
  - next (treeNode *)

  Methods:
  - constructor (chStr, frequency, code, left, right, next)
  - printNode (T, DeBugfile) // Need to print T's code!!!!
    in the format as below:
    (T's chStr, T's frequency, T's code, T's next chStr, T's left's chStr, T 's right's chStr);
    one print per text

- A linkedList class // required

    - listHead (treeNode *)
    - constructor (..)
    - (listNode) findSpot (listHead, newNode) // as in your project 1
    - insertOneNode (spot, newNode) // as in your project 1
    - printList (…) // **Call printNode** for every node on the list from listHead to the end of the list

- A BinaryTree class // required
    - Root (treeNode *)
    - constructor(s)
    - preOrderTraversal (Root, DebugFile) // see algorithm below
    - inOrderTraversal (Root, DebugFile) // on your own
    - postOrderTraversal (Root, DebugFile) // on your own
    - isLeaf (node) // a given node is a leaf if both left and right are null.

- charCountAry[256] (int) // a 1-D array to store the character counts.
- charCode [256] (string) // a 1-D array to store the Huffman code table,
- computeCharCounts (…) // Read a character from input file, use (int) to get index, asci code of the character; charCountAry[index]++. You should know how to do this method.

- printCountAry (…) // print the character count array to DebugFile, in the following format:
    **** >>>(DO NOT print any characters that have zero count.)

    char1 count
    char2 count
    char3 count
    char4 count
    :
    :
    :
- constructHuffmanLList (…) // Algorithm is given below

- constructHuffmanBinTree (…) // Algorithm is given below

- constructCharCode (T, code) // see algorithm below.
    // It will NOT output the codes to an out file,
    // instead the codes will be stored in the charCode array.

- Encode (orgFile, compFile) // See algorithm steps below

- Decode (compFile, deCompFile) // See algorithm steps below

- userInterface ( ) // See algorithm steps below.

****
IV. Main (….)
****

Step 0: nameInFile ← argv[1]
inFile ← open nameInFile
nameDebugFile ← nameInFile + "_DeBug"
DebugFile ← open nameDebugFile
Step 1: computeCharCounts (inFile, charCountAry)
Step 2: printCountAry (charCountAry, DebugFile)
Step 3: constructHuffmanLList (charCountAry, DebugFile) // see algorithm below.
Step 4: constructHuffmanBinTree (listHead, DebugFile) // see algorithm below.
Step 5: constructCharCode (Root, ‘’) // ‘’ is an empty string; see algorithm below.
Step 6: printList (listHead, DebugFile)
Step 7: preOrderTraversal (Root, DebugFile)
inOrderTraversal (Root, DebugFile)
postOrderTraversal (Root, DebugFile)
step 8: userInterface ( ) // given below
step 9: close all files.

****
V. constructHuffmanLList (charCountAry, DebugFile)
****

Step 0: listHead ← get a newNode as the dummy treeNode with ("dummy" ,0), listHead to point to.

Step 1: index ← 0

Step 2: if charCountAry[index] > 0

chr ← char (index)
prob ← charCountAry[index]
newNode ← get a new listNode (chr, prob, ‘’, null, null, null) // ‘’ is an empty string
insertNewNode (listHead, newNode) // use algorithm steps given in class
printList (listHead, DebugFile) // debug print
// print the list to DebugFile, from listHead to the end of the list
// using the format given in the above.

Step 3: index ++

Step 4: repeat step 2 – step 3 while index < 256.

****
VI. constructHuffmanBinTree (listHead, outFile)
****

Step 1: newNode ← create a treeNode // the following five assignments may be done in the constructor.
newNode's prob ← the sum of prob of the first and second node of the list // first is the node after dummy
newNode's chStr ← concatenate chStr of the first node and chStr of the second node in the list
newNode's left ← the first node of the list
newNode's right ← the second node of the list
newNode's next ← null

Step 2: insertNewNode (listHead, newNode)
Step 3: listHead's next ← listHead .next.next.next // third node after dummy node
Step 4: printList (listHead, outFile) // debug print
Step 5: repeat step 1 – step 4 until the list only has one node after the dummy node
Step 6: Root ← listHead's nex

****
VII. constructCharCode (T, code)
****

```
if  isLeaf (T)
      T's code ← code;
      Index ←cast T's chStr  to integer
      charCode[index] ← code
else
     constructCode (T's  left, code + "0") //string concatenation
     constructCode (T's  right, code + "1") //string concatenation
```

****
VIII. userInterface ( )
****

step 0: nameOrg (string)
nameCompress (string)
nameDeCompress (string)
yesNo (char)

Step 1: yesNo ← ask user if he/she want to encode a file

```
if yesNo == 'N'
        exit the program
else
        nameOrg ← as the user for the name of the file to be compressed
```

step 2: nameCompress ← nameOrg + "_Compressed"
nameDeCompress ← nameOrg + "_DeCompress"

Step 3: orgFile ← open nameOrg file for read
compFile ← open nameCompress file for write
deCompFile ← open nameDeCompress file for write

Step 4: Encode (orgFile, compFile) // see algorithm steps below

Step 5: close compFile

Step 6: re-open compFile

step 7: Decode (compFile, deCompFile) // see algorithm steps below

Step 8: close orgFile, compFile and deCompFile

step 9: repeat step 1 to step 8 until yesNo == 'N' in which the program exit

****
IX. Encode (inFile, outFile)
****

step 1: charIn ← get the next character from inFile, one character at a time

step 2: index ← cast charIn to integer

step 3: code ← charCode[index]

step 4: write index and code to outFile

step 5: repeat step 1 to step 4 until eof inFile

****
X. Decode (inFile, outFile)
****

step 1: Spot ← Root // root of the constructed Huffman binary tree.

step 2: if isLeaf (Spot)
write Spot's chr to outFile
spot ← Root // place spot back to Root

step 3: oneBit ← read a character from inFile // should be either '0' or '1'

step 4: if oneBit == '0'
Spot ← Spot's left
else if oneBit == '1'
Spot ← Spot's right
else
output error message: "Error! The compress file contains invalid character!"
exit the program.

step 5: repeat step 2 to step 4 until end of inFile

step 6: if eof inFile but Spot is not a leaf, output error message: "Error: The compress file is corrupted!"

****
XI. preOrderTraveral (T, outFile) // In recursion
****

```
if isLeaf (T)
    printNode (T, outFile) // output to outFile
else
    printNode (T, outFile)
    preOrderTraveral (T's left, outFile)
    preOrderTraveral (T's right, outFile)
```