Project 4: (Java) Huffman coding part 2 (continuation of part 1). You are to implement the complete Huffman coding scheme, from compute frequency to encoding and decoding.

Project summary:

I. Part 1: Your project 3.

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 from the linked list.
4) Traverse the Huffman binary tree in
    a) Pre-order
    b) In-order
    c) Post-order

II. Part 2:

5) Construct Huffman code.
6) At this point, you have Huffman code array (for encoding) and Huffman binary tree (for decoding.)
7) Closes the input file.
8) 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.
9) Asks the user for the name of a text file to be compressed (from console).
10) Opens the text file to be compressed.
11) Calls Encode (…) method to perform compression on the text file using the Huffman code table, and outputs the result to a compressed text file.

12) The name of the compressed file is to be created at run-time, using the original file name with an extension "_Compressed.txt". For example, if the name of the file is "test1", the name of the compressed file should be "test1_Compressed.txt". (This can be done simply using string concatenation.)

13) Close the compressed file.
14) 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.

15) The name of the de-compressed file is to be created at run-time, using the original file name with an extension "_deCompressed.txt". For example, if the name of the original text is "test1", then the name of the de-compressed file should be "test1_deCompressed.txt".

16) Closed the compressed file and the de-compressed file.
// after this step your directory should have these files: Data, test1_Compressed, and test1_deCompressed (all are .txt files)
17) Repeat 8) to 16) until user type "N" to exit the program.

18) In addition to the input file that you use to compute character counts, you will be provided with two test data files: test1 and test2 to test your encoding and de-coding of your program.

19) Include in your hard copies PDF file:
    a) Print input text file
    b) Print outFile.
    c) Print test1, test1_compressed, and test1_deCompresssed.
    d) Print test2, test2 _compressed, and test2_deCompresssed

****

Language: Java
Project points: 10 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

+1 (11/10 pts): early submission, 10/14/2022, Friday before midnight
-0 (10/10 pts): on time, 10/18/2022, Tuesday before midnight
-1 (9/10 pts): 1 day late, 10/19/2022, Wednesday before midnight
-2 (8/10 pts): 2 days late, 10/20/2022, Thursday before midnight
(-10/10 pts): non submission, 10/20/2022, Thursday 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 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.

****

I. Inputs: inFile (args [0]): A text file contains English language.

****

II. Outputs: outFile (args[1]): As the output of part 1 (in project 3) and as this program say so.

****

III. Data structure:

****

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

  Methods:
  - constructor (chStr, frequency, code, left, right, next)
  - printNode (T, outFile) // print the node 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 textline.

- A linkedList class // required
  - (treeNode) listHead // point to a dummy node!

  Method:
  - constructor (…) // create a new treeNode ("dummy", 0, "", null, null, null)
    //as dummy node for listHead to point to.
  - insertNewNode (…) // call findSpot then insert a newNode into the linked list after spot.
    // Algorithm is given in the specs of project 3 part 1.
  - (treeNode) findSpot (listHead, newNode) // The same idea as the findSpot method in your hash table project,
    // Here, Spot start from listHead instead of hashTable[index] and the comparison is
    // if spot.next.frequency < newNode.frequency
  - printList (…) // **Call printNode** for every node on the list from dummy node to the end of list.
- A BinaryTree class // required
  - (treeNode) Root

  Method:
  - constructor:
  - (bool) isLeaf (node) // returns true if both node's left and right are null, return false otherwise.
  - preOrderTraversal (Root, outFile) // Algorithm is given in the specs of project 3 (part 1).
  - inOrderTraversal (Root, outFile) // Algorithm is given in the specs of project 3 (part 1).
  - postOrderTraversal (Root, outFile) // on your own.

- A HuffmanCoding class
  - (int) charCountAry [256] // a 1-D array to store the character counts.
  - (string) charCode [256] // a 1-D array to store the Huffman code table

  Method:
  - computeCharCounts (…) // Read a character from input file, use (int) to get index, ascii 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 need to print any characters that have zero count.)

    char1 count
    char2 count
    char3 count
    char4 count
    :
    :
  - constructHuffmanLList (…) // Algorithm is given in the specs of project 3 (part 1).
  - constructHuffmanBinTree (…) // Algorithm is given in the specs of project 3 (part 1).
  - constructCharCode (…) // see algorithm below.
  - Encode (…) // See algorithm steps below
  - Decode (…) // See algorithm steps below
  - userInterface (…) // See algorithm steps below.

****
IV. Main (….)
****
Step 0: (string) nameInFile ← args[0]
inFile ← open nameInFile
outFile ← open args[1]
Step 1: computeCharCounts (inFile, charCountAry)
Step 2: printCountAry (charCountAry, outFile)
Step 3: constructHuffmanLList (charCountAry, outFile)
Step 4: constructHuffmanBinTree (listHead, outFile)
Step 5: printList (listHead, outFile)
Step 6: preOrderTraversal (Root, outFile)
inOrderTraversal (Root, outFile)
postOrderTraversal (Root, outFile)
Step 7: constructCharCode (Root, "")
step 8: userInterface (Root, outFile) // given below
step 9: close all files.

****
V. constructCharCode (T, code)
****
if isLeaf (T)
T's code ← code;
Index ←cast T's chStr to integer
charCode[index] ← code

else
constructCharCode (T's left, code + "0") //string concatenation
constructCharCode (T's right, code + "1") //string concatenation

****
VI. userInterface (Root, outFile)
****
step 0: (string) nameOrg
(string) nameCompress
(string) nameDeCompress
(char) yesNo
Step 1: yesNo ← ask user if he/she want to encode a file
if yesNo == 'N'
exit the program

else
nameOrg ← ask the user for the name (without .txt) of the file to be compressed

step 2: nameCompress ← nameOrg + "_Compressed.txt"
nameDeCompress ← nameOrg + "_DeCompress.txt"
nameOrg ← nameOrg + ".txt"
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, outFile) // see algorithm steps below
Step 5: close compFile
Step 6: re-open compFile
step 7: Decode (compFile, deCompFile, Root) // 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

****
VII. Encode (orgFile, compFile, outFile)
****
step 1: charIn ← get the next character from orgFile, 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**
write code to **compFile**
step 5: repeat step 1 to step 4 until end of orgFile

****
VIII. Decode (compFile, deCompFile, Root)
****
step 1: Spot ← Root
step 2: if isLeaf (Spot)
deCompFile ← write Spot's chr to deCompFile
spot ← Root // place spot back to Root
step 3: oneBit ← read a character from compFile // should be either '0' or '1'
step 4: if oneBit == '0'
Spot ← Spot's left
else if oneBit == '1'
Spot ← Spot's right
else
console ← output error message: "Error! The compress file contains invalid character!" exit the program.
step 5: repeat step 2 to step 4 until end of compFile

step 6: if end of compFile but Spot is not a leaf
console ←output error message: "Error: The compress file is corrupted!