Project 3: (Java) Huffman coding part 1. You are to implement the three steps of the Huffman coding scheme: compute frequency, construct ordered link-list, and construct Huffman binary tree.

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

****

Language: Java
Project points: 8 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):
    +1 (9/8 pts): early submission, 9/28/2022, Wednesday before midnight
    -0 (8/8 pts): on time, 10/1/2022, Saturday before midnight
    -1 (7/8 pts): 1 day late, 10/2/2022, Sunday before midnight
    -2 (6/8 pts): 2 days late, 10/3/2022, Monday before midnight
    (-8/8 pts): non submission, 10/3/2022, Monday 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. Input (args [0]): A text file contains English language.

****

II. Outputs: outFile1(args[1]): for all output the program produces.

****

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.
      // See algorithm below.

    - (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) // see algorithm below
    - inOrderTraversal (Root, outFile) // see algorithm below
    - 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 below
    - constructHuffmanBinTree (…) // Algorithm is given below

****

IV. Main (….)

****

Step 0: inFile, outFile ← args[0], args[0]

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: close all files.

****

V. constructHuffmanLList (charCountAry, DebugFile)

****

```
Step 0:  listHead ← get a newNode as the dummy treeNode with (“dummy” ,0,  null, null, null)
Step 1: index ← 0
Step 2: if charCountAry[index] > 0
                    chr  ← char (index)
                    frequency  ← charCountAry[index]
                    newNode ← get a new listNode (chr, frequency, ‘’,  null, null, null) // ‘’ is an empty string
                    spot ← findSpot ((listHead, newNode)
                    insertNewNode (listHead, newNode)
                    printList (listHead, outFile) // debug print
Step 3: index ++
Step 4: repeat step 2 to step 3 while index < 256.
```

****

VI. insertNewNode (listHead, newNode)

****

```
Step 0: Spot ← findSpot (listHead, newNode)
Step 1:  newNode.next ← Spot.next
          Spot.next ← newNode
```

****

VI. constructHuffmanBinTree (listHead, outFile)

****

```
 Step 1: newNode ← create a treeNode  // the following five assignments may be done in the constructor.
      newNode’s frequency ← the sum of frequency of the first node and second node of the list
                    // first node 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 to step 4 until the list only has one node after the dummy node
 Step 6:  Root ← listHead’s next
```

****

VII. 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)
```

****

VIII. inOrderTraveral (T, outFile) // In recursion

****

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