Project4: Implementation of Huffman coding step 2, and step 3. The remaining steps will be in the future projects. (Project 4 will be implemented in Java, the remain steps will be in C++.)

Step 2: Construct Huffman ordered linked list using insertion sort.

Step 3: Construct Huffman binary tree

After the tree construction, you are to traverse the Huffman binary tree in the order of:

a) pre-order
b) in-order
c) post-order

****

Language: Java

Due Date: Soft copy: 2/26/2019 Tuesday before midnight
1 day late: -1 pt 2/27/2019 Wednesday before midnight
After 2/27/2019 -12 pts for all students who did not submit soft copy on time

Due Date: Hard copy: 2/28/2019 Thursday in class,
-1 pt for late hard copy submission.
All projects without hard copy will receive 0 pts even you had submit soft copy on time.

****

I. Input (argv[1] Java): A file contains a list of <char prob> pairs with the following format. The input prob are integer, has been multiplied by 100, i.e., a prob equal to .40 will be given as 40, char should be treated as string.

****

char$_1$ prob$_1$
char$_2$ prob$_2$
char$_3$ prob$_3$
:
:
char$_n$ prob$_n$

****

II. Outputs: four output files. All need to be included in your Hard COPY

****

- outFile1 (argv[2]): for your debugging outputs. See output format below for detail.
- outFile2 (argv[3]): A text file contain the pre-order traversal of the Huffman binary tree
- outFile3 (argv[4]): A text file contain the in-order traversal of the Huffman binary tree
- outFile4 (argv[5]): A text file contain the post-order traversal of the Huffman binary tree

****

III. Data structure: You MUST have all the object classes as given below.

****

- A treeNode class // required

    - chStr (string)
    - prob (int)
    - next (treeNode *)
    - left (treeNode *)
    - right (treeNode *)
    - code (string)

Methods:

    - constructor(s)
    - printNode (T)

    // given a node, T, print T's chStr, T's prob, T's next chStr, T's left's chStr, T 's right's chStr
    // print one treeNode per text line

- linkedList class // required

    - listHead (treeNode *)
    - constructor (..)
    - findSpot (…) // Use the findSpot algorithm steps taught in class.
    - insertOneNode (spot, newNode)

            // inserting newNode between spot and spot.next.
            // only need two statements (was given in class)

    - printList (…)

    // print the list to outFIle1, from listHead to the end of the list in the following format:

    listHead -->("dummy", 0, next's chStr$_1$)-->(chStr$_1$, prob$_1$, next's chStr$_2$)...... --> (chStr$_j$, prob$_j$, NULL)--> NULL

    For example:
    listHead -->("dummy", 0, "b")-->("b", 5, "a") -->("a", 7, "d")........... --> ("x", 45, NULL)--> NULL

- A HuffmanBinaryTree class // required

    - Root (treeNode *)

    - Method:
        - constructor(s)
        - constructHuffmanLList (inFile, outFile)

```
- constructHuffmanBinTree (listHead)
- preOrderTraversal (Root, outFile) // algorithm is given below
- inOrderTraversal (Root, outFile)
- postOrderTraversal (Root, outFile)
- isLeaf (node)// a given node is a leaf if both left and right are null.
```

****

IV. Main (….)

****

```
Step 0: inFile ← open input  file from argv[1]
        outFile1, outFile2, …, outFile5 ← open from argv[2], …, argv[6]

Step 1: constructHuffmanLList  (inFile, outFile1) // see below

Step 2:  constructHuffmanBinTree (listHead, outFile1) // see below

Step 3: preOrderTraversal (Root, outFile2)  // In recursion, algorithm is given below

step 4: inOrderTraversal (Root, outFile3) // In recursion

step 5: postOrderTraversal (Root, outFile4)// In recursion

step 6: close all files
```

****

V. constructHuffmanLList (inFile, outFile)

****

```
Step 1:  listHead ← get a newNode as the dummy treeNode with ("dummy" ,0),  listHead to point to.

Step 3: chr  ← get  from inFile
        Prob  ← get  from inFile
        newNode ← get a new listNode
        newNode's chStr ← chr
        newNode's prob ← Prob
        newNode's next ← null

Step 4: spot ← findSpot (listHead, newNode) // see algorithm below

step 5:  insertOneNode (spot, newNode) // insert newNode after spot
```

```
Step 6: printList (listHead, outFile)
        // print the list to outFile, from listHead to the end of the list
        // using the format given in the above.

Step 7: repeat step 3 – step 5 until the end of inFile .
```

****

VI. constructHuffmanBinTree (listHead, outFile)

****

```
 Step 1: newNode ← create a treeNode
      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

Step 2: spot ← findSpot(listHead, newNode)

        insertOneNode (spot, newNode)  // inserting newNode between spot and spot.next.
                                        // only need two statements.
       listHead's next's next  ← listHead's next's next's next's next//  listHead is pointed to dummy node,
                           //therefore,  listHead's next's next is the dummy's next

       printList (listHead, outFile)

Step 3: repeat step 1 – step 2 until the list only has one node left which is the newNode

Step 4:  Root ← newNode
```

****

VII. preOrderTraveral (T, outFile) // In recursion

****

```
        if (T is null)
           do nothing
        else
            printNode (T) // output to outFile,  see printing format in treeNode in above
            preOrderTraveral (T's left, outFile)
            preOrderTraveral (T's right, outFile)
```

****
VIII. inOrderTraveral (Root, outFile) // In recursion
****

You should know how to write this method.

****
VIIII. postOrderTraveral (Root, outFile) // In recursion
****

You should know how to write this method.