Project 3 (in C++): You are to implement the Huffman Tree/Code construction as taught in class.

*** You will be given 3 data files: data1, data2, and data2. data1 is the first example taught in the class; data2 is the exercise example given and you did in class; data3 has a larger ascii set.

What to do as follows:
1) Implement your program based on the specs given below.
2) Run and debug your program with data1 until your program produces the same result
    as the first example given in class (i.e., check the code table and compare your program's pre-order, in-order, and post-order with your own.)
3) Then run your program with data2 to see if your program produces the same result as you did in class.
4) When both results are good, then run your program with data3.

Include in your PDF file:
- Cover page
- source code
- outFile1 and outFile2 with data1
- outFile1 and outFile2 with data2
- outFile1 and outFile2 with data3

****
Language: C++
****
Project points: 10 pts
Due Date: <u>Soft copy (*.zip) and hard copies (*.pdf)</u>:
+1 (11/10 pts): early submission, 3/1/2022, Tuesday before midnight
-0 (10/10 pts): on time, 3/5/2022 Saturday before midnight
-1 (9/10 pts): 1 day late, 3/6/2022 Sunday before midnight
-2 (8/10 pts): 2 days late, 3/7/2022 Monday before midnight
(-10/10 pts): non-submission, 3/7/2022 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 Blackboard.

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

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

char_1 prob_1
char_2 prob_2
char_3 prob_3
:
:
char_n prob_n

****
II. Outputs:
****
- outFile1 (argv[2]): for printing
    1) the final sorted linked list,
    2) the Huffman <char, code> pairs
    3) pre-order traversal, in-order traversal, and post-order traversal of Huffman binary tree
- outFile2 (argv[3]): All other intermediate results and debugging prints

****
III. Data structure:
****

- A HtreeNode class
  - (string) chStr
  - (int) prob
  - (string) code
  - (HtreeNode *) left
  - (HtreeNode *) right
  - (HtreeNode *) next
  - constructor (chStr, prob, code, left, right, next)
  - printNode (T)

  // given a HtreeNode, T, print T's chStr, T's prob, T's next chStr, T's left's chStr, T 's right's chStr
  // For example, if T's chStr is "A", T's prob is 10, T's code is 011, next's chStr is "C", left's chStr is "D", right's chStr is "E"), then
  (A, 10, 011, C, D, E) →

- A HuffmanBinaryTree class
  - (HtreeNode *) listHead
  - (HtreeNode *) Root

  Methods:
  - constructor(s) // Does whatever it needs
  - findSpot (…) // as in your project 1 and 2
  - listInsert(listHead , newNode) // as in your project 1 and 2
  - (HtreeNode *) constructHuffmanLList (inFile, outFile) // Algorithm is given below.
  - (HtreeNode *) constructHuffmanBinTree (listHead) // Algorithm is given below.
  - preOrderTraversal (Root, outFile) // Algorithm is given below.
  - inOrderTraversal (Root, outFile)
  - postOrderTraversal (Root, outFile)
  - constructCharCode (T, code, outFile) // Algorithm is given below
  - isLeaf (node) // a given node is a leaf if both left and right are null.
    // You should know how to do this!
  - printList (…) // Call printNode (listHead) to print every node in the list from listHead to the end of the list
    // including the dummy node
    For example:
    listHead → ("dummy", 0, next node's chStr, left's chStr, right's chStr) → . . . . . → NULL

****
IV. Main (….)
****

```
Step 0: inFile ← open input file from argv[1]
        outFile1, outFile2, outFile3 ← open from argv[2], argv[3], argv[4]
Step 1: listHead ← constructHuffmanLList (inFile, outFile2)
Step 2: printList (listHead, outFile1)
Step 3: Root ← constructHuffmanBinTree (listHead, outFile2)
Step 4: constructCharCode (Root, '', outFile1) // '' is an empty string
Step 5: preOrderTraversal (Root, outFile1)
Step 6: inOrderTraversal (Root, outFile1)
Step 7: postOrderTraversal (Root, outFile1)
Step 8: close all files
```

****
V. (HtreeNode *) constructHuffmanLList (inFile, outFile3)
****

```
Step 1:  listHead ← get a HtreeNode (“dummy”, 0, ‘’, null, null, null) as the dummy node  // use constructor
Step 2: chr  ← get from inFile
        prob  ← get from inFile
Step 3: newNode ← get a new HtreeNode (chr, prob, ‘’, null, null, null) // ‘’ is an empty string
Step 4: Spot ← findspot (listHead, newNode)
Step 5: listInsert (Spot, newNode)
Step 6: printList (listHead, outFile3)
Step 7: repeat step 2 – step 6 until the end of inFile
Step 8: return listHead
```

****
VI. (HtreeNode *) constructHuffmanBinTree (listHead, outFile3)
****

```
Step 1: newNode ← get a new HtreeNode (chr, prob, ‘’, null, null, null)
        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 after dummy node
        newNode’s right ← the second node after dummy node
        newNode’s next ← null
Step 2: Spot ← findspot (listHead, newNode)
Step 3: listInsert (Spot, newNode)
Step 4: listHead’s next ← the third node after dummy node // dummy next points to the 3rd node
Step 5: printList (listHead, outFile3)
Step 6: repeat step 1 – step 4 until the list only has one node left after the dummy node
Step 7: return listHead’s next
```

****
VII. constructCharCode (T, code, outFile1)
****

```
if isLeaf (T)
    T’s code ← code
    outFile1 ← output   T’s  chStr  and T’s code to outFile1   // space between the two and one per text line
else
    constructCharCode (T’s left, code + “0”, outFile1) //string concatenation
    constructCharCode (T’s right, code + “1”, outFile1) //string concatenation
```

****
X. preOrderTraveral (T, outFile1) // In recursion
****

```
if isLeaf (T)
    printNode (T) // output to outFile1, see printing format in treeNode class in above
else
    printNode (T)
    preOrderTraveral (T’s left, outFile1)
    preOrderTraveral (T’s right, outFile1)
```

****
XI. inOrderTraversal (Root, outFile1) // In recursion
****

Code is similar to preOrder except printNode() is in the middle

****
XII. postOrderTraveral (Root, outFile1) // In recursion
****

Code is similar to preOrder except printNode() is in the last