Project 1 (in C++): Linked-list implementation of Stack, Queue, and ordered list. Your program will perform the following tasks:

a) Build a stack: i) open an input file; ii) read one data at the time from the input file; iii) create a new node with data;
iv) push (newNode), on top of the stack. Print the entire stack to outFile1 after it is built.

b) Build a queue: i) pop the stack; ii) print the data in the node to outFile1; iii) insertQ (node), at the back of the queue.
Print the entire queue to outFile2 after it is built.

c) Build a list: i) remove a node from the front of the queue; ii) print the data in the node to outFile2; iii) insertLL (node) to the list, **in ascending order**; iv) print the entire list to outFile3 after it is built.

What you need to do:

1. Implement your program with respect to the specs given below and debug your program until your program compiles.
2. You will be given two data files: data1 and data2; data1 contains only a few words and data2 contains more words.
3. Run your program using data1 and eyeball the stack, queue, and list outputs of your program for correctness.
4. When your program produces correct output from data1, then run your program using data2.

*** Include in your hard copy *PDF.pdf file as follows:

- Cover page
- Source code
- inFile // with caption, i.e., "*** below is input file ***"
- outFile1 // with caption, i.e., "*** below is outFile1 ***"
- outFile2 // with caption, i.e., "*** below is outFile2 ***"
- outFile3 // with caption, i.e., "*** below is outFile3 ***"
- logFile // with caption, i.e., "*** below is logFile ***"

Note: You must use argv to open input file and 4 output files. (Your project0B show how to open file via argv)
-3 points if you hard-code your file names!

****

Language: C++

****

Project points: 10 pts

Project name: Linked-list implementation of stacks, queues, and lists

Due Date: Soft copy (*.zip) and hard copies (*.pdf):
(10/10 pts): on time, 9/11/2025. Thursday before midnight
(-10/10 pts): non-submission, 9/11/2025. Thursday after midnight

*** Name your soft copy and hard copy files using the naming convention given in Project Submission Requirements.

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

Email subject: (323.mw or 323.tth) first name last name <Project 1: Linked-list implementation of stacks, queues, and lists (C++)>

*** Inside the email body includes:

- Your answer to the 5 questions given in your email body. (-1 if not include.)
- Screen recoding link. (-1 if not submit screen recording).

****

I. Inputs:

****

1) inFile (use argv [1]): a text file contains a sequence of English words (strings), not in any particular format.

****

II. Outputs: There will be 4 output files.

1) outFile1 (use argv [2]): for stack outputs.
2) outFile2 (use argv [3]): for queue outputs.
3) outFile3 (use argv [4]): for list output.
4) logFile (use argv [5]): to monitor the progress of your program.

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

```
- listNode class
        - (string) data
        - (listNode*) next
      Methods:
        - constructor (data) //Assign listNode' data with given data and assign listNode's next null, ie.
                //this->data = data; this->next = nullptr;
        - printNode (node)  // print in the format as below:
           (node's data,  node's next's data) →
           For example:  (8, 11) →

- LLStack class
        - (listNode*) top
    Methods:
        - constructor (...) // create a listNode for top  (use new method) with "dummy' as its data,  i.e.,
                        //top ← new listNode ("dummy", null)
         - push (newNode) // insert newNode after top->next (code is given in class.)
        - (bool) isEmpty () // if top's next is null returns true, otherwise returns false.
         - listNode* pop () // if stack is not empty, removes and returns the node after top->next (code is given in class.)
                        // otherwise, print "stack is empty" to outfile1.
        - buildStack (inFile) // build a stack from the data in inFile. See algorithm steps below.
        - printStack (…) // The method calls printNode(...) to print all nodes in the stack to outFile1, in the format:
                Top → (dummy, next's data) → (data, next's data) → ...... → (data, NULL) → NULL
                For example:
                Top →(dummy, story) → (story, the) → (the, tells) → (tells, sea) → ...... → (the, NULL) → NULL
// print the entire stack to outFile1 //See algorithm below.
- LLQueue class
        - (listNode *) head // head always points to the dummy node!
        - (listNode *) tail // tail always points to the last node of the queue.

    Methods:
        - constructor(...) // // create a listNode for head (use new method) with "dummy' as its data and set tail to head,  i.e.,
                        head ← new listNode ("dummy", null)
                        tail ← head
        - insertQ (…) // insert the newNode after the tail of Q, i.e., after the node points by tail, i.e.,
                        // newNode's next ←Q's tail's next
                        // Q's tail's next ← newNode
                        // Q's tail ← newNode
        - (listNode *) deleteQ (…) // if Q is not empty, delete and return the node after Q.head->next).
                                // See algorithm below.
        - (bool) isEmpty (…)// Returns true if tail == head, returns false otherwise.
        - buildQueue (…) // build a queue from nodes in the stack. See algorithm below.
        - printQ (…) // The method calls printNode(...) to print all nodes in the queue to outFile2, in the format:
                head → (dummy, next's data) → (data, next's data) → ...... → ( data, NULL) → NULL
                For example:
                Head →(dummy, the) → (the, old → (old, Man) → (Man, and → ...... → (story, NULL) ← Tail
-  LLlist class
        - (listNode *) listHead

        Methods:
        - constructor (...) // create a listNode for listHead (use new method) with "dummy' as its data,  i.e.,
                        listHead ← new listNode ("dummy", null)
        - (listNode *) findSpot (…) // the method finds the location, called Spot, in the list to insert newNode; it returns
                Spot; See algorithm below.
```

```
- insertOneNode (…)  // inserts newNode between spot and spot's next.
                     // newNode's next ← spot's next
                       Spot's next ← newNode
- buildList (…) // build a linked list from nodes in the queue. See algorithm below.

- printList (...) // The method calls printNode(...) to print all nodes in the list to outFile3, in the following format:

      listHead → (dummy, next's data) → (data, next's data) → ...... → ( data, NULL) → NULL
      For example:
      listHead →(dummy, and) → (and, Man) → (Man, old) → (old, sea)............ → (the, NULL) → NULL
```

****

IV. main (…)

****

```
Step 0: check argc count is correct; check all files, one-by-one, it each can be opened. (Do as in your project0B)
        inFile ← open input file from argv [1]
        outFile1, outFile2, outFile3, logFile ← open from argv [2], argv [3], argv [4], argv [5]
        check all files can be opened.

Step 1:  S ←  define S is a LLStack
Step 2: logFile ← "calling buildStack ()"
        buildStack (S, inFile, logFile)
        printStack (S, outFile1)

Step 3:  Q ← define Q is a LLQueue.
Step 4: logFile ← "calling buildQueue ()"
        buildQueue (S, Q, outFile1, logFile)
        printQ (Q, outFile1)

Step 5:  LL ← define LL is a LLlist
Step 6: logFile ← "calling buildList ()"
        buildList (Q, LL, outFile2, logFile)
Step 7: logFile ← "Printing list"
        printList (LL, logFile)
        printList (LL, outFile3)
Step 8: close all files
```

****

V. buildStack (S, inFile, logFile)

****

```
Step 0: logFile ← write "entering buildStack () !"
Step 1: data ← read a string from inFile
        logFile ← "input data is " // write data  one data per text line
Step 2: newNode ← creates a listNode for data using constructor // new listNode(data, null)
Step 3: push (S, newNode)
Step 4: repeat step 1 to step 3 until inFile is empty.
Step 5: logFile ← write "leaving buildStack ()!"
```

```
*******************************
VI.  buildQueue (S, Q, outFile1, logFile)
******************************
Step 0: logFile ← write "entering buildQueue ()!"
Step 1: newNode ← pop (S)
        logFile ← "after pop stack, newNode's data is" //write newNode's data; one data per text line.
Step 2: outFile1 ← "after pop stack, newNode's data is" //write newNode's data; one data per text line.
Step 3: insertQ (Q, newNode)
Step 4: repeat step 1 to step 3 until S is empty.
Step 5: logFile ← write "leaving buildQueue ()!"
```

```
*******************************
VII. buildList (Q, LL, outFile2, logFile)
*******************************
Step 0: logFile ← "entering buildList ()"
Step 1: newNode ← deleteQ (Q, outFile2, logFile)
Step 2: outFile2 ← "delete a node from Q, newNode's data is" // write newNode's data, per text-line.
        logFile ← "delete a node from Q, newNode's data is" // write newNode's data, per text-line.
Step 3: Spot ← findSpot (LL, newNode, logFile)
Step 4: insertOneNode (Spot, newNode)
Step 5: repeat step 1 to step 4 until Q is empty.
Step 6: logFile ← "leaving buildList ()!"
```

```
*******************************
VIII. (listNode*) findSpot (LL, newNode, logFile)
*******************************
Step 0: logFile ←  "entering findSpot ()"
Step 1: Spot ← LL.listHead
Step 2: if Spot's next != null && Spot's next's data <  newNode's data // use str1.compare (str2) < 0
                 Spot ← Spot's next
Step 3: repeat step 2 until condition failed
Step 4: logFile ← "Spot's data is" //write Spot's data
Step 5: logFile ← "leaving findSpot ()"
Step 6: return Spot
```

```
*******************************
IV. (listNode*) deleteQ (Q, outFile2, logFile)
*******************************
Step 0: logFile ← "entering deleteQ ()"
Step 1: if isEmpty (Q) // Q's tail == Q's head
          outFile2 ←  "Q is empty"
          logFile ← "Q is empty"
          return null
Step 2:  (listNode*) temp ← Q.head's next
Step 3:  if tail == temp
            tail ←head
Step 4: Q.head's next ← temp's next
Step 5: temp's next ← null
Step 6: logFile ← write temp's data
         logFile ← write "leaving deleteQ ()"
Step 7: return temp
```