Project 1 (in Java): Linked-list implementation of Stack, Queue, and ordered list. (A simple review of what you had learn in csci 313)

****
Language: Java //A Java tutorial will be posted in Google Classroom of this class
****
Project points: 10 pts
Due Date: Soft copy (*.zip) and hard copies (*.pdf):

9/3/2020 Thursday before midnight
+1 early submission: 8/31/2020 Monday before midnight
-1 for 1 day late: 9/4/2020 Friday before midnight
-2 for 2 days late: 9/5//2020 Saturday before midnight
-10/10 : after 9/5/2020 Saturday after midnight

*** Name your soft copy and hard copy files using the naming convention as given in the project submission requirement given in the syllabus.

*** All on-line submission MUST include Soft copy (*.zip) and hard copy (*.pdf) in the same email attachments with correct file names given in the syllabus; otherwise, it would not count as submission.

****
I. Inputs:
****
1) inFile (use args[0]): a text file contains a set of positive integers, not in any particular format.

****
II. Outputs: There will be three output files.
1) outFile1 (use args[1]): for stack outputs.
2) outFile2 (use args[2]): for queue outputs.
3) outFile3 (use args[3]): for orderedList output.

****
III. Data structure:
****
- listNode class
    - data (int)
    - next (listNode)

  Method:
    - constructor (data) //create a node with given data

- LLStack class

    - top (listNode)

    Methods:
    - constructor (...) // create a new stack with a dummy node, set the data in the dummy node to -99999 and next to null; and let top points to the dummy node

    - push (S, newNode) // puts newNode at the top of S

- listNode pop(S) // if S is not empty, deletes and returns the front node of S.

- bool isEmpty (S) // check to see if the stack S is empty

- printTop (S) // print the data of node on the top of S.

- buildStack (inFile) // build a stack from the data in inFile. See algorithm steps below.

- dumpStack (S, outFile1) // Output to outFile1, the data of node on the top of S, then pop(S). repeat the process until S is empty.

- LLQueue class
  - head (listNode) // head always points to the dummy node!
  - tail (listNode) // tail always points to the last node of the queue.

  Methods:
  - constructor(...) // create a new Queue with a dummy node, set the data in the dummy node to -99999 and next to null; and let both head and tail point to the dummy node

  - insertQ (Q, newNode) // insert the newNode after the tail of Q

  - listNode deleteQ (Q) // if Q is not empty, delete the front node from Q and returns the // deleted front node.
  - bool isEmpty (Q)// check to see if Q is empty, i.e., tail points to the dummy node.

  - (listNode) buildQueue (inFile) // build a queue from the data in inFile. See algorithm steps below.

  - dumpQueue (Q, outFile2) // Output the data in the node after the dummy node to outFile2, then delete the node after the dummy node. Repeat the process until Q is empty

- LList class

  - listHead (listNode)

  Methods:
  - constructor (...)// create new list with a dummy node, set the data in the dummy node to -99999. and let listHead points to the dummy node

  - (listNode) findSpot (listHead, newNode)// the method uses a "pointer", called Spot, Spot walks from the beginning of the list until the data in Spot.next node is >= the data in newNode, it returns Spot.

  - insertOneNode (spot, newNode)
    // inserting newNode between spot and spot.next.
    // newNode.next ← spot.next
    // spot.next ← newNode

- (listNode) buildList (inFile) // build an orderedList from the data in inFile. See algorithm steps below.
- printList (...)
// from listHead (including dummy node), the method calls printNode(...) and output to outFile3, to the end of the list in the following format:

listHead -->(99999, next's $data_1$)-->($data_1$, next's $data_2$)..... --> ($data_j$, NULL)--> NULL

For example:
listHead -->(99999, 0, 8)-->(8, 11) -->(11, 21)..... --> (87, NULL)--> NULL

****
IV. main ()
****
Step 0: inFile ← open input file from args[0]
outFile1, outFile2, outFile3 ← open from args[1], args[2], args[3]

Step 1: S ← buildStack(inFile) // Algorithm steps is given below

Step 2: dumpStack(S, outFile1)
// on your own; see the method's description in the above
Step 3: close inFile
Step 4: re-open inFile

Step 5: Q ← buildQueue(inFile) // Algorithm steps is given below

Step 6: dumpQueue(Q, outFile2)
// on your own; see the method's description in the above
Step 7: close inFile
Step 8: re-open inFile

Step 9: LL ← buildList(inFile) // Algorithm steps is given below

Step 10: printList(LL, outFile3)
// on your own; see the method's description in the above
Step 11: close all files

****
V. (listNode) buildStack (inFile)
****
Step 0: S ← Use LLStack constructor to establish a stack

Step 1: data ← read one integer from inFile

Step 2: newNode ← get a listNode with data // Use listNode constructor

Step 3: push(S, newNode) // put newNode onto the top of the stack

Step 4: repeat step 1 to step 3 until inFile is empty.

Step 5: return S

****
VI. (listNode) buildQueue (inFile)
****
Step 0: Q ←Use LList constructor to establish a queue

Step 1: data ← read one integer from inFile

Step 2: newNode ← get a listNode with data // Use listNode constructor

Step 3: insertQ(Q, newNode) // put newNode in the back of Q

Step 4: repeat step 1 to step 3 until inFile is empty.

Step 5: return Q

****
VII. (listNode) buildList (inFile) // in ascending order
****
Step 0: listHead ← Use LList constructor to establish a LList

Step 1: data ← read one integer from inFile

Step 2: newNode ← get a listNode with data

Step 3: Spot ← findSpot(listHead, newNode)   // see algorithm below
        // locate where to insert newNode

Step 4: insertOneNode (Spot, newNode)

Step 5: repeat step 1 to step 3 until inFile is empty.

****
VIII. (listNode) findSpot(listHead, newNode)
****
Step 1: Spot ← listHead

Step 2: if Spot.net != null && newNode.data > Spot.next.data
                Spot ← Spot.next

Step 3: repeat step 2 until condition failed

Step 4: return Spot