Project 1 (in C++): Linked-list implementation of Stack, Queue, and ordered list.

What your program need to do as follow:

1) Build a stack: a) open an input file; b) read one data at the time from the input file; c) create a new node with data; d) push (newNode), on top of the stack.
2) Build a queue: a) pop the stack; b) print the data in the node to outFile1, c) insertQ (node), at the back of the queue.
3) Build a list: a) delete a node from the front of the queue; b) print the data in the node to outFile2, c) insertLL (node) to the list, in ascending order.
4) Print list: a) print the entire list from the beginning to the end of the list to outFile3.

Note: You must use argv to open input file and 3 output files. -5 points deduction if you hard-code your file names!
****

Language: C++
****

Project points: 10 pts

Due Date: Soft copy (*.zip) and hard copies (*.pdf):

+1 (11/10 pts): early submission, 2/10/2022, Monday before midnight
-0 (10/10 pts): on time, 2/14/2022 Monday before midnight
-1 (9/10 pts): 1 day late, 2/15/2022 Tuesday before midnight
-2 (8/10 pts): 2 days late, 2/16/2022 Wednesday before midnight
(-10/10 pts): non submission, 2/16/2022 Wednesday 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 Black Board.

*** 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. Inputs:

****

1) inFile (use argv[1]): 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 argv[2]): for stack outputs.
2) outFile2 (use argv[3]): for queue outputs.
3) outFile3 (use argv[4]): for list output.

****

III. Data structure:

****

- listNode class
  - (int) data
  - (listNode *)next

  Methods:
  - constructor (data) //create a listNode node for data and make sure assign node's next ← null
  - printNode (node) // print in the format as below:
    (node's data, node's next's data) →
- LLStack class
  - (listNode*) top

  Methods:
  - constructor (...) // create a new LLStack 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 (newNode) // insert newNode after dummy node
  - (bool) isEmpty () // if top's next is null returns true, otherwise returns false.
  - listNode* pop() // if stack is not empty, deletes and returns the top of the stack, i.e., the node after dummy. // otherwise, print "stack is empty" to outfile 1.

- buildStack (inFile) // build a stack from the data in inFile. See algorithm steps 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 new LLQueue 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, i.e., after the node points by tail, i.e.,
      // newNode's next ←Q.tail's next
      // Q.tail's next ← newNode
      // Q.tail ← newNode
    - (listNode *) deleteQ (…) // if Q is not empty, delete and return the node after the dummy (i.e., Q.head's next).
    - (bool) isEmpty (…)// Returns true if tail points to the dummy node, returns false otherwise.
    - buildQueue (…) // build a queue from nodes in the stack. See algorithm steps below.
- LList class
    - (listNode *) listHead

  Methods:
    - constructor (...)// create new LList with a dummy node, set the data in the dummy node to -99999 and next to null; and let listHead points to the dummy node.
    - (listNode *) findSpot (listHead, newNode) // the method finds the location, called Spot, in the list to insert newNode; it returns Spot.
    - insertOneNode (spot, newNode) // inserts newNode between spot and spot's next, i.e.,
      // newNode's next ← spot's next
      // spot's next ← newNode
    - buildList (…) // build a linked list from nodes in the queue. See algorithm steps below.
    - printList (...) // the method calls printNode(...) to print all nodes in the list to outFile3, in the following format:

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

  For example:
  listHead → (-99999, 4, 8) → (8, 11) → (11, 21) →..........→ (54, 87) → (87, NULL) → NULL

****
IV. main ()
****

```
Step 0: if argc != 4
            print to console : "need one inFile and three outfiles"
        else
            inFile ← open input  file from argv[1]
            outFile1, outFile2, outFile3 ← open from argv[2], argv[3], argv[4]

Step 1:  S ← creates a LLStack using constructor.
Step 2: buildStack(S, inFile)
Step 3:  Q ← creates a LLQueue using constructor.
Step 4: buildQueue(S, Q, outFile1)
Step 5:  listHead ← creates a LList using constructor
Step 6: buildList (Q, listHead, outFile2)
Step 7: printList(listHead, outFile3)
Step 8: close all files
```

****
V. buildStack (S, inFile)
****

```
Step 1: data ← read one integer from inFile
Step 2: newNode ← creates a listNode for data using constructor
Step 3: push(S, newNode)
Step 4: repeat step 1 to step 3 until inFile is empty.
```

****
VI. buildQueue (S, Q, outFile1)
****

```
Step 1: newNode ← pop (S)
Step 2: outFile1 ← output newNode's data to outFile1
Step 3: insertQ (Q, newNode)
Step 4: repeat step 1 to step 3 until S is empty.
```

****
VII. buildList (Q, listHead, outFile2)
****

```
Step 1: newNode ← deleteQ (Q, outFile2)
Step 2: outFile2 ← output newNode's data to outFile2
Step 3: Spot ← findSpot(listHead, newNode)   // see algorithm below
Step 4: insertOneNode (Spot, newNode)
Step 5: repeat step 1 to step 4 until Q is empty.
```

****
VIII. (listNode*) findSpot(listHead, newNode)
****

```
Step 1: Spot ← listHead
Step 2: if Spot's next != null && Spot's next's data <  newNode's data
                    Spot ← Spot's next
Step 3: repeat step 2 until condition failed
Step 4: return Spot
```

****
IV. (listNode*) deleteQ (Q, outFile2)
****

```
Step 0: if isEmpty (Q) // Q.tail == Q.head
            outFile2 ← output "Q is empty" to outFile2
            return null

Step 1:  (listNode*) temp
Step 2:  temp ← Q.head's next
Step 3: Q.head's next ←temp's next
Step 4: temp's next ← null
Step 5: return temp
```