

CS1234 Small-Scale Application Development
January 2026 MidSem

This MidSem has four questions. Q1, Q2, and Q4 expect you to use awk and carry 5 marks total, while Q3 is to be written in C and carries 15 marks.

Q1: [2 marks]

Write an AWK script that:

- Reads from a given file
- Script run as:
 `awk -f Q1.awk <File>`
 <File> is an input file in the testcases folder.
- Each line of the file contains: <ID> <name> <balance>

For every line, create a file in the current directory named: <ID>.txn

Each generated file must contain: <name> <balance>

[One output file per input line]

Note: In awk, we can use output redirection to create a file. For instance,

```
awk '{ print "abcd" > $1.txt;}' data.txt
```

creates a file for each row with the filename derived from the first column (of data.txt), and writes "abcd" in each such file.

Q2: [1 mark]

Write an AWK script that reads every .txn file in the working directory and prints the sum total of all the balances.

Script run as:

```
awk -f Q2.awk *.txn
```

*.txn are the files present in the input folder in testcases.

Q3: [15 marks]

You are required to implement a transaction management system in C that simulates deferred account updates using a doubly linked list (starter code is provided in Q3.c).

Each bank account is represented by a file named using its ID. (Use ID given in input and use the .txn to access balance details. This file has to be updated constantly during operations. We use the term "execution" to update the balance of that account by adding the corresponding amount. "Reversing" is used when we need to undo an executed transaction. In that case we subtract the amount reflected in the transaction and update the balance.)

The system maintains a transaction list using a doubly linked list with sentinel head and tail nodes and a cursor that tracks the last transaction executed in the list.

Initially, the list is empty and the cursor is at the head. [This empty list looks like: head <-> tail. All nodes need to be inserted between the head and tail.]

Input:

The first line contains an integer Q — the number of queries.

Each of the next Q lines contains one of the following commands:

1. Insert Transaction: I <pos> <ID> <amount>

Insert a transaction at <pos> positions relative to the cursor.

- pos > 0 : move forward from cursor [The transaction is inserted into the list but not executed]
- pos < 0 : move backward from cursor [this transaction is inserted and should be executed]
- pos is never 0
- If the position is invalid, ignore.
- <amount> may be positive (credit) or negative (debit)
[Note: pos = 0 points to the cursor itself, pos = 1 points to the node to be inserted one position just after the cursor, and pos = -1 points to the node to be inserted one position just before the cursor]
- CURSOR POSITION REMAINS UNCHANGED

Examples:

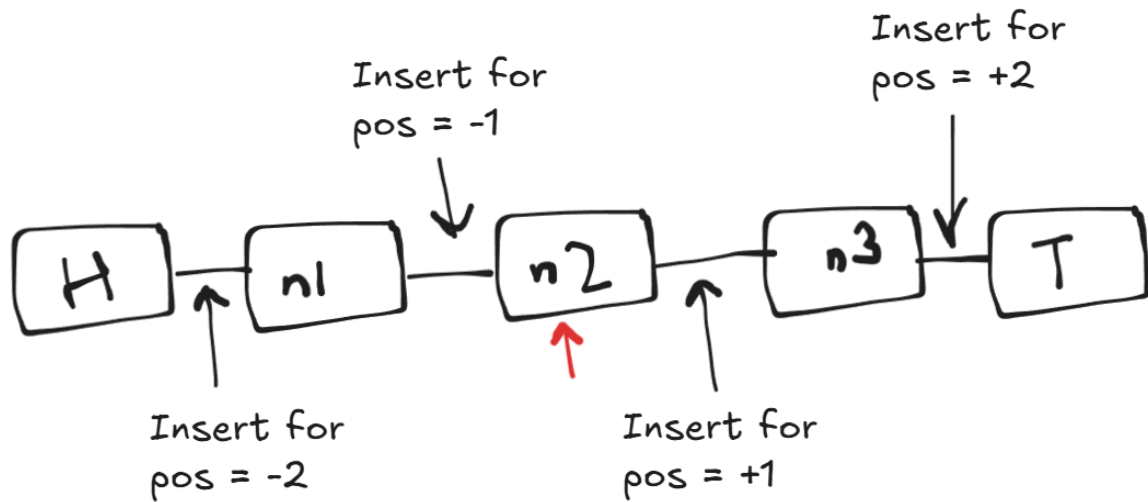
1) List is head <-> tail. Cursor is at head.

- if pos = 1: we need to insert a node (say n1) one node right from the head. List will look like: head<->n1<->tail. n1 should not have been executed.

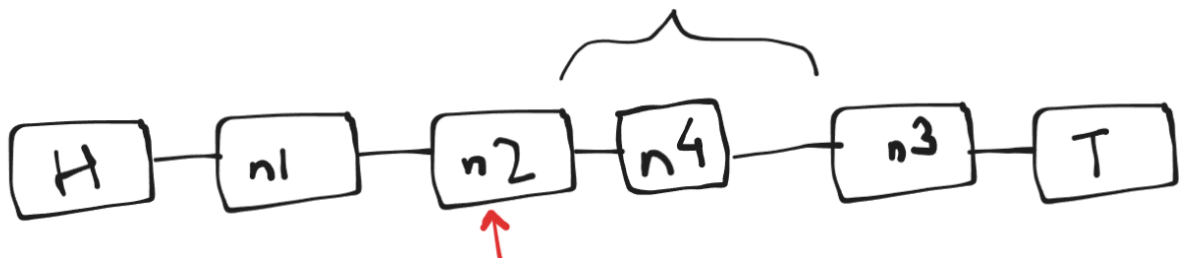
- if pos = -1: we need to insert a node (n1) one node left from the head. However this would be invalid, since the node should be inside the list.

- if pos = 2: we need to insert n1 2 nodes away from the head. Looking at the list, it would give us: head<->tail<->n1. But again, this is invalid. In fact any pos > 1 is invalid in this case.

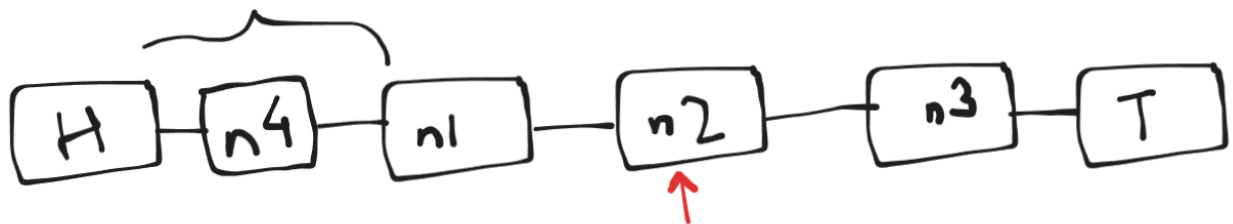
2) List is head $\leftrightarrow$ n1 $\leftrightarrow$ n2 $\leftrightarrow$ n3 $\leftrightarrow$ tail. Cursor is at n2. New node: n4 to be inserted.



- if pos = 1: List becomes head $\leftrightarrow$ n1 $\leftrightarrow$ n2 $\leftrightarrow$ n4 $\leftrightarrow$ n3 $\leftrightarrow$ tail. Transaction n4 is not executed.



- if pos = -2: List becomes head $\leftrightarrow$ n4 $\leftrightarrow$ n1 $\leftrightarrow$ n2 $\leftrightarrow$ n3 $\leftrightarrow$ tail. Transaction n4 HAS TO BE EXECUTED.



- if pos < -2 or pos > 1 then it is Invalid.

2. Delete Transaction: D <pos>

Delete the transaction located <pos> positions away from the cursor. If $\text{pos} \leq 0$, then the transaction should be reversed. When a transaction is deleted, the corresponding node in the list is also removed.

If the position is invalid, ignore the command.

CURSOR POSITION REMAINS UNCHANGED, unless the node you're deleting is the cursor itself (in that case, shift the cursor one node back)

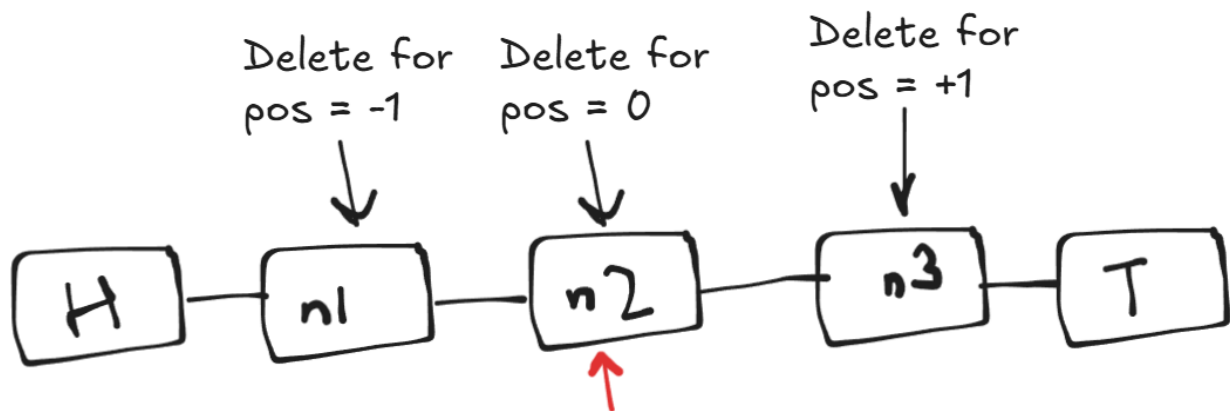
Examples:

1) List is $\text{head} \leftrightarrow n1 \leftrightarrow \text{tail}$. Cursor is at head.

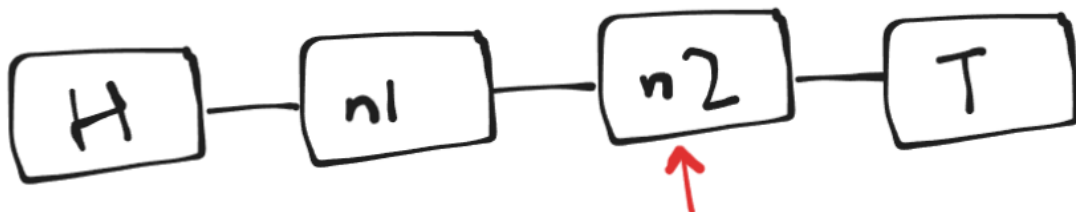
- $\text{pos} = 0$: Invalid.

- $\text{pos} = 1$: We need to delete $n1$. Since pos is positive, there's no need to reverse it (because it hasn't been executed!)

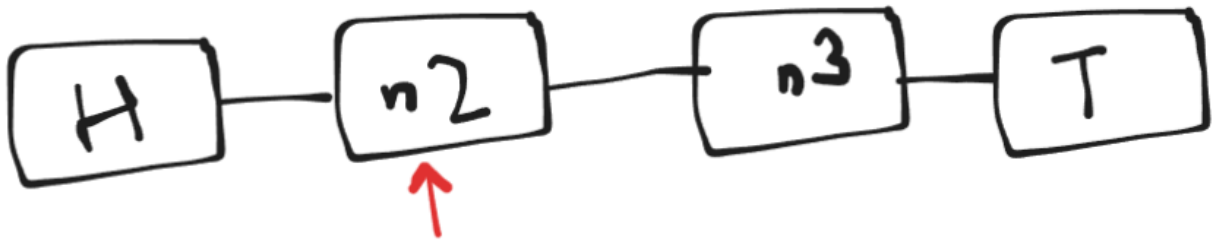
2) List is $\text{head} \leftrightarrow n1 \leftrightarrow n2 \leftrightarrow n3 \leftrightarrow \text{tail}$. Cursor is at $n2$.



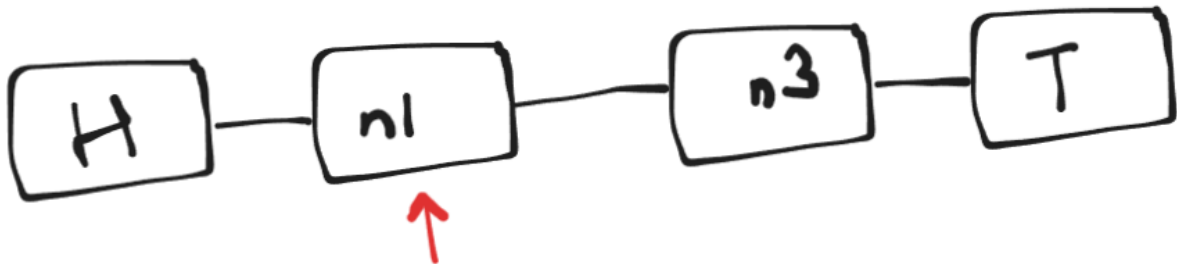
- $\text{pos} = 1$: We need to delete $n3$. [No need for transaction reversal]



- pos = -1: Reverse transaction corresponding to n1. Then delete it.



- pos = 0: Reverse transaction corresponding to n2. Then delete it. Since cursor is at n2, we need to shift it one node back, to n1



- pos < -1 or pos > 1: Invalid.

3. Undo: U

Undo the last executed transaction by reversing the transaction (indicated by the node) for the corresponding account file and then moving the cursor one position backward. If the cursor is at head, ignore.

4. Execute: E

Execute the next pending transaction by moving the cursor one position forward and executing the transaction (indicated by the node) for the corresponding account file. If the cursor is at the node previous to the tail sentinel node, then ignore.

5. Commit: C

Execute all pending transactions to the right of the cursor. [Note that the cursor would finally be at the node previous to the tail sentinel node]

6. Get Balance: G <ID>

Print the current balance of account <ID>.

Output

For each G command, print the balance of the specified account on a new line.

All transactions that have been completed should correctly reflect the balance in their respective .txn files; which have to be accessed by their ID as discussed before ("`<ID>.txn`").

Points to note:

- IDs are unique.
- Doubly Linked List should be used.
- Transactions operate directly on files.
- .txn files are assumed to be already present in the current directory.
- Invalid operations (e.g., out-of-bound cursor movement or deletion or insertion) should be ignored. [Example: Trying to Undo at the start of the transaction list].
- The transaction list uses sentinel head and tail nodes.
- All balances are integers. Initially they're all positive.
- Ensure memory is freed. 0.5 Marks will be deducted if memory is not freed.
- Save the file as "Q3.c" (starter code is given)

Sample Input:

```
10
I 1 ID001 -767
I 2 ID002 -14
E
G ID002
G ID001
E
U
G ID002
C
D -1
```

Expected Output:

```
2014
6000
2014
```

[Assuming we have two transaction files originally:

ID001.txn - John 6767

ID002.txn - Wick 2014

Final txn file contents should be:

ID001.txn - John 6767

ID002.txn - Wick 2000]

Constraints:

Q: 1 to 5000, including both

ID: Of the form - "IDXYZ" Where XYZ is a 3 digit number from 000 to 999

Account Name length: 1 to 50

amount: -1e4 to 1e4

Initial Balance: -1e7 to 1e7

Test cases Distribution:

If implemented : # of tcs pass (public+private)

- Only G : 1/30
- I & E & G : 7/30
- I & E & G & C : 9/30
- I & E & G & U : 9/30
- I & E & G & D : 10/30
- I & E & G & C & U : 15/30
- I & E & G & C & D : 13/30
- I & E & G & U & D : 12/30
- Everything : 30/30

Q4: [2 marks]

Write an AWK script to find all the defaulters, i.e, print the names of all users who have negative balances. Print "(none)" in case of no defaulters. Save the file as "Q4.awk"

Script run as:

```
awk -f Q4.awk *.txn
```

*.txn are the files present in the input folder in testcases.