INDIAN STATISTICAL INSTITUTE

MTech(CS) I year 2025-2026

Subject: Computing Laboratory

Lab Test 1 (September 15, 2025)

Total: 50 marks + 10 bonus marks Duration: 3 hours

GENERAL INSTRUCTIONS

1. All programs should take the required input via the standard input (terminal/keyboard), and print the desired output to the terminal.
2. Please make sure that your programs adhere strictly to the specified input and output format. Do not print extra strings asking the user for input, debugging messages, etc. These will cause the automatic checking system to fail.
3. Please make sure that the programs are free from memory errors and leaks, you will lose marks if they are not.

Q1. **(10 marks)**

Create a data structure TwoStacks that stores two stacks (containing integers) in a single array A of size N. The two stacks are stored starting from the two ends of A, and grow towards each other. TwoStacks must support the following operations.

- push1(x) pushes x (an integer) onto the first stack;
- push2(x) pushes x (an integer) onto the second stack;
- pop1() pops the top element from the first stack and prints it;
- pop2() pops the top element from the second stack and prints it;
- print() prints the contents of the first stack, followed by second stack, in order from top to bottom.

If push1(x) or push2(x) is called when the array A already contains N elements, only a `Stack is full` message should be printed. Similarly, if pop1() or pop2() is called when the corresponding stack is empty, only a `Stack is empty` message should be printed.

Input format: A positive integer N, followed by a list of operations (using the syntax given above), one operation per line. Note that the number of operations will **NOT** be given to you in advance.

Output format: The output (including any error messages) produced by the given sequence of push / pop / print operations. Any output produced by an operation should start from a new line.

Q2. **(15 marks)**

Design a data structure SpecialStack for storing integers. SpecialStack should support the standard stack operations push(), pop() and print(), as well as an additional operation getMin(), which returns the value of the smallest integer stored in the data structure (but does not make any other changes). For full credit, your implementation of getMin() must have a time complexity of $O(1)$.

Your stack should never overflow. If pop() or getMin() is called on an empty stack, a `Stack is empty` message should be printed; otherwise, the value returned should be printed.

hint: You can implement SpecialStack using **two** stacks. One will be used for storing the elements in the usual way. Use the second stack to keep track of the current minimum value.

Input format: A positive integer M, followed by a list of M push / pop / getMin operations, with one operation per line.

Output format: Any output produced by the getMin, pop and print operations. See the provided sample output for details.

Q3. **(20 marks)**

A *stack permutation* refers to a rearrangement of elements from an input queue Q_1 to an initially empty output queue Q_2 using a stack S, using only the following operations.

- dequeue on Q_1
- enqueue on Q_2, and
- push and pop on S.

Note that you are not permitted to call enqueue on Q_1 or dequeue on Q_2. You are also not permitted to use any other storage structure. Thus, any element dequeued from Q_1 must *immediately* be pushed onto S, or enqueued into Q_2; similarly any element popped from S must immediately be enqueued into Q_2.

Write a program that determines, given the contents of Q_1, whether a specified rearrangement of these elements can be obtained as a stack permutation as described above. If this is possible, your program should print the shortest sequence of steps by which the given rearrangement can be obtained in Q_2; otherwise it should print `Not possible`.

You may assume that the elements of Q_1 are all distinct.

Input format: A number n specifying the number of elements (all distinct integers) in Q_1, followed by the elements themselves in the order in which they are stored in Q_1 (from the head to the tail), followed by a rearrangement of these n integers.

Output format: Either the `Not possible` message, or a sequence of dequeue, enqueue, push and pop operations that results in the given rearrangement being stored in Q_2.

Sample input 1:

```
3
1 2 3
2 1 3
```

Sample output 1:

`dequeue`	$\longleftarrow$	remove 1 from Q_1
`push`	$\longleftarrow$	push 1 onto S
`dequeue`	$\longleftarrow$	remove 2 from Q_1
`enqueue`	$\longleftarrow$	add 2 to Q_2
`pop`	$\longleftarrow$	pop 1 from S
`enqueue`	$\longleftarrow$	add 1 to Q_2
`dequeue`	$\longleftarrow$	remove 3 from Q_1
`enqueue`	$\longleftarrow$	add 3 to Q_2

Sample input 2:

```
3
1 2 3
3 1 2
```

Sample output 2: `Not possible`

Q4. **(5+5+5 = 15 marks)**

(a) Recall that *height* is defined as the length of the longest root-to-leaf path in a tree. Write a function to compute the height of a given binary tree.

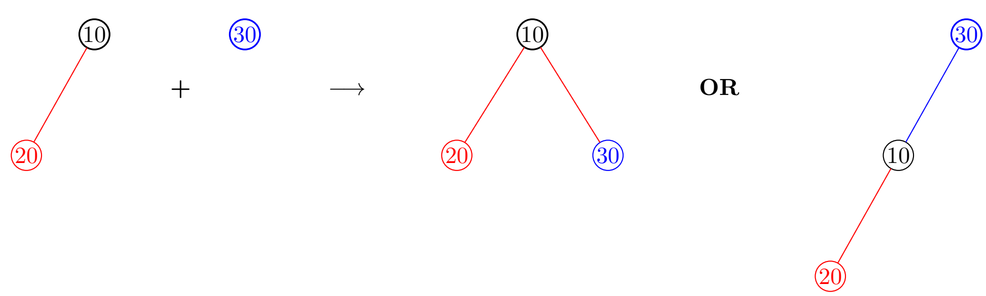
(b) Write a function to compute the depth of the node that has less than 2 children **and** is closest to the root of a given binary tree. The depth of the root is taken to be 0.

(c) Suppose you are given two binary trees, T_1 and T_2. Your task is to join T_1 and T_2 into a single binary tree that has the minimum possible height. In order to join the two trees, the root of one tree must be made a child of some node of the other tree that has less than two children. Write a program to accomplish this task.

If there are multiple ways of joining the trees such that all of them result in the same final height, you program should prefer

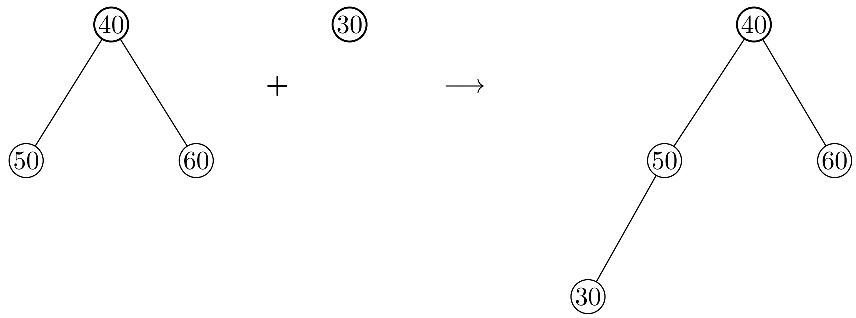
- making T_2 a sub-tree of T_1, rather than the other way around;
- attaching the root of one tree at the leftmost among all possible positions.

Example 1: Given the two trees in the left part of the figure below, two ways in which they can be joined are shown on the right side.



The tree in the middle corresponds to the correct way of joining, since the resultant tree has minimum height.

Example 2:



If 40 is made a left or right child of 30, we will also get a binary tree of height 2. Similarly, if 30 is attached as any child of 50 or 60, the resulting tree is of height 2. We prefer merging T_2 into T_1; we also prefer the leftmost among all attachment points that lead to the same height. Thus, the correct output is the tree shown above.

Input format: The first line of input will contain n_1, the number of nodes in T_1. This will be followed by n_1 lines, in the standard format discussed in class, i.e., each of these lines will correspond to one node in the tree, and will consist of 3 integers corresponding to the value in the node, and the indices of its left and right child nodes resp. The first of these n_1 lines is the root node, and its index is taken to be 0. The data for T_1 will be followed by the data for T_2 in the same format.

Output Format The output should correspond to the joined binary tree, using the format described above. It should thus start with the number $n_1 + n_2$, followed by $n_1 + n_2$ lines, one for each node. The first of these $n_1 + n_2$ lines should correspond to the root of the joined tree.

Sample input 1:

```
2
10 1 -1
20 -1 -1
1
30 -1 -1
```

Sample output 1:

```
3
10 1 2
20 -1 -1
30 -1 -1
```

Sample input 2:

```
3
40 1 2
50 -1 -1
60 -1 -1
1
30 -1 -1
```

Sample output 2:

```
4
40 1 2
50 3 -1
60 -1 -1
30 -1 -1
```