DATA STRUCTURE

Chapter 3 — STACKS

IMPORTANT PROBLEMS & SOLUTIONS

Set 2 (Problems 12–21)

Additional Exam-Type Problems — Conversions, Two-Stack Evaluation, Sorting, Recursion and More

(Diploma / B.Tech — Computer Science & Engineering)

About This Set

This is the second set of solved problems on Stacks, continuing from Set 1. It covers the remaining conversion types, the two-stack method of direct expression evaluation, and some higher-order stack problems (sorting, recursive reversal, and a well-known application) that are frequently asked as long-answer or practical questions in examinations. Problem numbering continues from Set 1 for easy reference. This set focuses on:

- Postfix to Infix and Prefix to Postfix / Postfix to Prefix conversions
- Direct evaluation of an infix expression using two stacks (operand stack + operator stack)
- Detecting duplicate (redundant) parentheses using a stack
- Implementing two stacks within a single array
- Sorting a stack using only one auxiliary stack
- Reversing a stack using recursion (no auxiliary stack)
- Application problem — Next Greater Element using a stack

Problem 12

Statement: Convert the postfix expression A B + C D - * E / to its infix form, using a stack.

Solution:

Rule: scan left to right; push every operand. On encountering an operator, pop the top two elements — the second-popped is treated as the left operand (op1) and the first-popped as the right operand (op2) — form the string "(op1 operator op2)", and push it back onto the stack as a single unit.

Token	Action	Stack (bottom → top)
A	Push	A
B	Push	A, B
+	Pop B, A → form (A+B) → Push	(A+B)
C	Push	(A+B), C
D	Push	(A+B), C, D
-	Pop D, C → form (C-D) → Push	(A+B), (C-D)
*	Pop (C-D), (A+B) → form ((A+B)*(C-D)) → Push	((A+B)*(C-D))
E	Push	((A+B)*(C-D)), E
/	Pop E, ((A+B)*(C-D)) → form (((A+B)*(C-D))/E) → Push	(((A+B)*(C-D))/E)

Answer: (((A+B)*(C-D))/E)

Problem 13

Statement: Convert the prefix expression * + A B - C D to postfix notation, using a stack.

Solution:

Rule: scan the expression from right to left; push every operand. On encountering an operator, pop the first element as op1 and the second as op2, form the string "op1 op2 operator" (postfix order), and push it back as a single unit.

Token (R→L)	Action	Stack (bottom → top)

D	Push	D
C	Push	D, C
-	Pop op1=C, op2=D → form "C D -" → Push	"C D -"
B	Push	"C D -", B
A	Push	"C D -", B, A
+	Pop op1=A, op2=B → form "A B +" → Push	"C D -", "A B +"
*	Pop op1="A B +", op2="C D -" → form "A B + C D - *" → Push	"A B + C D - *"

Check: the prefix expression represents (A+B) * (C-D), and its postfix form A B + C D - * evaluates in exactly the same way.

Answer: A B + C D - *

Problem 14

Statement: Convert the postfix expression A B C + * D - to prefix notation, using a stack.

Solution:

Rule: scan left to right; push every operand. On encountering an operator, pop the top element as op2 and the next as op1, form the string "operator op1 op2" (prefix order), and push it back as a single unit.

Token	Action	Stack (bottom → top)
A	Push	A
B	Push	A, B
C	Push	A, B, C
+	Pop op2=C, op1=B → form "+ B C" → Push	A, "+ B C"
*	Pop op2="+ B C", op1=A → form "* A + B C" → Push	"* A + B C"
D	Push	"* A + B C", D
-	Pop op2=D, op1="* A + B C" → form "- * A + B C D" → Push	"- * A + B C D"

Check: the postfix expression evaluates as (A * (B + C)) - D, and the prefix form - * A + B C D represents exactly the same expression.

Answer: - * A + B C D

Problem 15

Statement: Evaluate the infix expression 3 + 4 * 2 / (1 - 5) directly, without first converting it to postfix, using an operand stack (OPRD) and an operator stack (OPTR).

Solution:

Rule: push every operand onto OPRD. Push every operator onto OPTR, but before pushing, repeatedly pop and evaluate whenever the OPTR top has precedence greater than or equal to the current operator (and is not '('). On ')', evaluate back to the matching '('. At the end, evaluate all remaining operators.

Token	Action	OPRD (bottom → top)	OPTR (bottom → top)
3	Push to OPRD	3	
+	OPTR empty → Push	3	+

4	Push to OPRD	3, 4	+
*	Top '+' has lower prec. → Push	3, 4	+, *
2	Push to OPRD	3, 4, 2	+, *
/	Pop * (equal prec.); 4*2=8; Push 8; then Push /	3, 8	+, /
(	Push to OPTR	3, 8	+, /, (
1	Push to OPRD	3, 8, 1	+, /, (
-	Top '(' → Push	3, 8, 1	+, /, (, -
5	Push to OPRD	3, 8, 1, 5	+, /, (, -
)	Pop -; 1-5=-4; Push -4; pop '('	3, 8, -4	+, /
End	Pop /; 8÷(-4)=-2; Push -2	3, -2	+
End	Pop +; 3+(-2)=1; Push 1	1	(empty)

Answer: 1

Problem 16

Statement: Using a stack, check whether the expression ((A+B)) contains duplicate (redundant) parentheses.

Solution:

Rule: push every symbol onto the stack except ')'. When ')' is encountered, first look at the top of the stack — if the top is '(' at that moment, it means nothing (no operand or operator) was pushed between that '(' and this ')', so the pair is a duplicate/redundant parenthesis. Otherwise, pop the stack until the matching '(' is found, discard it, and continue.

Symbol Scanned	Action	Stack (bottom → top)
(	Push	(
(	Push	(, (
A	Push	(, (, A
+	Push	(, (, A, +
B	Push	(, (, A, +, B
)	Top = B (not '(') → pop B, +, A, then pop '(' — valid sub-expression, no duplicate here	(
)	Top = '(' at this point → DUPLICATE PARENTHESES DETECTED	(popped — error/flag raised)

Answer: The expression ((A+B)) contains duplicate parentheses — the outer pair adds nothing and is redundant.

Problem 17

Statement: Explain how two stacks can be implemented efficiently using a single array of size N, and illustrate with a short example (N = 6).

Solution:

Instead of splitting the array into two fixed halves (which wastes space if one stack grows faster than the other), both stacks share the same array from opposite ends: Stack-1 grows upward from index 0, and Stack-

2 grows downward from index N-1. This way, the combined free space is used efficiently by whichever stack needs it. TOP1 is initialised to -1 and TOP2 is initialised to N. Overflow occurs only when the two tops meet, i.e. when TOP1 + 1 = TOP2.

```
PUSH1(ITEM):                          PUSH2(ITEM):
  If TOP1 + 1 = TOP2, OVERFLOW          If TOP2 - 1 = TOP1, OVERFLOW
  Else: TOP1 = TOP1 + 1                 Else: TOP2 = TOP2 - 1
        ARR[TOP1] = ITEM                      ARR[TOP2] = ITEM

POP1():                               POP2():
  If TOP1 = -1, UNDERFLOW               If TOP2 = N, UNDERFLOW
  Else: ITEM = ARR[TOP1]                Else: ITEM = ARR[TOP2]
        TOP1 = TOP1 - 1                       TOP2 = TOP2 + 1
        Return ITEM                           Return ITEM
```

Example with N = 6 (indices 0 to 5), TOP1 = -1, TOP2 = 6 initially:

Operation	TOP1	TOP2	Array (index 0 → 5)
Initial	-1	6	_, _, _, _, _, _
PUSH1(10)	0	6	10, _, _, _, _, _
PUSH1(20)	1	6	10, 20, _, _, _, _
PUSH2(90)	1	5	10, 20, _, _, _, 90
PUSH2(80)	1	4	10, 20, _, _, 80, 90
PUSH1(30)	2	4	10, 20, 30, _, 80, 90

After these operations, TOP1 + 1 = 3 and TOP2 = 4, so there is still one free slot (index 3) — overflow would occur only on the next push if both stacks try to use that same slot.

Answer: Stack-1 (bottom→top): 10, 20, 30. Stack-2 (bottom→top): 90, 80. One free slot remains between them.

Problem 18

Statement: Sort a stack S so that the smallest element ends up on top, using only one additional (auxiliary) stack. Trace the algorithm for S (top → bottom) = 3, 1, 4, 2.

Solution:

```
Algorithm SORT_STACK(S)
Step 1: Create an empty auxiliary stack AUX
Step 2: While S is not empty:
           temp = S.pop()
           While AUX is not empty AND AUX.top() > temp:
                S.push(AUX.pop())
           AUX.push(temp)
Step 3: While AUX is not empty:
           S.push(AUX.pop())
Step 4: Now S is sorted with the smallest element on top
```

Trace of Step 2 (building AUX) for S (top → bottom) = 3, 1, 4, 2:

Iteration	temp popped from S	Elements moved back to S	AUX (bottom → top) after step
1	3	none	3
2	1	3 (since 3 > 1)	1

3	3 (popped again from S)	none (1 is not > 3)	1, 3
4	4	none	1, 3, 4
5	2	4, then 3 (both > 2)	1, 2
6	3 (popped again from S)	none	1, 2, 3
7	4 (popped again from S)	none	1, 2, 3, 4

S is now empty and AUX (bottom → top) = 1, 2, 3, 4. Step 3 then transfers every element from AUX back to S: popping AUX gives 4, 3, 2, 1 in that order, and pushing each in turn onto S makes the final S, from top to bottom, equal to 1, 2, 3, 4.

Answer: Sorted stack S (top → bottom): 1, 2, 3, 4 — smallest element (1) on top.

Problem 19

Statement: Write a recursive algorithm to reverse the contents of a stack without using any extra stack or array, and trace it for S (top → bottom) = 3, 2, 1.

Solution:

Two recursive functions are used together: REVERSE(S) removes the top element, recursively reverses what remains, and then inserts the removed element at the bottom using INSERT_AT_BOTTOM(S, item).

```
Algorithm INSERT_AT_BOTTOM(S, ITEM)
Step 1: If S is empty, then Push(ITEM); Return
Step 2: temp = S.pop()
Step 3: INSERT_AT_BOTTOM(S, ITEM)
Step 4: S.push(temp)

Algorithm REVERSE(S)
Step 1: If S is not empty, then
           temp = S.pop()
           REVERSE(S)
           INSERT_AT_BOTTOM(S, temp)
```

Trace for S (top → bottom) = 3, 2, 1:

1. REVERSE pops 3 (S becomes 2, 1) and calls REVERSE on the remaining stack.
2. That call pops 2 (S becomes 1) and calls REVERSE again.
3. That call pops 1 (S becomes empty) and calls REVERSE once more, which does nothing since S is now empty (base case).
4. Unwinding begins: INSERT_AT_BOTTOM(S, 1) is called on the empty stack, so 1 is simply pushed — S = 1.
5. INSERT_AT_BOTTOM(S, 2) is called next: it pops 1, inserts 2 at the (now empty) bottom, then pushes 1 back on top — S (top → bottom) = 1, 2.
6. Finally, INSERT_AT_BOTTOM(S, 3) is called: it pops 1, then pops 2, inserts 3 at the bottom, then pushes 2 back, then pushes 1 back — S (top → bottom) = 1, 2, 3.

Answer: Reversed stack S (top → bottom): 1, 2, 3.

Problem 20

Statement: For the array [4, 5, 2, 10, 8], find the Next Greater Element (NGE) for every element — the first element to its right that is larger than it (or -1 if none exists) — using a stack.

Solution:

Rule: scan the array from right to left. For each element, pop the stack while its top is less than or equal to the current element (those elements can never be the NGE for anything to their left). If the stack becomes empty, the NGE is -1; otherwise, the NGE is the new stack top. Then push the current element onto the stack.

Element (R→L)	Popped (≤ current)	NGE Found	Stack after step (bottom → top)
8	none (stack empty)	-1	8
10	8 (8 ≤ 10)	-1	10
2	none (10 > 2)	10	10, 2
5	2 (2 ≤ 5)	10	10, 5
4	none (5 > 4)	5	10, 5, 4

Reading the NGE values in the original left-to-right order of the array [4, 5, 2, 10, 8]:

Answer: NGE array: [5, 10, 10, -1, -1]

Practice Tip: For conversion problems, always double-check your final answer by mentally evaluating it and comparing with the meaning of the original expression, as shown in the checks above.