DATA STRUCTURE

Chapter 3 — STACKS

IMPORTANT PROBLEMS & SOLUTIONS

Set 3 (Problems 22–30)

Number-Base Conversion, Stack Permutations, Queue-from-Stacks, Min-Stack, Unary Minus, Depth & Memory Problems

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

About This Set

This is the third set of solved problems on Stacks, continuing the numbering from Sets 1 and 2. It moves into slightly more advanced and application-oriented problems that are common in both semester examinations and technical/competitive exams — number-base conversion, validity checking, building one data structure from another, and memory-related numerical problems tied to recursion. This set covers:

- Decimal-to-binary conversion using a stack
- Checking whether a given pop sequence is a valid stack permutation of a push sequence
- Implementing a queue using two stacks
- Designing a stack that supports getMin() in O(1) time
- Handling unary minus while converting infix to postfix
- Right-associative operators (exponentiation) in infix-to-postfix conversion
- Finding the maximum nesting depth of brackets using a stack
- Numerical problems on recursion depth versus available stack memory

Problem 22

Statement: Convert the decimal number 23 into its binary equivalent using a stack.

Solution:

```
Algorithm DEC_TO_BIN(N)
Step 1: Create an empty stack S
Step 2: While N > 0:
           R = N mod 2
           Push(R) onto S
           N = N div 2   (integer division)
Step 3: While S is not empty:
           Print Pop(S)
```

Trace for N = 23:

N (before)	N mod 2 (pushed)	N div 2 (new N)	Stack (bottom → top)
23	1	11	1
11	1	5	1, 1
5	1	2	1, 1, 1
2	0	1	1, 1, 1, 0
1	1	0	1, 1, 1, 0, 1

N is now 0, so the loop stops. Popping the stack one element at a time (top to bottom) gives the binary digits in the correct left-to-right order: 1, 0, 1, 1, 1.

Check: $10111_2 = 16 + 0 + 4 + 2 + 1 = 23$ ✓

Answer: 23 (decimal) = 10111 (binary)

Problem 23

Statement: Elements 1, 2, 3, 4, 5 are pushed onto a stack in that order, with pops allowed to be interleaved with the pushes. Determine whether each of the following output (pop) sequences is achievable: (a) 4, 5, 3, 2, 1 (b) 4, 3, 5, 1, 2

Solution:

Rule: push the input elements one at a time in order. After every push, compare the stack top with the next expected value in the target output sequence; while they match, pop and move to the next expected value. If, after all elements have been pushed, the stack becomes completely empty, the sequence is a valid stack permutation.

(a) Target output: 4, 5, 3, 2, 1

Action	Stack (bottom → top)	Matched / Popped
Push 1	1	top=1, expected=4 → no match
Push 2	1, 2	top=2, expected=4 → no match
Push 3	1, 2, 3	top=3, expected=4 → no match
Push 4	1, 2, 3, 4	top=4, expected=4 → pop! next expected = 5
(after pop)	1, 2, 3	top=3, expected=5 → no match
Push 5	1, 2, 3, 5	top=5, expected=5 → pop! next expected = 3
(after pop)	1, 2, 3	top=3, expected=3 → pop! next expected = 2
(after pop)	1, 2	top=2, expected=2 → pop! next expected = 1
(after pop)	1	top=1, expected=1 → pop! all matched
(after pop)	(empty)	stack empty, all inputs pushed

Answer: (a) VALID — the stack ends empty with the full sequence matched.

(b) Target output: 4, 3, 5, 1, 2

Action	Stack (bottom → top)	Matched / Popped
Push 1, 2, 3, 4	1, 2, 3, 4	top=4, expected=4 → pop! next expected = 3
(after pop)	1, 2, 3	top=3, expected=3 → pop! next expected = 5
(after pop)	1, 2	top=2, expected=5 → no match
Push 5	1, 2, 5	top=5, expected=5 → pop! next expected = 1
(after pop)	1, 2	top=2, expected=1 → no match, and no more elements left to push

Answer: (b) NOT VALID — no more elements remain to push, but the stack top (2) does not match the next expected output (1).

Problem 24

Statement: Explain how a queue (FIFO) can be implemented using two stacks, S1 and S2, and trace the following sequence of operations: Enqueue(1), Enqueue(2), Enqueue(3), Dequeue(), Enqueue(4), Dequeue(), Dequeue().

Solution:

```
ENQUEUE(ITEM):
   Push(ITEM) onto S1

DEQUEUE():
   If S2 is empty:
        While S1 is not empty:
```

```
            Push(Pop(S1)) onto S2
  Return Pop(S2)
```

S1 always receives new items (LIFO order); whenever S2 runs out of items, the entire contents of S1 are transferred onto S2, which reverses their order and makes the oldest item come out on top — this is what gives FIFO behaviour using two LIFO stacks.

Operation	Result	S1 (bottom → top)	S2 (bottom → top)
Enqueue(1)	—	1	
Enqueue(2)	—	1, 2	
Enqueue(3)	—	1, 2, 3	
Dequeue()	S2 empty → transfer all of S1 to S2, then pop → returns 1	(empty)	2, 3 → after pop: 3, 2
Enqueue(4)	—	4	3, 2
Dequeue()	S2 not empty → pop directly → returns 2	4	3
Dequeue()	S2 not empty → pop directly → returns 3	4	(empty)

Answer: Dequeue() calls return 1, then 2, then 3 — exactly the FIFO order in which they were enqueued.

Problem 25

Statement: Design a stack that supports the usual PUSH and POP operations plus a getMin() operation that returns the current minimum element, all in O(1) time. Trace: Push(5), Push(3), Push(7), Push(2), Pop(), getMin().

Solution:

Maintain a second stack, MinS, alongside the main stack S. On every PUSH(x): push x onto S; then push x onto MinS if MinS is empty or x is less than or equal to MinS's current top, otherwise push a repeat of MinS's current top onto MinS (so both stacks always stay the same height). On every POP: pop from both S and MinS together. getMin() simply returns MinS's top — no searching is required, so every operation is O(1).

Operation	S (bottom → top)	MinS (bottom → top)	Note
Push(5)	5	5	MinS empty → push 5
Push(3)	5, 3	5, 3	3 ≤ 5 → push 3
Push(7)	5, 3, 7	5, 3, 3	7 > 3 → repeat top (3)
Push(2)	5, 3, 7, 2	5, 3, 3, 2	2 ≤ 3 → push 2
Pop()	5, 3, 7	5, 3, 3	pop both stacks together; removed 2
getMin()	5, 3, 7	5, 3, 3	return MinS.top() = 3

Answer: Current minimum after the Pop(): 3.

Problem 26

Statement: Convert the infix expression -A + B * (-C - D) to postfix notation, treating the unary minus as a special right-associative operator '@' with the highest precedence.

Solution:

Step 1 — identify unary minus: a '-' is unary if it appears at the very start of the expression or immediately after '(' or another operator; otherwise it is the ordinary binary subtraction. Replacing each unary minus with '@' gives the modified expression:

```
@A + B * (@C - D)
```

Step 2 — convert using the usual algorithm, with precedence @ (highest) > * , / > + , - , and treating @ as applying to the single operand that immediately follows it:

Token	Action	Stack (bottom → top)	Output so far
@	Push	@	
A	Output; then pop @ (its one operand is done)	(empty)	A @
+	Push	+	A @
B	Output	+	A @ B
*	Push (higher prec. than +)	+ *	A @ B
(	Push	+ * (	A @ B
@	Push	+ * (@	A @ B
C	Output; then pop @	+ * (	A @ B C @
-	Push	+ * (-	A @ B C @
D	Output	+ * (-	A @ B C @ D
)	Pop - ; discard '('	+ *	A @ B C @ D -
End	Pop *, then +	(empty)	A @ B C @ D - * +

Check: reading the postfix, A@ means (-A); C@ means (-C); (-C) D - means ((-C) - D); B times that gives B * ((-C) - D); and finally (-A) plus that whole term reproduces the original expression.

Answer: A @ B C @ D - * +

Problem 27

Statement: Convert the infix expression 2 ^ 3 ^ 2 to postfix notation, keeping in mind that the exponent operator (^) is right-associative.

Solution:

Because ^ is right-associative, when the stack top is also ^ (equal precedence), it is NOT popped before pushing the new ^ — a same-precedence operator is only popped when it is left-associative. This is the key difference from +, -, *, / handled in earlier problems.

Token	Action	Stack (bottom → top)	Output so far
2	Output	(empty)	2
^	Push	^	2
3	Output	^	2 3
^	Top is ^ (equal prec., right-assoc.) → do NOT pop → Push	^ ^	2 3
2	Output	^ ^	2 3 2
End	Pop ^, then ^	(empty)	2 3 2 ^ ^

Evaluating the result confirms right-associativity: 3^2 = 9, then 2^9 = 512 — the same as 2^(3^2). (If ^ were treated as left-associative, the postfix would instead evaluate as (2^3)^2 = 64, a different and incorrect result.)

Answer: 2 3 2 ^ ^ (evaluates to 512, matching 2^(3^2))

Problem 28

Statement: Find the maximum depth of nested brackets in the expression: {[(A+B)*(C-D)]+[(E-F)/(G+H)]} using a stack.

Solution:

Rule: push a marker onto the stack for every opening bracket and pop for every closing bracket. The current stack size at any moment is the current nesting depth; keep track of the largest size reached.

Bracket Scanned	Action	Stack Size (depth)	Maximum So Far
{	Push	1	1
[	Push	2	2
(	Push (for A+B)	3	3
)	Pop	2	3
(	Push (for C-D)	3	3
)	Pop	2	3
]	Pop	1	3
[	Push	2	3
(	Push (for E-F)	3	3
)	Pop	2	3
(	Push (for G+H)	3	3
)	Pop	2	3
]	Pop	1	3
}	Pop	0	3

Answer: Maximum nesting depth = 3.

Problem 29

Statement: Check whether the expression (A + (B * C) is balanced, using a stack.

Solution:

Symbol Scanned	Action	Stack (bottom → top)
(	Push	(
A	ignore (operand)	(
+	ignore (operator)	(
(	Push	(, (
B, *, C	ignore (operands/operator)	(, (
)	Pop — matches the inner '('	(
End of expression	stack is NOT empty	(

Since a scan of the whole expression finishes with one '(' still left on the stack, the expression has one unmatched opening bracket — it is missing a closing bracket.

Answer: NOT balanced — one closing bracket ')' is missing at the end.

Problem 30

Statement: A program's call stack is allocated 1 MB (1,048,576 bytes) of memory. Each activation record created for a recursive call occupies 256 bytes. What is the maximum possible depth of recursion before a stack overflow occurs?

Solution:

Each recursive call pushes one activation record of fixed size onto the call stack (Section 3.7). The maximum number of activation records that can be held is the total stack memory divided by the size of one activation record.

Maximum depth = Total stack size ÷ Size of one activation record

Maximum depth = 1,048,576 bytes ÷ 256 bytes/record

Maximum depth = 4096

This means the recursion can safely go 4096 levels deep; the 4097th nested call would attempt to push an activation record into memory that is no longer available, causing a stack overflow.

Answer: Maximum recursion depth = 4096 calls.

Practice Tip: For validity/permutation and balancing problems, always state clearly at which exact step the mismatch or the leftover stack occurs — examiners award marks for correctly locating the failure point, not just the final Yes/No answer.